# *Grundlagen der Programmierung*

Dr. Christian Herzog
Technische Universität München

Wintersemester 2006/2007

## *Ergänzung zu Kapitel 5: Programmieren mit OCaml*

## *Einführung in OCaml*

- Verwendung des Interpreters
- Einfache Ausdrücke und Funktionen
- Funktionen als Parameter (Funktionen höherer Ordnung)
- Pattern Matching
- Der selbst definierte rekursive Datentyp intSequenz
- Übertragung einiger Funktionen aus Kapitel 5



## *Verwendung des Interpreters*

- Ocaml ist kostenlos verfügbar unter http://www.ocaml.org/
- Es steht für verschiedene Plattformen zur Verfügung (Unix, Linux, Windows, Mac).
- Es gibt zwei Möglichkeiten, Programme auszuführen:
  - Übersetzen und starten (ähnlich wie bei Java)
  - Interaktiv mit dem Interpreter arbeiten (ähnlich wie mit einem Taschenrechner)
- Start des Interpreters (unter Unix) durch Eingabe des Kommandos **`ocaml`**.
- Der Interpreter meldet sich:

```
        Objective Caml version 3.07+2

#
```

Eingabeaufforderung des Interpreters



## *Verwendung des Interpreters (Fortsetzung)*

- Nun können Vereinbarungen (Typvereinbarungen oder Funktionsvereinbarungen) oder auszuwertende Ausdrücke eingegeben werden.
- Jede Eingabe wird mit „`;;`“ abgeschlossen:

```
        Objective Caml version 3.07+2

# 1 + 2 * 3 ;;
- : int = 7
# 7.0 +. 4.0 ;;
- : float = 11.
# let pi =
    4.0 *. atan 1.0 ;;
- val pi : float = 3.14159265358979312
#
```

Eingabe desAnwenders

Ergebnismeldung des Interpreters und neue Eingabeaufforderung

Addition auf Gleitpunktzahlen

Vereinbarung einer nullstelligen Funktion (Konstanten)

## *Verwendung des Interpreters (Fortsetzung)*

- Vereinbarungen und Ausdrücke können auch in Dateien vorbereitet werden und mittels **#use** eingelesen werden:

```
# #use "ocaml_051115.ml" ;;
```

- Beendet wird der Interpreter mit **#quit**:

```
# #quit ;;
```



## *Einfache Funktionen*

```
# let square x =
    x *. x ;;
- val square : float -> float = <fun>
# square (sin pi) +. square (cos pi) ;;
- : float = 1.
# let add a b =
    a + b ;;
- val add : int -> int -> int = <fun>
# add 24 56 ;;
- : int = 80
# let mult a b =
    a * b ;;
- val mult : int -> int -> int = <fun>
#
```

Formaler Parameter

Currying!



## *Rekursive Funktionen*

Leitet rekursive Funktionen ein.

- Unsere bekannte Summe:

```
let rec summe n =
   if n = 0 then 0
            else n + summe (n-1) ;;
```

- Nach demselben Schema die Fakultät:

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



## *Funktionen höherer Ordnung*

- Zwei Funktionen mit demselben Schema

```
let rec summe n =
   if n = 0 then 0
            else n + summe (n-1) ;;
```

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

Funktionen höherer Ordnung haben Funtionen als Parameter

- Wir implementieren nun dieses Schema:

```
let rec schema op start n =
   if n = 0 then start
            else op n (schema op start (n-1)) ;;
```

- Damit lassen sich nun wieder summe und fak implementieren:

```
let summe = schema add 0 ;;
let fak = schema mult 1 ;;
```

## *„Pattern Matching“ bei Sequenzen als Parameter*

- Die Funktion first lässt sich mittels Pattern Matching definieren:

```
let first s =
   match s with
     Stock(z,r) -> z ;;
```

- Der neue Parameter r wird nicht benötigt; er kann durch die so genannte *wild card* „_“ ersetzt werden:

```
let first s =
   match s with
     Stock(z,_) -> z ;;
```

- Analog für rest:

```
let rest s =
   match s with
     Stock(_,r) -> r ;;
```



## *die Funktion istPraefix*

- Die Funktion istPraefix in Java (Kapitel 5):

```
boolean istPraefix (IntSequenz a, IntSequenz s) {
  return isEmpty(a) ? true
         : isEmpty(s) ? false
           : first(a) != first(s) ? false
             : istPraefix (rest(a), rest(s)) ;
}
```

- Und nun in OCaml:

```
let rec istPraefix a s =
  match a, s with
    (Create, _) -> true
  | (_, Create) -> false
  | (Stock(z1,r1),Stock(z2,r2)) when z1=z2 -> istPraefix r1 r2
  | _ -> false ;;
```

Das Muster kann noch durch die Angabe einer Bedingung ergänzt werden.

Die *wild card* steht hier für ein beliebiges Paar und fängt als letztes Muster alle übrigen Fälle ab



## *die Funktion istTeilsequenz*

- Die Funktion istTeilsequenz in Java (Kapitel 5):

```
boolean istTeilsequenz (IntSequenz t, IntSequenz s) {
  return isEmpty(t) ? true
         : isEmpty(s) ? false
           : istPraefix(t,s) ? true
             : istTeilsequenz(t, rest(s)) ;
}
```

- Und nun in OCaml:

```
let rec istTeilsequenz t s =
  match t, s with
    (Create, _) -> true
  | (_, Create) -> false
  | (a,b) when istPraefix a b -> true
  | (a,Stock(_,r)) -> istTeilsequenz a r ;;
```



## *Die eingebettete Funktion minimum*

- Die Funktion minimum in Java (Kapitel 5):

```
private int minimumBett (IntSequenz s, int altesMin) {
   return isEmpty(s) ? altesMin
          : first(s) < altesMin
            ? minimumBett(rest(s), first(s))
            : minimumBett (rest(s), altesMin) ; }
public int minimum (IntSequenz s) {
   return minimumBett(rest(s), first(s));}
```

- Und nun in OCaml:

```
let rec minimum s =
  let rec minimumBett s altesMin =
     match s, altesMin with
       (Create, m) -> m
     | (Stock(z,r), m) when z<m -> minimumBett r z
     | (Stock(_,r), m) -> minimumBett r m
  in
  match s with
    Stock(z,r) -> minimumBett r z;;
```