

Objective Caml



Benus Becker
benusb@gmail.com

Universität Freiburg

Sommercampus 2005

Überblick

- ① Grundlagen von Ocaml und der funktionalen Programmierung
- ② Crash Kurs
- ③ OcamlCore – Der funktionale Kern
- ④ Imperative Features
- ⑤ Sonstiges, Ausblicke und Ressourcen

Kapitelübersicht: Grundlagen von Ocaml und der funktionalen Programmierung

① Grundlagen von Ocaml und der funktionalen Programmierung

Ocaml

Ein kurzer Blick auf Ocaml

Der λ -Kalkül

Funktionale Programmierung

Typen

② Crash Kurs

③ OcamlCore – Der funktionale Kern

④ Imperative Features



Ocaml

- „*the programming tool of choice for discriminating hackers*“ [ICFP '99, '00, '02]
- „*language designed for smart people*“, nicht „*language designed for the masses*“ [Mike Vanier, Paul Graham]
- „*Ocaml ist französisch und elitär*“ [Stefan Franck]
- „*ist schön und macht Spass*“ [find ich]

- ermöglicht schnellen Programmentwurf
- sicher Code
- *beweisbarer* Code
- wartbarer Code
- für große Projekte geeignet
- sehr schnelle Programme
- keine Skriptsprache



Features von Ocaml (\approx Ziele dieses Kurses)

- die Sprache ...
 - ist funktional mit imperativen und objektorientierten Konstrukten
 - besitzt ausdrucksstarkes Typsystem
 - ist statisch und stark getypt
 - ist parametrisch polymorph
 - verfügt über Typinferenzsystem
 - hat Ausnahmemechanismus
 - Garbage Collection
 - ...
- dazu gibt es ...
 - viele Libraries
 - interatives Toplevel, Bytecode- und Maschinencode Compiler
 - Lexer, Parser-Generator
 - ...



Geschichte von Ocaml

- 1940 Alonzo Church: λ -Kalkül
- 1958 Paul McCarthy: Lisp
- 1973 Robin Milner veröffentlicht ML
- 1987 Guy Cousineau, INRIA: Caml
- 1991 Xavier Leroy & Damien Doligez: Caml-light
- 1995 dieselben: Objective Caml
- inzwischen Version 3.09



Ocaml als Taschenrechner

- Ocaml Toplevel starten mit Befehl `ocaml`
- „#“ ist der Prompt von Ocaml
- Ausdrücke mit „;;“ abschließen
- Kommentare zwischen „(*“ und „*)“ (können geschachtelt werden)

```
# 39 + 3 ;;  
- : int = 42  
# 3 / 2 + 1 ;;  
- : int = 2
```



Variablenbindung

- Variablenbindung erfolgt mit dem Keyword `let`

```
# let foo = 127 ;;  
val foo : int = 127
```

- Toplevel gibt Signatur der Eingabe aus
`val <gebundene Variable> : <Typ> = <Wert>`
- im Folgenden hat dann `foo` den Wert 127

```
# foo ;;  
- : int = 127
```

in der Ausgabe bezeichnet „-“, dass keine Variable gebunden wurde.



Funktionsdeklaration

- Funktionen werden ebenfalls mit `let` gebunden
- Parameter stehen direkt hinter Funktionsnamen

```
# let square x = x * x ;;  (* berechnet das Quadrat von x *)  
val square : int -> int = <fun>
```

```
# let succ n = n + 1 ;;  (* berechnet Nachfolger von n *)  
val succ : int -> int = <fun>
```

```
# let average a b = (a +. b) /. 2.0  
  (* berechnet das Mittel von a und b *) ;;  
val average : float -> float -> float = <fun>
```



Funktionsaufruf und Konditionalausdrücke

- Funktionsaufrufe

- Funktionsaufruf benötigt weder Klammern noch Kommas

```
# square 5 ;;  
- : int = 25
```

```
# average 3.0 6.0 ;;  
- : float = 4.5
```

- Klammern nur nötig für Präzedenzen

```
# square (succ 3) ;;  
- : int = 16
```

- If-then-else

```
# if 1 < 0 then "eins kleiner null!"  
   else "alles in ordnung.";;  
- : string = "alles in ordnung"
```

Fakultät

- mathematische Definition:

$$f : \mathcal{N} \rightarrow \mathcal{N}$$

$$x \mapsto \begin{cases} 1 & \text{wenn } x = 0 \\ x * f(x - 1) & \text{sonst} \end{cases}$$

- Definition in Ocaml:

```
# let rec fact x =  
  if x = 0 then 1  
  else x * fact (x-1) ;;  
val : fact : int -> int = <fun>  
# fact 5 ;;  
- : int = 120
```



Alonzo Church und der λ -Kalkül

- 1940 von Alonzo Church entwickelt
- Berechenbarkeitsmodell wie Turingmaschinen
- These der allgemeinen Berechenbarkeit entwickelt aus Gleichmächtigkeit von Turingmaschinen und λ -Kalkül
- λ -Kalkül ist für funktionale Sprachen das, was für imperative Sprachen die Turingmaschinen sind (also zum Lernen unnötig, nur theoretisches Fundament)



Syntax des λ -Kalküls

$$t ::= c \mid x \mid (\lambda x.t) \mid (t t)$$

- Konstanten c
- Variablen x
Wert ergibt sich aus der Umgebung
- Abstraktion $(\lambda x.t)$
 - „ungesättigter“ Term (anonyme Funktion)
 - kann als erster Term einer Applikation dienen
- Applikation $(t_1 t_2)$
 - Anwendung von t_1 auf t_2
 - wenn t_1 Wert $\lambda x.t$ hat, evaluiert es zu t , wobei darin alle Vorkommnisse von x durch den Wert von t_2 ersetzt sind



λ in Ocaml und Currying (Schönfinkeln)

- Abstraktion und Applikation Hauptbestandteile funktionaler Programmiersprachen
- Darstellung der λ -Abstraktion in Ocaml: `function x -> e` entspricht $\lambda x.e$
- Darstellung von Funktionen mit mehreren Argumenten (Currying) :
statt: $f(x, y) = t$ (gewohnt mathematisch)
jetzt: $f = \lambda x.\lambda y.t$ (λ -Kalkül)
oder verkürzt: $f = \lambda xy.t$
f ist Funktion, die ein Argument x erwartet, und eine Funktion zurückgibt, die ein weiteres Argument y erwartet, und dann t berechnet.



Beispiel zum λ -Kalkül

- $(\lambda x.f\ x)\ 3 = f\ 3$
- $(\lambda a.g\ a)\ ((\lambda x.x + 1)\ 2) = (\lambda a.g\ a)\ (2 + 1) = (\lambda a.g\ a)\ 3 = g\ 3$
- *Currying*: $(\lambda x.(\lambda y.x + y))\ 3\ 5 = (\lambda y.3 + y)\ 5 = 3 + 5 = 8$
- `true`, `false`, `if`
 - Eigenschaften:
 - `if true x y \rightarrow x`
 - `if true x y \rightarrow y`
 - Realisation:
 - `true = $\lambda x.\lambda y.x$`
 - `false = $\lambda x.\lambda y.y$`
 - `if = $\lambda zxy.z\ x\ y$`



Funktionale Programmierung ...

... lässt etwas von der Eleganz, Klarheit und Präzision der Mathematik in die Welt der Informatik einfließen. [Peter Pepper]

Paradigma der funktionalen Programmierung stellt sehr low-level, aber hardware-abstrakte Mittel zur eleganten Lösung von Problemen zur Verfügung.



Imperative Programmierung

- entstanden im Prozess des Abstrahierens von der Hardware-Ebene (Assembler, FORTRAN, C)
- beruht auf dem Verändern von Werten, die den Zustand des Programms definieren
- Anfangszustand σ wird über Zwischenzustände $\sigma_1 \cdots \sigma_n$ in Endzustand σ' gebracht
- σ enthält das Input und σ' das Output:

$$\sigma = \sigma_0 \rightarrow \sigma_1 \rightarrow \cdots \rightarrow \sigma_n = \sigma'$$

- Programmausführung bedeutet, mit Sequenz von imperativen Statements vom Anfangszustand zum Endzustand zu gelangen



Funktionale Programmiersprachen

- entstanden im Prozess des Abstrahierens von Algorithmen-Spezifikationen
- Programmieren nahe an der Spezifikation
- in der funktionalen Programmierung wird eine Funktion (im mathematischen Sinne) gesucht, die aus dem Anfangszustand den Endzustand berechnet:

$$\text{OUTPUT} = \text{PROGRAMM}(\text{INPUT})$$

- das Programm ist eine Expression
- Ausführung eines funktionalen Programmes heißt, diese Expression auszuwerten



Programmieren ohne Seiteneffekte

in funktionaler Programmierung gibt es ...

- keine Variablen — keinen Zuweisungsoperator
- keinen Zustand
- keine sequenzielle Ausführung

große Mächtigkeit durch ...

- ein sehr ausdrucksstarke funktionale Ausdrücke
- Funktionen als „erste-Klasse-Objekte“
- Funktion nehmen andere Funktionen als Parameter (Funktionen höherer Ordnung)
- Funktionen als Ergebnis von Funktion
- lokale und anonyme Definition von Funktionen (Abstraktion)



Typen

„If it typechecks it works“ [ML Folklore]

- wichtiger Bestandteil funktionaler Programmierung
- Typen ähnlich wie in der Mathematik
- jede Expression (also jeder Wert, jede Funktion) hat genau einen Typ
 - primitive Typen: `float`, `int`, `string`, ...
 - zusammengesetzte Typen: Paare, Funktionen, Listen, ...

3		ganze Zahl
3.141		reelle Zahl
("wentz", 12)		Paar von einem String und einem ganzen Zahl
succ		Funktion, die ganze Zahl auf ganze Zahl abbildet

- Datenstrukturen werden in funktionaler Programmierung mithilfe des mächtigen Typsystems entworfen
- Typvariablen aus dem griechischen Alphabet: α , β , γ , ...



starke, statische Typisierung

- statische Typisierung alle Typen sind zur Compile-Zeit festgelegt
- starke Typisierung
 - jeder Ausdruck in Ocaml hat genau *einen* Typ
 - jedes Programm muss zu Ausführung typkonsistent sein
 - keine impliziten Typkonversionen
- bewirkt klaren und durchdachten Code
- erhöht Sicherheit und Produktivität
- keine dynamischen Typtests, daher schnellere Ausführung



Zusammengesetzte Typen

- `int * int` bezeichnet den Type der Paare natürlicher Zahlen ($\mathbb{N} \times \mathbb{N}$)
- `float -> string` bezeichnet den Typ der Funktionen, die Fließkommazahlen auf String abbilden ($\mathbb{R} \rightarrow \mathbb{S}$)
- `(float * float) -> bool` bezeichnet den Typ der Funktionen, die Paare von reellen Zahlen auf Boolesche Werte abbilden ($\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{B}$)

Syntax: Primitive Typen

```
int | float | char | string | unit
```

Syntax: Typen (Ausschnitt)

```
<primitive-type>  
| <primitive-type> * <primitive-type>  
| <primitive-type> -> <primitive-type>
```



Typinferenz und Polymorphie

- Compiler schlussfolgert alle Typen
- Benutzung von Variablen in Ausdrücken, die bereits getypt sind, engt Typ ein
- Typ einer Variablen ergibt sich aus ihrer Benutzung
- manche Typen können nicht vollständig inferiert werden: Polymorphie
- polymorphe Werte können mit Werten jedes Typ benutzt werden



Ende der Theorie



Kapitelübersicht: Crash Kurs

① Grundlagen von Ocaml und der funktionalen Programmierung

② Crash Kurs

Primitive Typen (Zahlen und Boolesche Werte)

Definitionen

Funktionen

if-then-else

Zwischenstopp und Aufgaben

③ OcamlCore – Der funktionale Kern

④ Imperative Features

⑤ Sonstiges, Ausblicke und Ressourcen



Ziel

- Lernen einer ungefähr Turing-mächtigen Teilsprache von Ocaml
- mit Arithmetik, Zeichenketten, Tupeln und Listen



Ganze Zahlen

- Ganze Zahlen (\mathbb{N}) sind vom Typ `int`:

```
# 123 ;;  
- : int = 123  
# -1 ;;  
- : int = -1
```

- `+`, `-`, `*`, `/` übliche Operatoren auf `int`, `mod` ist Modulooperator

```
# 1 + 2 ;;  
- : int = 3  
# -123 * 2 ;;  
- : int = -246  
# 2 + 4 * 8 ;;  
- : int = 34  
# 13 mod 5 ;;  
- : int = 3
```

- <http://caml.inria.fr/pub/docs/manual->

Fließkommazahlen

- Fließkommazahlen (\mathbb{R}) haben den Typ `float`
- müssen einen Dezimalpunkt oder ein `e` enthalten (sonst ist es `int`)

```
# -123.456 ;;  
- : float = -123.456  
# 31.415926e-1 ;;  
- : float = 3.1415926
```

- Operatoren auf `float` haben einen „.“ als Suffix (sonst rechnen sie auf `int`!)

```
# 0.1 +. 1.3 *. 4.5 ;;  
- : float = 5.59  
# 1.0 /. 0.0 ;;  
- : float = inf
```

- http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html#6_Floatingpointarithmetic



Integers vs. Fließkommazahlen

- `float` und `int` niemals miteinander verrechnen!

```
# 2 + 3.0 ;;
```

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

- Funktionen zur Konversion: `float_of_int` und `int_of_float`

```
# 2 + int_of_float 3.0 ;;
```

```
- : int = 5
```

- weitere sinnvolle Funktionen auf `floats`:
`ceil`, `sqrt`, `log`, `sin`, `atan`, ...



Boolesche Werte

- bezeichnen Wahrheitswerte: `true` und `false`
- Ocamltyp: `bool`
- `&&`(und), `||`(oder) und `not` Operatoren
- Vergleichsoperatoren liefern Boolesche Werte (Operanden vom gleichen Typ!)
- `=` Gleichheit, `<>` Ungleichheit, `<`, `>`, `<=`, `>=`
- Präzedenz: `not` vor `&&` vor `||`
- http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html#6_Booleanoperations



Beispiele Boolesche Werte

```
# true ;;
- : bool = true
# true && false ;;
- : bool = false
# true && true || false ;;
- : bool = true
# 2 = 3 || true ;;
- : bool = true
# 3.0 <> 3 ;;
```

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

```
# 3.0 <> float_of_int 3 ;;
- : bool = true
```



Das globale let-Binding

- let-Bindings binden Bezeichner an Werte
- `let x = e ;;` bindet Wert der Expression `e` an den Bezeichner `x` in allen folgenden Aufrufen

```
# let x = 1 ;;  
val x : int = 1  
# let y = 2 ;;  
val y : int = 2  
# let z = x + y ;;  
val z : int = 3
```

- es ist das gleiche, einen *Wert*, eine *Expression*, die diesen Wert berechnet, oder einen *Bezeichner*, an den dieser Wert gebunden ist, zu benutzen.

Syntax: globales let-Binding

```
let <ident> = <expr> ;;
```

Das lokale let-Binding

```
let x = expr1 in expr2
```

- x hat Wert von expr₁ innerhalb von expr₂

```
# let x = exp 0. in x *. x ;;  
- : float = 1.  
# x ;;  
Unbound value x
```

- schachtelbar

```
# let erg = (let zw_erg = 1 + 1 in zw_erg * zw_erg) ;;  
val erg : int = 4
```

- ist auch eine Expression, mit Typ

Syntax: lokales let-Binding

```
let <ident> = <expr> in <expr>
```

Funktionsapplikation

- *Applikation*: schon erwähnt ...
 - Funktionsargumente werden direkt hinter die Funktion geschrieben
 - ohne Kommas, Klammern nur für Präzedenz
 - Applikation kann *teilweise* sein
- bindet am stärksten (vor Operatoren, etc.)
- links-assoziativ

Syntax: Applikation

`<expr>` _{$\alpha_1 \rightarrow \dots \rightarrow \alpha_n$} `<expr>` _{$\alpha_1$} ... `<expr>` _{$\alpha_{n-k-1}$}



Funktionsdeklaration (1)

- bisherige let-Bindings binden konstante Werte
- Funktionen sind „ungesättigte“ Werte, hängen von anderen Werten ab
- `let func arg1 ... argn = e` bindet eine Funktion `func`
- `arg1 ... argn` sind Parameter für `func`
- in `e` können `arg1 ... argn` vorkommen, Typen der Argument werden durch `e` bestimmt

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

- `'a -> 'b` bezeichnet Typ der Funktionen von `'a` nach `'b`
- `'a -> 'b -> 'c` Type der Funktionen von `'a` nach `'b -> 'c`



Funktionsdeklaration (2)

- rekursive Funktionen müssen mit `let rec` definiert werden

```
# let rec for_ever x = for_ever x ;;  
val for_ever : 'a -> 'b = <fun>
```

- Higher Order Functions: Funktionen als Parameter

```
# let apply f x = f x ;;  
val apply : ('a -> 'b) -> 'a -> 'b = <fun>
```

- können auch lokal definiert werden

```
# let double n = 2 * n in double 3 ;;  
- : int = 6
```



Funktionen Extra

- Operatoren als Funktion, wenn sie zwischen Klammern geschrieben werden

```
# (+.) ;;  
- : float -> float -> float
```

- man beachte die Möglichkeit der partiellen Anwendung von Funktionen

Syntax: Funktionsdeklaration

```
let [rec]? <ident> <ident> ... <ident> = <expr>
```



Konditionale Kontrollstrukturen (if-then-else)

`if` `cond` `then` `expr1` `else` `expr2`

- ist Expression, evaluiert bei Programmausführung zu *einem* Wert
- `cond` muss eine Expression vom Typ `bool` sein
- Wert von `if-then-else` ist der von `expr1`, falls `cond` zu `true` evaluiert, der von `expr2` sonst
- `expr1` und `expr2` müssen gleichen Typ haben
- `else`-Teil notwendig

Syntax: if-then-else

```
if <expr>bool then <expr>α else <expr>α
```



Soweit ... sogut ...

Bisher eine ziemlich normale Programmiersprache ... Zeit für Übungen ...

- ① schreibe eine Funktion `twist`, die für eine Funktionsaufruf mit zwei Argumenten die Reihenfolge dieser vertauscht.
- ② schreibe eine Funktion `compose` die zwei Funktionen `f` und `g` und einen Wert `x` erwartet, und den Wert der Komposition von `f` und `g` an der Stelle `x` berechnet.
- ③ schreibe eine Funktion `iter` die für `iter f n` die Funktion f^n berechnet.
Schreibe dazu erst Funktionen für f^1 , f^2 und f^3 .
- ④ schreibe eine rekursive Funktion `val power : float -> int -> float` die für `power x n` den Wert von x^n ausrechnet.

Mit „`#use "file";;`“ wird eine Datei "file" ins Toplevel geladen, als wäre sie eingetippt worden.



Kapitelübersicht: OcamlCore – Der funktionale Kern

① Grundlagen von Ocaml und der funktionalen Programmierung

② Crash Kurs

③ OcamlCore – Der funktionale Kern

Primitive Typen (Rest)

Anonyme Funktionen

Einfaches Patternmatching

Tupel und Listen

Deklaration zusammengesetzter Typen und Typeconstraints

Vereinigungen und Records

④ Imperative Features



Ziel

Erkennen der funktionalen Mächtigkeit von Ocaml



Buchstaben und Zeichenketten

- Buchstaben (ASCII) zwischen Singlequotes: 'A', '0', '@', ...
- haben Typ `char`
- sinnvolle Funktionen: `char_of_int` und `int_of_char`
- Zeichenketten zwischen Doublequotes: "der string ...", "4apfel+2eier", ...
- gehören zu Typ `string`
- Zeichenketten mit `^` verknüpfen

```
# char_of_int 111 ;;
- : char = 'o'
# "hello" ^ "caml";;
- : string = "hellocaml"
```

- http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html#6_Stringconversionfunctions



Vergleichsoperatoren

Vergleichsoperatoren operieren auf gleichen Typen, liefern **bool**

- **=**, **<>** strukturelle Gleichheit, Ungleichheit („das gleiche“)
- **==**, **!=** physikalische Gleichheit, Ungleichheit („das selbe“)
- bei **bools** **ints** und **chars** hat **=** und **==** gleiche Semantik
- **<**, **<=**, **>=**, **>** Größenvergleich

```
# 1234 > -1234 ;;
- : bool = true
# let x = "2001ab" ;;
val x : string = "2001ab"
# x = "2001ab" ;; (* strukturelle Gleichheit *)
- : bool = true
# x == "2001ab" ;; (* physikalische Gleichheit *)
- : bool = false
```

- **max** und **min** geben das größere/kleiner von zwei Argumenten zurück



Unit

- Typ `unit` besitzt nur den Wert „void“: `()`
- entspricht Typ `void` aus C/C++
- Rückgabetyt von Prozeduren (Funktionen nur mit Seiteneffekt)
- nur in imperativer Programmierung relevant (später ...)



Anonyme Funktionen

- entspricht Abstraktion im λ -Kalkül
- Wert von `function x -> e` ist eine Funktion

```
# function x -> x mod 2 = 0 ;;
- : int -> bool = <fun>
```

- wird durch Applikation auf Wert ausgewertet

```
# (function x -> x mod 2 = 0) 4 ;; (*  $\equiv 4 \bmod 2 = 0$  *)
- : bool = true
```

- Werte wie alle anderen auch: Parameter für Funktionen, ...
- als Wert von let-Bindings

```
# let even = function x -> x mod 2 = 0 ;;
val even : int -> bool = <fun>
```

Syntax: Abstraktion

```
function <ident> -> <expr>
```

Funktionen in Aktion

```

# let derive f x e = (f (x +. e) -. f x) /. e ;;
val derive:(float->float) -> float -> float -> float=<fun>
# derive sin 0.0 1e-10 ;;
- : float = 2.0
# derive (function x -> x *. x) 1.0 1e-10 ;;
- : float = 2.000000165480742
# let sin' = function x -> derive sin x 0.1 ;;
val sin' : float -> float = <fun>
# let poly n = function x -> power x n ;;
val poly : int -> float -> float = <fun>
# let square = poly 2 ;; (* teilweise Funktionsanwendung *)
val square : float -> float = <fun>
# let cube = poly 3 ;;
val cube : float -> float = <fun>
# (square 3., cube 3.) ;; (* ein Paar! später. *)
- : float * float = (9., 27.)

```

Patternmatching

```
match expr0 with
```

```
| p1 -> expr1
```

```
⋮
```

```
| pn -> exprn
```

- „switch auf Anabolika“, komplexes if-then-else
- elegant Daten differenziert betrachten und zugleich benennen (Dekonstruktion)
- jeder möglichen Form (p_i) des Wertes ($expr_0$) wird eine Expression ($expr_i$) zur Auswertung zugeordnet
- $expr_0$ wird nacheinander mit $p_1 \dots p_n$ abgeglichen
- passt p_i , wird entsprechender $expr_i$ ausgewertet
- Patterns gleicher Typ / Expressions gleicher Typ!
- als Pattern dienen Konstanten, Variablen und „_“



Patternmatching Umgang

- Variablen und „_“ passen immer (Wildcards)
- erstes „|“ optional
- Patternmatchings müssen *alle* Fälle abdecken
- Warnungen des Compilers ernst nehmen!

Syntax: Pattern (Ausschnitt)

```
<constant> | <identifier> | _
```

Syntax: match

```
match <expr>α with
| <pattern>α -> <expr>β
  ⋮
| <pattern>α -> <expr>β
```

Patternmatching Beispiel

```
# let zaehlen n = match n with
  0 -> "nix"
  | 1 -> "eins"
  | n -> "viel: " ^ string_of_int n
val zaehlen : int -> string = <fun>
# zaehlen 3 ;;
- : string = "viel: 3"
```

eine Expression kann mehreren Patterns zugeordnet werden:

```
let digit n = match n with
  0 | 1 | 2 | 3 | 4
  | 5 | 6 | 7 | 8 | 0 -> true
  | _ -> false ;;
val digit : int -> bool = <fun>
```



Patterns sind überall

- Patterns können immer dann auftreten, wenn Variablen gebunden werden
- in `function` (relevant)
statt

```
# let string_of_bool1 = function x ->  
  match x with  
    true -> "true"  
  | false -> "false";;  
val string_of_bool1 : bool -> string = <fun>
```

jetzt

```
# let string_of_bool2 =  
  function true -> "true"  
  | false -> "false";;  
val string_of_bool2 : bool -> string = <fun>
```

Tupel

- einfachster zusammengesetzter Typ
- entsprechen Tupeln aus der Mathematik
- repräsentieren: Koordinaten, Adressen, etc.
- (geklammerte) Sequenz von Werten getrennt durch Kommas
- ein Paare aus `string` und `int`:

```
# let p = ("wentz", 12) ;;  
val p : string * int = ("wentz", 12)
```



Zugriff auf Tupel

- Patterns (Dekonstruktion) haben gleiche Syntax wie Konstruktion
- Dekonstruktion von Tupeln durch Pattern-Matching mit `match ... with`

```
# match p with
  ("wentz", n) -> "wentzinger"
  | _ -> "woanders";;
- : string = "wentzinger"
```

- Dekonstruktion durch Pattern-Matching in let-Bindings

```
# let (strasse, nummer) = p ;;
val strasse : string = "wentz"
val nummer : int = 12
```

- `fst` und `snd` liefern das erste/zweite Element eines *Paares*



Listen

- massiv genutzte Datenstruktur
- enthält Sequenz von Werten gleichen Typs
- Konstruktoren:
 - []: leere Liste
 - e :: l: hängt e vorne an Liste l ran

```
# "hallo" :: "liste" :: [] ;;  
- : string list = ["hallo" ; "liste"]
```

- Kurzschreibweise: [e₁ ; ... ; e_n]
- Typ einer Liste mit Elementen vom Typ t:
t list



Gebrauch auf Listen

- Zugriff durch Pattern-Matching nur auf die ersten Elemente

```
# let rec sum lst = (* summiert Elemente einer Liste *)
  match lst with
    [] -> 0
    | h :: t -> h + sum t ;;
val sum : int list -> int = <fun>
```

```
# let rec length lst = (* berechnet Länge einer Liste *)
  match lst with
    [] -> 0
    | _ :: rest -> 1 + length rest ;;
val length : 'a list -> int = <fun>
# length [("hans",3); ("peter",1); ("marie",2)] ;;
- : int = 3
```



Typdeklarationen

- zusammengesetzte Typen werden mit `type` deklariert

```
# type koordinate = float * float ;;
type koordinate = float * float
```

- können durch weiteren Typ parametrisiert sein
- parametrisierender Typ durch Typvariable ('<ident>) vor neuen Typnamen (mehrere in Tupelschreibweise)

```
# type 'a paar = 'a * 'a ;;
type 'a paar = 'a * 'a
```

Syntax: Typdeklaration (Ausschnitt)

```
type [( 'a1, ... , 'an )]?<ident> = <type>
```



Constraints

- Möglichkeit Typen explizit zu bestimmen (genauer)
- *kein Casting*: Typen können nur „gezwungen werden“, weniger allgemein zu sein
- selten nötig (z.B. polymorphe Funktionen schneller machen)
- `:` ist der „Typconstraintoperator“
- Variable `x` bekommt mit `x:t` bei der Deklaration ein Typconstraint zu Typ `t` Syntax wie in der Signatur

```
# let a = (3.0,-2.1) ;;
val a : float * float = (3.,-2.1)
# let b : koordinate = (3.0,-2.1) ;;
val b : koordinate = (3.,-2.1)
# let apply_int_to_string f (x:int) : string = f x ;;
val apply_int_to_string : (int -> string) -> int ->
string
```

Deklaration von Vereinigungen

- allgegenwärtig in Ocaml-Code
- ermöglicht (benannte) Vereinigung verschiedener Typen
- Werte nehmen eine der Möglichkeiten an
- die „Varianten“ des Typs werden deklariert mit `<konstruktor> [of <type>]`
- Konstruktoren beginnen mit Großbuchstaben
- Deklaration mit Schlüsselwort `type` :

```
# type zahl =  
    Fliesskomma of float  
    | Ganze of int ;;  
type zahl = Fliesskomma of float | Ganze of int
```



Programmieren mit Vereinigungen

Dekonstruktion von Vereinigungen mit Pattern-Matching:

```
# let addiere a b =
  match (a, b) with
    (Ganze m, Ganze n) -> Ganze (m + n)
  | (Ganze n, Fliesskomma f) ->
    Fliesskomma (float_of_int n +. f)
  | (Fliesskomma f, Ganze n) ->
    Fliesskomma (f +. float_of_int n)
  | (Fliesskomma f, Fliesskomma g) ->
    Fliesskomma (f +. g) ;;

val addiere : zahl -> zahl -> zahl = <fun>
# addiere (Ganze 3) (Fliesskomma 2.3) ;;
- : zahl = Fliesskomma 5.3
```



Aufzählungen

- Konstruktoren ganz ohne Argumente (Aufzählungen)

```
# type farbe1 = Rot | Gruen | Blau ;;  
type farbe1 = Rot | Gruen | Blau
```

- oder gemischt

```
# type farbe2 =  
    Rot | Gruen | Blau | RGB of int * int * int  
type farbe2 = Rot | Gruen | Blau | RGB of int * int *  
int
```



Parametrisierte Vereinigungen

- Typen in Vereinigungen können polymorph sein (generische Strukturen)
- Typvariable zwischen `type` und Vereinigungsname

```
# type 'a option = None | Some of 'a ;;  
type 'a option = None | Some 'a  
# let test_ergebnisse = [None; Some 4; Some 8; None] ;;  
val test_ergebnisse : int list = [None; Some 4; Some 8;  
None]
```

'a option ist in der Standardbibliothek



Deklaration von Records

- „benannte Tupel ohne Reihenfolge“ (gleiche physikalische Repräsentation)
- besitzen verschiedene Felder, in denen benannte Daten eines bestimmten Typs gespeichert sind
- ähneln Structs in C
- Deklaration eines Feldes durch `<name> : <typ>`
- Deklaration eines Records durch von Semikolons abgetrennte Deklarationen von Feldern zwischen „{“ und „}“

```
# type db_entry = { name : string ; number : int } ;  
type db_entry = { name : string ; number : int }
```



Umgang mit Records

- Kreation eines Records

```
# let entry31 = {name = "Robin Milner"; number = 1984};;  
val entry31 : db_entry = {name = "Robin Milner"; number  
= 1984}
```

- Zugriff auf die Felder durch Punk-Notation

```
# entry31.name ;;  
- : string = "Robin Milner"  
# entry31.number ;;  
- : int = 1984
```



Kapitelübersicht: Imperative Features

- ① Grundlagen von Ocaml und der funktionalen Programmierung
- ② Crash Kurs
- ③ OcamlCore – Der funktionale Kern
- ④ Imperative Features
 - Output
 - Imperative Kontrollstrukturen
 - Veränderbare Felder
 - Referenzen
 - Exceptions
- ⑤ Sonstiges, Ausblicke und Ressourcen



Output

- für jeden primitiven Datentyp Print-Funktion:
`print_string`, `print_int`, `print_endline`, ...
- Drucken ist Seiteneffekt
- Print-Funktionen geben „()“ zurück

```
# print_endline "hellocaml";;  
hellocaml  
- : unit = ()  
# print_float 2.714 ;;  
2.714- : unit = ()
```



Printf

- `open Printf ;;` (siehe Module) ermöglicht Benutzung des C-artigen `printf` Befehl
- damit der Compiler typchecken kann muss ihm der Formatierungsstring `plain` gegeben sein

```
# printf "ein string (%s), eine ganze zahl (%d) und  
eine reelle (%f)\n" "gnirts" 23 0.123 ;;  
ein string (gnirts), eine ganze zahl (23) und eine  
reelle (0.123000)  
- : unit = ()
```



Möglicher Vergleich C – Ocaml

C

```
int fak (n) {
    int counter = n ;
    int result = 1 ;
    while (counter > 0) {
        result =
            counter * result ;
        counter -- ;
    }
    return result ;
}
```

Ocaml

```
let fak (n) = begin
    let counter = ref n
    and result = ref 1 in
    while !counter > 0 do
        result :=
            !counter * !result ;
        decr counter
    done ;
    !result
end
```



Sequenzen

- Sequenz von Expressions wird durch Trennung durch „;“ zu einer Expression
- Sequenz evaluiert zum Wert der letzten Expression
- alle Expressions außer der letzten sollten Wert „()“ haben (Funktionen mit Seiteneffekte – sonst sinnlos!)
- ignore verwirft jeden Wert (`'a -> unit`)

```
# print_endline "gagah!" ;
  ignore (generiere_frage_nach_der_antwort ()) ;
  "uhuhhh!";;
gagah!
- : string = "uhuhhh!"
```

- Gedanke: „;“ ist rechtsassoziativer Operator
(`function x -> function y -> y`) (** x wird verworfen! **)



Schleifen

- Schleifen sinnvoll für Berechnung mittels Seiteneffekt
- Expression mit Wert „()“
- Rumpf von Schleifen zwischen `do` und `done`
- `while` `bedbool do expr(unit) done`
expr werden ausgeführt, bis Bedingung zu `false` evaluiert
- `for` `v = e1,int to e2,int do e3,(unit) done`
iteriert über das Intervall $[e_1, e_2] \subset \mathbb{N}$ aufzählend
- `for` `v = e1,int downto e2,int do e3,(unit) done`
iteriert über das Intervall $[e_1, e_2] \subset \mathbb{N}$ abzählend



Veränderbare Felder in Records

- Felder in Records können mit Schlüsselwort `mutable` als veränderbar deklariert werden
- in veränderbare Felder kann mit `<-` ein anderer Wert gespeichert werden
- Zuweisung hat Wert „()“

```
# type counter = {mutable index:int} ;;
type counter = { mutable index : int; }
# let i = {index = 0} ;;
val i : counter = {index=0}
# i.index <- 333 ;;  (* Verändern des Wertes *)
- : unit = ()
# i ;;
val i : counter = {index=333}
```



ref

- gewöhnliche Variablen sind nicht veränderbar, aber `'a ref`:
- `type 'a ref = {mutable contents = 'a}` (Referenz) ist vordefiniert
- Erzeugen von einer Referenz mit Wert `x` durch `ref x`
- Verändern von Referenzen durch `eine_ref := neuer_wert`
- Dereferenzierung durch `!eine_ref`

```
# let my_name = ref "Hubert";;
val my_name : string ref = { contents = "Hubert" }
# my_name := "Carlos";;
- : unit = ()
# !my_name ;;
- : string "Carlos"
# my_name := "Camillo" ; my_name ;;
- : string ref = {contents = "Camillo"}
```

Exception

- treten auf, z.B. wenn Definitionsbereich nicht mit Typ eines Parameters übereinstimmen
- unterbrechen Programmausführung
- müssen vor Gebrauch deklariert werden

```
# exception E ;;  
exception E
```

- können ausgelöst werden (mit `raise`)
- `raise X` hat keine Typ
- können gefangen werden mit
(mit `try expr with <pattern-matching>`)

```
# try  
  if Random.bool () then "glueckgehabt"  
  else raise E  
with E -> "geradenoch";;  
- : string = "geradenoch"
```

Exceptions mit Parametern

- Exceptions können parametrisiert werden
- in Parametern können Details über den Fehler erklärt werden

```
# exception Kein_boolean of string ;;
exception Kein_boolean of string
# let bool_of_string = function
  "true" -> true
  | "false" -> false
  | s -> raise (Kein_boolean s) ;;
val bool_of_string : string -> bool = <fun>
# bool_of_string "false" ;;
- : bool = false
# bool_of_string "vielleicht" ;;
Exception: Kein_boolean "vielleicht"
```



Syntax von Exceptions

- `<exc-name>` ist ein Bezeichner beginnend mit Großbuchstaben

Syntax: Exceptiondeklaration

```
exception <exc-name> [of <type>]?
```

Syntax: Exception-Patternmatching

```
[ | <exc-name> [( <exc-par>1 ...<exc-par>n) ]? -> <expr> ]+
```

Syntax: Exceptionhandling

```
try <expr> with <exc-pattern-matching>
```



Kapitelübersicht: Sonstiges, Ausblicke und Ressourcen

- 1 Grundlagen von Ocaml und der funktionalen Programmierung
- 2 Crash Kurs
- 3 OcamlCore – Der funktionale Kern
- 4 Imperative Features
- 5 Sonstiges, Ausblicke und Ressourcen
 - Module
 - Batch-Compiler und Make
 - Techniken
 - Objekte
 - Ressourcen



Module benutzen

- Module auf ersten Blick Sammlungen von Werten (Funktionen, etc.)
- ähnlich wie Packages in Java
- Namen fangen mit Großbuchstaben an
- auf Funktionen/Werte aus Modulen kann ohne öffnen durch Punktsyntax zugegriffen werde

```
# List.rev [1;2;3;4] ;;  
- : int list = [4;3;2;1]
```

- „Namensraum“ wird geöffnet mit `open`
- Programmierbibliotheken bestehen aus Module `List`, `Str`, `Unix`, ... (`Pervasives` grundlegend / autom. geladen)
<http://caml.inria.fr/pub/docs/manual-ocaml/libref/>
dokumentiert die Standardlib
- können mit Tool `ocamlbrowser` schnell gebrowst werden



Module deklarieren

```
# module RealMetric = struct
  type t = float (* Konvention: wichtigster Typ heißt t *)
  let abs_of_float x = if x > 0. then x else -.x
  let d x y = abs_of_float (x -. y)
end ;;

module RealMetric : sig
  type t = float
  val abs_of_float : float -> float
  val d : float -> float -> float
end

# let x = 2.3 and y = -1.6 in RealMetric.d x y ;;
- : float = 3.9
```

Syntax: Moduldeklaration (Ausschnitt)

```
module <module-name> = struct <definitions> end
```

Signaturen von Modulen

- „Typen“ von Modulen
(siehe Ausgabe des Toplevels nach Moduleingabe)
- Konvention: komplett groß geschrieben

```
# module type METRIC_ROOM = sig
  type t  (* abstrakter Typ – keine Angabe über den Inhalt *)
  val d : t -> t -> float
end ;;
module type METRIC_ROOM =
  sig type t val d : t -> t -> float end
```

- abstrakte Typen können nur mithilfe von Operationen der Schnittstelle verarbeitet werden
- verschiedene Module können eine Signatur implementieren

Syntax: Signaturdeklaration

```
module type <signatur-name> = sig <declarations> end
```

Module als Implementation von Signaturen

- Signaturen von Modulen können explizit definiert werden
- Signaturen schränken Eigenschaften von Modulen auf die Eigenschaften der Signatur ein

```
# module RealMetric : METRIC_ROOM = struct
  type t = float
  let abs_of_float x = if x > 0. then x else -.x
  let d x y = abs_of_float (x -. y)
end ;;
module RealMetric : METRIC_ROOM
# RealMetric.abs_of_float ;;
Unbound value RealMetric.abs_of_float
```

- Sinn: Verstecken von Informationen/Implementationsdetails etc.

Syntax: Module signieren

```
module <module-name> : <signatur-name> =
  struct <definitions> end
```

Funktoren

- Technik zum Code Recycling in funktionalen Programmiersprachen
- Funktoren sind Module die Module mit einer bestimmten Signatur als „Argument“ nehmen
- innerhalb des Funktors kann auf die Eigenschaften des Argument-Moduls zugegriffen werden
- ziemlich advanced ...

Syntax: Funktoren

```
module <mod-name>  
  (<mod-name>:<sign-name>) ... (<mod-name>:<sign-name>) =  
  struct <declarations> end
```



Kompilieren einzelner Dateien

- `.ml` Dateien enthalten Ocaml Implementationen
- kein „main“-Funktion – der gesamte Code wird ausgewertet
- keine `;;` mehr nötig
- `ocamlc/ocamlopt` wandelt Ocaml-Code in maschinenunabhängigen Bytecode um
- mit `-I dir` wird der Ordner `dir` in den Suchpfad aufgenommen
- mit `-I +dir` wird der Ordner `dir` relativ zum Installationsverzeichnis von Ocaml aufgenommen
- `-o filename` bestimmt man den Dateinamen der Ergebnisse (standard: `a.out`)
- `.cma/.cmxa` Dateien werden als Libraries verstanden



Kompilieren ohne Linken

- Kompiler erzeugt standardmäßig ausführbare Dateien (gelinkt)
- auf gelinkten Code kann aus anderen Dateien nicht zugegriffen werden
- Flag `-c` verhindert linking
- `ocamlc -c some.ml` erzeugt Datei `some.cmo`, die mit `# load "some.cmo"` ins Toplevel geladen werden kann
- gleicher Effekt, als wäre im Toplevel eingegeben worden

```
# module Some =  
    <code-aus=datei-some.ml>  
end ;;
```



Code in mehreren Dateien

- Code eines Programms kann auf mehrere Dateien verteilt sein
- `.ml`'s sind „Compilation Units“, können unabhängig voneinander kompiliert werden
- Dateien werden implizit als Module verstanden (Name des Moduls ist Dateiname, mit großem Anfangsbuchstaben)
- Zugriff auf Code in anderen Dateien über Punktschreibweise



Kompilieren mehrerer Dateien

- Dateien auf deren Code zugegriffen werden soll, müssen bereits kompiliert sein
- für verschränkt rekursive Dateien müssen Signaturen angegeben und zuerst kompiliert werden
- .mli Dateien enthalten Ocaml Signaturen
- findet der Kompiler in einem Verzeichnis <name>.ml und <name>.mli, so nimmt er <name>.ml als Implementation an
- wird dann name.cmo ins Toplevel hat das die Wirkung wie:

```
# module type NAME =  
  sig <inhalt-von-name.mli> end  
module Name : NAME =  
  struct <inhalt-von-name.ml> end
```

- Faustregel: erst .mli und dann .ml Dateien kompilieren (nicht linken). Anschliessend die .cmo Dateien linken.



Make

- enthält Abhängigkeiten der Quelldateien (\Rightarrow `ocamldep`)
- enthält Befehle, mit denen Dateien generiert werden
- Beispiel, wenn `datei1` aus `datei2` und `datei3` durch Befehle `befehl1`, `befehl2` `befehl3` erzeugt werden soll.

```
<datei1> : <datei2> <datei3>  
    <befehl1>  
    <befehl2>  
    <befehl3>
```

- für `datei2` und `datei3` werden die Abhängigkeiten rekursiv aufgelöst und entsprechend Befehle ausgeführt
- es gibt auch Patterns, um alle Dateien einer Sorte (Endung) zu erzeugen



OCamlMakefile

- sehr allgemeines Makefile für Ocaml
- erzeugt Abhängigkeiten und Befehle selbst

Makefile mit OCamlMakefile Beispiel

```
OCAMLMAKEFILE := <pfad-von-OCamlMakefile>
SOURCES := <ocaml-code-dateien-in-richtiger-reihenfolge>
INCDIRS := <verzeichnisse-von-libraries>
LIBS := <namen-der-benutzten-libraries>
RESULT := <name-ohne-endung>
include $(OCAMLMAKEFILE)
```

- Aufruf mit `make` aus der shell
- `make nc` erzeugt das Programm in native-code, `make bc` in byte-code



Closures

- bezeichnen Funktionen in denen Variablen von außerhalb der Funktion (*freie Variablen*) benutzt werden

```
# let x = ref 0 ;;  
val x : int ref = {content = 0}  
# let f y = x + y ;;  
val f : int -> int = <fun>
```

- Wert eines Funktionsaufrufs abhängig von Variablen der lexikalischen Umgebung bei Deklaration der Funktion
- Funktion kann nicht aus dem Kontext gerissen werden
- bei Funktionsaufruf muss die lexikalische Umgebung der Funktionsdeklaration klar sein
- Closures umgesetzt als Paar von Funktion und Pointer auf die lexikalische Umgebung



Typische Closure

```
# let next, reset =
  let index = ref 0 in
    (function () -> incr index ; !index)
    (function () -> index := 0) ;;
val next : unit -> int = <fun>
val reset : unit -> unit = <fun>
# for i=1 to 5 do print_int (next ()) done ;
  reset () ; next () ;;
12345- : int = 1
```

- hier Sinn und Ergebnis: die `index` Variable bleibt vor dem Benutzer „versteckt“
- nur spannend mit Seiteneffekten



Continuation Passing

- komplette Programme lassen sich in diesem Stil schreiben
- jede Funktion bekommt eine Funktion als Argument, die am Ende ihrer Ausführung aufgerufen wird (Continuation)
- Continuation enthält den Code, in dem die ursprüngliche Funktion eigentlich aufgerufen werden sollte.
- eine Funktion kehrt nicht zurück
- Continuation/Kontext der Funktion wird statt eines nicht-endrekursiven Aufrufs im nächsten Aufruf verändert
- higher-order Funktionen nötig
- ermöglicht es, jede rekursive Funktion endrekursiv zu machen

```
let rec fact n k = match n with
  0 | 1 -> k 1
  | n -> fact (pred n) (function x -> k (x*n))
in fact 4 (fun n -> printf "<%d>" n) ;;
<24> -: unit = ()
```

Eine Klasse

Hickey, Kapitel 12

```
# class point (x_init, y_init) = object
  val mutable x = x_init
  val mutable y = y_init
  method move dx dy =
    x <- x + dx ;
    y <- y + dy
  method to_string =
    "(" ^ string_of_int x ^ ", " ^ string_of_int y ^ ")"
end ;;

class point : int * int -> object ... end
# let p = new point (3,2) ;;
val p : point = <obj>
# p # move 3 8 ; p # to_string ;;
- : string = "(6, 10)"
```

Features des OO-Systems

- Mehrfachvererbung
- explizite Namesangabe der eigenen Instanz („this“) und der Mutterklassen („super“)
- Initialisierungscode („Konstruktor“)
- virtuelle Klassen/Methoden
- durch Typvariablen parametrisierte (generische) Klassen
- *aber: Vererbung ist nicht Subtyping*
- u.v.a.m.



ein paar Ressourcen – 1

O'Reilly Buch ist das Standardwerk um Ocaml zu lernen. Sehr ausführlich und teilweise nicht ganz trivial.

[<http://caml.inria.fr/pub/docs/oreilly-book/index.html>]

Introduction to the Objective Caml Programming Language angenehmes Buch zum Lernen und teilweise Grundlage und gute Erweiterung dieser Slides [www.cs.caltech.edu/courses/cs134/cs134b/book.pdf]

Mailing Listen ocaml_beginners@yahoo.com ist für Anfänger und caml-list@inria.fr für Gurus. Archive

[http://groups.yahoo.com/group/ocaml_beginners/] und

[<http://caml.inria.fr/pub/ml-archives/caml-list/>]

Ocaml Hump ist ein Depot von Ocaml Software und Libraries von und für Ocaml [<http://caml.inria.fr/cgi-bin/hump.en.cgi>]



ein paar Ressourcen – 2

die Dokumentation der Standardlibrary ist ein unverzichtbares Hilfsmittel:

<http://caml.inria.fr/pub/docs/manual-ocaml/libref/index.html>

ein Styleguide kann Syntaxstreits verhindern

[http://caml.inria.fr/pub/old_caml_site/FAQ/pgl-eng.html]

eine Sidebar für Mozilla/Firefox vereinigt die meisten wichtigen

Ressourcen [<http://caml.inria.fr/resources/mozilla-sidebar.en.html>]

