Sprachelemente von OCaml

für: Konzepte von Programmiersprachen, WS 03/04

Holger Gast
gast@informatik.uni-tuebingen.de

Dieses Handout ist eine Ansammlung der Anteile von OCaml, die wir für die Vorlesung brauchen werden. Es dient vor allem der Erinnerung dessen, was in der ersten Übungsstunde gezeigt wird, kann aber auch gern als Beispielvorlage für die Programme des ersten Blattes benutzt werden. Eine ausführliche Einleitung ist auf der Übungsseite unter *Material* zugänglich.

Die Beispiele in diesem Handout sind nicht schmückendes Beiwerk, sondern der Hauptteil: Es ist im Zweifel wichtiger, die Beispiele zu verstehen als den Text dazwischen. Als Test für das Verständnis kann man sich fragen, ob man die Stellen in den Beispiel findet, auf die sich Bemerkungen im Text beziehen. Umgekehrt kommen die Beispiele in abgeänderter Form auch wirklich vor – es lohnt sich also, sie gleich jetzt zu verstehen, dann sind die Aufgaben sehr viel einfacher.

Hinweis: Die mit **Leitlinie:** gekennzeichnten Abschnitte sollten auf jeden Fall gelesen und beherzigt werden; **Achtung:** gibt Hinweise auf Fehlermöglichkeiten, schließlich enthält ein **Hinweis:** allgemeine Hinweise auf wichtige Punkte.

Zu den Programmen: Die meisten Beispielprogramme werden mit `nuweb` direkt aus diesem Handout extrahiert. Im Text bekommen sie eine Überschrift und das Ende-Zeichen ⋄. Beide gehören also nicht zu OCaml, sondern zur Darstellung.

1 Funktionen und ihre Aufrufe

In OCaml ist *die* Grundlage der gesamten Programmierung der *Ausdruck*. Ein Ausdruck ist ein Stückchen Programm, das einen Wert berechnet – und Werte berechnen ist doch letztlich das, was wir wollen.

Die kleinsten Elemente eines OCaml-Programms sind die *einfachen Ausdrücke*. Sie sind dadurch gekennzeichnet, daß sie schon beim ersten Hinschauen als eine Einheit gelesen werden: Ein einfacher Ausdruck s ist ein Bezeichner (`x`, `i`, `find_var`), eine Konstante (`5`, `'a'`, `"Hallo, Welt"`) oder ein geklammerter Ausdruck (e). Grundsätzlich ist jeder einfache Ausdruck auch ein Ausdruck, und die Klammerung kann beliebig fortgesetzt werden: $(((e)))$ bedeutet dasselbe wie (e). Die allgemeinen *Ausdrücke* können auch komplizierter sein: Zum Beispiel kann man

$$e_1 + e_2 \qquad e_1 * e_2 \qquad e_1 \hat{} e_2 \ldots$$

für Berechnungen mit ganzen Zahlen schreiben. Außerdem gibt es Funktionen, die in der bekannten Notation $x \mapsto f(x)$ geschrieben werden.

$$\texttt{fun}\ x\ \texttt{->}\ 2 * x$$

Funktionen machen nur dann Sinn, wenn sie auch angewendet (oder aufgerufen) werden können. Wenn zum Beispiel f eine Funktion ist, dann kann man sie (analog zu sin x) so verwenden:

$$f\ 4$$

Allgemeiner kann eine Funktion auch mehr als ein Argument haben

$$\texttt{fun}\ x_1 \ldots x_m\ \texttt{->}\ e$$

und man kann sie auf n Argumente anwenden:

$$f a_1 \ldots a_n$$

Achtung: Solche Anwendungen sind nur für *einfache* Ausdrücke f, $a_1 \ldots a_n$ definiert; wenn man allgemeine Ausdrücke benötigt, muß man immer Klammern setzen. Beispielsweise wäre folgendes möglich:

$$(\texttt{fun}\ xy\ \texttt{->}\ 2 * x)\ (3 * 8)\ 4$$

aber das ist nicht legal:

$$f 3 + 4\ 6$$

denn es würde gelesen als

$$((f 3) + 4)\ 6$$

Aber wenn man $((f 3) + 4)$ ausrechnet, kommt eine Zahl heraus, also keine Funktion, die man auf 6 anwenden könnte.

Fazit: Im Gegensatz zu imperativen Sprachen (C, Java) wird beim Funktionsaufruf nicht der Argumentvektor $(a_1, \ldots, a_n)$ als Ganzes geklammert, sondern die einzelnen Argumente $(a_1) \ldots (a_n)$. Die Klammern können nur dann entfallen, wenn die Argumente einfache Ausdrücke, also etwa Namen oder Konstanten, sind. Wenn man dies einmal vergißt, bekommt man gewöhnlich die Fehlermeldung

```
This function is applied to too many arguments
```

weil nämlich ohne die Klammern die Teile eines Arguments als *zwei* Argumente gelesen werden.

Wenn aber `fun`-Ausdrücke wieder Ausdrücke sind, dann kann man sie auch schachteln:

"ex1.ml" 2a ≡
```
print_int
  (((fun x ->
      (fun y ->
         2 * (x * y))) 3) 7)◇
```

Die Auswertung funktioniert in drei Schritten, wie immer von innen nach außen:

1. $(\texttt{fun}\ x\ \texttt{->}\ \ldots)3 \rightsquigarrow (\texttt{fun}\ y\ \texttt{->}\ 2 * (x * y)); x = 3$
2. $(\texttt{fun}\ y\ \texttt{->}\ 2 * (x * y))7; x = 3 \rightsquigarrow 2 * (x * y); y = 7, x = 3$
3. $2 * (x * y); y = 7, x = 3 \rightsquigarrow 42$

In OCaml kann man die innere Klammer auch weglassen, denn diese Klammerung ist der Normalfall (ähnlich der „Punkt-vor-Strich“Regel):

"ex01.ml" 2b ≡
```
print_int
  ((fun x ->
      (fun y ->
         2 * (x * y))) 3 7)◇
```

Anders gesagt kann man eine solche geschachtelte Funktion auf *zwei* Argumente (3, 7) anwenden! Die Auswertung funktioniert so, daß zunächst die Funktion in x angewandt wird mit $x = 3$. Dies ergibt eine weitere Funktion (in y), die auf $y = 7$ angewandt wird. Endlich wird `2*(x*y)` mit diesen Werten für x und y ausgerechnet. Das Endergebnis ist also dasselbe, wie wenn wir direkt geschrieben hätten:

"ex2.ml" 2c ≡
```
print_int ((fun x y -> 2 * (x * y)) 3 7)◇
```

Fazit: Funktionen sind in OCaml Ausdrücke wie alle anderen, sie haben keine Sonderrolle und müssen auch gar nicht benannt werden. Das ist evtl. etwas gewöhnungsbedürftig, aber später extrem hilfreich. Außerdem gilt

$$\texttt{fun}\ x_1 \ldots x_n\ \texttt{->}\ e = \texttt{fun}\ x_1\ \texttt{->}\ \texttt{fun}\ x_2\ \texttt{->}\ \cdots \texttt{fun}\ x_n\ \texttt{->}\ e$$

Auch das wird neu sein; diese Äquivalenz wird auch *Currying* genannt. Wir benutzen sie nur am Rande, nämlich wenn wir Funktionen *teilweise* anwenden:

$$(\texttt{fun}\ x\ y\ \texttt{->}\ x + y)4$$

ist eine Funktion, die noch eine Zahl erwartet, um mit der Berechnung fortzufahren.

Technik: *Funktionen höherer Ordnung* werden bei den *Listen* (Abschnitt 3.7) behandelt.

Achtung: Um eine Funktion ohne Argumente aufzurufen, übergibt man den besonderen Wert `()`. Ein bekanntes Beispiel ist `print_newline`, das einen Zeilenvorschub ausgibt. Der Ausdruck

```
print_newline
```

tut *absolut gar nichts*, denn die Funktion steht für sich selbst. Erst

```
print_newline ()
```

führt wirklich zu einer Ausgabe. Man kann auch selbst solche Funktionen definieren:

```
fun () -> 42
```

ist eine Funktion, die `42` liefert, wenn man sie anwendet (mittels `(fun () -> 42) ()`).

Man kann Ausdrücke auch *hintereinander ausführen*:

$$e_1\texttt{;}e_2$$

Dann wird zunächst e_1 berechnet, sein Ergebnis muß der spezielle Wert `()` sein (man kann immer ein anderes Ergebnis mit `ignore` verwerfen). Dann wird e_2 berechnet und sein Ergebnis ist das Ergebnis des ganzen. Natürlich kann man wieder schachteln, diesmal sogar ohne Klammern:

$$e_1\texttt{;}\cdots\texttt{;}e_n$$

Solche Ausdrücke sind nur dann sinnvoll, wenn e_1 einen Effekt hat, etwa eine Bildschirmausgabe

```
print_int i; print_newline (); i
```

Eine Fallunterscheidung ist durch `if`-Ausdrücke möglich. Anders als in imperativen Sprachen berechnen diese Ausdrücke einen *Wert*:

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$$

Falls e_1 den Wert `true` hat, ist der Wert des Gesamten der von `e_2`, ansonsten ist e_1 = `false` und der Gesamtwert ist e_3. Beispielsweise könnte man schreiben:

```
if a=0 then 0 else a+1
```

2 Definitionen

Mit

$$\texttt{let } x \texttt{ = } e\texttt{;;}$$

kann man den Namen x für den (allgemeinen) Ausdruck e vergeben. Beispielsweise kann man ein Programm schreiben, das `10` ausgibt:

"ex3.ml" 3a ≡

```
let f=fun x -> 2 * x;;
print_int (f 5)◇
```

Achtung: Anders als in anderen Sprachen kann man die Werte von Variablen *nicht im Nachhinein ändern*. Man kann also nicht schreiben `let a=5;; a=7` o.ä. Es gibt eine weitere Form der Definition, die *lokale Definition* (oder *Bindung*, weil sie eine eine Variable an einen Wert bindet)

$$\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2$$

Diese ist ein allgemeiner Ausdruck, was vielleicht ungewöhnlich erscheint. Er wird ausgewertet, indem zuerst e_1 ausgewertet wird und dann e_2. Wenn x in e_2 vorkommt, hat es dort den Wert von e_1; außerhalb von e_2 hat x diese neue Bedeutung nicht mehr.

Besonders interessant ist, daß sowohl e_1 als auch e_2 allgemeine Ausdrücke sind, also evtl. auch wieder solche Definitionen. Man kann also stark geschachtelte Ausdrücke schreiben und ohne Probleme lokale Abkürzungen einführen wo immer man sie braucht.

"ex4.ml" 3b ≡

```
print_int (let a = (let f=fun x -> 3*x in f 4) in
      let b=6 in
        a+b)◇
```

Manche Definitionen sind auch rekursiv. Zum Beispiel kann man die Summe von 1 bis n so brechnen:

$$\texttt{sum}(n) := \begin{cases} 0 & n = 0 \\ n + \texttt{sum}(n-1) & \text{sonst} \end{cases}$$

Das Besondere ist, daß `sum` in der Definition von `sum` wieder vorkommt – allerdings mit einem kleineren Argument. In OCaml braucht man für rekursiv definierte Namen ein `let rec` statt dem `let` und dies gilt für beide Sorten von Definitionen.

"ex5.ml" 4a ≡

```
let rec sum =
  fun n ->
    if n=0
    then 0
    else n+(sum (n-1));;
print_int (sum 10)◇
```

Wieder ist `let rec` auch ein Ausdruck, wobei f eine Funktion sein *muß*!

$$\texttt{let rec } x \texttt{ = } f \texttt{ in } e$$

Gleichzeitige Definitionen Manchmal ist es sinnvoll, mehr als eine Variable gleichzeitig zu definieren; dies geschieht durch `let` bzw. `let rec` und `and`

$$\begin{aligned}&\texttt{let } x_1 = e_1\\ &\texttt{and } x_2 = e_2 \cdots\\ &\texttt{in } e\end{aligned}$$

Dabei werden *alle* Ausdrücke e_i ausgewertet, bevor die Variablen x_i definiert werden. Die Reihenfolge der Auswertung ist nicht vorgegeben, es kann also durchaus e_4 vor e_1 und dann e_6 ausgewertet werden.

Achtung: Wenn man `let .. and` benutzt, sollte man sich immer kurz überlegen, ob es *wirklich* egal ist, in welcher Reihenfolge die e_i ausgewertet werden. Es gilt die Regel, daß wenn einer der Ausdrücke Seiteneffekte (Ausgaben, etc.) hat, die Reihenfolge relevant ist; natürlich gilt es auch, wenn einer der Ausdrücke eine Funktion aufruft, die Seiteneffekte hat. Wenn man im Zweifel ist, schreibt man besser

$$\texttt{let } x_1 = e_1 \texttt{ in let } x_2 = e_2 \texttt{ in } \cdots$$

Das `and` ist manchmal unumgänglich, nämlich bei rekursiven Funktionen, die sich gegenseitig aufrufen:

"ex6.ml" 4b ≡

```
let rec even =
  fun x ->
    if x=0
    then true
    else odd (x-1)
and odd =
  fun x ->
    if x=0
    then false
    else even (x-1);;

if (even 6)
then print_string "even"
else print_string "odd"◇
```

3 Datentypen

Typen sind grundsätzlich dazu da, dem Compiler mitzuteilen, wie genau die Daten aufgebaut sind, mit denen Programme rechnen. Daraufhin kann der Compiler überprüfen, daß Daten nicht versehentlich falsch verwendet werden.

```
3 + "5"
```

ist beispielsweise illegal, denn wie soll man mit einer Zahl und einer Zeichenkette rechnen? Die sogenannten *primitiven* Datentypen sind gerade `int` für ganze Zahlen, `char` einzelne Zeichen, `string` für Zeichenketten, und `bool` für die Werte `true` und `false`.

Außerdem muß der Compiler irgendwie nachvollziehen, wie Funktionen Werte behandeln. Beispielsweise erwartet `not` einen Wert `bool` und liefert den negierten `bool` Wert zurück. Wir schreiben

```
not : bool -> bool
```

und sagen „`not` ist eine Funktion von `bool` nach `bool`“; der Doppelpunkt bedeutet hier „hat den Typ“, `->` ist der Funktionstyp, bei dem links das erwartete Argument und rechts der Rückgabewert steht. Dann kann der Compiler sehen, daß `(not true)` einen Wert vom Typ `bool` haben muß:

1. `true` ist vom Typ `bool`
2. `not` ist vom Typ `bool -> bool`
3. `not true` ist die Anwendung von `not`; diese ist legal, weil `not` ein `bool` Argument erwartet.
4. Das Ergebnis ist wieder vom Typ `bool`

Bei Funktionen mit mehr als einem Argument verwendet man Currying: Beispielsweise hat `+` den Typ `int -> int -> i` genauso wie die Funktion

$$\texttt{fun } x\texttt{->fun } y\texttt{->}x+y$$

und entsprechend

$$\texttt{+} : \texttt{int -> int -> int} = \texttt{int -> (int -> int)}$$

Um nun zu überprüfen, daß $(1+3) : \texttt{int}$, rechnet der Compiler aus:

1. $1 : \texttt{int}$
2. $\texttt{+} : \texttt{int -> (int -> int)}$
3. $(\texttt{+1}) : \texttt{int -> int}$
4. $3 : \texttt{int}$
5. $((\texttt{+1})3) : \texttt{int}$

Hinweis: Typen werden auch in der Vorlesung nochmals behandelt. Bis dahin nehmen wir sie als Hilfsmittel, unsere Programme korrekt zu schreiben:

> Wenn der Compiler einen Typfehler meldet, haben wir irgendwelche Daten oder Funktionen nicht so verwendet, wie es eigentlich gedacht war. Entweder haben wir die Definition falsch aufgeschrieben oder sie falsch verwendet. In jedem Fall würde ein Programm, das die Typüberprüfung nicht besteht, zur Laufzeit Unsinn berechnen; *Typfehler sind also kein Ärgernis, sondern frühe Hinweise auf Fehler, die man ohnehin irgendwann beseitigen muß.*

Bei Definitionen `let` x`=`e `in` e' hat x den Wert von e innerhalb von e' – es ist also nur natürlich, daß x in e' auch den *Typ* von e hat. Diesen Typ kann man explizit angeben, und das ist auch drigend anzuraten, da sonst die Typfehler mitunter recht unverständlich sind. Vor allem aber helfen solche Angaben, sich zuerst einmal selbst über die Struktur der Eingabe klar zu werden

$$\texttt{let } x \texttt{ : } \mathit{Typ} \texttt{ = } e \texttt{ in } e'$$

Beispielsweise würde man *immer* schreiben:

"ex7.ml" 5 ≡

```
let rec sum : int -> int =
  fun n ->
    if n=0
    then 0
    else n+(sum (n-1));;
print_int (sum 10)◇
```

Leitlinie: Der OCaml-Compiler kann zwar grundsätzlich die Typen, die man beim `let` aufschreiben kann, bei typkorrekten Programmen auch selbst berechnen — wenn jedoch Typfehler vorhanden sind, verkommt diese Berechnung oft zum Raten mit unerwarteten Ergebnissen, so daß man am Ende nicht mehr weiß, ob der eigentliche Fehler in der Definition oder der Verwendung liegt.

Bei Übungsaufgaben sind bei globalen Definitionen die Typen auf jeden Fall anzugeben!

Gelegentlich, aber in dier Vorlesung äußerst selten, kommt es vor, daß es einer Funktion egal ist, wie die Werte aussehen, auf denen sie rechnet.

$$\texttt{id} = \texttt{fun}\ x\texttt{->}x$$

ist eine solche Funktion. Wie man sieht, tut sie nichts, und das ist auch die Gemeinsamkeit aller solcher Funktionen: Sie dürfen die Werte nur weiterkopieren, aber niemals wirklich damit rechnen. Schon daher werden sie ziemlich spärlich sein.

Wenn es jedoch einer Funktion wirklich egal ist, kann man dies durch *Typvariablen* `'a`, `'b`, ... ausdrücken. Diese stehen für beliebige, aber feste Typen. Die Funktion `id` hat etwa den Typ `'a -> 'a`, man sagt sie sei *polymorph*. Diese Besonderheit brauchen wir jedoch nur beim Lesen zu verstehen, aktiv schreiben werden wir solche Funktionen nicht.

3.1 Paare und Tupel

Einige Datentypen kommen so häufig vor, daß ihre Definition in die Sprache eingebaut ist. *Paare*

$$\texttt{(}x\texttt{,}y\texttt{)}$$

haben den Typ

$$s\ \texttt{*}\ t$$

wenn x und y die Typen s und t haben. Beispielsweise gilt:

$$(\texttt{"a"}, 42) : \texttt{string} * \texttt{int}$$

Dies ist analog zu Funktionen, wo ja eine Funktion den Typ s`->`t hat, wenn sie Argumente vom Typ s nimmt und Werte vom Typ t liefert.

Man kann die Komponenten der Paare durch `match` wieder trennen:

$$\texttt{match}\ e\ \texttt{with}\ P\ \texttt{->}\ e'$$

Dabei ist P ein *Pattern*. Bis jetzt kennen wir nur eine Möglichkeit, Paare:

$$\texttt{match}\ p\ \texttt{with}\ (a,b)\ \texttt{->}\ a+b$$

Außerdem kann man die Funktionen `fst` und `snd` verwenden. Der Bezug ist so:

"ex8.ml" 6 ≡

```
let fst' : ('a * 'b) -> 'a =
  fun p ->
    match p with
        (a,b) -> a
let snd' : ('a * 'b) -> 'b =
  fun p ->
    match p with
        (a,b) -> b◇
```

Paare werden erweitert zu *Tupeln* $(x_1 \dots x_n)$, die den Typ $(t_1\texttt{*}\cdots\texttt{*}t_n)$ haben, wenn x_i den Typ t_i hat. Paare und Tupel dienen einfach dazu, mehrere Werte zu einem zusammenzufassen.

3.2 Zusammenfassung

Ein Sonderfall ist noch zu besprechen, nämlich der besondere Wert `()`, der benötigt wird, um Funktionen ohne Argumente aufzurufen. Er hat den besonderen Typ

$$\texttt{()} : \texttt{unit}$$

Mit dem bisher gesagten haben also Typen immer den Aufbau:

$$\begin{aligned} t = {} & \texttt{int} \mid \texttt{string} \mid \texttt{bool} \mid \texttt{char} \\ & \mid \texttt{unit} \\ & \mid t_1\ \texttt{*}\ t_2 \cdots\ \texttt{*}\ t_n \\ & \mid t_1\ \texttt{->}\ t_2 \\ & \mid (t) \end{aligned}$$

Die letzte Klausel ist analog zu Ausdrücken: Man kann Typen immer beliebig tief klammern, um die Anteile abzugrenzen.

3.3 Fallunterscheidungen

Wir haben schon ohne Umschweife mit ganzen Zahlen gerechnet. Man kann sich jedoch Datentypen auch selbst als Fallunterscheidungen definieren. Wenn man etwa schreibt

```
type a_oder_b = A | B
```

dann sind `A` und `B` neue *Konstruktoren* für den Typ `a_oder_b`. Wenn man nun einen Wert x hat, kann man mit `match` prüfen, welcher der Fälle vorliegt. Die Rückgabe des `match` Konstruktes ist die Rückgabe des zutreffenden Falles.

"ex9.ml" 7a ≡
```
type a_oder_b = A | B
let welches =
  fun x ->
    match x with
        A -> "A"
      | B -> "B"◇
```

OCaml ist dabei *typsicher*, d.h. es können immer nur die Fälle *eines* Typs zusammen auftreten:

"ex10.ml" 7b ≡
```
type a_oder_b = A | B
type c_oder_d = C | D
let welches =
  fun x ->
    match x with
        A -> "A"
      | C -> "C"◇
```

Das ergibt einen Fehler:

```
This pattern matches values of type c_oder_d
but is here used to match values of type a_oder_b
```

Diese Art der Fehlerüberprüfung wird später sehr hilfreich sein, denn der Compiler kann herausfinden, ob wirklich alle Fälle abgedeckt sind, und natürlich auch Denkfehler abfangen, bei denen man nicht sicher ist, welche der Fälle überhaupt zu unterscheiden sind.

Leitlinie: Wann immer man eine Eingabe von einem Typ mit Fallunterscheidung bekommt, beginnt man die Funktion mit einer Fallunterscheidung, in der man *alle* Fälle des entsprechenden Typs auflistet – am besten kopiert man erst einmal die Definition des Typs als `match` Ausdruck.

3.4 Konstruktoren mit Argumenten

Die Konstruktoren können auch Argumente haben, damit man bei jedem Fall auch die relevanten Daten festhalten kann. Beispielsweise kann man einen Typ `anzahl` definieren mit den Fällen `Keines`, `Eins`, `Zwei`, `Mehr(n)`. Beim `match` wird dann ein Muster (*pattern*) angegeben, das auf den jeweiligen Fall passt. Ein Pattern ist ein Ausdruck, in dem nur Konstruktoren, Variablen und Konstanten vorkommen. Die Variablen stehen für beliebige Argumente des jeweiligen Konstruktors und wenn das Pattern auf den Wert passt, dann haben die Variablen den entsprechenden Wert.

"ex11.ml" 7c ≡
```
type num =
    Float of float
  | Int of int
⟨ add 8a ⟩
⟨ is_int ? ⟩
let x = add (Float 0.34) (Int 3)
and b = add (Int 2) (Int 3)
in ();;◇
```

⟨ add 8a ⟩ ≡

```
let add =
  fun a b ->
    match (a,b) with
        (Int i, Int j) -> Int (i+j)
      | (Int i, Float y) -> Float ((float_of_int i) +. y)
      | (Float x, Int j) -> Float (x +. (float_of_int j))
      | (Float x, Float y) -> Float (x+.y);;◇
```

Macro referenced in 7c.

Eine besondere Variable in Patterns ist „`_`“ (gelesen als *don't care*). Diese namenlose Variable passt auf jeden Wert, aber dieser kann nicht mehr abgefragt werden. Im Gegensatz zu normalen Variablen, die in jedem Pattern nur einmal vorkommen dürfen, kann „`_`“ mehrfach vorkommen und steht dann für möglicherweise verschiedene Werte.

"is_int" 8b ≡

```
let is_int =
  fun i ->
    match i with
        Int _ -> true
      | _ -> false◇
```

3.5 Datentypen mit Polymorphie

Viele Datentypen definieren nur einen Teil ihrer Strukur. Beispielsweise könnte man Tripel so definieren:

```
type 'a 'b 'c tripel = Tripel of 'a * 'b * 'c
```

Das einzige, was `tripel` mit ihren Argumenten tun, ist sie speichern und beim `match` wieder abzugeben. Wie schon bei Funktionsdefinitionen stehen die Typvariablen `'a`, `'b`, `'c` für beliebige, aber feste Typen. Wenn man einen solchen Typ verwenden will, muß man die Parameter einsetzen; bei dieser Art von „Anwendung“ stehen die Argument *links* vom Typen der angewendet wird, nicht rechts wie bei Funktionen.

```
int string int tripel
```

benutzt den Typ `tripel` mit `'a=int`,`'b=string`. Dabei können Typen wieder geklammert werden, und wie bei den Funktionen gilt, daß man grundsätzlich Argumente klammert, es sei denn diese bestehen nur aus einem Namen.

3.6 Rekursion

Wer kennt das nicht? Wenn man einmal aufräumt, findet man in jeder Schachtel (fast) immer eine weitere Schachtel, und mitten drin irgendwelche Dinge.

⟨ Schachteln 8c ⟩ ≡

```
type schachtel =
    Ding of string
  | Schachtel of string * schachtel◇
```

Macro referenced in 8d.

Dieser Datentyp ist *rekursiv*: Wenn man eine Schachtel aufmacht, findet man evtl. wieder eine Schachtel. Für eine Funktion, die alle Dinge in einer Schachtel auflistet, müssen wir also eine Fallunterscheidung machen (siehe vorheriger Abschnitt) und dann evtl. rekursiv vorgehen (für die Schreibeweise `Printf.printf` siehe Abschnitt 8):

"ex12.ml" 8d ≡

```
⟨ Schachteln 8c ⟩
let rec dinge =
  fun s ->
    match s with
        Ding(d) -> Printf.printf "%s\n" d
      | Schachtel(d,s') ->
          Printf.printf "%s\n" d;
          dinge s'◇
```

Leitlinie: Rekursive Datentypen behandelt man mit rekursiven Funktionen. Die Punkte, an denen sich die Funktion rekursiv aufruft sind gerade die Punkte, an denen auch der Datentyp rekursiv wird. *Diese Form der Rekursion ist also besonders einfach.* Man nennt sie auch die *strukturelle Rekursion*, da sie entlang der Struktur der Daten vorgeht.

3.7 Listen

Listen sind eingebaute Datentypen, die ungefähr so aussehen wie

```
type 'a list = Cons of 'a * 'a list | Nil
```

Allerdings schreibt man `Cons` als `::` und `Nil` als `[]`. Bei `::` heißen die Argumente *head* und *tail* und man kann mit

`List.hd : 'a list -> 'a` und `List.tl : 'a list -> 'a list`

darauf zugreifen. Die (rekursive) Fallunterscheidung funktioniert wie vorher schon.

"ex13.ml" 9a ≡
```
let rec len : 'a list -> int =
  fun l ->
    match l with
        [] -> 0
      | hd :: tl -> 1+(len tl)◇
```

Listen können auch direkt notiert werden, indem man ihe Elemente mit `[ ]` eingeklammert und durch `;` trennt.

```
len [ 2; 3; 5 ]
```

Man beachte, daß man in einer Liste immer nur *einen* Typ von Elementen haben kann, es ist nicht legal

```
[ 2; "Hallo" ]
```

zu schreiben.

Fazit: Listen sind Folgen von gleichartigen Elementen, die durch einen Konstruktor `::` aufgebaut werden. Das Ende der Liste ist durch `[]` markiert. Funktionen auf Listen werden üblicherweise rekursiv durch einfache Fallunterscheidung definiert.

Die Listen sind einer der wichtigsten Datentypen in OCaml überhaupt und es gibt sehr viele mächtige eingebaute Funktionen.

Assoziationslisten Eine Liste von Paaren heißt eine *Assoziationsliste*. Man kann die Funktion `List.assoc` benutzen; diese Funktion sucht in einer Liste von Paaren nach einem Wert x. Beim ersten (x', y), bei dem $x = x'$ ist, liefert sie y als Resultat.

"ex14.ml" 9b ≡
```
let l = [ ("a", 3); ("b",7); ("c",9) ];;
print_int (List.assoc "b" l); print_newline ()◇
```

Technik: *Funktionen höherer Ordnung* sind Funktionen, die andere Funktionen als Argumente nehmen oder zurückliefern. Wenn h eine Funktion höherer Ordnung ist, dann ist ihr Verhalten sozusagen *im wesentlichen* bestimmt; an den Stellen, an denen das Funktionsargument vorkommt, kann sich das Verhalten noch verändern. Erst bei einem Aufruf $h\ g$ werden diese letzten Stellen auch noch festgelegt.

Funktion höherer Ordnung sind bei Listen extrem hilfreich, da man oft dieselbe Funktion einfach auf alle Elemente einer Liste, hintereinder oder gleichzeitig, anwenden will. Um etwa bei einer `int list` alle Argumente zu verdoppeln, kann man schreiben:

"ex15.ml" 9c ≡
```
let rec double_list =
  fun l ->
    match l with
        [] -> []
      | hd :: tl -> (2*hd) :: (double_list tl);;

double_list [ 2;3;4 ]◇
```

Das Schema von `double_list` ist uns nun sattsam bekannt: Mit `match` wird die Liste durchlaufen und dann jeweils eine Aktion ausgeführt (`+1` rechnen, verdoppeln). Das ist langweilig!

"ex16.ml" 10a ≡
```
let rec with_each_element : ('a -> 'b) -> 'a list -> 'b list =
  fun f l ->
    match l with
        [] -> []
      | hd :: tl -> (f hd) :: (with_each_element f tl);;
with_each_element (fun x->2*x) [ 2;3;4 ]◇
```

Diese Funktion ist eingebaut als `List.map`. Sie erzeugt aus `l` eine neue Liste, indem sie die Funktion `f` auf jedes der Elemente aus `l` anwendet und aus den Ergebnissen eine neue Liste aufbaut.

Hinweis: Eine weitere wichtige Funktion ist `List.fold_left` (bzw. `List.fold_right`). Damit kann man häufig sehr elegante Definitionen angeben, aber man braucht etwas Übung. Ohne diese Funktionen muß man etwas mehr `let rec` schreiben, kommt aber auch immer durch. Lies die Dokumentation zu diesen beiden gelegentlich durch, wenn Du Dich an die Listen selbst gewöhnt hast.

3.8 Ein Wort zur Repräsentation

Wenn man Daten im Rechner ablegt, gibt es eigentlich immer zwei Arten von „Gleichheit“:

x==y: x und y sind an derselben Stelle im Speicher abgelegt, d.h. sie sind durch ein und dasselbe Speicherobjekt repräsentiert.

x=y: x und y haben die gleiche Struktur, ihre Komponenten sind alle `=`.

Wenn x==y, dann ist offenbar auch x=y.

"ex17.ml" 10b ≡
```
type tst = A | B of int;;

let x = B(5)
in (fun y ->
      print_string ("0"^(if x == y then "Y " else "N ")))
     x;;

let tst_equ =
  fun n x y ->
    print_string ((string_of_int n)^(if x = y then "Y " else "N "));;

let tst_eq =
  fun n x y ->
    print_string ((string_of_int n)^(if x == y then "Y " else "N "));;

tst_eq  1 A A;;
tst_eq  1 'a' 'a';;
tst_eq  2 (B(3)) (B(3));;
tst_equ 3 (B(3)) (B(3));;
let x=B(3) in tst_eq 4 x x;;
tst_eq  5 "a" "a";;
tst_equ 6 "a" "a";;
tst_eq  7 2 2;;◇
```

Die Ausgabe besteht für jeden Testfall aus einer Ziffer, die man leicht im Programm wiederfindet und dem Ergebnis `Y` oder `N` des Testes. Der Testfall `0Y` hat besondere Bedeutung: Weil dor `Y` herauskommt, können wir überhaupt *Funktionen* `tst_equ` und `tst_eq` für die anderen Fälle benutzen. (Warum?)

```
0Y 1Y 1Y 2N 3Y 4Y 5N 6Y 7Y
```

In OCaml gilt:

- Beim Funktionsaufruf (bei Definitionen) bleibt `==` erhalten, d.h. die Variablen stehen genau für das übergebene Objekt (für den errechneten Wert), es werden niemals Kopien angelegt. (Beispiel `0Y`)

- Konstruktoren ohne Argumente sind immer `==`; sie sind sozusagen Konstanten.
- Auf `int`, `char` sind `=` und `==` gleichbedeutend.
- Auf `string` unterscheiden sie sich: Strings sind schon `=`, wenn sie die gleichen Zeichen enthalten.

Fazit: Normalerweise wird man die Gleicheit `=` in Programmen verwenden; nur sehr selten ist es notwendig, `==` (die in gewisser Weise eine *schärfere* Form der Gleichheit ist) zu benutzen – das geht eigentlich nur dann gut, wenn man sicher ist, daß die zu vergleichenden Objekte nicht kopiert wurden.

4 Imperative Elemente

In OCaml programmiert man nicht imperativ, wenn es sich verhindern läßt.

In imperativen Sprachen haben Variablen einen Speicherplatz, dessen Inhalt man im Nachhinein ändern kann. In OCaml muß man dies explizit vorsehen, in dem man die Variable nicht an den Wert direkt, sondern an eine Art Box bindet. Die Funktion `ref` erzeugt eine neue Box, die ihr Argument als Wert enthält. Die Zuweisung, d.h. die Änderung des Inhaltes einer Box, wird mit dem Operator `:=` geschrieben.

```
let a = ref(5) in a:= !a + 1
```

Die Typen werden entsprechend erweitert um einen Typen `'a ref`, der genau Boxen mit Inhalt vom Typ `'a` bezeichnet. Dieser Datentyp ist wieder polymorph, der Inhalt wird nur festgehalten und wiedergegeben.

Achtung: Selbst im obigen Beispiel wird der Wert der Variablen `a` *nicht* verändert: `a` ist immer ein und dieselbe Box, die durch `ref(5)` erzeugt wurde – nur der *Inhalt* dieser Box ändert sich.

Auf `ref`-Boxen greift man immer mit diesen Operatoren zu:

`:=` (Zuweisung) und `!` (Dereferenz)

Wenn man erst einmal solche veränderbaren Werte hat, möchte man natürlich auch Schleifen schreiben

`while` e `do` e' `done`

und

`for` x `=` i `to` j `do` e' `done`

leisten das Gewünschte. Außerdem kann man das einarmige

`if` e `then` e'

verwenden. Dies ist gleichbedeutend mit

`if` e `then` e' `else ()`

5 Records

Records sind im Grunde einfach Tupel mit benannten Feldern. Anstatt beispielsweise eine Variablenbelegung als `("a",5)` festzuhalten, könnte man lesbarer schreiben:

```
{ var_name = "a"; var_value = 5 }
```

Dann sind `var_name` und `var_value` *Labels* („Marken“). Records werden streng in Typen gegliedert und jedes Label darf nur in einem Record-Typ vorkommen. Man kann auf die Felder eines Records sowohl über das normale `match`, als auch über die Labels zugreifen.

"ex18.ml" 12a ≡
```
type r = { var_name : string; var_value : int };;
let l = [ { var_name="a"; var_value=5 };
          { var_name="b"; var_value=6 } ]
in List.iter
     (fun v ->
   match v with
       { var_name = n } ->
             print_string (n^"=");
         print_int v.var_value;
         print_string " ")
     l◇
```

Indem `var_name` innerhalb des `match`-Patterns vorkommt, ist schon festgelegt, daß `v` den Typ `r` haben muß.

Hinweis: Beim `match` auf einen Record muß man nicht alle Felder aufzählen, die fehlenden werden mit `_` ergänzt. Beispielsweise könnte man oben äquivalent schreiben:

```
match v with { var_name = n; var_value = _ } ->
```

Beim *Erzeugen* von Record-Werten müssen aber immer *alle* Felder angegeben sein (dort kann der Compiler ja keine sinnvollen Werte „erraten“).

Veränderbare Felder Man kann einzelne Felder auch als `mutable` markieren und dann die dort gespeicherten Werte überschreiben; anders als bei Boxen (s.o.) wird die Zuweisung hier als `<-` geschrieben.

"ex19.ml" 12b ≡
```
type r = { var_name : string; mutable var_value : int };;
let l = [ { var_name="a"; var_value=5 };
          { var_name="b"; var_value=6 } ];;
List.iter (fun v -> v.var_value <- 0) l◇
```

Achtung: Die Zuweisung geschieht durch den Operator `<-` und *nicht* durch den Operator `:=`.

6 Arrays

Arrays sind ähnlich wie Listen Folgen von Werten. Allerdings kann man auf alle Elemente sofort zugreifen, man muß nicht von vorn nach hinten durch das Array laufen. Anders als bei Listen können die Einträge auch verändert werden. Wieder geschieht die Zuweisung mit dem Operator `<-`, auslesen kann man einzelne Elemente mit `.(`i`)` (für die Schreibeweise `Printf.printf` siehe Abschnitt 8):

"ex20.ml" 12c ≡
```
let a = [| 2; 5; 7; 9 |];;
for i = 0 to ((Array.length a)-1) do
  Printf.printf "%d -> %d\n" i a.(i)
done;;
Array.iteri (fun i _ -> a.(i) <- 0) a;;
for i = 0 to ((Array.length a)-1) do
  Printf.printf "%d -> %d\n" i a.(i)
done◇
```

In der Standard-Bibliothek gibt es eine reichhaltige Auswahl an Funktionen für Arrays: Kopieren, Abschnitte auswählen, Laufen über das Array mit und ohne Index, ein `map` ähnlich wie für Listen.

7 Ausnahmen

Ausnahmen zeigen an, daß das erwartete Resultat nicht berechnet werden konnte, sei es, daß ein echter Fehler passiert ist (Datei nicht gefunden, etc.) oder einfach ein Spezialfall aufgetreten ist. In OCaml sind Ausnahmen (ziemlich) normale Werte, und man kann sie mit Ausdrücken berechnen etc. An der Stelle, an der der Fehler entdeckt wird, wird die Ausnahme *geworfen*:

```
raise e
```

Wenn man umgekehrt weiß, daß ein Ausdruck e vielleicht eine Ausnahme a wirft, dann kann man sich dafür zuständig erklären, diese Ausnahme zu behandeln:

$$E = \texttt{try } e; e' \texttt{ with } a \texttt{ -> } e''$$

Wenn e einen Wert berechnet, ohne die Ausnahme zu werfen, wird dieser Wert als Wert von E verwendet. Ansonsten wird e' erst gar nicht ausgeführt, sondern direkt e'' – und dessen Wert ist dann der Wert von E. Wenn allgemeiner eine Ausnahme in einem Unterausdruck von e vorkommt, wir der Rest der Berechnung in e gar nicht mehr zu Ende geführt.

"ex21.ml" 13a ≡
```
let l = [ ("a", 3); ("b",7); ("c",9) ];;
print_int
  (try
     List.assoc "f" l
   with
       Not_found -> 0)◇
```

Man kann neue Ausnahmen definieren, und diese können dann auch Werte haben, die den aufgetretenen Fehler genauer beschreiben (für `Printf.printf` siehe Abschnitt 8):

"ex22.ml" 13b ≡
```
exception Unbound of string
let lookup_var : string -> (string * int) list -> int =
  fun s l ->
    try
      List.assoc s l
    with Not_found -> raise (Unbound(s));;

try
  print_int (lookup_var "a" [ ("b",5) ])
with Unbound(v) ->
  Printf.printf "Variable '%s' was undefined\n" v◇
```

Fazit: Das Interessante an Ausnahmen ist, daß man sie dort wirft, wo der Fehler auftritt, d.h. das erwartete Ergebnis nicht berechnet werden kann. An der Stelle, wo man mit dem Fehler umgehen kann (möglicherweise erst im Hauptprogramm durch Programmabbruch!) behandelt man sie. Zwischen diesen Punkten kann man das Programm so schreiben, als würden alle erwarteten Ergebnisse auch wirklich fehlerfrei bereit gestellt, man muß nicht umständliche Fallunterscheidungen einbauen.

"ex23.ml" 13c ≡
```
let lookup_var : string -> (string * int) list -> int option =
  fun v l ->
    let rec findin l =
      match l with
     [] -> None
   | (x,i) :: tl when x = v -> Some(i)
   | _ :: tl -> (findin tl)
    in findin l;;

let findv : string -> int option =
  fun v->
    match (lookup_var v [ ("b",5) ]) with
   None -> None
      | Some(i) -> Some(i);;

let v = "a"
in match (findv v) with
    Some(i) -> print_int i
  | None -> Printf.printf "Variable '%s' was undefined\n" v ◇
```

8 Module

Module in OCaml sind Ansammlungen von mit `let` global definierten Funktionen. Wir werden nur sehr am Rand damit zu tun haben, da die Rahmenprogramme Module zwar verwenden, aber ihr nie selbst welche

schreiben müsst. Ein Modul wird „zusammengepackt“ durch Klammern

```
struct ... end
```

die eine Folge von `let` Definitionen einschließen. Man kann Module auch benennen:

```
module M = struct ... let f = ... end
```

und dann mit

```
M.f
```

auf das `f` im Modul `M` zugreifen. `M.f` ist der *qualifizierte Name* von `f`. Um Schreibarbeit zu sparen, kann man auch sagen

```
open M
```

Danach ist (bis zum Ende des aktuellen Moduls) `f` eine Abkürzung für `M.f`.

Jede Datei *xy*`.ml` ist auch gleichzeitig ein Modul *Xy*, d.h. der erste Buchstabe wird groß geschrieben.

Geschachtelte Module In OCaml können Module wieder Module enthalten. Das einzige, was sich dabei ändert, ist der Zugriff auf die enthaltenen Funktionen über die `.`-Notation.

"ex27.ml" 14a ≡

```
module A =
struct
  module B =
  struct
    module C =
    struct
      let f = fun () ->
        print_string "Hallo, verschachtelte Welt!\n"
    end
  end
end;;
A.B.C.f ()◇
```

Modultypen zählen nur die Typen der Definitionen in einem Modul auf. Modultypen heißen auch *Signaturen* und werden entsprechend „zusammengepackt“ durch Klammern `sig .. end`.

"ex24.ml" 14b ≡

```
module type TWICE =
sig
  val twice : int -> int
end

module M : TWICE =
struct
  let two : int = 2
  let twice : int -> int =
    fun x -> two*x
end;;

M.twice 2◇
```

Mit der Zuschreibung `M : TWICE` überprüft der Compiler, daß die aufgezählten Definitionen im Modul wirklich vorhanden sind und beschränkt den Zugriff auf diese *exportierten* Werte. Beispielsweise kann man am Ende der Datei nicht mehr auf `M.two` zugreifen, es ist ein interner Wert von `M`. Man sagt auch, daß `TWICE` die *Schnittstelle* von `M` ist.

Offenbar ist es auf diese Art möglich, die Implementierung von Funktionen in Modulen zu verstecken und nur mit den Schnittstellen zu arbeiten; man kann sogar einen Schritt weiter gehen und auch die Typen zu *abstrakten Typen* machen:

⟨ Interface 15a ⟩ ≡

```
module type I =
sig
  type t
  val a : t
  val f : t -> t
  val print_t : t -> unit
end◇
```

Macro referenced in 15c.

⟨ Module 15b ⟩ ≡

```
module M =
struct
  type t=int
  let a = 5
  let f i = 2*i
  let print_t = print_int
end◇
```

Macro referenced in 15c.

"ex25.ml" 15c ≡

```
⟨ Interface 15a ⟩
⟨ Module 15b ⟩
module Ma = (M : I);;
print_int (M.f 4);;
Ma.print_t (Ma.f Ma.a);;
(* Nicht erlaubt: print_int (Ma.f Ma.a)
   This expression has type Ma.t but is here used with type int
*)◇
```

Parametrisierte Module Man kann also in OCaml Module an Bezeichner binden (s.o.); der logisch (?) nächste Schritt ist es, auch Konstruktionen zu erlauben wie: *Wenn M ein Module mit Schnittstelle I ist, dann ist N auch ein Modul.* Solche *parametrisierten Module* heißen *Funktoren.* Wieder gilt, daß sie zwar im Programmtext manchmal vorkommen, aber ihr müßt sie ind den Übungen nicht selbst schreiben.

Die Parameter werden durch Moduletypen eingeschränkt. Beispielsweise definiert `COMP` einen Typ, den man vergleichen kann.

⟨ COMP 15d ⟩ ≡

```
module type COMP =
sig
  type t
  val leq : t -> t -> bool
end◇
```

Macro referenced in 16.

Auf dieser Grundlage kann man Mengen als geordnete Listen verwalten:

⟨ Set 15e ⟩ ≡

```
module Set(Cmp : COMP) =
struct
  type set = Cmp.t list
  let empty : set = []
  let rec insert : Cmp.t -> set -> set =
    fun x s ->
      match s with
    [] -> [x]
   | y :: tl when Cmp.leq y x -> y::(insert x tl)
   | _ :: tl -> x :: s
end◇
```

Macro referenced in 16.

Schließlich kann man `Set` mit einem Modul `CmpI` instanziieren:

"ex26.ml" 16 ≡

```
⟨ COMP 15d ⟩
⟨ Set 15e ⟩
module CmpI =
struct
  type t = int
  let leq i j = i <= j
end
module S=Set(CmpI)
let s=S.insert 3 (S.insert 2 (S.insert 5 (S.empty)));;
List.iter print_int s◇
```

9 Implizites `match`

Bisher wurden die elementaren Ausdrücke aufgezeigt und mit diesen kommt man durch alle Übungsaufgaben; Anfängern in OCaml sei sogar ausdrücklich nahegelegt, sich zunächst auf diese Ausdrucksmittel zu beschränken.

Die hier geschilderten Abkürzungen braucht man nur zum Lesen der Rahmenprogramme; später einmal wird man sich dabei ertappen, wie man ohne Nachzudenken diese „Phrasen“ übernimmt.

Schon nach kurzer Zeit merkt man, daß sich die Muster bei Funktionsdefintionen wiederholen:

```
let f =
  fun v ->
    match v with
        P1 ->
      | P2 ->
```

Ärgerlich sind hier zwei Dinge:

- Man muß das `v` einführen, bloß um direkt seinen Wert zu zerlegen.
- Man muß explizit `fun` schreiben, obwohl doch klar ist, daß `f` eine Funtion darstellt.

Für beides kann man Abhilfe schaffen, aber bitte alles der Reihe nach.

```
let f = function
     P1 ->
   | P2 ->
```

Das kommt der Ansicht nahe, daß `f` eine Funktion ist, die allein durch Fallunterscheidung *auf dem ersten Argument* definiert wird. Man beachte, daß anders als `fun`, `function` nicht mehr als ein Argument haben kann.

Wenn eine Fallunterscheidung nicht nötig ist, weil der Typ von `v` nur eine mögliche Form zuläßt, dann kann man auch `fun` statt `function` benutzen, und das dann auch mit mehreren Argumenten:

```
let pair_first = fun (a,_) (b,_) -> (a,b)
```

Um auf den zweiten Punkt zurückzukommen: Man kann auch das `fun` einsparen: Wann immer nach der definierten Variable kein `=` (oder `Typ`) folgt, wird der Compiler annehmen, daß eigentlich eine Funktion definiert werden soll und die folgende Form zur oben gegebenen umwandeln:

```
let pair_first (a,_) (b,_) = (a,b)
```

In dieser Form sind endlich beide „Ärgernisse“ beseitigt.