**4. Übungsaufgabe zu**
**Fortgeschrittene funktionale Programmierung**
**Thema: Rücksetzsuche, Dynamische Programmierung**
**ausgegeben: Mi, 13.04.2016, fällig: Mi, 20.04.2016**

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.

1. Die Funktion höherer Ordnung `searchDfs` aus Kapitel 3.2 der Vorlesung realisiert eine Rücksetztiefensuche (engl. *depth-first-search backtracking*).

   Wandeln Sie die Funktion `searchDfs` zu einer Funktion `searchBfs` ab, die konzeptuell mit `searchDfs` übereinstimmt, aber eine Rücksetzbreitensuche realisiert:

   ```
   searchBfs :: (node -> [node]) -> (node -> Bool) -> node -> [node]
   searchBfs succ goal x = ...
   ```

   Die Implementierung von `searchBfs` möge dabei den Datentyp

   ```
   data Stack a = EmptyStk
                | Stk a (Stack a)
   ```

   aus Kapitel 3.2 zur Realisierung von Stacks zugrundelegen, um auf das "Absammeln" von Modulen für diese Aufgabe verzichten zu können.

2. Wir betrachten noch einmal das Springerproblem von Aufgabenblatt 3, interpretieren den Parameter `AnzahlZuege` der Funktion `springer` jetzt aber im Sinn von "die Zielposition kann ausgehend von der Startposition in $n$ oder weniger Zügen erreicht werden". Erreicht eine Zugfolge die Zielposition in weniger als $n$ Zügen, so endet die Zugfolge hier: Wir suchen für solche Zugfolgen keine Verlängerungen, die möglicherweise wieder zur Zielposition zurückführen.

   Wir wollen dieses Springerproblem mithilfe der Rücksetzbreiten- und -tiefensuche lösen. Dazu sollen Funktionen `spgSucc`, `spgGoal` sowie `spgDfs` und `spgBfs` geschrieben werden, wobei sich `spgDfs` und `spgBfs` auf die Funktionale `searchDfs` und `searchBfs` abstützen. Dabei wird jeweils ein Baum aufgebaut, desssen Knoten mit folgenden Informationen benannt sind: der Startposition, der Zielposition, der maximal zulässigen Zahl von Zügen, der aktuellen Position, der Zahl der Züge bis zu dieser aktuellen Position und der Zugfolge von der Startposition zur aktuellen Position.

```
type SpgNode =
       (StartPosition,ZielPosition,AnzahlZuege,
        AktPosition,VerbrauchteZuege,[Zugfolge])
```

Dabei werden folgende weitere von Aufgabenblatt 2 bekannte Typen und Deklarationen verwendet:

```
data Reihe = A | B | C | D | E | F | G | H
                                        deriving (Eq,Ord,Enum,Show)
data Linie = Eins | Zwei | Drei | Vier | Fuenf
             | Sechs | Sieben | Acht  deriving (Eq,Ord,Enum,Show)
type StartPosition    = (Reihe,Linie)
type ZielPosition     = (Reihe,Linie)
type AnzahlZuege      = Int
typz VerbrauchteZuege = Int
type VonPosition      = (Reihe,Linie)
type NachPosition     = (Reihe,Linie)
type Zug              = (VonPosition,NachPosition)
type Zugfolge         = [Zug]
```

Gesucht sind nun Funktionen

```
spgSucc :: SpgNode -> [SpgNode]
spgSucc (sp,zp,az,ap,zf) = ...

spgGoal :: SpgNode -> Bool
spgGoal (sp,zp,az,ap,zf) = ...

spgDfs :: StartPosition -> ZielPosition -> AnzahlZuege -> [Zugfolge]
spgDfs sp zp az = ...
       where ... = ... searchDfs ...

spgBfs :: StartPosition -> ZielPosition -> AnzahlZuege -> [Zugfolge]
spgBfs sp zp az = ...
       where ... = ... searchBfs ...
```

so dass die Aufrufe von `spgDfs` und `spgBfs` alle Zugfolgen bestimmen, die das Gewünschte leisten. Dabei operieren alle Funktionen auf dem nachstehend illustrierten $8 \times 8$-Schachbrett – unzulässige Züge, die aus dem Schachbrett herausführen, sind also nicht möglich.

```
         A   B  C  D  E  F  G  H
Acht                              Acht
Sieben                            Sieben
Sechs                             Sechs
Fuenf                             Fuenf
Vier                              Vier
Drei                              Drei
Zwei                              Zwei
Eins                              Eins
         A   B  C  D  E  F  G  H
```

Weiters wird für die nächsten Züge eines Springers die im folgenden illustrierte Reihenfolge festgelegt:

```
    1   2
8       3
   Spg
7       4
    6   5
```

Der Springer wählt die Züge also im Uhrzeigersinn, d.h. der erste Zug aus einer Position heraus führt ihn nach oben links, der zweite nach oben rechts, der dritte nach rechts links, der vierte nach rechts rechts usw.

*Hinweis:* Implementieren Sie die ADTs `Stack` und `Table` und die darauf festgelegten Operationen als gewöhnliche Typen und Operationen in Ihrer Abgabedatei AufgabeFFP4.hs, da die Definition mehrerer Module in einer Datei (wie z.B. AufgabeFFP4.hs) nicht unterstützt wird. Implementieren Sie den ADT `Stack` insbesondere so, wie in Aufgabenteil 1 angegeben.

3. 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.