Inhaltsverzeichnis

1 Vererbung

Der vererbende Vorfahr wird auch *Superklasse* oder *Oberklasse* genannt, der erbende Nachfahre heißt *Subklasse* oder *Unterklasse*. Vererbt werden Methoden (engl. methods) und Felder (engl. fields), also alle Glieder (engl. members) einer Klasse[1]. Ob die Unterklasse jedoch Zugriff auf das Erbe hat, ist eine andere Frage. Das wird geregelt durch die *Modifier* `public`, `protected`, `private`.

Der Sinn der Vererbung ist, dass die Nachfahren von vorne herein alles haben und können, was ihre Vorfahren haben und können. Die Nachfahren dürfen aber alles nach eigenen Bedürfnissen ändern und anpassen. Sehr wichtig ist dabei die Unterscheidung zwischen statischen und nicht statischen Bestandteilen. Die Feinheiten werden in den folgenden Abschnitten besprochen.

1.1 Begriffe

Alles was im Speicher des Computers abgelegt wird, wird in Java mit einem Namen angesprochen. Ungenaue Sprache spricht in allen Fällen von Variablen, aber es vereinfacht die Sache, wenn man ein bisschen strenger unterscheidet. Im folgenden Beispiel werden die Begriffe erläutert, ohne dass es nötig wäre, die Funktionsweise des Programms vollständig zu verstehen.

```
public class Person {
  String name;
  String vorname;
  boolean mann;
  Person gatte;

  public Person(String v, String n, boolean m){
    vorname=v;
    name=n;
    mann=m;
  }

  public boolean heirate(Person p){
    if(gatte!=null || p.gatte!=null) return false;
    gatte=p;
    p.gatte=this;
    return true;
```

[1] Vererbt werden auch noch sog. member-classes, aber die werden wir nicht behandeln.

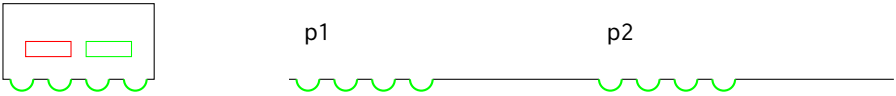
```
  }

  public String toString(){
    return vorname+" "+name;
  }

  public void ausgabe(){
    System.out.println(toString());
  }

  public static void gruppenausgabe(Person[] ps){
    for(int i=0; i<ps.length; ++i)
      System.out.print(ps[i].toString());
  }
}
```

Genau genommen ist hier `i` die einzige Variable. `v`, `n`, `m`, `p` und `ps` nennt man Argumente oder Parameter. Die restlichen `name`, `vorname`, `mann` und `gatte` nennt man Felder oder Attribute.

`Person(...)` ist ein Konstruktor, während die ganz ähnlich aussehenden Programmteile `heirate(...)`, `toString()`, `ausgabe()` und `gruppenausgabe(...)` Methoden genannt werden.

1.2 Felder in Klassen und Instanzen

Die Klasse `Person` Zwei Instanzen von `Person` im Speicher



Der linke Teil des Bildes veranschaulicht die Klasse `Person` mit den vier nicht statischen, grün gefärbten Feldern `vorname`, `name`, `mann` und `gatte`. Die Klasse kann man sich als Stempel veranschaulichen, mit dem man im Speicher Abdrücke herstellen kann. Das Schreiben des Programms entspricht also dem Schnitzen des Stempels.

Jeder Abdruck ist mit seinen Attributen in der Lage, die Daten einer Person zu speichern. Deshalb nennt man einen Abdruck auch Instanz der Klasse Person oder noch kürzer *eine Person*. (Die Bedeutung der farbigen Rechtecke wird weiter unten erklärt.) Im rechten Teil des Bildes sieht man zwei `Personen` mit den beliebig gewählten Namen `p1` und `p2`. Dass die vier Attribute eines Abdrucks gleich groß dargestellt werden, obwohl sie so unterschiedliche Dinge wie Strings oder booleans speichern, ist kein Fehler: Strings brauchen zwar viel mehr Platz, aber an dieser Stelle wird ja nur ein Zeiger auf einen String gespeichert und der braucht nicht nennenswert mehr Platz als ein boolean.

Die Instanzen werden meist in einem ganz anderen Programm gemacht: dort, wo Personen gebraucht werden. Sie entstehen beim Aufruf des new-Befehls. Um sie unterscheiden zu können, verwendet man verschiedene Variablennamen. (Diese sind aber natürlich nicht Teil der Abdrücke. Deshalb schweben die Namen im rechten Teil des Bildes so unverbindlich über den Instanzen.) Die new-Anweisung sucht nach freiem Speicherplatz, macht dort die Instanz und liefert einen Zeiger darauf zurück. Dieser wird dann mit dem gewählten Variablennamen angesprochen.

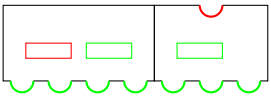
```
Person p1=new Person("Hans", "Schneider", true);
Person p2=new Person("Gertrud", "Schmidt", false);
Person p=p1;    //zweiter Zeiger auf dieselbe Person p1
p.heirate(p2);
p1.vorname="Johann";
System.out.println(p2.gatte.vorname); //ergibt Johann
```

1.3 Vererbung von Feldern

Nun wollen wir die Unterklasse Mensch von Person erzeugen. Instanzen davon sind Personen, die zwei Eltern haben und eine Menge von Kindern. Die Eltern sind selber wieder Menschen, die einzeln abgespeichert werden, die Kinder werden als Mensch[] gespeichert. Es kommen also drei Attribute hinzu, die jeder einzelne Mensch für sich haben muss. Der Stempel muss vergrößert werden.

Weil wir im Moment noch keine bessere Technik kennen, als die Kinder in einem Array abzulegen, begrenzen wir uns auf eine Höchstzahl von Kindern, z. B. 10. Da diese Zahl an verschiedenen Stellen im Programm gebraucht wird, soll sie zentral abgelegt werden. Dafür gibt es in Java Klassenvariable, die durch den Zusatz static deklariert werden. Klassenvariable kann man sich als Speicher im Stempel selber vorstellen. Sie existieren nur einmal (im Bild rot dargestellt) und machen bei new keinen eigenen Abdruck.

Die Klasse Mensch



Eine Instanz von Mensch im Speicher

Man sieht, dass der neue Stempel links den alten als Teil beinhaltet. Den neuen erzeugt man so:

```
public class Mensch extends Person {
  static final int MAX_KINDER=10;
  Mensch vater;
  Mensch mutter;
  Mensch[] kinder;
```

```
  public Mensch(String v, boolean bub, Mensch papa, Mensch mama){
    super(v, papa.name, bub);
    vater=papa;
    mutter=mama;
    kinder=new Mensch[MAX_KINDER];
    papa.kinder[papa.kinderanzahl()]=this;
    mama.kinder[mama.kinderanzahl()]=this;
  }

  public Mensch(String v, String n, boolean bub){
    super(v, n, bub);
    kinder=new Mensch[MAX_KINDER];
  }

  public int kinderanzahl(){
    int i=0;
    while(kinder[i]!=null) ++i;
    return i;
  }

  public String toString(){
    String s=vorname+" "+name;
    if(mutter!=null) s+=", Mutter: "+mutter.vorname;
    return s;
  }
}
```

Die Erzeugung und Verwendung von Instanzen sieht so aus:

```
  Mensch a=new Mensch("Adam", "Lehm", true);
  Mensch e=new Mensch("Eva", "Rippe", false);
  Mensch ka=new Mensch("Kain", true, a, e);
  Mensch ab=new Mensch("Abel", false, a, e);
  e.heirate(a);  //legalize it
  System.out.println(Mensch.MAX_KINDER);
  ab.mann=true;  //Geschlechtsumwandlung
  System.out.println(e.kinder[0].vorname);
```

Man sieht, dass ein `Mensch` auch eine `Person` ist, sonst könnte `e` nicht `a` heiraten. Die Umkehrung gilt nicht! Wäre `a` nur eine `Person`, so könnte er nicht als Vater für `ka` und `ab` dienen. Deren Konstruktor verlangt einen `Mensch`, denn nur bei dem kann man Kinder eintragen.

In dieser Klasse ist erstmals auch ein zweiter Konstruktor vonnöten, weil sonst die Erzeugung der ersten Menschen nicht möglich wäre, wenn immer Vorfahren angegeben werden müssten.

1.4 Statische Methoden

Kommen wir nun zu den kleinen farbigen Rechtecken in den Stempeln. Sie stehen für statische (rote) und nicht statische (grüne) Methoden einer Klasse.

Statische Methoden sind die einfacheren. Sie existieren für sich und ohne Bezug auf eine Instanz von irgendwas. Daten, die sie verarbeiten sollen, müssen entweder als Argument übergeben werden oder in statischen Variablen gespeichert sein. Die bekannteste statische Methode ist die `main`-Methode.

Wegen ihrer Unabhängigkeit kann eine statische Methode an vielen Stellen stehen. Nur wenn sie auf eine nicht öffentliche statische Variable einer Klasse zugreifen muss, ist man gezwungen, sie auch innerhalb dieser Klasse zu formulieren. Ansonsten schreibt man sie dort nieder, wo man sie dem Namen nach am ehesten vermuten würde. Die Methode `gruppenausgabe(Person[])` steht in der Klasse `Person`, weil sie Personen ausgibt. Das ginge aber an jeder anderen Stelle auch. Die `main`-Methode bekommt oft nur aus Faulheit keine eigene Datei.

1.5 Instanz-Methoden

Nicht-statische Methoden nennt man Instanzmethoden, weil sie immer über eine Instanz aufgerufen werden, z. B. `a.heirate(e)`. Die Methode scheint nur das eine Argument `e` zu bekommen, in Wirklichkeit hat sie aber auch Zugriff auf `a` und wird gerade die Attribute von `a` besonders intensiv nutzen. Warum nicht gleich `heirate(a,b)`? Weil es natürlicher klingt, wenn der Akt des Heiratens von jemandem ausgeht. Das allein wäre aber nicht sehr überzeugend.

Interessant wird die Sache dann, wenn es um Vererbung geht. Die Anweisung `p2.ausgabe()` schreibt den Namen von `p2` auf den Bildschirm, also *Gertrud Schmidt*. Dieser Text mit dem Leerzeichen zwischen Vor- und Nachname wird geliefert von `toString()` und dann ausgegeben.

`Mensch` hat ebenfalls die Methode `ausgabe()`, aber nur als Erbe von `Person`. Was bewirkt also `a.ausgabe()`? Was auszugeben ist, wird von `toString()` geliefert. Aber von welchem? `a` ist ein `Mensch` und hat folglich zwei davon, die geerbte und die neue, die auch noch den Namen der Mutter ausgibt. Da es hier um wichtige Feinheiten geht, untersuchen wir die Sache sehr ausführlich:

Wäre `ausgabe` statisch, so wäre der Aufruf `toString()` sinnlos, weil unklar wäre, wessen `toString()`-Methode aufgerufen werden soll. Nun ist aber `ausgabe` nicht statisch und damit die Methode eines bestimmten Objekts (in diesem Fall von `a`). Wenn da also steht `toString()`, dann wird sinnvollerweise die `toString()` von `a` aufgerufen und davon gibt es zwei Stück.

Wären die beiden `toString()`-Methoden (die aus der Klasse `Person` und die aus der Klasse `Mensch`) `static`, so würde diejenige aufgerufen, die zur Kompilierzeit schon bekannt ist, also diejenige aus der Klasse `Person`. Dass es die Klasse `Mensch`

später auch geben würde, hätte den Compiler nicht interessieren müssen. Der Compiler hätte also die einzig vorhandene Methode verwendet – das nennt man übrigens *statische Bindung*.

Nun ist aber die `toString()`-Methode *nicht* `static`. Es wird also gewünscht, dass erst nachher, wenn das Programm läuft, nachgesehen wird, was für ein Ding das ist, dessen `toString()` aufgerufen werden soll und ob es nicht vielleicht selber eine hat, die „aktueller“ ist. Das nennt man *dynamische Bindung*. Obwohl also die `ausgabe`-Methode der Vorfahrklasse aufgerufen wird, verwendet diese automatisch die `toString` von `Mensch`, weil `a` ein `Mensch` ist und nicht nur eine `Person`.

Dieses Verhalten ist in den allermeisten Fällen das, was der Programmierer will, weshalb statische Methoden fast keine Rolle spielen, wenn es um Vererbung geht!

In unserem Beispiel ist der Unterschied zwischen den beiden `toString()` so gering, dass man geneigt wäre, die Vorfahrmethode aufzurufen, um Vor- und Nachname zu bekommen und dann nur noch den Namen der Mutter anzuhängen. Dies erreicht man mit dem `super`-Aufruf:

```
public String toString(){
  return super.toString()+", Mutter: "+mutter.vorname;
}
```

1.6 Verdecken, Überschreiben, Überladen

Wird im Nachfahren eine statische Methode neu definiert, die der Vorfahre schon hat, so sagt man, die neue *verdeckt* die alte, weil zwar die Nachfahrklasse die neue verwendet, die Vorfahrklasse aber weiterhin nur die alte sieht und nutzt (statische Bindung).

Wird im Nachfahren eine nicht-statische Methode `m` neu definiert, die der Vorfahre schon hat, so sagt man, die neue *überschreibt* die alte, weil nun sogar eine Methode `k` der Vorfahrklasse, die bisher die Methode `m` der Vorfahrklasse aufgerufen hat, nun automatisch die entsprechende neue Methode `m` aufruft (dynamische Bindung), auch wenn `k` gar nicht neu definiert wird.

Von *Überladen* spricht man, wenn zwei Methoden den gleichen Namen haben, sich aber in der Art oder Anzahl der Argumente unterscheiden. Der Compiler verwechselt sie deshalb nicht, weil er ja an den Argumenten erkennt, welche Methode gemeint ist. Den Methodennamen zusammen mit Art und Reihenfolge der Argumente nennt man die *Signatur* der Methode.

Der Rückgabetyp von überladenen Methoden muss nicht gleich sein. Zwei Methoden mit gleicher Parameterliste, die sich nur im Rückgabetyp unterscheiden, sind aber verboten, weil der Rückgabetyp nicht zur Unterscheidung herangezogen wird. Merke: Es darf keine zwei Methoden mit gleicher Signatur innerhalb der gleichen Klasse geben.

1.7 Wissenswertes zu Konstruktoren

1. Einen Konstruktor schreibt man fast genauso hin, wie eine Methode. Man erkennt ihn daran, dass kein Rückgabetyp angegeben wird, sondern der Name der Klasse.
2. Einen Konstruktor ruft man mit `new` auf, was dazu führt, dass einige selbstverständliche Aufgaben ausgeführt werden, auch wenn man sie nicht ausdrücklich verlangt hat. Das Selbstverständliche ist, dass der Speicher reserviert wird, der für die Attribute einer neuen Instanz nötig ist. Der Inhalt von Attributen ist anfangs immer `0` oder `0.0` oder `null` oder `false`. Wenn etwas anderes gewünscht wird, kann man das im Konstruktor verlangen. Die folgende Klasse `Dummy` hat zwei Konstruktoren.

```
public class Dummy{
  int x;

  public Dummy(){} //Default-Konstruktor

  public Dummy(int wert){
    x=wert;
  }
}

class Fremd{
  public static void main(String[] args){
    Dummy d=new Dummy();
    d.x=14;
    Dummy e=new Dummy(1);
  }
}
```

   Der erste scheint gar nichts zu tun, macht aber mit jedem `new Dummy()` einen neuen Abdruck für ein `int x` im Speicher und liefert einen Zeiger darauf zurück. Der zweite tut das auch, befüllt das `x` aber noch mit dem mitgegebenen Wert. Danach gibt es also zwei Dummys. Der eine speichert den Wert 14, der andere den Wert 1.
3. Manchmal braucht man nur einen Konstruktor wie den ersten, der außer der Speicherreservierung nichts tut. In diesem Fall ist es erlaubt, gar keinen Konstruktor notieren. Der Compiler denkt sich dann den Default-Konstruktor automatisch dazu. Hat man aber einen anderen als den Default-Konstruktor selber formuliert und man möchte auch den Default-Konstruktor haben, dann muss man ihn formulieren. Formuliert man ihn nämlich nicht, denkt der Compiler, dass man ihn ausdrücklich nicht haben will.

4. Bei der Formulierung eines Nachfahren nennt man bekanntlich nur noch die Attribute, die hinzukommen! Was man als vorhanden voraussetzt, erkennt der Compiler am `extends`. Jede Instanz der folgenden Klasse hat also nicht nur das genannte `y`, sondern auch ein `x`.

```
public class Subdummy extends Dummy{
  int y;

  public Subdummy(int wert){
    super(2*wert);
    y=-wert;
  }
}

class Fremd{
  public static void main(String[] args){
    Dummy d=new Dummy(1);
    Dummy e=new Subdummy(1);
  }
}
```

   Hier werden zwei Dummys erzeugt. Das eine speichert den Wert 1, das andere die Werte 2 und −1. Beachten Sie, dass ein `Subdummy` in einer Variable des Typs `Dummy` abgespeichert werden kann, weil jedes `Subdummy` auch ein `Dummy` ist.

   Beachten Sie zudem, dass *unbedingt* als allererster Befehl eines Konstruktors ein Konstruktor der Vorfahrklasse aufgerufen werden muss, damit auch Speicher für die Vorfahrattribute reserviert wird. Das geschieht durch den Befehl `super(2*wert)`. In diesem Beispiel muss also der Vorfahr einen Konstruktor haben, der ein `int` nimmt. So einen gibt es.

   Ohne diesen eigens formulierten `super`-Aufruf würde der Compiler selbständig ein `super()` als ersten Befehl einfügen! Damit das gut geht, muss die Vorfahrklasse aber den Default-Konstruktor haben. Wenn man den weggewünscht hat (siehe vorheriger Punkt), ist er nicht mehr da und man bekommt dann eine Fehlermeldung.

5. Des Öfteren kommt es vor, dass man einen Konstruktor schon ausformuliert hat und einen anderen implementieren möchte, der fast das gleiche tut, aber dessen Argumente leicht abweichen. Dann kann der andere auch den einen direkt aufrufen mit `this(...)`. Hier hat `this` eine ganz eigene Bedeutung. Man könnte z. B. in der obigen Klasse `Subdummy` den Default-Konstruktor so definieren:

```
  public Subdummy(){
    this(1);
  }
}
```

Damit würde dann der andere Konstruktor mit dem Wert 1 aufgerufen.

1.8 Übungsaufgaben

1. Im Programm `Mastermindinvers` muss für jedem Durchgang gespeichert werden, welcher Tip wieviele Plusse und Minusse erbracht hat, damit die folgenden Tips so gestaltet werden können, dass sie nicht im Widerspruch zu den bisherigen stehen.

   Implementieren Sie eine Klasse `TipPlusMinus`, die das leistet und die zusätzlich die Methode `boolean widerspruchZu(String vorschlag)` anbietet. Die Methode soll ermitteln, ob ein neuer Vorschlag überhaupt in Frage kommt, was ja nur dann der Fall ist, wenn kein bisheriger `TipPlusMinus` im Widerspruch dazu steht.

2. Erzeugen Sie in einem Hilfsprogramm einen Menschen `m` und untersuchen Sie den Unterschied zwischen
   `System.out.println(m.toString())` und
   `System.out.println(m)`.

3. Implementieren Sie in der Klasse `Mensch` die folgenden Methoden:

   `boolean istGeschwister(Mensch m)` soll ermitteln, ob `m` ein Geschwister ist. (Als Geschwister gelten Menschen, die den gleichen Vater oder die gleiche Mutter haben. Außerdem gilt jeder Mensch als sein eigenes Geschwisterchen.)

   `Mensch[] alleGeschwister()` soll alle Geschwister (und Halbgeschwister) eines Menschen liefern und zwar genau einmal. Der Mensch selber soll auch dabei sein.

   Schreiben Sie immer auch eine `main`-Methode, die einige Objekte erzeugt und ihre Funktionalität testet!

4. Ein Array hat den Nachteil, dass es nicht wachsen kann. Braucht man mehr Platz als anfänglich angenommen, so muss man ein neues Array erzeugen, das größer ist und die Daten des alten in das neue kopieren. Ab dann verwendet man das neue Array. Um dieses Problem nicht jedesmal neu lösen zu müssen, wenn es wieder auftritt, implementieren wir eine Klasse `Speicher`, die genau diese Leistung durch geeignete Konstruktoren und Methoden anbietet.

   - `Speicher()`, `Speicher(int anz)` sowie `Speicher(Object[] os)` sind angemessene Konstruktoren. Überlegen Sie, was sie leisten sollen.
   - `int size()` gibt die Anzahl der gespeicherten Elemente zurück.
   - `Object get(int pos)` gibt das Element an Position `pos` zurück.
   - `void insert(Object o, int pos)` fügt `o` an Position `pos` ein und gibt die Einfügeposition zurück. Wenn `pos` größer ist als die erste freie Position

ganz hinten, so wird stattdessen diese verwendet. Bei `pos<0` wird ganz vorne eingefügt. `null` soll nicht eingefügt werden können. Wird es versucht, wird -1 zurück gegeben.

- `Object delete(int pos)` entfernt das Element an Position `pos` (die anderen Elemente rücken zusammen) und gibt das Object zurück. Bei illegalen `pos`-Werten verhält sich die Methode analog zu `insert`.
- `Object[] getArray()` gibt ein Array zurück, das genau alle vorhandenen Elemente beinhaltet.

Schreiben Sie auch eine `main`-Methode, die alles ausführlich testet.

2 Lineare Listen und Verwandte

Einer der wichtigsten Datentypen in der Informatik ist die *Lineare Liste* oder kurz *Liste*. Die einzelnen Elemente sind von Nachbar zu Nachbar verbunden wie eine Perlenkette. Der Vorteil gegenüber einem Array ist, dass die Lineare Liste nicht in ihrer Größe begrenzt ist. Kommt ein neues Element hinzu, so wird es irgendwo gespeichert und das vormals letzte Element bekommt einen Zeiger auf das neue. Bei einem Array ist das nicht möglich, weil alle Elemente in aufeinander folgenden Speicherzellen liegen müssen und nicht davon ausgegangen werden kann, dass *direkt* rechts davon noch Platz ist, wenn noch welcher gebraucht wird. Die Nachteile der Linearen Liste sind folglich der größere Speicherverbrauch für die Verkettung der Nachbarn und der langsamere Zugriff. Will man das n-te Element, so muss man alle Elemente vom nullten bis zum $(n-1)$-ten ebenfalls abklappern.

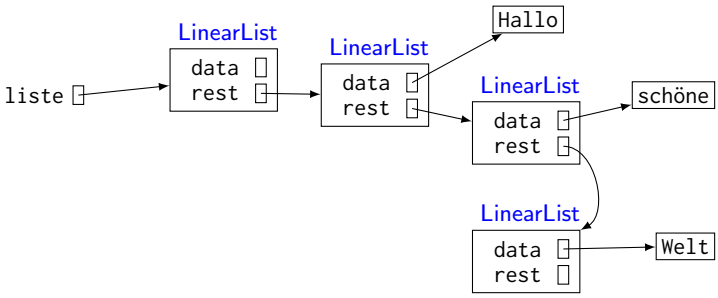
Damit die leere Liste kein Sonderfall ist, hat jede Liste ein Kopfelement, das nur als Anfang dient und dessen Datum ignoriert wird. Die leere Liste hat nur das Kopfelement.

Da es im Sinne objektorientierter Programmierung sehr lehrreich ist, sehen wir uns zwei verschiedene Implementierungen an.

2.1 Religiöse Implementierung von `LinearList`

In dieser Philosophie *ist* eine Perlenkette eine Perle, an der eine *Perlenkette* hängt.

Eine Lineare Liste trägt ihre Daten in folgender Form: Sie trägt *ein* Datum und eine restliche Lineare Liste, die entsprechend die restlichen Daten trägt. Ist die restliche Liste leer, gilt die Lineare Liste, die nur noch das Kopfelement hat, als leer[1].



[1] Man könnte auch `liste=null` als leere Liste definieren, aber das wäre eine schlechte Idee, weil `null` keine Lineare Liste ist. Befehle wie `null.insert(...)` gehen nicht.

LinearList
Object data # LinearList rest
+ LinearList() # LinearList(Object) # LinearList forward(int) + boolean isempty() + int length() + Object get(int) + void insert(Object, int) + Object delete(int) + int indexOf(Object, int) + boolean equals(Object) + String toString()

```
public class LinearList{
  protected Object data;
  protected LinearList rest;

  protected LinearList(Object obj){
    data=obj;
  }

  public LinearList(){this(null);}

  protected LinearList forward(int i){
    LinearList wander=this;
    while(wander.rest!=null && --i>=0)
      wander=wander.rest;
    return wander;
  }

  public boolean isempty(){return rest==null;}

  public int length(){
    int len=0;
    LinearList wander=this;
    while(wander.rest!=null){
      wander=wander.rest;
      len++;
    }
    return len;
  }

  public Object get(int weiter){
    LinearList vor=forward(weiter);
    if(vor.isempty()) return null;
    else return vor.rest.data;
  }

  public void insert(Object obj, int weiter){
    LinearList neu=new LinearList(obj);
    LinearList vor=forward(weiter);
    neu.rest=vor.rest;
    vor.rest=neu;
  }
```

```
public Object delete(int weiter){
  LinearList vor=forward(weiter);
  if(vor.isempty()) return null;
  Object obj=vor.rest.data;
  vor.rest=vor.rest.rest;
  return obj;
}

public int indexOf(Object obj, int start){
  LinearList wander=this;
  for(int i=0; i<start; i++){
    if(wander.rest==null) return -1;
    wander=wander.rest;
  }
  while(wander.rest!=null){
    wander=wander.rest;
    if(wander.data==obj) return start;
    start++;
  }
  return -1;
}

public boolean equals(Object obj){
  if(!(obj instanceof LinearList)) return false;
  LinearList ll=(LinearList)obj;
  LinearList wander=this;
  while(wander.rest!=null){
    if(ll.rest==null) return false;
    wander=wander.rest;
    ll=ll.rest;
    if(!wander.data.equals(ll.data)) return false;
  }
  return ll.rest==null;
}

public String toString(){
  LinearList wander=this;
  StringBuffer sb=new StringBuffer();
  while(wander.rest!=null){
    wander=wander.rest;
    sb.append(wander.data);
    sb.append(" ");
  }
  return sb.toString();
}
```

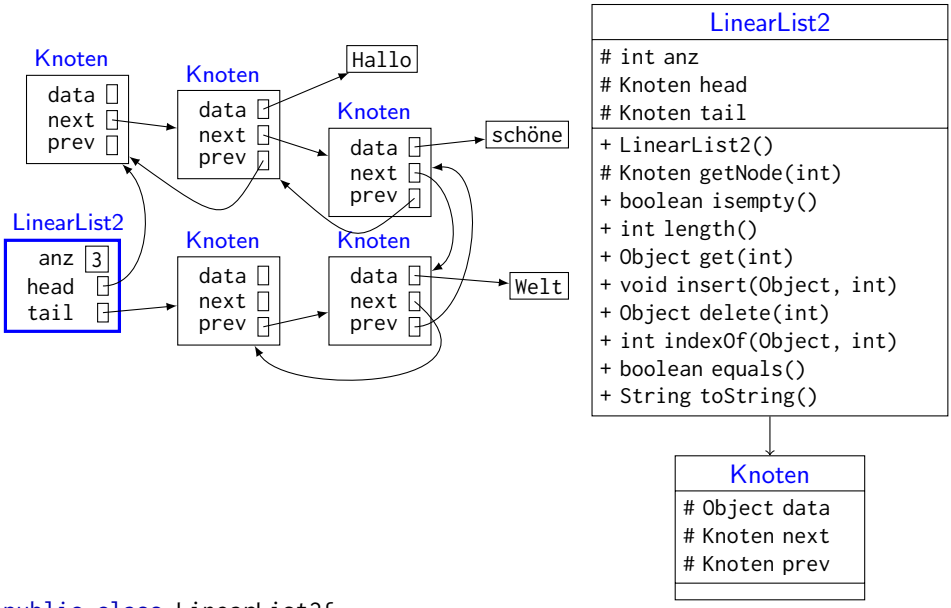
2.2 Pragmatische Implementierung von `LinearList`

Die oben geschilderte Liste ist wegen ihrer Minimalität von theoretischer Schönheit. Technisch gesehen ist sie aber an mancher Stelle unpraktisch. Zur Bestimmung der Länge etwa ist es jedesmal nötig, die ganze Liste zu durchlaufen, um zu zählen, aus wie vielen Elementen sie besteht.

Diese Länge könnte man sich ganz billig auch in einem `int len` merken und immer hochzählen, wenn ein Element angefügt wird, bzw. herunterzählen, wenn eines entnommen wird. Wenn man `len` aber der Liste als Attribut gibt, hat es jedes Element. Das ist unschön und verschwenderisch.

Deshalb baut man um die Kette herum eine Schachtel, in der die Kette liegt. Der Benutzer kennt nur die Schachtel, die deshalb nun `Liste` genannt wird, während man die Kettenelemente dann lieber `Knoten` nennt. Die Schachtel gibt es für jede Kette nur einmal, und sie kann zwanglos das Attribut `len` bekommen. Außerdem schadet es nicht, wenn man die ganzen Methoden auch gleich der Schachtel angliedert und nicht den Knoten.

Wir zeigen hier eine Implementierung mit doppelter Verkettung, wo jeder Knoten nicht nur einen Zeiger `next` auf seinen Nachfolger hat, sondern auch einen Zeiger `prev` auf seinen Vorgänger. Das kostet zusätzlich Speicherplatz, macht aber einen Durchlauf in umgekehrter Richtung möglich.




```
public class LinearList2{
  protected int anz;
  protected Knoten head;
  protected Knoten tail;
```

```
public LinearList2(){
  head=new Knoten(null);
  tail=new Knoten(null);
  head.next=tail;
  tail.prev=head;
  anz=0;
}

protected Knoten getNode(int i){
  Knoten wander=head;
  while(wander.next!=tail && i-->=0)
    wander=wander.next;
  return wander;
}

public boolean isempty(){return anz==0;}

public int length(){return anz;}

public Object get(int wo){
  if(wo<0 || wo>=anz) return null;
  else return getNode(wo).data;
}

public void insert(Object obj, int wo){
  Knoten neu=new Knoten(obj);
  Knoten vor=getNode(wo-1);
  Knoten nach=vor.next;
  neu.next=nach;
  neu.prev=vor;
  vor.next=nach.prev=neu;
  anz++;
}

public Object delete(int wo){
  if(wo<0 || wo>=anz) return null;
  Knoten vor=getNode(wo-1);
  Knoten akt=vor.next;
  Knoten nach=akt.next;
  vor.next=nach;
  nach.prev=vor;
  anz--;
  return akt.data;
}
```

```
  public int indexOf(Object obj, int start){
    if(start>=0 && start<anz){
      Knoten wander=getNode(start-1);
      while(wander.next!=tail){
        wander=wander.next;
        if(wander.data==obj) return start;
        start++;
      }
    }
    return -1;
  }

  public boolean equals(Object obj){
    if(!(obj instanceof LinearList2)) return false;
    LinearList2 ll=(LinearList2)obj;
    if(anz!=ll.anz) return false;
    Knoten a=head.next;
    Knoten b=ll.head.next;
    while(a!=tail){
      if(a.data!=b.data) return false;
      a=a.next;
      b=b.next;
    }
    return true;
  }

  public String toString(){
    Knoten wander=head.next;
    StringBuffer sb=new StringBuffer();
    while(wander!=tail){
      sb.append(wander.data.toString());
      sb.append(" ");
      wander=wander.next;
    }
    return sb.toString();
  }

  public static class Knoten{
    public Object data;
    public Knoten next;
    public Knoten prev;

    public Knoten(Object d){data=d;}
  }
}
```

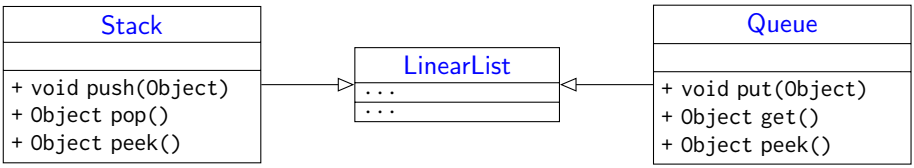
2.3 Stack und Queue

Wichtige Verwandte von Linearen Listen, sind *Stack* und *Queue*. Ein Stack, auch *Stapel* oder *Keller* genannt, ist eine Struktur, die die aufgenommenen Elemente in umgekehrter Reihenfolge wieder abgibt (FILO = first in last out = last in first out = LIFO). Eine Queue oder *Schlange* ist eine Struktur, die die aufgenommenen Elemente in der gleichen Reihenfolge wieder abgibt wie sie sie aufnimmt (FIFO = first in first out = last in last out = LILO).

Eine LinearList hat die Fähigkeiten von Stack und Queue schon, die Methoden haben nur noch nicht die traditionellen Namen. Beim Stack sind dies `push`, `peek` und `pop`, bei der Queue `put`, `peek` und `get` (alle ohne Argumente). Wie löst man dieses kleine Problem? Es gibt drei Möglichkeiten:

2.3.1 Schlampige Implementierung

Oberflächlich betrachtet sind Stack und Queue Nachfahren von LinearList; es fehlen nur noch ein paar kleine Methoden. Der Stack *ist* also eine LinearList, ebenso die Queue. Die beiden haben zwar auch Methoden, die sie gar nicht haben dürften (z. B. Löschen eines Elements irgendwo in der Mitte der Liste), aber diese Methoden werden einfach nicht verwendet. Objektorientiert betrachtet ist das ziemlich unsauber.

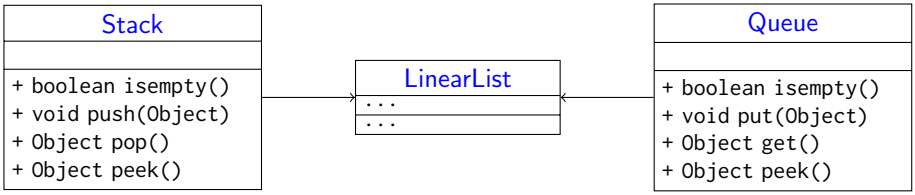



```
public class Stack1 extends LinearList{
  public void push(Object obj){insert(obj, 0);}
  public Object pop(){return delete(0);}
  public Object peek(){return get(0);}
}
```

```
public class Queue1 extends LinearList{
  public void put(Object obj){insert(obj, 0);}
  public Object get(){return delete(length()-1);}
  public Object peek(){return get(length()-1);}
}
```

2.3.2 Pragmatische Implementierung

In der strengen Lehre *ist* ein Stack keine Liste, weil er nicht deren Fähigkeiten hat. Weil er aber die Elemente sehr ähnlich verwalten will, *hat* er eine Liste, derer er sich bedient. Für den Benutzer des Stacks wird diese Tatsache nirgends sichtbar. Bezüglich der Queue gilt das gleiche.




```
public class Stack{
  LinearList list=new LinearList();

  public void push(Object obj){list.insert(obj, 0);}
  public Object pop(){return list.delete(0);}
  public Object peek(){return list.get(0);}
  public boolean isempty(){return list.isempty();}
}

public class Queue{
  LinearList list=new LinearList();

  public void put(Object obj){list.insert(obj, list.length());}
  public Object get(){return list.delete(0);}
  public Object peek(){return list.get(0);}
  public boolean isempty(){return list.isempty();}
}
```

2.3.3 Religiöse Implementierung

Konsequent gedacht haben Stack, Queue und LinearList viel gemeinsam, aber keiner kann Vorfahr der anderen sein, weil jeder mindestens eine Methode hat, die die anderen nicht haben sollten. Alle verwalten ihre Elemente aneinander gehängt, alle haben eine `isempty`-Mathode. Aber Stack hat nur `push` und kein `insert`, während es bei LinearList gerade umgekehrt ist. Was tun?

Man erfindet einen gemeinsamen Vorfahr für alle drei, der genau das hat und kann, was alle drei gemeinsam haben und können. Einen solchen Vorfahr nennt man

üblicherweise `AbstractList` und deklariert ihn auch `abstract`. Dann ist es unmöglich, Instanzen davon zu erzeugen. Da dieses Thema im Unterricht kein ernsthaftes Gewicht hat, verfolgen wir es nicht weiter.

2.4 Exceptions

An dieser Stelle sollte man sich einmal Gedanken darüber machen, ob eine so wichtige Klasse wie `LinearList` korrekt funktioniert. Ruft man beispielsweise `get` mit einem Index außerhalb der vorhandenen Grenzen auf, so bekommt man `null` zurück, was den Aufrufer auf seinen Fehler aufmerksam machen soll. Das ist auch vernünftig, wenn `null` sonst nicht möglich ist. Da es aber problemlos möglich ist, `null` auch als Datum an einen Knoten zu übergeben, kann der Aufrufer bei Erhalt von `null` nicht sicher sein, ob er nun ein korrektes Datum erhalten hat oder eine Verlegenheitsrückgabe wegen eines falschen Indexes.

Bei Zugriff außerhalb der erlaubten Grenzen, wäre eine Exception das vernünftigere Ergebnis. Wir sehen uns hier kurz an, wie man eine solche erzeugt und wie man auf ihr Auftreten reagieren kann.

Exceptions sind sehr simple Objekte. Sie können normalerweise fast gar nichts, außer sich einen String merken und ihn ausgeben. Gemeinsamer Vorfahr ist die Klasse `Exception`, die wiederum von `Object` abstammt. Auf Grund ihrer besonderen Bedeutung bekommen Exceptions in Java zwei Schlüsselwörter, die nur auf sie angewandt werden dürfen: `throw` und `throws`. In der `get`-Methode würde das so aussehen:

```
public Object get(int wo) throws IndexOutOfBoundsException{
  if(wo<0 || wo>=anz)
    throw new IndexOutOfBoundsException("Problem bei "+wo);
  else return getNode(wo).data;
}
```

Die Methode droht also im Kopf schon an, dass evtl. eine Exception „geworfen“ wird. Wenn es dann tatsächlich passiert, gibt die Methode nichts zurück! Stattdessen erlebt der Aufrufer die Exception und kann hoffentlich vernünftig darauf reagieren. Wenn er mit einer Exception rechnet, dann steht im aufrufenden Programmteil

```
public void test(){
  Object o;
  try{
    o=liste.get(122);
  }
  catch (IndexOutOfBoundsException ex){
    System.out.println(ex.toString());
    return;
  }
```

```
  finally{
    System.out.println("Die Methode ist jetzt fertig");
  }
  System.out.println("Das 122. Objekt ist "+o);
}
```

Mit `try` wird etwas versucht, das vielleicht eine Exception erzeugt. Wenn eine auftritt, wird der ganze `try`-Block sofort abgebrochen und ein `catch`-Block gesucht, der die Exception „abfängt“. Darin steht, wie mit dem Problem umzugehen ist – in diesem Beispiel ist das nicht sehr raffiniert. Noch brutaler geht es auch: Wenn man in den `catch`-Block gar nichts schreibt, wird die Exception völlig ignoriert.

Der Inhalt von `finally` wird garantiert am Ende abgearbeitet, egal ob nun eine Exception aufgetreten ist oder nicht. Das `return` im `catch`-Block wird also erst nach der Meldung in `finally` ausgeführt. Die Nachricht, dass die Methode fertig ist, kommt folglich auf jeden Fall, die Ausgabe des 122. Objekts nur, wenn keine Exception auftritt.

Auf einen `try`-Block können auch mehrere `catch`-Blöcke für verschiedene Exceptions folgen. Tritt eine Exception auf, wird bei allen `catch`-Blöcken der Reihe nach durchprobiert, ob sie für die Exception zuständig sind. Wenn eine der Exceptions in einem `catch`-Block Nachfahre einer anderen, ebenfalls aufgeführten ist, sollte ihr `catch`-Block vor dem des Vorfahren genannt werden, weil er sonst nie erreicht wird. Der Vorfahr fängt dann alle Exceptions weg, weil er ja auch zuständig ist. Gäbe es in unserem Beispiel also einen weiteren `catch`-Block, der auf `Exception` reagiert, dann sollte der nicht als erster notiert sein, weil jede `IndexOutOfBoundsException` ja auch eine `Exception` ist und deshalb weggefangen wird.

Ein praktisch sehr wichtiges Detail ist noch zu nennen: Die Erfinder von Java waren sich bewusst, dass es Exceptions gibt, die man durch saubere Programmierung vermeiden kann. Wenn man sich dann sicher ist, dass die angedrohte Exception sicher nicht auftreten wird, hat man auch keine Lust eine `try-catch`-Struktur um seine Befehle herum zu basteln. (Unsere `IndexOutOfBoundsException` ist so eine, denn man kann ja durch Aufruf von `length()` durchaus wissen, ob das entsprechende Element existiert.)

In diesem Fall ist die `Exception` ein Nachfahre von `RuntimeException` (unsere ist auch ein Nachfahre davon) und man kann sich deshalb alles auch sparen. Wenn man sich aber irrt und die Exception tritt doch auf, wird sie zum einen auf der Konsole angezeigt und das Programm verhält sich wahrscheinlich nicht korrekt, weil es ja so geschrieben ist, als ob das nicht passieren könnte.

Ist die angedrohte Exception kein Nachfahre von `RuntimeException` so hat man zwei Möglichkeiten: Entweder setzt man sie in ein `try-catch`-Konstrukt oder man markiert die Methode innerhalb der man den gefährlichen Aufruf tätigt, selber mit `throws DieseException`. Kann man die Exception also nicht behandeln, so muss man sie selber androhen.

2.5 Übungsaufgaben

1. Bauen Sie Exceptions in die `LinearList` ein, wo dies angemessen ist!
2. Modellieren und implementieren Sie eine Klasse `Iterator`. Ein solcher kennt die Innereien einer Liste und bietet die Befehle `boolean hasMoreElements()` und `Object nextElement()`. Ein Iterator ist nur einen Listendurchlauf lang brauchbar und fängt immer vorne an.
3. In einer doppelt verketteten Liste kann man von einem Element auch direkt zu seinem Vorgänger gelangen. Damit ist dann sogar ein `Cursor` möglich, also so etwas wie ein `Iterator` der vorwärts und rückwärts gehen kann. Überlegen Sie sich, welche Methoden ein `Cursor` sinnvollerweise anbieten könnte und implementieren Sie ihn.
4. Mit Hilfe eines Stacks kann man Terme berechnen, wenn sie als UPN-String daher kommen. Einige Beispiele

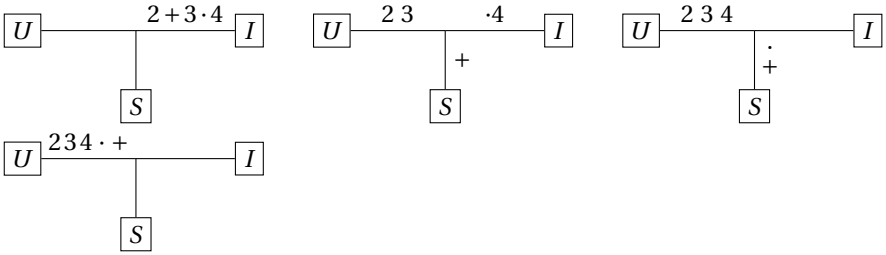
Term	UPN-String
$2+3\cdot 4$	`2 3 4 * +`
$2+(3\cdot 4)$	`2 3 4 * +`
$(2+3)\cdot 4$	`2 3 + 4 *`
$(1-(2+3\cdot 4))/(1.5+1)$	`1 2 3 4 * + - 1.5 1 + /`

   Es soll hier erst einmal nicht versucht werden, aus dem Term den UPN-String zu erstellen, das kommt in der nächsten Aufgabe. Es soll nur der UPN-String ausgerechnet werden, und das geht so:
   - Wenn eine Zahl kommt, wird sie vollständig eingelesen. (Wir gehen nur von nicht-negativen Zahlen aus.) Die eingelesene Zahl wird gepusht.
   - Wenn ein Operator kommt, popt er die korrekte Anzahl von Zahlen vom Stack (die vier Grundrechenarten nehmen immer zwei Argumente, es sind aber Operatoren denkbar, die mehr oder weniger nehmen). Nachdem er seine Operation durchgeführt haben, pusht er sein Ergebnis wieder.

   Im letzten Beispiel hat der Stack der Reihe nach folgende Inhalte: $\{1\}$, $\{1;2\}$, $\{1;2;3\}$, $\{1;2;3;4\}$, $\{1;2;12\}$, $\{1;14\}$, $\{-13\}$, $\{-13;1.5\}$, $\{-13;1.5;1\}$, $\{-13;2.5\}$, $\{-5.2\}$.

   Schreiben Sie ein Programm, das einen beliebigen korrekten UPN-String ausrechnet. Wenn Sie den Stack verwenden, der sich `Objects` merkt, können Sie die eingelesenen Zahlen als `Double` und die Operatoren als `String` darauf ablegen.

   Erweitern Sie das Programm so, dass es auch mit UPN-Queues zurecht kommt. In dieser sollen die Zahlen als `Double` abgelegt ankommen und die Operatoren als `String`. (Für die nachfolgende Aufgabe ist das von Vorteil.)
5. Nun soll aus einem String, der den Term in der üblichen Infix-Schreibweise enthält, ein UPN-String erzeugt werden. Auch dafür wird ein Stack gebraucht. Wir orientieren uns an Dreigleisbildern mit dem Stringgleis I (Infix), dem Stringgleis U (UPN) und dem Stackgleis S.



Dabei sind folgende Regeln einzuhalten:

(a) Zahlen fahren ungehindert von I nach U durch.

(b) Alle Operatoren auf S, die gleiche oder größere Priorität haben, als der bei I anstehende, gehen von S nach U. Der von I kommende Operator geht auf S.

(c) Der Operator „(“ geht auf S. Der Operator „)“ räumt S ab bis zum „(“. Keiner von beiden erscheint in U.

Erweitern Sie Ihr Programm von der vorigen Aufgabe um geeignete Methoden, um den UPN-String aus dem Term zu erstellen. Wenn Sie U als Queue anlegen statt wieder als String, hat es der andere Teil des Programms leichter.

6. Erweitern Sie die vorigen beiden Aufgaben so, dass es mehrere Operatoren beherrscht. Wichtig wären noch `+/-`, `swap`, `1/x`, `sqrt`, usw.

3 Rekursion

3.1 Zwei Beispiele

3.1.1 Sitzreihe

Ich sitze auf dem fünft-hintersten Platz einer Sitzreihe (in Java also auf Platz 4). Um heraus zu kommen, müssen alle vor mir auch aufstehen und ihren Platz verlassen. Ein typischer angemessener Algorithmus würde Platz 0 freimachen, dann Platz 1, usw. bis Platz 4. Damit wären alle Plätze verlassen, auch meiner. Für so einen *iterativen* Algorithmus würde man üblicherweise eine Schleife verwenden.

An dieses Problem kann man aber auch anders heran gehen, was umständlicher aussehen mag. Wir fangen nicht bei Platz 0 an sondern bei Platz 4 und halten uns für *alle* Plätze an genau das gleiche Rezept:

> *Wenn der Platz vorher besetzt ist, führen wir* dieses *Rezept einen Platz weiter vorne durch, andernfalls bzw. danach geht der Sitzende weg.*

Oder als Pseudoprogramm:

```
public void willraus(int wer){
  if(besetzt(wer-1)) willraus(wer-1);
  geheweg(wer);
}
```

So eine Vorgehensweise nennt man *rekursiv*. Wir wollen dieses Rezept nun ganz genau befolgen. Es wird auf Platz 4 angewandt:

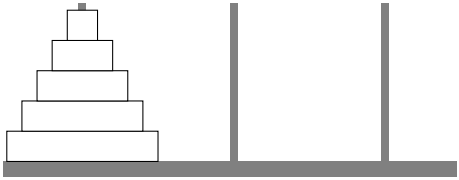
Platz 4 stellt fest, dass der Platz davor besetzt ist, also wird das Rezept auf Platz 3 angewandt: Platz 3 stellt fest, dass der Platz davor besetzt ist, also wird das Rezept auf Platz 2 angewandt: Platz 2 stellt fest, dass der Platz davor besetzt ist, also wird das Rezept auf Platz 1 angewandt: Platz 1 stellt fest, dass der Platz davor besetzt ist, also wird das Rezept auf Platz 0 angewandt: Platz 0 stellt fest, dass der Platz davor unbesetzt ist. Platz 0 geht jetzt weg. Platz 1 geht jetzt weg. Platz 2 geht jetzt weg. Platz 3 geht jetzt weg. Platz 4 geht jetzt weg.

Man erkennt, dass das geht, käme aber nie auf die Idee, so ein einfaches Problem so kompliziert anzugehen. Beim nächsten Problem sieht die Sache schon ganz anders aus:

3.1.2 Die Türme von Hanoi

Auf dem ersten von drei Stäben ist eine Menge von nach oben hin kleiner werdenden Klötzen aufgebaut. Dieser Turm soll vom ersten Stab zum letzten versetzt werden,

wobei folgende Regeln einzuhalten sind: Es darf immer nur ein Klotz bewegt werden und es darf niemals ein größerer Klotz auf einem kleineren liegen.



Dafür eine iterative Lösung zu finden, ist schon ziemlich schwer. Versuchen Sie es! Rekursiv ist die Lösung schockierend einfach. In der folgenden Methode haben die Stäbe die Nummern 0 bis 2. Es wird angegeben, `von` wo ein Teilturm genommen werden soll, `nach` wo er bewegt werden soll und wie `hoch` er von oben gemessen ist. Die Methode `vonNach` verschiebt einen einzelnen Klotz.

```
public void bewege(int von, int nach, int hoch){
  if(hoch>0){
    int hilf=3-von-nach;
    bewege(von, hilf, hoch-1);
    vonNach(von, nach);
    bewege(hilf, nach, hoch-1);
  }
}
```

In Worten wird also wenn ein 5er-Turm von 0 nach 2 zu versetzen ist, zuerst ein 4er-Turm (Deckel) von 0 nach 1 bewegt, dann der verbleibende Klotz von 0 nach 2 und dann wieder der Deckel von 1 nach 2. Die Bewegung des Deckels geht genauso vor sich wie die Bewegung des ganzen Turms. Nur dass der Deckel um einen Klotz kleiner ist. Das ist wichtig, weil man damit sicher sein kann, dass die Methode auch irgendwann fertig ist.

Man muss sich erst eine Weile daran gewöhnen, dass die Methode für alle Deckelhöhen funktionieren muss (also auch für `hoch==0`), weil sie ja für alle Deckelhöhen auch aufgerufen wird. Wenn ein Deckel wegzuschaffen ist, dann muss die Nummer des einzig möglichen Hilfsstabs ermittelt werden. Auch dieses `hilf` ändert sich bei jedem Aufruf automatisch.

3.2 Definition und Anmerkungen

Definition Ein Algorithmus heißt *rekursiv*, wenn er sich direkt oder indirekt selber aufruft. Ein indirekter Aufruf liegt vor, wenn zuerst ein anderer Algorithmus aufgerufen wird, der dann aber wieder diesen aufruft. Nicht rekursive Algorithmen nennt man *iterativ*.

Anmerkungen:

- Rekursive Algorithmen müssen eine Abbruchbedingung haben, sonst rufen sie sich immerfort selber auf und kehren nicht mehr zurück. Die Abbruchbedingung muss vor dem erneuten Aufruf stehen, sonst wird sie nicht erreicht.
- Rekursion wird oft angewandt, wenn es sich um ein Problem der Familie *Teile und herrsche* handelt. Das Hanoi-Problem ist so eines: Ungefähr die Hälfte der Arbeit macht der erste rekursive Aufruf (Deckel wegschaffen), dann wird fast nichts aber doch ein bisschen gearbeitet (ein Klotz versetzt) und dann wird nochmal ungefähr die Hälfte der Arbeit im zweiten rekursiven Aufruf erledigt (der weggeschaffte Deckel kommt oben drauf).

  Das erste Beispiel ist keines der Form *teile und herrsche*. Es wird bei jedem Aufruf fast nichts aber doch ein bisschen gearbeitet. Solche Probleme kann man meist iterativ einfacher lösen.
- Rekursion ist weniger mächtig als Iteration: Alle rekursiven Algorithmen kann man in iterative umwandeln, auch wenn das oft sehr schwierig ist. Aber nicht alle iterativen Algorithmen lassen sich in rekursive umwandeln.
- Wenn ein Problem eine einfache iterative Lösung hat, ist diese normalerweise vorzuziehen, weil rekursive Algorithmen langsamer laufen und meist mehr Speicher brauchen (siehe weiter unten). Braucht die iterative Lösung jedoch einen Stapel, so findet man oft eine bessere rekursive Lösung.

3.3 Technischer Hintergrund

Was passiert im Speicher des Computers, wenn eine Methode sich selber aufruft? Das, was immer passiert, wenn eine neue Methode aufgerufen wird. Alle Argumente, die die aktuelle Methode bekommen hat und alle Variablen, die sie deklariert, werden in einem *Stackframe* gesammelt und auf dem Systemstack abgelegt. Außerdem wird sich im Stackframe gemerkt, wo das Programm vor dem Aufruf war. Wenn dann die Methode fertig ist und zurückkehrt, wird der letzte Stackframe wieder vom Stack geholt. Alle Variablen haben dann wieder die Werte von vor dem Aufruf und das Programm macht genau an der Stelle hinter dem Aufruf weiter.

Den Stack mit den Stackframes stellen wir uns so vor, dass immer links ein Stackframe hinzugefügt und entnommen wird. Am Anfang ist der Stack leer. Ein Stackframe beinhaltet die Daten (*von, nach, hoch, hilf, woweiter*). Bei *woweiter* wird die Zeilennummer angegeben, wo das Programm weiter macht, nachdem es von einem Methodenaufruf zurückgekehrt ist.

Im weiter unten folgenden Schema wird blau der aktuelle Inhalt der Variablen angezeigt (ein Unterstrich bedeutet, dass der Wert vorerst undefiniert ist). Als letzter Eintrag erkennt man die Nummern der abgearbeiteten Zeilen. Rot erscheint der Stackinhalt, der durch die blauen Aktionen entsteht. Wird der Stack am Ende eines

Schrittes kleiner, so haben die Variablen die Werte im geholten Stackframe bekommen.

Wir sehen uns die Vorgänge an einem übersichtlichen Beispiel an, einem Hanoi-Turm der Höhe 2, der von Position 0 nach Positon 1 bewegt werden soll. Hier nochmal die Methode, die mit `bewege(0,1,2)` von außerhalb aufgerufen wird.

```
1 public void bewege(int von, int nach, int hoch){
2   if(hoch>0){
3     int hilf=3-von-nach;
4     bewege(von, hilf, hoch-1);
5     vonNach(von, nach);
6     bewege(hilf, nach, hoch-1);
7   }
8 }
```

Wir starten also mit `von=0`, `nach=1` und `hoch=2`. In Zeile 3 wird `hilf=2` berechnet und in Zeile 4 kommen vor dem `bewege`-Aufruf alle Variablen auf den Stack, also (0,1,2,2,5). (Die 5 rührt daher, dass es nach dem Aufruf von `bewege` in Zeile 5 weiter geht.) Der Aufruf in Zeile 4 lautet `bewege(0,2,1)` und damit beginnt die Geschichte wieder von vorne aber mit anderen Argumenten.

(0,1,2,_2, 1-2-3-4)		(0,1,2,2,5)
(0,2,1,_1, 1-2-3-4)		(0,2,1,1,5), (0,1,2,2,5)
(0,1,0,_, 1-2-7)		(0,1,2,2,5)
(0,2,1,1, 5-6)	Klotz_1 von 0 nach 2	(0,2,1,1,7), (0,1,2,2,5)
(1,2,0,_, 1-2-7)		(0,1,2,2,5)
(0,2,1,1, 7-8)		
(0,1,2,2, 5-6)	Klotz_2 von 0 nach 1	(0,1,2,2,7)
(2,1,1,_0, 1-2-3-4)		(2,1,1,0,5), (0,1,2,2,7)
(2,0,0,_, 1-2-7)		(0,1,2,2,7)
(2,1,1,0, 5-6)	Klotz_1 von 2 nach 1	(2,1,1,0,7), (0,1,2,2,7)
(0,1,0,_, 1-2-7)		(0,1,2,2,7)
(2,1,1,0, 7-8)		

Es erfordert ziemlich viel Konzentration, diese Vorgänge genau nachzuvollziehen. Viel einfacher hat man es, wenn man einem modernen Debugger bei der Arbeit zuschaut und dabei den Stack im Auge behält. Dies ist sowohl mit Netbeans als auch dem Röhner-Editor leicht möglich.

3.4 Aufgaben

Um rekursive Algorithmen zu verstehen, ist es manchmal hilfreich, den Stackinhalt auf einem Blatt Papier mitzuschreiben und den Algorithmus selber Schritt für Schritt durchzuführen. Will man selber rekursive Algorithmen zu einem Problem finden, so sollte man erst einmal versuchen, die Lösung sprachlich auszudrücken. Wir üben dies an mehreren Beispielen:

1. Gegeben ist eine `LinearList` mit etlichen Elementen. Die Reihenfolge der Elemente soll umgekehrt werden. Es soll also aus A-B-C-D die Anordnung D-C-B-A werden. Wir formulieren sprachlich:

   ```
   drehNochVorhandeneListeUm(){
     Wenn die Liste nicht leer ist{
       entnimm das vorderste Element;
       drehNochVorhandeneListeUm();
     }
     hänge das entnommene Element hinten an;
   }
   ```

   Die Umsetzung in die Programmiersprache ist schon nicht mehr der Rede wert. Geben Sie auch einen iterativen Algorithmus an!

2. Die Funktion *Fakultät* ist definiert für alle natürlichen Zahlen. Es gilt $f(0) = 1$ und $f(n) = n \cdot f(n-1)$. Diese Definition ist rekursiv und folglich ohne Probleme sofort programmierbar. Finden Sie in diesem einfachen Fall auch einen iterativen Algorithmus.

   ```
   public int fact(int n){
     if(n==0) return 1;
     return n*fact(n-1);
   }
   ```

3. Gegeben ist ein waagrechter Bereich in einer Graphik durch seine linke und rechte Koordinate. Zeichne ein Lineal, das in der Mitte einen langen Strich hat, in der Mitte der entstandenen Mitten einen etwas kürzeren und so immer weiter, bis die Strichlänge 0 erreicht ist.

   Diese Formulierung ist schon so klar rekursiv, dass man sie direkt in die Programmiersprache übersetzen kann.

4. Aus einer Reihe verschiedener Elemente sollen alle möglichen Anordnungen (Permutationen) gefunden werden, also etwa 1234, 1243, 1324, 1342,....

   Für eine Permutation vertauschen wir jedes Element einmal mit dem vordersten (auch das vorderste selber, was ja keine negativen Auswirkungen hat) und permutieren alle ab dem nächsten. Danach machen wir den Tausch wieder rückgängig, damit der nächste Tausch nicht mit dem falschen Element erfolgt. Wenn nichts mehr zu permutieren ist, geben wir die Menge aus. Beim ersten Aufruf muss darauf geachtet werden, dass das 0. Element das vorderste ist.

   Mit dieser sprachlich sauber formulierten Idee können wir das Programm sofort hinschreiben. Aber man sieht schon, dass die sprachliche Formulierung recht schwer war, wenn man keine Fehler machen will.

   ```
   public void permutiere(int[] array, int left){
     if(left>=array.length-1){
       for(int i=0; i<array.length; ++i)
   ```

```
      System.out.print(array[i]);
    System.out.println();
  } else
    for(int i=left; i<array.length; ++i){
      int h=array[left]; array[left]=array[i]; array[i]=h;
      permutiere(array, left+1);
      h=array[left]; array[left]=array[i]; array[i]=h;
    }
}
```

5. Was geben die folgenden Methoden aus, wenn man sie mit dem Wert 5 aufruft?

```
public void rek1(int i){
  if(i<0) return;
  System.out.print(i+" ");
  rek1(i-1);
}

public void rek2(int i){
  if(i<0) return;
  rek2(i-1);
  System.out.print(i+" ");
}

public void rek3(int i){
  if(i<0) return;
  rek3(i-1);
  System.out.print(i+" ");
  rek3(i-1);
}
```

6. Finden Sie das Bildungsgesetz hinter der Zahlenfolge
   0, 010, 0102010, 010201030102010, 01020103010201040102010301020 10, ...
   Schreiben sie dann ein rekursives Programm, das ein Argument n nimmt und dann die n-te dieser Zahlen ausgibt.

7. Das Straßennetz von modernen Städten ist schachbrettartig angelegt. Ein Taxifahrer befördert einen Stammkunden jeden Tag auf einer kürzesten Strecke von $A(0,0)$ nach $B(5,3)$, also etwa auf dem Weg $(0,0)$–$(1,0)$–$(1,1)$–$(2,1)$–$(2,2)$–$(3,2)$–$(3,3)$–$(4,3)$–$(5,3)$. Schreiben Sie ein rekursives Programm, das alle kürzesten Strecken auf dem Textschirm ausgibt.

8. Im Morsealphabet gibt es nur die Zeichen *Punkt* und *Strich*. Ein Punkt benötigt eine Zeiteinheit, ein Strich zwei. Schreiben Sie ein rekursives Programm, das alle möglichen Kombinationen aus Punkt und Strich erzeugt, die in n Zeiteinheiten passen.

   Man morst, indem man einen Punkt oder einen Strich macht und dann morst;

dies natürlich nur, falls noch Platz ist. Falls kein Platz mehr ist, ist der richtige Moment für die Ausgabe gekommen, nicht vorher!

Das folgende Programm zeigt zwei Lösungen. Aus Sicht der rekursiven Programmierung ist die erste viel schöner, aus praktischer Sicht ist die zweite besser.

```
public class Morse{
  public static void main(String[] args){
    int n=Integer.parseInt(args[0]);
    morse("", n);//elegant mit vielen Speicherleichen
    char[] wort=new char[n];//wiederverwendet aber unelegant
    System.out.println("Punkt=o   Strich=<>");
    morse(wort, 0);
  }

  public static void morse(String schon, int noch){
    if(noch<1) System.out.println(schon);
    else{
      morse(schon+"o", noch-1);
      if(noch>1) morse(schon+"-", noch-2);
    }
  }

  public static void morse(char[] wort, int pos){
    if(pos>=wort.length)// Wort fertig
      System.out.println(wort);
    else{
      if(pos<wort.length){// Punkt hat noch Platz
        wort[pos]='o';
        morse(wort, pos+1);
      }
      if(pos+1<wort.length){//sogar ein Strich hat Platz
        wort[pos]='<';
        wort[pos+1]='>';
        morse(wort, pos+2);
      }
    }
  }
}
```

9. Gesucht ist eine rekursive Methode, die alle Teilmengen einer Gesamtmenge erzeugt. Die Gesamtmenge wird als `int[]` mit paarweise verschiedenen Zahlen übergeben. Finden Sie auch eine iterative Lösung.
10. Schreiben Sie ein rekursives und ein iteratives Programm, das auf dem Textbildschirm die Flagge von Alphanumerika erzeugt.

```
        ********
    ****    ****
  **  **  **  **
* * * * * * * *
```

11. Im Jahre 1928 hat Wilhelm Ackermann eine Funktion genannt, die ungewöhnlich schnell anwächst. Sie hat zwei natürliche Argumente m und n. Es gilt

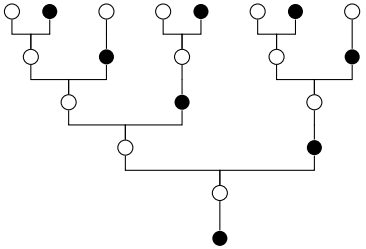
$$\begin{aligned} \text{ack}(0,n) &= n+1 \\ \text{ack}(m,0) &= \text{ack}(m-1,1) \\ \text{ack}(m,n) &= \text{ack}(m-1,\text{ack}(m,n-1)) \end{aligned}$$

    Berechnen Sie ein paar Werte, z.B. ack(0,0), ack(1,0), ack(2,2), ack(4,1), ack(4,4) auf dem Papier. Wenn die Rechnungen zu schwer werden, können Sie die Funktion leicht programmieren. Sehen Sie ein, warum Sie ack(5,5) nie erfahren werden?

12. Aus dem unbefruchteten Ei einer Bienenkönigin (○) schlüpft stets ein Männchen (Drohn ●). Das Bild zeigt den Stammbaum eines Drohns. Wenn wir mit $d(n)$ die Anzahl der Drohnen und mit $k(n)$ die Anzahl der Königinnen vor n Generationen bezeichnen, so gilt

$$d(n) = k(n-1) \quad \text{und} \quad k(n) = d(n-1) + k(n-1)$$

    Überprüfen Sie das an Hand des Bildes und schreiben Sie dann ein Programm, zur Berechnung von $d(n)$ und $k(n)$.



    Die beiden Funktionen sollten der Einfachheit halber rekursiv programmiert sein und sich gegenseitig aufrufen. Überlegen Sie sich jeweils ein geeignetes Abbruchkriterium.

13. Welche verschiedenen Arten gibt es, 1€ in 1, 2, 5, 10, 20, 50 Cent zu wechseln? Wie viele verschiedene Arten gibt es? (Die Beantwortung der zweiten Frage kann evtl. zu einem ganz anderen Programm führen als die erste.)

14. Von einem der Einfachheit halber vollständig geklammerten Term aus den vier Grundrechenarten und positiven Zahlen soll der Wert berechnet werden. Ein Term kann so einfach aussehen wie $(21 \cdot 3)$, aber auch komplizierter wie $(3 + (((12.1-8)-(4/7)) \cdot 12))$. Bei der Auswertung muss wegen der vollständigen Klammerung nicht die *Punkt-vor-Strich*-Regel berücksichtigt werden. Außerdem hat

jeder Operator genau zwei Operanden. Der Term wird dem teilweise schon vorliegenden Programm `BerechneKlammer` als String von der Konsole gegeben. Es ruft dann die Methode `berechne()` auf, die Sie programmieren sollen. Sie dürfen folgende Methoden verwenden:

- `char aktuell()` ist das als nächstes zu verarbeitende Zeichen.
- `void weiter()` rückt um ein Zeichen weiter.
- `double getZahl()` liest eine Zahl und rückt um die gelesenen Zeichen weiter.
- `boolean oneOf(char c, String s)` stellt fest, ob das Zeichen `c` in `s` enthalten ist. (Damit kann man leicht feststellen, ob ein Zeichen ein Rechenzeichen ist.)

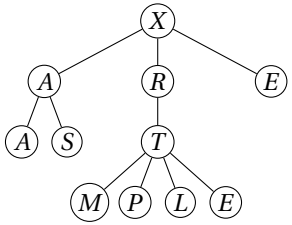
4 Bäume

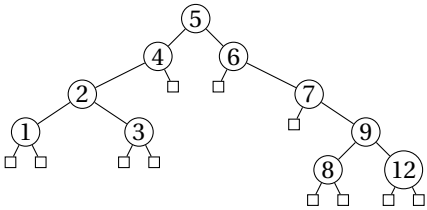
Graphen bestehen aus Knoten und Kanten; beide können Daten tragen. Bäume sind Graphen, bei denen alle Knoten zyklenfrei zusammenhängen und üblicherweise nur die Knoten Daten tragen.

Der folgende Abschnitt definiert die Begriffe Knoten, Kante, Pfad, Blatt, Endknoten, innerer/äußerer Knoten, Ebene, Unterbaum, geordneter Baum, Höhe, Pfadlänge, n-ärer Baum, binärer Baum, voller binärer Baum, vollständiger binärer Baum, Suchbaum.

4.1 Begriffe

Ein *Knoten* ist ein einfaches Objekt, das eine Information trägt und Verknüpfungen zu weiteren Knoten bietet, die wir *Söhne* oder *Nachfolger* nennen. Der Knoten, von dem die Verknüpfungen ausgehen, heißt *Vater* oder *Vorgänger*. Der Knoten, der keinen Vater hat, heißt *Wurzel*. Ein Knoten ohne Söhne heißt *äußerer* Knoten oder *Blatt* oder *Endknoten*, alle anderen heißen *innere* Knoten. Die bildliche Darstellung einer Verknüpfung nennt man *Kante* (seltener *Ast*). Ein *Pfad* ist eine Liste aufeinander folgender (durch Kanten verbundener) Knoten.





Bäume zeichnet man gewöhnlich nach unten wachsend, also mit der Wurzel oben. Liegt der Knoten k auf dem Pfad von Knoten z zur Wurzel, so sagt man auch, k liegt *über* z – im ersten Bild liegt also R über M. Jeder Knoten ist die Wurzel eines *Unterbaums*, welcher aus ihm und den Knoten darunter besteht. Im linken Bild gibt es sieben Unterbäume mit einem Knoten, einen Unterbaum mit drei Knoten, einen mit fünf Knoten und einen mit sechs Knoten.

Bei *geordneten* Bäumen ist die Reihenfolge der Nachfolger von Bedeutung, bei *ungeordneten* nicht. Knoten befinden sich auf *Ebenen*. Die Ebene eines Knotens ist die Anzahl der Knoten auf dem Pfad von der Wurzel zu ihm (die Wurzel nicht mitgezählt). Im linken Bild liegt also X auf Ebene 0 und S auf Ebene 2. Die *Höhe* eines Baums ist die größte Ebene, die vorkommt. Der linke Baum hat also die Höhe 3. Die *Pfadlänge* eines Baumes ist die Summe der Pfadlängen aller Knoten zur Wurzel. Der linke Baum im Bild hat also die Pfadlänge 21.

Falls in einem Baum jeder Knoten eine bestimmte Anzahl n von Nachfolgern haben muss, die in einer bestimmten Reihenfolge erscheinen, so spricht man von einem n-ären Baum. Der einfachste und am weitesten verbreitete n-äre Baum ist der binäre Baum, wie ihn das zweite Bild zeigt. (Die kleinen Rechtecke stehen für die `null`-Einträge der Blätter.)

Bei der Implementierung ist es zweckmäßig, einen Kopf-Knoten zu definieren, der keine eigene Information trägt, aber dazu dient, dass jeder tatsächliche Knoten einen Vorfahren hat. Diesen Kopfknoten zählen wir natürlich nirgends mit, er ist nur ein technisches Detail (ein Pseudoknoten).

Ein *voller* binärer Baum ist ein binärer Baum, in dem jede Ebene außer eventuell der letzten maximal gefüllt ist. Der rechte Baum im Bild ist nicht voll, weil z. B. die zweite Ebene zwischen 2 und 7 nicht gefüllt ist. Er ist aber ein *Suchbaum*, weil er Daten speichert, die sich der Größe nach unterscheiden lassen und weil das Datum jedes Knotens größer ist als alle Daten seines linken Teilbaums und kleiner als alle Daten des rechten Teilbaums.

Folgende rekursiv gefasste Definitionen sind besonders einfach: Ein Baum ist entweder ein einzelner Knoten oder ein als Wurzel dienender Knoten, der mit einer Menge von Bäumen verbunden ist. Ein binärer Baum ist entweder ein äußerer Knoten oder ein als Wurzel dienender Knoten, der mit einem linken binären Baum und einem rechten binären Baum verbunden ist. Bei der Implementierung sollte man sich wieder überlegen, ob ein Baum obiges *ist* oder *hat*.

4.2 Implementierung eines Suchbaumes

Bei Bäumen gibt es keine beste Musterimplementierung, die als Vorfahr für viele andere Anwendungsfälle dienen kann. Fast immer hat man ein besonderes Problem vor Augen, das mit einer ganz speziellen Implementierung am vorteilhaftesten behandelt werden kann. Deshalb sehen wir uns einen solchen speziellen Fall an, erfreuen uns an den Tricks und reagieren ggf. flexibel.

Ein binärer Suchbaum ist ein binärer Baum, bei dem für alle Knoten gilt, dass alle Werte im linken Teilbaum kleiner sind und alle im rechten größer. Das Einfügen neuer Knoten ergibt sich daraus ganz von selbst. Das Löschen eines Knotens hingegen will wohlüberlegt sein. Wir unterscheiden die drei Fälle:

1. Ein Knoten ohne Nachfolger (Blatt) wird einfach abgehängt.
2. Ein Knoten mit nur einem linken oder nur einem rechten Nachfolger wird dadurch entfernt, dass der Nachfolger statt des Knotens selber an den Vorgänger gehängt wird. Der Knoten wird sozusagen *entnommen*. Beachten Sie, dass dadurch evtl. ein dranhängender Teilbaum zwar hochrutscht, der Baum dadurch aber seine Ordnung nicht verliert.
3. Ein Knoten Z mit zwei Nachfolgern ist ein Problem. Man sucht erst einmal

den Knoten G auf der rechten Seite von Z, der am weitesten links liegt[1]. G hat höchstens einen Nachfolger und kann deshalb viel leichter entfernt werden als Z. Wir entfernen also G und legen seinen Inhalt in Z ab, ohne Z zu entfernen! Damit ist der Wert von Z weg, wie gewünscht und der Wert des gelöschten G ist ein guter Ersatz: Sein Wert ist größer als alle Werte links von Z und kleiner als alle seine nun rechts von ihm hängenden Nachfolger.

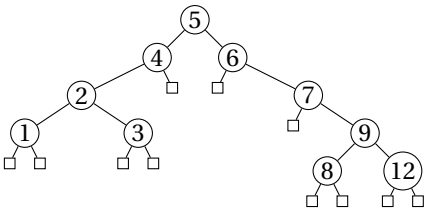
```
public class Knoten{
  int datum;
  Knoten lson;//linker  Nachfolger
  Knoten rson;//rechter Nachfolger

  public Knoten(int wert){ datum=wert; }
}

public class SuchBaum{
  protected final Knoten hohlkopf;

  public SuchBaum(){
    hohlkopf=new Knoten(Integer.MIN_VALUE);
  }

  protected Knoten vaterVon(int wert){
    Knoten vater=hohlkopf;
    Knoten lauf=hohlkopf.rson;
    while(lauf!=null && lauf.datum!=wert){
      vater=lauf;
      if(wert<lauf.datum) lauf=lauf.lson;
      else lauf=lauf.rson;
    }
    return vater;
  }

  public boolean insert(int wert){
    Knoten vater=vaterVon(wert);
    if(wert<vater.datum){
      if(vater.lson!=null) return false;
      vater.lson=new Knoten(wert);
    } else{
      if(vater.rson!=null) return false;
      vater.rson=new Knoten(wert);
    }
    return true;
  }
```

[1] Man könnte auch den rechtesten Knoten auf der linken Seite von Z suchen.

```
public boolean contains(int wert){
  Knoten vater=vaterVon(wert);
  Knoten sohn=wert<vater.datum?vater.lson:vater.rson;
  return sohn!=null && sohn.datum==wert;
}

public boolean delete(int z){
  Knoten vater=vaterVon(z);
  Knoten sohn=z<vater.datum?vater.lson:vater.rson;
  if(sohn==null) return false;// nichts zu löschen da
  if(sohn.lson!=null && sohn.rson!=null){
    Knoten bleib=vater=sohn;
    sohn=sohn.rson;// ein Schritt nach rechts...
    while(sohn.lson!=null){//... dann möglichst nach links
      vater=sohn;
      sohn=sohn.lson;
    }
    bleib.datum=z=sohn.datum;
  }
  if(sohn.lson==null) sohn=sohn.rson;
  else sohn=sohn.lson;
  if(z<vater.datum) vater.lson=sohn;
  else vater.rson=sohn;
  return true;
}

public String toString(){
  StringBuffer sb=new StringBuffer();
  alleWerteAb(hohlkopf.rson, sb);
  return sb.toString();
}

protected void alleWerteAb(Knoten ab, StringBuffer sb){
  if(ab!=null){
    alleWerteAb(ab.lson, sb);
    sb.append(ab.datum); sb.append(" ");
    alleWerteAb(ab.rson, sb);
  }
}

public void preorderAusgabe(Knoten w){
  if(w==null) return;
  System.out.print(w.datum+" ");
  preorderAusgabe(w.lson);
```

```
    preorderAusgabe(w.rson);
  }

  public void inorderAusgabe(Knoten w){
    if(w==null) return;
    inorderAusgabe(w.lson);
    System.out.print(w.datum+" ");
    inorderAusgabe(w.rson);
  }

  public void postorderAusgabe(Knoten w){
    if(w==null) return;
    postorderAusgabe(w.lson);
    postorderAusgabe(w.rson);
    System.out.print(w.datum+" ");
  }

  public void levelorderAusgabe(Knoten w){
    if(w==null) return;
    LinkedList<Knoten> queue=new LinkedList<Knoten>();
    queue.add(w);//hinten anfügen
    while(queue.size()>0){
      w=queue.poll();//vorderstes entnehmen
      System.out.print(w.datum+" ");
      if(w.lson!=null) queue.add(w.lson);
      if(w.rson!=null) queue.add(w.rson);
    }
  }
}
```

4.3 Traversierung



Von der *Traversierung* eines Baumes spricht man, wenn man alle Knoten genau einmal besucht (meist um irgendetwas mit ihnen anzustellen). Wir nehmen als Beispiel den vorher schon einmal erwähnten Suchbaum.

4.3.1 Preorder, Hauptreihenfolge

Da Bäume ziemlich unregelmäßige Gebilde sind, wenn sie nicht gerade voll sind, ist die Traversierung rekursiv wesentlich einfacher als iterativ. Bei der *Hauptreihenfolge* wird zuerst die Wurzel, dann der linke Sohn und dann der rechte Sohn besucht. Jeder Sohn kann natürlich wieder die Wurzel eines Unterbaums sein, aber das ist ja bei Rekursion gerade kein Problem!

Statt Hauptreihenfolge spricht man auch von *Preorder*-Traversierung, weil die Wurzel *vor* den Nachfolgern besucht wird. Studieren Sie den Algorithmus im obigen Quelltext und stellen Sie sicher, dass Sie einsehen, dass wirklich alle Knoten besucht werden!

Wie lautet die Ausgabe? `5 4 2 1 3 6 7 9 8 12`

Man könnte noch eine Zeile sparen, wenn man die `if`-Abfrage zweimal stellt. Wo? Und welche Folge hat das auf die rekursiven Aufrufe?

4.3.2 Inorder

Bei dieser Traversierung wird, wie der Name schon andeutet, zuerst der linke Sohn, dann, zwischen linkem und rechtem Sohn drin, die Wurzel und schließlich der rechte Sohn besucht.

Wie lautet die Ausgabe? `1 2 3 4 5 6 7 8 9 12`

4.3.3 Postorder

Bei dieser Traversierung wird, wie der Name schon andeutet, zuerst der linke Sohn, dann der rechte, und am Schluss die Wurzel besucht.

Wie lautet die Ausgabe? `1 3 2 4 8 12 9 7 6 5`

4.3.4 Level Order

Bei dieser Traversierung wird der Baum zeilenweise ausgegeben. Sie ist nicht mit einem rekursiven Algorithmus herstellbar.

Wie lautet die Ausgabe? `5 4 6 2 7 1 3 9 8 12`

4.4 Aufgaben

Mit einem binären Baum kann man eine Datenstruktur bauen, die alle Funktionen eines Arrays anbietet, manche davon aber deutlich schneller als das Array. Zu diesem

Zweck müssen die Knoten aber nicht nur das Datum, den linken und rechten Nachfolger abspeichern, sondern auch noch jeweils die Anzahl ℓ der Knoten im linken Teilbaum (oder im rechten; eines von beiden reicht).

Zum Einfügen eines Elementes an der Position i hangelt man sich von der Wurzel aus in den Baum und zwar nach links, wenn $i \leq \ell$, andernfalls nach rechts. Bei einem Schritt nach links zählt man das ℓ des Knotens hoch, von dem man gekommen ist (weil der ja damit einen Knoten mehr in seinem linken Teilbaum hat).

Bei einem Schritt nach rechts macht man i um $\ell + 1$ kleiner. Der ganze Vorgang wiederholt sich so lange, bis irgendwo das Ende des Baums erreicht wird und der neue Knoten dort Blatt wird. `get(i)` geht auf die gleiche Weise vor und `delete(i)` arbeitet wie beim Suchbaum.

1. Implementieren Sie die drei Methoden.
2. In welchen Situationen ist diese Datenstruktur schneller als ein Array, wann ist sie langsamer?
3. Geben Sie eine Folge von Einfügungen an, bei der sich die Datenstruktur ungeschickt anstellt.

5 Effizienz

Die Effizienz eines Algorithmus a ist umso höher, je schneller er sein Problem löst und je weniger Speicherplatz er dafür braucht. Die meisten Algorithmen können Probleme unterschiedlicher Größe n verarbeiten. So kann n z. B. bei einem Sortieralgorithmus die Anzahl der zu sortierenden Elemente sein, bei einem Algorithmus, der eine Zahl in Primfaktoren zerlegt, würde man für n die Anzahl der Ziffern der zu zerlegenden Zahl nehmen. Wir bezeichnen mit $t_a(n)$ den Zeitaufwand des Algorihmus a für sein Problem der Größe n und mit $s_a(n)$ seinen Speicherplatzverbrauch. Wir werden uns hauptsächlich für t interessieren, s wird nur am Rande betrachtet.

5.1 Beispiel: Abschnittssumme

An Hand eines ausführlichen Beispiels soll gezeigt werden, wie man das Zeitverhalten eines Algorithmus untersucht. Die Ergebnisse werden genauer sein als üblicherweise nötig, aber zum einen lernt man ein paar Tricks dabei und zum anderen sieht man klar, warum es später ausreicht, solche Untersuchungen weniger genau durchzuführen.

Wir suchen einen Algorithmus, der aus einem Array von ganzen Zahlen einen zusammen hängenden Bereich heraus greift und aufsummiert und zwar so, dass eine möglichst große Summe herauskommt. Sind alle Zahlen negativ, hat der Algorithmus 0 als Ergebnis, weil dann ein Abschnitt mit null Elementen den Maximalwert 0 ergibt.

Die erste Lösung, die einem einfällt ist sehr elementar. Die Zahl 3 im Namen zeigt an, dass sie nur die drittbeste Lösung des Problems ist. Wir werden den Algorithmus später verbessern.

```
public static int maxsumme3(int[] z){
  int s, max=0, n=z.length;
  for(int i=0; i<n; i++)
    for(int j=i; j<n; j++){
      s=0;
      for(int k=i; k<=j; k++)
        s+=z[k];  // Befehl B
      if(s>max) max=s;
    }
  return max;
}
```

Es handelt sich um drei ineinander verschachtelte Schleifen deren innerste die Zahlen in s aufsummiert. Ist der erreichte Wert größer als das bisherige Maximum,

so merkt sich `max` die neue Summe. Das ganze wird durchgeführt für alle möglichen linken Grenzen `i` und rechten Grenzen `j`.

Wir wollen nun berechnen, wie oft diese Methode den Befehl B `s+=z[k]` aufruft. Da `k` alle Werte von `i` bis `j` annimmt, wird B für gegebenes `i`,`j` genau $j-i+1$-mal ausgeführt. Dies geschieht aber mehrmals, weil ja `i` und `j` verschiedene Werte annehmen und die `k`-Schleife jedesmal durchläuft. Schauen wir uns also zuerst an, welche Werte `j` annimmt: Es läuft von i bis $n-1$. Am Anfang ($j=i$) bearbeitet die `k`-Schleife also $j-i+1=1$ mal B, dann 2 mal, dann 3 mal usw. bis $j-i+1=(n-1)-i+1=n-i$ mal. Wie oft ist das insgesamt? Es sind insgesamt $\frac{1}{2}(n-i)(n-i+1)$ mal B, wie schon der Grundschüler Carl-Friedrich Gauß wusste[1].

Leider sind wir damit noch nicht fertig! All das passiert ja mehrfach – je einmal für jeden Wert von `i`. Das erste Mal für $i=0$ bekommen wir laut Gauß $\frac{1}{2}n(n+1)$ mal B, dann für $i=1$ noch $\frac{1}{2}(n-1)n$ mal B usw. bis $\frac{1}{2}\cdot 2\cdot 3$ mal und $\frac{1}{2}\cdot 1\cdot 2$ mal. Wie oft ist nun das insgesamt? Auch dafür gibt es eine trickreiche Überlegung, die aber etwas anspruchsvoller ist. Das Ergebnis ist $\frac{1}{6}(n^3+3n^2+2n)$.

Solche Überlegungen können offensichtlich recht schwierig werden! Aber leider sind sie noch nicht ganz genau. Schließlich wissen wir jetzt erst, wie oft B abgearbeitet wird. Das Programm besteht aber aus noch weiteren Befehlen, die nicht alle in der innersten Schleife stehen und folglich weniger oft ausgeführt werden. Immerhin erkennen wir, dass der am häufigsten ausgeführte Befehl B bei wachsendem n nicht n-mal häufiger ausgeführt wird, sondern sogar mehr als n^3-mal häufiger. Weil uns eine solch grobe Einsicht ausreicht und sie viel leichter zu bekommen ist als eine genaue Formel, erlauben wir uns einige Vereinfachungen.

5.2 Vereinfachungen

- Da ein Algorithmus auf verschiedenen Rechnern verschieden lang braucht, interessieren wir uns nicht für die physikalische Zeit (gemessen in Sekunden), sondern die logische. Wir tun dabei so, als wenn *ein Befehl* immer *einen Schritt* dauern würde. Diese grobe Betrachtungsweise bewahrt uns davor, uns in praktischen Feinheiten des benutzten Systems zu verstricken, wenn wir uns nur für den *Geist* interessieren, der im Algorithmus steckt[2].
- Wir betrachten nicht alle Befehle des Algorithmus, sondern den *dominanten*, der am häufigsten ausgeführt wird.

[1] Gut, Gauß war genial, aber seine Idee sollte jeder verstehen: Um die Summe der Zahlen 1 bis 100 zu berechnen, schrieb er die Summe zweimal untereinander aber in umgekehrter Reihenfolge.

$$\begin{array}{rcl} 1 & + & 2 + \ldots + 99 + 100 \\ 100 + 99 & + & \ldots + 2 + 1 \end{array}$$

Das kann man jetzt als $100\cdot(100+1)$ schnell berechnen. Die Hälfte davon ist dann die gesuchte Summe. In unserer Rechnung ist die Obergrenze aber nicht 100, sondern $n-i$, daher die Formel.

[2] Für die praktische Programmiertätigkeit hingegen kann es manchmal hilfreich sein, die technischen Feinheiten experimentell zu messen durch den Aufruf von Zeitfunktionen.

- Wir betrachten nicht alle möglichen Fälle, die bei einer bestimmten Problemgröße auftreten können, sondern nur den ungünstigsten, den durchschnittlichen und den günstigsten.

  (Im obigen Beispiel gibt es keine unterschiedlichen Fälle. Das Programm behandelt alle Arrays einer bestimmten Größe gleich. Aber das ist eher unüblich!)

5.3 Effizienzklassen

Das Zeitverhalten eines Algorithmus gilt in der Regel als ausreichend genau untersucht, wenn man ihn einer Klasse zugeordnet hat. Man bemüht sich also nicht um die exakte Formel, sondern begnügt sich mit Aussagen darüber, wie der Zeitaufwand ungefähr steigt, wenn die Problemgröße anwächst. In unserem einführenden Beispiel haben wir festgestellt, dass der Zeitaufwand nicht stärker steigt als die dritte Potenz der Problemgröße. Man sagt, das Zeitverhalten von `maxsumme3` gehört zur Klasse $\mathcal{O}(n^3)$. Diese Klasse ist die Menge aller Funktionen, die nicht stärker als die Funktion $g(n) = n^3$ steigen. Wir sollten uns aber wenigstens einmal genau überlegen, was es bedeutet *nicht stärker als eine Funktion zu steigen.*

Die Menge aller Funktionen, die nicht stärker steigen als $g(n)$ wird definiert als

$$\mathcal{O}\big(g(n)\big) = \big\{f : \mathbb{N} \to \mathbb{R}^+ | \exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R}^+, \forall n > n_0 : f(n) \leq c \cdot g(n)\big\}$$

Damit kann man dann schreiben $t_{\text{maxsumme3}}(n) \in \mathcal{O}(n^3)$. Manchmal liest man auch die schlampige Schreibweise $t_{\text{maxsumme3}}(n) = \mathcal{O}(n^3)$, die mathematisch offensichtlich falsch ist, in menschlichen Ohren aber plausibel klingen mag.

Die meisten Algorithmen haben ein Zeitverhalten aus folgenden Klassen:

- $\mathcal{O}(1)$, wenn $t(n)$ unabhängig von n ist. Ein solcher Algorithmus läuft in *konstanter Zeit* ab und ist meist ziemlich uninteressant, weil er etwas so simples tut, dass alle Probleme gleich groß erscheinen.
- $\mathcal{O}(\log n)$, wenn $t(n)$ proportional zu $\log n$ ist (beliebige Basis!). Ein solcher Algorithmus läuft in *logarithmischer Zeit* ab und tritt meist dann auf, wenn wenige Elemente in einem Teile-und-herrsche-Verfahren behandelt werden. Bei wachsendem n wird der Algorithmus allmählich langsamer. Die Laufzeit verdoppelt sich aber erst, wenn sich die Problemgröße quadriert hat.
- $\mathcal{O}(n)$, wenn $t(n)$ linear ansteigt. Ein solcher Algorithmus läuft in linearer Zeit ab und tritt auf, wenn jedes zu bearbeitende Element eine bestimmte Bearbeitungszeit braucht. Ein Algorithmus, der alle Elemente einer Menge bearbeiten muss, kann nicht in einer schnelleren Klasse sein als dieser.
- $\mathcal{O}(n \log n)$ wenn ein Problem in kleinere Teilprobleme aufgeteilt wird, diese unabhängig voneinander gelöst und dann die Lösungen kombiniert werden. Bei doppelt so großem n wird die Laufzeit mehr als doppelt so groß, aber nicht wesentlich mehr. Sedgewick hat dafür das Adjektiv *linearithmisch* erfunden.

- $\mathcal{O}(n^2)$ wenn die Laufzeit *quadratisch* ist. Solche Algorithmen kann man nur für relativ kleine Probleme anwenden. Quadratische Laufzeiten sind typisch für Algorithmen, die alle paarweisen Kombinationen von Datenelementen verarbeiten (etwa in einer verschachtelten Schleife).
- $\mathcal{O}(n^e)$ mit $e \geq 2$. Hier spricht man allgemein von *polynomialem Laufzeitverhalten*. Wenn man $e = 2$ als unangenehm langsam bezeichnet, so kann man bei $e > 2$ von unerträglich langsam sprechen. Unser einführendes Beispiel ist aus dieser Klasse ($e = 3$) und braucht für 1000 Elemente schon 30 Sekunden.
- $\mathcal{O}(2^n)$ bei *exponentieller Laufzeit*. Diese tritt auf bei Algorithmen, die sich verzweigende Probleme gewaltsam lösen (backtracking). Schon bei kleinem n ist die Laufzeit groß oder sehr groß. Ein Problem von $n = 50$ kann dann schon Stunden dauern.

5.4 Verbesserung des Beispiels

Selten findet man sich in der glücklichen Lage, einen langsamen Algorithmus verbessern zu können. Das einführende Beispiel ist ein solcher. Die erste Verbesserung ist auch leicht zu finden: Bisher hat die innerste Schleife jedesmal die vollständige Summation von `i` bis `j` durchgeführt, wenn `j` um 1 größer wurde. Das muss nicht sein! Man addiere einfach zum bisherigen Wert von `s` das neu hinzu kommende Element und schon hat man die neue Summe. Damit spart man sich die dritte Schleife und unser Algorithmus verbessert sich von $\mathcal{O}(n^3)$ auf nicht schlechter als $\mathcal{O}(n^2)$.

```
public static int maxsumme2(int[] z){
  int s, max=0, n=z.length;
  for(int i=0; i<n; i++){
    s=0;
    for(int j=i; j<n; j++){
      s+=z[j];
      if(s>max) max=s;
    }
  }
  return max;
}
```

Aber das ist noch nicht das letzte Wort. Mit ein bisschen Zusatzanstrengung geht es tatsächlich noch um einen Grad besser und damit fängt der Algorithmus an, interessant zu werden. Der zum Ziel führende Gedankengang geht so:

Wir betrachten das Array als eine Serie von nacheinander eintreffenden Werten. Wir merken uns das Maximum `max` der bisherigen Summen (anfangs 0) und den größten Wert `endmax`, den eine Summe haben kann, die ganz hinten beginnt (anfangs auch 0). Wenn nun ein neues Element hinzu kommt, wird `endmax` neu berechnet (es

wird nie kleiner als 0). Ist endmax nun größer geworden als max, so übernimmt max den neuen Wert, andernfalls behält es den alten.

Nehmen wir als Beispiel die Zahlenfolge 3, 1, −3, 1, 2, −8, 5. max fängt an mit 0, endmax ebenso. Dann bekommt endmax nacheinander die Werte 3 (→max), 4 (→max), 1, 2, 4, 0, 5 (→max). Am Ende hat max den größten je erreichten Wert 5.

```
public static int maxsumme1(int[] z){
  int max=0, endmax=0, n=z.length;
  for(int i=0; i<n; i++){
    endmax+=z[i];
    if(endmax<0) endmax=0;
    if(endmax>max) max=endmax;
  }
  return max;
}
```

5.5 Beispiele unterschiedlicher Komplexität

5.5.1 Schnelle Potenz

Für manche Anwendungen der Kryptologie müssen hohe Potenzen berechnet werden. Der naive Ansatz $b^n = b^{n-1} \cdot b$ lässt sich leicht in einer Schleife realisieren, hat damit aber lineare Zeitkomplexität. Legendre hat 1798 ein besseres Verfahren vorgestellt, das den Zusammenhang $b^n = \left(b^{\frac{n}{2}}\right)^2$ ausnutzt.

Der Wert von b^8 wird rekursiv vereinfacht zu $\left(b^4\right)^2$, dann $\left(\left(b^2\right)^2\right)^2$ und schließlich zu $\left(\left(\left(b^1\right)^2\right)^2\right)^2$. Es wird also dreimal quadriert statt acht Mal multipliziert. Aber einmal quadrieren ist eben einmal multiplizieren. Bei ungünstigeren Exponenten können doppelt so viele Multiplikationen nötig sein wie bei günstigen, aber schlimmer wird es sicher nicht mehr. Z. B. b^7 lässt sich nur vereinfachen zu $\left(b^3\right)^2 \cdot b = \left(\left(b^1\right)^2 \cdot b\right)^2 \cdot b$.

Die folgende Methode implementiert diesen Algorithmus. Machen Sie sich bitte klar, dass der Zeitaufwand $\mathcal{O}(\log n)$ beträgt.

```
public static double potenz(double b, int n){
  if(n==0) return 1;
  if(n%2==1) return b*potenz(b, n-1);
  b=potenz(b, n/2);
  return b*b;
}
```

5.5.2 Rucksack

Gegeben ist ein Rucksack, der mit einem maximalen Gewicht (`int r`) gefüllt werden darf und eine Menge unterschiedlich schwerer Goldklumpen (`int[] g`). Es soll möglichst viel Gold mitgenommen werden aber nicht zu viel, weil sonst der Rucksack reißt. Gesucht ist eine Methode, die `r` und `g` bekommt und als Ergebnis zurück gibt, ob der Rucksack mit einem Teil der vorliegenden Goldklumpen bis zur Berstgrenze `r` gefüllt werden kann. Hat man $r = 10$ und $g = \{5,4,3,2\}$, so ist das Problem lösbar. Als Ergebnis reicht ein `boolean`, für praktische Zwecke halten wir aber außerhalb der Methode ein `boolean[] nimm` bereit, das speichert, welche Elemente von `g` gewählt wurden, im genannten Beispiel also $nimm = \{1,0,1,1\}$.

Dieses Problem ist berühmt dafür, dass es keine elegante Lösung gibt. Man muss im wesentlichen alle Kombinationsmöglichkeiten ausprobieren. Wir streben ein Programm an, das für beliebig vorgegebenes `nimm` die nächste Lösung findet. Das geht rekursiv und iterativ.

Die rekursive Idee ist folgende: Die Methode ändert das Bit des sie betreffenden Goldstücks. Wenn es also vorher gewählt war, wird es diesmal nicht gewählt und umgekehrt. Dann ruft sie sich rekursiv selber für das nächste Goldstück auf und, falls das zu keiner Lösung führt, ein zweites Mal. Die Methode ist also so ausgelegt, dass sie zweimal aufgerufen wird, um beide Möglichkeiten für ihr Goldstück ausprobieren zu können.

Beim iterativen Weg erzeugen wir durch „Hochzählen“ des `boolean[] nimm` alle möglichen Kombinationen und Rechnen die dabei entstehenden Summen aus.

```
public static boolean kombi_iter(int voll, int[] gold){
  int i=-1;
  while(++i<nimm.length)
    if(nimm[i]^=true)
      if(summe(gold, nimm)==voll) return true;
      else i=-1;
  return false;//Füllung unmöglich
}
```

In beiden Fällen haben wir die Methode `summe` gebraucht:

Da zur Lösung der Aufgabe alle Kombinationen ausprobiert werden müssen, ist dieses Problem aus der Klasse $\mathcal{O}(2^n)$, d. h. es erfordert exponentiellen Zeitaufwand.

5.6 Aufgaben

1. Verständnis der $\mathcal{O}$-Mengen:
   Zeigen Sie, dass die Funktionen $t_1(n) = 300+5n$ und $t_2(n) = 300+5n^2$ enthalten sind in der Menge $\mathcal{O}(n^2)$, indem Sie jeweils die beiden Konstanten n_0 und c berechnen.

2. Gefühlter Zeitverbrauch:

   Legen Sie eine Versuchsreihe an, in der Sie der `maxSumme`-Methode verschieden große Arrays geben und die Zeit messen, die sie dann braucht. Damit Sie nicht jedesmal riesige Mengen von Zahlen auf der Kommandozeile angeben müssen, lesen Sie die Werte bitte mit `LineInput.readInt()` ein und leiten Sie eine vorbereitete Datei um. Das geht so:

   ```
   java MaxSumme < werte.txt
   ```

   Bereiten Sie hierzu die Datei `werte.txt` vor mit 10000 Werten zwischen −500 und 500. Sie können also den Zeitverbrauch einmal mit 2000 Werten messen und danach nochmal mit 4000 Werten. Dann mit 6000 Werten usw. Überprüfen Sie, ob der Zeitanstieg ungefähr mit der dritten Potenz wächst. Zur Zeitmessung können Sie die Methode `System.nanoTime()` verwenden. Speichern Sie den Wert vor dem zu messenden Abschnitt und danach. Der Unterschied der beiden Zahlen sollte recht genau die verbrauchte Zeit wiedergeben.

3. Wir wissen nicht, wie lang *ein Befehl* dauert. Ein `System.out.println(...)` dauert eventuell hundert mal so lang wie ein `x+=5`. Befindet sich letzerer in einer Schleife, die entweder 10 oder 20 mal durchlaufen wird und nachher 1 mal `System.out.println` aufruft, so unterscheiden sich die Laufzeiten nicht sehr, weil 10 oder 20 im Vergleich zu 100 nur wenig auffallen. Entweder nimmt man also deutlich größere Unterschiede in der Problemgröße (2000, 4000, ... statt 10, 20, ...) oder man misst nicht die wirkliche Zeit, sondern eine selber gemachte. Dazu könnten Sie einen Zähler einbauen, der einfach mitzählt, wie oft eine bestimmte Stelle in der Methode erreicht wird. Tun Sie dies in der vorigen Aufgabe. Erhalten Sie ähnliche Ergebnisse?

4. Ein Algorithmus hat für ein Problem der Größe $n = 1000$ die Zeit 0,02 Sekunden gebraucht. Wie lang braucht er für $n = 10000$, wenn er zu folgender Klasse gehört:

   a) $\mathcal{O}(n)$ b) $\mathcal{O}(\log n)$ c) $\mathcal{O}(n \log n)$ d) $\mathcal{O}(n^2)$ e) $\mathcal{O}(2^n)$

   Geben Sie kurz an, wie Sie gerechnet haben.

5. Schreiben Sie eine Methode `search`, die für ein vorliegendes Array von ganzen Zahlen entscheidet, ob eine bestimmte Zahl darin enthalten ist. Welcher Effizientzklasse gehört die Methode (im besten Fall) an?

   ```
   public static boolean search(int[] z, int s){
     for(int i=0; i<z.length; ++i)
       if(s==z[i]) return true;
     return false;
   }
   ```

   Gehen Sie nun davon aus, dass das Array aufsteigend sortiert ist. Damit können Sie eine deutlich verbesserte Methode `binarysearch` schreiben, die den zu durchsuchenden Bereich bei jedem Schritt halbiert. Welche Effizienzklasse liegt nun vor?

   ```
   public static boolean binarysearch(int[] z, int s){
   ```

```
    int li=0, re=z.length-1, m=0;
    while(li!=re){
      m=(li+re)/2;
      if(z[m]<s) li=m+1;
      else re=m;
    }
    return s==z[m];
  }
}
```

6. Auf einem quadratischen Schachbrett mit $n \times n$ Feldern sollen n Damen so platziert werden, dass sie sich nicht gegenseitig bedrohen. Gesucht ist ein Programm, das alle möglichen Feldbesetzungen findet und ausgibt. Bei dieser Aufgabe sollte man auf die Zeit achten, weil die Lösung sehr lang dauert!

   Hier ein paar hilfreiche Gedanken: In jeder Spalte kann nur eine Dame stehen, also reicht es, wenn die Ausgabe einer Besetzung n Zahlen listet, die angeben, in welcher Zeile (wie hoch) die Damen stehen. Wir können also die Positionen in einem `int[]` der Länge n erfassen. Um alle Lösungen zu finden, setzen wir die i-te Dame auf jede Zeile einmal und testen, ob sie von den vorherigen bedroht wird. Wenn ja, setzen wir sie gleich um eine Position höher. Wenn nein, setzen wir (rekursiv) die nächste mit der gleichen Strategie. (Außer bei der letzten! Wenn die einen unbedrohten Platz gefunden hat, geben wir die ganze Besetzung aus.)

```
public class Damen{
  public static int anz=0;

  public static void main(String[] args){
    int n=Integer.parseInt(args[0]);
    int[] hoch=new int[n];
    setze(hoch, 0);
    System.out.println("Das waren "+anz+" Möglichkeiten.");
  }

  public static void setze(int[] hoch, int d){
    for(int h=0; h<hoch.length; ++h){
      hoch[d]=h;
      if(gehtdas(hoch, d))
        if(d==hoch.length-1){// Gefunden! -> Ausgabe
          for(int i=0; i<hoch.length; ++i)
            System.out.print(1+hoch[i]+" ");
          System.out.println();
          anz++;
        } else setze(hoch, d+1);
    }
  }
```

```
  public static boolean gehtdas(int[] hoch, int d){
    for(int i=0; i<d; ++i){//Bedrohungen bei...
      if(hoch[i]==hoch[d])     return false;//  0 Grad
      if(hoch[i]+d-i==hoch[d]) return false;//+45 Grad
      if(hoch[i]-d+i==hoch[d]) return false;//-45 Grad
    }
    return true;
  }
}
```

6 Datenbanken

Die Theorie zu Datenbanken lernt man im Datenbanken-Buch. Hier geht es um den praktischen Einsatz von SQL mit einem MySQL-Server. Das Referenzhandbuch vom Hersteller findet man bei `http://dev.mysql.com/doc/`.

6.1 Installation und Inbetriebnahme

MySQL ist ein recht professioneller Datenbankserver und hat deshalb eine ausgefeilte Benutzerverwaltung. Nach der Installation gibt es bereits eine Datenbank namens `mysql`, die diese Benutzerverwaltung beinhaltet. Bei manchen Installationen sind die Rechte eher eingeschränkt, so dass der Anfänger leicht den Eindruck gewinnt, nicht reinzukommen.

Unter Windows verwendet man XAMPP, das die Rechte sehr großzügig vergibt. Unter Linux ruft das Installationsskript `mysql_install_db` auf, wodurch einige Benutzer angelegt werden. Der Zugriff auf die Dienste des Datenbankservers erfolgt im primitivsten Fall mit dem Clientprogramm `mysql` (der Serverprozess heißt im Gegensatz dazu meist `mysqld`).

Nach der Installation schaut man sich am besten erst mal die Benutzer an, die von Anfang an vorhanden sind. Als `root` hat man vorerst ohne Passwort Zugang. Deshalb fehlt im folgenden Befehl die Option `-p`.

```
mysql -u root
mysql> SELECT host, user FROM mysql.user;
```

Als nächstes richten wir die Datenbank `schuledb` ein und gewähren dem Benutzer `schule` mit Passwort `schulepass` vollen Zugriff auf alle (noch nicht vorhandenen) Tabellen zum Üben. Dieser loggt sich ein, legt eine Tabelle an, trägt zwei Zeilen ein und fragt diese gleich wieder ab.

```
mysql -u root
mysql> CREATE DATABASE schuledb;
mysql> GRANT ALL ON schuledb.* TO schule IDENTIFIED BY 'schulepass';
mysql> exit

mysql -u schule -p schuledb
password: schulepass
mysql> CREATE TABLE leute(id INT AUTO_INCREMENT PRIMARY KEY,
    -> name VARCHAR(30), vorname VARCHAR(30));
mysql> INSERT INTO leute(vorname,name) VALUES('Pop','Eye'),
```

```
    -> ('Olivia','Öl');
mysql> SELECT vorname FROM leute;
```

Man hätte das Passwort schulepass auch direkt (ohne Leerzeichen) hinter -p angeben können. Da wir das nicht getan haben, werden wir in der nächsten Zeile danach gefragt. Man sieht, dass Befehle so lange als nicht abgeschlossen betrachtet werden, bis das Semikolon folgt. Möchte man die Eingabe eines Befehls abbrechen, so kann man an beliebiger Stelle (außer innerhalb Strings) die Zeichenfolge \c angeben und mit Enter abschließen.

Eine vorhandene Datenbank kann man als Textdatei exportieren mit

```
mysqldump -u schule -p schuledb > schuledb.txt
```

Diese Textdatei kann man aber auch wieder in eine leere (schon existierende!) Datenbank einlesen

```
mysql -u schule -p schuledb < schuledb.txt
```

Einen vollständigen Export bekommt man mit

```
mysqldump -u root -p -A --add-drop-database > backup.txt
```

Eingelesen wird der Export mit

```
mysql -u root -p < backup.txt
```

An dieser Stelle sollte man kurz über die Codierung von Umlauten nachdenken, weil das sonst später ein ewiges Ärgernis wird. Standardmäßig arbeiten sowohl der MySQL-Server als auch das Clientprogramm mysql mit ISO-8859-1, kurz latin1. Meist wird man aber das modernere utf8 wollen. Dazu muss man sowohl bei der Erzeugung einer Tabelle diesen Zeichensatz anfordern als auch bei der Verbindungsaufnahme den Client darauf einstellen:

```
mysql --default-character-set=utf8 -u root -p
mysql> CREATE TABLE leute(id INT AUTO_INCREMENT PRIMARY KEY,
    ->   name VARCHAR(30), vorname VARCHAR(30)
    -> ) DEFAULT CHARSET=utf8;
mysql> INSERT INTO leute(vorname,name)
    ->   VALUES('der Schreckliche','Hägar');
```

Will man mittels Java mit einem MySQL-Server kommunizieren, so braucht man den JDBC-Datenbanktreiber mysql-connector-java-xxx-bin.jar. Die jeweils aktuelle Version findet man unter www.mysql.com/products/connector/j/. Man legt ihn am besten in der Java-Installation im Verzeichnis lib/ext/ ab oder liefert ihn im Falle von Applets einfach jedesmal mit. Wie man den Zugriff herstellt, zeigt folgende Datei:

```
import java.sql.*;

public class MyLibrary{
```

```
  public static void main(String[] args){
    try{
      Connection conn;
      Statement stmt;
      ResultSet res;
      Class.forName("com.mysql.jdbc.Driver").newInstance();
      conn=DriverManager.getConnection(
        "jdbc:mysql://localhost/mylibrary", "gowler", "gow"
        );
      stmt=conn.createStatement();
      res=stmt.executeQuery(
        "select publID, publName from publishers order by publName"
        );
      while(res.next()){
        int id=res.getInt("publID");
        String name=res.getString("publName");
        System.out.println("ID: "+id+"  Name: "+name);
      }
      res.close();
    }
    catch (Exception e){
      System.out.println("Fehler: "+e.toString());
    }
  }
}
```

Auch mit OpenOffice kann man auf den MySQL-Datenbankserver zugreifen, und zwar auf dem gleichen Weg. Dazu nimmt man die oben genannte Connector-Jar-Datei in den Classpath von OpenOffice mit auf (OO verwendet nämlich auch Java). Das geht auf dem Weg Tools-Options-OpenOffice-Java.

Als nächstes muss dann noch ein OpenOffice-Datenbank Projekt darauf zugreifen. Das geht mit folgenden Schritten:

- *Connect to an existing Database* (Typ MySQL)
- *Connect using JDBC* (anstatt ODBC)
- *Name of the Database:* `test`
  Server URL: `localhost`
  Port Number: `3306`
  MySQL JDBC driver class: `com.mysql.jdbc.Driver`
  (diese Klasse ist Teil des Jar) *Test class* sollte Erfolg melden.
- *Username:* schule *Password required:* auswählen
- *Register the database for me* muss nicht sein.

OpenOffice legt nun eine Datei an, die den Zugriff herstellt.

6.2 Operatoren und Operanden

In MySQL sind Strings in einfachen und doppelten Anführungszeichen gleichbedeutend. Braucht man ein solches Anführungszeichen als Teil des Strings, so escaped man es am besten, z. B. `'geht\'s noch'`.

In SQL gibt es drei Arten von Kommentaren. Die wird man zwar bei der direkten Eingabe von Befehlen niemals verwenden, in Skripten mit einer Menge von Befehlen können Sie aber hilfreich sein.

```
SELECT 1 /* sowas braucht man selten */, 2
SELECT 3 -- Leerzeichen nach den Minussen ist nötig, sonst wie #
SELECT 4 # so geht es neuerdings auch
```

Wie viele andere schwach typisierten Sprachen bemüht sich das SQL von MySQL selbständig, bei Operationen mit zwei unterschiedlichen Datentypen einen gemeinsamen Nenner zu finden. Integerzahlen werden automatisch in Fließkommazahlen umgewandelt, wenn einer der Operanden eine Fließkommazahl ist. Zeichenketten werden in Berechnungen automatisch in Zahlen umgewandelt. (Wenn der Beginn einer Zeichenkette sich nicht als Zahl interpretieren lässt, rechnet MySQL mit 0.)

```
SELECT '3.5hallo' + 1;
 4.5
```

Logische und arithmetische Operatoren mit `NULL` als Operand ergeben immer `NULL`. Bei MySQL liefert auch die Division durch 0 das Ergebnis `NULL`.

Das Ergebnis von Vergleichsoperatoren ist in guten Programmiersprachen ein eigener Typ. In MySQL gibt es zwar die Werte `TRUE` und `FALSE`, aber sie werden als 1 bzw. 0 gespeichert, unterscheiden sich also nicht von ganzen Zahlen. Alle Vergleiche mit `NULL` liefern `NULL`. Nur bei `IS NULL` und `IS NOT NULL` bekommt man 0 oder 1 heraus. Beispiele:

```
SELECT NULL=NULL, NULL=0;
  NULL, NULL
SELECT NULL IS NULL, 0 IS NULL;
  1, 0
```

Beim Mustervergleich mit `LIKE` wird nicht zwischen Groß- und Kleinschreibung unterschieden. Man kann die beiden Platzhalterzeichen _ und % verwenden. Ersteres steht für genau ein beliebiges Zeichen, letzteres für keines oder beliebig viele beliebige Zeichen. So findet das Kommando
`SELECT einkommen FROM authors WHERE name LIKE 'M__%r'`
das Einkommen der Autoren namens Meir, Meier, Maurer, Mützenstricker, aber nicht Mar. Braucht man die beiden Platzhalter selber, so muss man sie escapen als `\_` und `\%`. Wesentlich leistungsfähiger ist `REGEXP`, das aber hier nicht erklärt wird.

Die Aggregatsfunktionen haben nur eine Zeile, wo man ohne sie viele hätte. Der folgende Befehl gibt das Durchschnittseinkommen aller meierigen Mitarbeiter an.

Tabelle wichtiger Operatoren und Funktionen

+, -, *, /	Grundrechenarten
%, MOD	Rest einer Division
|	arithmetisches Oder
&	arithmetisches Und
~	arithmetisches Not (invertieren aller Bits einer Zahl)
<<	verschiebt die Zahl bitweise um *n* Stellen nach links
>>	verschiebt die Zahl bitweise um *n* Stellen nach rechts
=	Test auf Gleichheit
!=, <>	Test auf Ungleichheit
>, <=, <, <=	Vergleichsoperatoren
IS NULL, IS NOT NULL	Vergleiche mit NULL
BETWEEN	Bereichsvergleich, z. B. x BETWEEN 1 AND 3
IN, NOT IN	Mengenvergleich, z. B. x NOT IN ('a','b','c')
LIKE, NOT LIKE	einfacher Mustervergleich, z. B. x LIKE 'Windows%'
REGEXP, NOT REGEXP	erweiterter Mustervergleich, z. B. x REGEXP '.*x$'
SOUNDS LIKE	Mustervergleich nach Klang
BINARY	z. B. BINARY 'a'='A' ergibt false
!, NOT	logische Verneinung
||, OR	logisches Oder
&&, AND	logisches Und
XOR	logisches Exklusiv-Oder
MAX(expr)	Findet den größten von mehreren Werten
MIN(expr)	Findet den kleinsten von mehreren Werten
SUM(expr)	Findet die Summe von mehreren Werten
AVG(expr)	Findet den Mittelwert von mehreren Werten
COUNT(expr)	Findet den Anzahl von mehreren Werten
COUNT(DISTINCT expr)	ebenso, zählt aber nur, verschiedene Werte

```
SELECT avg(einkommen) FROM authors WHERE name LIKE 'M__%r'
```

Mehr dazu im Abschnitt über den SELECT-Befehl.

6.3 Datentypen und Attribute

In der Tabelle der wichtigsten Datentypen haben die Werte *m* und *d* keine Auswirkungen auf die Genauigkeit der abgespeicherten Daten, sondern nur auf die Breite der Ausgabe von SELECT-Kommandos.

Bei der Definiton von Spalten mit CREATE TABLE oder ALTER TABLE können bei jeder Spalte verschiedene Optionen angegeben werden. Die folgende Tabelle fasst diese Optionen zusamen. Beachten Sie, dass nicht alle Optionen für alle Datentypen geeignet sind!

Tabelle der wichtigsten Datentypen

`TINYINT(m)`	8-bit-Integer
`SMALLINT(m)`	16-bit-Integer
`MEDIUMINT(m)`	24-bit-Integer
`INT(m)`, `INTEGER(m)`	32-bit-Integer
`FLOAT(m,d)`	32-bit-Fließkommazahl (ca. 8 Dezimalstellen genau)
`DOUBLE(m,d)`	64-bit-Fließkommazahl (ca. 16 Dezimalstellen genau)
`DECIMAL(p,s)`	p Stellen vor und s Stellen nach dem Komma
`NUMERIC`, `DEC`	synonym für `DECIMAL`
`DATE`	Datum in der Form `'2007-12-31'`
`TIME`	Zeitangabe in der Form `'23:59:59'`
`DATETIME`	Kombination aus den beiden `'2007-12-31 23:59:59'`
`YEAR`	Jahreszahl 1900…2155 (1 Byte)
`TIMESTAMP`	Zeitstempel, der in 4 Bytes Datum und Uhrzeit speichert
`CHAR(n)`	String mit fester Länge ($n \leq 255$ Zeichen)
`VARCHAR(n)`	String mit variabler Länge ($\leq n$)
`TINYTEXT`	String mit variabler Länge (maximal $2^8 - 1$ Zeichen)
`TEXT`	String mit variabler Länge (maximal $2^{16} - 1$ Zeichen)
`MEDIUMTEXT`	String mit variabler Länge (maximal $2^{24} - 1$ Zeichen)
`LONGTEXT`	String mit variabler Länge (maximal $2^{32} - 1$ Zeichen)
`xxBLOB`	Binärdaten mit variabler Länge; Größen wie in `xxTEXT`
`ENUM`	Aufzählung von maximal 65535 Strings

Tabelle der wichtigsten Optionen

`NULL`	gibt an, dass die Spalte auch `NULL` enthalten darf (default)
`NOT NULL`	lässt den Wert `NULL` nicht zu
`DEFAULT xxx`	gibt an, was eingefügt wird, wenn kein Wert angegeben wird
`PRIMARY KEY`	definiert die Spalte als (Teil von einem) Primärschlüssel
`AUTO_INCREMENT`	bewirkt, dass in die Spalte automatisch ein durchlaufender Zähler eingefügt wird. Geht nur bei ganzen Zahlen und wenn `PRIMARY KEY` oder `UNIQUE` als weitere Optionen angegeben sind. Dies wiederum bewirkt ein automatisches `NOT NULL`.
`UNSIGNED`	ganze Zahlen werden ohne Vorzeichen gespeichert, wodurch doppelt so große Zahlen möglich werden. Auch Rechnungen werden vorzeichenlos durchgeführt.
`BINARY`	bewirkt bei `CHAR`- und `VARCHAR`-Spalten, das Vergleichs- und Sortieroperationen binär ausgeführt werden. Das ist effizienter aber unpraktisch, wenn sortiert ausgegeben werden soll.
`CHARACTER SET xx`	gibt bei Strings den Zeichensatz an
`COLLATE yy`	und optional die gewünschte Sortierordnung
`COMMENT text`	speichert Text als Kommentar zur Spalte ab
`SERIAL`	Synonym für `BIGINT NOT NULL AUTO_INCREMENT UNIQUE`

6.4 Die wichtigsten Befehle

Die Befehlsvielfalt von SQL ist umwerfend und kann im MySQL-Referenzhandbuch erschöpfend nachgelesen werden. Hier werden ein paar Befehle mit Erklärungen vorgeführt, die zum Weiterlesen anregen sollen. Grundlage ist die Datenbank `world`, die von der MySQL-Seite herunter geladen werden kann. Sie besteht aus drei Tabellen, die folgenden Aufbau haben:

```
mysql> explain City;
+-------------+----------+------+-----+---------+----------------+
| Field       | Type     | Null | Key | Default | Extra          |
+-------------+----------+------+-----+---------+----------------+
| ID          | int(11)  | NO   | PRI | NULL    | auto_increment |
| Name        | char(35) | NO   |     |         |                |
| CountryCode | char(3)  | NO   |     |         |                |
| District    | char(20) | NO   |     |         |                |
| Population  | int(11)  | NO   |     | 0       |                |
+-------------+----------+------+-----+---------+----------------+

mysql> explain Country;
+----------------+-----------------+------+-----+---------+-------+
| Field          | Type            | Null | Key | Default | Extra |
+----------------+-----------------+------+-----+---------+-------+
| Code           | char(3)         | NO   | PRI |         |       |
| Name           | char(52)        | NO   |     |         |       |
| Continent      | enum('Asia','Eu | NO   |     | Asia    |       |
| Region         | char(26)        | NO   |     |         |       |
| SurfaceArea    | float(10,2)     | NO   |     | 0.00    |       |
| IndepYear      | smallint(6)     | YES  |     | NULL    |       |
| Population     | int(11)         | NO   |     | 0       |       |
| LifeExpectancy | float(3,1)      | YES  |     | NULL    |       |
| GNP            | float(10,2)     | YES  |     | NULL    |       |
| GNPOld         | float(10,2)     | YES  |     | NULL    |       |
| LocalName      | char(45)        | NO   |     |         |       |
| GovernmentForm | char(45)        | NO   |     |         |       |
| HeadOfState    | char(60)        | YES  |     | NULL    |       |
| Capital        | int(11)         | YES  |     | NULL    |       |
| Code2          | char(2)         | NO   |     |         |       |
+----------------+-----------------+------+-----+---------+-------+

mysql> explain CountryLanguage;
+-------------+---------------+------+-----+---------+-------+
| Field       | Type          | Null | Key | Default | Extra |
+-------------+---------------+------+-----+---------+-------+
| CountryCode | char(3)       | NO   | PRI |         |       |
```

```
| Language    | char(30)       | NO   | PRI |         |       |
| IsOfficial  | enum('T','F') | NO   |     | F       |       |
| Percentage  | float(4,1)     | NO   |     | 0.0     |       |
+-------------+----------------+------+-----+---------+-------+
```

Manche der nachfolgenden Befehle sind wörtliche Beispiele, andere sind als allgemeine Regeln wiedergegeben. In allgemeinen Regeln werden optionale Angaben in eckigen Klammern und sich ausschließende Angaben durch den senkrechten Strich | getrennt.

Die Tabelle City soll nachträglich das Attribut Area bekommen:

```
ALTER TABLE City ADD Area FLOAT(8,2) NOT NULL DEFAULT 0.0
```

Die Spalte Area soll umbenannt werden in Flaeche, der Typ soll dabei nicht geändert werden:

```
ALTER TABLE City CHANGE Area Flaeche FLOAT(8,2) NOT NULL
```

Die Spalte Area soll wieder entfernt werden.

```
ALTER TABLE City DROP Area
```

Ein Primärschlüssel soll nachträglich festgelegt werden.

ALTER TABLE tblname ADD PRIMARY KEY (spalte$_1$, spalte$_2$, ...)

Eine neue Datenbank soll angelegt werden (evtl. unter der Bedingung, dass sie noch nicht existiert.

CREATE DATABASE [IF NOT EXISTS] dbname

Die Tabelle Industry soll neu angelegt werden.

```
CREATE TABLE Industry
```

Die Tabelle soll wieder gelöscht werden.

```
DROP TABLE Industry
```

In der Tabelle City soll die Spalte Population umbenannt werden.

```
ALTER TABLE City CHANGE Population Einwohner
```

Die Datenbank schuledb soll gelöscht werden.

```
DROP DATABASE schuledb
```

Alle Inhalte der Tabelle CountryLanguage sollen gelöscht werden (die Tabelle selber aber nicht).

```
DELETE FROM CountryLanguage
```

Lösche alle Daten aus den Tabellen, die sich auf Deutschland oder seine Städte beziehen, lasse aber den Landeseintrag bestehen.

```
DELETE City, CountryLanguage
FROM City, Country, CountryLanguage
WHERE City.CountryCode=Country.Code
  AND CountryLanguage.CountryCode=Country.Code
  AND Country.Name='Germany'
```

Erlaube dem User Dummy vollen Zugriff auf alle Tabellen der Datenbank world und setze gleichzeitig sein Passwort auf xxx. Der Zugriff soll von beliebigen Rechnern her erfolgen dürfen. Dummy soll jedoch nicht in der Lage sein, anderen Usern Zugriff zu ge- oder verwehren.

```
GRANT ALL ON world.* TO Dummy IDENTIFIED BY 'xxx'
```

Dummy soll nachträglich doch befähigt werden, anderen Zugriff zu ge- oder verwehren, falls er das von dem Rechner tut, auf dem der MySQL-Server läuft. Bei Zugriff von diesem Rechner soll kein Passwort nötig sein.

```
GRANT ALL ON world.* TO Dummy@localhost WITH GRANT OPTION
```

Dummy hat sich dumm angestellt. Die Rechte werden wieder entzogen.

```
REVOKE ALL ON world.* FROM Dummy, Dummy@localhost
```

Bayrisch und Hessisch fehlen noch in der Tabelle der in Deutschland gesprochenen Sprachen.

```
INSERT INTO CountryLanguage(Percentage,Language,CountryCode)
VALUES(20,Bavarian,DEU,T),(9,Hessian,DEU)
```

Die englischen Tabellennamen sollen in deutsche umbenannt werden. Die ganze Datenbank kann nicht so einfach umbenannt werden.

```
RENAME Country TO Land, City TO Stadt, CountryLanguage TO Sprache
```

Eine Auswahl soll aufgelistet werden. Es wird nur die allglemeine Form beschrieben. Beispiele folgen in einem nachfolgenden Abschnitt.

```
SELECT [options] column1 [[AS] name1] column2 [[AS] name2] ...
[FROM tablelist]
[WHERE condition]
[GROUP BY groupfield [ASC][DESC]]
[HAVING condition]
[ORDER BY ordercolumn1 [DESC] ordercolumn2[DESC] ...]
[LIMIT [weglass,] rows]
```

Die Einwohnerzahl von Hamburg hat sich um 2135 verringert. (Weitere Änderungen könnten durch Komma getrennt angegeben werden.)

```
UPDATE City SET Population=Population-2135 WHERE Name='Hamburg'
```

6.5 Joins

Joins werden in MySQL automatisch durchgeführt, wenn man in der FROM-Klausel durch Komma getrennt mehrere Tabellen angibt und in WHERE die Joinbedingung angibt. (Ohne diese Bedingung bekäme man das Kreuzprodukt, in dem jeder Eintrag der ersten mit jedem Eintrag der zweiten usw. Tabelle kombiniert wird.)

Schreibt man in der FROM-Klausel statt des Kommas zwischen zwei Tabellen den Befehl JOIN, so bekommt man im wesentlichen die gleichen Ergebnisse, kann aber die Bedingung mit ON oder USING angeben.

Gleichbedeutend mit JOIN ist in MySQL INNER JOIN und CROSS JOIN, vergessen wir das also gleich wieder. Ein kleiner Unterschied entsteht, wenn man statt JOIN einen OUTER JOIN macht. Davon gibt es zwei Sorten und bei beiden darf das Wort OUTER entfallen, vergessen wir es also gleich wieder. LEFT JOIN und RIGHT JOIN. Der NATURAL JOIN ist ein JOIN, bei dem alle Spalten mit gleichem Namen berücksichtigt werden. USING braucht dann natürlich nicht mehr angegeben werden, es werden ja alle in Frage kommenden Spalten verwendet.

Betrachten wir die folgenden Beispiele:

1. `... FROM table1, table2 WHERE table1.xyID=table2.xyID`
2. `... FROM table1 JOIN table2 ON table1.xyID=table2.xyID`
3. `... FROM table1 JOIN table2 USING(xyID)`
4. `... FROM table1 NATURAL JOIN table2`
5. `... FROM table1 LEFT JOIN table2 USING(xyID)`
6. `... FROM table1 RIGHT JOIN table2 USING(xyID)`

Varianten 1 bis 3 sind gleichbedeutend. Die Syntax würde jeweils aber auch noch kompliziertere Bedingungen erlauben. So können z. B. bei USING auch mehrere Spaltennamen durch Komma getrennt angegeben werden.

Variante 4 ist dann gleichbedeutend mit den ersten dreien, wenn xyID die einzige Tabellenüberschrift ist, die in beiden Tabellen vorkommt. Gibt es noch weitere, werden sie vom NATURAL JOIN automatisch berücksichtigt.

Die beiden OUTER JOINs der Varianten 5 und 6 verwendet man, wenn man sicher stellen will, dass die Einträge einer Tabelle auch dann genannt werden, wenn in der anderen kein joinbares Element vorhanden ist. In diesem Fall wird statt eines Eintrags aus der anderen Tabelle NULL angefügt. Beim LEFT JOIN werden also Kombinationen mit allen Elementen der linken Tabelle erzeugt und rechts kann NULL auftreten, beim RIGHT JOIN Kombinationen mit allen Elementen der rechten Tabelle und links kann NULL auftreten.

6.6 Select

Die vielfältigen Möglichkeiten und Feinheiten rechtfertigen einen eigenen Abschnitt.

Berchne Sinus von 45°

```
SELECT SIN(PI()/4)
```

Wie viele Länder gibt es?

```
SELECT count(*) FROM Country
```

Welche Ländernamen enden auf y?

```
SELECT Name FROM Country WHERE Name LIKE '%y'
```

Gib Namen und Einwohnerzahl aller europäischen Städte an.

```
SELECT City.Name, City.Population
FROM City, Country
WHERE CountryCode=Code AND Continent='Europe';
```

Berechne die Gesamteinwohner aller europäischen Städte.

```
SELECT sum(City.Population)
FROM City, Country
WHERE CountryCode=Code AND Continent='Europe';
```

Erzeuge eine Fehlermeldung durch Mischen von Gruppen- und Einzelwerten!

```
SELECT City.Name, sum(City.Population)
FROM City, Country
WHERE CountryCode=Code AND Continent='Europe';
```

Berechne die Gesamteinwohner aller Städte.

```
SELECT sum(City.Population)
```

Berechne die Gesamteinwohner aller Städte gruppiert nach Kontinenten.

```
SELECT Continent, SUM(City.Population)
FROM City, Country
WHERE CountryCode=Code
GROUP BY Continent;
```

An dieser Stelle sollte man vielleicht einmal genauer über die Klausel `GROUP BY` nachdenken. Man fängt damit an, dass man sich erst einmal das Ergebnis der Abfrage ohne Gruppierung vorstellt – und natürlich ohne `SUM(...)`, weil das noch keinen Sinn hat.

In manchen Spalten tauchen wahrscheinlich Einträge mehrfach auf, im obigen Beispiel etwa in der Spalte `Continent`. Wendet man auf diese Spalte `GROUP BY` an, so kommen alle Zeilen mit dem gleichen Kontinent in eine Schachtel. Es gibt dann eine Schachtel mit allen Zeilen aus Europa, eine weitere Schachtel mit allen Zeilen aus Asien usw.

In einer Schachtel sein, heißt aber anschaulich auch, dass man nicht mehr alle Einzelheiten sehen kann, sondern nur noch, was alle gemein haben. Verlangt man im `SELECT` etwas anderes, so bekommt man eine Fehlermeldung. Man verlange also im obigen Beispiel nicht `City`, denn um die Stadt einer Zeile anzugeben, müsste man die Schachtel wieder aufmachen. Kein Problem hingegen ist natürlich `Continent`, weil der für alle Bewohner einer Schachtel der gleiche ist. Ebenso problemlos kann man sich von einer Schachtel irgendeine Summe geben lassen oder eine Anzahl (`COUNT`) oder eine andere Gruppenfunktion.

Im nächsten Beispiel wird `GROUP BY` weggelassen, sodass `SUM` nicht mehr weiß, was es summieren soll, sodass wegen der gleichzeitigen Nennung von Gruppen- und Einzelwerten eine Fehlermeldung entsteht:

```
SELECT Continent, SUM(City.Population)
FROM City, Country
WHERE CountryCode=Code;
```

Es soll festgestellt werden, welche Sprachen in der Schweiz zu welchem Anteil gesprochen werden und ob eine Sprache offiziell ist.

```
SELECT Language, Percentage, IsOfficial
FROM Country, CountryLanguage
WHERE Country.Code=CountryCode AND Name='Switzerland';
```

In welchen Ländern werden genau 2 Sprachen gesprochen. Liste ab dem siebten (es wird mit 0 zu zählen begonnen) höchstens fünf solche Länder.

```
SELECT Country.Name
FROM Country, CountryLanguage
WHERE Country.Code=CountryLanguage.CountryCode
GROUP BY Country.Name
HAVING COUNT(Language)=2
LIMIT 7,5
```

Die verschiedenen Sprachanteile in `CountryLanguage` sortiert nach Häufigkeit.

```
SELECT DISTINCT Percentage FROM CountryLanguage ORDER BY Percentage
```

Die Städte deren Name mit `M` beginnt und die alphabetisch nach `Mo` kommen.

```
SELECT Name FROM City WHERE Name>'Mo' AND Name<='N'
```

Liste alle Länder, in denen Deutsch und Englisch gesprochen wird.

```
SELECT Name
FROM Country, CountryLanguage s1, CountryLanguage s2
WHERE s1.CountryCode=s2.CountryCode AND s2.CountryCode=Country.Code
  AND s1.Language='English' AND s2.Language='German'
```

6.7 Sub-Selects

Seit Version 4.1 unterstützt MySQL auch Sub-Selects. In den meisten Fällen kann man die Anfragen auch mit Joins formulieren, als Sub-Selects sind sie aber manchmal leichter verständlich. Man spart Zeit, wenn man von Anfang an im Kopf behält, dass `ANY` und `SOME` das gleiche bedeuten. Es gibt folgende Varianten:

Wenn der Sub-Select nur einen Wert liefert, kann dieser mit den üblichen Vergleichsoperatoren >, >=, usw. in der Bedingung verwendet werden.

```
SELECT ... WHERE spalte=(SELECT...)
```

Die beiden folgenden Abfragen sind gleichwertig, letztere ist aber eingängiger. Die Bedingung ist erfüllt, wenn der Wert von `spalte` einer aus der Menge ist, die der

Sub-Select gefunden hat. Statt IN kann man auch NOT IN verwenden, um die Bedingung umzukehren.

```
SELECT ... WHERE spalte = ANY (SELECT...)
SELECT ... WHERE spalte IN (SELECT...)
```

Bei der folgenden Abfrage wird ein Ergebnis gelistet, wenn der Sub-Select mindestens einen bzw. keinen Datensatz liefert.

```
SELECT ... WHERE EXISTS (SELECT...)
SELECT ... WHERE NOT EXISTS (SELECT...)
```

Im Folgenden wird mit dem Sub-Select eine virtuelle Tabelle neueTab erzeugt, aus der der äußere Select eine Auswahl trifft.

```
SELECT ... FROM (SELECT...) AS neueTab WHERE ...
```

Will man testen, ob ein Select ein ganz bestimmtes Tupel von Werten listet, so fragt man danach mit ROW. Das Ergebnis ist entweder 1 (wahr) oder 0 (falsch).

```
SELECT ROW(wert1, wert2, ...) = (SELECT...)
SELECT ROW(wert1, wert2, ...) = ANY (SELECT...)
```

6.8 Insert und Delete bei verknüpften Tabellen

Das Einfügen von Daten in Tabellen erfolgt mit dem INSERT-Befehl. Falls die Option AUTO_INCREMENT verwendet wird, weiß man selber nicht, welchen Wert ein Datensatz beim Einfügen bekommen hat. Das muss man aber wissen, wenn der Wert in einer anderen Tabelle gebraucht wird.

Wollen wir z. B. Wetzlar in unsere Datenbank aufnehmen, so fügen wir Wetzlar erst einmal bei den Städten ein, ohne selber die ID anzugeben. Da diese Spalte das Attribut AUTO_INCREMENT hat, wird die ID automatisch festgelegt.

```
INSERT INTO City(Name,CountryCode,District,Population)
VALUES('Wetzlar','DEU','Hessen',52831)
```

Um die tatsächlich vergebene ID zu erhalten, kann man natürlich eine entsprechende Abfrage bei City machen. Gleich nach der Einfügung geht das aber leichter mit

```
SELECT LAST_INSERT_ID()
```

Den Wert 4080, den man dadurch bekommt, kann man nun entweder literal verwenden oder man verwendet die Funktion, wenn man mittlerweile keinen weiteren AUTO_INCREMENT-Vorgang hervorgerufen hat. Die world-Datenbank bietet leider keine Anwendung für diese Funktion, weil sie so einfach aufgebaut ist.

Beim Löschen von Datensätzen muss man natürlich auch beachten, dass die Inhalte von weiteren Tabellen tangiert werden könnten. Wollte man etwa ein Land löschen, so würde sein Code den Sinn verlieren und man müsste ebenfalls alle Städte mit dem gleichen CountryCode löschen, sowie die entsprechenden Einträge in in der Tabelle CountryLanguage. Man sagt: *Die referentielle Integrität muss gewahrt bleiben.* Hierzu mehr im nächsten Kapitel.

6.9 Aufteilung der Sprache in Sektionen

Die Sprache SQL bietet eine überbordende Anzahl von Befehlen, von denen jeder oft wiederum viele Optionen bietet. In diesem Skript ist nur ein winziger Bruchteil wiedergegeben. Erwähnt werden soll aber abschließend noch, dass die Sprache in Bereiche aufgeteilt ist, die nach Wichtigkeit geordnet lauten:

DML Data Manipulation Language: `SELECT`, `INSERT`, `DELETE`, `UPDATE`

DDL Data Definition Language: `CREATE DATABASE/TABLE`, `ALTER`, `DROP`, `RENAME`

DCL Date Control Lanugage: `GRANT`, `REVOKE`

7 Datensicherheit

7.1 ACID

In der Datenbanktheorie gibt es vier Grundanforderungen, die das DBMS (database management system) anstreben sollte. Sie werden durch das Akronym ACID zusammengefasst.

A (atomicity) Eine Transaktion ist eine unteilbare Verarbeitungseinheit und wird entweder ganz oder gar nicht ausgeführt. Wenn nach einem Teil der Transaktion ein Fehler auftritt, wird auch der fehlerlose Teil rückgängig gemacht.

C (consistency) Eine Transaktion führt die Datenbank *von* einem konsistenten Zustand *in* einen anderen konsistenten Zustand.

I (isolation) Eine Transaktion muss so ablaufen, als sei sie die einzige im System. Zwischenzustände (die ja inkonsistent sein können) dürfen für andere Transaktionen nicht sichtbar sein.

D (durability) Ergebnisse einer erfolgreich beendeten Transaktion sind dauerhaft, d.h. sie überleben jeden nachfolgenden Fehler.

Der letzte Punkt ist eher aus technischen Gründen der Rede wert. So werden üblicherweise viele Operationen im schnellen RAM des Rechners durchgeführt und erst später gesammelt auf die Festplatte geschrieben. Wenn nun nach mehreren erfolgreichen Transaktionen im RAM die nächste den Rechner zum Absturz bringt, sind die erfolgreichen Transaktionen verloren. Das ist ein Verstoß gegen D.

7.2 Konsistenz

Eine grobe Definition besagt, dass Daten dann *konsistent* sind, wenn sie korrekt und vollständig sind. Konsistent zu bleiben ist das Ziel eines jeden DBMS (data base management system).

Beispiele für Inkonsistenzen:

1. Referenz ins Leere in einer Tabelle nach Löschung eines Tupels in einer anderen.
2. Von Konto A wurde ein Betrag abgebucht, bei B aber nicht gutgeschrieben.
3. In einem Datensatz fehlt der Eintrag für den Primärschlüssel: `ProdNr=NULL`, `Produkt='Schraube'`

4. Ein Summenfeld hält nicht die korrekte Summe der Einzelbeträge.
5. Negativer Blutalkoholgehalt.
6. Bei Redundanz wurde eine Änderung nur an einer Stelle vorgenommen (Änderungsanomalie).

Mögliche Ursachen dafür:

1. falsche Eingabedaten
2. Programmierfehler in der Zugriffssoftware (php)
3. Falsche Behandlung paralleler Zugriffe
4. Absturz des Betriebssystems
5. Ausfall eines Datenträgers

Aufgabe: Überlegen Sie, welche ACID-Anforderungen die einzelnen Fälle verletzen und welche Fälle grundsätzlich gar nicht als das Problem des DBMS betrachtet werden können.

MySQL bietet Konzepte der Datensicherheit, wenn man Tabellen des Typs InnoDB verwendet. Da dieser Typ standardmäßig nicht verwendet wird, muss er ausdrücklich angefordert werden:

```
mysql> CREATE DATABASE transaktionen;
mysql> CREATE TABLE person(
            knr INT NOT NULL,
            name VARCHAR(20) NOT NULL,
            PRIMYRY KEY(knr)
       ) ENGINE=INNODB;
mysql> CREATE TABLE konto(
            knr INT NOT NULL,
            betrag INT DEFAULT 0,
            PRIMYRY KEY(knr),
            FOREIGN KEY(knr) REFERENCES person(knr)
              ON DELETE RESTRICT
              ON UPDATE CASCADE
       ) ENGINE=INNODB;
mysql> INSERT INTO person VALUES(1,'Apple Computer Inc.');
mysql> INSERT INTO konto VALUES(1,800000000);
mysql> INSERT INTO person VALUES(2,'Beslmeisl Franz');
mysql> INSERT INTO konto VALUES(2,2138);
```

Damit haben wir jetzt zwei geeignete Tabellen. Eine der interessanten Möglichkeiten ist die Definition[1] eines Fremdschlüssels. Damit weiß die Tabelle, dass als Einträge in `konto.knr` nur die Werte in Frage kommen, die auch in `person.knr` vorkommen.

[1] Technische Voraussetzung für die erfolgreiche Definition eines Fremdschlüssels ist übrigens, dass die referenzierten Spalten einen *Index* haben. Was das ist, wurde bisher noch nicht besprochen und muss es auch nicht werden, weil ein solcher Index automatisch zusammen mit einem Primärschlüssel angelegt wird. Wenn es jedoch einmal nötig sein sollte, einen Nicht-Prümärschlüssel als Fremdschlüssel

Ein Eintrag aus der Tabelle person kann erst dann entfernt werden, wenn es keinen Eintrag mehr in konto gibt, der darauf verweist. Versucht man es trotzdem, so wird der entsprechende DELETE-Befehl einfach nicht ausgeführt. Der Grund dafür ist das RESTRICT in der Definition des Fremdschlüssels.

Außer der RESTRICT-Taktik gibt es noch CASCADE, was dazu führen würde, dass beim Löschen eines Datensatzes aus person einfach alle darauf verweisenden Datensätze aus konto mitgelöscht würden. Der Einstz dieser Taktik ist nur mit äußerster Vorsicht anzuraten. Ansonsten gibt es noch SET NULL und NO ACTION. Erstere setzt die ziellosen Zeiger in konto.knr auf den Wert NULL, was immer noch besser ist als ein ganz sinnloser Eintrag. Letzteres tut gar nichts. Wer das will, würde das Attribut aber wohl einfach gleich gar nicht als Fremdschlüssel definieren.

Außer für DELETE kann auch eine Strategie für UPDATE festgelegt werden, also für den Fall, dass in person eine Kontonummer geändert wird. Die darauf zeigenden Einträge in konto werden dann lawinenartig mitangepasst (CASCADE). Auch hier wären die anderen drei Strategien möglich. Jedoch ist CASCADE hier nicht so gefährlich.

7.3 Eigene Versuche

Nun sehen wir uns an, wie mehrschrittige Transaktionen durchgeführt werden. Zu diesem Zweck brauchen wir mehrere Benutzer, die gleichzeitig an der Datenbank angemeldet sind, was im folgenden mit mysql1 und mysql2 im Prompt unterschieden wird.

```
mysql1> START TRANSACTION;
mysql1> UPDATE konto SET betrag=betrag-21380 WHERE knr=2;
mysql1> UPDATE konto SET betrag=betrag+21380 WHERE knr=1;

mysql2> SELECT * FROM konto;
        +-----+-----------+
        | knr | betrag    |
        +-----+-----------+
        |   1 | 800000000 |
        |   2 |      2138 |
        +-----+-----------+

mysql1> SELECT * FROM konto;
        +-----+-----------+
        | knr | betrag    |
        +-----+-----------+
        |   1 | 800021380 |
```

zu verwenden, muss für diesen ein Index erzeugt werden. Wie das geht, findet man in der MySQL-Dokumentation.

```
        |   2 |    -19242 |
        +-----+-----------+
mysql1> ROLLBACK;
mysql1> SELECT * FROM konto;
        +-----+-----------+
        | knr | betrag    |
        +-----+-----------+
        |   1 | 800000000 |
        |   2 |      2138 |
        +-----+-----------+
mysql1> START TRANSACTION;
mysql1> UPDATE konto SET betrag=betrag-2138 WHERE knr=2;
mysql1> UPDATE konto SET betrag=betrag+2138 WHERE knr=1;
mysql1> COMMIT;
mysql1> SELECT * FROM konto;
        +-----+-----------+
        | knr | betrag    |
        +-----+-----------+
        |   1 | 800002138 |
        |   2 |         0 |
        +-----+-----------+
```

MySQL erfährt also durch `START TRANSACTION`, dass jetzt etwas kommt, das zusammengehört und nur als Ganzes funktionieren kann oder als Ganzes missglücken. Ein `ROLLBACK` macht es ungeschehen und ein `COMMIT` macht es endgültig.

Aufgaben:

1. Versuchen Sie, die Person `Beslmeisl Franz` aus `person` zu löschen.
2. Versuchen Sie, in der Tabelle `konto` die Kontonummer 1 zu ändern auf einen Wert, zu dem es keine Person gibt, also etwa 5.
3. Versuchen Sie, in der Tabelle `person` die Kontonummer 1 zu ändern auf 5. Lassen Sie sich danach beide Tabellen vollständig ausgeben.
4. Für `INSERT` kann keine eigene Strategie angegeben werden. Testen Sie das Verhalten der Datenbank, indem Sie in `konto` ein Konto mit der Nummer 3 anlegen, obwohl es gar keine Person gibt, die diese Kontonummer besitzt.
5. Erklären Sie mit den nun vorhandenen Kenntnissen den Begriff *Referentielle Integrität*.

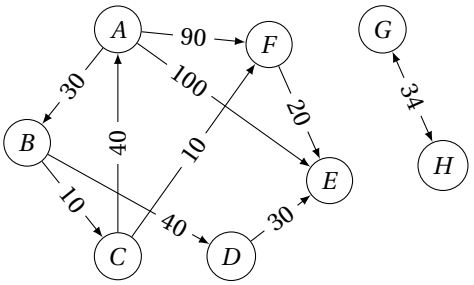
8 Bonusmaterial: Graphen

Graphen sind eine Verallgemeinerung von Listen und Bäumen. Wie diese bestehen Graphen aus Knoten und Kanten. Es gibt jedoch keinen ausgezeichneten Knoten, wie Listenanfang oder Baumwurzel. Wie schon bei einfach verketteten Listen und gewöhnlichen Bäumen haben die Kanten von Graphen meist eine bestimmte Richtung, die angibt, dass man von einem Knoten zum anderen gelangen kann, aber nicht vom anderen zum einen. Verallgemeinernd kommt hinzu, dass nicht nur die Knoten Information tragen können, sondern auch die Kanten. Meistens ist diese Information eine Art Kosten beim passieren einer Kante. Diese Kosten werden auch *Gewicht* genannt. Im allgemeinen hat man also gerichtete, gewichtete Graphen.

8.1 Darstellung

Der Graph im Bild wird noch an vielen Stellen Verwendung finden. Er besteht aus zwei nicht zusammen hängenden Komponenten. Wenn man im Besitz eines Knotens ist, kann man also nicht unbedingt alle anderen Knoten erreichen.

Von *G* zu *H* verläuft eine nicht gerichtete Kante. Dies wird dadurch ausgedrückt, dass an beiden Enden Spitzen eingezeichnet sind. Der Wert 34 gilt für beide Richtungen. Es wären aber auch verschiedene Werte in den beiden Richtungen erlaubt.



0	30	∞	∞	100	90	∞	∞
∞	0	10	40	∞	∞	∞	∞
40	∞	0	∞	∞	10	∞	∞
∞	∞	∞	0	30	∞	∞	∞
∞	∞	∞	∞	0	∞	∞	∞
∞	∞	∞	∞	20	0	∞	∞
∞	∞	∞	∞	∞	∞	0	34
∞	∞	∞	∞	∞	∞	34	0

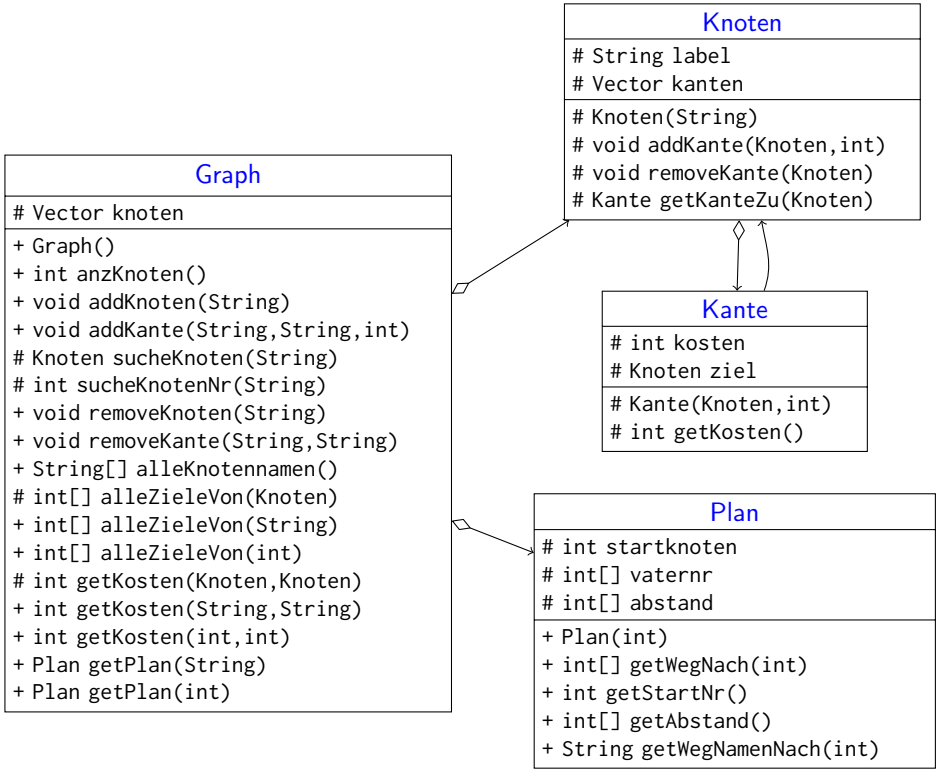
In Java implementiert man die Knoten als eigene Klasse. Jede Instanz besitzt eine Information. Für die Abspeicherung der Kanten existieren zwei weit verbreitete Techniken. Die einfachere verwendet eine *Adjazenzmatrix*. Dabei handelt es sich um ein zweidimensionales Array von Informationen – hier in Form von Zahlen. Die Kanten des obigen Graphen ergeben die Matrix rechts daneben. Diese Darstellung verschwendet viel Speicherplatz bei solch *lichten* Graphen, weil die meisten Einträge ∞ lauten.

Die andere Technik bedient sich linearer Listen. Dabei bekommt jeder Knoten eine Liste von Kanten. Jede Kante weiß, zu welchem Knoten sie geht und welches Gewicht sie hat. Im obigen Beispiel liegen also 8 lineare Listen vor mit folgendem

Inhalt: $A:(B,30)-(E,100)-(F,90)$; $B:(C,10)-(D,40)$; $C:(A,40)-(F,10)$; $D:(E,30)$; $E:$; $F:(E,20)$; $G:(H,34)$; $H:(G,34)$. Dabei kommt es nicht auf die Reihenfolge der Kantennennungen in einer solchen Liste an.

8.2 Implementierung

Wie bei den Listen und Bäumen legen wir Wert darauf, dass ein Anwendungsprogramm nicht Zugriff auf die nackten Knoten und Kanten hat. Zwar sollen die Klassen `Knoten` und `Kante` existieren und alle nötigen Mechanismen anbieten, jedoch muss es eine Klasse geben, die alle Knoten besitzt, die zu einem Graphen gehören, wir nennen Sie `Graph`. Diese bietet nach außen die Methoden an, die nötig sind, um Knoten zu erzeugen und Kanten zu ziehen. Außerdem wird `Graph` einige Algorithmen beinhalten, die man auf Graphen anwenden kann.

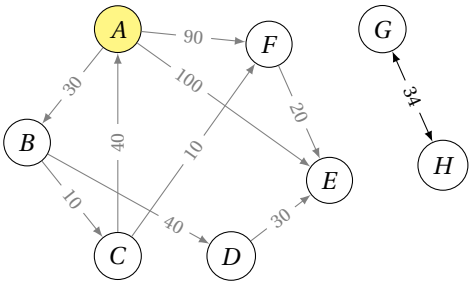


8.3 Algorithmus von Dijkstra

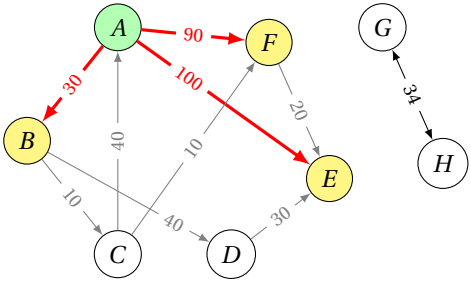
In einem gegebenen Graphen ist es meist interessant, kürzeste Wege zu finden, also solche, bei denen das Gesamtgewicht eines Pfades möglichst gering ist. Das Problem taucht z. B. auf, wenn die Knoten Städte und die Kanten Straßen repräsentieren oder wenn die Knoten Router und die Kanten Netzwerkleitungen repräsentieren.

Dijkstra hat 1959 einen Algorithmus veröffentlicht, der von einem gegebenen Knoten s zu allen erreichbaren Knoten die kürzesten Wege ermittelt. Johnson hat diesen Algorithmus 1979 durch die Verwendung eines Heaps verbessert. Die bisher beste Lösung stammt von Fredman und Tarjan 1987. Wir wollen uns diesen Algorithmus zuerst einmal bildlich veranschaulichen. Als Startknoten dient uns der Knoten A aus der einleitenden Graphik.

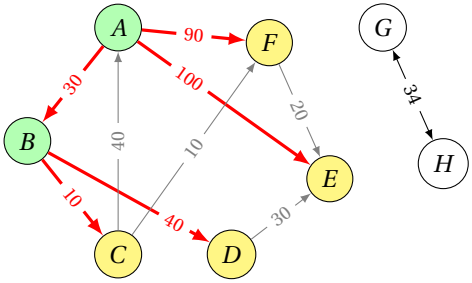
Knoten können sich in drei Zuständen befinden: unbekannt, besucht und fertig. Dies wird graphisch dargestellt durch die Färbungen weiß, gelb und grün. Kanten kennen zwei Zustände: normal und optimal. Dies wird durch die Farben grau und rot angezeigt. Am Anfang ist der Startknoten (in unserem Beispiel A) gelb und alle anderen weiß. Alle Kanten sind grau. Außer A gilt also alles als nicht untersucht.



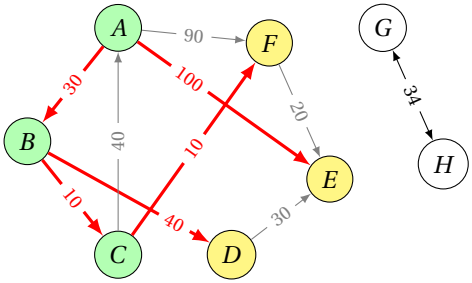
Von allen gelben Knoten suchen wir den mit dem kleinsten Abstand zu A. Es gibt eh nur einen und der hat den Abstand 0 (zu sich selber). Wir nehmen alle davon ausgehenden Kanten zur Kenntnis und färben sie rot, weil sie die kürzesten Wege zu den dadurch neu gefundenen Knoten sind. Diese färben wir gelb für den nächsten Durchgang. Damit ist das vormals gelbe A fertig behandelt und kann grün gefärbt werden.



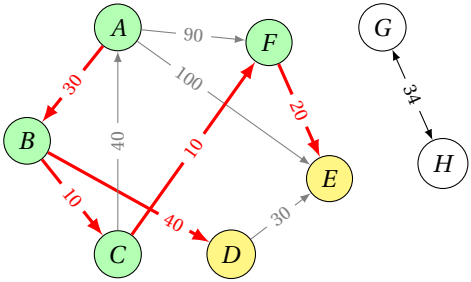
Von allen gelben Knoten suchen wir den mit dem kleinsten Abstand zu A. Es ist B mit Abstand 30. Wir nehmen alle von B ausgehenden Kanten zur Kenntnis und färben sie rot, wenn sie zu neuen Knoten führen, weil sie die kürzesten Wege zu diesen sind. Das ist mit C und D der Fall. Die neuen Knoten färben wir gelb für den nächsten Durchgang. Damit ist das vormals gelbe B fertig behandelt und kann grün gefärbt werden.



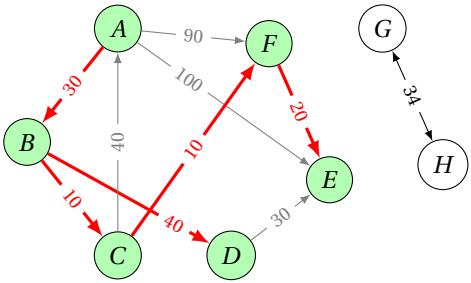
Von allen gelben Knoten suchen wir den mit dem kleinsten Abstand zu A. Es ist C mit (Gesamt-)Abstand 40. Wir nehmen alle von C ausgehenden Kanten zur Kenntnis und färben sie rot, wenn sie zu neuen Knoten führen, was sie aber nicht tun, weil schon alle Knoten bekannt sind. Wir färben sie trotzdem rot, wenn sie einen kürzeren Weg zum Zielknoten bieten, als dieser bisher hatte. Dies passiert bei F. Damit ist das vormals gelbe C fertig behandelt und kann grün gefärbt werden.



Von allen gelben Knoten suchen wir den mit dem kleinsten Abstand zu A. Es ist F mit (Gesamt-)Abstand 50. Wir nehmen alle von F ausgehenden Kanten zur Kenntnis und färben sie rot, wenn sie einen kürzeren Weg zum Zielknoten bieten, als dieser bisher hatte (die bisherige Kante färben wir grau). Dies passiert bei E. Damit ist das vormals gelbe F fertig behandelt und kann grün gefärbt werden.



Von allen gelben Knoten suchen wir einen mit dem kleinsten Abstand zu A. Es ist z. B. D mit (Gesamt-)Abstand 70. Wir nehmen alle von D ausgehenden Kanten zur Kenntnis und lassen sie grau, weil keine kürzeren Wege mehr entstehen. Damit ist das vormals gelbe D fertig behandelt und kann grün gefärbt werden. Der nächste Durchgang ergibt auch nichts interessantes, außer dass auch noch E grün wird und der Algorithmus endet, weil keine gelben Knoten mehr vorhanden sind.



Wie man sieht, wurden die Knoten G und H gar nicht entdeckt. Wie auch, wo zu beiden kein Weg von A führt? Die erklärenden Texte zu den Bildern zeigen zusammen den Algorithmus von Dijkstra. Der Klarheit halber soll der aber noch einmal ausführlich formuliert werden. Dazu denken wir uns die Knoten durchnummeriert von 0 bis 7 (A bis H) und führen drei Arrays ein:

`vaterNr` merkt sich für jeden Knoten den bisher besten Vorgänger auf dem Weg von A. Der Wert -1 bedeutet, dass ein Knoten noch nicht besucht wurde und also noch keinen Vorgänger hat. `abstand` merkt sich für jeden Knoten den bisher kürzesten Gesamtabstand zu A. -1 bedeutet, dass es noch keinen Weg von A zu diesem Knoten gibt. `fertig` merkt sich, ob ein Knoten schon grün ist. Gehen wir von folgender Tabelle aus (Eintragungen mit Bleistift):

	A 0	B 1	C 2	D 3	E 4	F 5	G 6	H 7
`fertig`								
`vaterNr`								
`abstand`								

Dann haben wir folgendes zu tun:

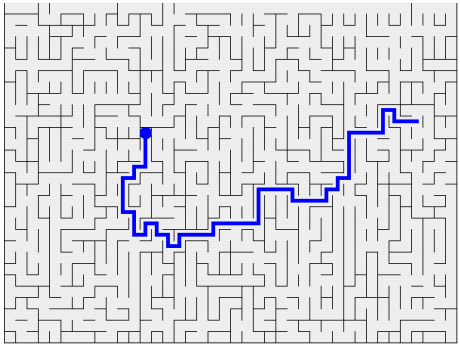
```
int aktuell = Nr des Startknotens (im Beispiel 0)
In fertig alles auf false setzen.
vaterNr[aktuell]=0 alle anderen -1 setzen.
abstand[aktuell]=0 alle anderen -1 setzen.
do{
  fertig[aktuell] = true // grün
  für alle abgehenden Kanten ka des aktuellen Knoten
    int zielnr = Knotennummer des Zielknotens
    if fertig[zielnr] weiter mit nächster Kante ka
    else
      int neudist = abstand[aktuell] + ka.kosten
      if Zielknoten ohne Vater oder neudist<abstand[zielnr]
        vaternr[zielnr] = aktuell
        abstand[zielnr] = neudist
  aktuell = unfertige Spalte mit kleinstem nichtnegativem Abstand
while (aktuell >=0)
```

Diesen Algorithmus implementiert man am besten innerhalb des Konstruktors von `Plan`. Ein `Plan` bezieht sich immer auf einen Startknoten und merkt sich diesen als Index in der Knotenmenge des Graphen. Er kennt alle kürzesten Wege in Form eines Arrays von Vaternummern für alle Knoten des Graphen und er kennt alle zugehörigen Abstände ebenfalls als Array von Werten.

8.4 Anwendung

Als Anwendung bietet sich ein Labyrinth an. Ein Labyrinth ist rechteckig mit einer Anzahl von Zeilen und Spalten. Jede dadurch definierte Zelle wird repräsentiert durch einen Knoten. Von jeder Zelle kann man in 0 bis 4 Nachbarzellen gelangen, wenn keine Wand dazwischen ist. Jeder Knoten hat also höchstens vier Kanten, die alle genau die Kosten 1 haben.

Das Labyrinth im Bild ist 40×30 groß, hat also 1200 Knoten. Die Anzahl der Kanten dürfte etwa 2500 betragen. Nachdem man mit der Maus einen Punkt als Startpunkt gewählt hat, werden die kürzesten Wege zu allen anderen Knoten berechnet. Danach geht man mit der Maus an eine beliebige Stelle und der Weg wird instantan eingezeichnet (wenn es ihn gibt).



9 Grundlagen der Theoretischen Informatik

In diesem Abschnitt werden einige Sachverhalte wiederholt, die Sie wahrscheinlich schon aus der Mittelstufe kennen und einige weitere Definitionen gemacht, die immer wieder gebraucht werden. Es werden auch zwei Beweise wiedergegeben, die einfach und doch beeindruckend sind.

Als Anfänger denkt man leicht, all die Symbole und Definitionen seien schwierig und abstrakt. Das ist Unsinn. Fast immer handelt es sich um Sachverhalte, die sehr anschaulich sind, aber trotzdem einmal genau festgelegt werden müssen. Wer sich die Mühe macht, sich durch die Schreibweisen zu kämpfen, wird also peinlich genau wissen, was gemeint ist. Dies ist typisch für die Sprache der Mathematik.

9.1 Funktionen

Eine (totale) Funktion f ist eine Vorschrift, die jedem Element aus einer Menge D genau ein Element aus einer Menge W zuordnet. D heißt Definitionsmenge, W heißt Wertemenge. Wichtig ist, dass *jedem* Element von D etwas zugeordnet wird. Man schreibt $f : D \to W$, um erst einmal die Namen von Funktion, Definitions- und Wertemenge zu nennen. (Selbstverständlich sind die Namen nicht immer f, D und W!)

Wie die Zuordnungsvorschrift lautet, ist damit noch nicht gesagt. Einzelne Zuordnungen einer Funktion gibt man mit dem Zeichen $\mapsto$ an. (Man beachte, dass dieser Pfeil einen Fuß hat.) So bedeutet etwa $3 \mapsto 9$, dass der Zahl 3 aus der Definitionsmenge die Zahl 9 aus der Wertemenge zugeordnet wird. Das gleiche bedeutet die Schreibweise $f(3) = 9$. Oft kann man alle Zuordnungen auf einmal mit Hilfe einer Formel angeben. Will man jeder Zahl ihre Quadratzahl zuordnen, so ist das am schnellsten durch $f(x) = x^2$ mitgeteilt.

Man nennt eine Funktion *injektiv*, wenn kein Element von W mehrfach zugeordnet wird und *surjektiv*, wenn jedes Element von W zugeordnet wird. Eine injektive und surjektive Funktion nennt man auch *bijektiv*.

In der theoretischen Informatik fast noch wichtiger ist der Begriff der *partiellen Funktion*, die ebenso notiert wird wie eine totale, also $f : D \to W$. Bei einer partiellen Funktion muss *nicht* jedem Element der linken Menge etwas zugeordnet werden. Elemente, denen nichts zugeordnet wird, nennt man Undefiniertheitsstellen. (Die linke Menge nennt man dann auch nicht mehr Definitionsmenge.) Ist u eine Stelle ohne Funktionswert, so schreibt man $f(u) = \perp$.

9.2 Mengen und ihre Kardinalitäten

Mengen sind ungeordnete Ansammlungen unterscheidbarer Dinge (Elemente). Wir unterscheiden die Elemente immer durch ihren Namen, weshalb es keine zwei mit dem gleichen Namen in der gleichen Menge geben kann. Die Menge ohne irgendwelche Elemente heißt *leere Menge* $\{\}$. Als Namen von Mengen verwenden wir Großbuchstaben oder Namen, die mit Großbuchstaben beginnen. Sind die Mengen sehr bekannt, bekommen sie sogar die mathematische Verfettung. Beispiele sind:

$$\begin{aligned}
S &= \{+,-\} = \text{Menge mit den beiden Vorzeichen}\\
\mathit{Ziffern} &= \{0,1,2,3,4,5,6,7,8,9\} = \text{Menge der dezimalen Ziffern}\\
\mathbb{N} &= \{1,2,3,\ldots\} = \text{Menge der natürlichen Zahlen}\\
\mathbb{N}_0 &= \{0,1,2,\ldots\} = \text{Menge der natürlichen Zahlen mit 0}\\
\mathbb{Z} &= \{\ldots,-3,-2,-1,0,1,2,3,\ldots\} = \text{Menge der ganzen Zahlen}\\
\mathbb{Q} &= \left\{\tfrac{z}{n} : z \in \mathbb{Z}, n \in \mathbb{N}\right\} = \text{Menge der rationalen Zahlen}\\
\mathbb{R} &= \{s\, a_0, a_1\, a_2\, a_3 \ldots : s \in S, a_0 \in \mathbb{N}_0, a_i \in \mathit{Ziffern}, i \in \mathbb{N}\}
\end{aligned}$$

Man sieht, dass einige Tricks in der Nennung der Elemente erlaubt sind, wenn die Mengen unendlich viele Elemente enthalten. Zum einen gibt es die Schreibweise mit den Punkten, die darauf vertraut, dass der Leser eine Systematik erkennt und von selber weiß, wie es weiter gehen soll. Wenn das nicht anzunehmen ist, spezifiziert man die Elemente nach einem Doppelpunkt in weiterer mathematischer Notation. Die rationalen Zahlen werden etwa als Brüche festgelegt mit beliebigem natürlichem Nenner und beliebigem ganzzahligem Zähler. Damit sind dann auch die negativen rationalen Zahlen eingeschlossen. Die reellen Zahlen wurden zeichenweise definiert. Das erste Komma ist also diesmal kein Zeichen der Definitionssprache wie der Doppelpunkt oder das Zeichen $\in$, sondern ein Zeichen der reellen Zahl selber.

Mit der *Mächtigkeit* oder *Kardinalität* $|M|$ einer Menge M ist die Anzahl ihrer Elemente gemeint, was bei endlichen Mengen sehr anschaulich ist. So ist offensichtlich im obigen Beispiel $|S| = 2$. Bei unendlichen Mengen unterscheiden wir solche, die ebenso viele Elemente wie $\mathbb{N}$ haben (abzählbare Mengen) und solche, die noch mehr Elemente enthalten (überabzählbare Mengen). Welche Art von Überabzählbarkeit vorliegt, werden wir nicht untersuchen.

Man nennt die Menge A dann nicht mächtiger als die Menge B (in Zeichen $|A| \leq |B|$), wenn es eine surjektive Funktion $n : B \to A$ gibt, also eine Funktion, die alle Elemente von A erreicht, wenn man die Elemente von B einsetzt. Zwei Mengen heißen gleichmächtig, wenn keine mächtiger als die andere ist.

9.2.1 Abzählbare Mengen

Mengen, die nicht mächtiger sind als die natürlichen Zahlen, nennt man abzählbar, die anderen überabzählbar.

- Alle endlichen Mengen sind offensichtlich abzählbar, wie man sich an einem Beispiel klar macht. Die Menge S aus den obigen Beispielen ist abzählbar, weil die Zuordnung $n : \mathbb{N} \to S$ mit $1 \mapsto +$, $2 \mapsto -$, $3 \mapsto -$, $4 \mapsto -$ … alle Elemente von S erreicht.
- $\mathbb{Z}$ ist abzählbar, wegen $n : \mathbb{N} \to \mathbb{Z}$ mit $n(x) = (-1)^x \cdot x/2$ (wobei / für die ganzzahlige Division steht).
- Die geraden Zahlen sind abzählbar, mit $n(x) = 2x$.
- Die ungeraden selbstverständlich auch, mit $n(x) = 2x - 1$.
- Auch $\mathbb{Q}$ ist abzählbar! Das scheint erst einmal unglaublich, weil es sich dabei ja um alle periodischen Dezimalzahlen handelt. (Abbrechende Dezimalzahlen kann man als periodische auffassen, wenn man bedenkt, dass $5 = 5{,}\overline{0}$ ist.) Da kommt man nicht so leicht auf die Zuordnung, aber es gibt eine!

9.2.2 Cantors Diagonalverfahren

Für $\mathbb{R}$ wurde bis heute keine aufzählende Funktion gefunden und Georg Cantor hat bewiesen, dass das auch niemals passieren wird. Es ist also $\mathbb{R}$ mächtiger als $\mathbb{N}$. Man spricht von Überabzählbarkeit. Der Beweis gehört zu den schönsten Tricks in der Mathematik, hier ist er:

Wir zeigen, dass schon die reellen Zahlen zwischen 0 und 1 nicht abzählbar sind. Angenommen es gäbe eine abzählende Funktion für die reellen Zahlen zwischen 0 und 1, dann könnte man sich diese Zahlen untereinander hingeschrieben denken in einer endlosen Liste $n(1)$, $n(2)$, usw.

$i \in \mathbb{N}$	$n(i) \in \mathbb{R}$
1	0,974427895935…
2	0,248198051210…
3	0,800499046492…
4	0,666099584861…
5	0,175544666793…
6	0,821262425094…
⋮	⋮

Alle diese Zahlen haben eine 0 vor dem Komma und irgend einen unendlichen, periodischen oder nichtperiodischen Nachkommateil. Nun setzen wir eine Trickzahl d folgendermaßen zusammen: Vor dem Komma notieren wir eine 0. Von der Zahl $n(1)$ betrachten wir die erste Nachkommastelle. Wenn sie nicht 4 ist, notieren wir für d eine 4. Wenn sie 4 ist, notieren wir eine 5. Von $n(2)$ schauen wir uns die zweite Nachkommastelle an und verfahren wie gehabt. d hat nun schon zwei Nachkommastellen. Wir verfahren nun für alle $n(i)$ gleich und erhalten im vorliegenden Beispiel $d = 0{,}454454\ldots$

Offensichtlich ist $d \neq n(1)$, weil die erste Nachkommaziffer von d nicht 9 ist. Es ist auch $d \neq n(2)$, weil die zweite Nachkommaziffer nicht 4 ist usw. d ist also keine Zahl

der Liste, obwohl d offensichtlich reell ist. Das ist ein Widerspruch und die Annahme, es gäbe eine Abzählung der reellen Zahlen zwischen 0 und 1, ist falsch.

9.3 Potenzmengen und ihre Mächtigkeiten

Von jeder Menge kann man Teilmengen bilden. Z. B. kann man von $\{A, B, C\}$ die Teilmengen $\{\}$, $\{A\}$, $\{B\}$, $\{C\}$, $\{A, B\}$, $\{A, C\}$, $\{B, C\}$, $\{A, B, C\}$ bilden. Das sind 8 Mengen, weil die ursprüngliche Menge 3 Elemente hat und jedes in der Teilmenge enthalten sein kann oder nicht. Das ergibt $2^3 = 8$ mögliche Teilmengen. Die Menge aller Teilmengen nennt man *Potenzmenge* und kürzt sie sinnigerweise mit $2^{\{A,B,C\}}$ ab. Diese Schreibweise steht also nicht für die Zahl 8, sondern für die Menge $\{\{A\}, \{B\}, \ldots, \{A, B, C\}\}$. Offensichtlich hat bei endlichen Mengen die Potenzmenge immer mehr Elemente als die Menge selbst. Ist das bei unendlichen Mengen auch der Fall, etwa bei $\mathbb{N}$? Auch dafür taugt der Beweis von Cantor in leicht abgewandelter Form.

Man beweist, dass $|2^{\mathbb{N}}| > |\mathbb{N}|$ ist, indem man annimmt, $2^{\mathbb{N}}$ sei abzählbar und die Elemente lägen in einer Liste untereinander geschrieben vor uns. (Diesmal haben wir keine reellen Zahlen, sondern Zahlenmengen.) Links von jeder Zahlenmenge denken wir uns die laufende Nummer, also die Zeilennummer.

Wir färben nun alle Zeilen rot oder grün. Grün färben wir sie, wenn die Zeilennummer auch in der Zahlenmenge vorkommt, andernfalls färben wir die Zeile rot. Nun bilden wir eine böse Menge D, die alle roten Zeilennummern enthält. D ist also eine Teilmenge der natürlichen Zahlen. Also muss D eine der Zeilen sein, die ja alle Teilmengen auflisten, nennen wir die Zeilennummer einfach mal d. Welche Farbe hat diese Zeile? Ist sie rot, weil d nicht in D vorkommt? Dann müsste d in D vorkommen, weil D die Menge der Zeilennummern ist, die nicht in der zugehörigen Menge vorkommen. Oder ist die Zeile grün, weil d doch in D vorkommt? Das würde aber bedeuten, dass d eine Zeilennummer ist, die nicht in ihrer zugehörigen Menge D vorkommt. In beiden Fällen erhalten wir einen Widerspruch. Damit ist die Annahme falsch und $2^{\mathbb{N}}$ ist überabzählbar.

9.4 Aufgaben

1. Zeigen Sie, dass sowohl die Menge der geraden, wie auch die Menge der ungeraden Zahlen nicht mächtiger ist als die Menge der natürlichen Zahlen.
2. Geben Sie eine Zuordnungsvorschrift in Formelschreibweise an, die die rationalen Zahlen aufzählt.
3. Geben Sie für die rationalen Zahlen die Umkehrvorschrift an, die aus Zähler und Nenner ermittelt, die wievielte rationale Zahl es ist.

4. Vergewissern Sie sich, dass die Menge der Brüche tatsächlich die Menge der periodischen Dezimalzahlen beinhaltet. Wandeln Sie die Zahlen $0{,}\overline{81}$ und $3{,}1\overline{430}$ in einen Bruch um.
5. Vergewissern Sie sich, dass die Menge der periodischen Dezimalzahlen tatsächlich die Menge der Brüche beinhaltet. Überlegen Sie sich an Hand von Beispielen, warum das Ausrechnen von Brüchen immer zu periodischen Ergebnissen führt und wie lang die Periode höchstens werden kann.

   Berechnen Sie $\frac{3}{7}$ und $\frac{14}{11}$.
6. Die Menge der rationalen Zahlen x mit $0 \leq x < 1$ ist unendlich groß. Begründen Sie, ob man sie mit folgender Methode abzählen kann:

   Bilde die Zahlen $0{,}n$ wobei n erst alle einstelligen Ziffernfolgen durchläuft (beginnend mit 0), dann alle zweistelligen (beginnend mit 01) dann alle dreistelligen (beginnend mit 001) usw. Nach 0,99 kommt also 0,001.
7. Machen Sie sich klar, dass es *viel* mehr irrationale als rationale Zahlen gibt, indem Sie sich einen Papierstreifen der Breite 1 denken, den Sie in jedem Schritt in zwei gleiche Teile zerschneiden und mit der einen Hälfte die nächste rationale Zahl auf dem Zahlenstrahl abdecken. Mit dem Rest verfahren Sie genauso. (Welche rationale Zahl jeweils die nächste ist, ermitteln Sie mit dem Verfahren, das Sie im Unterricht kennen gelernt haben.) Warum ist dieser Gedanke erleuchtend?
8. Machen Sie sich klar, dass in einem Quadrat der Seitenlänge 1 die Diagonale die Länge $\sqrt{2}$ hat. Beweisen Sie, dass $\sqrt{2}$ irrational ist.

10 Grammatiken und Sprachen

10.1 Zeichen, Wörter, formale Sprachen

Ein *Alphabet* Σ ist eine endliche Menge von Zeichen oder Symbolen. Ein *Wort* über Σ ist eine endliche Folge $w = a_1 a_2 \dots a_r$ von Symbolen $a_i \in \Sigma$. Die Länge des Wortes bezeichnen wir mit $|w| = r$. Das Wort mit der Länge 0 heißt *leeres* Wort und wird als ε notiert.

Die Menge aller (endlichen) Wörter über Σ bezeichnet man mit Σ^*. Die gleiche Menge ohne das leere Wort bezeichnet man mit $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

Sind $w = a_1 a_2 \dots a_r \in \Sigma^*$ und $v = b_1 b_2 \dots b_s \in \Sigma^*$, dann bezeichnet man das aus beiden zusammen gesetzte Wort $a_1 a_2 \dots a_r b_1 b_2 \dots b_s$ mit $w \circ v$ oder kurz wv. Es ist $w\varepsilon = \varepsilon w = w$. Für ein $w \in \Sigma^*$ und $n \in \mathbb{N}_0$ definiert man auch $w^n = \underbrace{ww \dots w}_{n \text{ mal}}$, wobei zu beachten ist, dass $w^1 = w$ und $w^0 = \varepsilon$.

Eine Teilmenge $L \subseteq \Sigma^*$ nennt man *Sprache* über dem Alphabet Σ. Sind L und K Sprachen über Σ, so bezeichnet man $\overline{L} = \Sigma^* \setminus L$ als das Komplement von L und die Menge der Wörter wv, wobei $w \in L$ und $v \in K$ als die Konkatenation von L und K, in Zeichen $L \circ K$ (oder kurz LK).

Wie schon bei den Wörtern definiert man für eine Sprache $L \subseteq \Sigma^*$ und ein $n \in \mathbb{N}_0$ die neue Sprache $L^n = \underbrace{LL \dots L}_{n \text{ mal}}$, wobei zu beachten ist, dass $L^1 = L$ und $L^0 = \{\varepsilon\}$.

Ein sehr wichtiges Konzept ist der Kleeneabschluss L^* einer Sprache L. Dieser ist selber wieder eine Sprache über Σ

$$L^* = \bigcup_{n \geq 0} L^n$$

Manchmal trifft man auch auf die Schreibweise $L^+ = \bigcup_{n \geq 1} L^n$. Dabei ist zu beachten, dass $\varepsilon \in L^*$ für alle Sprachen L und $\varepsilon \in L^+$ genau dann, wenn $\varepsilon \in L$.

Auf den ersten Blick ist das ganze also doch recht verwirrend, da man unterscheiden muss zwischen dem leeren Wort ε, der Sprache $\{\varepsilon\}$, die nur das leere Wort enthält und der leeren Sprache $\{\}$, die gar nichts enthält.

10.2 Grammatiken

Eine Grammatik legt fest, nach welcher Maßgabe die Wörter einer Sprache generiert werden können. Als einfaches Beispiel betrachten wir die Sprache der Variablennamen. In den meisten Programmiersprachen sind als Variablennamen Wörter erlaubt,

die mit einem Buchstaben beginnen und dann noch beliebig viele Buchstaben oder Ziffern haben. Wenn wir die Alphabete $\mathscr{B} = \{A, B, \ldots, Z, a, b, \ldots, z\}$ und $\mathscr{Z} = \{0, 1, \ldots, 9\}$ definieren, ist die Sprache der Variablennamen

$$L_{Var} = \mathscr{B} \circ (\mathscr{B} \cup \mathscr{Z})^*.$$

10.2.1 Definition

Eine *Grammatik* ist ein Tupel $G = (V, \Sigma, \mathscr{P}, S)$ bestehend aus

- einer endlichen Menge V von *Non-Terminalen* (auch *Variablen* genannt),
- einem Alphabet Σ von *Terminalsymbolen*, die sich von den Variablen unterscheiden,
- einer endlichen Menge $\mathscr{P}$ von *Produktionen* oder *Regeln*, die der Einschränkung genügen $\mathscr{P} \subseteq (V \cup \Sigma)^* V (V \cup \Sigma)^* \times (V \cup \Sigma)^*$
- einem Startsymbol $S \in V$.

Es ist nicht üblich, die Regeln $\mathscr{P}$ in der Form $(u, v) \in \mathscr{P}$ anzugeben, wie es auf Grund der Definition als kartesisches Produkt anzunehmen ist. Stattdessen schreibt man kurz und intuitiv $u \to v$.

Etwas problematisch sind Regeln, die auf der rechten Seite ε haben. Um Ordnung in die Theorie zu bekommen, versucht man solche Regeln zu vermeiden. Wenn das leere Wort aber zu einer Sprache gehören soll, braucht man mindestens eine Regel, die zu ε führt. Es zeigt sich, dass man die Grammatik dann aber so umformen kann, dass sie genau eine ε-Regel hat, und zwar $S \to \varepsilon$. Außerdem erreicht man noch, dass S auf keiner rechten Seite vorkommt. Eine so umgeformte Grammatik nennt man ε-frei. Wir werden uns so gut wie möglich bemühen, ε-freie Grammatiken zu erzeugen.

10.2.2 Einführendes Beispiel

Am Beispiel L_{Var} studieren wir nun die vier Bausteine einer Grammatik. Wir nehmen als die Mengen $V = \{Start, Bst, Zif, Rest\}$, $\Sigma = \{a, A, b, B, \ldots, z, Z, 0, 1, \ldots 9\}$, $S = Start$ und am wichtigsten

$$\mathscr{P} = \left\{ \begin{array}{l} Start \to Bst\ Rest, \\ Rest \to \varepsilon \mid Bst\ Rest \mid Zif\ Rest, \\ Bst \to a \mid A \mid \ldots \mid z \mid Z, \\ Zif \to 0 \mid 1 \mid \ldots \mid 9 \end{array} \right\}$$

Dabei steht etwa $Zif \to 0 \mid 1 \mid \ldots \mid 9$ für $Zif \to 0, Zif \to 1, Zif \to 2 \ldots$ Der senkrechte Strich ist also eine Abkürzung für ein *oder*. Das Zeichen ε steht wieder für *nichts*; somit ist die Grammatik *nicht* ε-frei, was wir aber im nächsten Abschnitt beheben.

Zu lesen ist das so: Am Anfang hat man *Start*. Das ist aber kein Terminalsymbol; es muss ersetzt werden durch *Bst Rest*. Ersetzt muss so lange werden, bis nur

noch Terminalsymbole vorhanden sind. Dann hat man ein gültiges Wort erzeugt. Wir erzeugen zur Übung das Wort $a42$.

$$\begin{aligned} \mathit{Start} &\Rightarrow \mathit{Bst}\ \mathit{Rest} \Rightarrow \mathit{Bst}\ \mathit{Zif}\ \mathit{Rest} \Rightarrow \mathit{Bst}\ \mathit{Zif}\ \mathit{Zif}\ \mathit{Rest} \\ &\Rightarrow \mathit{Bst}\ \mathit{Zif}\ \mathit{Zif} \Rightarrow a\ \mathit{Zif}\ \mathit{Zif} \Rightarrow a4\ \mathit{Zif} \Rightarrow a42 \end{aligned}$$

Warum verwendet man hier das Zeichen $\Rightarrow$ und nicht $\rightarrow$? Weil es sich hier nicht um die Angabe von Produktionen handelt, sondern um die Herleitung eines speziellen Wortes! Wie $\Rightarrow$ verwendet wird, ist intuitiv klar, wir ersparen uns die genaue Definition. Als Besonderheit sei aber erwähnt, dass man die ganze Herleitung abkürzen kann mit der Schreibweise

$$\mathit{Start} \Rightarrow^* a42.$$

Diese Schreibweise verwendet man, wenn ohne Nennung der Details nur gesagt werden soll, dass eine Herleitung möglich ist.

10.2.3 Durch eine Grammatik erzeugte Sprache

Ist $G = (V, \Sigma, \mathscr{P}, S)$ eine Grammatik, so nennt man

$$\mathscr{L}(G) = \{w \in \Sigma^* : S \Rightarrow^* w\}$$

die durch die Grammatik G erzeugte Sprache. Sie besteht aus allen Wörtern, die sich aus S ableiten lassen.

In unserem Beispiel ist die von G erzeugte Sprache ziemlich offensichtlich die Sprache der Zeichenketten, die mit einem Buchstaben beginnen und dann noch beliebig viele Ziffern und Buchstaben haben.

10.2.4 Äquivalenz von Grammatiken

Zwei Grammatiken $G_1 = (V_1, \Sigma, \mathscr{P}_1, S_1)$ und $G_2 = (V_2, \Sigma, \mathscr{P}_2, S_2)$ mit dem selben Terminalalphabet Σ heißen *äquivalent*, wenn $\mathscr{L}(G_1) = \mathscr{L}(G_2)$.

Die Regeln einer zu unserem Beispiel äquivalenten, ε-freien Grammatik lauten

$$\mathscr{P} = \left\{ \begin{array}{l} \mathit{Start} \rightarrow \mathit{Bst}\ \mathit{Rest} \mid \mathit{Bst}, \\ \mathit{Rest} \rightarrow \mathit{Bst}\ \mathit{Rest} \mid \mathit{Zif}\ \mathit{Rest} \mid \mathit{Bst} \mid \mathit{Zif}, \\ \mathit{Bst} \rightarrow A \mid a \mid B \mid b \mid \ldots \mid Z \mid z, \\ \mathit{Zif} \rightarrow 0 \mid 1 \mid \ldots \mid 9 \end{array} \right\}$$

10.2.5 Nachtrag: Backus-Naur-Form, Syntaxdiagramme

In der Literatur findet man die Produktionen einer Grammatik oft in dieser vereinfachten Form angegeben. Statt $\rightarrow$ wird ::= verwendet. Für optionale Zeichen wird keine eigene Regel angegeben; sie werden einfach in eckigen Klammern genannt. Z. B.

bedeutet

$$S ::= x[W]z \quad \text{das gleiche wie} \quad S \to xz \mid xWz$$

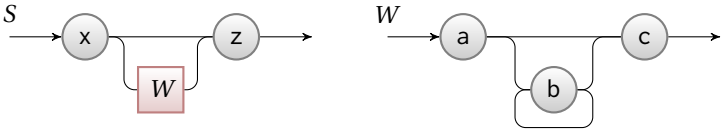
Beliebige Wiederholungen von Zeichen werden in geschweiften Klammern angegeben:

$$W ::= a\{b\}c \quad \text{steht für} \quad W \to aKc, \quad K \to \varepsilon \mid bK$$

Ebenfalls sehr beliebt ist eine äquivalente, etwas graphischer anmutende Darstellung der Produktionen in Form von Syntaxdiagrammen. Syntaxdiagramme existieren für reguläre und kontextfreie Sprachen (die Begriffe werden in Kürze definiert).

Ein *Syntaxdiagramm* besteht aus einer Anzahl von Kästchen, an denen genau zwei Linien enden. Es gibt rundliche Kästchen (Terminale) und eckige (Nonterminale). Der Eingang ist stets links oder oben, der Ausgang rechts oder unten. Linien dürfen sich nicht kreuzen. Jedes Syntaxdiagramm hat einen Namen (Nonterminal). Ein Wort wird dadurch erzeugt, dass man den Linien folgt und an Verzweigungspunkten einen beliebigen Weg einschlägt. Welche Wege erlaubt sind, zeigen Pfeile oder die Linien selber, wenn sie die Form von Gleisen haben.

Nehmen wir das Backus-Naur-Beispiel und geben es in Form zweier Syntaxdiagramme wieder.



10.3 Die Chomsky-Hierarchie

Grammatiken werden eingeteilt in verschiedene Komplexitätsgrade. Wenn die Regeln wenig eingeschränkt sind, kann man komplizierte Sprachen ableiten und die Herleitungen sind im allgemeinen schwierig. Bei stärker eingeschränkten Regeln sind die Sprachen übersichtlicher und leichter abzuleiten. Wir unterscheiden die Typen 0 (schwierig) bis 3 (einfach).

Ist $G = (V, \Sigma, \mathscr{P}, S)$ eine Grammatik, so nennt man G vom Typ

0 – *rekursiv aufzählbar*, wenn neben der Definition auf Seite 78 keine weiteren Einschränkungen gemacht werden.

1 – *kontextsensitiv*, wenn sie vom Typ 0 ist und für alle Regeln $u \to v$ gilt $|u| \le |v|$.

2 – kontextfrei, wenn sie vom Typ 1 ist und in allen Regeln auf der linken Seite genau eine Variable und sonst nichts steht.

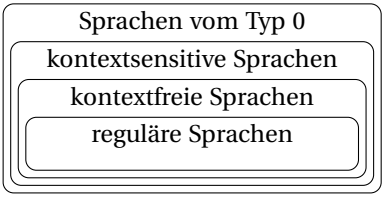
3 – regulär, wenn sie vom Typ 2 ist und in allen Regeln auf der rechten Seite nur ein Terminal oder ein Terminal gefolgt von einer Variable steht (rechtsregulär). Es ist auch erlaubt, dass in allen Regeln auf der rechten Seite ein Terminal oder eine Variable gefolgt von einem Terminal steht (linksregulär). Gemischt dürfen die Fälle aber nicht auftreten.

Bis hierher ist mit Grammatiken vom Typ 1 bis 3 niemals das leere Wort ε ableitbar, weil ja damit die rechte Seite einer Regel die Länge 0 hätte und nicht mehr länger als die linke Seite wäre. Das leere Wort wird aber in vielen Sprachen gewünscht, weshalb wir die ε-Sonderregel einführen: Soll ε zu einer Sprache gehören, so ist die Regel $S \to \varepsilon$ (wobei S das Startzeichen ist) ausnahmsweise erlaubt. Dann darf aber S in keiner Regel auf der rechten Seite mehr vorkommen (was man aber problemlos hinbekommt).

Wir werden die ε-Sonderregel bei Grammatiken vom Typ 2 oder 3 manchmal leichtfertig brechen, wenn wir uns vergewissert haben, dass wir sie mit mehr Anstrengung auch hätten einhalten können. Bei Typ 0 oder 1 werden wir sie aber stets streng einhalten.

Einher gehend mit den vier Grammatiktypen spricht man auch von Sprachtypen. Man sagt, eine Sprache $L \subseteq \Sigma^*$ ist vom Typ i, wenn es eine Grammatik vom Typ i gibt, die die Sprache erzeugt, also $\mathscr{L}(G) = L$. Je nach Grammatik nennt man auch die erzeugte Sprache rekursiv aufzählbar, kontextsensitiv, kontextfrei oder regulär.

Das Bild zeigt, dass jede Sprache vom Typ $i > 0$ auch vom Typ $i-1$ ist. Wenn wir eine Sprache begründet in die Chomsky-Hierarchie einordnen wollen, geben wir immer ihren größt möglichen Typ an und zeigen, dass sie dazu gehört. Dann begründen wir noch, warum sie nicht einen noch größeren Typ hat.



10.3.1 Beispiele

1. In den folgenden vier Beispielen verwenden wir durchgehend $V = \{S, A, B\}$ und $\Sigma = \{a, b, c\}$. Das Startsymbol ist stets S.
   - Die folgenden Regeln gehören zu einer Grammatik vom Typ 0 aber nicht vom Typ 1:

     $$S \to AS \mid ccSb, \quad cS \to a, \quad AS \to Sbb, \quad cSb \to c$$

   - Die folgenden Regeln gehören zu einer kontextsensitiven Grammatik, die nicht kontextfrei ist:

     $$S \to aAb, \quad aA \to Abc, \quad Abc \to aacc$$

   - Nun ein Regelsatz für eine kontextfreie Grammatik, die nicht regulär ist:

     $$S \to AB, \quad A \to aBb \mid Abc \mid bc, \quad B \to A$$

- Und schließlich eine reguläre Grammatik:

$$S \to b \mid bA, \quad A \to a \mid aA \mid aS.$$

2. Die folgende kontextfreie Grammatik $G = (V, \Sigma, \mathscr{P}, S)$ erzeugt alle aussagenlogischen Formeln über der Menge $X = \{x_1, \ldots, x_n\}$.

   $V = \{S\}$

   $\Sigma = \{x_1, \ldots, x_n, true, \neg, \wedge, (,),\}$

   $\mathscr{P} = \{S \to true \mid x_1 \mid \ldots \mid x_n \mid (S \wedge S) \mid (\neg S)\}$

   Falls Sie an dieser Stelle nicht vertehen, warum das *aussagenlogische Formeln* sein sollen, so ist das kein Wunder. Eine Grammatik erklärt nur den korrekten Aufbau der Wörter einer Sprache, also die *Syntax*. Sie erklärt nicht die Bedeutung der Wörter, also die *Semantik*. Die formale Erfassung von Semantik ist üblicherweise unglaublich viel schwieriger.

10.4 Reguläre Ausdrücke

Die nächste Sprache, die wir hier betrachten, kann zu Verwirrung führen, weil ihre Wörter dazu benutzt werden, eine andere Sprache zu beschreiben. Wir kümmern uns deshalb erst einmal um die Sprache selbst; es ist die Sprache der regulären Ausdrücke, genannt RegEx. Wozu man sie braucht, kommt nachher in einem zweiten Schritt.

Dummerweise kommen in der Sprache RegEx Symbole vor, die wir bisher immer zur Beschreibung von Grammatiken benutzt haben. Um Verwechslungen zu vermeiden, werden die Symbole bis auf Weiteres rot dargestellt:

$$\Sigma = \{ \mid, \varepsilon, (,), ^*, a, \ldots, z, A, \ldots, Z\}$$

Mit diesem Alphabet kann man nun die Grammatik der Sprache RegEx angeben:

$$\begin{aligned}\mathscr{R} &= (\{R, S\}, \Sigma, \mathscr{P}, S)\\ \mathscr{P} &= \{S \to \varepsilon \mid R,\\ &\qquad R \to \varepsilon \mid a \mid \ldots \mid Z,\\ &\qquad R \to (R) \mid RR \mid R|R \mid R^*\}\end{aligned}$$

Einige Wörter dieser Sprache sind ε (das leere Wort), ε (dieses hat einen Buchstaben), $M(a \mid e)(i \mid y)(\varepsilon \mid e)r$ (dieses Wort hat keine Leerzeichen). Durch die Angabe der Grammatik ist wie immer die Syntax der Sprache definiert. Aber welche Bedeutung haben all diese seltsamen Wörter, was ist die Semantik der Sprache?

Hier kommen wir nun zu der anderen Sprache, von der oben die Rede war. Sie hat fast das gleiche Alphabet, aber ohne die ersten fünf Zeichen.

$$\Sigma' = \{a, \ldots, z, A, \ldots, Z\}$$

Ab hier ist es also nicht mehr nötig, rot zu schreiben – wenn eines der fünf fragwürdigen Zeichen vorkommt, ist es sicher kein Buchstabe mehr in der Sprache, die wir nun betrachten.

Das Wort $M(a \mid e)(i \mid y)(\varepsilon \mid e)r$ in der Sprache RegEx steht für die folgende Menge von Wörtern: {Mair, Maier, Mayr, Mayer, Meir, Meier, Meyr, Meyer}, also eine Sprache über Σ', die durch das rote Musterwort beschrieben wird. Wollte man zusätzlich auch Maayer, Maaayer usw., so wäre die zugehörige RegEx $M(aa^* \mid e)(i \mid y)(\varepsilon \mid e)r$.

Wahrscheinlich haben Sie die Semantik der Sprache RegEx allein auf Grund dieses Beispiels schon durchschaut. Semantik zu vermitteln, ist ja üblicherweise viel schwerer als Syntax, in diesem Fall ist es jedoch ausnahmsweise recht einfach. Hierzu treffen wir für zwei reguläre Ausdrücke α und β folgende Vereinbarungen:

$$\begin{array}{ll} \mathscr{L}(\varepsilon) = \{\} & \mathscr{L}(\alpha\beta) = \mathscr{L}(\alpha) \circ \mathscr{L}(\beta) \\ \mathscr{L}(\varepsilon) = \{\varepsilon\} & \mathscr{L}(\alpha \mid \beta) = \mathscr{L}(\alpha) \cup \mathscr{L}(\beta) \\ \mathscr{L}(a) = \{a\} & \mathscr{L}(\alpha^*) = \mathscr{L}(\alpha)^* \end{array}$$

Etwas ausführlicher formuliert bedeutet das, dass der leere reguläre Ausdruck eine Sprache ohne Wörter definiert (die leere Sprache), der reguläre Ausdruck ε (mangels Verwechslungsgefahr nicht mehr rot) definiert die Sprache mit dem leeren Wort, der reguläre Ausdruck a definiert die Sprache mit dem einzigen Wort a usw. An dieser Stelle müssen Sie vielleicht nachsehen, was die Verkringelung zweier Sprachen ist, was die Vereinigung und was die Sternung.

Abschließend einige Anmerkungen:

- Bei regulären Ausdrücken im wirklichen Leben werden überflüssige Klammern weggelassen, weil man vereinbart, dass * stärker bindet als die Aneinanderreihung und diese stärker als $\mid$.
- Im wirklichen Leben gönnt man sich nicht nur Aneinanderreihung, $\mid$ und *, sondern etliche weitere Zeichen mit mächtigen Bedeutungen. In der Theorie reichen aber die wenigen Zeichen aus, um alle RegEx-Wünsche zu erfüllen.
- Beachten Sie, dass die Sprache RegEx nur kontextfrei ist, jedes Wort daraus aber eine reguläre Sprache beschreibt.
- Im nächsten Kapitel werden reguläre Sprachen behandelt. Es sollte sich an vielen Stellen die Möglichkeit bieten, RegExe zu üben!

10.5 Aufgaben

1. Erklären Sie, warum die Grammatik in 10.2.2 zur Erzeugung der Sprache L_{Var} nicht regulär ist. Geben Sie eine reguläre Grammatik an, die die gleiche Sprache erzeugt.
2. Erzeugen Sie zu jeder in den Beispielen 1 auf Seite 81 gegebenen Grammatik einige Wörter und formulieren Sie in eigenen Worten, wie die Wörter der je-

weiligen Sprache beschaffen sind. Bemühen Sie sich, dass Ihre Beschreibung der Sprachen diese korrekt wiedergibt.

3. Lässt man von einem algebraischen Term alle Zahlen und Operatoren weg, so bleiben nur öffnende und schließende Klammern übrig. Geben Sie eine Grammatik an, die alle diese wohlgeformten Klammerterme erzeugt. Beispiele sind () oder ()(()()).
4. Schreiben Sie ein Programm, das alle wohlgeformten Klammerausdrücke aufzählt.
5. Zeigen Sie die Richtigkeit der Behauptung, dass eine Sprache vom Typ i auch eine Sprache vom Typ $i-1$ ist.
6. Leiten Sie aus der Grammatik für aussagenlogische Formeln folgende Formel her: $(((\neg x_1) \wedge x_2) \wedge x_3)$
7. Welche Sprache erzeugt die Grammatik $(\{S, A, B\}, \{a, b\}, \mathscr{P}, S)$?
   Hierbei ist $\mathscr{P} = \{S \to bA \mid aB, \quad A \to a \mid aS \mid bAA, \quad B \to b \mid bS \mid aBB\}$.
8. Eine Grammatik einfacher deutscher Sätze wird durch folgende Produktionen gegeben. Das Startsymbol ist *Satz*. Bilden Sie einige Wörter dieser Grammatik.

$$\begin{aligned}
\textit{Satz} &\to \textit{Subjekt Prädikat Objekt},\\
\textit{Subjekt} &\to \textit{Eigenname} \mid \textit{Substantivgruppe},\\
\textit{Substantivgruppe} &\to \textit{Artikel Substantiv},\\
\textit{Prädikat} &\to \textit{Verb},\\
\textit{Objekt} &\to \textit{Akkusativergänzung},\\
\textit{Akkusativergänzung} &\to \textit{Substantivgruppe},\\
\textit{Verb} &\to \text{kauft} \mid \text{liebt} \mid \text{liest},\\
\textit{Artikel} &\to \text{die} \mid \text{das} \mid \text{ein},\\
\textit{Substantiv} &\to \text{Buch} \mid \text{Mädchen} \mid \text{Kartoffeln},\\
\textit{Eigenname} &\to \text{Peter}
\end{aligned}$$

9. Geben Sie eine kontextsensitive Grammatik für die Sprache

$$L = \left\{ w \in \{a, b, c\}^* : anz(w, a) = anz(w, b) = anz(w, c) \right\}$$

   an. Dabei ist $anz(w, x)$ die Häufigkeit des Zeichens x im Wort w. Leiten Sie mit Ihrer Grammatik die Wörter $aabbcc$ und $abccab$ her.

10. Gegeben ist eine Grammatik mit den Produktionen $\mathscr{P}$. Die Großbuchstaben sind Variable, die Kleinbuchstaben Terminale und S ist das Startsymbol.

$$\mathscr{P} = \left\{ \begin{array}{r} S \to aBC \\ S \to aSBC \\ CB \to BC \\ aB \to ab \\ bB \to bb \\ bC \to bc \\ cC \to cc \end{array} \right\}$$

    Welche Sprache wird durch diese Grammatik erzeugt? Beweisen Sie Ihre Behauptung, wenn Sie können.
    Welcher Zusammenhang besteht mit der vorigen Aufgabe?

11. Begründen Sie, warum Syntaxdiagramme für Sprachen des Typs < 2 im allgemeinen nicht existieren.

12. Geben Sie Grammatiken von möglichst hohem Typ an, die die folgenden Sprachen erzeugen:
    a) Palindrome aus den Zeichen 0 und 1, die mit einer 1 beginnen,
    b) Wörter aus den Zeichen 0 und 1, die das Teilwort 001 enthalten,
    c) $L = \{a^n b^n c^k : n \geq 0, k \geq 2\}$,
    d) $L = \{\varepsilon, 001, 010, 101\} \cup \{(01)^n (10)^m : n, m \geq 1\}$.
13. Die folgenden Produktionen sind im bisherigen Sinne uninteressant, weil sie die leere Sprache erzeugen. Ersetzt man aber in jedem Schritt *alle* Variablen und gibt den entstandenen Term einem Zeichenprogramm, so entsteht die sog. Drachenkurve. Das Startzeichen ist O. $O \to ON$, $N \to WN$, $W \to WS$, $S \to OS$. Schreiben Sie ein Programm, das die Variablen als Himmelsrichtungen auffasst und Striche konstanter Länge in diese Richtungen zeichnet. Die Länge sollte umso kleiner gewählt werden je öfter ersetzt wird.
14. Wie in der vorigen Aufgabe entsteht hier eine Figur: die Sierpinski-Kurve. Der Start ist SS (entspricht nicht den Regeln einer Grammatik). Wieder müssen alle Variablen gleichzeitig ersetzt werden. $S \to STST$, $T \to TTST$. Schreiben Sie ein Programm, das S als rechtwinkligen Haken *links-vorwärts-links-vorwärts* und T als *rechts-vorwärts-rechts-vorwärts-links-vorwärts* interpretiert.

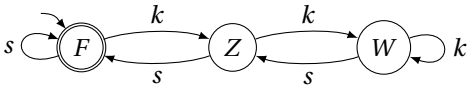
11 Reguläre Sprachen

Der nächste Schritt nach der genauen Definiton von Sprachen durch Grammatiken ist die Erkennung einer Sprache durch ein Computerprogramm. Den in der Chomsky-Hierarchie genannten Grammatiktypen entsprechen jeweils verschieden komplexe Mechanismen. Wir beginnen mit der einfachsten Sprachfamilie, den regulären Sprachen und definieren für ihre Verarbeitung den *deterministischen endlichen Automat, kurz DEA*.

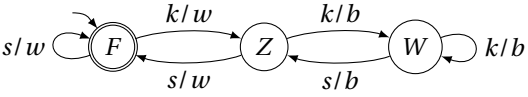
Ein DEA ist ein Mechanismus, der die Zeichen eines Wortes der Reihe nach einliest und dabei jeweils von einem aktuellen Zustand durch ein aktuelles Zeichen eindeutig (deterministisch) in einen neuen Zustand wechselt.

11.1 Beispiel: Der Hund, ein DEA

Ein sehr einfacher Automat *Hund* simuliert einen gutmütigen Hund, der nicht allzu stark misshandelt (geknufft k) oder gestreichelt (s) wird. Der Hund befindet sich dadurch in drei möglichen Zuständen: friedlich (F), zweifelnd (Z) oder wütend (W). Der Anfangszustand des Hundes sei friedlich, was durch den einleitenden Pfeil angedeutet wird. F wurde als Akzeptanzzustand gezeichnet, weil der Hund dort zufrieden ist.



Manchmal haben DEAs auch Ausgaben. Der Hund etwa könnte bellen (b) oder mit dem Schwanz wedeln (w). Ausgaben notiert man, durch einen Schrägstrich getrennt, rechts von der Eingabe. Der Hundeautomat sieht dann so aus.



11.2 Definitionen zum DEA

Da sie von größerem theoretischem Interesse sind, definieren wir nur solche deterministische endliche Automaten DEAs streng, die keine Ausgabe haben. (Im Englischen heißen sie DFA = deterministic finite automaton.)

Ein deterministischer endlicher Automat $\mathcal{M}$ ist ein Tupel $\mathcal{M} = (Q, \Sigma, \delta, q_0, A)$, bestehend aus einer endlichen Menge Q von Zuständen, einem endlichen Alphabet Σ, einer partiellen Funktion $\delta : Q \times \Sigma \to Q$, einem Anfangszustand $q_0 \in Q$ und einer Menge $A \subseteq Q$ von Endzuständen. Die Endzustände nennt man auch Akzeptanzzustände, δ heißt Übergangsfunktion.

Am Anfang liegt das Eingabewort $w \in \Sigma^*$ vor, der Lesekopf steht vor dem ersten Zeichen und der Automat befindet sich im Startzustand q_0. Bei jedem Schritt wird ein Zeichen $z \in w$ gelesen. Gibt es für z keinen Übergang heraus aus dem aktuellen Zustand q, weil $\delta(q, z) = \perp$, so hält der Automat verwerfend (also nicht-akzeptierend) an. Ist das Wort zu Ende gelesen, hält der Automat ebenfalls an. Befindet er sich dann in einem der akzeptierenden Zustände von A, so hat er akzeptierend angehalten, andernfalls verwerfend. Das Erreichen eines akzeptierenden Zustands beendet nicht den Lauf, wenn w noch nicht fertig gelesen ist.

Die Übergangsfunktion $\delta : Q \times \Sigma \to Q$ gibt nur an, wie es bei einem bestimmten Zustand mit einem bestimmten Zeichen weiter geht. (Im einleitenden Beispiel etwa gilt $\delta(W, s) = Z$, weil der Hund durch Streicheln vom wütenden in den zweifelnden Zustand wechselt.) Für die Definition eines DEA ist das ausreichend. Für eine einfache Darstellung dessen, was ein DEA der Reihe nach tut, ist es aber sinnvoll, die Funktion δ auf ganze Wörter zu erweitern, d. h. $\delta : Q \times \Sigma^* \to Q$. Will man also zeigen, welchen Zustand der anfangs friedliche Hund erreicht, wenn ihm $skks$ widerfährt, so schreibt man:

$$\delta(F, skks) = \delta(F, kks) = \delta(Z, ks) = \delta(W, s) = Z$$

oder auch kürzer $\delta(F, skks) = Z$. Die Folge der durchlebten Zustände $FFZWZ$ nennt man den *Lauf* des DEA *Hund* für das Wort $skks$. Ein Lauf kann auch $\perp$ als Ergebnis haben.

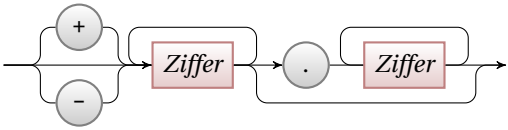
Die graphische Darstellung von DEAs sollte nicht verwechselt werden mit Syntaxdiagrammen! Die Zustände werden zwar in Kreisen dargestellt, sind aber weder Terminale noch Variable! Der Anfangszustand wird durch einen aus dem Nichts kommenden Pfeil kenntlich gemacht, die Endzustände durch fette oder doppelte Kreise, alle anderen Zustände durch einfache Kreise. Die Übergangsfunktion zeigt sich in der Beschriftung der Pfeile mit den gelesenen Zeichen. Eine vollständige Darstellung ist auch tabellarisch möglich (siehe nachfolgende Beispiele.)

11.3 Beispiele

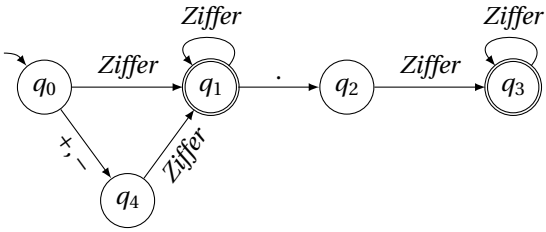
11.3.1 Dezimalzahlen

Dezimalzahlen haben ein optionales Vorzeichen und mindestens eine Ziffer. Dann können weitere Ziffern folgen, eventuell durch ein einziges Komma unterbrochen.

Das folgende Bild zeigt das Syntaxdiagramm. Das Teildiagramm von *Ziffer* ist trivial und wurde deshalb weggelassen.



Der zugehörige DEA kommt mit 5 Zuständen aus. Die Namen der Zustände dürften auch aussagekräftiger sein, aber die Benennung als q_i ist sehr verbreitet.

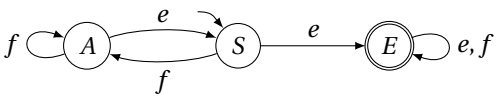


In der Darstellung als Tabelle markieren wir den Startzustand mit einem eingehenden Pfeil und die Endzustände mit ausgehenden. Das ist kein Standard, weil sich diese Darstellung wegen ihrer Unübersichtlichkeit nicht durchgesetzt hat. (Die Tabelle wäre eigentlich viel schmaler und höher, wenn man streng für jede Kombination aus Zustand und gelesenem Zeichen eine eigene Zeile eintrüge.)

q_i	Zeichen	q_{i+1}
$\rightarrow q_0$	0,1,2,3,4,5,6,7,8,9	q_1
$\rightarrow q_0$	+, −	q_4
$\leftarrow q_1$	0,1,2,3,4,5,6,7,8,9	q_1
$\leftarrow q_1$	.	q_2
q_2	0,1,2,3,4,5,6,7,8,9	q_3
$\leftarrow q_3$	0,1,2,3,4,5,6,7,8,9	q_3
q_4	0,1,2,3,4,5,6,7,8,9	q_1

11.3.2 Noname

Der folgende DEA mit dem Alphabet $\{e, f\}$ akzeptiert genau die Wörter, die mit e beginnen oder *fee* enthalten.



Es ist $\delta(S, fffee) = E$ mit dem akzeptierenden Lauf $SAAASE$ und $\delta(S, fef) = A$ mit dem verwerfenden Lauf $SASA$.

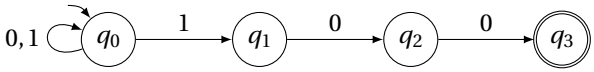
11.4 Definition des NEA

Die Erstellung von DEAs kann schon bei mittelmäßigen Problemen sehr schwierig werden. Deshalb erlaubt man sich Automaten, die pro Schritt mehrere Übergangsmöglichkeiten haben und deshalb weniger Zustände brauchen. Der Vorteil ist die geringere Komplexität, der Nachteil ist, dass man nicht mehr Schritt für Schritt durcharbeiten kann, was als nächstes passiert, weil ja mehrere Möglichkeiten bestehen. Hat der Automat aber für ein Wort wenigstens einen akzeptierenden Lauf, so findet man diesen, wenn man alle Übergangskombinationen ausprobiert. Man bezahlt also mit einem größeren Zeitaufwand beim Spielen für die geringere Komplexität. Spielt man einem Zuschauer nur den akzeptierenden Lauf vor, so fragt dieser natürlich, wie man es angestellt hat, gerade dieses Glück zu finden. Die übliche Antwort ist dann, dass man zufällig (nicht-deterministisch) den richtigen Weg gefunden hat, deshalb der Begriff *nicht-deterministischer endlicher Automat* NEA. Das Erfreuliche an der Geschichte ist, dass man jeden NEA in einen DEA umwandeln kann, was zwar auch eine Menge Arbeit bedeutet, aber keine besonders intelligente. Wir definieren:

Ein NEA $\mathscr{M}$ ist ein Tupel $\mathscr{M} = (Q, \Sigma, \delta, Q_0, A)$ wie beim DEA bestehend aus einer endlichen Menge Q von Zuständen, einem endlichen Alphabet Σ und einer Menge $A \subseteq Q$ von Endzuständen. Im Unterschied zum DEA kann es mehrere Anfangszustände $Q_0 \subseteq Q$ geben und die Übergangsfunktion ist von der Form $\delta : Q \times \Sigma \to 2^Q$. (Letztere ist die Potenzmenge von Q.)

11.5 Beispiel: Mustererkennung mit einem NEA

Als Beispiel suchen wir einen Automaten mit $\Sigma = \{0,1\}$, der genau die Wörter akzeptiert, die auf 100 enden. Das bekommen wir mit einem DEA noch gut hin, trotzdem geht es leichter mit einem NEA:



Es gibt zwei Möglichkeiten, wie q_0 auf eine 1 reagieren kann. Entweder geht der Automat danach in Zustand q_1 oder er bleibt in q_0. Es gibt offensichtlich einen akzeptierenden Lauf für jedes Wort, das auf 100 endet. Man muss nur lange genug im Zustand q_0 bleiben und dann für die letzten drei Zeichen die letzten Zustände durchlaufen.

Die meisten Wörter haben in einem NEA mehrere Läufe. Interessant ist aber nur, ob es einen akzeptierenden Lauf gibt und wie dieser ggf. lautet. Das Wort 100 hat nicht nur den akzeptierenden Lauf $q_0q_1q_2q_3$, sondern auch den verwerfenden $q_0q_0q_0q_0$.

11.6 Umwandlungen

Endliche Automaten haben viele gleichwertige Darstellungsformen. Da es oft nötig ist, diese ineinander umzuwandeln, sehen wir uns in den folgenden Unterabschnitten an, wie man das am besten anstellt.

11.6.1 DEA → reguläre Grammatik

In diesem Abschnitt werden wir einsehen, dass man einen DEA leicht in eine reguläre Grammatik umwandeln kann (die Umkehrung kommt später). Dazu fassen wir die Zustände als Nonterminale auf und die Übergänge als Regeln der Grammatik. Übergänge zu Endzuständen führen zu Terminalen in der Grammatik. Wir führen den Vorgang am obigen Noname-Beispiel durch und erhalten die Produktionen einer regulären Grammatik:

$$S \to e \mid eE \mid fA, \qquad A \to fA \mid eS, \qquad E \to e \mid f \mid eE \mid fE.$$

11.6.2 Umwandlung NEA ↔ DEA

Um einen DEA in einen NEA umzuwandeln, muss man im Grunde gar nichts tun. Lediglich bei der Tupelschreibweise sind dort, wo beim NEA Mengen angzugeben sind, Mengenklammern um die Einzelsymbole des DEA zu machen.

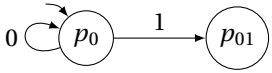
Dass man auch jeden NEA in einen DEA umwandeln kann, wird nicht streng bewiesen, weil die Formalismen recht unübersichtlich sind. Im Wesentlichen geht der Beweis aber so, wie das nachfolgend am Beispiel vorgeführte Verfahren. Wer also die Richtigkeit des Verfahrens einsieht, erkennt:

Zu jedem NEA gibt es einen DEA, der die gleiche Sprache akzeptiert.

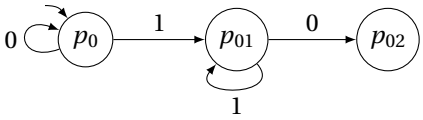
Wir wandeln nun den NEA aus dem letzten Abschnitt um in einen DEA. Bei der Umwandlung zeichnen wir Schritt für Schritt einen DEA. Da der NEA mehrere Anfangszustände haben kann, beginnen wir den DEA mit einem Zustand, den wir so nennen, wie alle Anfangszustände des NEAs kombiniert heißen. In unserem Beispiel ist aber nur q_0 ein Anfangszustand, also nennen wir den neuen Anfangszustand $p_0 = \{q_0\}$.

Da ein NEA beim Lesen eines Zeichens von einem Zustand in verschiedene andere Zustände gelangen kann, nennen wir einen neuen Zustand des DEA so, wie die erreichbaren Zustände des NEA kombiniert heißen. Von q_0 aus kann der NEA mit dem Zeichen 1 in die Zustände q_0 oder q_1 gelangen, also geben wir unserem DEA den Zustand $p_{01} = \{q_0, q_1\}$, der von p_0 aus mit dem Zeichen 1 erreichbar ist. Ebenso sehen wir, dass der NEA von q_0 aus mit dem Zeichen 0 nur wieder nach q_0 gelangen

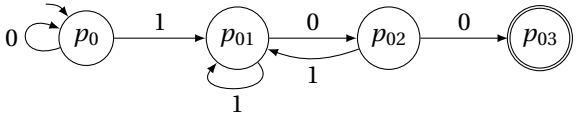
kann. Bis jetzt haben wir



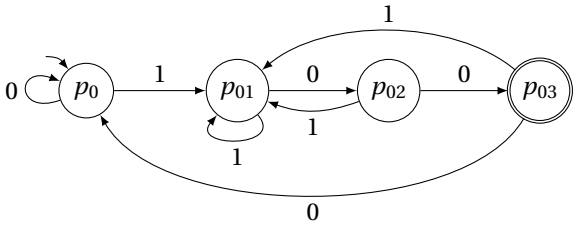
Sehen wir weiter, was von $p_{01} = \{q_0, q_1\}$ aus geschehen kann. Mit einer 1 kommt man wegen q_0 wieder nach q_0 oder q_1, auf Grund von q_1 kommt keine neue Möglichkeit hinzu, weil von dort mit 1 kein Weg herausführt. Diese beiden erreichbaren Zustände haben wir aber schon, nämlich p_{01}. Mit einer 0 kommt man wegen q_0 nach q_0, wegen q_1 aber nach q_2. Diese Kombination haben wir noch nicht und wir brauchen den neuen Knoten p_{02}. So kommen wir bis jetzt zu



Was kann von p_{02} aus geschehen? Der Gedankengang ist fast identisch zum vorigen Schritt. Den neu hinzukommenden Zustand nennen wir $p_{03} = \{q_0, q_3\}$. Dieser ist ein Endzustand, weil er einen Endzustand des NEA enthält! Wir erhalten bis jetzt



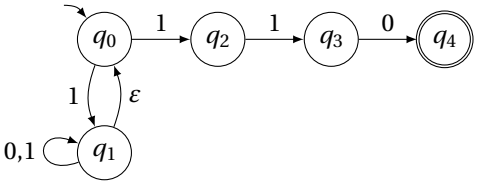
Schließlich untersuchen wir noch, was von p_{03} aus passieren kann. Auch diesmal kommen keine neuen Zustände dazu. Das Ergebnis ist



11.6.3 ε-NEA

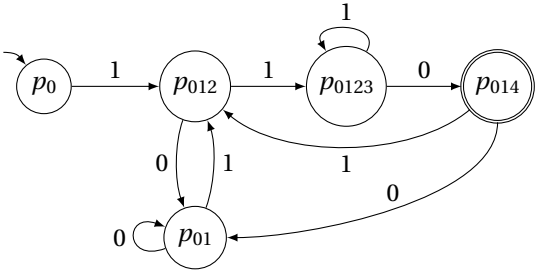
Es gibt praktische und theoretische Situationen, wo sogar die geringen Anforderungen, die ein NEA stellt, noch zu hoch sind. Im folgenden Beispiel wird zusätzlich erlaubt, dass der NEA spontan (ohne Lesen eines Zeichens) von einem in den anderen Zustand springt. Man könnte dafür die Beschriftung des Übergangspfeils einfach weglassen. Sicherheitshalber notiert man aber ein ε. Anlass sei ein Akzeptor für die Sprache aller Binärzahlen, die mit einer 1 beginnen und mit 110 enden. 110 selber ist

so eine Zahl. Bei der Zeichnung eines möglichen NEA kommt man schnell auf folgende seltsame Idee:



Jedoch auch einen NEA mit ε-Übergängen („Einfach-so-Übergänge“) kann man mit der vorher gezeigten Methode in einen DEA umwandeln. Man muss nur darauf achten, dass in die Benennung der neuen Knoten alle Namen einfließen, die erreichbar sind – einschließlich auf dem Wege eines ε-Übergangs!

Versichern Sie sich bitte, dass Sie mit dem vorher gezeigten Verfahren folgenden DEA erhalten!



11.6.4 Reguläre Grammatik → NEA

Weiter oben wurde gezeigt, wie man einen DEA in eine reguläre Grammatik umwandelt. Der umgekehrte Weg ist nicht immer ganz so einfach. Mit Hilfe eines NEAs geht es aber immer, wenn man folgendermaßen vorgeht:

- Die Zustandsmenge ist $Q = V \cup \{q_A\}$, also gerade die Menge aller Nonterminale zuzüglich eines Endzustands q_A.
- Die Menge der Startzustände ist einfach $Q_0 = \{S\}$, weil jede Grammatik nur eine einzige Startvariable hat.
- Die Endzustandsmenge A ist abhängig davon, ob die Grammatik die Regel $S \to \varepsilon$ enthält oder nicht. Man nehme:

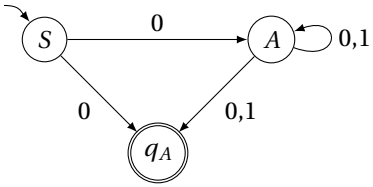
$$A = \begin{cases} \{S, q_A\} : \text{falls } S \to \varepsilon \\ \{q_A\} \quad\quad : \text{sonst} \end{cases}$$

- Zu jeder Regel der Form $X \to a$ bekommt der NEA einen Übergang von X zu q_A für das Lesen des Zeichens a. Zu jeder Regel der Form $X \to aY$ bekommt der NEA einen Übergang von X zu Y für das Lesen des Zeichens a.

Wir führen die Prozedur durch am Beispiel einer Grammatik mit den Regeln

$$S \to 0 \mid 0A, \qquad A \to 0 \mid 1 \mid 0A \mid 1A.$$

Diese erzeugt die Sprache bestehend aus allen Wörtern, die mit einer 0 beginnen. Der daraus konstruierte NEA hat die Gestalt:

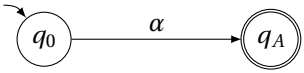


11.6.5 DEA → RegEx

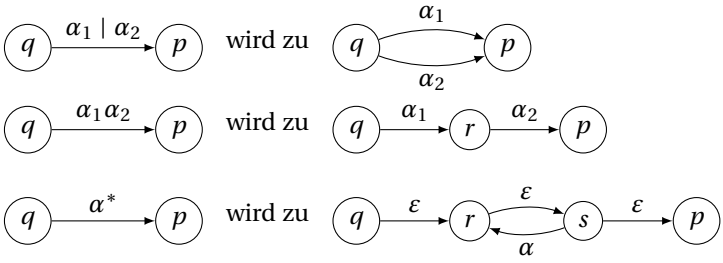
Einen DEA in eine RegEx umzuwandeln, ist oft recht schwierig. Das Thema ist ein fortgeschrittenes, das in einer späteren Version des Skripts ausgeführt wird. Für den Moment verlassen wir uns gegebenenfalls auf unsere Kreativität und finden eine Lösung (nicht-deterministisch sozusagen).

11.6.6 RegEx → NEA

Die Umwandlung von der RegEx zum NEA ist im Grunde ganz einfach. Da aber oft ε-Übergänge gebraucht werden, ist die nachfolgende Umwandlung in einen DEA meist ziemlich ausufernd. Man zeichnet sich eine Folge von Pseudo-NEAs auf, auf deren Übergängen reguläre Ausdrücke stehen. Diese vereinfacht man im nächsten Schritt. Man macht so viele Schritte, bis nur noch Zeichen aus Σ oder ein ε an den Übergängen stehen. Der letzte Graph ist also ein wirklicher NEA. Beginne mit

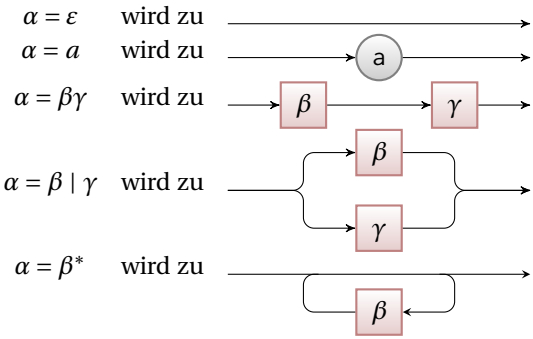


Wende nun bei jedem Schritt eine der folgenden Regeln an:

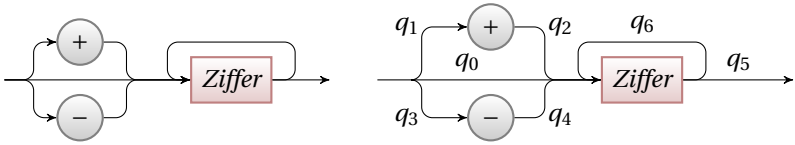


11.6.7 RegEx → Syntaxdiagramm → NEA

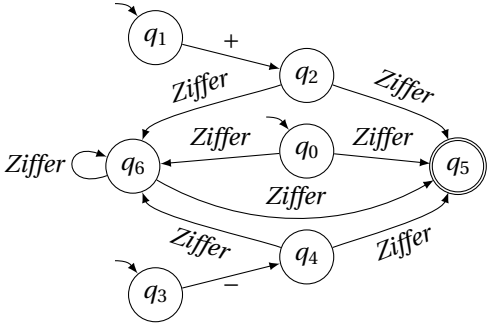
Gelegentlich will man die RegEx in ein Syntaxdiagramm umwandeln. Das ist recht einfach. Außerdem kann man dann das Syntaxdiagramm in einen NEA umwandeln, was aber in etwa die gleiche Arbeit macht, wie das im vorigen Abschnitt besprochene Verfahren. Gehen wir also aus von einem nicht-leeren regulären Ausdruck α, und halten uns an folgende sofort einleuchtende Regeln:



Sehen wir uns das am Beispiel ganzer Zahlen an, die der RegEx $(\varepsilon \mid + \mid -)\mathit{Ziffer}\mathit{Ziffer}^*$ genügen. (*Ziffer* wurde nicht weiter aufgeschlüsselt, weil das keinen Erkenntnisgewinn brächte.) Der linke Teil des folgenden Bildes ist das Syntaxdiagramm.



Die Umwandlung eines Syntaxdiagramms in einen NEA ist im Wesentlichen nur ein Wechsel des Standpunktes. Man betrachtet einfach die Kästchen als Übergänge und die Kanten des Diagramms als Zustände; man geht zum sog. *dualen Graphen* über (rechter Teil des Bildes). Zeichnet man das etwas schöner, so hat man schon den NEA mit den Anfangszuständen q_0, q_1 und q_3 und dem Endzustand q_5.



Man sieht, dass es so zwar geht, es aber kein vernünftiger Mensch so machen würde. Da knobelt man lieber ein bisschen und findet eine viel kürzere Lösung. Nach dem Schema geht man nur vor, wenn man gar nicht mehr weiter weiß.

11.6.8 Zusammenfassung

Wir haben in diesem Kapitel erkannt, dass es verschiedene Darstellungsformen für reguläre Sprachen gibt. Eine reguläre Sprache kann dargestellt werden durch

- eine reguläre Grammatik
- einen akzeptierenden DEA
- einen akzeptierenden NEA (evtl. mit ε-Übergängen)
- einen regulären Ausdruck

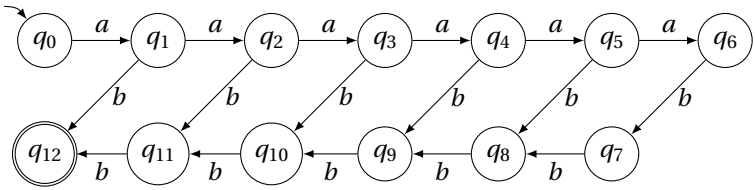
Alle diese Formen können ineinander umgewandelt werden. Die Erstellung der RegEx aus anderen Darstellungsformen haben wir nicht untersucht. Überhaupt kommt man oftmals durch Knobeln schneller ans Ziel als durch die formalisierten Verfahren.

Syntaxdiagramme sind mächtiger als reguläre Sprachen. Deshalb kann jede reguläre Sprache in ein Syntaxdiagramm überführt werden, aber nicht umgekehrt.

11.7 Pumping Lemma

Endliche Automaten gehen bei der Abarbeitung eines Wortes durch verschiedene Zustände. Die Tatsache, dass sich ein Automat in einem bestimmten Zustand befindet, ist eine gespeicherte Information, die der Automat zum korrekten Funktionieren braucht. Ein Akzeptor für Kommazahlen etwa muss wissen, ob das Komma schon gelesen ist, um ein zweites ablehnen zu können. Auf jeden Fall aber ist die Menge der Informationen, die ein DEA speichern kann, durch die Anzahl der endlich vielen Zustände stark beschränkt.

Als sehr einfaches Beispiel suchen wir einen DEA, der die Sprache ab, $aabb$, $aaabbb$, …, a^6b^6 entscheidet[1]. Als Lösung kommt man sehr schnell auf



und man sieht ein, dass es keinen DEA geben kann, der die Sprache $a^n b^n$ ($n \in \mathbb{N}$) erkennt, weil er dazu beliebig weit müsste zählen können, und das kann er mit endlich vielen Zuständen nicht.

Das *Pumping Lemma für endliche Automaten* ist eine kleine, aber hilfreiche Erkenntnis, die dieses Beispiel konsequent zu Ende denkt. Das Lemma lautet:

[1] Wenn ein Automat alle Wörter einer Sprache akzeptiert und zudem alle anderen ablehnt, so sagt man, er *entscheidet* oder *erkennt* die Sprache.

> Wenn ein DEA ein Wort akzeptiert, das mindestens so viele Zeichen hat wie der Automat Zustände hat, dann musste er dafür eine Schleife durchlaufen. Dann akzeptiert er auch die Wörter, die diese Schleife weniger oft oder öfter durchlaufen.

Diese Erkenntnis macht man sich zunutze, wenn man eine Sprache vorliegen hat und nachweisen möchte, dass es *keinen* DEA gibt, der sie entscheidet. Den Beweis, dass es keinen DEA gibt, erbringt man mit dem Pumping Lemma in folgender Weise:

1. In Form eines Widerspruchsbeweises denkt man sich einen Gegner, der behauptet, es gebe einen DEA für unsere Sprache und der habe z Zustände. Über z wissen wir, dass es vielleicht groß, aber jedenfalls endlich ist.
2. Wir suchen schlau ein Wort aus der Sprache, das mindestens Länge z hat. Der Gegner muss zugeben, dass ein Stück unseres Wortes dann aus einer Schleife entstanden ist.
3. Wenn wir nun diese Schleife wiederholen, entstehen längere Wörter. Wenn wir das ursprüngliche Wort tatsächlich schlau gewählt haben, gehören die längeren Wörter nun aber nicht zur Sprache und der Gegner muss zugeben, dass seine Behauptung, es gebe einen DEA, falsch war.

Bei diesem Gedankengang gibt es ein paar Feinheiten zu beachten:

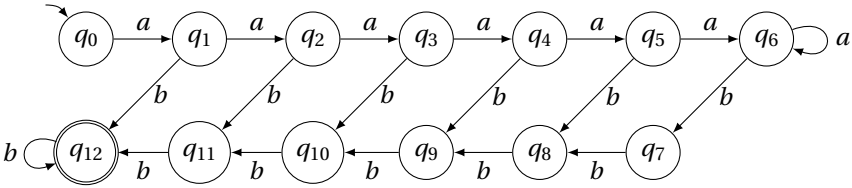
- Wenn wir kein solches schlaues Wort finden, heißt das noch lange nicht, dass es sicher einen DEA für die Sprache gibt!
- Hätten wir einen wirklichen Gegner, so würden wir ihn einfach bitten, seinen DEA vorzuzeigen und wir würden ihm zeigen, wo er einen Denkfehler gemacht hat. Falls er uns seinen DEA nicht zeigen wollte (vielleicht aus patentrechtlichen Gründen), so würde er uns wenigstens ein bestimmtes z nennen und wir wüssten, mit welcher Zahl wir zu argumentieren haben, z. B. $z = 10$. Das kann man sich leichter vorstellen. Da wir uns den Gegner aber nur denken, muss unsere Argumentation so beschaffen sein, dass sie für ein nicht genanntes z gilt. Das macht die Argumentation etwas unübersichtlich.

Der folgende Beweis zeigt, dass es keinen DEA gibt, der $\{a^n b^n : n \in \mathbb{N}\}$ entscheidet und geht dabei so vor, wie oben beschrieben:

Stellen wir uns also einen Gegner vor, der behauptet, es gebe einen DEA, der es tut und der habe z Zustände. Dann nehmen wir als schlaues Wort $a^z b^z$. Dieses Wort gehört zur Sprache und hat die schlaue (für den Gegner fiese) Eigenschaft, dass es so lang ist, dass schon im a-Teil eine Schleife durchlaufen werden musste. Es gibt also eine bestimmte Anzahl von a (sagen wir k Stück), die man noch anhängen könnte, und man wäre im gleichen Zustand wie nach a^z. Wenn der DEA also $a^z b^z$ akzeptiert hat, wird er auch $a^{z+k} b^z$ akzeptieren und dieses Wort gehört nicht zur Sprache. Ätsch, es konnte also keinen DEA mit nur z Zuständen geben.

Die Idee ist bildlich sehr anschaulich. Der Gegner, der da behauptet, er habe

einen geeigneten DEA konstruiert, hatte vielleicht folgende Idee im Kopf:



Dieser DEA akzeptiert tatsächlich alle Wörter a^7b^7, a^8b^8, a^9b^9, …. Er hat aber den Makel, dass er a^6b^7, a^8b^7, a^9b^7, … nicht ablehnt. Wenn wir einen konkreten nicht funktionierenden DEA wie diesen vorgelegt bekommen, können wir natürlich sofort auf das Problem zeigen, ganz ohne Verwendung des Pumping Lemmas. Wenn wir aber alle denkbaren DEAs aller denkbaren Gegner auf einen Schlag als sinnlos bloßstellen wollen, müssen wir abstrakt argumentieren und da kann das Pumping Lemma gute Dienste leisten.

11.8 Aufgaben

1. Zeichnen Sie einen DEA (mit Ausgabe), der einen Getränkeautomaten simuliert nach folgenden Regeln:

   Mögliche Eingaben sind f (0,50€), e (1€), ℓ (Limo-Taste), c (Cola-Taste), k (Korrekturtaste). Mögliche Ausgaben sind f und e (bei Geldrückgabe), ℓ (Limo), c (Cola) oder gar nichts. Mit ℓ oder c kann ein Getränk gewählt werden, aber nur wenn der Betrag 1,50€ erreicht war, wird auch eines ausgegeben. Wird versucht, mehr als 1,50€ einzuwerfen, wird die Münze sofort wieder ausgegeben. Drücken von k wirft jederzeit das bisher gezahlte Geld wieder aus.

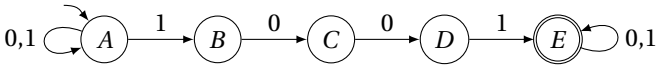
   Geben Sie den Automaten auch in Tabellenform an.

2. Beschreiben Sie eine Fahrstuhlsteuerung als endlichen Automaten. Angefahren werden drei Stockwerke. Höchste Priorität hat immer die Anforderung aus dem Stockwerk, in dem sich der Fahrstuhl gerade befindet. Liegen Anforderungen aus einem höheren und einem niedrigeren Stockwerk vor, so hat die Anforderung Priorität, die in der momentanen Fahrtrichtung liegt. Steht der Fahrstuhl, so hat das Erdgeschoss Priorität. Nach Erreichen des nächsten Stockwerks hält der Fahrstuhl und die Anlage überprüft erneut die Anforderungen.

3. Geben Sie einen DEA an, der genau die Wörter mit geradzahlig vielen Einsen und geradzahlig vielen Nullen akzeptiert. (Wenn ein Automat alle Wörter einer Sprache akzeptiert und alle anderen ablehnt, sagt man auch, er *entscheidet* die Sprache.) Geben Sie die zugehörige Grammatik an.

4. Gegeben ist die Sprache $L = \{a^i c b^j : i, j \in \mathbb{N}\}$. Geben Sie einen DEA an, der sie entscheidet.

5. Gegeben ist die Sprache $L = \{w \in \{a,b\}^* : (anz(w,a) - anz(w,b))\%4 = 3\}$. Geben Sie einen DEA an, der sie entscheidet. Zur Sprache gehören z. B. die Wörter *aaa*, *b*, *aaaaaaa*, *aabaa*, *bbbbb*.
6. Zeichnen Sie einen DEA, der genau die Wörter aus den Buchstaben *a*, *b* und *c* akzeptiert, die mit *abc* anfangen und mit *abc* enden. Zeichnen Sie auch den Fangzustand ein, den wir sonst immer weglassen.
7. Zeichnen Sie einen DEA, der genau die Wörter aus den Buchstaben *a*, *b* und *c* akzeptiert, die nicht mit *abc* anfangen oder nicht mit *abc* enden.
8. Gegeben ist die Sprache $L = \{w \in \{a,b\}^* : anz(w,a) \geq 4, anz(w,b)\%2 = 0\}$ Geben Sie einen DEA an, der sie entscheidet.
9. Berechnen Sie im Dezimalzahl-DEA $\delta(q_0, 10.3)$, $\delta(q_0, -0)$ und $\delta(q_0, 8+4)$
10. Wandeln Sie den Hund in eine reguläre Grammatik um. Welches Alphabet verarbeitet die Grammatik und welche Sprache akzeptiert sie?
11. Schreiben Sie ein Javaprogramm, das zwei Argumente nimmt. Das erste soll die Definition eines Automaten sein, das zweite ein zu testendes Wort. Das Programm soll ausgeben, ob der Automat das Wort akzeptiert oder nicht.
12. Geben Sie die Übergangsfunktion des NEAs zur Mustererkennung in 11.5 mathematisch korrekt an.
13. Erstellen Sie einen NEA über $\Sigma = \{0,1\}$, der genau die Wörter akzeptiert, in denen die Ziffernkombinationen 000 oder 010 vorkommen. Wandeln Sie ihn dann in einen DEA um.
14. Geben Sie einen DEA an, der genau die Wörter akzeptiert, bei denen nach einem *a* genau ein *b* kommt (oder die kein *a* haben).
15. Gegeben ist ein NEA durch das Zustandsdiagramm



    Prüfen Sie, ob 10110010 akzeptiert wird. Geben Sie den Lauf an.
16. Die Umwandlung von einer regulären Grammatik in einen NEA ist höchst einleuchtend für den Fall dass die Grammatik rechtsregulär ist. Wie hat man aber vorzugehen bei linksregulären Grammatiken? Tip: Linksreguläre Grammatiken beschreiben ein Wort von hinten her. Wenn das Wort akzeptiert ist (oder abgelehnt), ist man ganz vorne angelangt. Schreiben Sie also einen NEA auf, der diesen falsch herum laufenden Vorgang durchführt und denken Sie dann ein bisschen nach.
17. Geben Sie einen DEA an, der über $\Sigma = \{a,b,c\}$ die Sprache aller Wörter entscheidet, die auf *acc* enden. Wie lautet der zugehörige reguläre Ausdruck?
18. Konstruieren sie einen DEA, der $(0+11)0^*$ akzeptiert.
19. Geben Sie einen Akzeptor für Wörter an, die drei Vokale in alphabetischer Reihenfolge enthalten, z. B. *herzinsuffizienz*.

20. Als ziemlich ausufernd erweist sich die Umwandlung eines NEAs, der die Wörter akzeptiert, die an n-letzter Stelle eine 1 haben. Stellen Sie also einen NEA auf, der die Wörter akzeptiert, die an der dritten Stelle von hinten eine 1 haben und wandeln Sie ihn um in einen DEA.
21. In der letzten Regel von Abschnitt 11.6.6 scheint nur das mittlere ε nötig. Oft stimmt das auch! Finden Sie Fälle, wo es allein nicht reicht.
22. Erzeugen Sie einen DEA für ganze Zahlen auf drei Weisen: So, wie in 11.6.6 beschrieben, so wie in 11.6.7 und durch naives Knobeln.

12 Kontextfreie Sprachen

12.1 Kellerautomat NKA

Die nächst komplexere Automatenform ist der PDA (pushdown automaton), auch Stackautomat oder Kellerautomat DKA/NKA genannt. Mit Kellerautomaten entscheidet man genau die kontextfreien Sprachen, wenn man sie so definiert, dass sie genau die richtige Stärke haben. Die wesentliche Verbesserung gegenüber dem DEA ist ein einfacher, prinzipiell unendlich großer Speicher in Form eines Stacks, dessen oberstes Element der Automat bei jedem Schritt verändern kann. Die formell genaue Definition lautet:

12.1.1 Definition

Ein Kellerautomat ist ein 7-Tupel $\mathcal{K} = (Q, \Sigma, \Gamma, \delta, q_0, \#, E)$ mit einer endlichen Menge Q von Zuständen, einem Eingabealphabet Σ und einem Kelleralphabet Γ. Die Wörter, die der Automat verarbeitet, bestehen also aus Zeichen der Menge Σ und die Zeichen, die auf dem Stack abgelegt werden, sind aus Γ. Die beiden Mengen dürfen auch gleich sein. $q_0 \in Q$ ist der Startzustand und $\# \in \Gamma$ ist ein Zeichen, das anfangs auf dem Stack liegt (Kellerboden). E ist die Menge von Endzuständen, die auch leer sein kann.

Solange noch Buchstaben zu verarbeiten sind, vernichtet der Automat den aktuellen und das oberste Stackzeichen. (Ein möglicher Sonderfall wird weiter unten noch ausgeführt.) Gibt es kein oberstes Stackzeichen mehr, so bleibt er stehen – ob akzeptierend oder ablehnend, wird noch zu klären sein. Aus Buchstabe, Stackzeichen und aktuellem Zustand entscheidet der Automat, in welchen neuen Zustand er geht *und* was er auf den Stack legt. (Bei dem, was er ablegt, herrscht große Freiheit. Es reicht von nichts bis endlich viele Stackzeichen.)

Wenn der Automat stehen bleibt und es sind noch Buchstaben vorhanden, so gilt das Wort als abgelehnt. Hat der Automat Endzustände, so gilt das Wort genau dann als akzeptiert, wenn er sich am Ende in einem solchen befindet (egal was noch auf dem Stack liegt). Hat er keine Endzustände, so gilt ein Wort genau dann als akzeptiert, wenn am Ende der Stack ganz leer ist.

Deterministische Kellerautomaten mit Endzuständen sind zwar intellektuell befriedigend, aber leider gibt es solche nicht für alle kontextfreien Sprachen, sodass man auf Determinismus gelegentlich verzichten muss. In manchen Fällen muss man dem Automaten einen Übergang aus dem aktuellen Zustand auch ohne Verarbeitung des nächsten Buchstaben erlauben. (Das oberste Stackzeichen wird aber in jedem Fall entnommen.)

Da der Automat nicht-deterministisch sein kann, gilt für die Übergangsfunktion

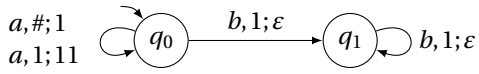
$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to 2^{Q \times \Gamma^*}$$

Die Schreibweise $\delta(q, a, g) \ni (q', g_1 \dots g_k)$ bedeutet, dass der Automat vom Zustand q durch Lesen des Zeichens a bei Vorliegen von g als oberstem Stackzeichen in den Zustand q' gehen *kann* und dabei das Zeichen g ersetzt durch das Wort $g_1 \dots g_k$. (Das geht auch, wenn $g = \varepsilon$.) Gibt man nicht nur eine Möglichkeit an, sondern alle, so schreibt man die Möglichkeiten als Menge auf und verwendet das Zeichen =, insbesondere dann, wenn es nur eine Übergangsmöglichkeit gibt $\delta(q, a, g) = \{(q', g_1 \dots g_k)\}$.

In diesem Beispiel wurde g als oberstes Stackzeichen entfernt und ersetzt durch $g_1 \dots g_k$. Damit ist für den nächsten Schritt g_1 das oberste Stackzeichen. Es ist ausdrücklich auch der Fall $k = 0$ erlaubt, also einfach die Entfernung (pop) des obersten Zeichens. Man schreibt dann (q', ε) und meint damit nicht, dass der Stack leer ist, sondern nur, dass das oberste Zeichen durch nichts ersetzt wurde. Ebenso möglich ist ein push von g_1, indem man g ersetzt durch $g_1 g$. Diesen Fall notiert man als $(q', g_1 g)$. Schließlich sei noch erwähnt, dass $\delta(q, \varepsilon, g) \ni (q', g_1 \dots g_k)$ den Übergang und die Stackveränderung bewirkt, ohne ein Eingabezeichen zu lesen. Wegen dieser ε-Übergänge macht die Beschränkung auf nur einen Startzustand keine Schwierigkeiten.

12.1.2 Graphische Darstellung

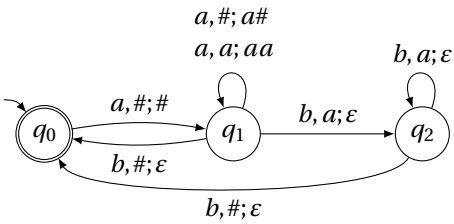
Viel prägnanter als die ganzen Schreibweisen ist wieder die graphische Darstellung. Nehmen wir als Beispiel den DKA, der die Sprache $a^n b^n$ entscheidet.



Sofort erkennt man die Menge der Zustände, den Startzustand und die Übergangsfunktion, in deren Beschreibung nur noch eingelesenes Zeichen, Stackzeichen und neues Stackzeichen genannt werden. Dass # das Start-Stackzeichen ist, kann man leicht erraten. Dieser Automat akzeptiert nicht das leere Wort, weil beim leeren Wort das Zeichen # auf dem Stack liegen bleibt und ein nicht-leerer Stack für ein Verwerfen des Wortes steht. Wollte man auch das leere Wort akzeptieren, so müsste eine Regel des Startzustands lauten $\varepsilon, \#; \varepsilon$, womit aber der Determinismus verloren wäre, weil z. B. ab nun einen akzeptierenden und einen verwerfenden Lauf hätte.

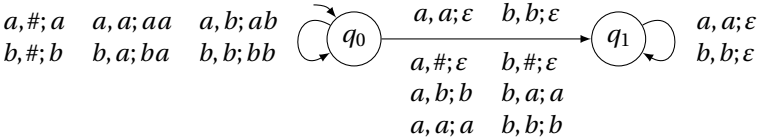
Der folgende Kellerautomat tut das Gleiche, ist deterministisch mit Endzustän-

den und akzeptiert auch das leere Wort.



Verifizieren Sie, dass er nicht *aaabbbab* akzeptiert.

Im nun folgenden, schwierigeren Fall ist es unmöglich, einen deterministischen Automaten anzugeben. Ein NKA soll entscheiden, ob ein Wort ein Palindrom über $\Sigma = \{a, b\}$ ist. Nicht-Determinismus ist hier essenziell, weil der Automat die Mitte des Wortes erraten muss.



Die linken Regeln sorgen dafür, dass bei Wörtern aus mehr als einem Zeichen die erste Hälfte auf den Stack gepusht werden kann, die rechten dafür, dass sie wieder gepopt werden, wenn die Zeichenreihenfolge stimmt. Die oberen Regeln in der Mitte bewirken, dass bei geradzahliger Anzahl von Zeichen die beiden mittleren Zeichen gepaart werden, die unteren, dass bei ungeradzahliger Anzahl das mittlere Zeichen folgenlos gelesen wird.

12.1.3 Konfigurationen

Konfiguration steht nur im weitesten Sinne für so etwas wie Einstellung eines Automaten. Eine Konfiguration ist vielmehr eine Momentaufnahme des Automaten bei der Arbeit. Sie zeigt den Zustand, in dem der NKA gerade ist, welche Zeichen er noch zu lesen hat und was der gesamte Stackinhalt ist. Jede Konfiguration ist also ein Tripel aus der Menge $Q \times \Sigma^* \times \Gamma^*$. Will man niederschreiben, wie ein NKA ein Wort abarbeitet, so tut man dies am besten als Folge von Konfigurationen, getrennt durch das Zeichen $\vdash$.

Als Beispiel schreiben wir auf, wie das Palindrom *abba* akzeptiert wird:

$$(q_0, abba, \#) \vdash (q_0, bba, a) \vdash (q_0, ba, ba) \vdash (q_1, a, a) \vdash (q_1, \varepsilon, \varepsilon)$$

Man kann auch Konfigurationswechsel aufschreiben, die nicht zum Ziel führen. Wichtig ist, dass man damit überhaupt eine brauchbare Möglichkeit hat, Vorgänge

niederzuschreiben. Viel einfacher schreibt man einen Vorgang für einen DEA auf. Dort reicht, wie schon erwähnt, die erweiterte δ-Notation: $\delta(q_1, hallo) = \delta(q_2, allo) = \ldots$

12.1.4 Kontextfreie Grammatik → NKA

Beide Umwandlungsvorgänge wären manchmal hilfreich. Aber vom NKA zur Grammatik ist im allgemeinen zu schwer und von der Grammatik zum NKA ist praktisch nicht brauchbar. Letzere Umwandlung sehen wir uns aber immerhin an, weil sie ganz einfach ist.

Sei $G = (V, \Sigma, \mathscr{P}, S)$ die kontextfreie Grammatik, die in einen NKA umgewandelt werden soll. Wir wählen $\Gamma = V \cup \Sigma$ und als Kellerboden nehmen wir S. Wir brauchen nur einen einzigen Zustand q und müssen für unseren NKA $(\{q\}, \Sigma, V \cup \Sigma, \delta, q, S, \{\})$ nur noch die Übergangsfunktion δ angeben. Die wählen wir so:

Jede Regel aus $\mathscr{P}$ hat die Form $X \to \xi$ und wir setzen für jede solche Regel

$$\delta(q, \varepsilon, X) \ni (q, \xi)$$

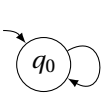
Außerdem setzen wir für jedes $x \in \Sigma$

$$\delta(q, x, x) \ni (q, \varepsilon)$$

D.h. immer wenn das oberste Stackzeichen ein Nichtterminal der Grammatik ist, wird es durch die entsprechende Regel der Grammatik ersetzt, ohne ein Zeichen zu lesen. Und immer wenn das oberste Zeichen ein Terminalzeichen ist und mit dem aktuellen Eingabezeichen übereinstimmt, wird es einfach gepopt.

Dieses Verfahren führt zu NKAs von geringem praktischem Wert, weil sie stark nicht-deterministisch sind. Als Beispiel sehen wir uns an, welcher Automat entsteht, wenn man die Grammatik für die Sprache $a^n b^n$ auf diese Weise in einen NKA umwandelt:

Die Regeln lauten $S \to ab \mid aSb$ und der daraus entstehende NKA



$\varepsilon, S; aSb$
$\varepsilon, S; ab$
$a, a; \varepsilon$
$b, b; \varepsilon$

Für die Ableitung eines Wortes hätte man genauso gut die Grammatik verwenden können. Der Kellerautomat nimmt einem keine Arbeit ab (was nicht verwundert, weil seine Erstellung auch keine Arbeit gemacht hat).

12.2 Mehrdeutige Grammatiken, Ableitungsbäume

Mit Kellerautomaten verarbeitet man kontextfreie Sprachen. Kontextfreie Grammatiken haben auf rechten Seiten von Regeln oft mehrere Variablen. Das führt dazu, dass es für ein Wort[1] mehrere Bedeutungen geben kann – je nachdem, welche Ableitung zu ihm geführt hat. In der menschlichen Sprache kennt man das: Der Sprecher hat eine Idee im Kopf und setzt sie unter korrekter Anwendung der Sprachgrammatik in gesprochenen Text um. Der Hörer bekommt aber nur den Text und nicht die Schritte, die der Sprecher grammatikalisch unternommen hat. Um aus dem Text die ursprüngliche Idee zu rekonstruieren, wendet er die Grammatik blitzschnell rückwärts an. Dumm nur, wenn der Vorgang nicht eindeutig ist.

„Der Hund bedroht den Mann mit dem Hammer." sagt uns, dass ein Hund einen Mann bedroht. Einer von beiden hat einen Hammer. Sehen wir uns die Problematik im Detail an, allerdings an einem interessanteren Beispiel.

Wir betrachten die Grammatik $G_1 = (\{P, S, T, Z\}, \Sigma, \mathscr{P}_1, S)$ mit $\Sigma = \{+, \cdot, (,), 0, \ldots, 9\}$ und den Regeln

$$\mathscr{P}_1 = \left\{ \begin{array}{l} S \to P \mid S + P, \\ P \to T \mid P \cdot T, \\ T \to Z \mid (S), \\ Z \to 0 \mid 0Z \mid \ldots \mid 9 \mid 9Z \end{array} \right\}$$

Dabei bedeuten T = Term, S = Summe, P = Produkt und Z = Zahl. Damit können wir Terme mit Plus und Mal ableiten, z. B. $3 + 4 \cdot 5$.

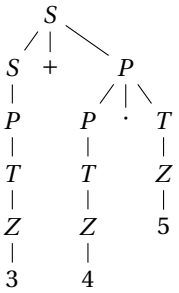
$$\begin{aligned} S \quad & \Rightarrow S + P \Rightarrow P + P \Rightarrow T + P \Rightarrow Z + P \Rightarrow a + P \Rightarrow 3 + P \cdot T \\ & \Rightarrow 3 + T \cdot T \Rightarrow 3 + Z \cdot T \Rightarrow 3 + 4 \cdot T \Rightarrow 3 + 4 \cdot Z \Rightarrow 3 + 4 \cdot 5 \end{aligned}$$

So etwas nennt man eine *Linksableitung*, weil immer das am weitesten links stehende Nicht-Terminal ersetzt wird. Entsprechend gibt es den Begriff der *Rechtsableitung*. (Für Durcheinander gibt es keinen eigenen Begriff.) Das ganze ist aber jedenfalls recht unübersichtlich, weshalb man den Syntaxbaum erfunden hat (siehe Bild).

Verfolgen Sie die Ableitung Schritt für Schritt mit dem Finger entlang des Ableitungsbaumes, um zu erkennen, wie er entstanden ist. Das abgeleitete Wort findet man im Ableitungsbaum, wenn man die Blätter von links nach rechts liest.

Machen Sie nun eine Rechtsableitung des obigen Terms und erstellen Sie daraus wieder einen Ableitungsbaum! Vergewissern Sie sich, dass er genauso aussieht!

[1] Wir bewegen uns hier in einem Bereich, wo man beim Begriff *Wort* lieber an einen ganzen *Satz* denken sollte.



Ob man von links oder rechts beginnt, ändert also nichts an der Struktur des Ableitungsbaumes. Versuchen wir es mit einer anderen Grammatik, die die gleiche Sprache erzeugt (Nachweis!): $G_2 = (\{Z, T\}, \Sigma, \mathscr{P}_2, T)$

$$\mathscr{P}_2 = \left\{ \begin{array}{l} T \to Z \mid (T) \mid T + T \mid T \cdot T \\ Z \to 0 \mid 0Z \mid \ldots \mid 9 \mid 9Z \end{array} \right\}$$

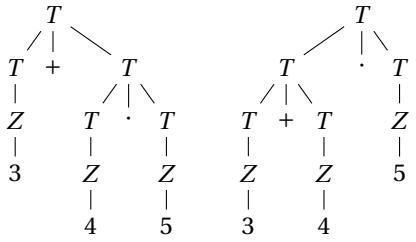
Dafür, dass G_2 so kurz und freundlich daher kommt, zahlt man einen hohen Preis. Für unseren Term $a + b \cdot c$ gibt es zwei strukturell verschiedene Ableitungen. (Es sind sogar beides Linksableitungen.)

$$\begin{aligned} T &\Rightarrow T + T \Rightarrow Z + T \Rightarrow 3 + T \Rightarrow 3 + T \cdot T \Rightarrow 3 + Z \cdot T \\ &\Rightarrow 3 + 4 \cdot T \Rightarrow 3 + 4 \cdot Z \Rightarrow 3 + 4 \cdot 5 \end{aligned}$$

und

$$\begin{aligned} T &\Rightarrow T \cdot T \Rightarrow T + T \cdot T \Rightarrow Z + T \cdot T \Rightarrow 3 + T \cdot T \Rightarrow 3 + Z \cdot T \\ &\Rightarrow 3 + 4 \cdot T \Rightarrow 3 + 4 \cdot Z \Rightarrow 3 + 4 \cdot 5 \end{aligned}$$

Wenn man sich die beiden Ableitungsbäume ansieht, sieht man die Verwandschaft mit dem Hund und dem Mann und dem Hammer.



Die linke Ableitung ist äquivalent zur obigen. Bei ihr geht Multiplikation vor Addition. Bei der rechten ist es umgekehrt. Schlecht ist nicht die Tatsache, dass die zweite Ableitung die Addition vor der Multiplikation durchführt, sondern dass die Grammatik beide Lesarten zulässt.

12.2.1 Definitionen

Eine Grammatik heißt *mehrdeutig*, wenn es mindestens ein Wort mit zwei verschiedenen Ableitungsbäumen gibt. Andernfalls heißt sie *eindeutig*.

Die gleiche Sprache kann eindeutige und mehrdeutige Grammatiken haben. Eine Sprache heißt *inhärent mehrdeutig*, wenn es nur mehrdeutige Grammatiken für sie gibt.

Eine inhärent mehrdeutige Sprache ist $\{a^i b^j c^k : i = j \vee j = k\}$. Der Beweis ist aber sehr schwierig.

12.3 Parser

Kellerautomaten sind in der Lage zu entscheiden, ob ein Wort zu einer kontextfreien Grammatik gehört oder nicht. Um die Bedeutung von Wörtern einer kontextfreien Grammatik zu erfassen, reicht es aber nicht aus zu wissen, dass ein Wort von der Grammatik erzeugt wird oder nicht, es ist vielmehr von Bedeutung, *wie* das Wort erzeugt wird. Dazu ist jedoch ein Kellerautomat nicht das geeignete Werkzeug.

Ein Parser ist ein Mechanismus, der wie der Kellerautomat ein Wort einer kontextfreien Grammatik einliest, der aber beim Abarbeiten die Struktur des Ableitungsbaums erfasst und aus der Kenntnis dieser Struktur mehr erreicht, als nur die Feststellung, ob es sich um ein korrektes Wort der Grammatik handelt oder nicht.

Wir wollen als Beispiel einen Parser bauen, der algebraische Terme mit Klammern, Zahlen und den vier Grundrechenoperatoren liest und am Ende das Ergebnis des Terms ausgibt. Dazu verwenden wir die obige Grammatik G_1, weil sie nicht mehrdeutig ist.

Das Programm bekommt in der `main`-Methode erzeugt eine Instanz `calc` der Klasse `InfixCalculation`, holt die Rechnung als String `aufg` von der Tastatur und ruft die Methode `calc.ergebnis()` auf, die das Ergebnis als double zurück liefert.

Da der Berechnungsvorgang start rekursiv abläuft, wird der String mit der Rechnung nicht jedesmal als Argument mitgegeben, sondern zentral in einem Attribut vorrätig gehalten. Der Einstiegspunkt ist die Methode `summe`. Im folgenden Listing sind neben einigen Hilfsmethoden die vier Methoden erst einmal grob angelegt.

```
public class InfixCalculation{
  private String rechnung; // die vollständige Rechnung
  private    int pos;      // Position des nächsten Zeichens
  private   char aktuell;  // das aktuelle Zeichen

  // Stellt einen gewöhnlichen Infix-Ausdruck mit den vier
  // Grundrechenarten und Klammern dar, z.B. 3*(-(2+4))
  public InfixCalculation(String rechnung){
    this.rechnung=rechnung+"_";//hinten ein Stopzeichen
  }

  // Rechnet einmal durch
  public double berechne(){
    pos=0;
    weiter();
    return summe();
```

```
  }

  private void weiter(){ aktuell=rechnung.charAt(pos++); }

  private boolean ziffer(){return aktuell>='0' && aktuell<='9';}

  private double zahl(){ return 1; // Unsinn, aber compilierbar.
  }

  // TERM -->  ZAHL | (SUMME)
  private double term(){ return 1; // Unsinn, aber compilierbar.
  }

  // PRODUKT --> TERM | TERM*TERM* ... | TERM/TERM/ ...
  //           | (auch * und / gemischt)
  private double produkt(){ return 1; // Unsinn, aber compilierbar.
  }

  // SUMME --> +PRODUKT | -PRODUKT  (evtl. führende Vorzeichen)
  //         | PRODUKT
  //         | PRODUKT+PRODUKT+ ...
  //         | PRODUKT-PRODUKT- ...
  //         | (auch + und - gemischt)
  private double summe(){ return 1; // Unsinn, aber compilierbar.
  }

  public static void main(String[] args) throws Exception{
    if(args.length!=1)
      System.out.println("Tip: java InfixCalculation \"3*(4-7)\"");
    else{
      InfixCalculation calc=new InfixCalculation(args[0]);
      System.out.println(args[0]+" = "+calc.berechne());
    }
  }
}
```

12.4 Aufgaben

1. Geben Sie einen DKA an, der die Sprache der Palindrome entscheidet, die aus den Zeichen a und b mit einem einzigen c in der Mitte bestehen. (Beispiel: $aabcbaa$)
2. Geben Sie einen DKA an, der korrekte Klammerausdrücke entscheidet.

3. Geben Sie einen DKA an, der die Sprache der Wörter mit gleich vielen a und b entscheidet.
4. Geben Sie je einen DKA zur Entscheidung folgender Sprachen an

$$L_1 = \{a^i b^{i+j} c^j : i + j \geq 1\} \qquad L_2 = \left\{wc : w \in \{a,b\}^*, anz(w,a) = 2 \cdot anz(w,b)\right\}$$

5. Erklären Sie, wie man notiert, dass ein Kellerautomat das leere Wort akzeptiert?
6. Nennen Sie einen plausiblen Grund, warum die Sprache der Palindrome aus den Zeichen a und b (wahrscheinlich) nicht mit einem DKA entscheidbar ist.
7. Nennen Sie einen plausiblen Grund, warum die Sprache aus Wörtern mit gleich vielen a, b und c (wahrscheinlich) nicht einmal mit einem NKA entscheidbar ist.
8. Wie viele Söhne kann ein Knoten in einem Ableitungsbaum haben?
9. Warum wurde der Begriff *Ableitungsbaum* nicht schon bei regulären Sprachen thematisiert?
10. Kann man jede reguläre Sprache mit einem DKA erkennen?
11. Zeigen Sie, dass die Grammatik mit den Regeln $S \to aB \mid Ac, \quad A \to ab, \quad B \to bc$ mehrdeutig ist und die von ihr erzeugte Sprache nicht inhärent mehrdeutig ist.
12. Gegeben ist der Kellerautomat $A = (\{q_0, q_1, q_2\}, \{a,b\}, \{\#, 1\}, \delta, q_0, \#\}$ mit

$$\delta: \begin{array}{l} (q_0, a, \#) \mapsto (q_0, 1\#) \\ (q_0, a, 1) \mapsto (q_0, 11) \\ (q_0, b, 1) \mapsto (q_1, 1) \end{array} \qquad \begin{array}{l} (q_1, a, 1) \mapsto (q_2, \varepsilon) \\ (q_1, b, 1) \mapsto (q_1, 1) \end{array} \qquad \begin{array}{l} (q_2, a, 1) \mapsto (q_2, \varepsilon) \\ (q_2, \varepsilon, \#) \mapsto (q_2, \varepsilon) \end{array}$$

Stellen Sie den Automaten graphisch dar. Wie deterministisch ist er? Welche Sprache entscheidet er? Finden Sie eine Grammatik für diese Sprache!

13 Typ-1- und Typ-0-Sprachen

Die kontextsensitiven Sprachen (Typ-1) und die rekursiv aufzählbaren (Typ-0) sind so komplex, dass sie nicht einmal von einem nichtdeterministischen Kellerautomaten akzeptiert werden können. Erst Turingmaschinen sind dazu in der Lage. Bei Turingmaschinen kann man zeigen, dass die Menge der nicht-deterministischen nicht mehr Sprachen akzeptieren kann als die Menge der deterministischen. Deshalb werden wir nur DTM behandeln, nicht NTM.

13.1 Turingmaschine

Die Turingmaschine ist benannt nach Alan M. Turing (1912–1954), der damit einen Mechanismus erfinden wollte, der alles das kann, was algorithmisch möglich ist. Was eine Turingmaschine nicht berechnen kann, sollte also grundsätzlich als unberechenbar gelten. Die Fachwelt ist immer noch (und immer stärker) davon überzeugt, dass ihm das gelungen ist.

Bei der Ersinnung seiner Maschine hatte Turing keinen Computer zur Verfügung, weil es die noch nicht gab. Stattdessen ist er davon ausgegangen, was ein Mensch mit Bleistift und Radiergummi auf Papier anstellen kann, um aus einer Vorgabe ein Ergebnis zu ermitteln. Der Einfachheit halber ging er davon aus, dass es sich nicht um eine Fläche Papier sondern nur um einen Streifen handeln sollte, das sog. Band. Dieses sollte aber in beide Richtungen unendlich lang sein[1].

Ein Mensch nimmt die Vorgabe dadurch wahr, dass er links davon anfängt. Nach einem bestimmten, evtl. sehr komplizierten Verfahren kann er ein Zeichen lesen, dadurch angeregt werden, in einen anderen Denkzustand zu gehen, dieses Zeichen ersetzen gegen ein anderes (oder das gleiche) und dann um eine Position nach links oder rechts gehen (oder auch stehen bleiben). Wenn es für den Algorithmus nötig ist, Daten zu speichern, so muss dies ebenfalls auf dem Band geschehen.

13.2 Definition von TM und Konfiguration

Eine Turingmaschine ist ein 7-Tupel $T = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ bestehend aus einer endlichen Zustandsmenge Q, dem Eingabealphabet Σ, dem Arbeitsalphabet $\Gamma \supset \Sigma$, der Übergangsfunktion δ, dem Startzustand $q_0 \in Q$, dem Leerzeichen $\square \in \Gamma \backslash \Sigma$ und der Menge der Endzustände $F \subseteq Q$.

[1] Man kann recht leicht beweisen, dass eine Fläche außer größerer Geschwindigkeit keine Vorteile gegenüber dem Band bringt.

Die Übergangsfunktion $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, N, R\}$ ordnet einer Kombination von Zustand und aktuell gelesenem Zeichen einen neuen Zustand, das geänderte Zeichen und eine Lesekopfbewegung (*L*inks, *N*eutral oder *R*echts) zu. Die Schreibweise $\delta(q, a) = (p, b, R)$ bedeutet also beispielsweise, dass der Automat, falls er sich beim Lesen des Zeichens a im Zustand q befand, danach in den Zustand p übergeht, das a gegen ein b austauscht und dann den Blick um ein Zeichen nach rechts wendet. Wir werden solche Angaben gelegentlich kürzen zu $qabRp$, weil darin die gleiche Information steckt.

Eine TM bleibt erst stehen, wenn sie einen Endzustand erreicht hat oder einen Zustand, aus dem es mit dem aktuellen Zeichen keinen Übergang gibt. Sie kann in manchen Fällen auch unendlich lang laufen. Wenn sie einen Endzustand erreicht hat, ist sie evtl. hin- und hermarschiert und hat das anfängliche Wort auch vielleicht verändert. Wenn wir eine TM als Akzeptor für eine Sprache verwenden, dann betrachten wir deshalb das anfängliche Wort als akzeptiert, falls die TM damit auf irgendeine Weise einen Endzustand erreicht, egal ob das Wort dann noch unzerstört vorliegt. Bei Anwendungen, in denen die TM etwas berechnen soll, markiert am Ende des Vorgangs die Kopfposition den Anfang des Ergebnisses. (In seltenen Fällen gilt der ganze nicht-leere Teil des Bandes als Ergebnis.)

Eine Konfiguration ist auch bei der TM wieder eine Momentaufnahme der Maschine bei der Arbeit. Weil der Kopf aber in beide Richtungen wandern kann, muss immer mindestens der gesamte nicht-leere Bandinhalt, der aktuelle Zustand und die Position des Kopfes angegeben werden. Als Schreibweise verwendet man Wörter aus der Menge $\Gamma^* Q \Gamma^*$. Beispielsweise bedeutet $1q_3001$, dass der aktuelle Bandinhalt 1001 ist, dass der Kopf auf dem zweiten Zeichen steht (der ersten 0) und dass q_3 der aktuelle Zustand ist. Ist der Kopf bis hinter die hinterste 1 gelangt, so schreibt man z. B. $1001q_5\square$. Entsprechend verfährt man am linken Ende des Bandinhalts. Für Konfigurationswechsel, also Abfolgen von Konfigurationen, verwendet man wieder das Zeichen $\vdash$.

13.3 Beispiel: 1-Addierer

Auf dem Band stehe anfangs eine Binärzahl. Gesucht ist eine DTM, die 1 addiert und den Kopf danach links auf den Anfang des Ergebnisses setzt. Wir wählen

$$T = (\{q_1, q_2, q_3, q_0\}, \{0, 1\}, \{0, 1, \square\}, \delta, q_1, \square, \{q_0\})$$

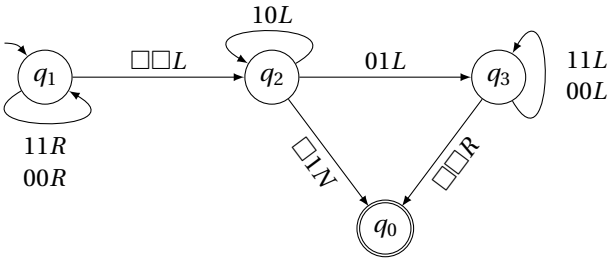
wobei

$$\delta: \begin{array}{lll} (q_1,0) \mapsto (q_1,0,R) & (q_2,0) \mapsto (q_3,1,L) & (q_3,0) \mapsto (q_3,0,L) \\ (q_1,1) \mapsto (q_1,1,R) & (q_2,1) \mapsto (q_2,0,L) & (q_3,1) \mapsto (q_3,1,L) \\ (q_1,\square) \mapsto (q_2,\square,L) & (q_2,\square) \mapsto (q_0,1,N) & (q_3,\square) \mapsto (q_0,\square,R) \end{array}$$

Aus 101 macht diese Maschine 110, wobei der Kopf danach auf der ersten 1 steht. Das geht so

$$q_1101 \vdash 1q_101 \vdash 10q_11 \vdash 101q_1\square \vdash 10q_21 \vdash 1q_200 \vdash q_3110 \vdash q_3\square110 \vdash q_0110$$

Natürlich kann man auch Turingmaschinen in graphischer Darstellung leichter verstehen als durch die bloße Angabe der Übergangsfunktion. Hier nochmal der 1-Addierer.



13.4 Aufgaben zu DTMs

1. Entwickeln Sie DTMs zu den folgenden Problemen. Alles was mit Zahlen zu tun hat, kann man im unären wie im binären System versuchen.
   (a) Zahl löschen (b) Zahl kopieren (c) Spiegeln des Wortes (d) Addition (e) Subtraktion (f) Multiplikation (g) Ganzzahlige Division bzw. Rest (h) Vergleich zweier Zahlen (i) Vertauschen zweier Zahlen (j) ggt(x,y) (k) max(x,y) (ℓ) min(x,y) (m) x^y (n) Wurzel (o) Logarithmus (p) Umwandlung unär $\leftrightarrow$ binär (q) Bubblesort (r) Fibonaccizahlen (s) Primzahlen (t) Ackermannfunktion
2. Entscheiden Sie die Sprache $a^n b^n$ mit einer DTM.
3. Entscheiden Sie die Sprache $a^n b^n c^n$ mit einer DTM.
4. Entscheiden Sie mit einer DTM die Sprache aus den Wörtern mit gleich vielen a, b und c.
5. Entscheiden Sie die Sprache der Palindrome über $\{a,b\}$ mit einer DTM.
6. Schreiben Sie ein Java-Programm, das eine/jede DTM simuliert.

13.5 Universelle Turingmaschine, Church-Turing-These

Turingmaschinen können viel mehr als die anderen besprochenen Automaten. Es ist sogar möglich, eine Turingmaschine U zu bauen, die auf dem Band zwei Argumente bekommt. Das erste ist eine geeignet codierte Turingmaschine T, das andere ein Wort, das T verarbeiten soll. Wenn U fertig damit ist, seine beiden Argumente abzuarbeiten, dann steht auf dem Band das Ergebnis, das T aus dem zweiten Argument erzeugen würde.

Uns wundert das nicht allzu sehr, haben wir doch schon ein Java-Programm geschrieben, das eine beliebige Turingmaschine T simuliert. Wenn man bedenkt, dass

eine Turingmaschine nicht weniger kann als Java[2], dann ist klar, dass es so eine universelle Maschine U geben muss. 1936, als Turing ein U angab, war das aber Neuland des Denkens und eine aufwühlende Erkenntnis. Es war die erste (theoretisch existierende) programmierbare Maschine, die jede andere Maschine simulieren konnte.

Nach den gewissenhaften Studien der von ihm erfundenen Maschine gelangte Turing zu der Ansicht, dass nichts und niemand komplexere symbolische Kalkulationen durchführen kann als eine TM, vorausgesetzt er arbeitet deterministisch. Jedes Problem also, das überhaupt maschinell lösbar ist, ist Turings Ansicht nach von einer TM lösbar. Beweisbar ist diese Aussage allerdings nicht.

Der Logiker Alonzo Church präzisierte etwa zur gleichen Zeit den Begriff der Berechenbarkeit durch sein so genanntes Lambda-Kalkül. Auch Emil Post entwickelte eine Berechenbarkeitstheorie und einen Automatentyp dazu. Nach einer Weile stellte sich heraus, dass all die Definitionen von verschiedenen Denkern gleichwertig sind (die Beweise hierzu sind recht anspruchsvoll). Deshalb *glaubt* man, dass das, was man sich intuitiv unter Berechenbarkeit vorstellt, tatsächlich durch Turingmaschinen darstellbar ist. Die Church-Turing-These kann man so ausdrücken:

> Die Klasse der Turing-berechenbaren Funktionen ist genau die Klasse der intuitiv berechenbaren Funktionen.

Wenn man also nachweisen kann, dass eine TM ein Problem nicht lösen kann, kann es niemand lösen, so glaubt man.

[2] Diese Aussage beweist man im Studium als Hausaufgabe.

14 Berechenbarkeit und Entscheidbarkeit

In diesem Abschnitt besprechen wir Aufgaben, die man Menschen oder Computern stellen kann und überlegen uns, ob die Aufgaben überhaupt in endlicher Zeit lösbar sind. Ob es eine *effiziente* Lösung gibt, wird erst in einem späteren Abschnitt interessieren.

14.1 Berechenbarkeit

Man nennt eine Funktion $f : \Sigma^* \to \Sigma^*$ berechenbar, wenn es eine Turingmaschine gibt, die das gleiche leistet wie f. Das heißt, wenn $f(x) = y$ ist, dann ergibt ein Lauf der TM $q_1 x \vdash^* q_0 y$ (wobei q_1 der Startzustand und q_0 ein Endzustand ist). Wenn $f(x)$ nicht definiert ist (partielle Funktion), darf die Turingmaschine nicht in einem Endzustand anhalten. Sie kann dann in einem anderen Zustand anhalten oder in einer Endlosschleife bleiben.

14.1.1 Die undefinierte Funktion

Diese ist nirgends definiert und ist berechenbar durch die Turingmaschine $(q_1, a) \mapsto (q_1, a, R)$ für alle $a \in \Sigma$. (Beachten Sie, dass das nicht *eine* Regel ist, sondern viele, weil a nicht ein Zeichen ist sondern Platzhalter für alle Zeichen!)

14.1.2 Die π-StartsWith-Funktion

Diese Funktion hat den Wert 1, wenn die Ziffern ihres Arguments die gleichen sind wie die ersten Ziffern von π. Andernfalls hat sie den Wert 0. Es ist also etwa $f(314) = 1$ aber $f(14) = 0$. Da es Berechnungsverfahren für die Zahl π gibt, ist diese Funktion berechenbar.

14.1.3 Die π-Contains-Funktion

Diese Funktion hat den Wert 1, wenn ihre Argument in der Ziffernfolge von π vorkommt. Es ist also etwa $f(14) = 1$. Ob diese Funktion berechenbar ist, ist noch unklar, weil von π noch nicht bekannt ist, ob es alle möglichen Ziffernfolgen beinhaltet (auch wenn im Mathematikum so getan wird, als ob das klar wäre). Man kann zwar π beliebig weit absuchen, aber wenn man eine Zahlenkombination nicht findet, kann sie

ja noch weiter hinten kommen. Wird eines Tages bewiesen, dass jede Ziffernkombination in π vorkommt, ist die Funktion natürlich sehr leicht berechenbar. Die TM muss nur das Band löschen und dann 1 draufschreiben, ohne irgend etwas zu rechnen. Wird jedoch bewiesen, dass manche Ziffernfolgen in π vorkommen und andere nicht, dann könnte diese Funktion nicht berechenbar sein, falls es sich als zu schwierig heraus stellt, das zu unterscheiden.

14.1.4 Die π-Fünferketten-Funktion

Diese Funktion hat den Wert 1, wenn in den Ziffern von π eine Fünferkette der Länge n enthalten ist, also $f(3) = 1$, wenn 555 irgendwo in π vorkommt. Andernfalls ist der Funktionswert 0. Diese Funktion ist lustigerweise sicher berechenbar. Sie ist konstant 1, falls es sich herausstellt, dass in π beliebig lange Fünferketten vorkommen. Wenn es sich herausstellt, dass in π Fünferketten von höchstens der Länge k vorkommen, dann ist die Funktion auch berechenbar. Es gilt dann $f(n) = 0$ für $n > k$ und $f(n) = 1$ für $n \leq k$. Falls sich nichts herausstellt, weil die Menschheit zu dumm ist, die Fakten über Fünferketten herauszufinden, so ist die Funktion doch berechenbar, wir wissen nur nicht, was rauskommt.

14.1.5 Die r-Funktionen

Jede Zahl $r \in \mathbb{R}$ kann man sich als unendliche Folge von Dezimalziffern vorstellen. Zur Ausgabe einer unendlich langen Zahl r stellt man sich eine Turingmaschine T_r vor, die als Argument die Anzahl n der Ziffern bekommt und dann die ersten n Ziffern von r aufs Band schreibt. Für $r = \pi$ gibt es eine solche Turingmaschine. Aber gibt es auch Turingmaschinen für all die anderen Zahlen? Offensichtlich nicht, denn die Menge der Turingmaschinen ist abzählbar, die Menge $\mathbb{R}$ aber nicht. Also ist der größte Teil der reellen Zahlen nicht als unendliche Folge von Ziffern angebbar[1]. Die meisten Zahlen sind *nicht berechenbar*!

14.1.6 Fleißige Biber

Eine DTM $(\{q_0, \ldots q_n\}, \{1\}, \{1, \square\}, \delta, q_1, \square, \{q_0\})$ mit $n+1$ Zuständen (einer davon Endzustand) nennen wir n-Biber, wenn sie auf das leere Band eine endliche Anzahl von Einsen schreibt und dann in den Endzustand geht. Wir nennen sie *fleißigen* n-Biber (busy beaver), wenn es keinen n-Biber gibt, der mehr Einsen schreibt. Der Kopf des Bibers muss am Ende nicht an einer bestimmten Stelle sein.

Die Anzahl der Einsen, die ein fleißiger n-Biber erzeugt, nennt man $\mathrm{bb}(n)$. Es gilt $\mathrm{bb}(1) = 1$, $\mathrm{bb}(2) = 4$, $\mathrm{bb}(3) = 6$ und $\mathrm{bb}(4) = 13$. Die Werte für $n \geq 5$ kennt niemand.

[1] Eine reelle Zahl ist zwar immer eine unendliche Folge von Dezimalziffern, aber in den meisten Fällen kann kein Verfahren angegeben werden, welche Ziffer als nächstes kommt

Bevor wir uns näher mit der Theorie befassen, versuchen wir, selber einige Biber zu erstellen.

Wir wären stolz, könnten wir den Biber $q_1\Box 1Rq_2$, $q_1 11Lq_1$, $q_2\Box 1Rq_4$, $q_2 11Lq_3$, $q_3\Box 1Lq_0$, $q_3 1\Box Lq_2$, $q_4\Box 1Rq_5$, $q_4 11Rq_3$, $q_5\Box\Box Lq_1$, $q_5 11Rq_4$ überbieten. Er macht in 15589 Schritten 165 Striche.

Die Biber-Funktion

Interessant ist die Anzahl der Einsen eines fleißigen Bibers deshalb, weil sie nachweislich nicht berechenbar ist, das ist neu. Gesucht ist also die Funktion bb(n), die angibt, wie viele Einsen ein fleißiger n-Biber auf ein leeres Band schreibt.

Nehmen wir einmal an, wir hätten eine Turingmaschine B, die auf dem Band eine Zahl n binär kodiert vorfindet und dazu bb(n) ausrechnet und als Binärzahl auf dem Band hinterlässt. Die Anzahl der Zustände von B nennen wir b und sie ist unbekannt aber endlich.

Außerdem stellen wir uns vor, wir haben einen fleißigen n-Biber, also eine Turingmaschine mit n Zuständen, die maximal viele Einsen auf ein leeres Band schreibt; nennen wir sie B_n. Die Maschine B_n schreibt also so viele Einsen aufs Band, wie B (angenommen) vorher ausgerechnet hat.

Nun denken wir uns noch zwei einfache Hilfs-TMs, die wir sogar leicht selber programmieren können: A schreibt die Zahl n binär auf das leere Band und C wandelt eine Binärzahl in die gleiche Anzahl von Einsen um. A hat nur ca. $\log n$ Zustände (bitte probieren!) und C hat auch nur ein paar wenige Zustände, sagen wir c Stück.

Wenn wir nun die drei TMs A, B und C hintereinander auf dem anfangs leeren Band arbeiten lassen, dann schreibt A die Zahl n aufs Band, B rechnet aus, wie viele Einsen ein fleißiger n-Biber schreibt und C erzeugt dann auch noch diese Anzahl von Einsen. Damit ist ABC selber ein fleißiger n-Biber, hat aber nur $\log n + b + c$ Zustände. Das sind weniger als die n Zustände die ein fleißiger n-Biber hat, nämlich n Stück. (Wenn n groß genug ist, gilt $n > \log n + b + c$.) Das ist ein Widerspruch und die Annahme, es gäbe die Turingmaschine B muss falsch sein, weil es die anderen TMs ja gibt. Das ganze noch einmal übersichtlicher:

Name	Tätigkeit	Anzahl der Zustände
B	berechnet angeblich bb einer gegebenen Zahl	$b = const < \infty$
B_n	fleißiger n-Biber	n
C	wandelt binär um in unär	$c = const < \infty$
A	schreibt n binär aufs Band	$\log n$
ABC	wie B_n	$\log n + b + c$

14.2 Entscheidbarkeit

Eng verwandt mit der Berechenbarkeit ist der Begriff der *Entscheidbarkeit*. Man verwendet ihn mit Mengen oder Sprachen (die ja Mengen sind) und möchte wissen, ob ein bestimmtes Element oder Wort zu einer gegebenen Menge gehört. Dazu muss man das Wort mit einer Maschine verarbeiten und bekommt am Ende der Berechnung die Antwort *ja, das Element gehört zur Menge* oder *nein, das Element gehört nicht zur Menge*. Wenn der Algorithmus in beiden Fällen in endlicher Zeit fertig wird, nennt man die Menge *entscheidbar*. Wenn der Algorithmus nur im Falle der Zugehörigkeit in endlicher Zeit fertig wird, nennt man die Menge *semi-entscheidbar*. Unentscheidbar ist eine Menge, wenn sie nicht entscheidbar ist. Sie kann dann immer noch semi-entscheidbar sein! Der Sachverhalt lässt sich mit Hilfe folgender Funktionen ausdrücken:

Die charakteristische Funktion

Die charakteristische Funktion χ_M einer Menge M hat den Wert 1, wenn das Argument zu M gehört und 0, wenn nicht:

$$\chi_M(x) = \begin{cases} 1 & \text{wenn } x \in M \\ 0 & \text{wenn } x \notin M \end{cases}$$

Es ist also gerade die charakteristische Funktion einer Menge, die berechenbar sein muss, wenn die Menge als entscheidbar gelten soll. Semi-entscheidbar nennt man eine Menge, wenn die Funktion χ' – man könnte sie die *halbe charakteristische Funktion* nennen – berechenbar ist

$$\chi'_M(x) = \begin{cases} 1 & \text{wenn } x \in M \\ \text{undefiniert} & \text{wenn } x \notin M \end{cases}$$

14.2.1 Unentscheidbarkeit von Typ-0-Sprachen

Die charakteristischen Funktionen von Sprachen des Typs 3 bis 1 sind berechenbar, die Sprachen also entscheidbar. Typ-0-Sprachen sind im allgemeinen aber nur semi-entscheidbar, d. h. wenn die Sprache „schwierig“ genug ist, kann nur die Zugehörigkeit eines Elements in endlicher Zeit nachgewiesen werden. Wenn ein Element nicht dazu gehört, bemüht sich die TM unendlich lang, nachzuweisen, dass es dazu gehört, schafft es aber nicht.

Woran liegt es, dass Sprachen von den Typen 3 bis 1 entscheidbar sind, solche vom Typ 0 aber im allgemeinen nur semi-entscheidbar? Bei den Typen 3 bis 1 ist Voraussetzung, dass die rechten Seiten der Regeln in den Grammatiken mindestens so lang sein müssen wie die linken. Hat man nun ein Wort, das nicht zur Sprache gehört, so ist dies in endlicher Zeit feststellbar, weil die linken Seiten der Regeln höchstens so

lang sind wie das Wort. Hat man alle Kombinationen von Ableitungsregeln durchprobiert und das Wort nicht ableiten können, so gehört es sicher nicht zur Sprache.

Bei Typ-0-Sprachen würde man auch gern alle Kombinationen von Ableitungsschritten durchprobieren, aber das geht im allgemeinen nicht, weil ihre Zahl nicht begrenzt ist. Während der Ableitung eines Wortes können die Zwischenergebnisse nämlich beliebig viel länger sein als das zu testende Wort. Es wird also nie der Punkt erreicht, an dem man weiß, dass man nichts mehr probieren muss. Hat man ein Wort, das nicht zur Sprache gehört, so kann man unendlich lang versuchen, seine Ableitung finden.

14.2.2 Halteproblem

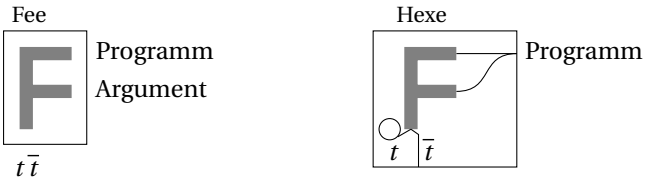
Computerprogramme haben meist Schleifen und brauchen deshalb manchmal sehr lang, bis sie fertig werden. Wenn man lange gewartet hat, das Programm aber noch nicht fertig ist, kommt die Frage auf, ob es überhaupt jemals terminiert oder ob es in einer Endlosschleife ist. Wir wollen hier nur Programme betrachten, die genau ein Argument bekommen. Für mehr Argumente verfährt man ganz ähnlich.

Praktisch wäre es da, wenn man ein Hilfsprogramm *Fee* hätte, das zwei Argumente bekommt: Das zu testende Programm P und das Argument A, das man P geben will. Die Fee soll nun analysieren, ob P mit A terminieren wird (t) oder nicht ($\overline{t}$). Wir beweisen, dass es keine Fee gibt, die alle Programme auf die genannte Weise analysieren kann. Der Beweis ist erbracht, wenn wir ein Programm finden, von dem die Fee nicht entscheiden kann, ob es terminiert.

Gäbe es eine Fee, so würden wir ein Programm *Hexe* schreiben, das genau ein Argument X bekommt, dieses verdoppelt und damit Fee als Unterprogramm aufruft. (Damit Fee damit etwas anfangen kann, muss X ein Programm sein.) Fee findet nun heraus, ob das Programm X terminiert, wenn es die Daten X verarbeitet. (Dass das Programm X damit eine Kopie seiner selbst verarbeiten soll, ist ein bisschen seltsam, aber nicht verboten.) Als nächstes programmieren wir in Hexe eine Abfrage: Wenn die Fee in der Hexe sagt, dass X mit X terminiert, schicken wir Hexe in eine Endlosschleife. Wenn Fee sagt, dass X mit X nicht terminiert, ist Hexe fertig und terminiert.

Nun geben wir der Hexe ein Programm, lassen sie laufen und überlegen uns, ob die Hexe terminiert. Raffinierterweise geben wir ihr als Programm sie selber! Die Frage ist also: Terminiert Hexe mit dem Argument Hexe? Um das zu ermitteln betrachten wir einfach das Innere von Hexe und sehen, dass da eine Fee ist, die uns das Denken größtenteils abnimmt. Leider erkennen wir, dass die Hexe mit Hexe genau dann terminiert, wenn Hexe mit Hexe nicht terminiert. Das ist ein Widerspruch und

die Annahme, es könnte Fee geben, war falsch.



14.2.3 Die Collatz-Folge

Man beginne mit einer natürlichen Zahl. Wenn sie gerade ist, teile man sie durch zwei, wenn nicht multipliziere man sie mit 3 und addiere 1. Wiederholt man dieses Verfahren fortwährend, so kommt man manchmal auf das Ergebnis 1. Ab dann wiederholen sich die Zahlen 4, 2, 1. *Manchmal* ist allerdings stark untertrieben. Bis jetzt hat noch niemand eine Startzahl gefunden, die nicht zu 1 führt. Für einen Beweis, dass das immer der Fall ist, gibt es mehrere Hundert Euro Preisgeld.

Wenn man durch das Verfahren von einer gegebenen Zahl zur 1 gelangt, sagen wir, die Zahl hat die Collatz-Eigenschaft. Ist diese Eigenschaft entscheidbar? Vielleciht! Aber im Moment kennt man kein Verfahren außer die oben gegebene Rechenvorschrift. Und die kommt nur dann zum Ende, wenn irgendwann 1 rauskommt. Die Collatz-Eigenschaft ist also im Moment nur semi-entscheidbar.

14.2.4 Die Goldbach-Vermutung

Im Jahr 1742 stellte Christian Goldbach die Vermutung auf, dass alle geraden Zahlen von 4 aufwärts als Summe zweier Primzahlen darstellbar sind. Es konnte bisher weder ein Gegenbeispiel gefunden werden, noch ein Beweis.

Eine gerade Zahl hat also die Goldbach-Eigenschaft, wenn sie als Summe zweier Primzahlen darstellbar ist. Ist diese Eigenschaft entscheidbar?

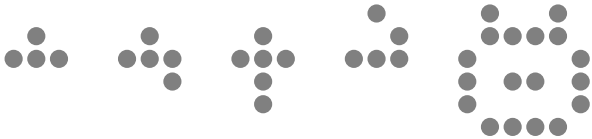
14.2.5 Conways Game of Life

Im *Scientific American* stellt John H. Conway 1970 erstmals ein Spiel vor, das seitdem viel Aufsehen und mathematische Untersuchung erregt hat. Die Kästchen eines karierten Blattes heißen Zellen und sind tot oder lebendig. Die Zeit vergeht schrittweise und bei jedem Schritt entscheidet sich für jede Zelle, ob sie lebt oder stirbt durch Betrachtung der direkten acht Nachbarn:

- Eine tote Zelle wird leben, wenn jetzt genau drei Nachbarn leben.
- Eine lebende Zelle wird sterben, wenn jetzt vier oder mehr Nachbarn leben.

- Eine lebende Zelle wird sterben, wenn sie jetzt weniger als zwei lebende Nachbarn hat.

Schreiben Sie ein Programm, das Zellkulturen aus einer Datei einliest und ihre Entwicklung graphisch darstellt. Untersuchen Sie das Verhalten der folgenden Kulturen:



Eine interessante Frage ist, ob eine Zellkultur irgendwann aussterben wird. Es wurde bewiesen, dass diese Eigenschaft nur semi-entscheidbar ist. Wenn die Kultur ausstirbt, merkt man es dadurch, dass man das Spiel lang genug laufen lässt. Wenn sie nicht ausstirbt, weiß man das im allgemeinen nie (obwohl es natürlich in einfachen Fällen zu Wiederholungen kommen kann, woran man dann sieht, dass die Kultur nicht ausstirbt).

Unter den folgenden Adressen findet man ein schönes fertiges Programm, sowie viele interessante Life-Muster.

- `http://www.bitstorm.org/gameoflife/`
- `http://www.bitstorm.org/gameoflife/lexicon/`

14.3 Aufgaben

1. Was besagt die Church-Turing These?
2. Was ist eine berechenbare Funktion?
3. Welche Funktionen sind berechenbar, nicht berechenbar, vielleicht berechenbar?
4. Warum sind Typ-0-Sprachen nur semientscheidbar?