JG|U

JOHANNES GUTENBERG
UNIVERSITÄT MAINZ

Programmiersprachen (08.079.030) 7 - Funktionale Programmierung mit SML

Tim Süß
Institut für Informatik
Johannes Gutenberg-Universität Mainz

Funktionale Programmierung

Themen

- Typen und Operationen in SML
- Funktionen und Kontrollstrukturen
- Pattern-Matching
- Rekursion
- End-Rekursion
- Datenstrukturen
- „Currying“ und Funktionen höherer Ordnung

Typen und Operationen in SML

Einfache Typen	Ergebnisse/Ausgaben
`1;`	`val it = 1 : int`
`1.2;`	`val it = 1.2 : real`
`-1;`	`ERROR`
`~1;`	`val it = ~1 : int`
`~1.2;`	`val it = ~1.2 : real`
`"s";`	`val it = "s" : = string`
`#"a";`	`val it = #"a" : char`
`true;`	`val it = true : bool`
`false;`	`val it = false :  bool`
`~~5;`	`ERROR`
`~ ~5;`	`val it = 5 : int`

Typen und Operationen in SML

```
Tupel                           Ergebnisse/Ausgaben

(1,2,3);                        val it = (1,2,3) : int * int * int
(1, 1.2, "test", true);         val it = (1, 1.2, "test", true) : int
                                * real * string * bool
();                             val it = () : unit
(1);                            val it = 1 : int
(());                           val it = () : unit

#1(1,2,3,4);                    val it = 1 : int
#3(3,4,5,6);                    val it = 5 : int
#0(1,2,3,4);                    ERROR
```

Typen und Operationen in SML

Records/Verbunde

```
{name = "tim", lectures = 20,
prof = true };






{1 = "tim", 2 = 42 };
{0 = "tim" };
{2 = "tim", 3 = 42 };




#name({name = "tim", lectures =
20, prof = true });
```

Ergebnisse/Ausgaben

```
val it =
{lectures=20,name="tim",prof=true}
  : {lectures:int, name:string,
prof:bool}
(* Sortierung alphabetisch (praktisch
bei Koordinaten) *)
val it = ("tim",42) : string * int
ERROR
val it = {2="tim",3=42} : {2:string,
3:int}



val it = "tim" : string
```

Typen und Operationen in SML

Listen	Ergebnisse/Ausgaben
`[1,2,3];`	`val it = [1,2,3] : int list`
`1 :: 2 :: 3 :: nil;`	`val it = [1,2,3] : int list`

`[];`	`val it = [] : 'a  = 'a list`
	`(*alpha list; alpha definiert nach erstem einfügen*)`

`[(0, false), (1, true)];`	`val it = [(1,true),(0,false)] : (int * bool) list`
`hd([1,2,3]);`	`val it = 1 : int`
`hd [1,2,3];`	`val it = 1 : int`
`tl([1,2,3]);`	`val it = [2,3] : int list`
`tl [1,2,3];`	`val it = [2,3] : int list`

Typen und Operationen in SML

Operation	Ergebnisse/Ausgaben
Zuweisung	

`val x = 42;`	`val x = 42 : int`
	`(* Zuweisung nur bei Verwendung von val, sonst vergleich *)`

`val tmp = (3 = 5);`	`val tmp = false : bool`

Typen und Operationen in SML

Operation	Ergebnisse/Ausgaben
1 + 1;	val it = 2 : int
1.2 + 1.3;	val it = 2.5 : real
1 + 1.2;	ERROR
10 / 4;	ERROR
10 div 4;	val it = 2 : int
10 mod 3;	val it = 1 : int
3 = 3;	val it = true : bool
2 = 3;	val it = false :bool
2 >= 3;	val it = false : bool
2 <> 3;	val it = true : bool (*ungleich*)
Int.abs(~1);	val it = 1 : int
Int.min(1,2);	val it = 1 : int
Int.sign(~277364);	val it = ~1 : int
Int.min;	val it = fn : int * int -> int
Int.toString(2);	val it = "2" : string

Typen und Operationen in SML

Operation	Ergebnisse/Ausgaben
`1.0 + 1.0;`	`val it = 2.0 : real`
`10.0 / 2.0;`	`val it = 5.0 : real`
`2.0 > 1.0;`	`val it = true : bool`
`2.0 <> 1.0;`	`ERROR (*real ist kein equality type*)`
`Real.!=(2.0, 1.0);`	`val it = true : bool`
`Real.==(2.0, 1.0);`	`val it = false : bool`
`Real.floor;`	`val it = fn : real -> int`
`Real.Round(1.5);`	`val it = 1 : int`
`Real.Round(1.50001);`	`val it = 2 : int`
`10.0 / 4;`	`ERROR`
`10.0 / real(4);`	`val it = 2.5 : real`
`Math.pi;`	`val it = 3.14159 : real`
`Math.sqrt(25.0);`	`val it = 5.0 : real`

Typen und Operationen in SML

Operation	Ergebnisse/Ausgaben
"hallo" + "welt"	ERROR
"hello" ^ "welt"	val it = "hellowelt" : string
"welt" = "welt"	val it = true : bool
"hello" < "welt"	val it = true : bool
String.size("hello");	val it = 5 : int
String.sub("hello", 1);	val it = #"e" : char
String.sub("hello world", 0, 5);	val it = "hello" : string
#"a" = #"a";	val it = true : bool
Char.ord(#"A");	val it = 65 : int
Char.chr(65);	Val it = #"A" : char

Funktionen und Kontrollstrukturen

Funktion	Ergebnisse/Ausgaben
val myfunc = Int.min;	val myfunc = fn : int * int -> int
myfunc(2,1);	val it = 1 : int
val mul(a, b) fn => a * b;	val mul = fn : int*int -> int
fun mul(a, b) = a * b;	val mul = fn : int*int -> int
fun mulReal(a:real, b:real) = a * b;	val mulReal = fn : real*real -> real
fun mul(a:real, b) = a * b;	val mul = fn : real*real -> real
fun mulMixed(a, b) = real(a) * b;	val mulMixed = fn : int*real -> real
if 4 mod 2 = 0 then "even" else "odd";	val it = "even" : string
val iseven = fn(number) => if number mod 2 = 0 then true else false;	val iseven = fn : int -> bool
val iseven = fn(number) => number mod 2 = 0;	iseven = fn : int -> bool
val isodd = fn(number) => not(iseven(number));	val isodd = fn : int -> bool
if true then "hello" else 42;	ERROR
use "dateiname.sml";	[opening dateiname.sml]

Pattern-Matching

Funktion mit Pattern

```
val test =
   fn (42) =>
        "sinn von allen, dem
        universum, dem leben und
        dem ganzen rest"
   | (23) =>
        "ich hab angst ..."
   | (_) =>
        "mir doch egal";
```

Ergebnisse/Ausgaben

```
val test = fn : int -> string
```

Rekursion

Rekursion

Ergebnisse/Ausgaben

```
val rec stringTimesN =
  fn (str, n) =>
    if n = 1
    then str
    else str ^ stringTimesN(str, n-1);     val stringTimesN = fn :
                                           string * int -> string

fun stringTimesN (str, n) =
    if n = 1
    then str
    else str ^ stringTimesN (str, n-1);    val stringTimesN = fn :
                                           string * int -> string
```

Rekursion

Rekursionsauflösung

```
val rec sum =
  fn (n) =>
    if n = 0
    then 0
    else n + sum(n-1);
```

```
sum(3);
if n = 1 then 1 else n + sum(n-1);
if 3 = 1 then 1 else 3 + sum(3-1);
if false then 1 else 3 + sum(3-1);
                    (3 + sum(2));
                    (3 + sum(if n = 1 then 1 else n + sum(n-1)));
                    (3 + sum(if 2 = 1 then 1 else 2 + sum(2-1)));
                    (3 + sum(if false then 1 else 2 + sum(2-1)));
                    (3 + sum(                     2 + sum(1)));
                    (3 + sum(                     2 + (if n = 1 then 1 else n + sum(n-1))));
                    (3 + sum(                     2 + (if 1 = 1 then 1 else n + sum(1-1))));
                    (3 + sum(                     2 + (if true  then 1 else n + sum(1-1))));
                    (3 + sum(                     2 + (              1)));
                    (3 + (3))
                    (6)
                     6
```

End-Rekursion

Rekursion

```
val fac = fn (n) =>
   let
      val rec facEnd = fn (n, accu) =>
         if n = 1
         then accu
         else facEnd(n-1, accu * n)
   in
      facEnd(n, 1)
   end;
```

Ergebnisse/Ausgaben

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

End-Rekursion

End-Rekursionsauflösung

```
val sum = fn (n) =>
   let
      val rec sumER = fn (n, accu) =>
         if n = 0
         then accu
         else sumER(n-1, accu+n)
   in
      sumER(n, 0)
   end;
sum(2);
```

```
SumER(2, 0)
if n = 1 then accu else sumER(n-1, accu+n);
if 2 = 1 then 0    else sumER(2-1, 0   +2);
if false then 0    else sumER(2-1, 0   +2);
                        sumER(1,   2);
                        if n = 0 then accu else sumER(n-1, accu+n);
                        if 1 = 0 then 2    else sumER(1-1, 2   +1);
                        if false then 2    else sumER(1-1, 2   +1);
                                                sumER(0, 3);
                                                if n = 0 then accu else sumER(n-1, accu+n);
                                                if 0 = 0 then accu else sumER(0-1, 3+0);
                                                if true  then accu else sumER(0-1, 3+0);
                                                              3
```

Datenstrukturen

Datenstruktur

```
datatype baum = blatt
      | ast of baum*int*baum;
val einBaum = ast(blatt, 3, blatt);
```

Funktionen

```
fun insert (i, blatt) =
  branch(blatt, i, blatt)
  | insert(i, stamm as ast (r, i, l)) =
    case Int.compare (i,j) of
    EQUAL => stamm
  | LESS => ast (insert (i, r), j, l)
  | GREATER => ast(l, j, insert(i, r));
val neuerBaum = insert(5, einBaum);
fun find(i, blatt) = false
  | find(i, ast(l, j, r)) =
    case Int.compare(i,j) of
    EQUAL => true
  | LESS => find (i, l)
  | GREATER => find(i, r));
find(3, neuerBaum);
```

Ergebnisse/Ausgaben

```
datatype tree =
  branch of tree * int * tree
  | leaf
```

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

```
val find = fn : int * tree ->
bool
```

Currying und Funktionen höherer Ordnung

Funktionen

```
(* standard *)
val add = fn(a, b) => a + b;
add(1, 2);

(* currying *)
val add = fn(a) => fn(b) => a+b;
add(1)(2);

add(1);
val addTwoTo = add(2);
addTwoTo(7);

val addFourTo = add(4);
addFourTo(7);

(* Funktion höherer Ordnung *)
val add = fn(a) => fn(b) => a+b;
```

Ergebnisse/Ausgaben

```

val add = fn : int * int -> int
val it = 3 : int


val add = fn : int -> int -> int
val it = 3 : int

val it = fn : int -> int
val addTwoTo = fn : int -> int
val it = 9 : int

val addFourTo = fn : int -> int
val it = 11 : int


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

Currying und Funktionen höherer Ordnung

Funktionen

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

val addThree = fn (a) => fn(b, c) =>
a+b+c;
val addThree = fn (a) => fn(b) =>
fn(c) => a+b+c;

fun addThree(a, b, c) = a + b + c;

fun addThree(a)(b)(c) = a + b + c;

fun addThree a b c = a + b + c;

(* Funktion höherer Ordnung *)
val add = fn(a) => fn(b) => a+b;
```

Ergebnisse/Ausgaben

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

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


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

Currying und Funktionen höherer Ordnung

Funktionen

```
val rec incrementAllInList =
  fn (nil) =>
    nil
  | (head :: tail) =>
    (head + 1) :: incrementAllInList(tail);

val rec squareAllInList =
  fn (nil) =>
    nil
  | (head :: tail) =>
    (head * head) :: squareAllInList(tail);

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

Ergebnisse/Ausgaben (kurz)

```
val incrementAllInList =
fn : int list -> int
list

val squareAllInList = fn
: int list -> int list

(*[2,3,4,5,6]*)
(*[1,4,9,16,25]*)
```

Currying und Funktionen höherer Ordnung

Funktionen

```
val rec incrementAllInList =
  fn (nil) =>
    nil
  | (head :: tail) =>
    (head + 1) :: incrementAllInList(tail);

val rec squareAllInList =
  fn (nil) =>
    nil
  | (head :: tail) =>
    (head * head) :: squareAllInList(tail);

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

Ergebnisse/Ausgaben (kurz)

```
val incrementAllInList =
fn : int list -> int
list

val squareAllInList = fn
: int list -> int list

(*[2,3,4,5,6]*)
(*[1,4,9,16,25]*)
```

Currying und Funktionen höherer Ordnung

Funktionen   Ergebnisse/Ausgaben (kurz)

```
val increment = fn(n) => (n + 1);                  int -> int
val square    = fn(n) => (n * n);                  int -> int
val isEven    = fn(n) => n mod 2 = 0;              int -> bool

val rec incrementAllInList =
  fn (nil) =>
    nil
  | (head :: tail) =>
    increment(head) :: incrementAllInList(tail);

val rec squareAllInList =
  fn (nil) =>
    nil
  | (head :: tail) =>
    square(head) :: squareAllInList(tail);
```

Currying und Funktionen höherer Ordnung

Funktionen — Ergebnisse/Ausgaben (kurz)

```
val increment = fn(n) => (n + 1);                int -> int
val square    = fn(n) => (n * n);                int -> int
val isEven    = fn(n) => n mod 2 = 0;            int -> bool

val rec applyFuncOnList =
 fn (func) =>
 fn (nil) =>
  nil                                            val applyFuncOnList =
 | (head :: tail) =>                             fn  : ('a -> 'b) ->
  func(head) :: applyFuncOnList(func)(tail);     'a list -> 'b list

applyFuncOnList(increment) ([1,2,3,4,5]);        val it = [2,3,4,5,6] :
                                                 int list
applyFunOnList(isEven) ([1,2,3,4,5]);            val it = [false,true,
                                                 false,true,false] :
                                                 bool list
val isOdd = applyFuncOnList(fn (n)=> n mod 2=1);
```

Currying und Funktionen höherer Ordnung

Funktionen   Ergebnisse/Ausgaben (kurz)

```
val listSum =
    fn(list) =>
    let
        val rec helper =
            fn(nil, accu) =>
               accu
            | (head :: tail, accu) =>
               helper(tail, accu + head)
    in
        helper(list, 0);
    end;                                         int list -> int

listSum ([1,2,3,4,5]);                           val it = 15 : int

foldr(op+) (0) ([1,2,3,4,5]);                    val it = 15 : int
(* op+; val it = fn : int * int -> int  *)
```

Currying und Funktionen höherer Ordnung

Funktionen	Ergebnisse/Ausgaben (kurz)
`Foldr;`	`val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b`
`foldr(op^) ("") (["a", "b", "c"]);`	`val it = "abc" : string`
`foldl(op^) ("") (["a", "b", "c"]);`	`val it = "cba" : string`

`foldr(op* ) (1) ([1,2,3,4,5]);`
`(*ACHTUNG WHITESPACE NACH * WICHTIG*)`
	`val it = 120 : int`

`foldl(fn (a, b) => 2*a*b)(1)([1,2,3,4,5]);`	`val it = 3840 : int`

Currying und Funktionen höherer Ordnung

Funktionen	Ergebnisse/Ausgaben (kurz)
`map;`	`val it = fn : ('a -> 'b) -> 'a list -> 'b list`
`map(fn(n)=>2*n)([1,2,3,4,5]);`	`val it = [2,4,6,8,10] : int list`

Danke für Eure Aufmerksamkeit!