

## 41. Knobelaufgabe #25

Zwei Listen sind *Permutationen* voneinander, wenn sie die gleichen Elemente enthalten — die Reihenfolge der Elemente spielt dabei keine Rolle. Eine Liste mit  $n$  verschiedenen Elementen besitzt insgesamt  $n!$  Permutationen.

Eine Liste heißt *Megapermutation* von  $x$ , wenn sie jede Permutation von  $x$  als zusammenhängende Teilliste enthält.

$x$	Megapermutation von $x$
abc	abcabacba
abcd	abcdabcadbcbdcabacdbacbdabcadcb

Schreiben Sie ein Programm, das zu einer gegebenen Liste eine möglichst kurze Megapermutation bestimmt.

*Hinweis:* Verwenden Sie das Strukturf Entwurfsmuster für Listen.

## 41. Klassen

Ein Bankinstitut, eine Sammlung von Bankkonten, kann alternativ durch eine sogenannte *Klasse* (engl. class) modelliert werden.

```

type TrustMe (seed : Nat) =
  class
    let mutable funds = seed
    member self.Deposit (amount : Nat) =
      funds ← funds + amount
    member self.Withdraw (amount : Nat) =
      funds ← monus (funds, amount)
    member self.Balance =
      funds
  end

```

☞ Eine Klasse ist ein *Zwitterwesen*: *TrustMe* ist sowohl ein Typ, genauer: eine Schnittstelle, als auch ein Wert, der die Schnittstelle implementiert, ein Objektkonstruktor.

## 41. Objekterzeugung

Mit **new** *TrustMe* 4711 oder kurz *TrustMe* 4711 wird ein neues Konto eröffnet, in das ein initialer Betrag von 4711€ eingezahlt wird.

```
Mini> let lisas = TrustMe 4711
```

```
val lisas : TrustMe
```

```
Mini> lisas.Deposit 815
```

```
()
```

```
Mini> lisas.Balance
```

```
5526
```

```
Mini> let ludwigs = new TrustMe 815
```

```
val ludwigs : TrustMe
```

☞ Jargon: *TrustMe* 4711 ist eine Instanz der Klasse *TrustMe*.

☞ Eine Klassendefinition **type** *T* (*x*) = **class** ... **end** lässt sich als Schablone (engl. template) auffassen; mit **new** *T* *e* wird die Schablone instantiiert; das Ergebnis ist ein Objekt vom Typ *T*.

## 41. Klassenvariablen und -methoden

```
type TrustMe (seed : Nat) =  
  class  
    static do putline "TrustMe is founded."  
    static let mutable no = 0  
    do no ← no + 1  
    let mutable funds = seed  
    static member BIC = 4711  
    static member no-of-accounts = no  
    member self.Deposit (amount : Nat) =  
      funds ← funds + amount  
    member self.Withdraw (amount : Nat) =  
      funds ← monus (funds, amount)  
    member self.Balance =  
      funds  
end
```

## 41. Klassenvariablen und -methoden

Klassenmethoden und -eigenschaften können unabhängig von der Existenz eines Objekts sprich eines Bankkontos verwendet werden.

☞ Zum Zeitpunkt der Gründung einer Bank existieren keine Konten.

```
Mini> TrustMe.BIC
4711
Mini> let lisas = TrustMe 4711
val lisas : TrustMe
Mini> TrustMe.no-of-accounts
1
```

☞ Die jeweilige Nachricht wird an die Klasse in unserem Beispiel an die Bank *TrustMe* geschickt.

## 41. Felder, Methoden und Eigenschaften

- ▶ **let**  $a = e$   
führt eine Konstante ein, einen Bezeichner für einen Wert (diese Konstante kann allerdings die Adresse einer Speicherzelle sein: **let**  $funds = ref\ seed$ ). Die Bindung wird bei der Erzeugung eines Objekts mit **new** ausgewertet. Eine Konstante ist stets privat und kann nicht öffentlich gemacht werden.
- ▶ **let**  $f\ x = e$   
führt eine Funktion ein. Eine Funktion ist stets privat.
- ▶ **let mutable**  $s = e$   
führt eine Instanzvariable ein, auch *Feld* (engl. field) genannt. Auch eine Instanzvariable ist stets privat.
- ▶ **do**  $e$   
kennzeichnet einen Ausdruck, der bei der Erzeugung eines Objekts mit **new** ausgeführt wird.

# 41. Felder, Methoden und Eigenschaften

- ▶ **member** *self.p* **with** *get () = e<sub>1</sub>* **and** *set v = e<sub>2</sub>*  
**member** *self.p* **with private** *get () = e<sub>1</sub>* **and** *set v = e<sub>2</sub>*  
führt eine Eigenschaft ein; es kann auch nur der Getter bzw. nur der Setter angegeben werden. Eigenschaften sind öffentlich; die Sichtbarkeit kann aber mit **private** selektiv eingeschränkt werden.
- ▶ **member** *self.m* *x = e*  
**member private** *self.m* *x = e*  
führt eine Methode ein. Methoden sind ebenfalls öffentlich, es sei denn, die Sichtbarkeit wird mit **private** eingeschränkt.

# 41. Felder, Methoden und Eigenschaften

- ▶ **static let**  $a = e$   
führt eine „Klassenkonstante“ ein. Die Bindung wird bei der Abarbeitung der Klassendefinition ausgewertet; in  $e$  sind nur mit **static** gekennzeichnete Bezeichner sichtbar.
- ▶ **static let**  $f x = e$   
führt eine „Klassenfunktion“ ein. Im Rumpf  $e$  sind nur mit **static** gekennzeichnete Bezeichner sichtbar. Eine Funktion ist stets privat.
- ▶ **static let mutable**  $s = e$   
führt eine Klassenvariable ein. Auch eine Klassenvariable ist stets privat.
- ▶ **static do**  $e$   
kennzeichnet einen Ausdruck, der bei der Abarbeitung der Klassendefinition ausgeführt wird. In  $e$  sind nur mit **static** gekennzeichnete Bezeichner sichtbar.

# 41. Felder, Methoden und Eigenschaften

- ▶ **static member**  $p$  **with**  $get () = e_1$  **and**  $set v = e_2$   
**static member**  $p$  **with private**  $get () = e_1$  **and**  $set v = e_2$   
führt eine Klasseigenschaft ein. Auf die Eigenschaft wird über den Klassennamen zugegriffen. Auch Klasseigenschaft sind öffentlich; die Sichtbarkeit kann aber mit **private** eingeschränkt werden.
- ▶ **static member**  $m x = e$   
**static member private**  $m x = e$   
führt eine Klassenmethode ein. Eine Klassenmethode ist der Natur nach eine ordinäre Funktion; lediglich der Aufruf erfolgt über den Klassennamen.

# 41. Module versus Klassen

Sichtbarkeit (engl. visibility, auch accessibility):

- ▶ *Modul*: Position des Bezeichners: global oder lokal;
- ▶ *Klasse*: durch vorhandene oder fehlende ***static*** Kennzeichnung.

## 41. Klassen und Schnittstellen

Eine alternative Implementierung einer Bank:

```
type Cheap'N'Easy (seed : Nat) =  
  class  
    let mutable funds-log = [seed]  
    member self.Deposit (amount : Nat) =  
      funds-log ← (head funds-log + amount) :: funds-log  
    member self.Withdraw (amount : Nat) =  
      funds-log ← monus (head funds-log, amount) :: funds-log  
    member self.Balance =  
      head funds-log  
    member self.Cancel =  
      funds-log ← tail funds-log  
  end
```

## 41. Klassen und Schnittstellen

*Beobachtung:* Weder das Objekt *TrustMe* 4711 noch das Objekt *Cheap'N'Easy* 815 kann an die Funktion *transfer* übergeben werden!

*transfer* : *IAccount* \* *Nat* \* *IAccount* → *Unit*

*TrustMe* 4711 : *TrustMe*

*Cheap'N'Easy* 815 : *Cheap'N'Easy*

☞ Alle drei Typen, *IAccount*, *TrustMe*, *Cheap'N'Easy*, sind verschieden und damit inkompatibel.

„All types are created unequal.“

☞ Die Gleichheit der Methodennamen könnte rein zufällig sein. Uns geht es aber nicht um Äußerlichkeiten (Wie heißt eine Methode?), sondern um innere Werte (Wie verhält sich eine Methode?).

## 41. Klassen und Schnittstellen

```

type Cheap'N'Easy (seed : Nat) =
  class
    let mutable funds-log = [seed]
    interface IAccount with
      member self.Deposit (amount : Nat) =
        funds-log ← (head funds-log) + amount :: funds-log
      member self.Withdraw (amount : Nat) =
        funds-log ← monus (head funds-log, amount) :: funds-log
      member self.Balance =
        head funds-log
      member self.Cancel =
        funds-log ← tail funds-log
    end

```

 *Lies:* `Cheap'N'Easy` implementiert die Schnittstelle `IAccount`. Damit ist die Klasse ein Untertyp der Schnittstelle: `Cheap'N'Easy`  $\preccurlyeq$  `IAccount`.

## 41. Klassen und Schnittstellen

Jetzt gelingt die Überweisung: Beim Aufruf der Funktion *transfer* wird automatisch der Typ mit Hilfe der Subsumptionsregel angepasst.

```
Mini> let herberts = Cheap'N'Easy 4711
Mini> let harrys = Cheap'N'Easy 10
Mini> transfer (herberts, 100, harrys)
()
Mini> (harrys :> IAccount).Balance
110
Mini> harrys.Cancel
()
Mini> (harrys :> IAccount).Balance
10
```

Eine explizite Typanpassung ist notwendig, wenn auf die Methoden der Schnittstelle zugegriffen wird: Von Haus aus versteht *harrys* die Nachricht *Balance* nicht. (Wir sehen später, warum das sinnvoll ist.)

## 41. Klassen und Schnittstellen

Möchte man aus Gründen der Bequemlichkeit den direkten Zugriff auf die Schnittstellenmethoden ermöglichen, kann man diese *zusätzlich* als Methoden der Klasse „veröffentlichen“.

```
type Cheap'N'Easy (seed : Nat) =
  class
    ...
    // expose interface
    member self.Deposit amount = (self :> IAccount).Deposit amount
    member self.Withdraw amount = (self :> IAccount).Withdraw amount
    member self.Balance          = (self :> IAccount).Balance
  end
```

Die Gleichungen geben jeweils an, dass die Methode der Klasse (links: *self.Withdraw*) durch die Schnittstellenmethode (rechts: *(self :> IAccount).Withdraw*) definiert wird.

☞ Lässt man die Typanpassung weg, erfolgt ein rekursiver Aufruf mit der Konsequenz der Nichtterminierung!

## 41. Klassen und Schnittstellen

Man kann auch den umgekehrten Weg beschreiten und die Schnittstellenmethoden durch Methoden der Klasse definieren.

```
type TrustMe (seed : Nat) =
  class
    ...
  interface IAccount with
    // these are not recursive definitions
    member self.Deposit amount = self.Deposit amount
    member self.Withdraw amount = self.Withdraw amount
    member self.Balance          = self.Balance
  end
```

Die Gleichungen geben jeweils an, dass die Schnittstellenmethode (links: *self.Withdraw*) durch die Methode der Klasse (rechts: *self.Withdraw*) definiert wird.

## 41. Klassen und Schnittstellen

Nach diesen Vorarbeiten lassen sich sowohl Banküberweisungen tätigen als auch die Kontostände direkt ohne Umweg über die Schnittstelle abrufen.

```
Mini> let herberts = TrustMe 4711
Mini> let harrys = Cheap'N'Easy 0
transfer (herberts, 100, harrys)
()
Mini> herberts.Balance
4611
Mini> harrys.Balance
100
```

Der direkte Zugriff mit *Balance* ist möglich, weil entweder die Schnittstelle bekanntgegeben wurde oder die Schnittstelle mit den Methoden der Klasse implementiert wurde.

## 41. Klassen und Schnittstellen

Eine Klasse kann auch mehrere Schnittstellen gleichzeitig implementieren.

```
type IPositive =  
  abstract Answer : string  
type INegative =  
  abstract Answer : string  
type Undecided () = // don't forget '()'   
  class  
    member self.Answer = "maybe"  
  interface IPositive with  
    member self.Answer = "yup"  
  interface INegative with  
    member self.Answer = "nope"  
end
```

## 41. Klassen und Schnittstellen

Die Schnittstellen repräsentieren unterschiedliche Stimmungen; je nach konkreter Stimmung fällt die „Antwort“ unterschiedlich aus.

```
Mini> let turncoat = Undecided ()
Mini> turncoat.Answer
val it : string = "maybe"
Mini> (turncoat :> IPositive).Answer
val it : string = "yup"
Mini> (turncoat :> INegative).Answer
val it : string = "nope"
```

Damit wird klar, warum bei Methodenaufrufen *keine* automatischen Typanpassungen vorgenommen werden (*Undecided*  $\preccurlyeq$  *IPositive* oder *Undecided*  $\preccurlyeq$  *INegative*): Kein Automatismus kann die vorhandene *Mehrdeutigkeit* auflösen.

## 41. Generische Klassen

In Teil V haben wir endliche Abbildungen mit Hilfe von Listen, Suchlisten und Suchbäumen implementiert, siehe Skript.

Ein Auszug aus der Schnittstelle:

```
type Map <'key, 'value when 'key : comparison>  
val empty : Map <'key, 'value>  
val add : 'key * 'value → Map <'key, 'value> → Map <'key, 'value>  
val look-up : 'key → Map <'key, 'value> → 'value option
```

☞ Ein Wörterbuch assoziiert mit einem Schlüssel einen bestimmten Wert. Sowohl Schlüssel als auch Wert können einen beliebigen Typ haben.

☞ Die oben genannten Implementierungen sind *persistent*.

☞ Im Folgenden implementieren wir eine ephemere Variante von endlichen Abbildungen: Beim Erweitern wird die ursprüngliche Abbildung überschrieben; sie ist danach *nicht* mehr verfügbar.

# 41. Generische Klassen

Die folgende Interaktion zeigt die Implementierung in Aktion.

```
Mini> let perfume = SearchTree <Nat, String> ()  
Mini> perfume.Add (4711, "Kölnisch Wasser")  
()  
Mini> perfume.Add (5, "Chanel")  
()  
Mini> perfume.Lookup 5  
"Chanel"
```

Das Wörterbuch *perfume* ordnet natürlichen Zahlen Strings zu. Mit der Methode *Add* wird das Wörterbuch um einen Eintrag erweitert; *Lookup* schlägt im Wörterbuch nach. Die ephemere Natur wird deutlich: *Add* hat den Ergebnistyp *Unit*; zu jedem Zeitpunkt existiert nur eine Version der endlichen Abbildung *perfume*.

# 41. Generische Klassen

Für beide Operationen bieten wir syntaktischen Zucker an:

```
Mini> perfume.[5]
```

```
"Chanel"
```

```
Mini> perfume.[0] ← "Eau"
```

```
()
```

```
Mini> perfume.[0]
```

```
"Eau"
```

```
Mini> perfume.[0] ← "Wasser"
```

```
()
```

```
Mini> perfume.[0]
```

```
"Wasser"
```

☞ `perfume.[k] ← v` fügt ein neues Schlüssel-Wert Paar hinzu (bzw. aktualisiert einen bestehenden Eintrag für den Schlüssel `k`); `perfume.[k]` schlägt den angegebenen Schlüssel nach.

## 41. Generische Klassen

☞ Wie Record-, Varianten- und Schnittstellentypen können auch Klassentypen mit einem oder mehreren Typen parametrisiert werden.

```
type SearchTree <'key, 'value when 'key : comparison> () =
class
  let mutable mapping : Map <'key, 'value> = empty
  member self.Add (key, value) =
    mapping ← add (key, value) mapping
  member self.Lookup key =
    match look-up key mapping with
    | None      → raise (KeyNotFoundException ())
    | Some value → value
  ...
end
```

☞ Eine Klasse ist ein Zwitterwesen: *SearchTree* ist ein *parametrisierter* Typ und ein *polymorpher* Objektkonstruktor.

# 41. Generische Klassen

```

type SearchTree ⟨'key', 'value when 'key : comparison⟩ () =
  class
    ...
    member self.Item
      with get key      = self.Lookup key
      and set key value = self.Add (key, value)
    end
  
```

Der syntaktische Zucker für Erweiterung und Zugriff wird mittels der Eigenschaft *Item* realisiert. Es handelt sich um eine sogenannte *indizierte Eigenschaft* (engl. indexed property).

☞ So wie die Eigenschaft  $l$  die Abkürzungen  $e.l$  und  $e.l \leftarrow e'$  ermöglicht, so erlaubt eine indizierte Eigenschaft den syntaktischen Zucker  $e.l [i]$  und  $e.l [i] \leftarrow e'$ .

☞ Ist weiterhin  $l = \textit{Item}$ , dann kann man  $l$  auch auslassen und kurz  $e.[i]$  und  $e.[i] \leftarrow e'$  schreiben.

# 41. Smalltalk und Java

☞ In den Programmiersprachen Smalltalk (1972 geboren) und Java (1995 geboren) spielen Objekte und Klassen eine weitaus größere Rolle als in Mini-F#.

- ▶ *Smalltalk*: Alles ist ein Objekt, auch Wahrheitswerte und Zahlen. Jedes Objekt ist Instanz einer Klasse; auch Klassen sind Objekte. Im Gegensatz zu Mini-F# ist Smalltalk *dynamisch getypt*.
- ▶ *Java*: Fast alles ist ein Objekt; neben Klassen gibt es auch primitive Typen wie Wahrheitswerte und Zahlen. Wie Mini-F# ist Java *statisch getypt*.

One of the most important influences on the design of Java was a much earlier language called Simula.  
— James Gosling

Your development cycle is much faster because Java is interpreted. The compile-link-load-test-crash-debug cycle is obsolete.  
— James Gosling

Like the creators of sitcoms or junk food or package tours, Java's designers were consciously designing a product for people not as smart as them.  
— Paul Graham

# 41. Java: Klassen

```
class TrustMe {
    private long funds = 0;

    TrustMe (long seed) {
        this.funds = seed;
    }

    void deposit (long amount) {
        funds = funds + amount;
    }

    void withdraw (long amount) throws Insufficient {
        if (amount > funds)
            throw new Insufficient (funds);
        else
            funds = funds - amount;
    }

    long balance () { return funds; }
}
```

 TrustMe (long seed) ist ein *Konstruktor*. Die Methodenrumpfe verwenden C-Syntax.

# 41. Java: Klassen

```
class TrustMe {
    ...
    static private int no = 0;
    static int BIC          = 4711;
    static {
        System.out.println ("TrustMe is founded.");
    }

    TrustMe (long seed) {
        this.funds = seed;
        no++;
    }
    ...
    static long no_of_accounts () { return no; }
}
```

# 41. Java: Schnittstellen

```
interface IAccount {  
    void deposit (long amount);  
    void withdraw (long amount) throws Insufficient;  
    long balance ();  
}
```

☞ Die Tatsache, dass `withdraw` möglicherweise eine Ausnahme wirft, ist im Typ vermerkt.

## 41. Java: Klassen, da capo

```
class TrustMe implements IAccount {
    private long funds    = 0;
    static private int no = 0;
    static int BIC        = 4711;

    ...
    public void deposit (long amount) {
        ...
    }

    public void withdraw (long amount) throws Insufficient {
        ...
    }

    public long balance () {
        ...
    }
    ...
}
```

# 41. Java: Module und Funktionen

```
import java.math.*;

public final class Factorial2 {
    public static BigInteger factorial (BigInteger n) {
        if (n.compareTo (BigInteger.ZERO) == 0)
            return BigInteger.ONE;
        else
            return n.multiply (factorial (
                n.subtract (BigInteger.ONE)));
    }

    public static void main (String[] args) {
        BigInteger n = new BigInteger("99");

        System.out.println (factorial (n));
    }
}
```

☞ Funktionen sind Klassenmethoden; Module sind Klassen (die nur Klassenvariablen und -methoden enthalten).

## 41. Lösung Knobelaufgabe #16

Kann man einem regulären Ausdruck ansehen, ob alle in der von dem regulären Ausdruck bezeichneten Sprache enthaltenen Wörter eine gerade Anzahl von  $a$ s enthalten?

---

- ▶  $\emptyset$
- ▶  $\epsilon$
- ▶  $(aa)^*$
- ▶  $a(aa)^*$
- ▶  $(\epsilon \mid aa)(\epsilon \mid aa)$
- ▶  $(a \mid aaa)(a \mid aaa)$
- ▶  $(\epsilon \mid aa)(a \mid aaa)$
- ▶  $(ab)^*$
- ▶  $(a \emptyset b)^*$

## 41. Lösung Knobelaufgabe #16

- ▶ Wir nennen eine Sprache *gerade*, wenn alle in der Sprache enthaltenen Wörter eine gerade Anzahl von *a*s enthalten.
- ▶ Wir nennen eine Sprache *ungerade*, wenn alle in der Sprache enthaltenen Wörter eine ungerade Anzahl von *a*s enthalten.
- ▶ Wir sind daran interessiert, ob eine Sprache gerade ist.
- ▶ Um das festzustellen, benötigen wir auch die Information, ob eine Sprache ungerade ist.

$(a \mid aaa) (a \mid aaa)$

- ▶ Einige Sprachen sind weder gerade noch ungerade.

$(\epsilon \mid aa) \mid (a \mid aaa)$

- ▶ Eine Sprache ist sowohl gerade als auch ungerade!

$\emptyset$

## 41. Lösung Knobelaufgabe #16

```

type Abstract =
  | Both // Durchschnitt von Even und Odd (nur  $\emptyset$ ).
  | Even // Sprachen, deren Wörter eine gerade # as enthalten.
  | Odd // Sprachen, deren Wörter eine ungerade # as enthalten.
  | None // Vereinigung von Even und Odd.

let rec even-odd (reg : Reg) : Abstract =
  match reg with
  | Eps → Even
  | Sym x → match x with A → Odd | B → Even
  | Cat (r1, r2) → cat (even-odd r1, even-odd r2)
  | Empty → Both
  | Alt (r1, r2) → alt (even-odd r1, even-odd r2)
  | Rep r → rep (even-odd r)

```

 Wir müssen noch *cat*, *alt* und *rep* definieren.

## 41. Lösung Knobelaufgabe #16

```
let cat (a1 : Abstract, a2 : Abstract) : Abstract =  
  match (a1, a2) with  
  | (Both, -)   | (-, Both)   → Both  
  | (None, -)   | (-, None)    → None  
  | (Even, Even) | (Odd, Odd)  → Even  
  | (Even, Odd)  | (Odd, Even) → Odd
```

 *Both* ist das Nullelement der Konkatenation, *Even* ist das Einselement.

## 41. Lösung Knobelaufgabe #16

```
let alt (a1 : Abstract, a2 : Abstract) : Abstract =  
  match (a1, a2) with  
  | (Both, a) | (a, Both) → a  
  | (Even, Even)           → Even  
  | (Odd, Odd)             → Odd  
  | _                       → None
```

 *Both* ist das Einselement der Alternative.

## 41. Lösung Knobelaufgabe #16

```
let rep (a : Abstract) : Abstract =  
  match a with  
  | Both | Even → Even  
  | _          → None
```

☞  $\emptyset^* = \epsilon$ , deswegen ist  $\text{rep } \text{Both} = \text{Even}$ .