Grundlagen der Programmierung

Funktionale Programmierung:
Arbeiten mit Kollektionen ♦ Currying

Überblick über Programmierstile und -sprachen

Funktionale Programmierung

- Programme: mathematische Berechnungen
- Programme bestehen (nur) aus
  - Funktionsdefinitionen
  - Funktionsaufrufen
- Funktionsaufrufe können weitere Funktionsaufrufe enthalten
  → Realisierung von **Komposition**
  - *Keine Zuweisungen der Form* `x = x + 1`
    → `f(x)` mit $f(x) = x + 1$
    und ggf. Weiterverwendung als Argument einer Funktion
- **Rekursion**
  - *Keine Berechnungen mit Hilfe von Schleifen!*

Wdh.: Funktionen höherer Ordnung

- Funktionen, die Funktionen als Argumente (oder Funktionswerte) haben
- Wichtige Funktionen höherer Ordnung für Operationen auf Kollektionen (Listen):
  - **map**
  - **filter**
  - **reduce**

  → sind in funktionalen Sprachen oft vordefiniert

Die Funktion map

- **Argumente:** eine Funktion, eine Kollektion
  (map ist Funktion zweiter Ordnung)
- **Funktionswert:** eine Kollektion

$$\boxed{\mathrm{map}(f, L) = K \quad \mathrm{mit} \quad K[i] = f(L[i])}$$

- *Beispiel:*
  map(lambda x: 2**x, [0,1,2,3]) = [1,2,4,8]

- map in Python vordefiniert:
  `list(map(lambda x: 2**x, [0,1,2,3]))`

Die Funktion filter

- **Argumente:** ein Prädikat, eine Kollektion *(Fkt. zweiter Ordnung)*
- **Funktionswert:** eine Kollektion

$$\boxed{\text{filter}(p,L) = K \;\text{ mit }\; L[i] \text{ in } K \;\text{ gdw. }\; p(L[i]) = \text{True}}$$

- *Beispiel:*

  filter(lambda x: x%2 == 0, [0,1,2,3]) = [0,2]

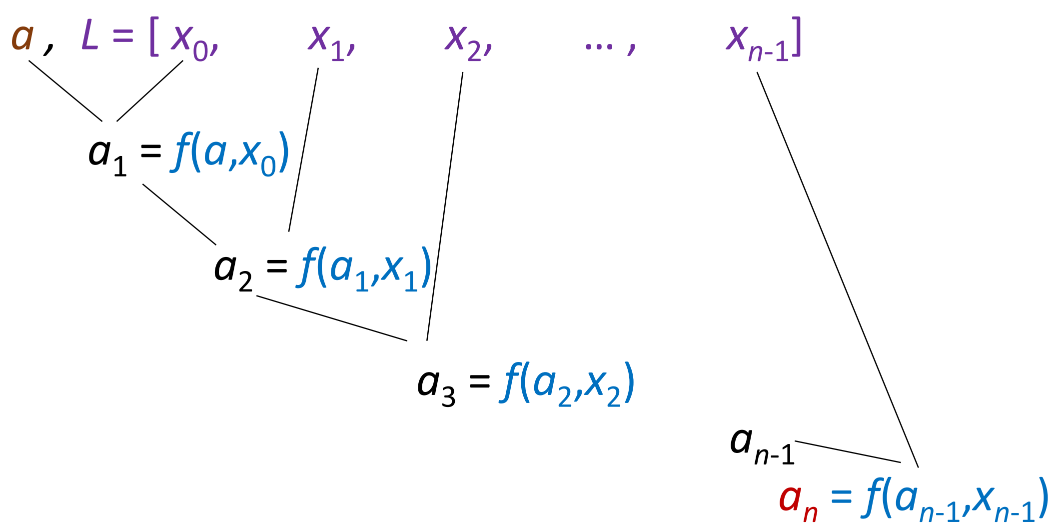
- filter in Python vordefiniert:

```
list(filter(lambda x: x%2==0, [0,1,2,3]))
```

Die Funktion reduce

- **Argumente:** eine zweistellige Funktion, eine Kollektion, ein Anfangswert (Akkumulator) *(Fkt. 2. Ordnung)*
- **Funktionswert:** ein Wert

$$\boxed{\text{reduce}(f, L, a) = f(\ldots f(f(a, L[0]), L[1]), \ldots, L[\text{len}(L)-1])}$$



Die Funktion reduce: Beispiel

- reduce(lambda x,y: x+y, [0,1,2,3], 100)
  $\rightarrow$ ((((100 + 0) + 1) + 2) + 3) = 106
- mit $a = 0$: Funktionswert 6 (Summe über alle Elemente in L)
- in Python vordefiniert im Modul **`functools`**

```
from functools import reduce
reduce(lambda x,y: x+y, [0,1,2,3], 0)    # 6
```

Linkssequenzen

- Spezielle Repräsentation von Listen
- *Trägermenge:*
  Menge aller Paare (first, rest) sowie ()
  - **first**: erstes Listenelement
  - **rest**: Linkssequenz mit den restlichen Listenelementen

Datentyp: Tupel

- *Beispiel:* Liste [1,2,3] als Linkssequenz:

(1, r_1) mit r_1 = (2, r_2) mit r_2 = (3, r_3) mit r_3 = () = **empty**

$\rightarrow$ ls = (1, (2, (3, ()))) = (1, (2, (3, empty)))

Tupel in Python

... genau wie Listen, aber

- Tupel sind **unveränderlich**
- Definition in runden Klammern:
```
tpl = (4, 9, 12, -1, 3)
tpl[0] # 4
tpl[4] # 3
len(tpl) # 5
```
- alle Operationen wie bei Listen, die das Tupel nicht verändern

Rechtssequenzen

- Menge aller Paare (rest, last) sowie () = **empty**
  - **last**: letztes Element
  - **rest**: Rechtsequenz mit den restlichen (vorderen) Elementen

- *Beispiel:* Liste [1,2,3] als Rechtssequenz:

  rs = (((empty, 1), 2), 3)

  last: 3 , **rest**: ((empty, 1), 2)

Links und Rechtssequenzen (rekursiv)

- Sei $D = (M_D, \circ_1, \circ_2, ..., \circ_n)$ ein Datentyp.
- **Linkssequenz über *D*** (*rekursive Definition*):
  1. () = **empty** ist eine Linkssequenz über *D*.
  2. Das Paar (*v*, *xs*) ist eine Linkssequenz über *D*, falls $v \in M_D$ und *xs* eine Linkssequenz über *D* ist.
  3. Es gibt keine weiteren Linkssequenzen über *D.*

- **Rechtssequenz über *D*** (*rekursive Definition*):
  1. () = **empty** ist eine Rechtssequenz über *D*.
  2. Das Paar (*xs, v)* ist eine Rechtssequenz über *D*, falls $v \in M_D$ und *xs* eine Rechtssequenz über *D* ist.
  3. Es gibt keine weiteren Rechtssequenzen über *D.*

Linkssequenzen - funktional

Basis-Funktionen:

- **first**(*xs*) gibt erstes Element von *xs* zurück
- **rest**(*xs*) gibt die zweite Komponente von *xs* zurück

xs = (1, (2, (3, empty)))

first(*xs*) = 1, **rest**(*xs*) = (2, (3, empty))

first(**rest**(*xs*)) = 2, ...

Komposition

Linkssequenzen – Basisoperationen

- first(*xs*) gibt erstes Element von *xs* zurück:

```
def first(xs):
    if xs == empty:
        print("Empty Sequence.") # Fehler
    else: return xs[0]
```

- rest(xs) gibt rest(xs) zurück:

```
def rest(xs):
    if xs == empty:
        print("Empty Sequence.") # Fehler
    else: return xs[1]
```

Linkssequenzen – Operationen (1)

- concat(*xs*,*ys*)   Verknüpfen zu einer Linkssequenz:

```
def concat(xs, ys):
    if xs == empty:
        return ys
    else:
        return (first(xs), concat(rest(xs), ys))
```

Datentypen

Sei LS_D die Menge aller Linkssequenzen über *D*.

first : $(LS_D \setminus \{\text{empty}\}) \rightarrow M_D$

rest: $(LS_D \setminus \{\text{empty}\}) \rightarrow LS_D$

concat: $LS_D \times LS_D \rightarrow LS_D$

Linkssequenzen – Operationen (2)

- length(*xs*)   Anzahl der gespeicherten Elemente

  length: $LS_D \rightarrow \mathbb{N}$

```
def length(xs):
    if xs == empty:
        return 0
    else:
        return 1 + length(rest(xs))
```

Linkssequenzen – Operationen (3)

- append(x, xs) x als neues letztes Element

  append: $M_D \times LS_D \rightarrow LS_D$

```
def append(x,xs):
   if xs == empty:
      return (x, empty)
   else:
      return (first(xs), append(x, rest(xs)))

# alernativ: return concat(xs, (x,empty))
```

Linkssequenzen – Operationen (4)

- map(*f*, *xs*) Linkssequenz, nachdem *f* auf alle Elemente von *xs* angewandt wurde

map: $(M_D \to M_D) \times \mathrm{LS}_D \to \mathrm{LS}_D$

```
def map(f, xs):
    if xs == empty:
        return empty
    else:
        return (f(first(xs)), map(f, rest(xs)))
```

Linkssequenzen – Operationen (5)

- filter(p, xs) Linkssequenz, die alle Elemente von xs enthält, die das **Prädikat** p erfüllen

$\boldsymbol{p : M_D \to \{0,1\}}$

filter: $(M_D \to \{0,1\}) \times \mathrm{LS}_D \to \mathrm{LS}_D$

```
def filter(p, xs):
    if xs == empty: return empty
    elif p(first(xs)):
         return (first(xs), filter(p,rest(xs)))
    else:
         return filter(p,rest(xs))
```

Linkssequenzen – Operationen (6)

- reduce(f, xs, a) iterierte Komposition von f, angefangen mit $a \in A$ und first(xs):

$f(\ldots f(f(f(a,x_0),x_1),x_2), \ldots,x_n)$ falls $xs = (x_0,(x_1,(x_2,\ldots,(x_n,\text{empty})\ldots)))$

reduce: $(A \times M_D \to A) \times \mathrm{LS}_D \times A \to A$

```
def reduce(f,xs,a):
    if xs == empty:
        return a
    else:
        return reduce(f, rest(xs), f(a,first(xs)))
```

Akkumulator

Zusammenfassung

Kennzeichen funktionaler Programmierung:

- Funktionsaufrufe und Komposition
  (*statt Veränderung von Variablenwerten*
  → Variablenwerte sind unveränderlich!)
- Rekursion *(statt Schleifen)*
- Funktionen als gleichberechtigte Datenobjekte
  (Funktionen höherer Ordnung)
- anonyme Funktionen (lambda-Ausdrücke)

Weitere Merkmale (sprachabhängig)

- nur **reine** Funktionen und Ausdrücke
- **nicht strikte** („lazy") Auswertung
- nur einstellige Funktionen → **Currying**
- **Typsystem**

Reine Funktionen und Ausdrücke

- geben nur ihren Wert zurück; **keine Seiteneffekte**
- nicht reiner Ausdruck: `++n` reiner Ausdruck: `n+1`
- reine Funktion: keine Änderung an Daten, die nicht lokal sind
- *Python-Funktionen*: Keine Änderungen an Parametern, die nicht **immutable** sind!
- wichtige Datentypen, die in Python immutable sind:
  - int, float, bool, str
  - tuple
  - range

Strikte *versus* nicht strikte Auswertung

- Strikte Auswertung:
  - zuerst alle aktuellen Parameter auswerten
  - danach zur ersten Anweisung im Funktionsrumpf
- Nicht strikte Auswertung:
  - Auswertung der aktuellen Parameter erst, wenn deren Werte benötigt werden
- `f([2.5, 3/4, 1/0]) # gibt L[0] zurück`
  - nicht strikt: Ergebnis 2.5
  - strikt: Fehler

Currying – Idee

- Überführung mehrstelliger in einstellige Funktionen
- $f(x,y) = 2x + y$
  - **zweistellige Sicht:**
    1. x und y festlegen — $x = 2,\ \ y = 3$
    2. Wert berechnen — $f(2,3) = 2 \cdot 2 + 3 = 7$
  - **Currying:**
    1. x festlegen — $x = 2 \ \rightarrow\ h_2(y) = 4 + y$
    2. y festlegen — $y = 3 \ \rightarrow\ h_2(3) = 4 + 3 = 7$

Currying – Verfahren

Gegeben: $f : A \times B \to C$ (*zweistellig*)

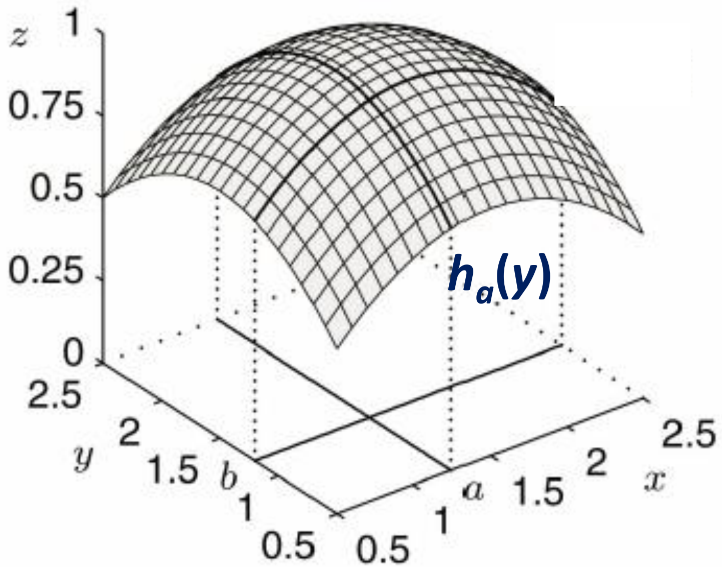
1. einstellige Funktion
   $h : A \to (B \to C)$ mit $h(x) = h_x$

2. einstellige Funktion
   $h_x : B \to C$ mit $h_x(y) = f(x,y)$

   $x \mapsto (y \mapsto z)$

- *mehrstellig*: $f(u,v,w,x,y) = z$

  $u \mapsto (v \mapsto (w \mapsto (x \mapsto (y \mapsto z))))$



Currying in Python

- *traditionell:*

```
def f(x,y):
    return 2*x + y
```

- *curried:*

```
def f(x):
    return lambda y: 2*x+y

f(2)(3)    # 7
```

alternativ:

```
h_2 = f(2)
h_2(3)    # 7
```

- *als reiner lambda Term:*

```
f_lambda = lambda x: lambda y: 2*x+y

f_lambda(2)(3)    # 7
```

Typsystem

- Mit der *Definition* der Funktion:
  Deklaration der Datentypen
  - der Parameter
  - und des Rückgabewertes
- Programmiersprache
  - **prüft** bei jedem *Aufruf*, ob die aktuellen Parameter passen
  - **garantiert,** dass dann der Rückgabewert passt.
- *Beispiel:* Länge eines Strings

```
int len(String s)
```

(*nicht in Python*)

Taxonomie von Programmierstilen

Imperativer Stil

Programm legt fest,
wie gerechnet wird

→ Befehlsfolge

Prozedurale
Programmierung

Objektorientierte
Programmierung

Deklarativer Stil

Programm legt fest,
was berechnet wird

→ Rechenweg wird vom **Laufzeitsystem** festgelegt

Funktionale
Programmierung

Logische
Programmierung

Laufzeitsystem

- Sammlung von Prozeduren, die den Ablauf von Programmen ermöglichen
  - Anlegen von Variablen im Speicher
  - Verwaltung des Aufrufstacks
  - ...

→ s. Vorlesung zu Interpretern und Compilern

Imperativer Stil

- **Prozedurale Programmierung**
  - Definition von **Prozeduren** als Folgen von Befehlen
  - Aufruf von Prozeduren durch Prozeduren
  - **Funktionen** als Prozeduren mit Rückgabewert
  - *Sprachen:* Fortran, Cobol, Algol 60, Algol 68, Pascal, C, ...

- **Objektorientierte Programmierung**
  - Weiterentwicklung der prozeduralen Programmierung
  - Zuordnung von Funktionen zu den Datenobjekten, die sie manipulieren (Methoden zu Objekten)
  - *Sprachen:* Smalltalk, Java, C#, ...

Deklarativer Stil

- **Funktionale Programmierung**
  - *Intuitiver Algorithmus-Begriff (Kerngedanke):* Abbildung von Eingabedaten auf Ausgabedaten, maschinell ausführbar
    → algorithmisch berechenbare Funktion
  - Prozeduraufrufe durch **Komposition von Funktionen**
  - *Sprachen:* LISP, Scheme, ML, Caml, OCaml, Haskell, ...

- **Logische Programmierung** (*Sprache:* Prolog)
  - Definition von **Fakten/Axiomen** und **Regeln** als Wissensbasis
  - Beantwortung von Anfragen durch Anwendung der Regeln auf Axiome nach den Prinzipien der math. Logik

Prinzip logischer Programmierung

- Drei Personen werden des Mordes verdächtigt.
- Axiome:
  - *A* oder *B* oder *C* ist schuldig. ($\boldsymbol{A} \lor \boldsymbol{B} \lor \boldsymbol{C}$)
  - *A* ist unschuldig. ($\neg \boldsymbol{A}$) *// A hat ein Alibi.*
- Regel:
  - Wenn *B* unschuldig ist, dann auch *C*. ($\neg \boldsymbol{B} \rightarrow \neg \boldsymbol{C}$)
    // Aussage von B.
- Anfrage: Ist *B* schuldig? → Ja.

B	*C*
~~0~~	~~1~~
1	0
1	1

Axiome und Regeln

- **Axiom**
  wahre Aussage (Werte von Prädikaten)

- **Regel**
  Aussage der Form $A \longleftarrow B_1, B_2, \ldots, B_n$
  - A – **Kopf** der Regel
  - $B_1, B_2, \ldots, B_n$ – **Rumpf** der Regel
  - steht für $B_1 \wedge B_2 \wedge \ldots \wedge B_n \rightarrow A$
  - Gewinnen neuer (A) aus bekannten Fakten ($B_1, B_2, \ldots, B_n$)
  - Spezialfall $n = 0$: Axiom

Beispiel in Prolog

```
% Axiome
male(Frank).
male(Egon).
female(Anna).
father(Egon, Frank).

% Regel
son(X,Y) :-
       father(Y,X), male(X).
daughter(X,Y) :-
       father(Y,X), female(X).
```

```
% Anfragen
?- male(Frank).
yes.
?- male(Anna).
no.

?- son(Frank, Egon).
yes.
?- daughter(Anna,Egon).
no.
```

Hybride Sprachen

- Viele Sprachen unterstützen nicht nur ein Paradigma.
  - Python: prozedural, objektorientiert, funktional
  - C++: prozedural, objektorientiert, funktional
  - Delphi: prozedural, objektorientiert
  - Scala: funktional, objektorientiert