

Einführung in die Programmiertechnik

Unterprogramme

Unterprogramme

- Zerlegung eines Algorithmus in kleinere Teile
 - Zur Verbesserung der Lesbarkeit
 - Zur mehrfachen Verwendung im gleichen Programm (Vermeidung von Codedopplungen)
 - Zur Wiederverwendung in anderen Programmen (Bibliothek)
- Python: Funktionsdefinitionen
def funktionsname(parameter):
funktionskörper
- Variablen, die innerhalb der Funktion belegt werden, sind nur bis zum Ende der Funktion gültig
 - Lokale Variablen
- Sollen Variablen den Wert auch außerhalb der Funktion behalten, so müssen sie mit `global` deklariert werden
`global zahl_der_streichhoelzer`
 - Deklaration optional, falls globale Variable nur gelesen wird

Unterprogramme: Ein Beispiel

```
import sys
def Sterne(k):
    for i in range(k):
        sys.stdout.write('*')

for zeile in range(5):
    Sterne(zeile)
    sys.stdout.write('\n')
```

Prozedurale Abstraktion

- Zerlegung des Programms in Teilschritte mithilfe von Prozeduren: prozedurale Abstraktion
 - Unterprogramme, die „wie Anweisungen“ verwendet werden
 - Prozeduren liefern kein Ergebnis

```
# Zeichne einen Weihnachtsbaum
import sys
def Sterne(zeichen, anzahl):
    "Sterne(zeichen, anzahl) gibt anzahl zeichen auf sys.stdout"
    for i in range(anzahl):
        sys.stdout.write(zeichen)

# Zeichne zuerst die Krone
H = 5
for zeile in range(H):
    Sterne(" ", H - zeile - 1)
    Sterne("*", 2*zeile+1)
    sys.stdout.write("\n")
# jetzt noch der Stamm
for zeile in range(3):
    Sterne(" ", H - 1)
    sys.stdout.write("*\n")
```

Funktionale Abstraktion

- Funktionen: Unterprogramme, die Werte liefern
 - Verwendbar als Ausdrücke
 - i.d.R. keine Seiteneffekte
- Python: return-Anweisung bestimmt Funktionsergebnis
 - Konvention: möglichst nur eine return-Anweisung am Ende der Funktion

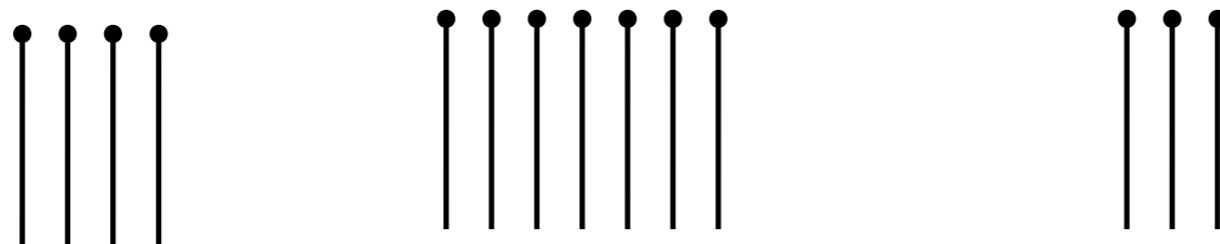
```
def ggT(x, y):  
    while x<>0 and y<>0:  
        if x > y:  
            x = x%y  
        else:  
            y = y%x  
    return x+y
```

Top-Down-Entwurf

- Annahme: Spezifikation gegeben
 - Ansonsten: Zunächst Problemanalyse
- Beginn der Entwicklung: Programmgerüst
 - Verlagerung von Teilfunktionen in leere Unterprogramme
 - Prozedurrümpfe (*stubs*): abstrakte Operationen
 - Python: Leeraanweisung „pass“
def anmelden(vorname, nachname):
 pass
- Schrittweises Ausfüllen des Gerüsts: Verfeinerung
 - iterativer Prozess: in jedem Schritt werden u.U. neue Prozedurrümpfe eingeführt

Top-Down-Entwurf: Ein Beispiel

- NIM-Spiel (*Nim Game*): Das Spiel beginnt mit drei Reihen von Streichhölzern. Zwei Spieler ziehen abwechselnd. Ein Zug besteht darin, eine Reihe auszuwählen und aus dieser Reihe beliebig viele – jedoch mindestens ein – Streichholz wegzunehmen. Wer das letzte Streichholz nimmt, gewinnt.
- Aufgabe: Gesucht ist ein Programm, das dieses Spiel spielt



NIM-Spiel: Programmgerüst (v1)

```
init()
fertig = False
spieler = 1
while not fertig:
    zeige_spiel()
    mache_zug()
    if spiel_ende():
        fertig = True
    else:
        spielerwechsel()
gratuliere_dem_sieger()
```

NIM-Spiel: Programmgerüst (v2)

```
reihe1 = reihe2 = reihe3 = 0
def init():
    global reihe1, reihe2, reihe3
    reihe1, reihe2, reihe3 = 4, 7, 3
def zeige_spiel():
    pass
def mache_zug():
    pass
def spiel_ende():
    return True
def spieler_wechsel():
    pass
def gratuliere_dem_sieger():
    pass

# ... weiter wie auf letzter Folie
init()
...
```

Top-Down-Entwurf: Verfeinerung

- Aufgabenstellung unterspezifiziert; deshalb mehrere mögliche Verfeinerungen
 1. Vervollständigung der Routinen, so dass ein Demospiel vorführbar ist
 2. Einbau einer Strategie, um den bestmöglichen Zug zu ermitteln
 3. Erweiterung der Nutzerinteraktion
 - Initiale Abfrage der Streichholzzahlen
 - Integration des Nutzers als einer der Spieler
 4. graphische Ausgabe statt textueller

Top-Down-Entwurf: Vervollständigung zu Demo-Programm

```
def init():
```

```
    global reihe1, reihe2, reihe3, zugnummer  
    reihe1, reihe2, reihe3 = 4, 7, 3  
    zugnummer = 0
```

```
def zeige_spiel():
```

```
    "Ausgabe des Spielfelds auf die Standardausgabe"  
    print "Zugnummer", zugnummer  
    print "Reihe 1", reihe1  
    print "Reihe 2", reihe2  
    print "Reihe 3", reihe3
```

```
def spiel_ende():
```

```
    # Das Spiel ist fertig, wenn keine Streichhölzer mehr übrig sind  
    return reihe1+reihe2+reihe3 == 0
```

```
def spieler_wechsel():  
    global spieler  
    # Aus Spieler 1 wird Spieler 2 und umgekehrt  
    spieler = 3-spieler  
  
def gratuliere_dem_sieger():  
    # gewonnen hat, wer den letzten Zug gemacht hat;  
    # dieser Wert ist noch in Variable spieler gespeichert  
    print "Gewonnen hat der Spieler", spieler  
    print "Herzlichen Glückwunsch"
```

NIM-Spiel: Der bestmögliche Zug

- Spieler sollte versuchen, dem Gegner ein Spielfeld vorzulegen, bei dem $\text{reihe1} \mathbf{xor} \text{reihe2} \mathbf{xor} \text{reihe3} = 0$
 - nach gegnerischem Zug kann dann nicht mehr $\text{reihe1} \mathbf{xor} \text{reihe2} \mathbf{xor} \text{reihe3} = 0$ gelten
 - Spieler kann nach eigenem Zug diesen Zustand immer wieder herstellen
 - nach letztem Zug gilt $\text{reihe1} \mathbf{xor} \text{reihe2} \mathbf{xor} \text{reihe3} = 0$

```
def mache_zug():
    global zugnummer, reihe1, reihe2, reihe3
    zugnummer += 1
    if reihe1 > (reihe2 ^ reihe3):
        reihe1 = reihe2 ^ reihe3
    elif reihe2 > (reihe1 ^ reihe3):
        reihe2 = reihe1 ^ reihe3
    elif reihe3 > (reihe1 ^ reihe2):
        reihe3 = reihe1 ^ reihe3
    else:
        verlegenheitszug()
```

Datenfluss zwischen Haupt- und Unterprogramm

- Änderung von globalen Variablen: Seiteneffekt
 - Variablenänderung nicht in Funktionsergebnis sichtbar
 - Abhängigkeiten zwischen Prozeduren werden unüberschaubar
- Kapselung: Zahl der Prozeduren, die auf gemeinsame Variablen zugreifen, sollte klein sein
 - *information hiding*
 - im Beispiel: eine Prozedur
def nimm_von_reihe(reihe, zahl):

...
würde das eigentliche Wegnehmen von der Strategie trennen
 - Objektorientierte Programmierung: Kapselung von Werten in Objekten
 - im Beispiel: „Spielfeld“ als Klasse, mit Lese- und Schreiboperationen für die Reihen (Alternativ: „Spielstand“)

Rekursion

- Beschreibung einer mathematischen Funktion oft durch Fallunterscheidung; ein Fall bezieht sich wieder auf die Funktion
 - Beispiel: Fakultät $\text{fac}(n) = n!$
 $\text{fac}(0) = 1$
 $\text{fac}(n) = n \cdot \text{fac}(n-1)$, falls $n > 0$
- Ableitung einer Berechnungsvorschrift: sequentielles durchtesten der Fälle

```
def fac(n):
```

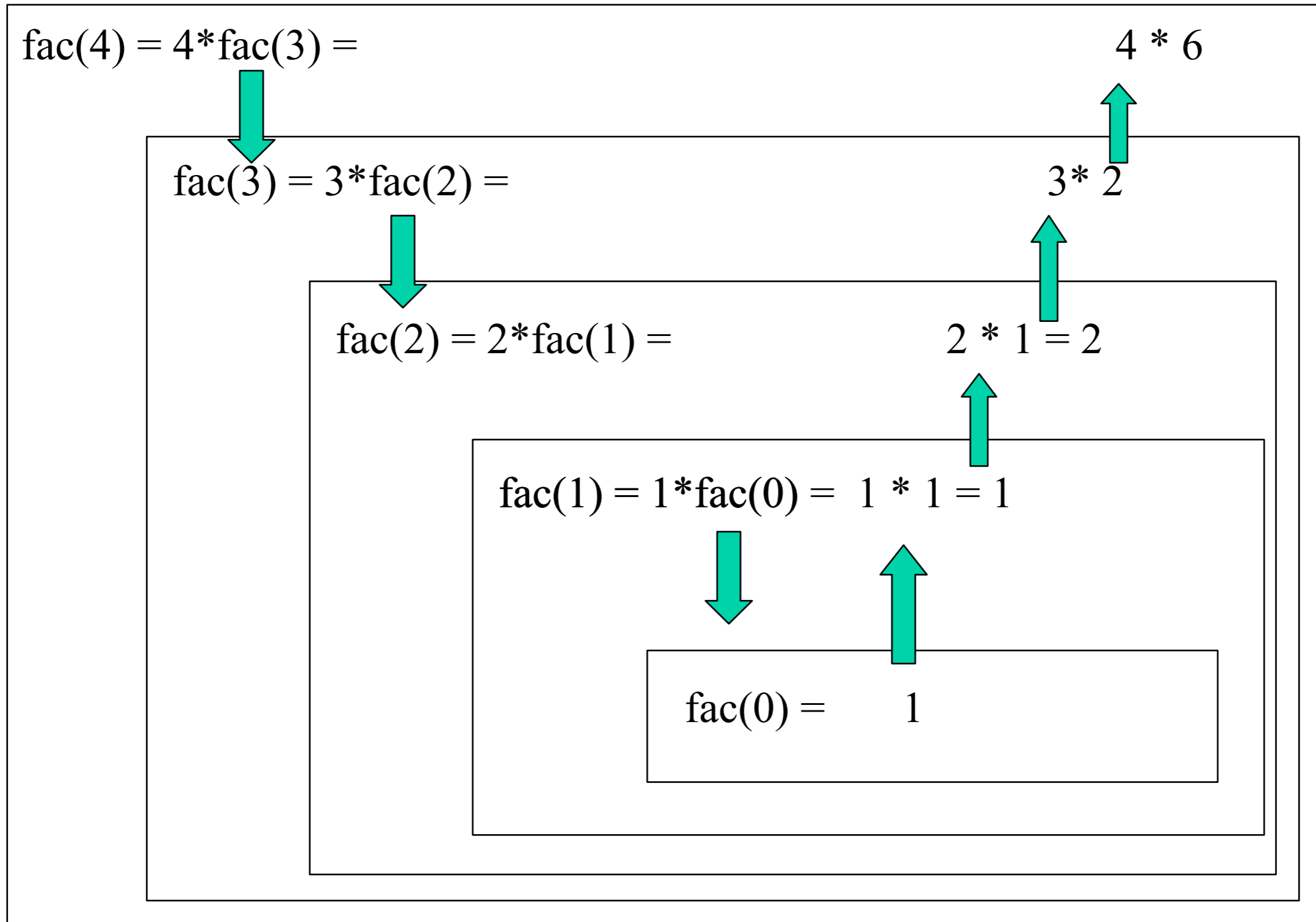
```
    if n == 0:
```

```
        return 1
```

```
    if n > 0:           # Test auf n>0 ist hier redundant
```

```
        return n * fac(n-1)
```

Verarbeitung der Rekursion



Formulierung von Rekursion

- Zerlegung des Problems in Teilprobleme:
 - ein Teilproblem ist „gleichartig“ dem Originalproblem
 - Beispiel: $\text{fac}(n-1)$ ist „im Prinzip“ genauso wie $\text{fac}(n)$
 - Teilproblem muss „leichter“ lösbar sein als Originalproblem
 - anderes Teilproblem kombiniert die Lösung des ersten Teilproblems mit weiterem Rechenschritt zur Gesamtlösung:
 - Beispiel: $\text{fac}(n) = n * \text{fac}(n-1)$ (Multiplikation der Teillösung mit n)
- Rekursionsabbruch: „einfachstes“ Teilproblem wird nicht weiter zerlegt; Lösung wird „direkt“ bestimmt
- Teile-und-herrsche-Prinzip
 - Vereinfachung des Problems durch Zerlegung
 - engl.: divide-and-conquer
 - lat.: divide-et-impera
 - Historisch falsch: Teile werden nicht gegeneinander ausgespielt

Rekursive Prozeduren

- Umkehrung der Berechnungsreihenfolge durch Rekursion
- Beispiel: erzeuge Binärdarstellung einer Zahl
 - 1. Versuch: Ziffern werden in falscher Reihenfolge ausgegeben

```
while n > 0:
```

```
    print n%2,
```

```
    n /= 2
```

- rekursive Lösung: gib zuerst die höherwertigen Bits aus, danach das letzte:

```
def writeBin(n):
```

```
    if (n < 2):
```

```
        print n,
```

```
    else:
```

```
        writeBin(n/2)
```

```
        print n%2,
```

Die Türme von Hanoi

- Auf einem Stapel liegen N Scheiben verschiedener Durchmesser; der Durchmesser nimmt von unten nach oben schrittweise ab.
- Der Turm steht auf einem Platz A und soll zu einem Platz C bewegt werden, wobei ein Platz B als Zwischenlager benutzt werden kann.
- Dabei müssen 2 Regeln eingehalten werden:
 - Es darf immer nur eine Scheibe bewegt werden
 - Es darf nie eine größere auf einer kleineren Scheibe liegen

Die Türme von Hanoi (2)

- Lösungsstrategie: induktive Lösung
 - Verschieben einer Scheibe: Scheibe von Platz 1 (z.B. A) auf Platz 2 (z.B. C)
 - Verschieben von K Scheiben: Verschiebe K-1 Scheiben von Platz 1 auf Hilfsplatz H (etwa: B), verschiebe K-te Scheibe von Platz 1 auf Platz 2, verschiebe K-1 Scheiben von H auf Platz 2
- Rekursive Sicht:
 - Angenommen, wir können bereits K-1 Scheiben verschieben, dann wissen wir auch, wie wir K Scheiben verschieben
 - Wir wissen, wie wir eine Scheibe verschieben (Rekursionsende)
 - Problem: Keine feste Zuordnung von symbolischen Plätzen (1, 2, H) zu tatsächlichen Plätzen (A, B, C)
 - Lösung: symbolische Plätze sind Variablen/Parameter, tatsächliche Plätze die Werte von dieser Variablen

Die Türme von Hanoi (3)

```
def ziehe_scheibe(nummer, von, nach):  
    print "Scheibe", nummer, " wird von", von, "nach", nach, "verschoben"
```

```
def hanoi(N, platz1, hilfsplatz, platz2):  
    if N == 1:  
        ziehe_scheibe(N, platz1, platz2)  
    else:  
        hanoi(N-1, platz1, platz2, hilfsplatz)  
        ziehe_scheibe(N, platz1, platz2)  
        hanoi(N-1, hilfsplatz, platz1, platz2)
```

```
hanoi(4, "A", "B", "C")
```

Backtracking

- Weitere Verwendung rekursiver Prozeduren: Spielstrategien
 - Annahme: Spiel mit vollständiger Information (alle Konsequenzen eines Zugs sind vorhersehbar), z.B. Schach, Go, ...
- Idee: „In Gedanken“ wird das Spiel zuende gespielt und versucht, jeweils den optimalen Zug zu ziehen
- im aktuellen Spielstand werden „in Gedanken“ der Reihe nach alle möglichen Züge ausprobiert
- mehrere mögliche Ergebnisse:
 - egal welchen Zug man spielt, man gewinnt immer
 - Ergebnis: Man wird gewinnen, ein beliebiger Zug ist geeignet
 - egal welchen Zug man spielt, man verliert immer
 - Ergebnis: entsprechend
 - Bei manchen Zügen wird man gewinnen, bei manchen verlieren
 - Ergebnis: man wähle einen Zug, bei dem man gewinnen wird, und verbuche das als „man wird gewinnen“
- Algorithmus sucht der Reihe nach alle Varianten ab, bis er einen Zug gefunden hat, der zum Sieg führen wird
 - anderenfalls nimmt man den Zug „in Gedanken“ zurück, und probiert einen anderen: Backtracking

Wechselseitige Rekursion

- in der Definition der Funktion f wird die Funktion g aufgerufen, und in der Definition von g wird f aufgerufen
 - Konsequenz: Verwendung im Programm textuell vor Definition
 - Python, Java: Reihenfolge der Definitionen irrelevant
 - C, Pascal: Vorwärtsdeklarationen
- Beispiel

```
def gerade(n):  
    if n == 0:  
        return True  
    return ungerade(n-1)
```

```
def ungerade(n):  
    if n == 0:  
        return False  
    return gerade(n-1)
```

Allgemeine Rekursion

- Definition der Funktion greift mehrfach auf dieselbe Funktion zurück
- Beispiel: Fibonacci-Funktion
 - Modell zur Populationsentwicklung z.B. bei Kaninchen

$$fib(n) = \begin{cases} 1, & \text{falls } n \leq 1 \\ fib(n-1) + fib(n-2), & \text{sonst.} \end{cases}$$

Endrekursion

- Eine Funktionsdefinition ist **endrekursiv** (*tail recursive*), falls sie die Form hat

$$f(x) = \begin{cases} g(x), & \text{falls } P(x) \\ f(r(x)), & \text{sonst.} \end{cases}$$

- Beispiel:

$$gerade_1(n) = \begin{cases} (n = 0), & \text{falls } n \leq 0 \\ gerade_1(n - 2), & \text{sonst.} \end{cases}$$

Endrekursion (2)

- Falls dem rekursiven Aufruf noch eine Berechnung folgt, liegt keine Endrekursion vor:

$$gerade_2(n) = \begin{cases} (n = 0), & \text{falls } n < 1 \\ \neg gerade_2(n - 1), & \text{sonst.} \end{cases}$$

- Bei Endrekursion kann der ursprüngliche Aufruf gänzlich ersetzt werden:
 - $gerade_1(4) = gerade_1(2) = gerade_1(0) = (0=0) = \text{True}$
 - aber: $gerade_2(4) = \neg gerade_2(3) = \neg \neg gerade_2(2) = \neg \neg \neg gerade_2(1) = \neg \neg \neg \neg gerade_2(0) = \neg \neg \neg \neg \neg \neg \text{True} = \dots = \text{True}$
- Endrekursive Definition kann leicht in iterative Definition übertragen werden
 - In manchen Programmiersprache (z.B. LISP, Scheme) passiert das automatisch

Lineare Rekursion

- Verallgemeinerung der Endrekursion
 - für $h(x,y)=y$ ergibt sich Endrekursion

$$f(x) = \begin{cases} g(x), & \text{falls } P(x) \\ h(x, f(r(x))), & \text{sonst} \end{cases}$$

- $f(x) = h(x, f(r(x)))$
 $= h(x, h(r(x), f(r^2(x))))$
 $= h(x, h(r(x), h(r^2(x), f(r^3(x))))))$
 $= \dots$

Lineare Rekursion (2)

- im Allgemeinen Berechnung nur durch Stack s möglich
 - Berechnung von $r^n(x)$, Speichern auf Stack, Auslesen in umgekehrter Reihenfolge (LIFO: Last-In-First-Out)

while not $P(x)$:

 push(x, s)

$x = r(x)$

$f = g(x)$

while not empty(s):

$x = \text{top}(s)$

 pop(s)

$f = h(x, f)$

Lineare Rekursion (3)

- Lineare Rekursion lässt sich u.U. in Endrekursion umformulieren (siehe Gumm, Sommer, S. 168):

- Beispiel: Fakultät $fac(n)=n!$

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

- Ersetzbar durch $fac(n) = facAux(n, 1)$

$$facAux(n, a) = \begin{cases} a, & \text{falls } n = 0 \\ facAux(n - 1, n \times a), & \text{sonst.} \end{cases}$$

- zusätzlicher Parameter übernimmt Rolle des *Akkumulators*

Transformation von Fibonacci-Zahlen

- Akkumulatorverfahren u.U. erweiterbar auf allgemeine Rekursion
- Beispiel: $\text{fib}(n) = \text{fibAux}(n, 1, 1)$ mit

```
def fibAux(n, acc1, acc2):
```

```
  if n == 0:
```

```
    return acc1
```

```
  return fibAux(n-1, acc2, acc1+acc2)
```

- Iterative Definition: Auflösen der Endrekursion:

```
def fib(n):
```

```
  acc1 = acc2 = 1
```

```
  while n > 0:
```

```
    n, acc1, acc2 = n-1, acc2, acc1+acc2
```

```
  return acc1
```