Vorlesungsmodul Nichtprozedurale Programmierung
- VorlMod NpProg -

Matthias Ansorg

02. Oktober 2003 bis 22. Januar 2004

Zusammenfassung

Studentische Mitschrift zur Vorlesung Nichtprozedurale Programmierung bei Prof. Dr. Thomas Letschert (Wintersemester 2003 / 2004) im Studiengang Informatik an der Fachhochschule Gießen-Friedberg.

- **Bezugsquelle:** Die vorliegende studentische Mitschrift steht im Internet zum Download bereit. Quelle: Persönliche Homepage Matthias Ansorg :: InformatikDiplom http://matthias.ansorgs.de/InformatikAusbldg/.
- **Lizenz:** Diese studentische Mitschrift ist public domain, darf also ohne Einschränkungen oder Quellenangabe für jeden beliebigen Zweck benutzt werden, kommerziell und nichtkommerziell; jedoch enthält sie keinerlei Garantien für Richtigkeit oder Eignung oder sonst irgendetwas, weder explizit noch implizit. Das Risiko der Nutzung dieser studentischen Mitschrift liegt allein beim Nutzer selbst. Einschränkend sind außerdem die Urheberrechte der angegebenen Quellen zu beachten.
- **Korrekturen und Feedback:** Fehler zur Verbesserung in zukünftigen Versionen, sonstige Verbesserungsvorschläge und Wünsche bitte dem Autor per e-mail mitteilen: Matthias Ansorg <mailto:matthias@ansorgs.de>.
- **Format:** Die vorliegende studentische Mitschrift wurde mit dem Programm LyX(graphisches Frontend zu LaTeX) unter Linux geschrieben und mit pdfLaTeXals pdf-Datei erstellt. Grafiken wurden mit dem Programm xfig unter Linux erstellt und als pdf-Dateien exportiert.
- **Dozent:** Prof. Dr. Thomas Letschert.
- **Verwendete Quellen:** <quelle> {<quelle>}.
- **Klausur:**
  - Es gibt keine Hausübungen, Bonusaufgaben oder sonstigen formalen Klausurvoraussetzungen.
  - Es dürfen alle Unterlagen als Hilfsmittel in der Klausur verwendet werden, u.a. also das Skript [1].
  - Die Übungsaufgaben dienen als Klausurvorbereitung und werden in den Übungen besprochen.
  - Die Klausur enthält zu großen Teilen Programmieraufgaben.

Inhaltsverzeichnis

1 Einführung

Es wird zu weiten Teilen »nichtprozedural« programmiert. Konkret: für die Klausur werden Grundkenntnisse in C++ und Python vorausgesetzt. Python hat dabei Konstrukte zur nichtprozeduralen Programmierung, die in C++ so nicht vorkommen. Man muss sich also mit Python beschäftigen, um die Klausur bestehen zu können.

In der Klausur werden keine eleganten Python-Skripts erwartet, sondern das Grundverständnis.

Außerdem wird XSLT zur nichtprozeduralen Programmierung verwendet. Auch hier wird das Grundverständnis gelehrt, es ist kein Programmierkurs.

»Nichtprozedurale Programmierung« ist die letzte Programmierveranstaltung im Studium, es gibt also auch etwas Philosophie der Programmierung.

1.1 Prozedurale Programmierung

Ein prozedurales Programm bedeutet: es gibt manipulierbare Speicherplätze, die von einem Programm als Sequenz von Befehlen manipuliert. Es ist die Beschreibung einer virtuellen Maschine, die aus einer Serie von Speicherplätzen besteht und aus Befehlen, die die Inhalte der Speicherplätze manipulieren.

1.2 Deklarative Programmierung

Bei der prozeduralen Programmierung ist eine »Variable« ein benamter Speicherplatz, der einen veränderbaren Wert enthält. Die Abbildung von einem Namen auf einen Wert ist also indirekt, zweistufig über den Speicherplatz. In der Mathematik und in der nichtprozeduralen Programmierung dagegen besteht eine direkte Abbildung eines Namens auf einen festen Wert. Auch in der objektorientierten Programmierung gilt dieses Konzepts

Im Paradigma der nichtprozeduralen Programmierung gibt es also nichts, was man als Änderung eines benamten Wertes deuten könnte - die Werte sind fest.

Während die prozedurale Programmierung Programme als Beschreibung des Verhaltens der Maschine auffasst, geht die nichtprozedurale Programmierung den umgekehrten Weg: sie fasst Programme als Beschreibung des Problems und seiner Lösung auf. Sie verwendet etwa die mathematische Notation als »Problemlösungsnotation«, ohne darüber zu informieren, wie die Problemlösung auf einer Maschine ausgeführt werden soll.

Auch die höheren prozeduralen Programmiersprachen haben Ausdrücke, die nicht direkt das Verhalten der Maschine beschreiben (etwa `v1=v1+2`), also bereits nichtprozedural sind. Sie beschreiben ja einen auszuwertenden Ausdruck, keine Befehlssequenz. Diese wird vom Compiler ermittelt. FORTRAN war die erste Programmiersprache, die solche mathematische, nichtprozedurale Notation einführte. Daher die Abkürzung für »FORmula TRANslator«. Alle Programmiersprachen außer Assembler enthalten also nichtprozedurale Elemente; die beiden Paradigmen sind keine Gegensätze!

1.2.1 Warum Nichtprozedurale Programmierung?

Historie der Paradigmen:

Späte 50er

- FORTRAN von IBM: eine prozedurale Sprache. Das Programm ist eine Folge von Befehlen zur Manipulation von Speicherstellen. Es gibt Ausdrücke als funktionale Elemente der nichtprozeduralen Programmierung. FORTRAN war eine sehr pragmatische Entwicklung, ohne wissenschaftlichen Hintergrund.
- Der Wissenschaftler McCarthy erfand LISP. Er hielt die prozedurale Programmierung a la FORTRAN für primitiv. Er wollte Wissen verarbeiten, Intelligenz auf Rechner abbilden. Allerdings war die Hardware bisher lächerlich. Die Ideen hinter LISP:
  - maschinenunabhängige Problembeschreibungssprache mit mathematischer Notation. In richtigem LISP gibt es keine Anweisungen und Schleifen, denn es gibt kein Konzept »Speicherplatz«, sondern »benamte Werte«.
  - Datenstrukturen, in LISP[1] die »Liste«, ein Binärbaum. Eine Erfindung von McCarthy. Bisher gab es das Konzept »Datenstrukur« nicht! Dies fand als Synergieeffekt Eingang in die prozeduralen Programmiersprachen.
  - LISP soll Programme als Daten verarbeiten können. Jedes Programm ist ein Wert, das von einem anderen Programm verarbeitet werden kann. Jede Funktion in LISP, etwa `(define fx (+xx))` ist ein solches Programm! Diese Idee, dass Programme sich selbst verändern, findet die Softwaretechnik heute schwachsinnig.

80er

- Japan war führende Industrienation. Die Japaner wollten »Computer der fünften Generation« produzieren; diese sollten Künstliche Intelligenz besitzen und alle Probleme dieser Welt lösen. Ziel war, dass die Japaner die »Weltherrschaft über die Informatik« mit Hilfe der KI haben wollten. Als Reaktion wurden auch in den USA und Europa Institute für KI gegründet. Hier wurden nichtprozedurale Programmiersprachen eingesetzt, beides war allein aufgrund der Geschichte miteinander verknüpft.
- Software-Krise. Entdeckt 1965: »Programme funktionieren nicht«. Weil sie schlecht geschrieben und falsch sind. Damalige Lösung: weil sie nicht wissenschaftlich aufgebaut sind. Alles nichtmathematische wurde deshalb entfernt, allem voran die indirekte Abbildung von Name zu Wert über den Speicherplatz, damit auch die Zuweisungen und alle Anweisungen und alle Schleifen. Übrig blieb die nichtprozedurale Programmierung. Ziel war es, dann alle Programme als korrekt beweisen zu können. So aber funktioniert die Mathematik nicht: Beweise sind Geistesblitze, keine Produkte. Komplizierte Programme kann die Mathematik also im Normalfall nicht beweisen.

Nichtprozedurale Programmierung enthält interessante Programmkonstrukte. Das ist ein Hauptgrund, sich mit nichtprozeduraler Programmierung zu beschäftigen. Die interessanten Konzepte sind:

- Funktionen
- Funktionen erzeugende Funktionen
- Generatoren und Iteratoren
- Mustererkennung und Transformationsmuster
- Typfreiheit, deshalb am besten durch einen Interpreter bearbeitet

[1] List Processing Language

2 Ausdrücke

2.1 Interpretierte Sprachen

2.2 Namen

2.3 Ausdrücke mit Funktionen

In der funktionalen Welt gibt es n Kategorien, deren Elemente nicht gegeneinander austauschbar sind:

- Werte. Ein Wertausdruck ist äquivalent zu einem Wert. Ein Wertausdruck ist der Aufruf einer Funktion beliebiger Ordnung, deren Rückgabewert ein Wert ist.
- Funktionen 1. Ordnung. Es sind keine Werte, sondern Abbildungsvorschriften für beliebige Werte. Ausdrücke, die zu Funktionen 1. Ordnung ausgewertet werden, sind äquivalent mit diesen.
- Funktionen 2. Ordnung. Es sind keine Funktionen 1. Ordnung, sondern Abbildungsvorschriften für beliebige Funktionen 1. Ordnung. Sie erfüllen mindestens eine der folgenden Bedingungen:
  - Ihr Rückgabewert ist eine Funktion erster Ordnung. So beim Verkettungsfunktional »∘« oder auch bei folgender Funktion in Python:

    ```
    def funktional(x):
      return lambda y: y+x
    ```

  - Mindestens ein Argument ist eine Funktion erster Ordnung. Die in Python vordefinierte Funktion `map` ist solch ein Funktional: das erste Argument muss eine Funktion erster Ordnung sein, das zweite eine Werteliste, der Rückgabewert ist eine Werteliste.
- ...
- Funktionen n. Ordnung. Es sind keine Funktionen $n-1$. Ordnung, sondern Abbildungsvorschriften für beliebige Funktionen $n-1$. Ordnung.

3 Funktionen in OO-Sprachen

3.1 Funktionen

3.2 Funktionen als Parameter

3.3 Funktionale Objekte

3.4 Dynamisch erzeugte Funktionen

Im Beispiel im Skript soll die Verkettung von Funktionen emuliert werden entsprechend:

```
def compose(f, g) return lambda x : g(f(x))
```

Implementiert ist dann aber versehentlich eine Emulation von `f(g(x))`, also nicht der Verkettung $f \circ g = g(f(x))$!

3.5 Konstruktion funktionaler Objekte

4 Rekursion

4.1 Rekursion, Induktion, Iteration

4.2 Rekursive Daten und Rekursive Funktionen

Ein Fehler im folgenden Codebeispiel:

```
def mult(x,y):
  if x == 0:
    return x // FEHLER: muss sein: return 0
  else:
    return mult (x-1, y) + y
```

4.3 Klassifikation der Rekursion

4.4 Entwicklung Rekursiver Programme

S. 62: cons(a,l) muss cons(x,l) sein.

5 Übungsaufgaben

5.1 Kopiersemantik einer Liste in C++

Blatt 1, Aufgabe 1, Punkt 9

5.2 Warum vollständig geklammerte Ausdrücke und Prefix-Notation in LISP?

Aufgabe »Warum verwendet LISP vollständig geklammerte Ausdrücke und Prefix Notation?« Quelle:

5.3 Statische und dynamische Bindung

Aufgabe »Welche Ausgabe erzeugt folgendes Python Programm:

```
i = 3
def fun(x):
  return i+x

l = [1, 2, 3, 4, Boo , 5]

for i in l:
  print fun(i)
```

Erläutern Sie mit diesem Beispiel im Vergleich zu einem äquivalenten C++-Programm die Konzepte der statischen (C++) und der dynamischen (Python) Bindung. Definieren Sie diese Begriffe. Was versteht man unter statisch und dynamisch im Bereich der Programmiersprachen? Ist die beobachtete dynamische Bindung mit dem Sprachkonzept von Python vereinbar?«[2, Blatt 2, Aufg. 2, Teil 4]

Lösung Das Python-Programm erzeugt die Ausgabe:

```
2
4
6
8
BooBoo
10
```

Statische Bindung liegt vor, wenn eine freie Variable an den Wert bzw. Speicherplatz gebunden ist, den sie an der Definitionsstelle hat. Dynamische Bindung liegt vor, wenn eine freie Variable an den Wert bzw. Speicherplatz gebunden wird, den sie an der Aufrufstelle hat [1, Kap. 2.2, S. 18]. »Statisch« bedeutet hier »zur Übersetzungszeit«, »dynamisch« bedeutet »zur Laufzeit«.

In funktionalen Sprachen wird an Werte gebunden, in prozeduralen Sprachen an Speicherplätze. Python soll hier als Vertreter einer funktionalen Sprache gelten: das Beispielprogramm nutzt keine prozeduralen Elemente wie die Möglichkeit, Objekte zu ändern. C++ soll hier als Vertreter einer prozeduralen Sprache gelten.

Wäre die Bindung an Werte in Python statisch, so würde `i` in `fun(x)` immer den Wert haben, den es an der Definitionsstelle der Funktion hat: 3. Das ist offensichtlich nicht der Fall. Sondern `i` wird so ausgewertet als würde man einfach den Aufruf durch den Funktionsrumpf ersetzen. Die Bindung von `i` an Werte ist dynamisch.

Nun zu einem äquivalenten C++-Programm. In C++ werden Bezeichner nicht an Werte, sondern an Speicherplätze gebunden. Das äquivalente Programm muss also einen Bezeichner `i` enthalten, der im Programmtext einmal an einen, dann an einen anderen Speicherplatz gebunden ist. Das ist mit einer einzigen Variable natürlich nicht machbar, weshalb wir eine zweite, lokale Variable einführen müssen, die sie überdeckt:

```
#include <iostream>
using namespace std;

int i=3; //globales i
int fun(int x) {
  return i+x;
}

int main() {
  for (int i=1; i<6; ++i) { //lokales i
    cout << fun(i) << endl;
  }
  return 0;
}
```

Durch die statische Bindung in C++ bezieht sich `i` in der Definition von `fun(x)` stets auf den Speicherplatz, den `i` an der Definitionsstelle meint: das globale `i`. Wir erhalten daher die Ausgabe:

```
4
5
6
7
8
```

Achtung: Die Bindung von Bezeichnern an Speicherplätze ist in C++ zwar statisch. Die Zuordnung von Speicherplätzen zu Werten jedoch dynamisch. Deshalb ist die Zuordnung von Bezeichnern zu Werten in C++ ebenso dynamisch wie die Bindung von Bezeichnern an Werte in Python. Wir sprechen bei dieser zweistufigen Zuordnung jedoch nicht mehr von Bindung.

`i` in der Definition von `fun(x)` ist also stets an denselben Speicherplatz gebunden, aber nicht an denselben Wert! Das zeigt sich, wenn wir den Wert des globalen `i` verändern, indem wir die `for`-Schleife schreiben als:

```
for (i=1; i<6; ++i) { //globales i
    cout << fun(i) << endl;
}
```

Nun entsteht dieselbe Ausgabe wie beim Python-Code.

Ist nun die dynamische Bindung mit dem Sprachkonzept von Python vereinbar? Ja, denn es passt zu Python als einer interpretierten Sprache, in der alles zur Laufzeit geschieht. Sie verträgt sich auch, genauso aber die statische Bindung, zu funktionalen Sprachen bzw. Sprachelementen generell.

5.4 Generische Listen in C++, Python und Java

Quelle: [2, Blatt 2, Aufg. 3].

Kommentar zum C++-Quellcode: Warum gibt es einen leeren Konstruktor der Basisklasse? Wichtig ist nicht, dass der Konstruktor eine leere Implementierung hat, er könnte vielleicht auch rein virtuell sein. Wichtig ist, dass er virtuell ist, also polymorph. Wenn Objekte abgeleiteter Klassen über Basisklassenzeiger manipuliert werden und dann `delete` aufgerufen wird, so wird durch die Polymorphie der Destruktor der abgeleiteten Klasse statt nur der Destruktor der Basisklasse aufgerufen. Vergleiche [4, Kap. 8.3.2, S. 213].

5.5 Wirkung eines Python-Programms für `[1, 'Hallo', 3]`

Aufgabe »Welche Wirkung hat folgendes Programm bei der Eingabe von `[1, 'Hallo', 3]`:«

```
def f(x):
  return x+x

def g(x):
  return 3*x
```

```
list = input()
for i in list:
  print ( lambda x : (lambda p, q : p(x)+q(x))(f, g)) (i)
```

Quelle: [2, Blatt 2, Aufg. 4, Teil 2]

Lösung

$$
\begin{aligned}
& (\lambda x.\,(\lambda p, q.p\,(x) + q\,(x))\,(f, g))\,(i) \\
\Leftrightarrow\quad & (\lambda x.f\,(x) + g\,(x))\,(i) \\
\Leftrightarrow\quad & (\lambda x.\,(\lambda x.x + x)\,(x) + (\lambda x.3 \cdot x)\,(x))\,(i) \\
\Leftrightarrow\quad & (\lambda x.\,(x + x) + (3 \cdot x))\,(i) \\
\Leftrightarrow\quad & (\lambda x.5 \cdot x)\,(i) \\
\Leftrightarrow\quad & 5 \cdot i
\end{aligned}
$$

Wie man hier sieht, kann jeder geschachtelte Funktionsaufruf vereinfacht werden zu einer nicht geschachtelten Version. Dazu beginnt man bei den innersten Funktionsaufrufen und ersetzt schrittweise jeden Funktionsaufruf durch seine Defintion, in die die Argumente symbolisch eingesetzt werden. Es entsteht der nicht geschachtelte Funktionsaufruf, den man dann noch symbolisch so weit wie möglich vereinfachen kann. In diesen können nun die konkreten Argumente (hier (f, g) bzw. (i)) eingesetzt werden. Statt Funktionsufrufen sind natürlich auch stets Lambda-Ausdrücke zulässig.

Man kann jedoch auch anders auswerten und so früh wie möglich die konkreten Argumente einsetzen, trotz dass geschachtelte Funktionsaufrufe enthalten sind. In jedem Fall wird man folgende Ausgabe ermitteln:

```
5
HalloHalloHallo
15
```

5.6 Typstruktur und Wert von Lambda-Ausdrücken

Aufgabe »Analysieren Sie die Typstruktur und berechnen Sie die Werte und Typen folgender Ausdrücke:«

$$(\lambda z.\,(\lambda y.y + z))\,(2)$$

$$(\lambda x.\,(\lambda y.\,(\lambda x.x + y))\,(x + 1))\,(5)$$

$$(\lambda f.f\,(\lambda x.x + x))\,(\lambda g.g\,(g\,(1)))$$

$$(\lambda f.f\,(\lambda x.x))\,(\lambda h.h\,(1))$$

Quelle: [2, Blatt 2, Aufg. 4, Teil 4]

Lösung Die Analyse der Typstruktur wird hier nicht explizit dokumentiert. Die korrekte Auswertung eines Ausdrucks impliziert ja, dass man richtig erkannt hat welche Argumente Funktionen und welche Werte sind. Auch der Typ der Ausdrücke ist in ihrer vereinfachten Version offensichtlich, nämlich entweder eine Funktion oder ein Wert.

$$
\begin{aligned}
& (\lambda z.\,(\lambda y.y + z))\,(2) \\
\Leftrightarrow\quad & \lambda y.y + 2
\end{aligned}
$$

$$
\begin{aligned}
& (\lambda x.\,(\lambda y.\,(\lambda x.x + y))\,(x + 1))\,(5) \\
\Leftrightarrow\quad & (\lambda x.\,(\lambda y.\,(\lambda z.z + y))\,(x + 1))\,(5) \\
\Leftrightarrow\quad & (\lambda x.\,(\lambda z.z + x + 1))\,(5) \\
\Leftrightarrow\quad & \lambda z.z + 5 + 1 \\
\Leftrightarrow\quad & \lambda z.z + 6
\end{aligned}
$$

Bei diesem Ausdruck ergibt sich ein kürzerer Weg, indem man die Argumente von außen nach innen einsetzt. Zuerst ist also x durch 5 zu ersetzen. Aber Vorsicht: x im innersten Lambda-Ausdruck ist dessen unabhängige Variable und nicht die des äußersten Lambda-Ausdrucks. Es darf also nicht durch 5 ersetzt werden.

$$\begin{aligned} & (\lambda x.\,(\lambda y.\,(\lambda x.x + y))\,(x+1))\,(5) \\ \Leftrightarrow\ & (\lambda y.\,(\lambda x.x + y))\,(5+1) \\ \Leftrightarrow\ & \lambda x.x + 6 \end{aligned}$$

$$\begin{aligned} & (\lambda f.f\,(\lambda x.x + x))\,(\lambda g.g\,(g\,(1))) \\ \Leftrightarrow\ & (\lambda g.g\,(g\,(1)))\,(\lambda x.x + x) \\ \Leftrightarrow\ & (\lambda x.x + x)\,((\lambda x.x + x)\,(1)) \\ \Leftrightarrow\ & (\lambda x.x + x)\,(1+1) \\ \Leftrightarrow\ & 2 + 2 \\ \Leftrightarrow\ & 4 \end{aligned}$$

$$\begin{aligned} & (\lambda f.f\,(\lambda x.x))\,(\lambda h.h\,(1)) \\ \Leftrightarrow\ & (\lambda h.h\,(1))\,(\lambda x.x) \\ \Leftrightarrow\ & (\lambda x.x)\,(1) \\ \Leftrightarrow\ & 1 \end{aligned}$$

5.7 Verkettungsfunktional

Aufgabe »Definieren Sie informal und in Python eine Funktion `compose`, derart, dass `compose(f, g)` eine Funktion `h` liefert mit $h\,(x) = f \circ g = f\,(g\,(x)) = \mathrm{compose}\,(f, g)\,(x)$.« Quelle: [2, Blatt 2, Aufg. 4, Teil 6].

Lösung

$$\begin{aligned} c &= \lambda p, q.p \circ q \\ &= \lambda p, q.p\,(q\,(x)) \end{aligned}$$

```
def f(x):
  return x*x

def g(x):
  return x+2

def compose(p,q):
  return lambda x: p(q(x))

print compose(f,g)(0)
```

Warum ist es notwendig, in `compose(p,q)` einen λ-Ausdruck zurückzugeben statt einfach den Ausdruck $p\,(q)$? $p\,(q)$ ist ein Ausdruck, wäre also äquivalent zu einem Wert. Wir wollen jedoch eine Funktion zurückliefern! Zusätzlich ergibt sich das Problem, dass ein Funktionsaufruf $p\,(q)$ nicht ausgewertet werden kann: p ist vom Typ »Funktion, die einen Wert erwartet«, q aber ist eine Funktion und kein Wert. Werte und Funktionen sind inkompatible Typen.

Dies ist sozusagen ein Rest von Typsicherheit selbst in funktionalen Sprachen. Auch diesen hätte man natürlich vermeiden können, man hat es aber nicht getan. Dazu hätte (bezogen auf obiges Beispiel) das Ergebnis der Multiplikation von Funktionen y, z definiert werden als eine Funktion $k = y\,(x) \cdot z\,(x)$. Ebenso wäre die Multiplikation von Funktionen ab zweiter Ordnung zu definieren. Damit wäre dann jede Funktion auch ein Funktional, aber noch lange nicht jedes Funktional eine Funktion.

Fazit: Es gibt $n + 1$ inkompatible Argumenttypen für Python-Funktionen: Werte Funktionen n. Ordnung.

5.8 Werte bei rekursivem Funktionsaufruf

Aufgabe Quelle: [2, Blatt 2, Aufgabe 5].

Lösung

1. $$(\lambda x, y.\text{if } x = 0 \text{ then } 1 \text{ else } y)\,(0, f\,(2)) \quad = \quad 1$$

   Ob dieser Wert ermittelt werden kann hängt von der Auswertungsreihenfolge ab. Würde zuerst versucht, $f\,(2)$ auszuwerten, bedeutet das einen endlosen Abstieg in Rekursionen. Das Ergebnis von $f\,(2)$ wird jedoch gar nicht benötigt, weshalb mit »lazy evaluation« hier in endlicher Zeit ein Ergebnis ermittelt werden kann.

2. f in Python umzusetzen ist problematisch, weil Python (noch) keine bedingten Ausdrücke kennt. Man muss also eine Ersatzlösung mit gleichem Verhalten finden. Weil unser Beispiel einen rekursiven Ausdruck enthält muss diese Ersatzlösung sogar verzögerte Auswertung unterstützen. Denn würde der rekursive Ausdruck sofort ausgewertet, also ohne je vorher die Rekursionsendebedingung zu überprüfen, ergäbe sich ja stets eine Endlosrekursion. Eine Ersatzlösung, die inkl. verzögerter Auswertung funktioniert, zeigt folgendes Beispiel:

   ```
   f = lambda x: ( (x==0) and [1] or [f(x-1)*x] ) [0]
   f(5)
   ```

   Damit ist eine Umsetzung von f in Python:

   ```
   f = lambda x: ( (x==0) and [1] or [f(x)] ) [0]
   ```

3. Man kann diesen Ausdruck in Python schreiben:

   ```
   f = lambda x: ( (x==0) and [1] or [f(x)] ) [0]
   (lambda x,y: ((x==0) and [1] or [y])[0] ) (0,f(2))
   ```

   Die Auswertung jedoch ist unmöglich, weil Python eine strikte Auswertungsreihenfolge verwendet und deshalb $f\,(2)$ zu berechnen versucht. Dies endet in einer Endlos-Rekursion.

4. Es gibt strikte und nicht-strikte Auswertung. Python, Java und C++ verwenden strikte Auswertung, außer für bedingte Ausdrücke. Dort ist nicht-strikte Auswertung nötig, um Rekursion zu ermöglichen, und wird schon aus Effizienzgründen verwendet. Denn bedingte Ausdrücke und die logischen Funktionen »and« und »or« gehören zu den wenigen Fällen, die nicht-strikte effizienter als strikte Auswertung machen. Weil Operationen, die für das Ergebnis letztlich nicht relevant sind, gar nicht erst ausgeführt werden.
   Nicht-strikte Auswertung bei Funktionen kann man dem symbolischen Einsetzen bei mathematischen Rechnungen vergleichen: die Argumente der Funktion werden symbolisch (»als Zeichenkette«) eingesetzt, ohne zu diesem Zeitpunkt selbst ausgewertet zu werden.

5.9 Funktionen bis vierter Ordnung in Python

Aufgabe »Schreiben Sie in Python eine Funktion die folgender Definitionen in Lamba-Notation entspricht:«

$$\text{let } f1 = \lambda x.\lambda f.f\,(x)$$

Quelle: [2, Blatt 3, Aufg. 2, Teil 1]

Lösung Eigentlich müsste in der Aufgabenstellung noch der Typ von x angegeben werden. Möglichkeiten:

- x ist vom Typ »Wert« und habe den konkreten Wert c. Dann ist $f1$ eine Funktion dritter Ordnung, die eine Funktion zweiter Ordnung $\lambda f.f\,(c)$ liefert. Letztere übernimmt eine Funktion und gibt ihre Anwendung auf einen konstanten, internen Wert c zurück. Diese Version wurde in den Lösungshinweisen [3, Blatt 3, Aufg.2, Teil 1] realisiert:

  ```
  f1 = lambda x : lambda f : f(x)
  f2 = f1(2)
  print f2(lambda x: x*x)
  ```

- x ist vom Typ »Funktion« und habe den konkreten Wert $g\,(y)$. Dann ist $f1$ eine Funktion vierter Ordnung, die eine Funktion dritter Ordnung $\lambda f.f\,(g\,(y))$ liefert. Letztere übernimmt eine Funktion zweiter Ordnung f und gibt eine Funktion erster Ordnung zurück[2], die mit Hilfe von f aus einer internen Funktion $g\,(x)$ erzeugt wurde. In Python:

```
f1 = lambda x: lambda f: f(x)
f2 = f1(lambda x: x+3)
f3 = f2(lambda x: lambda y: x(y)*x(y) )
print f3(2)
```

  Merke: soll in Python ein λ-Ausdruck einen Wert liefern, so muss seine rechte Seite ein Ausdruck sein. Soll er dagegen eine Funktion beliebiger Ordnung liefern, so muss seine rechte Seite ein λ-Ausdruck sein. Ist ein Argument eines λ-Ausdrucks eine Funktion beliebiger Ordnung, so muss jede Verwendung dieses Arguments die Form eines Funktionsaufrufs haben. Argument kann dabei die unabhängige Variable einer neuen Funktion (im Beispiel oben: y) oder ein Wert sein. Diese Merkregeln für Python betreffen syntaktische Unterscheidungen zwischen Werten und Funktionen beliebiger Ordnung, die auch in der informalen λ-Notation zu finden sind. Der Wert des letzten Ausdrucks aus dem Beispiel oben ist:

$$\begin{aligned}
f3\,(2) &= \left(f2\left(\lambda x.\lambda y..x\,(y)^2\right)\right)(2)\\
&= \left(\left(f1\,(\lambda x.x+3)\right)\left(\lambda x.\lambda y.x\,(y)^2\right)\right)(2)\\
&= \left(\left((\lambda x.\lambda f.f\,(x))\,(\lambda x.x+3)\right)\left(\lambda x.\lambda y.x\,(y)^2\right)\right)(2)\\
&= \left(\left(\lambda f.f\,(\lambda x.x+3)\right)\left(\lambda x.\lambda y.x\,(y)^2\right)\right)(2)\\
&= \left(\left(\lambda x.\lambda y.x\,(y)^2\right)(\lambda x.x+3)\right)(2)\\
&= \left(\lambda y.\,(\lambda x.x+3)\,(y)^2\right)(2)\\
&= \left((\lambda x.x+3)\,(2)^2\right)\\
&= (2+3)^2 = 5^2 = 25
\end{aligned}$$

5.10 Dynamisch erzeugte Funktionen in Python

Aufgabe »Betrachten Sie folgendes Python-Programm:«

```
def compose(f, g):
  return lambda x: g(f(x))

fl = [lambda x: 2*x, lambda x: 3*x, lambda x: 4*x]
F = lambda x:x

for f in fl:
  F = compose(F, f)

print F(2)
```

»Welche Wirkung hat es, was passiert? Vergleichen Sie das obige mit dem folgenden Programm. Hat es die gleiche Wirkung? Sollte es die gleiche Wirkung haben?«

```
def compose(f, g):
  return lambda x: g(f(x))

F = lambda x:x
```

[2]Eine Funktion dritter Ordnung muss keine Funktion zweiter Ordnun *zurückgeben*. Auch wenn sie eine Funktion zweiter Ordnung als Parameter braucht, nennt man sie eine Funktion dritter Ordnung. Damit haben wir hier den Fall, dass es drei Funktionen $f1$, $f2$, $f3$ gibt, die von vierter, dritter und erster Ordnung sind. Es gibt keine Funktion zweiter Ordnung hier!

```
for i in [2, 3, 4]:
  F = compose(F, lambda x : i*x)

print F(2)
```

Quelle: [2, Blatt 3, Aufg. 4, Teil 1-2]

Lösung Im ersten Python-Programm werden die Elemente einer Liste von Funktionen zusammen mit der Identitätsfunktion zu folgender Hintereinanderausführung kombiniert:

$$\begin{aligned}(\lambda x.4 \cdot x)\left((\lambda x.3 \cdot x)\left((\lambda x.2 \cdot x)\left(\lambda x.x\right)\right)\right) &= (\lambda w.4 \cdot w)\left((\lambda x.3 \cdot x)\left((\lambda y.2 \cdot y)\left(\lambda z.z\right)\right)\right) \\ &= (\lambda w.4 \cdot w)\left((\lambda x.3 \cdot x)\left(2 \cdot (\lambda z.z)\right)\right) \\ &= (\lambda w.4 \cdot w)\left(3 \cdot (2 \cdot (\lambda z.z))\right) \\ &= (4 \cdot (3 \cdot (2 \cdot (\lambda z.z)))) \\ &= \lambda z.24 \cdot z\end{aligned}$$

Im zweiten Python-Programm passiert nicht dasselbe. Zwar wird auch eine Hintereinanderausführung von Funktionen erzeugt, jedoch noch ohne jeden konkreten Wert für i. Denn in Python besteht dynamische Bindung, also Bindung freer Vaiablen wie i an der Aufrufstelle, nicht an der Definitionsstelle. Freie Variablen werden bis zum Aufruf symbolisch (sozusagen »textuell«) gespeichert. Damit entsteht folgende Hintereinanderausführung:

$$\begin{aligned}(\lambda x.i \cdot x)\left((\lambda x.i \cdot x)\left((\lambda x.i \cdot x)\left(\lambda x.x\right)\right)\right) &= (\lambda w.i \cdot w)\left((\lambda x.i \cdot x)\left((\lambda y.i \cdot y)\left(\lambda z.z\right)\right)\right) \\ &= (i \cdot (i \cdot (i \cdot (\lambda z.z)))) \\ &= \lambda z.i^3 \cdot z\end{aligned}$$

An der Aufrufstelle `print F(2)` sollte i nun noch den Wert haben, mit dem es die `for`-Schleife verlassen hat, also 4. Damit erwarten wir als Ausgabe:

$$\text{let}\, i = 4 \,\text{in}\, \left(\lambda z.i^3 \cdot z\right)(2) = 4^3 \cdot 2 = 128$$

5.11 ggt-Funktion in der allgemeinen Form repetitiv rekursiver Funktionen

Aufgabe »Formen Sie folgende repetitiv rekursive Funktion zur Berechnung des *ggt* so um, dass sie dem Muster

$$f\left(x\right) = \text{if}\, p\left(x\right)\, \text{then}\, g\left(x\right)\, \text{else}\, f\left(h\left(x\right)\right)$$

solcher Funktionsdefinitionen entspricht. Definieren Sie also geeignete Funktionen (in Python) p, h und natürlich f entsprechend in der allgemeinen Form.« Quelle: [2, Blatt 4, Aufg. 3, Teil 9]

Lösung Die umzuformende repetitiv rekursive *ggt*-Funktion soll wohl sein:

$$\text{letrec}\, ggt = \lambda x, y.\text{if}\, x = y\, \text{then}\, x\, \text{else}\, ggt\left(max\left(x, y\right) - min\left(x, y\right), min\left(x, y\right)\right)$$

Umgesetzt in die allgemeine Form repetitiv rekursiver Funktionen, vorerst noch in informaler λ-Notation:

$$\begin{aligned}f\left(x, y\right) &= \text{if}\, p\left(x, y\right)\, \text{then}\, g\left(x, y\right)\, \text{else}\, f\left(h\left(x, y\right)\right) \\ g\left(x, y\right) &= \lambda x, y.x \\ h\left(x, y\right) &= \lambda x, y.\left(max\left(x, y\right) - min\left(x, y\right), min\left(x, y\right)\right) \\ p\left(x, y\right) &= \lambda x, y.x = y\end{aligned}$$

Und nun in Python:

```
def f(x):
  if p(x):
    return g(x)
  else:
```

```
    return f(h(x))

def g(x):
  return x[0]

def h(x):
  return (max(x[0],x[1])-min(x[0],x[1]), min(x[0],x[1]))

def p(x):
  return x[0]==x[1]

def max(x,y):
  if x>y:
    return x
  else:
    return y

def min(x,y):
  if x<y:
    return x
  else:
    return y

print f((24,36))
```

5.12 Repetitive Version von `reverse`

Aufgabe »Wandeln Sie auch die Funktion `reverse`

```
def reverse(l):
  if l == []:
    return []
  else:
    return append(reverse(cdr(l)), cons(car(l),[]))
```

entsprechend in eine Funktion `reverseCont1` um. Benutzen Sie dabei die Orginalversion von `append`, nicht Ihre gerade definierte Fortsetzungsvariante `appendCont`.« Quelle: [2, Blatt 4, Aufg. 7, Teil 3]

Lösung Erstmal die Version mit akkumulierendem Parameter, ganz ohne `append`:

```
def reverseCont1(l,rev):
  if l==[]:
    return rev
  else:
    return reverse( cdr(l),cons(car(l),rev) )

def reverseCont(l):
  return reverseCont1(l,[])
```

Und jetzt die Version mit Fortsetzungsfunktion und `append`:

```
def reverseCont1(l,cont):
  if l==[]:
    return cont([])
  else:
    return reverseCont1( cdr(l), lambda x: cons(car(l),cont(x)) )

def reverseCont(l):
  return reverseCont1(1, lambda x: x)
```

Literatur

[1] Prof. Dr. Thomas Letschert: »Funktionale Programmierung«. Quelle: Homepage von Prof. Dr. Letschert http://homepages.fh-giessen.de/~hg51/. Das offizielle Skript zu dieser Veranstaltung.

[2] Prof. Dr. Thomas Letschert: Aufgabenblätter zu Nichtprozedurale Programmierung. Die Nummerierung der Aufgabenblätter bezieht sich auf die entsprechenden Kapitel in [1].

[3] Prof. Dr. Thomas Letschert: Lösungshinweise zu den Aufgabenblättern zu Nichtprozedurale Programmierung.

[4] Prof. Dr. Thomas Letschert: Skript zu Programmierung II.