

Script generated by TTT

Title: Seidl: Info2 (24.11.2017)

Date: Fri Nov 24 08:32:16 CET 2017

Duration: 89:43 min

Pages: 27

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**.

Rekursive Funktionen

Eine Funktion ist **rekursiv**, wenn sie sich selbst (direkt oder indirekt) 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.

Definition durch Fall-Unterscheidung

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

Definition durch Fall-Unterscheidung

```
# let rec len = fun l -> match l
                    with [] -> 0
                     | x::xs -> 1 + len xs;;

val len : 'a list -> int = <fun>
# len [1;2;3];;
- : int = 3
```

... kann kürzer geschrieben werden als:

```
# let rec len = function [] -> 0
                    | x::xs -> 1 + len xs;;

val len : 'a list -> int = <fun>
# len [1;2;3];;
- : int = 3
```



164

Fall-Unterscheidung bei mehreren Argumenten

```
# let rec app l y = match l
                    with [] -> y
                     | x::xs -> x :: app xs y;;

val app : 'a list -> 'a list -> 'a list = <fun>
# app [1;2] [3;4];;
- : int list = [1; 2; 3; 4]
```

Handwritten red text:
~~# let a1 = app [1;2];;~~
~~# a1 [3;4];;~~
~~# a1 [4;3];;~~

165

Fall-Unterscheidung bei mehreren Argumenten

```
# let rec app l y = match l
                    with [] -> y
                     | x::xs -> x :: app xs y;;

val app : 'a list -> 'a list -> 'a list = <fun>
# app [1;2] [3;4];;
- : int list = [1; 2; 3; 4]
```

... kann auch geschrieben werden als:

```
# let rec app = function [] -> fun y -> y
                    | x::xs -> fun y -> x::app xs y;;

val app : 'a list -> 'a list -> 'a list = <fun>
# app [1;2] [3;4];;
- : int list = [1; 2; 3; 4]
```

166

Lokale Definitionen

Definitionen können mit **let** lokal eingeführt werden:

```
# let x = 5
  in let sq = x*x
    in sq+sq;;
- : int = 50

# let facit n = let rec
                iter m yet = if m>n then yet
                             else iter (m+1) (m*yet)
                in iter 2 1;;

val facit : int -> int = <fun>
```

Handwritten red text:
A: $n = 2$, $yet = 1$
 $if (m > n) then$ $return yet$
 $else$ $\{$ $yet = m * yet$
 $iter (m + 1) yet$
 $\}$ $return A$;

167

2.7 Benutzerdefinierte Typen

Beispiel: Spielkarten

Wie kann man die Farbe und den Wert einer Karte spezifizieren?

1. Idee: Benutze Paare von Strings und Zahlen, z.B.

```
("Schellen",10) ≡ Schellen Zehn  
("Eichel",11) ≡ Eichel Unter  
("Gras",14) ≡ Gras As
```

168

Nachteile

- Beim Test auf eine Farbe muss immer ein String-Vergleich stattfinden
→ ineffizient!
- Darstellung des Unter als 11 ist nicht intuitiv
→ unleserliches Programm!
- Welche Karte repräsentiert das Paar ("Eichel",9)?
(Tippfehler werden vom Compiler nicht bemerkt)

Besser: Aufzählungstypen von **Ocaml**.

169

2.7 Benutzerdefinierte Typen

Beispiel: Spielkarten

Wie kann man die Farbe und den Wert einer Karte spezifizieren?

1. Idee: Benutze Paare von Strings und Zahlen, z.B.

```
("Schellen",10) ≡ Schellen Zehn  
("Eichel",11) ≡ Eichel Unter  
("Gras",14) ≡ Gras As
```

168

2.7 Benutzerdefinierte Typen

Beispiel: Spielkarten

Wie kann man die Farbe und den Wert einer Karte spezifizieren?

1. Idee: Benutze Paare von Strings und Zahlen, z.B.

```
("Schellen",10) ≡ Schellen Zehn  
("Eichel",11) ≡ Eichel Unter  
("Gras",14) ≡ Gras As
```

168

Nachteile

- Beim Test auf eine Farbe muss immer ein String-Vergleich stattfinden
→ ineffizient!
- Darstellung des Unter als 11 ist nicht intuitiv
→ unleserliches Programm!
- Welche Karte repräsentiert das Paar ("Eichel",9)?
(Tippfehler werden vom Compiler nicht bemerkt)

Besser: Aufzählungstypen von **Ocaml**.

169

Vorteile

- Darstellung ist intuitiv.
- Tippfehler werden erkannt:

```
# (Eichel,Neun);;  
Unbound constructor Eichel
```

- Die interne Repräsentation ist **effizient**.

Bemerkungen

- Durch **type** wird ein **neuer Typ** definiert.
- Die Alternativen heißen **Konstruktoren** und werden durch `|` getrennt.
- Jeder Konstruktor wird groß geschrieben und ist **eindeutig** einem Typ zugeordnet.

171

Beispiel: Spielkarten

2. Idee: Aufzählungstypen

```
# type farbe = Schellen | Herz | Gras | Eichel;;  
type farbe = Schellen | Herz | Gras | Eichel  
# type wert = Sieben | Acht | Neun | Unter | Ober | Koenig |  
           Zehn | Ass;;  
type wert = Sieben | Acht | Neun | Unter | Ober | Koenig |  
           Zehn | Ass  
  
# Eichel;;  
- : farbe = Eichel  
# let gras_unter (Gras,Unter);;  
val gras_unter : farbe * wert = (Gras,Unter)
```

170

Beispiel: Spielkarten

2. Idee: Aufzählungstypen

```
# type farbe = Schellen | Herz | Gras | Eichel;;  
type farbe = Schellen | Herz | Gras | Eichel  
# type wert = Sieben | Acht | Neun | Unter | Ober | Koenig |  
           Zehn | Ass;;  
type wert = Sieben | Acht | Neun | Unter | Ober | Koenig |  
           Zehn | Ass  
  
# Eichel;;  
- : farbe = Eichel  
# let gras_unter (Gras,Unter);;  
val gras_unter : farbe * wert = (Gras,Unter)
```

170

Vorteile

- Darstellung ist intuitiv.
- Tippfehler werden erkannt:

```
# (Eichel,Neun);;
Unbound constructor Eichel
```
- Die interne Repräsentation ist **effizient**.

Bemerkungen

- Durch **type** wird ein **neuer Typ** definiert.
- Die Alternativen heißen **Konstruktoren** und werden durch `|` getrennt.
- Jeder Konstruktor wird groß geschrieben und ist **eindeutig** einem Typ zugeordnet.

171

Beispiel: Spielkarten

2. Idee: Aufzählungstypen

```
# type farbe = Schellen | Herz | Gras | Eichel;;
type farbe = Schellen | Herz | Gras | Eichel
# type wert = Sieben | Acht | Neun | Unter | Ober | Koenig |
Zehn | Ass;;
type wert = Sieben | Acht | Neun | Unter | Ober | Koenig |
Zehn | Ass

# Eichel;;
- : farbe = Eichel
# let gras_unter (Gras,Unter);;
val gras_unter : farbe * wert = (Gras,Unter)
```

170

Aufzählungstypen (cont.)

Konstruktoren können verglichen werden:

```
# Eichel < Schellen;;
- : bool = false;;
# Eichel > Schellen;;
- : bool = true;;
```

Pattern Matching auf Konstruktoren:

```
# let ist_Trumpf = function
    (Herz,_) -> true
    (_,Unter) -> true
    (_,Ober) -> true
    (_,_) -> false
```

172

Aufzählungstypen (cont.)

Konstruktoren können verglichen werden:

```
# Eichel < Schellen;;
- : bool = false;;
# Eichel > Schellen;;
- : bool = true;;
```

Pattern Matching auf Konstruktoren:

```
# let ist_Trumpf = function
    { (Herz,_) -> true
    | (_,Unter) -> true
    | (_,Ober) -> true
    | (_,_) -> false
```

172

```
val ist_Trumpf : farbe * wert -> bool = <fun>
```

Damit ergibt sich z.B.:

```
# ist_Trumpf (Gras,Unter);;
- : bool = true
# ist_Trumpf (Eichel,Neun);;
- : bool = false
```

Eine andere nützliche Funktion:

173

```
# let string_of_farbe = function
  Schellen -> "Schellen"
| Herz -> "Herz"
| Gras -> "Gras"
| Eichel -> "Eichel";;
val string_of_farbe : farbe -> string = <fun>
```

Beachte

Die Funktion `string_of_farbe` wählt für eine Farbe in **konstanter Zeit** den zugehörigen String aus (der Compiler benutzt – hoffentlich – **Sprungtabellen**).

174

Jetzt kann **Ocaml** schon fast Karten spielen:

```
# let sticht = function
  ((f1,Ober),(f2,Ober)) -> f1 > f2
  ((_,Ober),_) -> true
  (_,(_,Ober)) -> false
  ((f1,Unter),(f2,Unter)) -> f1 > f2
  ((_,Unter),_) -> true
  (_,(_,Unter)) -> false
  ((Herz,w1),(Herz,w2)) -> w1 > w2
  ((Herz,_),_) -> true
  (_,(Herz,_)) -> false
  ((f1,w1),(f2,w2)) -> if f1=f2 then w1 > w2
  else false;;
```

175

```
...
# let nimm (karte2,karte1) =
  if sticht (karte2,karte1) then karte2 else karte1;;

# let stich (karte1,karte2,karte3,karte4) =
  nimm (karte4, nimm (karte3, nimm (karte2,karte1)));;

# stich ((Gras,Ass),(Gras,Neun),(Herz,Zehn),(Eichel,Unter));;
- : farbe * wert = (Eichel,Unter)
# stich ((Eichel,Acht),(Eichel,Koenig),(Gras,Zehn),
  (Eichel,Neun));;
- : farbe * wert = (Eichel,Koenig)
```

176

```

...
# let nimm (karte2,karte1) =
  if sticht (karte2,karte1) then karte2 else karte1;;

# let stich (karte1,karte2,karte3,karte4) =
  nimm (karte4, nimm (karte3, nimm (karte2,karte1)));;

# stich ((Gras,Ass),(Gras,Neun),(Herz,Zehn),(Eichel,Unter));;
- : farbe * wert = (Eichel,Unter)
# stich ((Eichel,Acht),(Eichel,Koenig),(Gras,Zehn),
  (Eichel,Neun));;
- : farbe * wert = (Eichel,Koenig)

```

176

Char ist ein Modul, der Funktionalität für char sammelt.

Ein Konstruktor, der mit `type t = Con of <typ> | ...` definiert wurde, hat die Funktionalität `Con : <typ> -> t` — muss aber stets **angewandt** vorkommen ...

```

# Digit;;
The constructor Digit expects 1 argument(s),
but is here applied to 0 argument(s)
# let a = Letter 'a';;
val a : hex = Letter 'a'
# Letter 1;;
This expression has type int but is here used with type char
# hex2dez a;;
- : int = 10

```

178

Summentypen

$1A \rightarrow 10$ $1D \rightarrow 13$
 $1B \rightarrow 11$ $1E \rightarrow 14$
 $1C \rightarrow 12$ $1F \rightarrow 15$

Summentypen sind eine Verallgemeinerung von Aufzählungstypen, bei denen die Konstruktoren **Argumente** haben.

Beispiel: Hexadezimalzahlen

```

type hex = Digit of int | Letter of char;;
let char2dez c = if c >= 'A' && c <= 'F'
  then (Char.code c)-55
  else if c >= 'a' && c <= 'f'
  then (Char.code c)-87
  else -1;;
let hex2dez = function
  Digit n -> n
  | Letter c -> char2dez c;;

```

177

Char ist ein Modul, der Funktionalität für char sammelt.

Ein Konstruktor, der mit `type t = Con of <typ> | ...` definiert wurde, hat die Funktionalität `Con : <typ> -> t` — muss aber stets **angewandt** vorkommen ...

```

# Digit;;
The constructor Digit expects 1 argument(s),
but is here applied to 0 argument(s)
# let a = Letter 'a';;
val a : hex = Letter 'a'
# Letter 1;;
This expression has type int but is here used with type char
# hex2dez a;;
- : int = 10

```

178