6 Das Modulsystem von OCAML

- → Strukturen
- → Signaturen
- → Information Hiding
- → Funktoren
- → Getrennte Übersetzung

6.1 Module oder Strukturen

Zur Strukturierung großer Programmsysteme bietet Ocaml Module oder Strukturen an:

```
module Pairs =
  struct
    type 'a pair = 'a * 'a
    let pair (a,b) = (a,b)
    let first (a,b) = a
    let second (a,b) = b
  end
```

Auf diese Eingabe antwortet der Compiler mit dem Typ der Struktur, einer Signatur:

```
module Pairs :
  sig
    type 'a pair = 'a * 'a
    val pair : 'a * 'b -> 'a * 'b
    val first : 'a * 'b -> 'a
    val second : 'a * 'b -> 'b
  end
```

Die Definitionen innerhalb der Struktur sind außerhalb nicht sichtbar:

```
# first;;
Unbound value first
```

Zugriff auf Komponenten einer Struktur:

Über den Namen greift man auf die Komponenten einer Struktur zu:

```
# Pairs.first;;
- : 'a * 'b -> 'a = <fun>
```

So kann man z.B. mehrere Funktionen gleichen Namens definieren:

```
# module Triples = struct
    type 'a triple = Triple of 'a * 'a * 'a
    let first (Triple (a,_,_)) = a
    let second (Triple (_,b,_)) = b
    let third (Triple (_,_,c)) = c
  end;;
...
```

```
...
module Triples :
sig
  type 'a triple = Triple of 'a * 'a * 'a
  val first : 'a triple -> 'a
  val second : 'a triple -> 'a
  val third : 'a triple -> 'a
end
# Triples.first;;
- : 'a Triples.triple -> 'a = <fun>
```

... oder mehrere Implementierungen der gleichen Funktion:

```
# module Pairs2 =
  struct
    type 'a pair = bool -> 'a
    let pair (a,b) = fun x -> if x then a else b
    let first ab = ab true
    let second ab = ab false
  end;;
```

Öffnen von Strukturen

Um nicht immer den Strukturnamen verwenden zu müssen, kann man alle Definitionen einer Struktur auf einmal sichtbar machen:

```
# open Pairs2;;
# pair;;
- : 'a * 'a -> bool -> 'a = <fun>
# pair (4,3) true;;
- : int = 4
```

Sollen die Definitionen des anderen Moduls Bestandteil des gegenwärtigen Moduls sein, dann macht man sie mit `include` verfügbar ...

```
# module A = struct let x = 1 end;;
module A : sig val x : int end
# module B = struct
    open A
    let y = 2
  end;;
module B : sig val y : int end
# module C = struct
    include A
    include B
  end;;
module C : sig val x : int val y : int end
```

Geschachtelte Strukturen

Strukturen können selbst wieder Strukturen enthalten:

```
module Quads  = struct
    module Pairs = struct
        type 'a pair = 'a * 'a
        let pair (a,b) = (a,b)
        let first (a,_) = a
        let second (_,b) = b
      end
    type 'a quad = 'a Pairs.pair Pairs.pair
    let quad (a,b,c,d) =
        Pairs.pair (Pairs.pair (a,b), Pairs.pair (c,d))
    ...
```

```
    ...
    let first q = Pairs.first (Pairs.first q)
    let second q = Pairs.second (Pairs.first q)
    let third q = Pairs.first (Pairs.second q)
    let fourth q = Pairs.second (Pairs.second q)
  end

# Quads.quad (1,2,3,4);;
- : (int * int) * (int * int) = ((1,2),(3,4))
# Quads.Pairs.first;;
- : 'a * 'b -> 'a = <fun>
```

6.2 Modul-Typen oder Signaturen

Mithilfe von Signaturen kann man einschränken, was eine Struktur nach außen exportiert.

Explizite Angabe einer Signatur gestattet:

- die Menge der exportierten Variablen einzuschränken;
- die Menge der exportierten Typen einzuschränken ...

... ein Beispiel:

```
module Sort = struct
    let single list = map (fun x->[x]) list
    let rec merge l1 l2 = match (l1,l2)
        with ([],_) -> l2
        |    (_,[]) -> l1
        | (x::xs,y::ys) -> if x<y then x :: merge xs l2
                                  else y :: merge l1 ys
    let rec merge_lists = function
            [] -> [] | [l] -> [l]
    | l1::l2::ll -> merge l1 l2 :: merge_lists ll
    let sort list = let list = single list
        in let rec doit = function
           [] -> [] | [l] -> l
        |   l -> doit (merge_lists l)
    in doit list
end
```

Die Implementierung macht auch die Hilfsfunktionen `single`, `merge` und `merge_lists` von außen zugreifbar:

```
# Sort.single [1;2;3];;
- : int list list = [[1]; [2]; [3]]
```

Damit die Funktionen `single` und `merge_lists` nicht mehr exportiert werden, verwenden wir die Signatur:

```
module type Sort = sig
    val merge : 'a list -> 'a list -> 'a list
    val sort : 'a list -> 'a list
  end
```

Die Funktionen `single` und `merge_lists` werden nun nicht mehr exportiert :-)

```
# module MySort : Sort = Sort;;
module MySort : Sort
# MySort.single;;
Unbound value MySort.single
```

Signaturen und Typen

Die in der Signatur angegebenen Typen müssen Instanzen der für die exportierten Definitionen inferierten Typen sein :-)

Dadurch werden deren Typen spezialisiert:

```
module type A1 = sig
    val f : 'a -> 'b -> 'b
  end
module type A2 = sig
    val f : int -> char -> int
  end
module A = struct
    let f x y = x
  end
```

```
# module A1 : A1 = A;;
Signature mismatch:
Modules do not match: sig val f : 'a -> 'b -> 'a end
                                  is not included in A1
Values do not match:
  val f : 'a -> 'b -> 'a
is not included in
  val f : 'a -> 'b -> 'b
# module A2 : A2 = A;;
module A2 : A2
# A2.f;;
- : int -> char -> int = <fun>
```

6.3 Information Hiding

Aus Gründen der Modularität möchte man oft verhindern, dass die Struktur exportierter Typen einer Struktur von außen sichtbar ist.

Beispiel:

```
module ListQueue = struct
  type 'a queue = 'a list
  let empty_queue () = []
  let is_empty = function
      [] -> true | _  -> false
  let enqueue xs y = xs @ [y]
  let dequeue (x::xs) = (x,xs)
end
```

Mit einer Signatur kann man die Implementierung einer Queue verstecken:

```
module type Queue = sig
  type 'a queue
  val empty_queue : unit -> 'a queue
  val is_empty : 'a queue -> bool
  val enqueue : 'a queue -> 'a -> 'a queue
  val dequeue : 'a queue -> 'a * 'a queue
end
```

```
# module Queue : Queue = ListQueue;;
module Queue : Queue
# open Queue;;
# is_empty [];;
This expression has type 'a list but is here used with type
  'b queue = 'b Queue.queue
```

$\Longrightarrow$

Das Einschränken per Signatur genügt, um die wahre Natur des Typs queue zu verschleiern :-)

Soll der Datentyp mit seinen Konstruktoren dagegen exportiert werden, wiederholen wir seine Definition in der Signatur:

```
module type Queue =
sig
   type 'a queue = Queue of ('a list * 'a list)
   val empty_queue : unit -> 'a queue
   val is_empty : 'a queue -> bool
   val enqueue : 'a -> 'a queue -> 'a queue
   val dequeue : 'a queue -> 'a option * 'a queue
end
```

6.4 Funktoren

Da in Ocaml fast alles höherer Ordnung ist, wundert es nicht, dass es auch Strukturen höherer Ordung gibt: die Funktoren.

- Ein Funktor bekommt als Parameter eine Folge von Strukturen;
- der Rumpf eines Funktors ist eine Struktur, in der die Argumente des Funktors verwendet werden können;
- das Ergebnis ist eine neue Struktur, die abhängig von den Parametern definiert ist.

Wir legen zunächst per Signatur die Eingabe und Ausgabe des Funktors fest:

```
module type Decons = sig
  type 'a t
  val decons : 'a t -> ('a * 'a t) option
end
module type GenFold = functor (X:Decons) -> sig
  val fold_left : ('b -> 'a -> 'b) -> 'b -> 'a X.t -> 'b
  val fold_right : ('a -> 'b -> 'b) -> 'a X.t -> 'b -> 'b
  val size : 'a X.t -> int
  val list_of : 'a X.t -> 'a list
  val app : ('a -> unit) -> 'a X.t -> unit
end
...
```

```
...
  module Fold : GenFold = functor (X:Decons) ->
  struct
  let rec fold_left f b t = match X.decons t
      with None -> b
      | Some (x,t) -> fold_left f (f b x) t
  let rec fold_right f t b = match X.decons t
      with None -> b
      | Some (x,t) -> f x (fold_right f t b)
  let size t = fold_left (fun a x -> a+1) 0 t
  let list_of t = fold_right (fun x xs -> x::xs) t []
  let app f t = fold_left (fun () x -> f x) () t
  end;;
```

Jetzt können wir den Funktor auf eine Struktur anwenden und erhalten eine neue Struktur ...

```
module MyQueue = struct open Queue
  type 'a t = 'a queue
  let decons = function
     Queue([],xs) -> (match rev xs
          with [] -> None
          | x::xs -> Some (x, Queue(xs,[])))
   | Queue(x::xs,t) -> Some (x, Queue(xs,t))
end

module MyAVL = struct open AVL
  type 'a t = 'a avl
  let decons avl = match extract_min avl
      with (None,avl) -> None
       | Some (a,avl) -> Some (a,avl)
end
```

```
module FoldAVL = GenFold (MyAVL)
module FoldQueue = GenFold (MyQueue)
```

Damit können wir z.B. definieren:

```
let sort list = FoldAVL.list_of (
                    AVL.from_list list)
```

Achtung:

Ein Modul erfüllt eine Signatur, wenn er sie implementiert !

Es ist nicht nötig, das explizit zu deklarieren !!

6.5 Getrennte Übersetzung

- Eigentlich möchte man Ocaml-Programme nicht immer in der interaktiven Umgebung starten :-)
- Dazu gibt es u.a. den Compiler `ocamlc` ...

```
> ocamlc Test.ml
```

  interpretiert den Inhalt der Datei `Test.ml` als Folge von Definitionen einer Struktur `Test`.
- Als Ergebnis der Übersetzung liefert `ocamlc` die Dateien:

`Test.cmo`	Bytecode für die Struktur
`Test.cmi`	Bytecode für das Interface
`a.out`	lauffähiges Programm

- Gibt es eine Datei `Test.mli` wird diese als Definition der Signatur für `Test` aufgefasst. Dann rufen wir auf:

```
> ocamlc Test.mli Test.ml
```

- Benutzt eine Struktur `A` eine Struktur `B`, dann sollte diese mit übersetzt werden:

```
> ocamlc B.mli B.ml A.mli A.ml
```

- Möchte man auf die Neuübersetzung von `B` verzichten, kann man `ocamlc` auch die vor-übersetzte Datei mitgeben:

```
> ocamlc B.cmo A.mli A.ml
```

- Zur praktischen Verwaltung von benötigten Neuübersetzungen nach Änderungen von Dateien bietet Linux das Kommando `make` an. Das Protokoll der auszuführenden Aktionen steht dann in einer Datei `Makefile` :-)