

Teil XII

Funktionale Programmiersprachen

Teil XII.1

Überblick

Was macht eine "Funktionale" Programmiersprache aus?

- Funktionen als "first class values"
 - Funktionen höherer Ordnung
 - Currying von Funktionen
- Striktes Typsystem
 - Typinferenz
 - Typechecking
 - Parametrisierte Typen (Polymorphismus)
 - Patternmatching
- Freiheit von Seiteneffekten

Beispiel

- Imperative Programmiersprache: (Pascal)

```
j:=1;  
for i:=1 to LineLength do  
  if line[i] < 10 then  
    begin newline[j]:= line[i]; j:=j+1 end;
```

- Funktionale Programmiersprache: (OCaml)

```
filter (lessthan 10) int_sequence  
filter (notequal "fred") string_sequence
```

Entwicklung von Funktionalen Programmiersprachen

- **Lisp**: Listprocessing auf Basis des Lambda-Kalküls
Weiterentwicklung: Scheme
- **FP**: Functional Programming Sprache in Turing Award Rede von Backus.
- **ML**: Metalanguage. Starkes Typsystem
Derivate
 - Gofer
 - Miranda
 - Gofer
 - Haskell
 - Caml light und OCaml

Teil XII.2

Variablen und Funktionen

Variablen

- Syntax:

```
"let" name = expr ["in" expr]
```

- Beispiele:

```
# let x = 1;;  
val x : int = 1      (* Der Typ int wird inferiert *)  
# let y = 2;;  
val y : int = 2  
# let z = x + y;;  
val z: int = 3  
# let x = 1 in  
  let y = 2 in x + y;;  
- : int = 3          (* Das Ergebnis ist an keine explizite  
                     Variable gebunden *)
```

Lexikalisches Scoping

Der Wert einer Variable ist jeweils der Wert des innersten let Ausdrucks.

```
# let x = 1 in
  let x = 2 in          (* x wird innerhalb des in-Ausdrucks
                        ueberschrieben *)
    let y = x + x in   (* y = 2 + 2 = 4 *)
      x + y;;         (* 2 + 4 *)
- : int = 6
```

```
# let x = 1;;          (* x wird fuer das gesamte
                       folgende Programm gebunden *)
val x : int = 1
  let z =
    let x = 2 in      (* x wird innerhalb des in-Ausdrucks
                      ueberschrieben *)
      let x = x + x in (* x = 2 + 2 = 4 *)
        x + x;;       (* 4 + 4 *)
  val z: int = 8;;
```

Datentypen

Boolesche Werte, Fest- und Gleitpunktzahlen

```
# false;;  
- : bool = false  
# 1;;  
- : int = 1  
# 1.5;;  
- : float = 1.5
```

Operatoren:

```
bool: && || not  
int: + - * / mod  
float: +. -. *. /.
```

```
# 1. +. 3.;;  
- : float = 4.  
# 4.0E-1 *. 5.;;  
- : float = 2.
```

Aufzählungstypen

(Spezialfall der Union Types)

```
# type farbe = Rot | Gruen | Gelb | Blau | Schwarz | Weiss;;  
type farbe = Rot | Gruen | Gelb | Blau | Schwarz | Weiss
```

Operatoren: nur Test auf (Un)Gleichheit

```
# Rot = Gruen;;  
- : bool = false  
# Rot != Blau;;  
- : bool = true
```

Zeichen und Zeichenketten

```
# 'a';;  
- : char = 'a'  
# "string";;  
- : string = "string"
```

Operatoren: ^ sub index length

```
# open String;;  
...  
# "ab" ^ "bcd";;  
- : string = "abbcd"  
# sub "abcde" 2 2;;  
- : string = "cd"  
# index "abcde" 'd';;  
- : int = 3  
# length "abcde";;  
- : int = 5
```

Funktionen

- Syntax:

```
"fun" {par} "->" expr
```

- Beispiele:

```
# let incr = fun i -> i + 1;;  
val incr : int -> int = <fun> (* int -> int ist der Typ aller Funktionen  
                             von int nach int *)
```

```
# incr 2;;  
- : int = 3  
# incr 1 3;; (* Typfehler 1 *)
```

```
Toplevel input:  
# incr 1 3;;
```

This function is applied to too many arguments

```
# incr "a";; (* Typfehler 2 *)
```

```
Toplevel input:  
# incr "a";;
```

This expression has type string but is here used with type int

Funktionen als "first class values"

- Funktionen sind ganz normale Werte.
- Funktionen können wie normale Werte als Parameter an andere Funktionen (höherer Ordnung) übergeben werden
- Funktionen können Rückgabewert einer Funktion sein.

Currying

Jede Funktion $f: (\text{typ1}, \dots, \text{typn}) \rightarrow \text{rtyp}$ kann auch als Funktion $f: \text{typ1} \rightarrow (\text{typ2} \rightarrow \dots (\text{typn} \rightarrow \text{rtyp}) \dots)$ dargestellt werden.

f wird angewandt auf typ1 und gibt eine Funktion $f1: \text{typ2} \rightarrow \dots (\text{typn} \rightarrow \text{rtyp}) \dots$ zurück.

```
# let sum = fun i j -> i + j;;  
val sum: int -> int -> int = <fun>
```

```
(* sum: int -> (int -> int) ist eine Funktion mit einem  
   int Argument und einer Funktion vom Typ (int -> int)  
   als Rueckgabewert (currying) *)
```

```
# sum 3 4;;  
- : int = 7
```

```
# let incr = sum 1;;  
val incr: int -> int = <fun>
```

```
(* Das Ergebnis von sum: int -> (int -> int) angewandt auf  
   die Konstante 1, ist eine Funktion mit Typ int -> int *)
```

```
# incr 5;;  
- : int = 6
```

Funktionen höherer Ordnung (1)

Funktionen, die als Parameter Funktionen erhalten, heißen Funktionen höherer Ordnung.

Beispiel: Approximation der numerischen Ableitung einer beliebigen Funktion:

```
# let deriv f =  
  
  (* f ist der funktionswertige Parameter von derivation *)  
  let dx = 1e-10 in  
    (fun x -> (f (x +. dx) -. f x) /. dx);;  
  
  (* fun x ist eine "anonyme" Funktion mit einem Parameter  
    +., -. und /. sind Addition, Subtraktion und Multiplikation  
    fuer float; keine Ueberladung wie in C oder Pascal *)  
  
val deriv : (float -> float) -> float -> float = <fun>  
  
  (* deriv ist eine Funktion, die als Eingabe eine Funktion vom  
    Typ float -> float erwartet und als Ausgabe ebenfalls eine  
    Funktion vom Typ float -> float ergibt *)
```

Funktionen höherer Ordnung (2)

```
# let g = (fun x -> x *. x *. x);;  
val g : float -> float = <fun>  
  
# let g' = deriv g;;  
val g' : float -> float = <fun>  
# g' 10.0;;  
- : float = 300.000237985 (* ~ 3*10.0^2 *)  
# g' 5.0;;  
- : float = 75.0000594962 (* ~ 3*5.0^2 *)  
  
# let g'' = deriv g';;  
val g'' : float -> float = <fun>  
# g'' 10.0;; (* numerischer Fehler *)  
- : float = 0  
  
# let f'' = deriv (fun x-> 3.0 *. x *. x);;  
val f'' : float -> float = <fun>  
# f'' 10.0;;  
- : float = 59.9999339101 (* ~6*10.0 *)  
# f'' 5.0;;  
- : float = 29.9999669551 (* ~6*5.0 *)
```

Funktionen höherer Ordnung (3)

Komposition von Funktionen:

```
# let compose = fun f g x -> g (f (x));;
val compose : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c = <fun>
(* Polymorphe Typen:
   Eingabe: Funktionen f: 'a -> 'b, g: 'b -> 'c
   Ausgabe: Funktion f o g: 'a -> 'c;
   wobei 'a, 'b, 'c beliebige konkrete Typen sein koennen *)

# let identity = compose sqrt (fun x -> x**2.);;
val identity : float -> float = <fun>
(* 'a, 'b und 'c werden "instanziiert" mit float *)
# identity 2.0;;
- : float = 2.
```

Rekursive Funktionen

Rekursive Funktionen müssen explizit mit `rec` vereinbart werden

```
# let rec power =  
  fun i x ->  
    if i =0  
    then 1.0  
    else x *. (power (i-1) x);;  
val power : int -> float -> float = <fun>  
  
# power 2 4.0;;  
- : float = 16
```

Verschränkte Rekursion

```
# let rec f i j =  
  
(* Alternative Syntax fuer:  
   let rec f = fun i j -> *)  
  
    if i = 0 then  
      j  
    else  
      g (j-1)  
and g j =  
    if j mod 3 = 0 then  
      j  
    else  
      f (j-1) j;;  
val f : int -> int -> int = <fun>  
val g : int -> int = <fun>  
  
# g 7;;  
- : int = 6  
# g 5;;  
- : int = 3
```

Teil XII.3

Pattern Matching

Pattern Matching

- Syntax:

```
"match" expr "with  
  pattern -> expr  
  {pattern -> expr}
```

- Beispiel:

```
# let rec fib i =  
  match i with  
    0 -> 1  
  | 1 -> 1  
  | n -> fib (n-2) + fib (n-1);;  
val fib : int -> int = <fun>  
# fib 1;;  
- : int = 1  
# fib 2;;  
- : int = 2  
# fib 3;;  
- : int = 3  
# fib 5;;  
- : int = 8
```

Syntaktische Abkürzungen

- Patterns für Funktionen

```
# let rec fib = function
  0 -> 1
  | 1 -> 1
  | i -> fib (i-1) + fib (i-2);;
val fib : int -> int = <fun>
```

- Bereichspatterns

```
# let is_uppercase = function
  'A' .. 'Z' -> true
  | _ -> false;;
val is_uppercase : char -> bool = <fun>
# is_uppercase 'D';;
- : bool = true
# is_uppercase 'a';;
- : bool = false
# is_uppercase "a";;
Toplevel input:
# is_uppercase "a";;
```

This expression has type string but is here used with type char

Unvollständige Matches

```
# let is_uppercase = function
  'A' .. 'Z' -> true;;
Toplevel input:
# let is_uppercase = function
# .....
  'A' .. 'Z' -> true;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
'a'
val is_uppercase : char -> bool = <fun>

# is_uppercase 'a';;
Exception: Match_failure ("", 19, 50).
```

Teil XII.4

Polymorphismus

Tupel

Listen

Parametrisierte Typen

- Parameter-Polymorphismus in OCaml

```
# let identity x = x;;  
val identity : 'a -> 'a = <fun>  
# identity 1;;  
- : int = 1  
# identity "Hello";;  
- : string = "Hello";;  
# identity (fun x-> x+1);;  
- : int -> int = <fun>
```

- Vergleich zu C

```
int int_identity(int i) { return i; }  
  
char* string_identity(char* s) {return s;}
```

Parameter Polymorphism vs. Adhoc Polymorphism

- Addition in OCaml

```
# let addint x y = x + y;;  
val addint : int -> int -> int = <fun>  
# let addfloat x y = x +. y;;  
val addfloat : float -> float -> float = <fun>  
# let addstring x y = x ^ y;;  
val addstring : string -> string -> string = <fun>
```

- Addition in Java

```
class Adder {  
    int Add(int i, int j) {return i + j;}  
    float Add(float i, float j) {return i + j}  
    String Add(String i, String j) {return i.concat(j);}  
}
```

Tupel (1)

```
# type coord = float * float;;
type coord = float * float
(* Syntax fuer Tupel-Typen: typ * typ *)

# let make_coord x y = x , y;;
val make_coord : 'a -> 'b -> 'a * 'b = <fun>
(* Syntax fuer Tupel-Konstruktion: expr, expr *)

# let x_of_coord (x,_) = x;;
val x_of_coord : 'a * 'b -> 'a = <fun>
(* Syntax fuer Tupel-Dekonstruktion (Pattern):
   variable,variable *)

# let y_of_coord (_,y) = y;;
val y_of_coord : 'a * 'b -> 'b = <fun>
# let a = make_coord 1.0 2.0;;
val a : float * float = 1.0, 2.0
# x_of_coord a;;
- : float = 1.0
```

Tupel (2)

```
# let make_polar_coord (x,y) =  
  sqrt x *. x +. y *. y,  
  atan y /. x ;;  
val make_polar : float * float -> float * float = <fun>
```

```
# make_polar a;;  
- : float * float = 5, 1.10714871779
```

```
# make_polar 1.0 2.0;;  
Toplevel input:  
# make_polar 1.0 2.0;;
```

This function is applied to too many arguments

Listen

```
# let rec sum = function
  [] -> 0
  (* Leere Liste *)
  | head :: tail -> head + sum tail;;
  (* Dekonstruktion und Konstruktion *)
val sum : int list -> int = <fun>
  (* Syntax: typ "list" *)

# sum [1;2;3;4;5];;
  (* Flache Schreibweise *)
- : int = 15
```

Polymorphe Listen

```
# let rec member element list =
  match list with
  [] -> false
  | head :: tail -> element = head || member element tail;;
val member : 'a -> 'a list -> bool = <fun>
(* member ist auf Listen mit beliebigem Elementtyp 'a definiert,
   allerdings muss der Typ von element gleich dem Elementtyp der
   Liste sein *)

# member "Thomas" ["Elke"; "Anna"; "Paul"; "Thomas"; "Peter"; "Mary"];;
- : bool = true

# member 4 [1;3;2;5;2];;
- : bool = false
# member "Thomas" [1;2;4;5];;
Toplevel input:
# member "Thomas" [1;2;4;5];;
      ^

This expression has type int but is here used with type string
```

Funktionen höherer Ordnung auf Listen

- **Map:** Wende eine Funktion auf alle Elemente einer Liste an

```
# let rec map f = function
  [] -> []
  | head::tail -> f head :: map f tail;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
# map incr [1;3;5;2];;
- : int list = [2; 4; 6; 3]
# map (fun x -> "Mc" ^ x) ["Gregor"; "Loud"; "Queen"; "Donald"];;
- : string list = ["McGregor"; "McLoud"; "McQueen"; "McDonald"]
```

- **Filter:** Filtere eine Liste von Elementen mit einem Prädikat

```
# let rec filter p = function
  [] -> []
  | h::t -> if p h then h::filter p t else filter p t;;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

```
# filter (fun x -> x < 10) [17; 13; 4; 2; 15; 9];;
- : int list = [4; 2; 9]
# filter (fun x -> x <> "McLoud") ["McGregor"; "McLoud"; "McQueen"; "McDonald"];;
- : string list = ["McGregor"; "McQueen"; "McDonald"]
```

Fold

```
# let rec fold f x list =  
  match list with  
  | [] -> x  
  | head::tail -> f head (fold f x tail);;  
val fold : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

```
# fold (+) 0 [1;2;3;5;7];;  
- : int = 18  
# fold (^) "" ["Was "; "kommt "; "hier "; "raus?"];;  
- : string = "Was kommt hier raus?"  
# fold (@) [] [[1;2;3;5];["this"]];;  
Toplevel input:  
# fold (@) [] [[1;2;3;5];["this"]];;
```

This expression has type string but is here used with type int

```
# fold (@) [] [[1;2;3;5];[7;8;9];[13;14]];;  
- : int list = [1; 2; 3; 5; 7; 8; 9; 13; 14]
```

```
# let rec intlist x =  
  if x = 0 then [] else intlist (x-1) @ [x];;  
val intlist : int -> int list = <fun>  
# fold ( * ) 1 (intlist 12);;  
- : int = 479001600
```

Quicksort in OCaml

```
# let rec qsort order x =
  match x with
  | []          -> []
  | head::tail ->
    let lesser =
      qsort order (filter (fun x -> order x head) tail)
    and greater =
      qsort order (filter (fun x -> not (order x head)) tail) in
    lesser @ [head] @ greater;;
val qsort : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
(* qsort erwartet eine boolesche Funktion vom Typ 'a -> 'a -> bool,
   und eine Liste vom Typ 'a list als Input und gibt
   eine Liste vom Typ 'a list zurueck. *)

# qsort (<) [17; 13; 4; 2; 15; 9];;
- : int list = [2; 4; 9; 13; 15; 17]
# qsort (>) ["McGregor"; "McLoud"; "McQueen"; "McDonald"];;
- : string list = ["McQueen"; "McLoud"; "McGregor"; "McDonald"]
# qsort (fun x y -> String.length x < String.length y)
  ["McDonald"; "McGregor"; "McLoud"; "McQueen"];;
- : string list = ["McLoud"; "McQueen"; "McDonald"; "McGregor"]
```

Quicksort etwas effizienter

```
# let rec split order elm list =
  match list with
  | [] -> [], []
  | head::tail ->
    match split order elm tail with
    | lesser, greater ->
      if order head elm then
        head::lesser, greater
      else
        lesser, head::greater;;
val split : ('a -> 'b -> bool) -> 'a -> 'b list -> 'b list * 'b list = <fun>

# let rec qsort1 order list =
  match list with
  | [] -> []
  | head::tail ->
    match split order head tail with
    | lesser, greater ->
      (qsort1 order lesser) @ [head] @ (qsort1 order greater);;
val qsort1 : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
```

Teil XII.5

Union Types

Binäre Bäume

```
# type 'a bbaum =  
  (* "Knoten" und "Blatt" sind Konstruktoren *)  
  Knoten of 'a bbaum * 'a * 'a bbaum  
  | Blatt;;  
  
type 'a bbaum = Knoten of 'a bbaum * 'a * 'a bbaum | Blatt  
  
(* Ein bbaum ist ein "Knoten" oder ein "Blatt" *)  
  
# Blatt;;  
- : 'a bbaum = Blatt  
# Knoten (Blatt, 1, Blatt);;  
- : int bbaum = Knoten (Blatt, 1, Blatt)
```

Patternmatching mit Union Types

```
# let rec anzahl = function
  Blatt -> 0
  | Knoten (links, _, rechts) ->
      anzahl links + anzahl rechts + 1;;
val anzahl : 'a bbaum -> int = <fun>
# anzahl (Knoten
      (Knoten (Blatt, 1, Blatt), 2, Knoten (Blatt, 3, Blatt)));;
- : int = 3
```

```
# let rec unsinn = function
  Knoten (links, _, rechts) -> unsinn links;;
```

Toplevel input:

```
# let rec unsinn = function
  Knoten (links, _, rechts) -> unsinn links;;
```

Warning: this pattern-matching is not exhaustive.

Here is an example of a value that is not matched:

Blatt

```
val unsinn : 'a bbaum -> 'b = <fun>
```

Ungeordnetes Einfügen

```
# let leer = Blatt;;
val leer : 'a bbaum = Blatt

# let einfuegen x s = Knoten (Blatt, x, s);;
val einfuegen : 'a -> 'a bbaum -> 'a bbaum = <fun>

# let rec list2bbaum = function
  [] -> leer
  | x :: l -> einfuegen x (list2bbaum l);;
val list2bbaum : 'a list -> 'a bbaum = <fun>

#let z = list2bbaum [3; 4; 7; 11; 21; 12];;
z : int bbaum =
Knoten (Blatt, 3,
  Knoten (Blatt, 4,
    Knoten (Blatt, 7,
      Knoten (Blatt, 11, Knoten (Blatt, 21, Knoten (Blatt, 12, Blatt))))))

(* der resultierende Baum ist voellig unbalanciert *)
```

Suche in einem Baum

```
# let rec mitglied x = function
  Blatt -> false
  | Knoten (links,y,rechts) ->
    (x = y) || mitglied x links || mitglied x rechts;;
(* x muss sowohl links als auch rechts gesucht werden *)
```

```
val mitglied : 'a -> 'a bbaum -> bool = <fun>
```

```
# mitglied 4 z;;
- : bool = true
# mitglied 5 z;;
- : bool = false
```

Geordnetes Einfügen

```
# let rec einfuegen x = function
  Blatt -> Knoten (Blatt, x, Blatt)
| Knoten (links, y, rechts) ->
  if x < y then
    Knoten (einfuegen x links, y, rechts)
  else if x > y then
    Knoten (links, y, einfuegen x rechts)
  else Knoten (links, y, rechts);;
val einfuegen : 'a -> 'a bbaum -> 'a bbaum = <fun>

# list2bbaum [11; 7; 3; 4; 21; 12];;
- : int bbaum =
Knoten
  (Knoten (Knoten (Blatt, 3, Blatt), 4,
    Knoten (Blatt, 7, Knoten (Blatt, 11, Blatt))),
  12, Knoten (Blatt, 21, Blatt))
```

Suche in einem geordneten Baum

```
# let rec mitglied x = function
  Blatt -> false
  | Knoten (links,y,rechts) ->
    if x < y then mitglied x links
    else if x > y then mitglied x rechts
    else true;;

val mitglied : 'a -> 'a bbaum -> bool = <fun>
# let z = list2bbaum [11; 7; 3; 4; 21; 12];;
val z : int bbaum =
  Knoten
    (Knoten (Knoten (Blatt, 3, Blatt), 4,
      Knoten (Blatt, 7, Knoten (Blatt, 11, Blatt))),
    12, Knoten (Blatt, 21, Blatt))
# mitglied 3 z;;
- : bool = true
# mitglied 5 z;;
- : bool = false
```

Teil XII.6

Objektorientierte Programmierung in OCaml

Klassendeklaration

```
class name  $p_1, \dots, p_n$  =  
  object  
  ...  
  Instanzvariablen  
  ...  
  Methoden  
  ...  
end
```

Deklaration von Instanzvariablen und Methoden

Instanzvariablen:

`val name = expression`

oder

`val mutable name = expression`

`mutable` Variablen können ihren Wert später ändern, andere behalten stets ihren initialen Wert.

Methoden:

`method name p1 ... pn = expression`

Klassendefinition: Beispiel

```
# class point =  
  object  
    val mutable x = 0  
    method get_x = x  
    method move d = x <- x + d  
  end;;  
class point :  
  object val mutable x : int method get_x : int  
  method move : int -> unit end
```

Erzeugen eines Punktes als Instanz der Klasse point:

```
# let p = new point;;  
val p : point = <obj>
```

Aufruf von Methoden

```
# p#get_x;;
```

```
- : int = 0
```

```
# p#move 3;;
```

```
- : unit = ()
```

```
# p#get_x;;
```

```
- : int = 3
```

Instanziierung von Objekten

Klassenrumpf (expressions) wird nur bei der Instanziierung neuer Objekte ausgeführt:

```
# let x0 = ref 0;;

# class point =
  object
    val mutable x = incr x0; !x0
    method get_x = x
    method move d = x <- x + d
  end;;

# new point#get_x;;
- : int = 1

# new point#get_x;;
- : int = 2
```

Aufruf eigener Methoden

Erfordert Binden einer Variablen (hier: `s`) an die eigene Instanz:

```
# class printable_point x_init =  
  object (s)  
    val mutable x = x_init  
    method get_x = x  
    method move d = x <- x + d  
    method print = print_int s#get_x  
  end;;  
  
# let p = new printable_point 7;;  
  
# p#print;;  
7- : unit = ()
```

Initializer

Let-Anweisungen in Klassendefinitionen werden vor der Objektkonstruktion ausgeführt und können nicht überschrieben werden.

```
# class printable_point x_init =
  let origin = (x_init / 10) * 10 in
  object (self)
    val mutable x = origin
    method get_x = x
    method move d = x <- x + d
    method print = print_int self#get_x
    initializer print_string "new point at ";
                  self#print; print_newline()
  end;;

# let p = new printable_point 17;;
new point at 10
val p : printable_point = <obj>
```

Virtuelle Methoden

Deklaration von Methoden, ohne sie zu definieren.

Klassen mit virtuellen Methoden können nicht instanziiert werden.

```
# class virtual abstract_point x_init =
  object (self)
    val mutable x = x_init
    method virtual get_x : int
    method get_offset = self#get_x - x_init
    method virtual move : int -> unit
  end;;
```

```
# class point x_init =
  object
    inherit abstract_point x_init
    method get_x = x
    method move d = x <- x + d
  end;;
```

Private Methoden

Private Methoden sind nicht im Objektinterface sichtbar, können nur von anderen Methoden desselben Objekts aufgerufen werden.

```
# class restricted_point x_init =  
  object (self)  
    val mutable x = x_init  
    method get_x = x  
    method private move d = x <- x + d  
    method bump = self#move 1  
  end;;  
  
# let p = new restricted_point 0;;  
val p : restricted_point = <obj>
```

(Verbotener) Aufruf einer privaten Methode:

```
# p#move 10;;  
This expression has type restricted_point  
It has no method move
```

Aufruf einer sichtbaren Methode, die eine private Methode ruft:

```
# p#bump;;  
- : unit = ()
```

Vererbung:

Private Methoden werden vererbt und können in Subklassen sichtbar gemacht werden:

```
# class point_again x =  
  object (self)  
    inherit restricted_point x  
    method virtual move : _  
  end;;
```

Vererbung

```
# class colored_point x (c : string) =  
  object  
    inherit point x  
    val c = c  
    method color = c  
  end;;
```

```
# let p' = new colored_point 5 "red";;  
val p' : colored_point = <obj>
```

```
# p'#get_x, p'#color;;  
- : int * string = (5, "red")
```

point und colored_point haben inkompatible Typen!

Aber: `get_x` ist generische Funktion, die auf alle Objekte anwendbar ist, für die diese Funktion definiert ist!

```
# let get_succ_x p = p#get_x + 1;;  
val get_succ_x : < get_x : int; .. > -> int = <fun>
```

```
# get_succ_x p + get_succ_x p';;  
- : int = 8
```

Methoden müssen nicht vorher deklariert werden:

```
# let set_x p = p#set_x;;  
val set_x : < set_x : 'a; .. > -> 'a = <fun>
```

```
# let incr p = set_x p (get_succ_x p);;  
val incr : < get_x : int; set_x : int -> 'a; .. > -> 'a = <fun>
```

Mehrfachvererbung

Bezug auf Klasse, von der geerbt wird:

`inherit klasse p_1, \dots, p_n as name`

```
# class printable_colored_point y c =
  object (self)
    val c = c                method color = c
    inherit printable_point y as super
    method print =
      print_string "("; super#print;
      print_string ", "; print_string (self#color);
      print_string ")"
    end;;
# let p' = new printable_colored_point 17 "red";;
new point at (10, red)
val p' : printable_colored_point = <obj>
# p'#print;;
(10, red)- : unit = ()
```