4. Übungsaufgabe zu
Fortgeschrittene funktionale Programmierung
Thema: Backtracking, Dynamische Programmierung
ausgegeben: Mi, 25.04.2012, fällig: Mi, 02.05.2012

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften zur Lösung der im folgenden angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `AufgabeFFP4.hs` in Ihrem Gruppenverzeichnis ablegen, wie gewohnt auf oberstem Niveau. Kommentieren Sie Ihre Programme aussagekräftig und benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten.

- Wir betrachten eine Variante des Knapsack- bzw. Rucksackproblems von Aufgabenblatt 3.

  Gegeben ist eine endliche Menge von Gegenständen, die durch ihr Gewicht und ihren Wert gekennzeichnet sind. Aufgabe ist es, den Rucksack so zu bepacken, dass die Summe der Werte der eingepackten Gegenstände maximal ist, ohne ein vorgegebenes Höchstgewicht zu überschreiten.

  Dazu sollen drei Funktionen `succKnp`, `goalKnp` und `knapsack` geschrieben werden, so dass `knapsack` sich auf das Funktional `searchDfs` abstützt und das Optimierungsproblem mittels *backtracking* löst. Dabei wird ein Baum aufgebaut, dessen Knoten mit folgenden Informationen benannt sind: dem Wert und Gewicht des aktuellen Rucksackinhalts, dem nicht zu überschreitenden Höchstgewicht, der Liste der noch auswählbaren noch nicht eingepackten Gegenstände, sowie der Liste der bereits eingepackten Gegenstände.

  ```
  type NodeKnp = [Value,Weight,MaxWeight,[Object],SolKnp]
  ```

  Wir verwenden folgende Typen und Deklarationen zur Modellierung des Rucksackproblems:

  ```
  type Weight      = Int                -- Gewicht
  type Value       = Int                -- Wert
  type MaxWeight   = Weight             -- Hoechstzulaessiges Rucksackgewicht

  type Object      = (Weight,Value)     -- Gegenstand als Gewichts-/Wertpaar
  type Objects     = [Object]           -- Menge der anfaenglich gegebenen
                                           Gegenstaende
  type SolKnp      = [Object]           -- Auswahl aus der Menge der anfaenglich
                                           gegebenen Gegenstaende; moegliche
                                           Rucksackbeladung, falls zulaessig

  type NodeKnp     = (Value,Weight,MaxWeight,[Object],SolKnp]) -- s.o.


  succKnp :: NodeKnp -> [NodeKnp]
  succKnp (v,w,limit,objects,psol) = ...
  ```

```
goalKnp :: NodeKnp -> Bool
goalKnp (_,w,limit,((w',_):_),_) = ...

knapsack :: Objects -> MaxWeight -> (SolKnp,Value)
knapsack objects limit = ...
         where ... = ... searchDfs ...
```

Beispiel:

```
knapsack [(2,3),(2,3),(3,4),(3,4),(5,6)] 10
  ->> ([(2,3),(2,3),(3,4),(3,4)],14)
```

Hinweis: Die Reihenfolge der Elemente in der Ergebnisliste spielt keine Rolle. Gibt es mehr als eine beste Lösung, reicht es, eine davon auszuwählen.

- Für Binomialkoeffizienten gilt folgende Beziehung:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

für $0 < k < n$,

$$\binom{n}{k} = 1$$

für $k = 0$ oder $k = n$ und

$$\binom{n}{k} = 0$$

sonst.

Schreiben Sie nach dem Vorbild aus Kapitel 3.5 der Vorlesung eine Variante zur Berechnung der Binomialkoeffizienten mithilfe dynamischer Programmierung. Stützen Sie Ihre Implementierung dazu auf das Funktional `dynamic` und geeignete Funktionen `compB` und `bndsB` ab:

```
binomDyn :: (Integer,Integer) -> Integer
binomDyn (m,n) = ... where ... dynamic compB... bndsB...
```

Vergleichen Sie (ohne Abgabe!) das Laufzeitverhalten der Implementierung `binomDyn` mit denen der drei Implementierungen `binom`, `binomS` und `binomM` von Aufgabenblatt 3 miteinander.