Übung zur VL „Grundlagen der Programmierung“

7. Übung

Dr. Zubow

Vollständige Induktion

- Zeigen Sie mit der Methode der vollständigen Induktion, dass die Anzahl D(n) der Diagonalen in einem konvexen ebenen n-Eck nach der folgenden Formel berechnet werden kann:

$$D(n)= n(n-3)/2 \text{ für } n \geq 3.$$

n	3	4	5	6	7	8	9	10	11	12	13	14	15
d	0	2	5	9	14	20	27	35	44	54	65	77	90

Rekursion

- Prinzip:
  - Führe ein großes (schweres) Problem auf ein kleines (leichteres) Problem zurück
  - Notwendig für leistungsstarke Such- und Sortieralgorithmen
- Grundgedanke ist, dass sich Funktionen selbst aufrufen
- Aufgabe:
  - Schreiben Sie eine Funktion, der Sie eine Zahl übergeben und die dann von dieser Zahl ausgehend bis 1 herunterzählt. Die Funktion darf keine Schleife verwenden.
  - Schreiben Sie eine Funktion, der Sie eine Zahl übergeben und die dann von 1 aus bis zu dieser Zahl hochzählt. Die Funktion darf keine Schleife verwenden.

Rekursion

- Lösung:

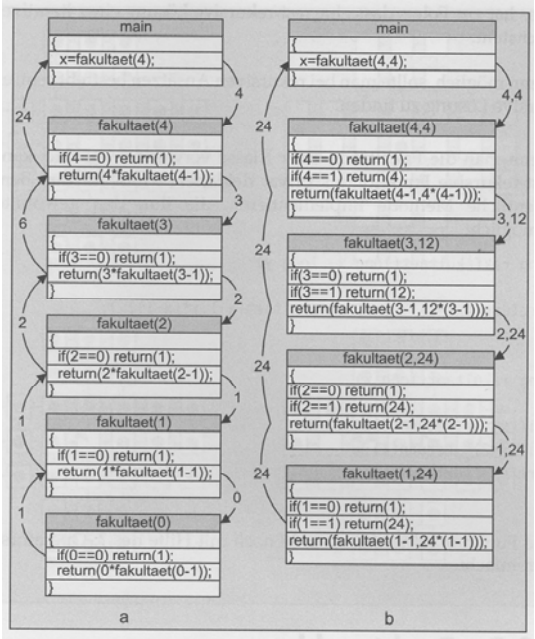
```
def zaehler1(int x) {
    println(x);
    if (x>1)
        zaehler1(x-1);
}

def zaehler2(int x) {
    if (x>1)
        zaehler2(x-1);
    println(x);
}
```

Rekursion

- Fakultät

```
def rekfakultaet(int x) {
 if (x < 0)
     return 0;
 if (x == 0)
     return 1;
 return x * rekfakultaet(x-1);
}
def rekfakultaet(int x, int y) {
 if (x < 0)
     return 0;
 if (x == 0)
     return 1;
 if (x == 1)
     return y;
 return rekfakultaet(x-1, y*(x-1));
}
```



Rekursion

- Tail-end Recursion
  - Rekursionen, deren rekursiver Schritt als allerletztes geschieht, dem also keinerlei weitere Berechnungen folgen.
  - Vorteil: sehr einfach in Schleifen umzuwandeln
  - Schema:

```
func(n) {
  if (Abbruchbedingung) {
    return Wert(n)
  } else {
    m = berechne_einfacheres(n)
    return func(m) // Rekursion
  }
}
```

wird zu

```
while (!Abbruchbedingung) {
  m = berechne_einfacheres(n)
}
return Wert(n)
```

Formale Modelle

- Das λ-Kalkül ist ein Modell zur formalen Beschreibung von Programmiersprachen.
- Anforderungen an formale Modelle
  - **Mächtigkeit,**
  - **Einfachheit**
- Zwei der bekanntesten formalen Modelle sind
  - **Turingmaschine** - verwandt mit imperativen Sprachen (Pascal oder C).
  - **Lambda-Kalkül** - verwandt mit funktionalen Sprachen (ML oder Lisp).

Imperative vs. funktionale Sprachen

- In **imperativen Sprachen**
  - werden Programme als eine **Folge von Anweisungen** interpretiert, die nacheinander abgearbeitet werden
  - können Variablen **wechselnde Werte zugewiesen werden**
- In **funktionalen Sprachen**
  - besteht ein Programm *allein aus Funktions**definition,*** Funktions**anwendung und Funktionskomposition**
  - steht jeder Bezeichner in einen Kontext für **genau einen** Wert
  - Bsp.: Funktionsdefinition: square (n) = (n * n) oder twice (f x) = f (f (x))
    Funktionsanwendung: square (3) → 9
    Anwendung + Komposition: twice (square 3)

Lambda-Kalkül

- Der Lambda-Kalkül besteht aus zwei Teilen:
  - **Funktionsabstraktion**: λ x . A
    definiert eine (anonyme) Funktion, die ein x bekommt, und einen Ausdruck A als Funktionskörper hat
  - **Funktionsapplikation**: F A
    bedeutet, dass die Funktion F auf den Ausdruck A angewandt wird

Beispiele für Funktionen

- Die Identität: λ x . x
- Eine Funktion, die jedes Argument auf die Identitätsfunktion abbildet: λ y . (λ x . x)
- Ein komplexerer Ausdruck:
  (λ f . (λ x . f (f x))) u v
  (λ f . (λ x . f (f x))) u v
  →(λ x . u (u x)) v
  →u (u v)
- Diese Funktion wendet also eine Funktion zweimal auf ein Argument an.

Eigenschaften des λ-Kalküls

- Als Bausteine gibt es nur Funktionabstraktion und –applikation:
  - Es gibt keine Zahlen, Funktionsnamen, Arithmetische Funktionen, Wahrheitwerte.
- Die Funktionen werden nicht benannt, sie sind anonym.
- Der Lambda-Kalkül ist ungetypt.

Logische Funktionen - True, False, If

- Die Werte True und False werden mit zwei Funktionen repräsentiert:

  True sei die folgende Funktion:

  λ x y . x (Projektion auf erstes Argument)

  False sei die folgende Funktion:

  λ x y . y (Projektion auf zweites Argument)

  Damit können wir ein If definieren:

  If .. then .. else .. entspricht hier λ b x y . b x y,

  wobei b entweder True oder False ist.

Natürliche Zahlen

- Im λ-Kalkül gibt es nur Funktionen. Um Zahlen zu erhalten, werden bestimmte Funktionen mit den natürlichen Zahlen assoziiert:
- $z_n \equiv \lambda$ f x . f^n (x) (die n-malige Anwendung von f auf x)
- Bsp.:
  - z_3: λ f x . f (f (f (x)))
  - Z_{42}: λ f x . f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (
    f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (
    f (f (f (f (x))
)))))))

Arithmetische Funktionen

- Definition von Plus: λ z z' f x . z f (z' f x)
- Bsp.: Plus z_1 z_2 (→ z_3)

  z_1 = (λ f x . f (x))

  z_2 = (λ f x . f (f (x)))
- Definition von Mal: λ z z' f x . z (z' f) x

  oder: λ z z' f . z (z' f)