Imperative und funktionale Sprachen

Merkmale imperativer Sprachen:

- Konzepte: Variablen, Zuweisungen, Sequentialität
- Programm-Semantik spiegelt Rechnerarchitektur wider
- Schleifen spielen eine große Rolle

Merkmale funktionaler Sprachen:

- Konzepte: Datenobjekte und mathematische Funktionen
- Programme haben keine Seiteneffekte
- Rekursion und funktionale Formen statt Schleifen

Funktionen

Eine Funktion bildet eine „domain“ in einen „range“ ab

Die Signatur der Funktion bestimmt „domain“ und „range“
Beispiel: square: integer $\rightarrow$ natural

Die Abbildungsregel spezifiziert den Wert im „range“ der mit jedem Wert in der „domain“ assoziiert ist
Beispiel: square(n) $\equiv$ n $\times$ n

Bei der Anwendung ersetzt ein Argument den Parameter
Beispiel: square(2)

Variablen in funktionalen Sprachen haben eine andere Bedeutung als in imperativen Sprachen – „single assignment“

Funktionale Formen und Rekursion

Funktionale Formen verknüpfen Funktionen zu Funktionen
Beispiel: F $\equiv$ G $\circ$ H

Funktionale Formen ersetzen sequentielle Ausführung

Beispiele für Rekursion:

```
n! ≡ if n = 0 then 1 else n * (n − 1)!

prime(n) ≡ if n = 2 then true else p(n, n div 2)

p(n,i) ≡ if (n mod i) = 0 then false
           else if i = 2 then true
                else p(n, i − 1)
```


Werte, Bindungen, Funktionen

Beispiele für Bindungen in ML:
```
val A = 3;
val B = "a";
```

Funktionen sind Werte:
```
val sq = fn(x:int) => x * x;
fun square(n:int) = n * n;
```

Funktionsanwendungen:
```
(fn(x:int) => x * x) 2
2 * sq(A)
```

Ausdrücke im Lambda-Kalkül

Ausdrücke im Lambda-Kalkül sind

- Namen (Identifier); Beispiele: x, 3, $*$
- Funktionsdefinition der Form (λx.e), wobei x ein Name (Parameter) und e ein Ausdruck (Funktionsrumpf) ist
  Beispiel: (λx.x $*$ x)
- Funktionsanwendung der Form (e1 e2), wobei der Ausdruck e1 auf den Ausdruck e2 angewandt wird
  Beispiel: ((λx.x $*$ x) 2)

Klammern können weggelassen werden
Beispiele: e1 e2 e3 = ((e1 e2) e3)
λx.y z = (λx.(y z))

Currying

Übliche Schreibweise f(x) im Lambda-Kalkül ersetzt durch f x

Aber: Es gibt kein direktes Äquivalent zu f(x,y,z)
Stattdessen: geschachtelte Funktionen („currying“)

Beispiel für geschachtelte Funktion und deren Anwendung:

(λx.λy.λz.x $*$ y $*$ z) 1 2 3

Klammerung und schrittweise Abarbeitung ergibt:

((((λx.(λy.(λz.(x $*$ y $*$ z)))) 1) 2) 3)
(((λy.(λz.(1 $*$ y $*$ z))) 2) 3)
((λz.(1 $*$ 2 $*$ z)) 3)
(1 $*$ 2 $*$ 3)

Ersetzung

Ersetzung aller freien (ungebundenen) Vorkommen von x in e2 durch e1 (formal [e1/x]e2):

- [e/x1]x2 = e wenn x1 = x2
  = x2 wenn x1 $\neq$ x2
- [e1/x1](λx2.e2) = (λx2.e2) wenn x1 = x2
  = (λx2.[e1/x1]e2) wenn x1 $\neq$ x2 und x2 nicht frei in e1 vorkommt
  = (λx3.[e1/x1][x3/x2]e2) sonst, wobei x1 $\neq$ x3 $\neq$ x2 und x3 nicht frei in e1 und e2 vorkommt
- [e1/x](e2 e3) = ([e1/x]e2 [e1/x]e3)

Äquivalenz von Lambda-Ausdrücken

Zwei Ausdrücke sind äquivalent wenn sie durch Anwendung folgender Regeln ineinander umgeformt werden können:

- Umbenenennen von Parametern (α conversion):
  λx1.e $\leftrightarrow$ λx2.[x2/x1]e (wobei x2 nicht frei in e)
- Funktionsanwendung (β conversion):
  (λx.e1) e2 $\leftrightarrow$ [e2/x]e1
- Elimination redundanter Funktionen (η conversion):
  λx.(e x) $\leftrightarrow$ e (wobei x nicht frei in e)

Diese Regeln sind nicht gerichtet

Reduktion und Normalform

Reduktionsregeln entsprechen obigen Regeln, wobei die beiden letzten Regeln (β- und η-Reduktion) nur von links nach rechts angewandt werden dürfen.

Ein Ausdruck ist in Normalform wenn β- und η-Reduktion nicht anwendbar sind.

Wenn es für einen Ausdruck einen äquivalenten Ausdruck in Normalform gibt, dann führen alle möglichen wiederholten Anwendungen der Reduktionsregeln zum selben Ausdruck in Normalform (bis auf Umbenennen von Parametern).

Nicht jeder Ausdruck hat eine entsprechende Normalform

Beispiel für Reduktion, Varianten

Beispiel mit Normalform:

$$
\begin{aligned}
&(\lambda x.(\lambda y.x + y)\ 5)\ ((\lambda y.y * y)\ 6) = \\
&(\lambda x.x + 5)\ ((\lambda y.y * y)\ 6) = \\
&(\lambda x.x + 5)\ (6 * 6) = \\
&(6 * 6) + 5
\end{aligned}
$$

Der reine Lambda-Kalkül behandelt die Auswertung der Operatoren ($*$, $+$, etc.) nicht selbst.
Es gibt aber Varianten des Lambda-Kalküls, die auch primitive Operationen bereitstellen.

Es gibt auch typisierte Varianten des Lambda-Kalküls mit ähnlichen Eigenschaften.

LISP

Beispiele für Symbole:	A	(Atom)
	AUSTRIA	(Atom)
	68000	(Zahl)

Beispiele für Listen:
(PLUS A B)
((MEAT CHICKEN) WATER)
(UNC TRW SYNAPSE RIDGE HP)
NIL bzw. () entsprechen leerer Liste

Eine Zahl representiert ihren Wert direkt.
Ein Atom ist der Name (l-Wert) eines r-Wertes.

(SET X (A B C)) bindet X global an (A B C).
(LET (X A) (Y B)) bindet X lokal an A und Y an B.

Primitive Funktionen in LISP

Das erste Element einer Liste wird normalerweise als Funktion interpretiert, angewandt auf die restlichen Listenelemente.

(QUOTE A) oder kurz 'A liefert das Argument A selbst als Ergebnis. Es wird also nicht der r-Wert von A berechnet.

Beispiele für primitive Funktionen:

```
(CAR '(A B C)) = A             (ATOM 'A) = T
(CAR 'A) = error               (ATOM '(A)) = NIL
(CDR '(A B C)) = (B C)         (EQ 'A 'A) = T
(CDR '(A)) = () = NIL          (EQ 'A 'B) = NIL
(CONS 'A '(B C)) = (A B C)     (COND ((EQ 'X 'Y) 'B)
(CONS '(A) '(B)) = ((A) B)           (T 'C)) = C
```

Definition von Funktionen in LISP

(LAMBDA (X Y) (PLUS X Y)) ist eine Funktion mit zwei Parametern.

((LAMBDA (X Y) (PLUS X Y)) 2 3) wendet die Funktion an; Ergebnis ist 5.

(DEFINE (ADD (LAMBDA (X Y) (PLUS X Y)))) definiert einen globalen Namen für die Funktion.

Beispiel:

```
(DEFINE (REVERSE (LAMBDA (L) (REV NIL L))))
(DEFINE (REV (LAMBDA (OUT IN)
           (COND ((NULL IN) OUT)
                 (T (REV (CONS (CAR IN) OUT)
                         (CDR IN)))))))
```


Funktionale Formen, Eigenschaften

Bekannteste funktionale Form: (MAPCAR SQUARE L) wendet SQUARE auf jedes Element der Liste L an und gibt die Liste der Ergebnisse zurück.

In den meisten LISP-Dialekten sind weitere funktionale Formen vordefiniert. Eigene funktionale Formen sind definierbar.

Bemerkenswerteste Eigenschaft von LISP ist die semantische Einfachheit: Ein einfacher Interpreter lässt sich in wenigen Zeilen schreiben.

Heute übliche LISP-Interpreter beinhalten Bibliotheken mit einer Vielzahl von Funktionen und funktionalen Formen für die verschiedensten Anwendungsgebiete.
Deswegen ist LISP in einigen Bereichen noch immer populär.

APL

In APL verwendet man zwar häufig destruktive Zuweisungen, aber im Grunde ist APL eine applikative Sprache.

Datenobjekte sind „scalars“ (Nummern und Zeichen) und „arrays“ (Folgen von durch Leerzeichen getrennten Elementen). 1 und 0 werden als wahr bzw. falsch interpretiert.

Beispiel für Zuweisungen: $X \leftarrow 123$
$X \leftarrow$ 'b'
$X \leftarrow 5\ 6\ 7\ 8\ 9$

Auch Zuweisungen produzieren Werte (wie in C).
Beispiel: $X \leftarrow (Y \leftarrow 5\ 6\ 7\ 8\ 9) \times (Z \leftarrow 9\ 9\ 7\ 6\ 5)$
$W \leftarrow Y - Z$

Funktionen in APL

Es gibt zahlreiche primitive Funktionen (Operationen):
Arithmetische Operationen: $+$, $-$, $\times$, $\div$, $|$ (Divisionsrest)
Boolsche Operationen: $\wedge$, $\vee$, $\sim$, $<$, $\leq$, $=$, $\geq$, $>$, $\neq$

Alle Operationen auf „scalars“ sind (elementweise) auch auf „arrays“ anwendbar.

„ι“ generiert „arrays“: $\iota 5 \equiv 1\ 2\ 3\ 4\ 5$

„;“ hängt zwei „arrays“ zusammen: $\iota 4; \iota 5 \equiv 1\ 2\ 3\ 4\ 1\ 2\ 3\ 4\ 5$

„ρ“ gruppiert „arrays“: $2\ 3\ \rho\ 1\ 2\ 3\ 4\ 5\ 6 \equiv \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$

„/“ komprimmiert „arrays“: $1\ 0\ 0\ 1\ /\ \iota 4 \equiv 1\ 4$

Funktionale Formen in APL

- Der Reduktions-Operator „/“ wendet eine Operation schrittweise auf ein „array“ an.
  Beispiele: $+/(\iota 3) \equiv 6$
  $+/(2\ 3\ \rho\ \iota 6) \equiv +/[2](2\ 3\ \rho\ \iota 6) \equiv 6\ 15$
  $+/[1](2\ 3\ \rho\ \iota 6) \equiv 5\ 7\ 9$
- Der innere-Produkt-Operator „.“, verwendet als X f.g Y auf Matrizen X und Y, wendet g elementweise auf X (zeilenweise) und Y (spaltenweise) an und führt schließlich eine Reduktion mit Hilfe von f aus.
  Beispiel: $X +.\times Y$ multipliziert die Matrizen X und Y
- Der äussere-Produkt-Operator „∘“, verwendet als A ∘.f B auf „arrays“ A und B, wendet f auf jedes Element aus A und jedes aus B an.
  Beispiel: $1\ 2 \circ.\times\ 3\ 4\ 5 \equiv \begin{bmatrix} 3 & 4 & 5 \\ 6 & 8 & 10 \end{bmatrix}$

Ein APL-Programm

Beispiel: Berechnung der Primzahlen von 1 bis N

Schritt 1. $(\iota N) \circ.|\ (\iota N)$

Schritt 2. $0 = (\iota N) \circ.|\ (\iota N)$

Schritt 3. $+/[2]\ 0 = (\iota N) \circ.|\ (\iota N)$

Schritt 4. $2 = (+/[2]\ 0 = (\iota N) \circ.|\ (\iota N))$

Schritt 5. $(2 = (+/[2]\ 0 = (\iota N) \circ.|\ (\iota N)))\ /\ \iota N$

ML

Beispiele für Listen:
[2, 3, 4]
["a", "b", "c"]
[true, false]
[] ≡ nil

Beispiele für Listenoperationen:
hd([1, 2, 3]) ≡ 1
tl([1, 2, 3]) ≡ [2, 3]
1::[2, 3] ≡ [1, 2, 3]
[1, 2]@[3] ≡ [1, 2, 3]

Lokale Bindungen:
```
let val x = 5 in 2 * x * x end;
local val x = 5 in val sq = x * x
                   val cube = x * x * x end;
```


Einfache Programme in ML

```
fun reverse([]) = []
|   reverse(x::xs) = reverse(xs)@[x];

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

fun sort([]) = []
|   sort(s::xs) = insert(x, sort(xs));
```

Funktionen in ML

Funktion ohne Namen: fn(x:int) => −x;
fn(x:int, y:int) => x ∗ y;

Verwendung: val intnegate = fn(x:int) => −x;
Typ: fn:int−>int

Funktion höherer Ordnung: fun compose(f,g) = f(g(x));
Typ: fn:('a−>'b ∗ 'c−>'a)−>('c−>'b)

Curried Funktion: fun times(x:int)(y:int) = x ∗ y;
Typ: fn:int−>(int−>int)
Anwendung: fun multby5(x) = times(5);

Funktionale Formen in ML

map entspricht MAPCAR in LISP; map is curried.
Example: map length [[], [1, 2, 3], [3]] ≡ [0, 3, 1]

Folgende Funktion entspricht dem Reduktionsoperator / in APL:

```
fun fold(f, s, nil) = s
|   fold(f, s, (h::t)) = f(h, fold(f, s, t));
```

Anwendungsbeispiel: fold((op+), 0, [1,2,3,4]) ≡ 10

Generell können funktionale Formen leicht als Funktionen höherer Ordnung (mit „currying") realisiert werden.

Typen in ML

Vordefinierte Typen: bool, int, real, string

Listen: [1, 2, 3] : int list
["a", "b"] : string list
nil : 'a list

Tupel: (true, "fact", 7) : bool * string * int
(true, nil) : bool * ('a list)

Verbunde: {name = "A", id = 9} : {name:string, id:int}

Funktionen: identitaetsfunktion : 'a−>'a

Typdefinition: type intpair = int * int;
type 'a pair = 'a * 'a;
type boolpair = bool pair;

Datentypen in ML

datatype color = red | green | blue;

datatype publications = nopubs | journal of int | conf of int;

datatype 't stack = empty | push of 't ∗ 't stack;
Werte von int stack sind: empty
push(2, empty)
push(2, push(3, push(4, empty)))

datatype 't list = nil | :: of 't ∗ 't list;