Script generated by TTT

Title: Seidl: Info2 (02.12.2016)

Date: Fri Dec 02 08:23:44 CET 2016

Duration: 89:42 min

Pages: 26

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.

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.

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 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

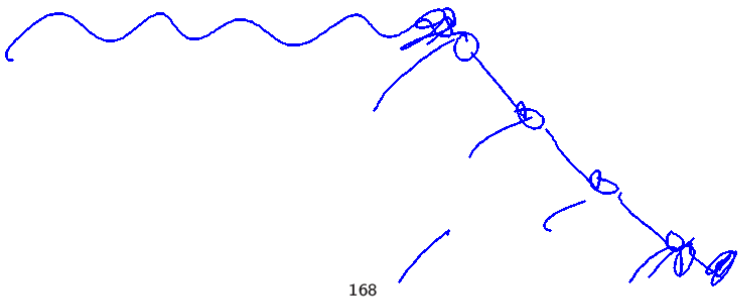
```
# let rec len = fun l -> match l
                         with [] -> 0
                         | x::xs -> 1 + len xs;;
val len : 'a list -> int = <fun>
# len [1;2;3];;
- : int = 3
```

... 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
```


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]
```



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]
```

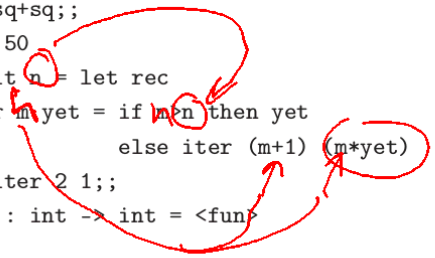
... kann auch geschrieben werden als:

```
# 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]
```

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>
```



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)`	$\equiv$	Schellen Zehn
`("Eichel",11)`	$\equiv$	Eichel Unter
`("Gras",14)`	$\equiv$	Gras As

Nachteile

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

Besser: Aufzählungstypen von Ocaml.

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)
```

Vorteile

- → Darstellung ist intuitiv.
- → Tippfehler werden erkannt:

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

- → 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.

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)
```


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
```


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
```

```
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:


```
# 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.

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;;
```


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;;
```

```
  ...
# 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)
```


Summentypen

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;;
```


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
```


Datentypen können auch rekursiv sein:

```
type folge = Ende | Dann of (int * folge)

# Dann (1, Dann (2, Ende));;
- : folge = Dann (1, Dann (2, Ende))
```

Beachte die Ähnlichkeit zu Listen!

Datentypen können auch rekursiv sein:

```
type folge = Ende | Dann of (int * folge)

# Dann (1, Dann (2, Ende));;
- : folge = Dann (1, Dann (2, Ende))
```

Beachte die Ähnlichkeit zu Listen!

Datentypen können auch rekursiv sein:

```
type folge = Ende | Dann of (int * folge)

# Dann (1, Dann (2, Ende));;
- : folge = Dann (1, Dann (2, Ende))
```

Beachte die Ähnlichkeit zu Listen!

Rekursive Datentypen führen wieder zu rekursiven Funktionen:

```
# let rec n_tes = function
                  (_,Ende) -> -1
                | (0,Dann (x,_)) -> x
                | (n,Dann (_, rest)) -> n_tes (n-1,rest);;
val n_tes : int * folge -> int = <fun>

# n_tes (4, Dann (1, Dann (2, Ende)));;
- : int = -1
# n_tes (2, Dann (1, Dann(2, Dann (5, Dann (17, Ende)))));;
- : int = 5
```