CARL VON OSSIETZKY universität OLDENBURG

Sommersemester 2020

Objektorientierte Modellierung und Programmierung

Dr. Christian Schönberg

CARL VON OSSIETZKY universität OLDENBURG

Funktionale Programmierung

CARL VON OSSIETZKY universität OLDENBURG

Lernziele

- Standard ML
  - Funktionen
  - Datentypen
  - Funktionen höherer Ordnung
- Funktionale Konzepte in Java

CARL VON OSSIETZKY universität OLDENBURG

Programmierparadigmen

- Imperative Programmierung
- Objektorientierte Programmierung
- Aspektorientierte Programmierung
- Agentenorientierte Programmierung
- Funktionale Programmierung
- Logikorientierte Programmierung
- Regelbasierte Programmierung
- …

CARL VON OSSIETZKY universität OLDENBURG

Funktionale Programmierung

- Basiert auf Funktionen
- Keine Schleifen und ähnliche Kontrollstrukturen
- Dafür Funktionsaufrufe, Verkettung von Funktionen und Rekursion
- Genauso mächtig wie imperative Programmierung oder OOP
- Zustandslos, frei von Seiteneffekten
- Automatische Inferenz von Typen/Signaturen
- Viele funktionale Sprachen
  - SML, Lisp, Haskell, Mathematica, Erlang, F#, …
- Viele Sprachen mit funktionalen Konstrukten
  - Java, JavaScript, C++, C#, XQuery, XSLT, Scala, …

CARL VON OSSIETZKY universität OLDENBURG

Standard ML

Standard ML

- Standard Meta Language, SML
- Standardisiert 1997
- Formale Spezifikation (Typregeln und Ausführungssemantik)
  → Beweisbarkeit
- Funktionen sind *First-Class Citizens*
- Einige imperative Anleihen
- Verschiedene Implementierungen
  - Standard ML of New Jersey (SML/NJ)
    - Referenzimplementierung
    - https://www.smlnj.org/
  - …

Funktionen und Werte: Beispiel

$$factorial(n) = \begin{cases} 1, & \text{falls } n = 0 \\ n \cdot factorial(n-1), & \text{sonst} \end{cases}$$

```
fun factorial(0) = 1
  | factorial(n) = n * factorial(n - 1);
```

pattern matching

```
val factorial = fn : int -> int
```

automatische Typinferenz

```
val fact5 = factorial(5);
```

```
val fact5 = 120 : int
```

```
factorial(3);
```

```
val it = 6 : int
```

```
fun factorial(n) =
      if n = 0
      then 1
      else n * factorial(n-1);
```

CARL VON OSSIETZKY universität OLDENBURG

Funktionen: Beispiel (2)

```
fun plus(a, b) = a + b;
```

```
val plus = fn : int * int -> int
```

```
fun selFirst(x, y) = x;
```

```
val selFirst = fn : 'a * 'b -> 'a
```

```
fun swap(a, b) = (b, a);
```

```
val swap = fn : 'a * 'b -> 'b * 'a
```

CARL VON OSSIETZKY universität OLDENBURG

Polymorphie

Allgemeine Type wie **'a** erlauben Polymorphie:

```
selFirst(3, 5);
```

```
val it = 3 : int
```

```
selFirst(true, false);
```

```
val it = true : bool
```

```
selFirst("a", 7);
```

```
val it = "a" : string
```

```
swap(3.14, 9);
```

```
val it = (9, 3.14) : int * real
```

CARL VON OSSIETZKY universität OLDENBURG

Gültigkeit und Nebeneffekte: Beispiel

```
val const = 3;
val const = 3 : int
fun giveConst() = const;
val giveConst = fn : unit -> int
giveConst();
val it = 3 : int
const = 4;
val it = false : bool
val const = 4;
val const = 4 : int
giveConst();
val it = 3 : int
```

Neue Definitionen überdecken alte Definitionen.
Jede Definition verwendet die zum Zeitpunkt der Definition gültigen Definitionen.

CARL VON OSSIETZKY universität OLDENBURG

Basis-Datentypen: Beispiel

```
3;
```

```
val it = 3 : int
```

```
3.14;
```

```
val it = 3.14 : real
```

```
"abc";
```

```
val it = "abc" : string
```

```
#"x";
```

```
val it = #"x" : char
```

```
true;
```

```
val it = true : bool
```

CARL VON OSSIETZKY universität OLDENBURG

Standard-Operatoren und Typ-Konvertierung: Beispiel

```
(3 + 4) div 3 * ~2;
val it = ~4 : int
6.4 / 2.5;
val it = 2.56 : real
3 + 3.14;
Error: operator and operand do not agree [overload conflict]
real(3) + 3.14;
val it = 6.14 : real
"ab" ^ "cd";
val it = "abcd" : string
str(#"x") ^ "y";
val it = "xy" : string
(true orelse false) andalso not false;
val it = true : bool
((4 > 3) orelse (2 <= 1)) andalso not (5 <> 5);
val it = true : bool
```

CARL VON OSSIETZKY universität OLDENBURG

Tupel und Records: Beispiel

```
();
val it = () : unit
(1, 2, 3);
val it = (1,2,3) : int * int * int
(1, "a", false);
val it = (1,"a",false) : int * string * bool
((true, "b"), 3, 4, #"z");
val it = ((true,"b"),3,4,#"z") : (bool * string) * int * int * char
val omp = { name = "OMP", semester = 2019 };
val omp = {name="OMP",semester=2019} : {name:string, semester:int}
omp = { semester = 2019, name = "OMP" };
val it = true : bool
```

Peano-Axiome für die Natürlichen Zahlen

- Formalisierung der Natürlichen Zahlen $\mathbb{N}$ über fünf Axiome
  1. $0 \in \mathbb{N}$
  2. $\forall x \in \mathbb{N}: succ(x) \in \mathbb{N}$
  3. $\forall x \in \mathbb{N}: succ(x) \neq 0$
  4. $\forall x, y \in \mathbb{N}: succ(x) = succ(y) \Rightarrow x = y$
  5. $\forall S \supseteq \{ 0, succ(0), succ(succ(0)), \dots \}: \mathbb{N} \subseteq S$

CARL VON OSSIETZKY universität OLDENBURG

Natürliche Zahlen als Datentyp

```
(* natural numbers *)
datatype natural = Zero | Succ of natural;
```

```
fun add(x, Zero) = x
  | add(x, Succ y) = add(Succ x, y);
```

```
val add = fn : natural * natural -> natural
```

```
fun sub(x, Zero) = x
  | sub(Zero, y) = Zero
  | sub(Succ x, Succ y) = sub(x, y);

fun times(Zero, _) = Zero
  | times(_, Zero) = Zero
  | times(Succ x, y) = add(times(x, y), y);
```

CARL VON OSSIETZKY universität OLDENBURG

Natürliche Zahlen als Datentyp

```
fun eq(Zero, Zero) = true
  | eq(Zero, _) = false
  | eq(_, Zero) = false
  | eq(Succ x, Succ y) = eq(x, y);

fun gt(Succ _, Zero) = true
  | gt(Zero, _) = false
  | gt(Succ x, Succ y) = gt(x, y);
```

CARL VON OSSIETZKY universität OLDENBURG

Natürliche Zahlen als Datentyp

```
fun toInt(Zero) = 0
  | toInt(Succ x) = 1 + toInt(x);

fun toNat(0) = Zero
  | toNat(x) = if x < 0 then Zero else Succ (toNat(x-1));
```

CARL VON OSSIETZKY universität OLDENBURG

Natürliche Zahlen als Datentyp

```
add(Succ (Succ Zero), Succ Zero);
val it = Succ (Succ (Succ Zero)) : natural
sub(Succ (Succ Zero), Succ Zero);
val it = Succ Zero : natural
times(Succ (Succ Zero), Succ Zero);
val it = Succ (Succ Zero) : natural
times(Succ (Succ Zero), Succ (Succ Zero));
val it = Succ (Succ (Succ (Succ Zero))) : natural
eq(Succ (Succ Zero), Succ Zero);
val it = false : bool
eq(Succ (Succ Zero), Succ (Succ Zero));
val it = true : bool
gt(Succ (Succ Zero), Succ Zero);
val it = true : bool
toInt(Succ (Succ Zero));
val it = 2 : int
toNat(3);
val it = Succ (Succ (Succ Zero)) : natural
```

CARL VON OSSIETZKY universität OLDENBURG

Listen: Beispiel

```
[1,2,3,4];
val it = [1,2,3,4] : int list
[true,false,false];
val it = [true,false,false] : bool list
["a", "bc", "def", "gh"];
val it = ["a","bc","def","gh"] : string list
[(3, false), (7, true)];
val it = [(3,false),(7,true)] : (int * bool) list
[3, 9, 3.14, 2];
Error: operator and operand don't agree [literal]
```

CARL VON OSSIETZKY universität OLDENBURG

Listen: Beispiel (2)

```
1 :: [2, 3, 4];
```

```
val it = [1,2,3,4] : int list
```

```
1 :: 2 :: [3, 4];
```

```
val it = [1,2,3,4] : int list
```

```
[1, 2] @ [3, 4];
```

```
val it = [1,2,3,4] : int list
```

```
1::2::3::4::nil;
```

```
1::2::3::4::[];
```

```
val it = [1,2,3,4] : int list
```

```
[[1, 2], [3, 4]];
```

```
val it = [[1,2],[3,4]] : int list list
```

CARL VON OSSIETZKY universität OLDENBURG

Listen: Beispiel (3)

```
fun member(x, nil)    = false
  | member(x, y::ys) = x=y orelse member(x, ys);
```

```
val member = fn : ''a * ''a list -> bool
```

```
member(3, [1,2,3,4]);
```

```
val it = true : bool
```

```
member(7, [1,2,3,4]);
```

```
val it = false : bool
```

CARL VON OSSIETZKY universität OLDENBURG

Listen: Beispiel (4)

```
fun insert(x, nil) = [x]
  | insert(x, y::ys) = if x<y then x::y::ys
                               else y::insert(x, ys);
```

```
val insert = fn : int * int list -> int list
```

```
insert(3, [1,2,4,5]);
```

```
val it = [1,2,3,4,5] : int list
```

```
fun getLarger(l1,l2) =
      if length(l1) > length(l2) then l1 else l2;
```

```
val getLarger = fn : 'a list * 'a list -> 'a list
```

CARL VON OSSIETZKY universität OLDENBURG

Listen: Nützliche Funktionen

- **`null(l)`**: Liste **`l`** ist leer (**`nil`**)
- **`length(l)`**: Anzahl der Elemente von **`l`**
- **`hd(l)`**: Kopfelement von **`l`** (head)
- **`tl(l)`**: Alle bis auf das Kopfelement von **`l`** (tail)
- **`List.nth(l, i)`**: Das **`i`**-te Element von **`l`**
- **`map f l`**: Wendet die Funktion **`f`** auf jedes Element von **`l`** an

CARL VON OSSIETZKY universität OLDENBURG

Listen und Strings

```
concat(["list", " of ", "strings"]);
val it = "list of strings" : string
explode("hello");
val it = [#"h",#"e",#"l",#"l",#"o"] : char list
implode([#"h",#"e",#"l",#"l",#"o"]);
val it = "hello" : string
```

CARL VON OSSIETZKY universität OLDENBURG

Anonyme Funktionen

```
val factorial = fn : int -> int
```

- Auch bekannt als Lambda-Funktionen (λ)
- Eine Funktionsdefinition, die keinem Identifikator (Funktionsnamen) zugewiesen ist
- Funktionen in SML sind *First-Class Citizens*
  - Funktionen können Variablen zugewiesen werden oder in Datenstrukturen referenziert werden
  - Funktionen können Parameter oder Rückgabetyp von anderen Funktionen sein
- Vergleich: Methoden in Java sind **keine** *First-Class Citizens*

CARL VON OSSIETZKY universität OLDENBURG

Anonyme Funktionen: Beispiel

```
fun plus(a, b) = a + b;
```

```
val plus = fn : int * int -> int
```

```
val plus = fn (a, b) => a + b;
```

```
val plus = fn : int * int -> int
```

```
plus(3, 6);
```

```
val it = 9 : int
```

```
fun factorial(0) = 1
  | factorial(n) = n * factorial(n - 1);
```

```
val factorial = fn : int -> int
```

```
val factorial = fn 0 => 1
                 | n => n * factorial(n-1);
```

```
val factorial = fn : int -> int
```

CARL VON OSSIETZKY universität OLDENBURG

Listen: Beispiel (5)

`map f l`: Wendet die Funktion **`f`** auf jedes Element von **`l`** an

```
map (fn x => x+1) [1, 2, 3, 4];
```

```
val it = [2,3,4,5] : int list
```

CARL VON OSSIETZKY universität OLDENBURG

SML Funktionen Höherer Ordnung

```
fun apply(f, x) = f(x);
```

```
val apply = fn : ('a -> 'b) * 'a -> 'b
```

```
apply(factorial, 3);
```

```
val it = 6 : int
```

```
fun applyTwice(f, x) = f(f(x));
```

```
val applyTwice = fn : ('a -> 'a) * 'a -> 'a
```

CARL VON OSSIETZKY universität OLDENBURG

SML Funktionen Höherer Ordnung (2)

```
fun createAdder(x) = fn y => x + y;
```

```
val createAdder = fn : int -> int -> int
```

```
val plus3 = createAdder(3);
```

```
val plus3 = fn : int -> int
```

```
plus3(5);
```

```
val it = 8 : int
```

SML Funktionen Höherer Ordnung (2)

```
fun createAdder(x) = fn y => x + y;
```

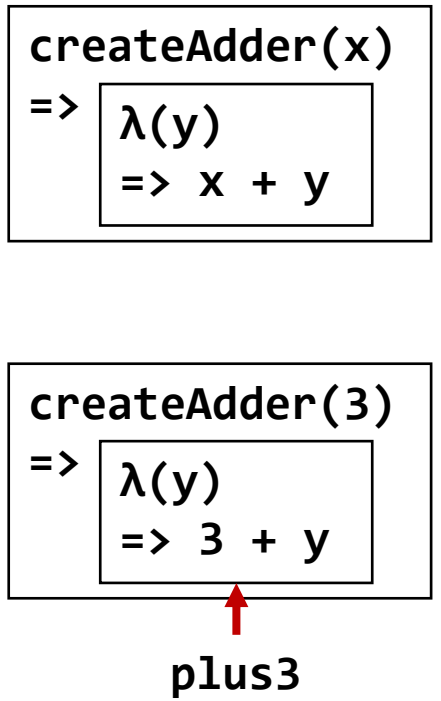
```
val createAdder = fn : int -> int -> int
```

```
val plus3 = createAdder(3);
```

```
val plus3 = fn : int -> int
```

```
plus3(5);
```

```
val it = 8 : int
```



SML Funktionen Höherer Ordnung (2)

```
fun createAdder(x) = fn y => x + y;
```

```
val createAdder = fn : int -> int -> int
```

```
val plus3 = createAdder(3);
```

```
val plus3 = fn : int -> int
```

```
plus3(5);
```

```
val it = 8 : int
```

```
createAdder(x)
=> λ(y)
   => x + y
```

```
createAdder(3)
=> plus3(y)
   => 3 + y
```

```
plus3(5)
=> 3 + 5
```

CARL VON OSSIETZKY universität OLDENBURG

SML Funktionen Höherer Ordnung (3)

```
fun high(x, f) = fn (y, g) => (f(x, y) + g(f(x, y)), y);
```

```
val high = fn : 'a * ('a * 'b -> int) -> 'b * (int -> int) -> int * 'b
```

CARL VON OSSIETZKY universität OLDENBURG

Currying

- Currying, auch Schönfinkelisation, ist die Umwandlung einer Funktion mit mehreren Argumenten in eine Folge von Funktionen mit jeweils einem Argument
- Vorteile
  - zweiter Parameter (und dritter, …) können später definiert werden
    → teilweise Auswertung der Funktion
  - in der theoretischen Informatik: Beweise über Funktionen
    → alle Funktionen haben die gleiche Form
    (ein Parameter)

CARL VON OSSIETZKY universität OLDENBURG

Currying: Beispiel

- f: $\mathbb{R}\times\mathbb{R}\times\mathbb{R} \rightarrow \mathbb{R}$
  - $f(x, y, z) = x + y + z$
- wird zu
- g: $\mathbb{R} \rightarrow (\mathbb{R} \rightarrow (\mathbb{R} \rightarrow \mathbb{R}))$
  - $g(x) = x => x + h$
  - h: $\mathbb{R} \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$
  - $h(y) = y => y + i$
  - i: $\mathbb{R} \rightarrow \mathbb{R}$
  - $i(z) = z$
  - $g(x)(y)(z) = x + h(y)(z) = x + y + i(z) = x + y + z$

CARL VON OSSIETZKY universität OLDENBURG

Currying: Beispiel (2)

- **`fun plus(x, y) = x + y`**
  - **`val plus = fn : int * int -> int`**
  - **`plus`** ist eine Funktion mit zwei Parametern
- **`fun cplus(x)(y) = x + y`**
  - **`val cplus = fn : int -> int -> int`**
  - **`cplus`** ist eine Funktion mit einem Parameter
  - **`cplus(x)`** ist eine Funktion mit einem Parameter
  - **`cplus`** ist die „curried“ Version von **`plus`**
- **`plus(3, 5) = 8`**
- **`cplus(3)(5) = `*`cplus3`*`(5) = 8`**
  - *`cplus3`*`(x) = x + 3`

CARL VON OSSIETZKY universität OLDENBURG

Currying: Beispiel (3)

```
fun addThree(x, y, z) = x + y + z;
```

```
val addThree = fn : int * int * int -> int
```

```
fun addThree(x) = fn (y) => fn (z) => x + y + z;
```

```
val addThree = fn : int -> int -> int -> int
```

CARL VON OSSIETZKY universität OLDENBURG

Currying: Beispiel (4)

```
fun plus(x, y) = x + y;
```

```
val plus = fn : int * int -> int
```

```
fun curry(f) = fn x => fn y => f(x,y);
```

```
val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

```
val cplus = curry(plus);
```

f x y f(x, y)

```
val cplus = fn : int -> int -> int
```

```
cplus(3)(5);
```

```
val it = 8 : int
```

```
fun uncurry f (x, y) = f x y;
```

```
val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

```
uncurry(curry(plus));
```

```
val it = fn : int * int -> int
```

CARL VON OSSIETZKY universität OLDENBURG

Funktionale Konzepte in Java

CARL VON OSSIETZKY universität OLDENBURG

Lambda-Ausdrücke (Wiederholung)

- Lambda-Ausdrücke sind keine anonymen Klassen, sondern anonyme Methoden
- Sie werden aber eingesetzt, um die Implementierung von funktionalen Interfaces durch einen einfachen, kompakten Ausdruck zu ersetzen
  - funktionale Interfaces sind Interfaces, die nur eine einzige nicht-statische und nicht-default Methode definieren
  - Instanzen von funktionalen Interfaces können direkt durch einen Lambda-Ausdruck ersetzt werden
  - dabei wird die Lambda-Funktion auf die eine Methode des Interfaces abgebildet
- Aufbau:

```
FunctionalInterface instance = (parameters) -> expression/block;
```

Lambdas in Java und SML

- Lambda-Ausdrücke in Java sind **keine** Methoden, sondern nur eine verkürzte Schreibweise für eine Klassendefinition
- Lambda-Funktionen in SML sind echte Funktionen

```
fun f(x) = fn y => x * y;
```

```
val f = fn : int -> int -> int
```

```
f(3)(4);
```

```
val it = 12 : int
```

```
public interface Function {
    int calc(int x);
}
```

```
public interface FunctionOfFunction {
    Function calc(int x);
}
```

```
FunctionOfFunction f = (x) -> (y) -> x * y;
System.out.println(f.calc(3).calc(4));
```

```
12
```

CARL VON OSSIETZKY universität OLDENBURG

Lernziele

- Standard ML
  - Funktionen
  - Datentypen
  - Funktionen höherer Ordnung
- Funktionale Konzepte in Java