2.3 Wert-Definitionen

Mit `let` kann man eine Variable mit einem Wert belegen.

Die Variable behält diesen Wert für immer :-)

```
# let seven = 3+4;;
val seven : int = 7
# seven;;
- : int = 7
```

Achtung: Variablen-Namen werden klein geschrieben !!!

Eine erneute Definition für `seven` weist nicht `seven` einen neuen Wert zu, sondern erzeugt eine neue Variable mit Namen `seven`.

```
# let seven = 42;;
val seven : int = 42
# seven;;
- : int = 42
# let seven = "seven";;
val seven : string = "seven"
```

Die alte Definition wurde unsichtbar (ist aber trotzdem noch vorhanden :-)

Offenbar kann die neue Variable auch einen anderen Typ haben :-)

2.4 Komplexere Datenstrukturen

- Paare:

```
# (3,4);;
- : int * int = (3, 4)
# (1=2,"hallo");;
- : bool * string = (false, "hallo")
```

- Tupel:

```
# (2,3,4,5);;
- : int * int * int * int = (2, 3, 4, 5)
# ("hallo",true,3.14159);;
-: string * bool * float = ("hallo", true, 3.14159)
```

Simultane Definition von Variablen:

```
# let (x,y) = (3,4.0);;
val x : int = 3
val y : float = 4.

# val (3,y) = (3,4.0);;
val y : float = 4.0
```

Records: Beispiel:

```
# type person = {vor:string; nach:string; alter:int};;
type person = { vor : string; nach : string; alter : int; }
# let paul = { vor="Paul"; nach="Meier"; alter=24 };;
val paul : person = {vor = "Paul"; nach = "Meier"; alter = 24}
# let hans = { nach="kohl"; alter=23; vor="hans"};;
val hans : person = {vor = "hans"; nach = "kohl"; alter = 23}
# let hansi = {alter=23; nach="kohl"; vor="hans"}
val hansi : person = {vor = "hans"; nach = "kohl"; alter = 23}
# hans=hansi;;
- : bool = true
```

Bemerkung:

... Records sind Tupel mit benannten Komponenten, deren Reihenfolge irrelevant ist :-)

... Als neuer Typ muss ein Record vor seiner Benutzung mit einer `type`-Deklaration eingeführt werden.

... Typ-Namen und Record-Komponenten werden klein geschrieben :-)

Bemerkung:

... Records sind Tupel mit benannten Komponenten, deren Reihenfolge irrelevant ist :-)

... Als neuer Typ muss ein Record vor seiner Benutzung mit einer `type`-Deklaration eingeführt werden.

... Typ-Namen und Record-Komponenten werden klein geschrieben :-)

Zugriff auf Record-Komponenten

... per Komponenten-Selektion:

```
# paul.vor;;
- : string =  "Paul"
```

... mit Pattern Matching:

```
# let {vor=x;nach=y;alter=z} = paul;;
val x : string = "Paul"
val y : string = "Meier"
val z : int = 24
```

... und wenn einen nicht alles interessiert:

```
# let {vor=x} = paul;;
val x : string = "Paul"
```

Fallunterscheidung: `match` und `if`

```
match n
 with 0 -> "Null"
    | 1 -> "Eins"
    | _ -> "Soweit kann ich nicht zaehlen!"

match e
 with true  -> e1
    | false -> e2
```

Das zweite Beispiel kann auch so geschrieben werden (-:

```
if e then e1 else e2
```

Vorsicht bei redundanten und unvollständigen Matches!

```
# let n = 7;;
val n : int = 7
# match n 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 ("", 5, -13).
# match n
   with 0 -> "null"
      | 0 -> "eins"
      | _ -> "Soweit kann ich nicht zaehlen!";;
Warning: this match case is unused.
- : string = "Soweit kann ich nicht zaehlen!"
```

2.5 Listen

Listen werden mithilfe von `[]` und `::` konstruiert.

Kurzschreibweise: `[42; 0; 16]`

```
# let mt = [];;
val mt : 'a list = []
# let l1 = 1::mt;;
val l1 : int list = [1]
# let l = [1;2;3];;
val l : int list = [1; 2; 3]
# let l = 1::2::3::[];;
val l : int list = [1; 2; 3]
```

Achtung:

Alle Elemente müssen den gleichen Typ haben:

```
# 1.0::1::[];;
This expression has type int but is here used with type float
```

Achtung:

Alle Elemente müssen den gleichen Typ haben:

```
# 1.0::1::[];;
This expression has type int but is here used with type float
```

`tau list` beschreibt Listen mit Elementen vom Typ `tau` :-)

Der Typ `'a` ist eine Typ-Variable:

`[]` bezeichnet eine leere Liste für beliebige Element-Typen :-))

Pattern Matching auf Listen:

```
# match l
    with []     -> -1
       | x::xs -> x;;
-: int = 1
```

2.6 Definitionen von Funktionen

```
# let double x = 2*x;;
val double : int -> int = <fun>
# (double 3, double (double 1));;
- : int * int = (6,4)
```

→ Nach dem Funktionens-Namen kommen die Parameter.

→ Der Funktionsname ist damit auch nur eine Variable, deren Wert eine Funktion ist :-)

→ Alternativ können wir eine Variable einführen, deren Wert direkt eine Funktion beschreibt ...

```
# let double = fun x -> 2*x;;
val double : int -> int = <fun>
```

→ Diese Funktionsdefinition beginnt mit `fun`, gefolgt von den formalen Parametern.

→ Nach `->` kommt die Berechnungsvorschrift.

→ Die linken Variablen dürfen rechts benutzt werden :-)

Achtung:

Funktionen sehen die Werte der Variablen, die zu ihrem Definitionszeitpunkt sichtbar sind:

```
# let faktor = 2;;
val faktor : int = 2
# let double x = faktor*x;;
val double : int -> int = <fun>
# let faktor = 4;;
val faktor : int = 4
# double 3;;
- : int = 6
```

Achtung:

Eine Funktion ist ein Wert:

```
# double;;
- : int -> int = <fun>
```

Rekursive Funktionen:

Eine Funktion ist rekursiv, wenn sie sich selbst aufruft.

```
# let rec fac n = if n<2 then 1 else n * fac (n-1);;
val fac : int -> int = <fun>
# let rec fib = fun x -> if x <= 1 then 1
                         else fib (x-1) + fib (x-2);;
val fib : int -> int = <fun>
```

Dazu stellt Ocaml das Schlüsselwort `rec` bereit :-)

Rufen mehrere Funktionen sich gegenseitig auf, heißen sie verschränkt rekursiv.

```
# let rec even n = if n=0 then "even" else odd (n-1)
      and odd  n = if n=0 then "odd"  else even (n-1);;
val even : int -> string = <fun>
val odd : int -> string = <fun>
```

Wir kombinieren ihre Definitionen mit dem Schlüsselwort `and` :-)

Definition durch Fall-Unterscheidung:

```
# let rec len = fun x -> match x
                          with [] -> 0
                          | x::xs -> 1 + len xs;;
val len : 'a list -> int = <fun>
# len [1;2;3];;
- : int = 3
```