

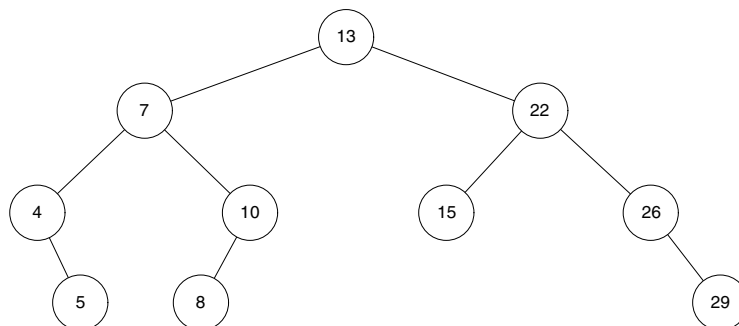
## Übungen zu Einführung in die Informatik I

### Aufgabe 18      **Bäume in OCaml**

In dieser Aufgabe wenden wir uns nochmal (vgl. Vorlesung) den binären Bäumen zu.

Zur Erinnerung:

- Ein Binärbaum ist entweder leer, oder besteht aus einem Knoten, sowie zwei mit der Wurzel verbundenen Binärbäumen, dem linken und dem rechten Teilbaum.
  - Ein Knoten kann beliebige Informationen speichern, also z.B. integer, floats, strings, oder auch eigene Datentypen.
  - Die Anzahl der Kanten zwischen einem Knoten und der Wurzel wird als *Niveau des Knotens* bezeichnet.
  - Die Anzahl der verschiedenen Knotenniveaus in einem Baum wird als *Höhe des Baumes* bezeichnet.
- a) Entwerfen Sie eine OCaml-Typ-Deklaration für einen binären Baum mit wählbaren, aber einheitlichen Datenelementen. Der Datenwert soll hierbei das erste Typ-Tupleelement darstellen.
- b) Verwenden Sie die in a) entworfene Deklaration um den abgebildeten Binärbaum im Rechner zu speichern.



- c) Entwerfen Sie eine OCaml-Funktion, welche die Anzahl der Knoten in einem Binärbaum bestimmt.

### Aufgabe 19      **Induktion über den Termaufbau: Negationsnormalform**

Ein Boolescher Term ist in Negationsnormalform, wenn in ihm Negationszeichen nur unmittelbar vor Atomen oder vor Identifikatoren vorkommen. Zeigen Sie mit Induktion über den Termaufbau, daß sich jeder Boolesche Term auf einen semantisch äquivalenten Term in Negationsnormalform

reduzieren läßt. Es genügt beim Beweis, sich auf Terme zu beschränken, in denen als Operatoren nur  $\neg$ ,  $\vee$  und  $\wedge$  vorkommen. Die anderen faßt man als Schreibabkürzungen auf, ebenso die Konstanten `true` und `false`. Die Normalform ist natürlich nicht eindeutig, z.B. sind die folgenden semantisch äquivalenten Terme in Negationsnormalform.

$$(a \vee b) \Rightarrow c, (\neg a \wedge \neg b) \vee c, (\neg a \vee c) \wedge (\neg b \vee c)$$

## **Aufgabe 20**      **Schach: Bewegung der Figuren in OCaml**

Der Bauer ist ein Spielstein des Schachspiels. Jeder Spieler hat zu Partiebeginn 8 Bauern, die einen Wall vor den übrigen Figuren bilden. Umgangssprachlich gilt der Bauer nicht als Figur; als Figuren werden König, Dame, Läufer, Springer und Turm angesehen (die FIDE-Regeln unterscheiden allerdings nicht). Im Unterschied zu diesen kann der Bauer sich nie rückwärts bewegen, sondern nur immer weiter nach vorne. Der Bauer ist wegen seiner Zugmöglichkeiten einer der vielseitigsten Steine im Schachspiel. Wegen der Vielzahl der Bauern und deren langsamer Gangart könnte man annehmen, dass es sich um recht wertlose Steine handelt. Allerdings wird der Bauer im Verlauf einer Schachpartie immer stärker, insbesondere wegen der Möglichkeit, sich bei Erreichen der gegnerischen Grundreihe in eine stärkere Figur umzuwandeln.

- In der Ausgangsstellung kann sich der Bauer wahlweise einen Schritt nach vorne bewegen, sofern das Zielfeld leer ist, oder aber einen Doppelschritt vornehmen, sofern das Feld vor dem Bauern und das Zielfeld leer sind.
  - Befindet sich der Bauer nicht in der Ausgangsstellung (2. bzw. 7.Reihe), dann kann er sich nur um ein einziges Feld nach vorne bewegen (wenn er nicht schlägt).
  - Der Bauer schlägt diagonal. Der Bauer ist der einzige Spielstein, der in eine andere Richtung als die Zugrichtung schlägt.
  - Der Bauer kann sich nur vorwärts bewegen. Er ist der einzige Spielstein, der nicht auf ein bereits besuchtes Feld zurückkehren kann.
  - Der Bauer kann en passant schlagen. Man versteht darunter das Recht eines Spielers, einen gegnerischen Bauern auch dann zu schlagen, wenn er von der Ausgangsstellung heraus durch einen Doppelschritt über den Schlagbereich des eigenen Bauern hinaus zieht. In diesem Fall ist der Bauernzug so zu werten, als ob er nur einen Feldschritt von der Ausgangsstellung nach vorne getan hätte statt derer zwei. En passant darf nur unmittelbar nach dem Doppelschritt des Bauern geschlagen werden.
- a) Entwickeln Sie eine Funktion `findTile tile board`, die eine `int List` zurückgibt, in der die Positionen des entsprechenden `tile` gespeichert sind.
- b) Entwickeln Sie eine Funktion `getColumn position`, die zu einer `position` in der Liste die entsprechende `column` berechnet.
- c) Entwickeln Sie eine Funktion `getRow position`, die zu einer `position` in der Liste die entsprechende `row` berechnet.
- d) Entwickeln Sie eine Funktion `getPosition column row`, die zu einer `column` und einer `row` die entsprechende `position` berechnet.
- e) Definieren Sie einen neuen Typ `move`, der einen Zug repräsentiert.
- f) Entwickeln Sie eine Funktion `whichPiece tile`, die das `piece` aus `tile` zurückgibt.

- g) Entwickeln Sie eine Funktion `whichColor tile`, die die `color` aus `tile` zurückgibt.
- h) Entwickeln Sie eine Funktion `moveTop position board`, die einen Bewegung nach vorne (aus Sicht der weissen Steine) erlaubt, sofern das Feld leer ist. Rückgabe ist eine `int List`.
- i) Entwickeln sie analoge Funktionen `moveTopLeft`, `moveTopRight`, `moveRight`, `moveBottomRight`, `moveBottom`, `moveBottomLeft` und `moveLeft`.
- j) Entwickeln Sie analoge Funktionen (zum Beispiel: `beatTop position board color`), die eine schlagende Bewegung in die 8 möglichen Richtungen widerspiegeln. `color` ist hierbei die zu schlagende Farbe.
- k) Testen Sie Ihre Lösungen geeignet.

### **Aufgabe 21 (H) Balancierte Bäume in OCaml**

**(10 Punkte)**

Verwenden Sie für diese Aufgabe die Datenstruktur aus Aufgabe 18 a).

- a) Entwerfen Sie eine OCaml-Funktion, welche einen bestimmten Knoten in einem Binärbaum sucht, und dessen Niveau zurückgibt. Sie können davon ausgehen, dass der Baum sortiert und jeder Eintrag nur einmal im Baum vorhanden ist.
- b) Entwerfen Sie eine OCaml-Funktion, welche die Höhe eines Binärbaumes bestimmt.
- c) Ein Baum ist balanciert, wenn jeder Knoten die folgende Eigenschaft besitzt: Die Höhe des linken und rechten Teilbaumes unterscheiden sich höchstens um eins. Schreiben Sie eine OCaml-Funktion die prüft, ob der Baum balanciert ist.

### **Aufgabe 22 (H) Schach: Bewegung der Bauern in OCaml**

**(1 + 1 + 2 + 3 + 1 + 2 Punkte)**

Verwenden Sie für Ihre Hausaufgabe die Methoden, die Sie in der Übung implementiert haben.

- a) Entwickeln Sie das Gerüst einer Funktion `getPossibleMoves position board lastMove`, die alle möglichen Züge ausgehend von der Position `position` berechnen soll. Benutzen Sie hierbei die Fallunterscheidungen `Empty`, `Tile(White, Pawn)` und `Tile(Black, Pawn)`. Die übrigen Fälle sollten mit Hilfe der Fehlermeldung "not implemented" ausgewertet werden. Diese Fallunterscheidungen dienen Ihnen als Gerüst für die kommenden Teilaufgaben.
- b) Implementieren Sie eine inline Funktion `color`, die aus der Farbe an `position` die inverse Farbe berechnet. Erweitern Sie hierfür den entsprechenden `type`. Fügen Sie die Funktion an geeigneter Stelle in `getPossibleMoves position board lastMove` ein.
- c) Implementieren Sie die Doppelbewegung nach vorne. Hierfür müssen Sie die Einzelbewegung bereits implementiert haben.
- d) Implementieren Sie eine inline Funktion `passant`, die die "en passant" Bewegung von `Tile(White, Pawn)` darstellt. Die Position der Funktion ist hierbei entscheidend.
- e) Implementieren Sie die analoge inline Funktion `passant` für `Tile(Black, Pawn)`. Denken Sie daran diese an der richtigen Stelle zu plazieren.
- f) Setzen Sie die Bewegung der Bauern nun aus den gegebenen Funktionen zusammen.
- g) Testen Sie Ihre Programme geeignet.