Datentyp Deklarationen

Datentyp Zusammenfassung von Wertebereichen und Operationen zu einer Einheit

a) elementare Datentypen

Name	Konstanten	Operationen
unit	()	keine
bool	true, false	not, =, <>, andalso, orelse
int	17, ~23, 001	+, -, * div, mod, abs, =, <>,<, >, >=,<=
real	0.01, 3.1415	+, -, *, /, sin, ln, exp, sqrt, =, <>, <, >, >=, <=
string	"say \"no\""	size(..), ^, chr, ord

b) zusammengesetzte Datentypen (Datenstrukturen)

Bsp. datatype CARD = King | Queen | Jack | Other of int;

Formen von Typdeklarationen:

type Verwendung von Standardkonstruktoren: *, {...}, (..), →

datatype beschreibt einen neuen Typ mit benutzerdef. Konstruktoren

Konstruktoren für Datentypen

- **Enumeration** (Bildung von Aufzählungstypen)
  datatype $D = d_1 \mid d_2 \mid ... \mid d_n$ $\quad d_i \in$ Wertebereich(D)
- **Restriktion** (Einschränkung der Wertemenge)
  D = D {x | P(x)}
- **Aggregation** (Bildung des kartesischen Produkts von Datentypen)
  - anonyme Tupel type $D = D_1 * ... * D_n$
  - bezeichnete Tupel datatype $D = c$ of $D_1*...*D_n$
  - Records type $D = \{s_1:D_1, ..., s_n:D_n\}$
- **Generalisation** (Vereinigung von disjunkten Datentypen)
  datatype $D = c_1$ of $D_1 \mid c_2$ of $D_2 \mid ... \mid c_n$ of D_n
- **Rekursion** (Übergang zu abzählbar unendlichen Wertebereichen)
  datatype $D = c_0$ of D' | c of D'' , wobei D in D“ vorkommt
- **Potenzmengenbildung** (alle (endlichen) Teilmengen eines Datentyps)
  $D = 2^D$
- **Funktionsräume** (Übergang zur Menge aller Abbildungen über Datentypen)
  D = (fn D‘ => D“)

Selektoren der Aggregation

- Zugriff auf eine Record-Komponente mit #
  Bsp. - val Bruch = {Zahler=2, Nenner=3};
  - #Zaehler Bruch;
  3:int
- Zugriff auf eine Tupel-Komponente
  a) mit PatternMatching Bsp. fun Zaehler(Bruch(a,b))=a;
  b) als „Spezial-Record“ Bsp. #2 Bruch(a,b);

Pattern matching

- leistungsfähiges und übersichtliches Konzept zur Parameterübergabe
- erlaubt den Verzicht auf explizite Anwendung von Selektoren und der damit verbundenen expliziten Fallunterscheidung durch geschachtelte bed. Ausdrücke
- für jedes Muster wird festgelegt, welchen Wert die Funktion besitzt, wenn der Parameter in das Muster passt.

Bsp. Funktion and: bool×bool → bool

```
a) fun and x y = if x then
                    if y then true else false
                 else false;
b) fun and true true = true | and (_, _) = false;
```

Ein **Pattern** in ML ist ein Ausdruck, der nur aus Bezeichnern für Variablen, Konstruktoren und dem Zeichen _ (Wildcard) besteht. Dabei müssen alle Variablenbezeichner paarweise verschieden sein.

Polymorphie

- Ein Datentyp heißt polymorph, wenn er einen Typparameter enthält.
- Eine Funktion heißt polymorph, wenn der Typ eines ihrer Argumente oder des Ergebnisses polymorph ist

Bsp. - fun id x = x;
    val id = fn : 'a -> 'a

Polymorphe Typen werden in ML durch Typausdrücke definiert, die mithilfe von

- Typvariablen	'a
- Typkonstanten	int, real
- Typkonstruktoren	*, →
- Typfunktionen	list, stack
- Klammern	

ausgedrückt werden.

Prioritäten:
Typfunktionen > * > →

```
Bsp.  type 'param paare = 'param * 'param;   (* Typ aller Paare *)
      type ip = int paare;                   (* Konkretisierung für integer *)
```


Formen der Polymorphie

- **Universelle Polymorphie**: bei der Funktionen auf gleiche Weise auf verschiedenen Typen agieren, deren Struktur im wesentlichen übereinstimmt.
  - **Parameterpolymorphie**: ein Typparameter wird bei jeder Anwendung mit einem (anderen) konkreten Typ belegt.
  - **Inklusionspolymorphie**: ein Objekt gehört zu mehreren (ggf. disjunkten) Typen, so dass die auf diesen Typen definierten Funktionen auch für das Objekt definiert sind (Vererbung).
- **Ad-hoc Polymorphie**: den Typen liegt keine gemeinsame Struktur zugrunde. Die Funktion kann auf mehrere Typen angewendet werden, verhält sich jedoch auf jedem Typ anders. (anderer Code wird durchlaufen).
  - **Polymorphie durch Überladen**: derselbe Name wird für strukturell verschiedene Objekte verwendet. Der zu verwendende Code ergibt sich aus dem Zusammenhang.
  - **Polymorphie durch Typanpassung**: das Objekt wird durch automatische Typanpassung an den erforderlichen Argumenttyp einer Funktion angeglichen

Listen

<u>datatype</u> 'a list = nil | :: <u>of</u> 'a * 'a list

[] infix-Konstruktor

Bsp.

```
- 2::[3,5];
  val it = [2,3,5] : int list
```

Standardfunktionen

```
fun null [] = true | null (_::_) = false                    (* Test auf Leerheit *)
fun hd (x::_) = x                                            (* Erstes Element einer Liste *)
fun tl (_::x) = x                                            (* Letztes Element einer Liste *)
fun len [] =0 | len (_::x) = 1 + len (x)                     (* Länge einer Liste *)
fun append [] x = x | append (x::y) z = x::(append y z)      (* Konkatenation *)
fun member (x::_) x = true | member (_::y) x = member x y    (* Elementtest *)
```


Standardfunktionale auf Listen

Generation von Listen:

```
fun generate f a 0 = [a] | generate f a k = [a]@generate f (f a) (k-1)
Bsp.  generate (fn x => x+1) 0 10;
> [0,1,2,3,4,5,6,7,8,9,10]: int list
```

Transformation einer Liste:

```
fun map f [] = [] | map f (x::y) = (f x)::(map f y)
Bsp. map sin [0.1, 0.2, 0.3, 0.4, 0.5]
> val it =[0.09983,0.198669,0.295,0.3894,0.4794]  : real list
```

Akkumulation von Listenelementen:

```
fun akku f a [] = a | akku f a (x::y) = f(x, akku f a y)
Bsp. val sum = akku(fn(x,y)=>x+y) 0;
>val sum = fn : int list -> int
```

Filtern von Elementen einer Liste, die ein Prädikat p erfüllen:

```
fun filter p [] = [] | filter p (x::y) = if p x then x::(filter p y) else filter p y
Bsp. filter (fn x => x>0) [~7,~3, 8,~2]
>val it = [8] : int list
```


„Lazy lists“ in ML

```
fun nonsense x y = if x then y else 0;
nonsense false (2 div 0);
> ERROR

fun junk ()=2 div 0;
fun nonsense x y = if x then y () else 0;
nonsense false junk;
> val nonsense = fn : bool -> (unit -> int) -> int

(* lazy lists in ML *)
datatype 'type lazylist = empty | c of 'type * (unit -> 'type lazylist).
```

Eine lazy list ist also entweder leer, oder sie hat die Form c(x,y), wobei x das erste Element ist und y eine Funktion ist, die den Rest der Liste berechnet.

Standardfunktionen auf lazy lists

```
fun head (c(x,_))=x;
> val head=fn: 'a lazylist -> 'a

fun tail (c(_,y))=y();
> val tail=fn: 'a lazylist -> 'a lazylist.
```

(* Standardfunktion :: für lazy lists *)

```
fun colon x y=c(x,fn () => y)          (* ! colon(x,E) in ML nicht zulässig ! *)

(* angenommen ML würde lazy evaluation durchführen *)
fun from n = n::(from (n+1))

(* Korrekte Darstellung in ML *)
fun from n = c(n,fn () => from(n+1))
```


Strings

```
"a string";
> val it = "a string" : string

explode it;
> val it = [#"a",#" ",#"s",#"t",#"r",#"i",#"n",#"g"] : char list

implode it;
> val it = "a string" : string
```


Gleichheit von Typen

Zwei Funktionen f,g: heißen
a) intensional gleich, wenn ihre Beschreibungen identisch sind
b) extensional gleich, wenn für alle x aus dem Definitionsbereich gilt: f(x) = g(x)

Bsp. extensional gleicher, aber intensional verschiedener Funktionen

```
fun double1 n:int = 2*n;
fun double2 n:int = n + n;
fun double3 n:int = 3*n-n;
```

Gleichheitstypen kann man induktiv definieren durch:
(1) int, bool, nat, real, char sind Gleichheitstypen.
(2) Jede Gleichheitstypvariable ''a ist ein Gleichheitstyp.
(3) Sind 'a1 ,..., 'aN Gleichheitstypen, so die mit den üblichen Typkonstruktoren definierten Typen (Bsp. 'a1 * ... * 'aN).

Polymorphe Gleichheitsfunktion **nur** auf Gleichheitstypen:
fun Eq x y = (x=y);   > val Eq = **fn : (''a -> ''a) -> bool**

Exceptions

```
1 div 0; >uncaught exception divide by zero

exception Head;                                    (* exception binding *)
> exception Head

fun head(nil) = raise Head | head (x::_) = x;
> val head = fn : 'a list -> 'a

head(nil);
> uncaught exception Head

fun head2 l = head(l) handle Head => 0;            (* exception handler *)
> val head2 = fn : int list -> int

head2(nil);
> val it = 0 : int
```


Abstrakte Typen

abstype d with b

Bsp.

```
abstype time = time of int * int
 with
  fun maketime(hrs, mins) =
       if hrs<0 orelse 23<hrs orelse mins <0 orelse 59<mins
       then raise BadTime
       else time(hrs, mins)
  and hours (time(t1, t2)) = t1
  and minutes(time(t1,t2)) = t2
 end;
> type time
> val maketime = fn : int * int -> time
> val hours = fn : time -> int
> val minutes = fn : time -> int
```

```
val t = maketime(8, 30);
> val t = - : time

(hours t, minutes t);
> val it = (8,30) : int * int
```


Imperative Eigenschaften von ML

(References and assignments)

```
x:=1;
> stdIn:11.1 Error: unbound variable or constructor: x

val x = ref 1;
> val x = ref 1 : int ref

x;
> val it = ref 1 : int ref

x:=6;
> val it = () : unit

!x;
> 6 : int
```