

# Functional Programming

WS 2012/13

Harald Zankl (VO)

Cezary Kaliszyk (PS) Thomas Sternagel (PS)

Computational Logic  
Institute of Computer Science  
University of Innsbruck

week 1



# Lecture

## Facts

- ▶ LV-Nr. 703024
- ▶ VO 2
- ▶ <http://cl-informatik.uibk.ac.at/teaching/ws12/fp/>
- ▶ lecture notes are available online (only `.uibk.ac.at`)
- ▶ lecture notes are available in Studia
- ▶ office hours (HZ): Thursday 14:15-15:45 in 3M12
- ▶ `evaluation`: written exam (closed book)

# Exercises

## Facts

- ▶ LV-Nr. 703025
- ▶ PS 1
- ▶ three groups:

group 1	Friday 10:15–11:00	HS 11
group 2	Friday 10:15–11:00	HS 10
group 3	Friday 11:15–12:00	HS 10
- ▶ office hours:

CK	Thursday 09:30 – 11:00	in 3M12
TS	Wednesday 13:00 – 14:30	in 3M03
- ▶ online registration **required**
- ▶ evaluation: **weekly exercises + performance at blackboard**
- ▶ **exercises start on October 12**

# Schedule

## Slots

week 1	October	5	week 8	December	7
week 2	October	12	week 9	December	14
week 3	October	19	week 10	January	11
week 4	November	9	week 11	January	18
week 5	November	16	week 12	January	25
week 6	November	23	exam 1	February	1
week 7	November	30	exam 2	March	1

# Covered Topics

## Part I: Practice

lists, strings,  
trees, sets,  
parsing, efficiency,  
lazy lists, monads,  
...

## Part II: Theory

$\lambda$ -calculus,  
induction,  
type checking,  
type inference,  
...

# This Week

## Practice I

OCaml introduction, lists, strings, trees

## Theory I

lambda-calculus, evaluation strategies, induction, reasoning about functional programs

## Practice II

efficiency, tail-recursion, combinator-parsing

## Theory II

type checking, type inference

## Advanced Topics

lazy evaluation, infinite data structures, monads, . . .

## Notions - Side-Effects

### Definition

A function has **side-effects** if it modifies some state in addition to producing a value.

### Example (side-effect.c)

```
int calls = 0; // state

int power2(int i) {
  calls++; // side-effect
  printf("Call %i to 'power2'.\n", calls); // side-effect
  return(i * i); // actual result
}
```

# Notions - Purity

## Definition

A function is **pure** if it has same output on same input.

## Example (pure.c - Pure)

```
int suc(int i) { return(i + 1); }
```

## Example (pure.c - Impure)

```
int rnd(int m, int n) { // random number in  $\{i \mid -n < i < n\}$   
  return(m + random() % n);  
}
```



# Notions - Mutable Data

## Definition

**Mutable** data can be modified after its initial construction.

## Example (`mutable_string.c` - Mutable strings)

```
char* uppercase(char* s) {
    int i = 0;
    while (s[i] != '\0') s[i] = toupper(s[i++]);
    return s;
}
```

## Example (`ImmutableString.java` - Immutable strings)

```
public static String uppercase(String s) {
    return s.toUpperCase();
}
```

# Notions - Recursion

Definition (Recursion)

Definition (Recursion)

Definition (Recursion)

Definition (Recursion)

Definition (Recursion)

...

see

see

see

see

see

# Notions - Recursion

## Definition

A function is **recursive** if it is used in its own definition.

## Example (factorial.c - Factorial Numbers)

```
int factorial(int n) {  
    if (n < 2) { return 1; } else { return(n * factorial(n - 1)); }  
}
```

## Example (fib.c - Fibonacci Numbers)

```
int fib(int n) {  
    if (n < 2) { return n; } else { return(fib(n-1) + fib(n-2)); }  
}
```

## Notions - Strict vs. Lazy

- ▶ Strict:  $\text{double}(3+3) = \text{double}(6) = 6+6 = 12$
- ▶ Lazy:  $\text{double}(3+3) = (3+3)+(3+3) = 6+(3+3) = 6+6 = 12$

### Remark

Strict evaluation is similar to **call-by-value**

Lazy evaluation is similar to **call-by-name**

## Some Functional Languages (alphabetical)

### Curry (1996)

pure, lazy, logic-programming,  
'Haskell B. Curry'

### Erlang (1987)

concurrent, strict, 'A. K. Erlang'

### F# (2005)

object-oriented, strict

### Haskell (1990)

pure, lazy, 'Haskell B. Curry'

### Lisp (1958)

'List processing language', Scheme

### Mathematica (1988)

computer algebra

### ML (1973)

'metalanguage', StandardML,  
OCaml

### Scala v2.0 (2006)

'scalable language', strict, lazy,  
object-oriented, concurrent, JVM

# Benefits of Functional Languages

- ▶ concurrency for free (lack of side-effects)
- ▶ garbage collection (Lisp)
- ▶ close to mathematics (proving properties)
- ▶ compact code (maintainance, readability)

# Basic Types

- ▶ bool (e.g., `true`, `false`)
- ▶ char (e.g., `'a'`, `'b'`, `'c'`, ..., `'A'`, `'B'`, `'C'`, ..., `'0'`, `'1'`, `'2'`, ...)
- ▶ float (e.g., `0.`, `1e-3`, `3.1415`, ...)
- ▶ int (e.g., ..., `-2`, `-1`, `0`, `1`, `2`, ...)
- ▶ string (e.g., `"Hello, \u25b6world!\n"`)
- ▶ unit (e.g., `()`)

# Basic Operations

## Comparison

- ▶ '=' equality test
- ▶ '<>' inequality test
- ▶ '<' smaller than
- ▶ '>' greater than
- ▶ '<=' smaller than or equal
- ▶ '>=' greater than or equal
- ▶ 'compare' comparison
- ▶ 'min' minimum of 2 values
- ▶ 'max' maximum of 2 values

## Example

```
# 'c' <> 'h';;  
- : bool = true  
# compare "A" "A";;  
- : int = 0  
# compare "A" "B";;  
- : int = -1  
# compare "B" "A";;  
- : int = 1  
# max 1 2;;  
- : int = 2  
# min 1 2;;  
- : int = 1
```



# Basic Operations (cont'd)

## Booleans

- ▶ `'&&'` logical and
- ▶ `'||'` logical or
- ▶ `'not'` logical not

## Note

$A \ \&\& \ B$  ( $A \ || \ B$ ): if  $A$  is **false** (**true**) then  $B$  is not evaluated

# Basic Operations (cont'd)

## Integers

- ▶ `'~-'` unary negation
- ▶ `'succ'` successor function  
( $x \mapsto x + 1$ )
- ▶ `'pred'` predecessor function  
( $x \mapsto x - 1$ )
- ▶ `'+'` addition
- ▶ `'-'` subtraction
- ▶ `'*'` multiplication
- ▶ `'/'` division
- ▶ `'mod'` remainder of division
- ▶ `'abs'` absolute value
- ▶ `'max_int'` greatest representable integer
- ▶ `'min_int'` smallest representable integer

# Basic Operations (cont'd)

## Floating Point Numbers

- ▶ `'~-'` unary negation
- ▶ `'+'` addition
- ▶ `'-'` subtraction
- ▶ `'*'` multiplication
- ▶ `'/'` division
- ▶ `'**'` exponentiation
- ▶ `'sqrt'` square root
- ▶ `'truncate'` drop decimal places
- ▶ ...

# Basic Operations (cont'd)

## Strings

- ▶ `'^'` string concatenation

## Example

```
# "Hello" ^ ", world!";;  
- : string = "Hello, world!"
```

# Types

- ▶ basic types (`bool`, `char`, `float`, `int`, `string`, `unit`)
- ▶ type variables (`'a`, `'b`, `'c`, ...)
- ▶ tuple types (`int * float`, `'a * 'a`, `int * char * int`, ...)
- ▶ function types (`int -> int`, `bool -> bool -> bool`, ...)
- ▶ **user-defined types**

# User-Defined Types

## Type Abbreviations (new name for existing type)

- ▶ `type coord = int * int`

## Algebraic Datatypes (Variant Types)

- ▶ `type nat = Zero | Succ of nat`

- ▶ `type direction = North | East | South | West`

- ▶ `type number = Int of int | Float of float`

- ▶ `type 'a mylist = Nil | Cons of 'a * 'a mylist`

# Values (Instances of Types)

- ▶ tuples `((1, 2) : int * int)`
- ▶ anonymous functions `(fun x -> x + 1 : int -> int)`
- ▶ functions `(let succ x = x + 1 : int -> int)`
- ▶ variants (instances of algebraic datatypes;
  - ▶ `Zero : nat`
  - ▶ `Succ(Succ(Succ(Zero))) : nat`
  - ▶ `East : direction`
  - ▶ `Int 3 : number`
  - ▶ `Float 3.0 : number`
  - ▶ `Cons(3,Cons(5,Cons(7,Nil))) : int mylist`
  - ▶ `Cons('c',Cons('e', Nil)) : char mylist`
  - ▶ `Nil: 'a mylist)`

# Recursive Functions

- ▶ functions calling themselves

- ▶ e.g.,

```
let rec sum n = if n < 1 then 0 else n + sum(n-1)
```

## Example

```
sum 3
```

```
= if 3 < 1 then 0 else 3 + sum(3-1)
```

```
= 3 + sum 2
```

```
= 3 + if 2 < 1 then 0 else 2 + sum(2-1)
```

```
= 3 + 2 + sum 1
```

```
= ...
```

```
= 3 + 2 + 1 + 0
```

```
= 6
```



# Pattern Matching

- ▶ e.g.,

```
let rec map(f,ls) = match ls with
| Nil          -> Nil
| Cons(x,xs)  -> Cons(f(x),map(f,xs))
```

- ▶ pattern

$$p ::= x \mid c \mid C(p, \dots, p) \mid p \text{ as } x \mid (p) \mid p \mid p$$

## Example

```
map (succ,Cons(1,Cons(2,Nil))) = Cons(2,Cons(3,Nil))
```

# Currying

► function

```
let rec map(f,ls) = match ls with
| Nil          -> Nil
| Cons(x,xs)  -> Cons(f(x),map(f,xs))
```

has type  $('a \rightarrow 'b) * 'a \text{ mylist} \rightarrow 'b \text{ mylist}$

► compare to

```
let rec map f ls = match ls with
| Nil          -> Nil
| Cons(x,xs)  -> Cons(f x,map f xs)
```

of type  $('a \rightarrow 'b) \rightarrow 'a \text{ mylist} \rightarrow 'b \text{ mylist}$

## Currying (cont'd)

- ▶ every function has just **one** argument
- ▶ how to define functions with more arguments (e.g.,  $x + y$ )?
- ▶ either use tuples (`let add(x,y) = x + y`)
- ▶ or curried (`let add = (fun x -> (fun y -> x + y))`)
- ▶ curried form is OCaml standard (e.g., `let f x y z = b` equals `let f = (fun x -> (fun y -> (fun z -> b)))`)

# Scoping

- ▶ values cannot be changed
- ▶ consider

```
let w = 1
```

```
let x =  
  let y = w in  
  let w = 2 in  
  let z = w in  
  y + z
```

finally `x` has value `3` and `w` (still) has value `1`