5. Typen und Module
Datentypen mit Konstruktoren

Definition von Datentypen, die verschiedene Wertebereiche zu einem allgemeineren zusammenfassen.

Beispiel: Modellierung britischer Adelspersonen

King	eindeutig, einmalig	
Peer	Rang, Territorium, Erbfolge	z. B. 7th Earl of Carlisle
Knight	Name	z. B. Galahad
Peasant	Name	z. B. Jack Cade

Allgemeines Konzept:
unterscheidbare Vereinigung von Wertebereichen (**discriminated union**).

Verschiedene Techniken in verschiedenen Sprachen:

- Oberklasse u. **spezielle Unterklassen** in objektorientierten Sprachen
- **Record mit Varianten** in Pascal, Modula, Ada
- **Vereinigungstyp** in Algol68
- **`struct`** mit Unterscheidungskomponente und **`union`** in C
- **`datatype`** in SML

Discriminated Union mit Konstruktor-Funktionen

Allgemeines Konzept: **discriminated union**; In SML realisiert durch:

```
datatype person =
     King
   | Peer    of string * string * int
   | Knight  of string
   | Peasant of string;
```

Definiert den Typ **`person`** mit seinen Konstruktoren:

```
King:   person
Peer:   string * string * int -> person
Knight: string -> person
Peasant:string -> person
```

Notation für Werte:
`King`, **`Peer ("Earl", "Carlisle", 7)`**, **`Peasant ("Jack Cade")`**

Fallunterscheidung mit Konstruktoren in Mustern:

```
fun title King                 = "His Majesty the King"
|   title (Peer (deg, terr, _)) = "The "^deg^" of "^terr
|   title (Knight name)         = "Sir "^name
|   title (Peasant name)        = name;
```

Jede **`datatype`**-Definition führt einen **neuen** Typ ein.
Vorsicht beim Verdecken durch Redefinieren!

Verallgemeinerte Datentypkonstruktion

Aufzählungstyp als Vereinigung 1-elementiger Wertebereiche:

```
datatype degree = Duke | Marquis | Earl | Viscount | Baron;
```

Hier sind alle Konstruktoren Konstante.

```
datatype order = LESS | EQUAL | GREATER;
```

verwendet in **`String.compare: string * string -> order`**

Konstruktion polymorpher Typen:

allgemeines Konzept „Fehlerwert“:

```
datatype 'a option = NONE | SOME of 'a;
```

z. B. in **`Real.fromString: string -> real option`**

allgemeines Konzept „Wertebereiche paarweise vereinigen":

```
datatype ('a,'b) union = In1 of 'a | In2 of 'b;
```

z. B. **`[(In1 5), (In2 3.1), (In1 2)]`**

rekursiv definierte polymorphe Typen:

lineare Listen: **`datatype 'a list = nil | :: of ('a * 'a list);`**

binäre Bäume: **`datatype 'a tree = Lf | Br of 'a * 'a tree * 'a tree;`**

allgemeine Bäume:

```
datatype 'a mtree =
     Mleaf | MBranch of 'a * ('a mtree) list;
```


Binäre Bäume

Typdefinition: **`datatype 'a tree = Lf | Br of 'a * 'a tree * 'a tree;`**

ein Baum-Wert: **`val t2 = Br (2, Br (1, Lf, Lf), Br (3, Lf, Lf));`**

Rekursionsmuster: Fallunterscheidung und Rekursion wie in der Typdefinition

```
fun  size Lf               = 0
|    size (Br (v,t1,t2))   = 1 + size t1 + size t2;
fun  preorder Lf           = []
|    preorder (Br(v,t1,t2))= [v] @ preorder t1 @ preorder t2;
```

mit akkumulierendem Parameter:

`preord` stellt der schon berechneten Liste den linearisierten Baum voran:

```
fun  preord (Lf, vs)          = vs
|    preord (Br(v,t1,t2), vs) = v :: preord (t1, preord (t2, vs));
```

inverse Funktion zu **`preorder`** baut **balancierten Baum** auf:

```
balpre: 'a list -> 'a tree

fun  balpre nil      = Lf
|    balpre (x :: xs)=
       let val k = length xs div 2
       in Br(x, balpre (List.take (xs, k)),
                balpre (List.drop (xs, k)))
       end;
```

Gekapselte Typdefinition

abstype definiert **neuen** ggf. **polymorphen Typ mit Operationen darauf**.

Außen sind **nur die Operationen sichtbar**,
nicht aber die **Konstruktorfunktionen** des Datentyps (Implementierung).

Beispiel Wörterbuch mit Binärbaum implementieren:

```
abstype 'a t =
   Leaf | Branch of key * 'a * 'a t * 'a t
with
   exception NotFound of key;
   val empty = Leaf;
   fun lookup (Branch(a,x,t1,t2), b) = ...
   fun insert (Leaf, b, y) = ...
   |   insert (Branch(a,x,t1,t2), b, y) = ...
   fun update (Leaf, b, y) = raise NotFound b
   |   update (Branch(a,x,t1,t2), b, y) = ...
end;
```

Anwendung:

```
val wb = insert (insert (empty,"Hund","dog"), "Katze","cat");
val w = lookup (wb,"Hund");
```


Zusammenfassung von Typdefinitionen

SML-Konstrukt	Bedeutung	Vergleich mit anderen Sprachen
`datatype`	neuer Typ mit Konstruktoren	**`type`** in Pascal, Klassen in objektorientierten Sprachen
`type`	anderer Name für existierenden Typ	**`typdef, #define`** in C
`abstype`	neuer Typ mit Operationen aber verborgenen Konstruktoren	ADT mit verborgener Implementierung, opaque in Modula, Ada
Modul-Konstrukte:		
`structure`	Modul mit neuem Namensraum	Record in Pascal, Modula-Modul
`signature`	beschreibt Schnittstelle bzw. Signaturen	Interface in Java, Modula, Header-Dateien in C,
`functor`	parametrisierte Struktur	Templates in C++, Generics in Ada, Java Interface als Parametertyp

Ausnahmebehandlung (Exceptions)

Motivation:
Partielle Funktion `f` total machen ist umständlich:

- Ergebnistyp **`'a`** ersetzen durch **`datatype 'a option = NONE | SOME of 'a;`**
- Fallunterscheidung bei jedem Aufruf von **`f`**
- dasselbe für jede Funktion, die **`f`** aufruft;
  Fehlerwert „durchreichen“ bis er „behandelt“ wird.

SML-Ausnahmen propagieren Fehler von der Auslösestelle auf dem Berechnungsweg bis zur Behandlung - ohne Änderung von Typen und Funktionen.

Deklaration:

```
exception Failure;
exception Fail of string;
```

ergänzt vordefinierten Typ **`exn`** um eine Variante

auslösen durch Ausdruck

```
raise Failure
raise (Fail "too small")
```

Behandlung im Ausdruck
(zu prüfender Ausdruck)

```
handle Failure => E1
|      Fail (s) => E2
```

Das Ergebnis bestimmt der zu prüfende Ausdruck oder **`E1`** oder **`E2`**

Beispiel zur Ausnahmebehandlung

Beispiel Münzwechsel
Eine Lösung wird durch Backtracking im Lösungsraum gesucht.
Wenn ein Zweig keine Lösung liefern kann, wird eine Ausnahme ausgelöst: **`raise Change`**
Das Backtracking verweigt am zuletzt durchlaufenen **`handle Change`**
Gibt es keine Lösung, so bleibt die Ausnahme unbehandelt
`Uncaught exception Change`

```
exception Change;

fun backChange (coinvals, 0)      = []
  | backChange ([],  amount)      = raise Change
  | backChange (c::coinvals, amount) =
      if amount<0 then raise Change
      else c :: backChange(c::coinvals, amount-c)
             handle Change =>
               backChange(coinvals, amount);
```

Module in SML

`structure`:
Implementierungsmodul
(vgl. Modula 2);
kapselt Folge von Deklarationen:

```
structure Seq =
 struct
   exception Empty;
   fun
     hd (Cons(x,xf)) = x
   | hd Nil = raise Empty;
   ...
 end;
```

Qualifizierte Namen wie **`Seq.hd`** benennen exportierte Größen.

Module und Sichtbarkeit von Typen, Varianten:

1. Der Modul implementiert eine **Sammlung von Funktionen** zu einem Datentyp;
   der Datentyp wird außerhalb des Moduls definiert
   `datatype 'a seq = Nil | Cons of 'a ...`
2. Der Modul implementiert einen Datentyp mit Operationen darauf;
   die **Implementierung** (Konstruktorfunktionen) soll aussen **qualifiziert sichtbar** sein (**`Seq.Cons`**);
   der Datentyp wird innerhalb des Moduls definiert
   ```
   structure Seq = struct
     datatype 'a t = Nil | Cons of ...
     ... exportierte Funktionen
   end;
   ```
   Namenskonvention: t für **den** exportierten Typ.
3. wie (2) aber mit **verborgener Implementierung** des Datentyps;
   Konstruktorfunktionen sind nicht benutzbar):
   ```
   structure Bar = struct
     abstype 'a t = Nil | Cons of ...
     with ... exportierte Funktionen
     end
   end;
   ```


Schnittstellen

`signature`:
Schnittstelle von Modulen
definiert eine Folge von typisierten Namen (specifications),
die ein Modul mindestens implementieren muss, um die **`signature`** zu erfüllen;
`signature` ist eigenständiges, benanntes Programmobjekt (vgl. Java Interfaces):

```
signature QUEUE =
sig  type 'a t
     exception E
     val empty: 'a t
     val enq:   'a t * 'a -> 'a t
     ...
end;
```

Mehrere Module können eine Schnittstelle unterschiedlich implementieren:

```
structure QueueStd: QUEUE = struct ... end;
structure QueueFast: QUEUE = struct ... end;
```

Man kann auch zu einer existierenden Implementierung eine Definition zufügen, die erklärt, dass sie eine Schnittstelle implementiert (fehlt in Java):

```
structure MyQueue = struct ... end;
...
structure QueueBest: QUEUE = MyQueue;
```


Generische Module

Ein **generischer Modul** (**`functor`**) hat Strukturen als generische Parameter.
(vgl. Klassen als generische Parameter von generischen Definition in C++, Ada, Java)

Formale generische Parameter sind mit einer Signatur typisiert.
Damit können Anforderungen an den aktuellen generischen Parameter formuliert werden, z. B.

- „muss eine Schlangenimplementierung sein“,
- „muss Typ mit Ordnungsrelation sein“.

Garantiert Typ-sichere Benutzung von Modulfunktionen (nicht in C++ und Ada).

Beispiel: **`functor`** für Breitensuche mit Schlange:

```
functor BreadthFirst (Q: QUEUE) =
struct  fun enqlist q xs =
           foldl (fn (x,r)=> Q.enq(r,x)) q xs;
        fun search next x = ...
end;
```

Der Funktor wird mit einem zur Signatur passenden Modul **instanziiert**:

```
structure Breadth = BreadthFirst (QueueFast);
```


Beispiel im Zusammenhang: Wörterbuch-Funktor (1)

Aufzählungstyp (**`datatype`** mit Konstruktorfunktionen):

```
datatype order = LESS | EQUAL | GREATER;
```

Typen mit Ordnung als Schnittstelle (**`signature`**):

```
signature ORDER =
   sig  type t
        val compare: t * t -> order
   end;
```

Konkreter Modul (**`structure`**) für String-Vergleiche:

```
structure StringOrder: ORDER =
   struct  type t = string;
           val compare = String.compare
   end;
```

Beispiel im Zusammenhang: Wörterbuch-Funktor (2)

Generischer Modul (**functor**) für Wörterbücher implementiert mit binärem Suchbaum:

```
functor Dictionary (Key: ORDER): DICTIONARY =
   struct
      type key = Key.t;
      abstype 'a t = Leaf | Bran of key * 'a * 'a t * 'a t
      with exception E of key;
           val empty = Leaf;
           fun  lookup (Leaf, b)           = raise E b
           |    lookup (Bran(a,x,t1,t2), b)=
                (case Key.compare (a, b) of
                     GREATER => lookup (t1, b)
                   |   EQUAL => x
                ...
           ...
      end
   end;
```

Instanziierung des Funktors für einen Wörterbuch-Modul mit String-Schlüsseln:

```
structure StringDict = Dictionary (StringOrder);
```

Erzeugung und Benutzung eines Wörterbuches:

```
val dict = StringDict.update(..(StringDict.empty,“Kastens“,6686));
val tel = StringDict.lookup (dict, “Kastens“);
```