Informatik I

5. Bedingungen, bedingte Ausführung, While-Schleifen und Rekursion

Bernhard Nebel

Albert-Ludwigs-Universität Freiburg

29. Oktober 2013

Informatik I

29. Oktober 2013 — 5. Bedingungen, bedingte Ausführung, While-Schleifen und Rekursion

5.1 Bedingungen und der Typ `bool`

5.2 Bedingte Anweisungen

5.3 While-Schleifen

5.4 Rekursion

5.1 Bedingungen und der Typ `bool`

Der Typ bool

- Neben *arithmetischen Ausdrücken* gibt es noch Boolesche Ausdrücke, die den Wert `True` oder `False` haben können.
- Die einfachsten Booleschen Ausdrücke sind Vergleiche mit dem Gleichheitsoperator ==.
- Die Werte `True` und `False` gehören zum Typ `bool` und werden automatisch nach `int` konvertiert:

Python-Interpreter

```
>>> 42 == 42
True
>>> 'egg' == 'spam'
False
>>> type('egg' == 'spam')
<class 'bool'>
>>> True + True
2
```

Vergleichsoperatoren

- Es gibt die folgenden Vergleichsoperatoren:

symbolisch	Bedeutung
`x == y`	Ist x gleich y?
`x != y`	Ist x ungleich y?
`x > y`	Ist x echt größer als y?
`x < y`	Ist x echt kleiner als y?
`x >= y`	Ist x größer oder gleich y?
`x <= y`	Ist x kleiner oder gleich y?

- Strings werden anhand der lexikographischen Ordnung verglichen, wobei für Einzelzeichen der Unicode-Wert (Ergebnis der `ord`-Funktion) benutzt wird.
- Werte unvergleichbarer Typen sind ungleich. Bei den Anordnungsrelationen gibt es einen Fehler!

Vergleichsoperatoren in Aktion

Python-Interpreter

```
>>> 'spamer' < 'spam'
False
>>> 'Spam' < 'spam'
True
>>> 2.1 - 2.0 == 0.1
False
>>> False < True
True
>>> 42 == 'zweiundvierzig'
False
>>> 41 < '42'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError:  unorderable types:  int() < str()
```

Logische Operatoren

- Es gibt die folgenden logischen Operatoren: `or`, `and`, `not` – mit aufsteigender Operatorpräzedenz.
- Bedeutung wie in Boolescher Logik, d.h.
  - `x < 10 or y > 100` hat den Wert `True`, wenn x kleiner als 10 ist, oder falls das nicht der Fall ist, wenn y größer als 100 ist.
  - `1 <= x and x <= 10` hat den Wert `True`, wenn x zwischen 1 und 10 (inklusive) liegt.
  - Dies kann in Python auch so geschrieben werden (wie in mathematischer Notation): `1 <= x <= 10`.
  - `not(x < y)` ist `True` wenn $x \geq y$ ist.
- Alle Nullwerte, d.h. `None`, `0`, `0.0`, `(0 + 0j)` und '', werden wie `False` behandelt, alle anderen Werte wie `True`.
- Die Auswertung wird beendet, wenn das Ergebnis klar ist (Unterschied bei Seiteneffekten und Werten äquivalent zu `True`).

Logische Operatoren in Aktion

Python-Interpreter

```
>>> 1 < 5 < 10
True
>>> 5 < 1 or 'spam' < 'egg'
False
>>> 'spam' or True
'spam'
>>> '' or 'default'
'default'
>>> 'egg' and 'spam'
'spam'
>>> 0 and 10 < 100
False
>>> not 'spam' and (None or 0.0 or 10 < 100)
False
```

5.2 Bedingte Anweisungen

Bedingte Ausführung

- Bisher wurde jede eingegebene Anweisung ausgeführt.
- Manchmal möchte man aber eine Anweisung oder einen Anweisungsblock nur unter bestimmten Bedingungen ausführen: `if`-Anweisung.

Python-Interpreter

```
>>> x = 3
>>> if x > 0:
...     print('x ist strikt positiv')
...
x ist strikt positiv
>>> x = 0
>>> if x > 0:
...     print('x ist strikt positiv')
...
>>>
```

If-else

- Möchte man im positiven und im negativen Fall etwas machen: `if-else-Anweisung`.

Python-Interpreter

```
>>> x = 3
>>> if x%2 = 0:
...     print('x ist gerade')
... else:
...     print('x ist ungerade')
...
x ist ungerade
```

- Soll ein Anweisungsblock leer bleiben, kann man dafür `pass` einsetzen.

Verkettete bedingten Anweisungen

- Will man mehrere Fälle behandeln, gibt es die verketteten bedingten Anweisungen

Python-Interpreter

```
>>> x = 3
>>> y = 0
>>> if x < y:
...     print('x ist kleiner als y')
... elif x > y:
...     print('x ist größer als y')
... else:
...     print('x und y sind gleich')
...
x ist größer als y
```

- Es wird immer der Block ausgeführt, bei dem die Bedingung das erste Mal wahr wird.

Geschachtelte Konditionale

- Man kann auch bedingte Anweisungen als Block in bedingten Anweisungen unterbringen.

Python-Interpreter

```
>>> x = 100
>>> if x > 0:
...     if x < 10:
...         print('kleine positive Zahl')
...  else:
...         print('negative Zahl')
...
>>>
```

- Durch Einrückung ist immer klar, wozu die bedingte Anweisung gehört!

5.3 While-Schleifen

While-Schleifen

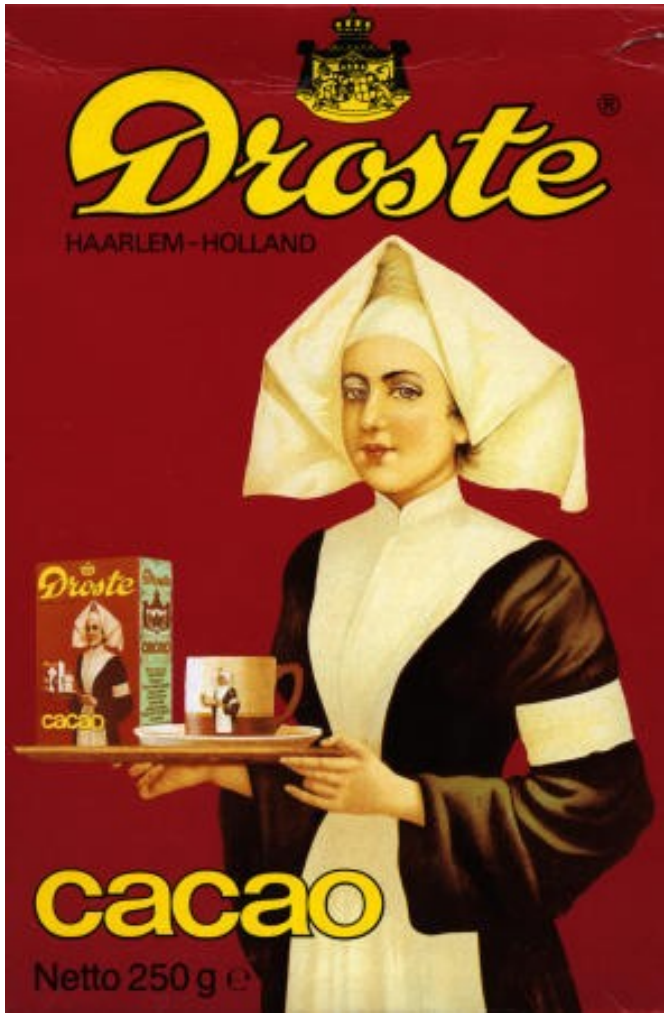
- Oft muss etwas wiederholt gemacht werden, bis ein bestimmter Wert erreicht wird.
- Hier bentzt man gerne eine `while`-Schleife.
- Wir wollen alle Zahlen von 0 bis n aufsummieren: $\sum_{i=0}^{n} i$.

Python-Interpreter

```
>>> def sum(n):
...     i = 0
...     result = 0
...     while i <= n:
...         result = result + i
...         i = i + 1
...     return result
...
>>> sum(100)
5050
```

5.4 Rekursion

Rekursion verstehen



Um Rekursion zu verstehen, muss man zuerst einmal Rekursion verstehen.

Abb. in Public Domain, Quelle Wikipedia

Rekursion als Definitionstechnik: Fakultätsfunktion

- Bei einer rekursiven Definition wird das zu Definierende unter Benutzung desselben (normalerweise in einer einfacheren Version) definiert.
- Beispiel Fakultätsfunktion
  - Auf wie viele Arten kann man n Dinge sequentiell anordnen?
  - Berechne, auf wie viele Arten man $n-1$ Dinge anordnen kann. Für jede dieser Anordnungen können wir das „letzte“ Ding auf n Arten einfügen.
  - D.h. wir können die Fakultätsfunktion $n!$ wie folgt definieren:

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

  - Berechne 4!:

$$4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2! = 4 \cdot 3 \cdot 2 \cdot 1! = 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! = 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 24$$

Fakultätsfunktion in Python

- Wir können in Funktionsdefinitionen bisher undefinierte (z.B. die gerade zu definierende) Funktion benutzen:

Python-Interpreter

```
>>> def fak(n):
...     if n == 0:
...         return 1
...     else:
...         return n*fak(n-1)
...
>>> fak(4)
24
>>> fak(50)
30414093201713378043612608166064768844377641568960512000000000000
```

Rekursive Aufrufe

- Was passiert genau?

Aufrufsequenz

→ `fak`(4) wählt else-Zweig und ruft auf:
→ `fak`(3) wählt else-Zweig und ruft auf:
→ `fak`(2) wählt else-Zweig und ruft auf:
→ `fak`(1) wählt else-Zweig und ruft auf:
→ `fak`(0) wählt if-Zweig und:
← `fak`(0) gibt 1 zurück
← `fak`(1) gibt 1 zurück
← `fak`(2) gibt 2 zurück
← `fak`(3) gibt 6 zurück
← `fak`(4) gibt 24 zurück

Tracing ...

- Wenn man die rekusiven Aufrufe nachvollziehen will, kann man auch Kontrollausgaben in die Funktion einbauen:

Python-Interpreter

```
>>> def fak(n):
...     space = ' ' * (2 * n)
...     print(space, 'fak(', n, ')')
...     if n == 0:
...         print(space, 'fak(', n, '):', 1)
...         return 1
...     else:
...         recurse = fak(n-1)
...         result = n * recurse
...         print(space, 'fak(', n, '):', result)
...         return result
...
```

Die Berechnung . . .

Python-Interpreter

```
>>> fak(4)

          fak( 4 )
        fak( 3 )
      fak( 2 )
    fak( 1 )
  fak( 0 )
  fak( 0 ): 1
    fak( 1 ): 1
      fak( 2 ): 2
        fak( 3 ): 6
          fak( 4 ): 24
24
```

Endrekursion

- Die rekursive Definition der Fakultätsfunktion ist eine sog. Endrekursion (engl. tail recursion): Der rekursive Aufruf erfolgt am Ende.
- Solche Endrekursionen können einfach in Iterationen umgewandelt werden (vgl. Definition der sum-Funktion):

Python-Interpreter

```
>>> def fak(n):
...     i = 1
...     result = 1
...     while i <= n:
...         result = result * i
...         i = i + 1
...     return result
...
```

Fibonacci-Folge: Das Kanninchenproblem

- Manchmal sind kompliziertere Formen der Rekursion notwendig, z.B. bei der Definition der Fibonacci-Folge
- Eingeführt von *Leonardo da Pisa*, genannt *Fibonacci* in seinem Buch *Liber abbaci* (1202), das u.a. die arabischen Ziffern und den Bruchstrich in die westliche Welt einführten.
- Anzahl der Kannichen-Paare, die man erhält, wenn jedes Paar ab dem zweiten Monat ein weiteres Kannichen-Paar erzeugt (und kein Kannichen stirbt). Wir beginnen im Monat 0, in dem das erste Paar geboren wird:

Monat	vorhanden	geboren	gesamt
0.	0	1	1
1.	1	0	1
2.	1	1	2
3.	2	1	3
4.	3	2	5

Fibonacci-Zahlen

- Die Fibonacci-Zahlen werden normalerweise wie folgt definiert (und beschreiben damit die vorhandenen Kanninchen-Paare am Anfang des Monats):

$$\texttt{fib}(n) = \begin{cases} 0, & \text{falls } n = 0; \\ 1, & \text{falls } n = 1; \\ \texttt{fib}(n-1) + \texttt{fib}(n-2), & \text{sonst.} \end{cases}$$

- D.h. die Folge beginnt mit 0 und nicht mit 1.
- Beachte: Hier gibt es zwei rekursive Verwendungen der Definition.
- Die Fibonacci-Zahlen spielen in vielen anderen Kontexten eine wichtige Rolle (z.B. Goldener Schnitt).

Fibonacci-Zahlen in Python

- Umsetzung in Python folgt direkt der mathematischen Definition:

Python-Interpreter

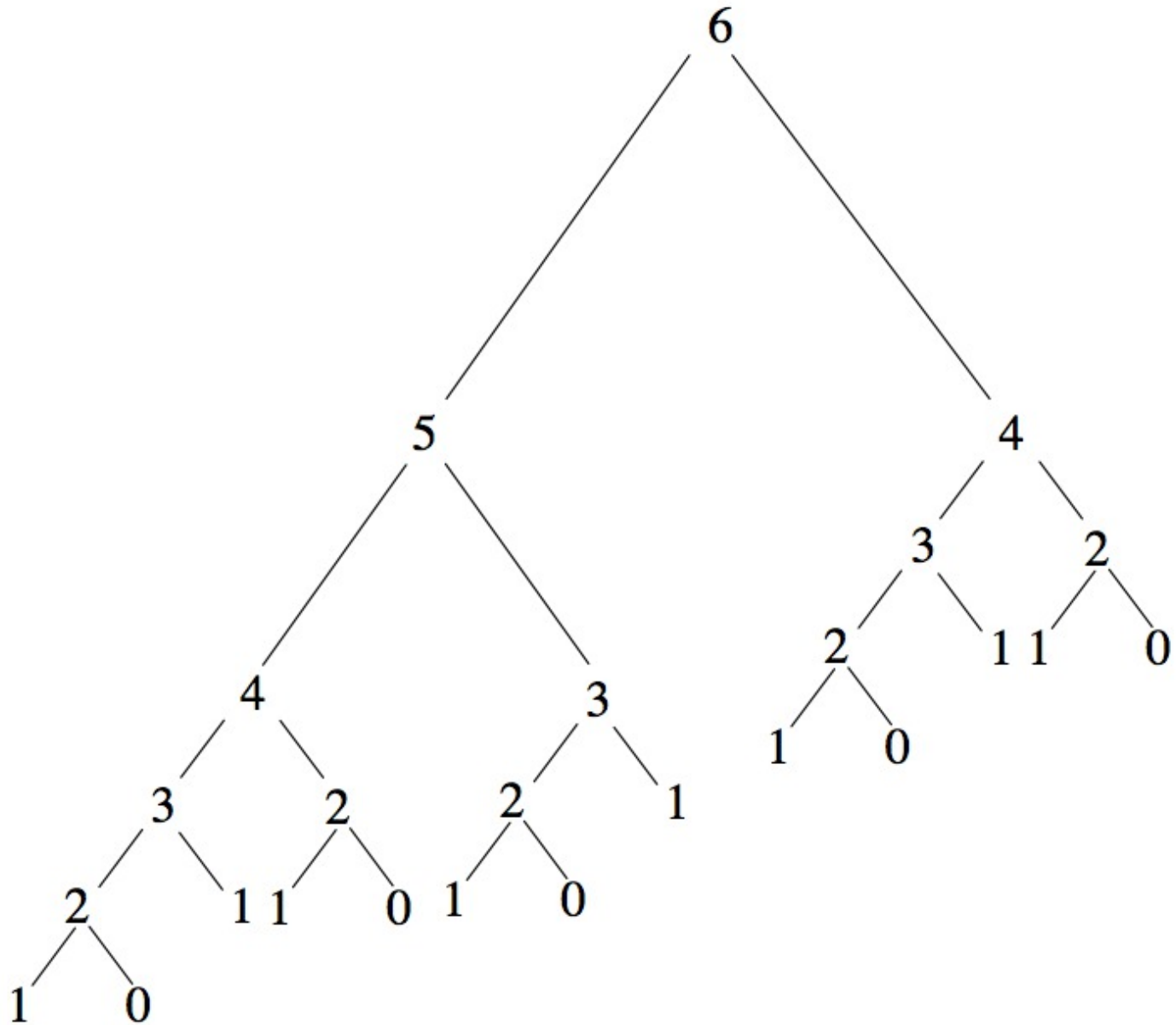
```
>>> def fib(n):
...     if n == 0:
...         return 0
...     elif n == 1:
...         return 1
...     else:
...         return fib(n-1)+fib(n-2)
...
>>> fib(35)
9227465
```

Der Aufrufbaum

Aufrufbaum

→ `fib`(3) wählt else-Zweig und ruft auf:
| → `fib`(2) wählt else-Zweig und ruft auf:
| | → `fib`(1) wählt elif-Zweig und
| | ← `fib`(1) gibt 1 zurück
| `fib`(2) ruft jetzt auf:
| | → `fib`(0) wählt if-Zweig und
| | ← `fib`(0) gibt 0 zurück
| ← `fib`(2) gibt 1 zurück
`fib`(3) ruft jetzt auf:
| → `fib`(1) wählt elif-Zweig und
| ← `fib`(1) gibt 1 zurück
← `fib`(3) gibt 2 zurück

Aufrufbaum graphisch



Komplexe Rekursion: Verständnis und Laufzeit

- Es gibt komplexere Formen der Rekursion: mehrfach, *indirekt*, *durch Argumente*.
- Es ist nicht ganz einfach, den Verlauf der Ausführung der `fib`-Funktion nachzuvollziehen.
- Dies ist aber auch nicht notwendig! Es reicht aus, sich zu vergegenwärtigen, dass:
  - falls die Funktion alles richtig macht für Argumente mit dem Wert $< n$,
  - dann berechnet sie das Geforderte

  $\rightarrow$ Prinzip der vollständigen Induktion
- Die mehrfachen rekursiven Aufrufe führen zu sehr hoher Laufzeit!
- Auch hier wäre eine iterative Lösung (`while`-Schleife) möglich.
- Rekursive Definition vor allen Dingen sinnvoll auf rekursiven Datenstrukturen.