

- Um Pattern Matching zu benutzen, kann man `match ... with` für das entsprechende Argument einsetzen.
- Bei einem einzigen Argument bietet sich `function` an ...

```
# function None    -> 0
      | Some x    -> x*x+1;;
- : int option -> int = <fun>
```

Namenlose Funktionen werden verwendet, wenn sie nur **einmal** im Programm vorkommen. Oft sind sie **Argument für Funktionale**:

```
# map (fun x -> x*x) [1;2;3];;  
- : int list = [1; 4; 9]
```

Oft werden sie auch benutzt, um eine Funktion **als Ergebnis** zurückzuliefern:

```
# let make_undefined () = fun x -> None;;  
val make_undefined : unit -> 'a -> 'b option = <fun>  
# let def_one (x,y) = fun x' -> if x=x' then Some y  
                               else None;;  
val def_one : 'a * 'b -> 'a -> 'b option = <fun>
```

## 4 Praktische Features in Ocaml

- Ausnahmen
- Ein- und Ausgabe als Seiteneffekte
- Sequenzen

## 4.1 Ausnahmen (Exceptions)

Bei einem Laufzeit-Fehler, z.B. Division durch Null, erzeugt das Ocaml-System eine **exception** (Ausnahme):

```
# 1 / 0;;  
Exception: Division_by_zero.  
# List.tl (List.tl [1]);;  
Exception: Failure "tl".  
# Char.chr 300;;  
Exception: Invalid_argument "Char.chr".
```

Hier werden die Ausnahmen `Division_by_zero`, `Failure "tl"` bzw. `Invalid_argument "Char.chr"` erzeugt.

Ein anderer Grund für eine Ausnahme ist ein **unvollständiger Match**:

```
# match 1+1 with 0 -> "null";;
```

```
Warning: this pattern-matching is not exhaustive.
```

```
Here is an example of a value that is not matched:
```

```
1
```

```
Exception: Match_failure ("", 2, -9).
```

In diesem Fall wird die Exception `Match_failure ("", 2, -9)` erzeugt :-)

## Vordefinierte Konstruktoren für Exceptions:

`Division_by_zero`

`Invalid_argument` of string

`Failure` of string

`Match_failure` of string \* int \* int

`Not_found`

`Out_of_memory`

`End_of_file`

`Exit`

Division durch Null

falsche Benutzung

allgemeiner Fehler

unvollständiger Match

nicht gefunden :-)

Speicher voll

Datei zu Ende

für die Benutzerin ...

Eine Exception ist ein **First Class Citizen**, d.h. ein Wert eines Datentyps `exn ...`

```
# Division_by_zero;;  
- : exn = Division_by_zero  
# Failure "Kompletter Quatsch!";;  
- : exn = Failure "Kompletter Quatsch!"
```

Eigene Exceptions werden definiert, indem der Datentyp `exn` **erweitert** wird ...

```
# exception Hell;;  
exception Hell  
# Hell;;  
- : exn = Hell
```

```
# Division_by_zero;;  
- : exn = Division_by_zero  
# Failure "Kompletter Quatsch!";;  
- : exn = Failure "Kompletter Quatsch!"
```

Eigene Exceptions werden definiert, indem der Datentyp `exn` **erweitert** wird ...

```
# exception Hell of string;;  
exception Hell of string  
# Hell "damn!";;  
- : exn = Hell "damn!"
```



# Ausnahmebehandlung:

Wie in **Java** können Exceptions ausgelöst und behandelt werden:

```
# let teile (n,m) = try Some (n / m)
    with Division_by_zero -> None;;
```

```
# teile (10,3);;
- : int option = Some 3
# teile (10,0);;
- : int option = None
```

So kann man z.B. die `member`-Funktion neu definieren:

```

let rec member x l = try if x = List.hd l then true
                        else member x (List.tl l)
                    with Failure _ -> false

# member 2 [1;2;3];;
- : bool = true
# member 4 [1;2;3];;
- : bool = false

```

Das Schlüsselwort `with` leitet ein Pattern Matching auf dem Ausnahme-Datentyp `exn` ein:

```

try <exp>
with <pat1> -> <exp1> | ... | <patN> -> <expN>

```

⇒ Man kann mehrere Exceptions gleichzeitig abfangen :-)

Der Programmierer kann selbst Exceptions auslösen.

Das geht mit dem Schlüsselwort `raise ...`

```
# 1 + (2/0);;
```

```
Exception: Division_by_zero.
```

```
# 1 + raise Division_by_zero;;
```

```
Exception: Division_by_zero.
```

Eine Exception ist ein Fehlerwert, der jeden Ausdruck ersetzen kann.

Bei Behandlung wird sie durch einen anderen Ausdruck (vom richtigen Typ) ersetzt — oder durch eine andere Exception `;-)`

Exception Handling kann nach jedem beliebigen Teilausdruck, auch geschachtelt, stattfinden:

```
# let f (x,y) = x / (y-1);;
# let g (x,y) = try let n = try f (x,y)
                    with Division_by_zero ->
                        raise (Failure "Division by zero")
                    in string_of_int (n*n)
  with Failure str -> "Error: "^str;;

# g (6,1);;
- : string = "Error: Division by zero"
# g (6,3);;
- : string = "9"
```

## 4.2 Textuelle Ein- und Ausgabe

- Lesen aus der Eingabe und Schreiben auf die Ausgabe sprengt den rein funktionalen Rahmen !
- Diese Operationen werden darum mit Hilfe von Seiteneffekten realisiert, d.h. mit Hilfe von Funktionen, deren Rückgabewert uninteressant ist (etwa `unit`).
- Während der Ausführung wird dann aber die entsprechende Aktion ausgeführt  
⇒ nun kommt es genau auf die Reihenfolge der Auswertung an !!!

- Selbstverständlich kann man in **Ocaml** auf den Standard-Output schreiben:

```
# print_string "Hello World!\n";;  
Hello World!  
- : unit = ()
```

- Analog gibt es eine Funktion: `read_line : unit -> string`

...

```
# read_line ();;  
Hello World!  
- : "Hello World!"
```

Um aus einer **Datei zu lesen**, muss man diese zum Lesen **öffnen ...**

```
# let infile = open_in "test";;  
val infile : in_channel = <abstr>  
# input_line infile;;  
- : "Die einzige Zeile der Datei ...";;  
# input_line infile;;  
Exception: End_of_file
```

Gibt es keine weitere Zeile, wird die Exception **End\_of\_file** geworfen **:-)**

Benötigt man einen Kanal nicht mehr, sollte man ihn geregelt **schließen ...**

```
# close_in infile;;  
- : unit = ()
```

## Weitere nützliche Funktionen:

```
stdin           : in_channel
input_char      : in_channel -> char
in_channel_length : in_channel -> int
input : in_channel -> string -> int -> int -> int
```

- `in_channel_length` liefert die Gesamtlänge der Datei.
- `input chan buf p n` liest aus einem Kanal `chan n` Zeichen und schreibt sie ab Position `p` in den String `buf`  
:-)



Die **Ausgabe in Dateien** erfolgt ganz analog ...

```
# let outfile = open_out "test";;  
val outfile : out_channel = <abstr>  
# output_string outfile "Hello ";;  
- : unit = ()  
# output_string outfile "World!\n";;  
- : unit = ()  
...
```

Die einzeln geschriebenen Wörter sind mit Sicherheit in der Datei erst zu finden, wenn der Kanal geregelt **geschlossen wurde** ...

```
# close_out outfile;;  
- : unit = ()
```

## 4.3 Sequenzen

Bei Seiteneffekten kommt es auf die Reihenfolge an :-)

Mehrere solche Aktionen kann man mit dem **Sequenz-Operator** ; hintereinander ausführen:

```
# print_string "Hello";  
  print_string " ";  
  print_string "world!\n";;  
Hello world!  
- : unit = ()
```

Oft möchte man viele Strings ausgeben !

Hat man etwa eine Liste von Strings, hilft das Listenfunktional

`List.iter`: weiter:

```
# let rec iter f = function
  []      -> ()
| x::[]  -> f x
| x::xs  -> f x; iter f xs;;
```

```
val iter : ('a -> unit) -> 'a list -> unit = <fun>
```

## 5 Das Modulsystem von OCAML

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

## 5.1 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))
  ...
end
```