Prof. Dr. Th. Letschert
FB MNI

9. November 2003

Aufgabenblatt 4 - NPP

Aufgabe 1

1. Formen Sie folgende Funktionsdefinition in eine äquivalente Version um, die keine simultane Zuweisung nutzt:

```
def ggt(x, y):
  while x != y:
      (x,y) = (max(x,y)-min(x,y), min(x,y))
  return x
```

2. Geben Sie eine äquivalente rekursive Funktionsdefinition an.

Aufgabe 2

Die Quadratfunktion kann rekursiv wie folgt definiert werden:

$1^2 = 1$
$(n+1)^2 = n^2 + 2 * n + 1$

1. Transformieren Sie diese Definition mit Parametermustern in eine äquivalente Definition in informaler Lambda–Notation.
2. Was ist der Definitionbereich und der Wertebereich dieser Funktion?
3. Implementieren Sie diese Funktion in Python.
4. Beweisen Sie mit einem Induktionsbeweis, dass die rekusive Funktionsdefinition von oben korrekt ist, d.h. dass sie tatsächlich das Quadrat ihres Arguments berechnet.
5. Schreiben Sie (in Python) eine Funktion die das Quadrat *iterativ* in einer *while*–Schleife berechnet (Bitte Invariante angeben). Die Schleife soll dabei die in der rekusiven Variante zum Ausdruck gebrachte (und in Ihrem Beweis bewiesene) Eigenschaft der Quadratfunktion ausnutzen.
6. Handelt es sich bei der Rekursion in der Definition von oben um eine repetitive Rekursion? Wenn nein, ist es eventuell eine lineare Rekursion?
7. Geben Sie eine *repetitiv rekursive* Funktion zur Berechnung des Quadrats an, in der die repetitive Rekursion der Schleife (in der iterativen Variante) entspricht.

Aufgabe 3

1. Implementieren Sie die Funktion

   $\mathbf{letrec}\ f91 = \lambda x.\ \mathbf{if}\ x > 100\ \mathbf{then}\ x - 10\ \mathbf{else}\ f91(f91(x + 11))$

   in Python. Welche einfach zu beschreibende Wirkung hat diese Funktion?

2. Warum kann diese Funktion nicht mit Parametermustern definiert werden?

3. Folgt die Definition von $f91$ der Struktur seiner Definitionsmenge als induktive Menge? Was ist die Definitionsmenge? Ist es eine induktive Menge? Was ist das überhaupt?

4. Definieren Sie die induktive Menge der Sequenzen von Integer–Werten. Was ist die Basismenge, was die Konstruktionsfunktion? Definieren Sie eine, der Konstruktionsfunktion entsprechende, Selektionsfunktionen.

5. Warum sind die reellen Zahlen keine induktive Menge?

6. Beschreiben Sie die wesentlichen Unterschiede zwischen Sequenzen und den Listen im Sinne von Lisp. Welche Wirkung haben die Lisp–Operationen $cons$, car und cdr?

7. Bestimme Sie die Werte $car(l)$ und $cdr(l)$ für jede der folgenden Listen l:

   (a) (A, B)

   (b) (A, B, C)

   (c) $((A, B), (C, D))$

   (d) (A)

8. Definieren Sie in Python die Funktionen `cons`, `car` und `cdr`, die auf Python–Listen angewendet werden können und sich entsprechend ihrer Definition in Lisp verhalten.

9. Formen Sie folgende repetitiv rekursive Funktion zur Berechnung des ggt so um, dass sie dem Muster

   $$f(x) \;=\; \textbf{if}\; p(x)\; \textbf{then}\; g(x)\; \textbf{else}\; f(h(x))$$

   solcher Funktionsdefinitionen entspricht. Definieren Sie also geeignete Funktionen (in Python) p, h und natürlich f entsprechend in der allgemeinen Form.

10. Definieren Sie (in Python) die linear rekursive Fakultätsfunktion nach dem allgemeinen Muster

    $$f(x) \;=\; \textbf{if}\; p(x)\; \textbf{then}\; g(x)\; \textbf{else}\; \psi(f(h(x)), x)$$

    Bestimmen Sie also die Funktionen p, h, ψ und f.

11. Definieren Sie eine rekursive Funktion zur Berechnung der Fakultät. Zu welcher Kategorie rekursiver Funktionsdefinitionen gehören Ihre Definition?

Aufgabe 4

1. Die übliche Definition der Fakultätsfunktion ist linear rekursiv:

   $$\textbf{letrec}\; fak \;=\; \lambda x.\; \textbf{if}\; x = 0\; \textbf{then}\; 1\; \textbf{else}\; fak(x-1) * x$$

   Transformieren Sie sie mit Hilfe der Technik der Akkumulation in eine repetitiv rekursive Form und implementieren Sie diese dann in Python. Überführen Sie die repetitive Rekursion dann in eine Schleife.

2. Implementieren Sie in Python die Fibonacci–Funktion mit ihrer allgemein rekursiven Definition. Geben Sie eine entsprechende repetitiv rekursive Form an. (Hinweis: Man darf auch rückwärts von einer Schleife her denken.)

3. Transformieren Sie die folgenden *linear rekursiven* Funktionen in *repetitiv rekursive* Form

   (a) Berechnung der Summe aller Elemente in einer Sequenz:
   $sumL(NIL) = 0$
   $sumL(cons(a, l)) = a + sumL(l)$

   (b) Bestimmung des größten Elements in einer Sequenz:
   $maxL(NIL) = 0$
   $maxL(cons(a, l)) = max(a, maxL(l))$

   (c) Verketten von Listen:
   $append(NIL, l) = l$
   $append(cons(a, l), l') = cons(a, append(l, l')$

   (d) Invertieren einer Liste:
   $reverse(NIL) = NIL$
   $reverse(cons(a, l)) = append(reverse(l), cons(a, NIL))$

4. Definieren Sie (falls nocht nicht geschehen) die Listenfunktionen $cons$, car und cdr entsprechend ihrer Lisp–Definition in Python, so dass sie sich auf Python–Listen anwenden lassen.

5. Implementieren und testen Sie die obigen Funktionen auf Sequenzen mit Hilfe Ihrer Hilfsfunktionen $cons$, car und cdr.

6. Implementieren und testen Sie die von Ihnen definierten *repetitiv–rekursiven Varianten* der Funktionen in Python.

7. Transformieren Sie Ihre repetitiv–rekursiven Sequenzfunktionen in *iterative* Form.

Aufgabe 5

1. Definieren mit linearer Rekursion eine Funktion $quadL$, die eine Liste von Int–Werten als Argument akzeptiert und die Liste der entsprechenden Quadratwerte zurück liefert.

   Wandeln Sie Ihre Funktion in *repetitiv rekursive* Form um und implementieren Sie diese dann *iterativ* in Python.

2. Definieren mit linearer Rekursion eine Funktion $myMap$, die eine Funktion f vom Typ $Int \rightarrow Int$ sowie eine Liste $< e_0, e_1, \cdots >$ von Int–Werten als Argument akzeptiert und die Liste $< f(e_0), f(e_1), \cdots >$ zurück liefert.

   Wandeln Sie Ihre Funktion in *repetitiv rekursive* Form um und implementieren Sie diese dann *iterativ* in Python.

3. Python enthält eine Funktion namens `zip` mit der ein Paar von Listen in eine Liste von Paaren umgeformt werden kann. Das Element an Position i der Ergebnisliste ist dabei das Paar aus den Elementen von Position i der beiden Argumentlisten. Beispiel:

   ```
   >>> zip((1,2,3),('a','b','c'))
   [(1, 'a'), (2, 'b'), (3, 'c')]
   ```

   Schreiben Sie (in Python) eine möglichst einfache eigene linear rekursive Funktion `myZip` mit dieser Funktionalität.

   Wandeln Sie `myZip` in eine repetitiv rekursive Funktion um und implemetieren Sie sie als iterative Funktion in Python.

Aufgabe 6

1. Die Generierung rekursiver Funktionen ist etwas problematisch. Funktionen erzeugende Funktionen liefern ihr Ergebnis ja in Form von Lambda–Ausdrücken, diese sind namenlos und die Definition namenloser rekursiver Funktionen ist schwierig. Als Beispiel betrachte man folgende Variante der map Funktion die eine rekursive Funktion generiert:

   $$\begin{array}{l}\textbf{let } map \ = \ \lambda f.g \\ \quad \textbf{whererec } g \ = \ \lambda l.\textbf{if } l = NIL \textbf{ then } NIL \\ \qquad \textbf{else } cons(f(car(l)), g(cdr(l)))\end{array}$$

   Das kann man auch etwas anders fomulieren, ($map(f)$ ist die rekursive Funktion):

   $$\begin{array}{l}\textbf{letrec } map(f) \ = \ \lambda l.\textbf{if } l = NIL \textbf{ then } NIL \\ \qquad \textbf{else } cons(f(car(l)), map(f)(cdr(l)))\end{array}$$

   Schreiben Sie eine eine entsprechende map–Funktion in Python. Ihre map–Funktion hat also eine Funktion als Parameter und liefert eine Funktion auf Listen.

   Ihre Version von map soll beispielsweise für

   ```
   mapQuad = map(lambda x: x*x)
   mapQuad([1,2,3,4,5,6])
   ```

   den Wert `[1,4,9,16,25,36]` liefern.

2. Welchen Typ hat $map(map(lambdax : x * x))$? Testen Sie es mit einer Liste von Listen!

   Kann der Prozess iteriert werden? Was ist $map(map(map(lambdax : x * x)))$?

3. Was ist nun mit $map(map)$? Kann es auch auf etwas angewendet werden? Hinweis, testen Sie:

   ```
   for f in ((map(map))([lambda x: x, lambda x:2*x, lambda x: 3*x])):
      print f([1,2,3])
   ```

Aufgabe 7

Eine lineare Rekursion hat eine “hängende Operation” nach dem rekusiven Aufruf. Durch die Technik der Akkumulation kann die lineare in eine repetitive Rekursion umgewandelt werden. In einem zusätzlichen Parameter werden dabei die Zwischenergebnisse bis zum Endergebnis “aufgesammelt”.

Statt des Aufsammelns von Werten kann man auch direkt eine “nachhängende Operation” aufbauen. Dabei werden keine Zwischenergebnisse aufgesammelt, sondern, statt dessen, wird direkt die Gesamtheit der nachhängenden Operationen als Funktion aufgebaut. Die lineare Rekursion wird dabei zu einer repetiven Rekursion mit einer *Fortstezungfunktion* (*continuation function*). Diese sagt, was mit dem Ergebnis zu geschehen hat – wie es also weitergeht.

Als Beispiel dient uns wieder die Fakultätsfunktion:

```
def fakCont(x, cont):
    if x == 0:
        return cont(1)
    else:
        fakCont(x-1, lambda y : cont(y*x))
```

```
def printIt (x):
    print x

fakCont(5, printIt)
```

Hier wird die Fakultät von 5 berechnet. Man beginnt mit `fakCont(5,printIt)` als Auftrag an `fakCont` die Fakutät von 5 zu berechnen und diese dann auszugeben. Der Auftrag wird weitergereicht an `fakCont(4,..)` als Auftrag die Fakultät von 4 zu berechnen, diese mit 5 zu multiplizieren und das Ergebnis dann auszudrucken, und so weiter.

1. Machen Sie den "Schreibtischtest" für `fakCont(3,printIt)`. Werten Sie also per Kopf, Hand, Stift und Papier den Funktionsaufruf aus.

2. Wandeln Sie nach diesem Muster die unten definierte linear rekursive Funktion `append` in eine repetitiv rekursive Funktionen `appendCont` mit Fortsetzungsparameter um.

   ```
   def car(l):
       return l[0]

   def cdr(l):
       return l[1: ]

   def cons(x, l):
       return [x] + l

   def append(l1, l2):
       if l1 == []:
           return l2
       else:
           return cons(car(l1), append(cdr(l1),l2))
   ```

3. Wandeln Sie auch die Funktion `reverse`

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

   entsprechend in eine Funktion `reverseCont1` um. Benutzen Sie dabei die Orginalversion von `append`, nicht Ihre gerade definierte Fortsetzungsvariante `appendCont`.

4. Definieren Sie `reverseCont2` indem Sie in `reverseCont1`, Ihrer Fortsetzungsversion von `reverse`, die Hilfsfunktion `append` durch die von Ihnen definierte Fortsetzungsvariante `appendCont` ersetzen.