Technische Universität Darmstadt
Fachbereich Informatik
Prof. Dr. Johannes Fürnkranz

Allgemeine Informatik 1 im WS 2006/07

Übungsblatt 11

Bearbeitungszeit: 22.01. bis 28.01.2007

Aufgabe 1: Rekursion oder „iterativ“ vs. „rekursiv“

Als Rekursion bezeichnet man den Aufruf oder die Definition einer Funktion durch sich selbst. Dieses Verfahren eignet sich, in vielen Fällen Probleme auf eine elegantere Weise zu lösen, als sich des normalen, iterativen Konzeptes zu bedienen.

Das Prinzip der Rekursion: **Divide et impera** (teile und herrsche)

Ziel dieser Aufgabe wird sein die Rekursion zu verstehen und eine Intuition zu bekommen, was hinter diesem Prinzip zu verstehen ist.

Beispiel 1: Karel soll immer geradeaus gehen bis er auf eine Wand trifft.

ITERATIV (den Weg den sie bisher beim Programmieren verfolgt haben):

```
void moveToWall() {
     while (frontIsClear()){
          move();
     }
}
```

REKURSIV

```
void moveToWall() {
     if (frontIsClear()){
          move();
          moveToWall();
     }
}
```

Machen Sie sich klar was bei der Ausführung der beiden **`moveToWall()`** Methoden passiert! Denn beide Methoden erzielen bei der Ausführung genau das gleiche Ergebnis. Sie werden sich fragen: Warum also rekursiv programmieren? Die Antwort in genau diesem Fall: Es bringt keinen Vorteil. Die Lösungen sind gleichwertig.

Aber nun verändern wir den Aufgabentext ein wenig:

Beispiel 2: Karel soll immer geradeaus gehen bis er auf eine Wand trifft. Danach soll Karel genau wieder zu seinem Startpunkt zurückgehen. Bei der Programmierung können sie annehmen, dass es eine Methode **`schrittZurueck()`** gibt.

```
void schrittZurueck() {
     turnLeft();turnLeft();
     move();
     turnLeft();turnLeft();
}
```

ITERATIVE Lösung

```
void moveToWallAndBack() {
     int schritte = 0;
     while (frontIsClear()){
          move();
          schritte++;
     }
     loop (schritte){
          schrittZurueck();
     }
}
```

REKURSIVE Lösung

```
void moveToWallAndBack() {
     if (frontIsClear()){
          move();
          moveToWallAndBack();
          schrittZurueck();
     }
}
```

Machen Sie sich wiederum klar was bei der Ausführung der beiden **`moveToWallAndBack()`** Methoden passiert! Warum passiert bei beiden Programmen genau das gleiche, wenn Sie diese ausführen?

In diesem Beispiel wird schon deutlicher warum Rekursion sinnvoll sein kann – Aus 10 Zeilen Code wurden 7 und zusätzlich wurde die Lesbarkeit verbessert. Rekursiv kann in diesem Fall als „eleganter“ bezeichnet werden.

Konstruktion von Rekursion: *(mit Bezug zu unserem Beispiel)*

1.) Suche **das kleinste Teilproblem** und eine Bedingung die es erfasst.
   - *Der Roboter steht vor der Wand.*

2.) Was muss der Roboter in diesem Fall tun?
   - *Der Roboter muss in diesem Fall nichts tun.*

3.) Finde eine Möglichkeit, das **Gesamtproblem zu teilen und zu reduzieren.**
   - ***Reduktion:*** *Der Roboter kann ein Feld weiter dasselbe Problem bearbeiten und anschließen zurückgehen.*

4.) Stelle sicher, dass die **Reduktion des Problems zu dem kleinsten Teilproblem führt.**
   - ***Rückführbarkeit:*** *Solange irgendwo vor dem Roboter eine Wand ist, kommt der Roboter mit jedem* **`move()`** *Schritt dem kleinsten Teilproblem näher.*

Testen Sie die Rekursionsbeispiele in Karel. Kreieren sie sich dafür eigenständig eine passende Welt.

Aufgabe 2: Rekursion

Wir betrachten die Zahlenreihe „**siggi**“: 0, 1, 3, 8, 20, 49, 119, …..

Die Reihe ist wie folgt aufgebaut :

```
siggi[n] := (siggi[n-1]*2 ) + (siggi[n-2] +1)
siggi[1] := 1;
siggi[0] := 0;
```

In Worten: Das erste Element der Reihe ist „0“. Das zweite „1“. Ab dem 3. Element setzt sich die Zahlenreihe **siggi** aus der „Summe von zwei mal Vorgänger und Vorvorgänger plus eins“ zusammen.

Wie Sie in Aufgabe 1 gelernt haben, ist die oben definierte Reihe rekursiv aufgebaut, da sie immer erst zwei Vorgänger berechnen müssen um ein Element größer gleich 2 der **siggi**-Zahlenreihe zu erhalten. Beispielsweise muss man um **siggi[3]** zu berechnen **siggi[2]** und **siggi[1]** kennen. **siggi[1]** ist bekannt. Fehlt also noch die Berechnung von **siggi[2]. Diese setzt die Kenntnis von siggi[1]** und **siggi[0]** voraus. Da beide Werte vorhanden sind, können wir das Beispiel von hinten lösen!

Aufgabenbeschreibung

a) Implementieren Sie diese Reihe als **rekursive** Methode in KarelJ.

b) Es gibt **n** Beeper auf der Kreuzung [2,2]. Diese **n** Beeper geben die Stelle der **siggi**-Reihe an, die berechnet werden soll. Liegen also 5 Beeper auf der Kreuzung, dann muss die **siggi**-Zahl 49 herauskommen.

c) Das Ergebnis geben sie bitte in der Konsole in der Form „**siggi[5] := 49**“.

Hinweis: Für diese Aufgabe wird kein Grundgerüst bereitgestellt. Das bedeutet für sie, dass Sie sich selbst Gedanken machen müssen, ob und welche Klassen und oder Methoden zu programmieren sind.

Aufgabe 3: Arbeiten ohne Roboter

In der folgenden Aufgabe werden wir in der KarelJIDE etwas programmieren, ohne dass Roboter benutzt werden. Die Aufgabe besteht aus der Umwandlung einer Farbcodierung aus dem RGB-Farbraum in den HSV-Farbraum.

Der **RGB-Farbraum** wird durch die drei Grundfarben Rot, Grün und Blau (englisch red, green und blue) definiert. Für die folgende Aufgabe definieren wir den R, B, G-Wertebereich im Intervall von [0,1].

Der **HSV-Farbraum** beschreibt man die Farbe mit Hilfe des Farbtons (englisch **h**ue), der Sättigung (**s**aturation) und dem Grauwert (**v**alue). Der HSV-Wertebereich definiert sich wie folgt: *H* Werte liegen im Intervall [0, 360] wobei *S* und *V* Werte sich im Intervall [0, 100]% befinden.

Farbe	R	G	B	H	S	V
Gelb	1	1	0	60	100	100
Rot	1	0	0	0 oder 360	100	100
Blau	0	0	1	240	100	100

Tabelle 1: Umrechnungstabelle RGB zu HSV

Aufgabenbeschreibung

a) Schreiben Sie ein Klasse **`RGBKonvertierer`**. Diese Klasse besitzt die drei double Attribute **`r`**, **`g`** und **`b`**. Diese Attribute werden über einen Konstruktor initialisiert.

Hinweis: Da wir uns in der **`KarelJ`** Umgebung befinden, müssen wir unsere Klasse vom **`UrRobot`** ableiten. Dies bedeutet auch, dass wir im **`RGBKonvertierer`**-Konstruktor den übergeordneten **`UrRobot`**-Konstruktor bedienen müssen. Falls Sie dies nicht tun, bekommen Sie eine entsprechende Fehlermeldung. Da wir aber keine Roboter benötigen können Sie sich beliebige Dummy-Werte ausdenken, die Sie an den **`UrRobot`**-Konstruktor weiterleiten.

b) Bitte implementieren Sie nun eine Methode ohne Rückgabewert **`wandeleInHSV()`**, die auf Basis der Attribute des RBG Instanz die entsprechenden HSV-Werte ermittelt. Richten Sie sich dabei an folgenden Algorithmus:

$$MAX = max(R,G,B),\ \ MIN = min(R,G,B)$$

$$if(MAX = MIN)\, H = 0 \text{ else } \ H = \begin{cases} \left(0 + \frac{G-B}{MAX-MIN}\right) \cdot 60, & \text{if } R = MAX \\ \left(2 + \frac{B-R}{MAX-MIN}\right) \cdot 60, & \text{if } G = MAX \\ \left(4 + \frac{R-G}{MAX-MIN}\right) \cdot 60, & \text{if } B = MAX \end{cases}$$

$$if(H < 0)\, H = H + 360$$

$$if(MAX = 0)\, S = 0 \text{ else } \ S = 100 \cdot \frac{MAX - MIN}{MAX}$$

$$V = 100 \cdot MAX$$

Bitte geben Sie das HSV-Ergebnis am Ende der Methode auf der Konsole aus.

c) Erzeugen Sie nun 3 Instanzen des **`RGBKonvertierers`**. Nutzen Sie die Werte aus der Tabelle auf der vorherigen Seite. Denn für diese besitzen Sie die entsprechenden HSV-Werte und können ihr Programm direkt auf Richtigkeit überprüfen.

d) **Hinweis**: Für weitere Informationen können Sie sich die Wikipedia-Internetseite http://de.wikipedia.org/wiki/HSV-Farbraum ansehen. Dort finden Sie neben einer ausführlicheren Beschreibung auch weitere Farbwerte.