Kapitel 3. Funktionale Programmierung in OCAML

Arbeitsweise von OCAML

OCAML hat zwei Modi:

- Interaktiver Modus: Man gibt Definitionen ein; OCAML wertet sie aus und zeigt den Wert an.
- Compilierender Modus: Man schreibt ein OCAML Programm in eine oder mehrere Datei. Der OCAML Compiler übersetzt sie und liefert ein ausführbares Programm (“EXE-Datei”).

Wir befassen uns hauptsächlich und zunächst mit dem interaktiven Modus.

OCAML-Sitzung

Eröffnen einer OCAML Sitzung durch Eingabe von `ocaml`.

Es erscheint die Ausgabe:

```
        Objective Caml version 3.06

#
```

Das Zeichen # ist ein *Prompt*. Es fordert uns auf, eine Eingabe zu tätigen.

```
# let a = 18.35;;
val a : float = 18.35
# let aquadrat = a *. a;;
val aquadrat : float = 336.7225
# let b = -0.31 /. a;;
val b : float = -0.01689373297
```

Die Eingaben nach dem Prompt wurden vom Benutzer getätigt, die mit `val` beginnenden Zeilen sind Ausgaben von OCAML.

OCAML und Xemacs

Mit `xemacs` rufen Sie den Xemacs-Editor auf.

Jede Student(in) der Informatik sollte Xemacs kennen.

Mit dem Kommando `M-x shell` machen Sie ein Terminal Fenster auf.

Dort geben Sie das Kommando

```
export TERM=xemacs;ocaml
```

ein. Dann können Sie wie gewohnt mit OCAML arbeiten, haben aber die Möglichkeit, mit `M-p` und `M-n` die letzten Eingaben wieder heraufzuholen und die Kommandozeile zu editieren.

Hinweis: die Notation `M-x` bedeutet gleichzeitiges Drücken von `Alt` und `x`.

Fließkommaarithmetik in OCAML

Der OCAML Datentyp `float` umfasst *Fließkommazahlen*.

Das sind “reelle” Zahlen, wie sie im Taschenrechner vorkommen, z.B. `17.82, 2.3e-1, -25.1`.

Die Zahl der Nachkommastellen der *Mantisse* ist beschränkt (auf ca. 15). Der *Exponent* ist auch beschränkt (auf 304).

Grund: Eine `float` Zahl muss in 64bit passen.

Die arithmetischen Operationen $+, -, \times, /$ werden in OCAML als `+. -. *. /.` geschrieben.

Konstanten des Typs `float` haben entweder einen Dezimalpunkt oder ein `e` (großes `E` ist auch erlaubt).

`10` ist kein Element des Typs `float`, sondern ein Element des Typs `int`.

Vorsicht mit Rundungsfehlern

```
# let originalpreis = 3e14;; (* 300 Billiarden Euro *)
val originalpreis : float = 3e+14
# let sonderpreis = originalpreis -. 0.05;; (* 5ct Rabatt *)
val sonderpreis : float = 3e+14
# originalpreis -. sonderpreis;;
- : float = 0.0625 (* Ups *)
#
```

Kommentare

Kommentare werden in `(*...*)` eingeschlossen.

Sie haben auf den Programmablauf keinen Einfluss, dienen aber der Verständlichkeit und der Dokumentation.

Es gibt auch das *Rauskommentieren* (*commenting out*) von derzeit nicht benötigten Programmteilen.

Leider werden oft zuwenig Kommentare geschrieben.

Leider wird oft ein Programm verändert, die Kommentare aber nicht.

Ganze Zahlen in OCAML

```
val x : int = 0
# let x = 1729;;
val x : int = 1729
# let z = 13;;
val z : int = 13
# z * x;;
- : int = 22477
#
```

Datentyp `int` = $\{-2^{30}, \ldots, 2^{30} - 1\} \approx \mathbb{Z}$.

Operationen: `+, -, *, /, mod`.

Einen Datentyp `nat` gibt es nicht!

Überlauf

```
let guthaben = 800000000;;
guthaben * 2;;
- : int = -547483648
```

Grund: 1.6Mrd $> 2^{30}$.

Bei *Überlauf* wird einfach ganz unten (bei -2^{30}) wieder angefangen. Vgl. “Asteroids”.

Funktionen in OCAML

```
let f = function (x:float) -> (1./.x : float);;
val f : float -> float = <fun>
```

Typannotate sind (meistens) unnötig.

```
# let f = function x -> 1./.x;;
val f : float -> float = <fun>
# let mittel = function (x,y) -> (x +. y)/. 2.;;
val mittel : float * float -> float = <fun>
# let loesung = function (a,b,c) ->
      (-.b +. sqrt(b*.b -. 4.*.a*.c))/.(2.*.a);;
val loesung : float * float * float -> float = <fun>
#   loesung(1.,-1.,-1.);;
- : float = 1.61803398875
```

Alternative Notationen für Funktionen

Statt

```
# let f = function x -> function y -> function z -> x+y+z;;
```

auch

```
# let f = fun x y z -> x+y+z;;
val f : int -> int -> int -> int = <fun>
```

oder

```
# let f x y z = x+y+z;;
```

Natürlich auch mit nur einem Argument:

```
# let f(x,y,z) = x + y + z;;
```

Rekursion

```
# let rec fak = function n ->
        if n = 0 then 1 else n * fak (n-1);;
val fak : int -> int = <fun>
# fak 10;;
- : int = 3628800
# fak (-1);;
Stack overflow during evaluation (looping recursion?).
```

Alternativnotation

```
# let rec fak n =
        if n = 0 then 1 else n * fak (n-1);;
```

Die Fibonaccizahlen mit floats

```
# let rec fib n =
    if n = 0 || n = 1 then 1. else fib(n-1)+.fib(n-2);;
val fib : int -> float = <fun>
```

Alternative Definition

```
# let rec fib2 n = if n=0 then (1.,1.) else
                   let (u,v)=fib2(n-1) in (v,u+.v);;
val fib2 : int -> float * float = <fun>
```

Es ist `fib2` $n = (\texttt{fib}\, n, \texttt{fib}\, (n+1))$.

Beweis durch Induktion.

```
# fib 40;;
```

Braucht mehrere Minuten

```
# fib2 40;;
```

Braucht nur ein paar Millisekunden. Warum?

Verschränkte Rekursion

```
# let rec gerade x = if x = 0 then true else ungerade (x-1)
  and ungerade x = if x = 0 then false else gerade (x-1);;
val gerade : int -> bool = <fun>
val ungerade : int -> bool = <fun>
# gerade 10;;
- : bool = true
```

Mehrere `let`-Definitionen (insbesondere) rekursive können mithilfe von `and` zu einer einzigen zusammengefasst werden.

Bezeichner

Bezeichner (engl.: *identifier*) dienen der Bezeichnung von Abkürzungen (Definitionen) und Variablen.

Sie müssen in OCAML mit einem Kleinbuchstaben oder dem *underscore* (_) beginnen.

Danach dürfen beliebige Buchstaben, Zahlen, sowie die Symbole _ (Underscore) und `'` (*Hochkomma*, engl. *quote*) benutzt werden.

Legale Bezeichner: `eins`, `x789`, `zahlen_wert`, `gueteMass`, `__DEBUG`, `x'`

Illegale Bezeichner: `Eins`, `1a`, `zahlen-wert`, `GueteMass`.

Ich rate von der Verwendung von Umlauten und ß in Bezeichnern ab.

Schlüsselwörter

Bestimmte Wörter, die in Sprachkonstrukten verwendet werden sind auch nicht Bezeichner erlaubt. Dazu gehören z.B.:

```
and else false fun function if in let
```

Eine vollständige Liste findet sich in der OCAML Dokumentation.

```
# let else = 9;;
Syntax error
# let let = 1;;
Syntax error
# let fun = 8;;
Syntax error
#
 let false = 0;;
This expression has type int but is here used with type boo
```

Polymorphie und Funktionsparameter

```
# let komponiere g f = fun x -> g(f x);;
val komponiere : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun
# let rec iteriere f n = if n=0 then (fun x->x) else
      komponiere f (iteriere f (n-1));;
val iteriere : ('a -> 'a) -> int -> 'a -> 'a = <fun>
```

Beachte: Typvariablen beginnen mit Hochkomma (`'a`, `'b`,...)

Zeichenketten

Zeichenketten bilden den Datentyp `string`.

```
# let vorname = "Ibn Musa";;
val vorname : string = "Ibn Musa"
# let nachname = "Al Chwarizmi";;
# let name = vorname ^ nachname;;
val name : string = "Ibn MusaAl Chwarizmi"
# let name = vorname ^ " " ^ nachname;;
val name : string = "Ibn Musa Al Chwarizmi"
# String.length name;;
- : int = 21
```

Die Funktion `String.length` bestimmt die Länge einer Zeichenkette,

die Funktion ^ (infixnotiert) verkettet zwei Zeichenketten.

Die Verkettung heißt auch *Konkatenation* (von lat. *catena*=Kette).

Zeichen

Eine Zeichenkette ist aus Zeichen zusammengesetzt.

Die Zeichen (engl. *character*) bilden den Datentyp `char`.

Zeichenkonstanten werden in Hochkommata eingeschlossen:

```
# let z1 = 'a';;
val z1 : char = 'a'
# let z2 = '#';;
val z2 : char = '#'
# let z3 = '%';;
val z3 : char = '%'
#
```

Zeichen sind Buchstaben (`A-Za-z`), Ziffern (`0-9`), druckbare Sonderzeichen. Am besten nur

```
^!"$%&/()=?{[]}\''~@-_.,;:#+*|<>.+
```

Vorsicht mit ßäöü, usw.

Nicht druckbare Zeichen

Außerdem gibt es nicht druckbare Zeichen wie “Zeilenumbruch” (*newline*).

In OCAML notiert man dieses Zeichen `\n`.

Den “Rückschrägen” (*backslash*) notiert man `\\`.

Das Hochkomma notiert man `\'`.

Das Anführungszeichen notiert man `\"`.

Vergleichsoperationen

Die Vergleichsoperationen `<, >, =, <=, >=` haben in OCAML den Typ

```
'a * 'a -> bool
```

Bei `int, float` bezeichnen sie die übliche Ordnung.

Bei `bool` gilt `false < true`.

Bei `char` gilt die ASCII-Ordnung, siehe [Kröger, S.33], also etwa `'/' < '0' < 'Q' < 'q' < 'r' < '{'`.

Bei `string` gilt die *lexikographische Ordnung*.

Bei allen anderen Typen genügt es zu wissen, dass `<` eine *totale Ordnung* ist.

Bei Funktionstypen ist die Verwendung von Vergleichsoperationen ein Fehler.

Lexikografische Ordnung

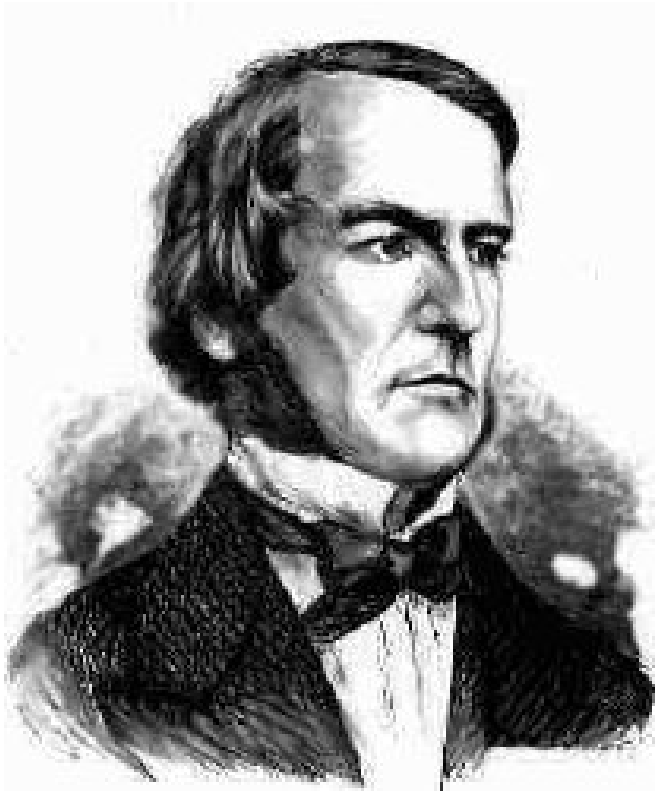
Seien $u = u_1, \ldots, u_m$ und $v = v_1, \ldots, v_n$ Zeichenketten (also $u_i, v_j \in$ `char`.

Es gilt $u < v$ gdw., entweder $m = 0$ und $n > 0$ oder $u_1 < v_1$ oder $u_1 = v_1$ und $u_2, \ldots, u_m < v_2, \ldots, v_n$.

Z.B.

```
"AAAAAAA" < "Anderer Schluesseldienst"
"Martin" < "Martina"
"Lexikografisch" < "Lexikon"
```

Boole'sche Operationen



GEORGE BOOLE 1815–1864

Pseudocode	OCAML
$\wedge$	`&&`
$\vee$	`||`
$\neg$	`not`

Schaltjahre sind durch 4 teilbar aber nicht durch 100 es sei denn durch 400.
Übung: Man schreibe einen Ausdruck, der `true` ist gdw. `jahr` ein Schaltjahr ist.

Zusammenfassung Basistypen, Basisfunktionen

Arithmetische Operationen

`+, -, *`	`int*int->int`	
`/`	`int*int->int`	Ganzzahlige Division
`mod`	`int*int->int`	Rest. Infixnotiert.
`+., -., *.,/.`	`float*float->float`	

Vergleichsoperationen

`=,<,>,<=,>=,<>`	`'a*'a->bool`	Undefiniert für Funktionstypen
Boolesche Operationen		`<>`: ungleich
`not`	`bool->bool`	
`||,&&`	`bool*bool->bool`	

Konkatenation

`^`	`string*string->string`

Für weitere Operationen siehe OCAML Standard Library

Dateieingaben

Sie können eine Folge von OCAML Eingaben auch in eine Datei schreiben.

Dabei dürfen Sie die `;;` weglassen.

Diese Datei können Sie dann mit

`# #use` *Dateiname*`;;`

laden. Der Effekt ist derselbe, als hätten Sie alle Eingaben von Hand getätigt.

Es empfiehlt sich die Dateiendung `.ml`