made light of

Typen in OCaml

`int`	vorzeichenbehafteter Integer
`float`	vorzeichenbehafteter Gleitkommazahl
`bool`	Boole'scher Wert: `true` oder `false`
`char`	Schriftzeichen
`string`	Zeichenkette
`'a list`	Liste, deren Elemente Typ 'a haben (`int list` = Integer Liste)
`unit`	nichts; der Typ des Resultats Funktionen, die kein Objekt zurück geben (leer).

```
type expression =
Var of string
| Const of int
| Add of expression * expression
| Mul of expression * expression
```

Selbstdefinierter (hier rekursiver) Typ (Beispiel).

z.B.: `# Add ( Mul (Const 2 , Var "x") , Var "y");;` $\rightsquigarrow$ $2x + y$

OCaml Toolkit

`#use "filename";;`	Die Inhalte der Datei "filename" evaluieren.
`let x=4 and y=5;;`	Die Variablen `x` und `y` global definieren.
`let x=9 in 3 * x;;`	Die Variable `x` local in der expression `3 * x` definieren.
`+, - , *, /`	Integer Addition, Substraktion, Multiplikation, Division
`+., -., *., /.`	Float Addition, Substraktion, Multiplikation, Division
`<, >, <=, >=`	Vergleichen von Zahlen, Zeichen, alles was geordnet ist.
`"OCaml"`	Die Zeichenkette `OCaml`.
`'a'`	Das Zeichen `a`
`List.length "Logic"`	Länge der Zeichenkette "Logic" `( - :  int = 5)`
`"pro" ^ "of" = "proof"`	Zeichenkettenkonkatenation
`"\n"`	Zeilenende bei Zeichenkettem.
`||`	Boole'scher „oder“
`&&`	Boole'scher „und“
`[]`	Die leere Liste (auch `Nil`)
`3::[4; 5; 7] = [3; 4; 5; 7]`	Am Anfang einer Liste einfügen.
`3::4::5::7::[] = [3; 4; 5; 7]`	Äquivalente Darstellungen einer Liste.
`fun x -> x + 1;;`	Die anonyme Funktion $x \mapsto x+1$.
`let f x = x + 1;;`	Eine Funtion definieren.
`let f = fun x -> x + 1;;`	Die gleiche Funktion wie oben definieren.

`let rec fact n =`
`  if n<=0 then 1`
`  else n * factorial (n-1);;`
	Eine rekursive Funktion definieren.

`f 2;;`	Funktionsaufruf der Funktion `f` mit dem Wert `2`.
`f ();;`	Funktionsaufruf def Funktion `f` ohne Parametern.

`let rec last list =`
`  match list with`
`  [] -> []`
`  | [a] -> [a]`
`  | first::rest -> last (rest);;`
	Rekursives pattern matching auf eine Liste

Manche der OCaml Funktionen, die Harrison benutzt

Um Funktionendefinitionen in dem Code unter Linux zu suchen, kann man im Terminal zu den Ordner wo Harrisons Code gespeichert ist[1] und da mit `grep "NAME" *.ml` bekommt man eine Liste alle Dateien, die die Funktion mit Name NAME beinhalten. Um den Definition eine Funktion `bla` zu finden, könnte man `grep "let bla" *.ml` oder `grep "let rec bla" *.ml` eingeben.

Drucken

```
# print_int (6 + 7) ;;
13- : unit = ()
```

```
# print_string "Hello\n";;
Hello
- : unit = ()
```

Die Bibliothek nums - Rationale Zahlen

```
# load "nums.cma";;
```

```
# Int 3 +/ Int 1 // Int 7;;
- : num = 22/7
```

```
# Int 3 // Int 4 </ num\_of\_string "12345";;
- : bool = true
```

```
gcd_num (Int 2) (Int 46) = (Int 2)
lcm_num (Int 2) (Int 5) = (Int 10)
2--8 = [2; 3; 4; 5; 6; 7; 8]
(Int 2)---(Int 4) = [(Int 2); (Int 3); (Int 4)]
```

Allgemeine Funktionen

```
let ( ** ) = fun f g x -> f(g x);;
let can f x = try f x; true with Failure _ -> false;;
let rec first n p = if p(n) then n else first (n +/ Int 1) p;;
let non p x = not(p x);;
let rec repeat f x = try repeat f (f x) with Failure _ -> x;;
let undef x = failwith "undefined function";;
let valmod a y f x = if x = a then y else f(x);;
```

[1] Den Code von Harrisons Buch kann man hier herunterladen: `http://www.cl.cam.ac.uk/~jrh13/atp/OCaml.tar.gz`.

Dienstfunktionen für Listen

```
butlast [1;2;3;4] = [1;2;3]
chop_list 3 [1;2;3;4;5] = ([1;2;3],[4;5])
do_list f [1;2;3] = (f 1; f 2; f 3)
el 2 [0;1;2;3] = 2
end_itlist f [1;2;3;4] = f 1 (f 2 (f 3 4))
exists p [1;2;3] = (p 1) or (p 2) or (p 3)
explode "hello" = ["h";"e";"l";"l";"o"]
implode ["w";"x";"y";"z"] = "wxyz"
forall p [1; 2; 3] = (p 1) & (p 2) & (p 3)
forall2 p [1; 2; 3] [a;b;c] = (p 1 a) & (p 2 b) & (p 3 c)
hd [1; 2; 3] = 1
tl [1;2;3;4] = [2;3;4]
insertat 3 9 [0;1;2;3;4;5] = [0;1;2;9;3;4;5]
itlist f [1;2;3] x = f 1 (f 2 (f 3 x))
itlist2 f [a;b;c] [1;2;3] x = f a 1 (f b 2 (f c 3 x))
last [1;2;3;4] = 4
length [1;2;3] = 3
map f [1;2;3] = [f 1; f 2; f 3]
map2 f [a;b;c] [1;2;3] = [f a 1; f b 2; f c 3]
mapfilter f [1;2;3] = [f 1; f 2; f 3]
replicate 4 9 = [9;9;9;9]
rev [1;2;3;4] = [4;3;2;1]
unzip [(1,a);(2,b);(3,c)] = ([1;2;3],[a;b;c])
zip [1;2;3] [a; b; c] = [(1,a); (2,b); (3,c)]
filter (fun x -> x = 3 || x = 1) [1; 2; 3; 4; 3; 2] = [1; 3; 3]
partition (fun x -> x = 3 || x = 1) [1; 2; 3; 4; 3; 2] = ([1; 3; 3], [2; 4; 2])
find (fun x -> x = 3 || x = 1) [1; 2; 3; 4; 3; 2] = 1
tryfind (fun x -> if x = 3 then x * 2 else failwith "not 3") [1; 2; 3; 4; 3; 2] = 6
tryfind (fun x -> if x = 3 then x * 2 else failwith "not 3") [1; 2] = Exception: Failure "tryfind".
index 4 [1; 2; 3; 4; 5; 6] = 3
earlier [1; 2; 3; 4; 5; 6] 6 4 = false
maximize (fun x -> - x * x) [-2; -1; 0 ; 1; 2] = 0
minimize (fun x -> - x * x) [-2; -1; 0 ; 1; 2] = 2
sort (<) [3;1;4;1;5;9;2;6;5;3;5] = [1; 1; 2; 3; 3; 4; 5; 5; 5; 6; 9]
uniq(sort (<) [3;1;4;1;5;9;2;6;5;3;5]) = [1; 2; 3; 4; 5; 6; 9]
sort (increasing length) [[1]; [1;2;3]; []; [3; 4]] = [[]; [1]; [3; 4]; [1; 2; 3]]
sort (decreasing length) [[1]; [1;2;3]; []; [3; 4]] = [[]; [1]; [3; 4]; [1; 2; 3]]
```

Endliche Mengen Alle endliche Mengen sind bei uns standardgeordnete Listen ohne Duplikate.

```
setify [4; 4; 6; 1; 9; 0] = [0; 1; 4; 6; 9]
union [1; 2] [2; 1; 3] = [1; 2; 3]
intersect [1; 2] [2; 1; 3] = [1; 2]
subtract [1; 2] [2; 1; 3] =  [ ]
subset [1; 2] [2; 1; 3] = true
psubset [1; 2] [2; 1] = false
set_eq [1; 2] [2; 1] = true
insert 5 [1; 2] = [1; 2; 5]
mem 4 [1; 2; 4; 6] = true
unions [[1;2;3]; [4; 8; 12]; [3;6;9;12]; [1]] = [1; 2; 3; 4; 6; 8; 9; 12]
image (fun x -> x mod 2) [1;2;3;4;5] = [0; 1]
allsubsets [1;2;3] = [[]; [1]; [1; 2]; [1; 2; 3]; [1; 3]; [2]; [2; 3]; [3]]
allnonemptysubsets [1;2;3] = [[1]; [1; 2]; [1; 2; 3]; [1; 3]; [2]; [2; 3]; [3]]
allsets 2 [1;2;3] = [[1; 2]; [1; 3]; [2; 3]]
allpairs (fun x y -> x * y) [2;3;5] [7;11] = [14; 22; 21; 33; 35; 55]
distinctpairs [1;2;3;4] = [(1, 2); (1, 3); (1, 4); (2, 3); (2, 4); (3, 4)]
```

Endliche Partielle Funktionen – finite partial functions

```
# let smallsqs = fpf [1;2;3] [1;4;9];;
val smallsqs : (int, int) func = <func>

# graph smallsqs;;
- : (int * int) list = [(1, 1); (2, 4); (3, 9)]

# graph (undefine 2 smallsqs);;
- : (int * int) list = [(1, 1); (3, 9)]

# graph ((3 |-> 0) smallsqs);;
- : (int * int) list = [(1, 1); (2, 4); (3, 0)]

# apply smallsqs 3;;
- : int = 9
```

Dazu: `( |=> )` `tryapply` `dom` `ran` `undefined`