Technische Universität München
Fakultät für Informatik
C. Buckl, D. Gmach, K. Schlüter,
Dr. M. Schneider

WS 2005/2006
Lösungsvorschläge zu Blatt 5
25. November 2005

Übungen zu Einführung in die Informatik I

Aufgabe 18 Bäume in OCaml (Lösungsvorschlag)

a) Entgegen der Vorlesung führen wir eine Baum-Definition ein, welche den Knoten nicht in der Mitte, sondern als erstes Element führt. Dies soll den Studenten zeigen, dass es mehrere unterschiedliche Realisations-Möglichkeiten gibt.

```
# type 'a btree = Empty | Node of ('a * 'a btree * 'a btree);;
type 'a btree = Empty | Node of ('a * 'a btree * 'a btree)
```

Die Deklaration von Typen mit `'a` sollte den Studenten aus der Vorlesung bekannt sein.

b)
```
# let sample_tree1 =
  Node(13,
       Node(7, Node(4,Empty,Node(5,Empty,Empty)),
               Node(10,Node(8,Empty,Empty),Empty)),
       Node(22, Node(15,Empty,Empty),
                Node(26,Empty,Node(29,Empty,Empty))));;
val sample_tree1 : int btree =
  Node
   (13,
    Node
     (7, Node (4, Empty, Node (5, Empty, Empty)),
      Node (10, Node (8, Empty, Empty), Empty)),
    Node
     (22, Node (15, Empty, Empty), Node (26, Empty, Node (29, Empty, Empty))))
```

c) Funktion zur Bestimmung der Anzahl der Knoten im Baum.

```
# let rec count btree =
  match btree with
    Empty     -> 0
  | Node(_, left, right) -> 1 + count left + count right;;
val count : 'a btree -> int = <fun>
# count sample_tree1;;
- : int = 10
```

Aufgabe 19 Induktion über den Termaufbau: Negationsnormalform (Lösungsvorschlag)

Vorbemerkung:
Aufgrund der Gesetze

$$\begin{aligned}(t_1 \Rightarrow t_2) &= (\neg t_1 \vee t_2)\\ (t_1 \Leftrightarrow t_2) &= (t_1 \wedge t_2) \vee (\neg t_1 \wedge \neg t_2)\\ true &= (x \vee \neg x)\\ false &= \neg true\end{aligned}$$

für beliebige Boolesche Terme (BT) t_1, t_2 und Identifikator x, können wir $\Rightarrow$, $\Leftrightarrow$, $true$, $false$ auf $\vee$, $\wedge$, $\neg$ zurückführen. Wir betrachten daher nur BT gebildet mit den Operatoren $\vee$, $\wedge$ und $\neg$ über Identifikatoren aus ID.

Induktion über den Termaufbau:
Aufgrund der induktiven Definition von BT gilt folgendes Induktionsprinzip für BT:
Sei $P(t)$ ein Prädikat über BT t. Wenn gilt

a) «««< .mine $P(t)$ gilt für alle Identifikatoren in ID. ======= $P(t)$ gilt für alle Identifikatoren in ID, d.h. für alle $t \in ID$. »»»> .r120

b) Aus $P(t)$ folgt $P(\neg t)$ für alle BT t.

c) Aus $P(t_1)$, $P(t_2)$ folgen $P(t_1 \wedge t_2)$ und $P(t_1 \vee t_2)$ für alle BT t_1, t_2.

so gilt $P(t)$ für alle BT t.

Induktionsbeweis für Existenz der Negationsnormalform (NNF):
Wir definieren als spezielles Prädikat

$P(t) =$ "Die BT t und $\neg t$ lassen sich auf semantisch äquivalente BT in NNF reduzieren"

Induktionsbeginn (IB):
Zu zeigen ist (a) für unser spezielles P. Sei $t \in ID$. t und $\neg t$ sind bereits in NNF und jeweils zu sich selbst semantisch äquivalent.

Induktionsschritt (IS):
Zu zeigen sind (b) und (c) für unser spezielles P.
zu (b): Als Induktionshypothese (IH) gelte $P(t)$, d.h. t und $\neg t$ lassen sich auf semantisch äquivalente BT in NNF reduzieren. Zu zeigen ist $P(\neg t)$, d.h. $\neg t$ und $\neg(\neg t)$ lassen sich auf semantisch äquivalente BT in NNF reduzieren. Wegen $\neg(\neg t) = t$ folgt dies aus der IH.
zu (c): Als IH gelte $P(t_1)$ und $P(t_2)$. Äquivalente BT in NNF für t_1, t_2 seien s_1, s_2 und für $\neg t_1$, $\neg t_2$ seien dies r_1, r_2. Dann gilt

$$\begin{aligned}(t_1 \wedge t_2) &= (s_1 \wedge s_2) \text{ ist in NNF}\\ (t_1 \vee t_2) &= (s_1 \vee s_2) \text{ ist in NNF}\\ \neg(t_1 \wedge t_2) &= ((\neg t_1) \vee (\neg t_2)) = (r_1 \vee r_2) \text{ ist in NNF}\\ \neg(t_1 \vee t_2) &= ((\neg t_1) \wedge (\neg t_2)) = (r_1 \wedge r_2) \text{ ist in NNF}\end{aligned}$$

Somit gilt $P(t)$ für alle BT t nach dem obigen Induktionsprinzip.

Aufgabe 20 Schach: Bewegung der Figuren in OCaml (Lösungsvorschlag)

Eine mögliche Implementierung könnte folgendermaßen aussehen:

a) `findTile tile board`

```
let findTile tile board =
  let checkTile tile test =
    match tile with
      tile when (test = tile) -> true
      | _ -> false
  in
  let rec findPosition tile board position =
    match board with
    Border -> []
    | TileList(a, tail) when (checkTile tile a) ->
      position::findPosition tile tail (position + 1)
    | TileList(a, tail) -> findPosition tile tail (position + 1)
  in
  findPosition tile board 0;;
```

b) `getColumn`

```
let getColumn position = position mod 8;;
```

c) `getRow`

```
let getRow position = position / 8;;
```

d) `getPosition`

```
let getPosition column row = column + (8 * row);;
```

e) `move`

Der Typ `move` repräsentiert einen Zug durch die Anfangs- und Endkoordinaten (`startCol`, `startRow`, `endCol`, `endRow`). Zum Beispiel wird der Zug $(2,5) \rightarrow (4,5)$ durch $(2,5,4,5)$ repräsentiert.

```
type move = Move of (int * int * int * int);;
```

f) `whichPiece tile`

```
let whichPiece tile =
  match tile with
    Empty -> failwith "no piece"
    | Tile(color, piece) -> piece;;
```

g) `whichColor tile`

```
let whichColor tile =
  match tile with
    Empty -> failwith "no piece"
    | Tile(color, piece) -> color;;
```

h) `moveForward position board`
Wir definieren uns zunächst eine Hilfsfunktion `getTile position board`, die uns die Figur an der Position `position` liefert.

```
let rec getTile position board =
  match board with
    Border -> failwith "position is to high"
    | TileList(a, tail) when (position = 0) -> a
    | TileList(a, tail) -> getTile (position - 1) tail;;

let moveForward position board =
  match getTile (position + 8) board with
    Empty -> [position + 8]
    | _ -> [];;
```

i) `Die 7 anderen Richtungen`

```
let moveTopRight position board =
  match position mod 8 with
    7 -> []
    | _ -> (
      match getTile (position + 9) board with
        Empty -> [position + 9]
        | _ -> []
    );;

let moveRight position board =
  match position mod 8 with
    7 -> []
    | _ -> (
      match getTile (position + 1) board with
        Empty -> [position + 1]
        | _ -> []
    );;

let moveBottomRight position board =
  match position mod 8 with
    7 -> []
    | _ -> (
      match getTile (position - 7) board with
        Empty -> [position - 7]
        | _ -> []
    );;
```

```
let moveBottom position board =
  match getTile (position - 8) board with
    Empty -> [position - 8]
    | _ -> [];;

let moveBottomLeft position board =
  match position mod 8 with
    0 -> []
    | _ -> (
      match getTile (position - 9) board with
        Empty -> [position - 9]
        | _ -> []
    );;

let moveLeft position board =
  match position mod 8 with
    0 -> []
    | _ -> (
      match getTile (position - 1) board with
        Empty -> [position - 1]
        | _ -> []
    );;

let moveTopLeft position board =
  match position mod 8 with
    0 -> []
    | _ -> (
      match getTile (position + 7) board with
        Empty -> [position + 7]
        | _ -> []
    );;
```

j) Die 8 möglichen Schlagrichtungen

```
let beatTop position board color =
  match getTile (position + 8) board with
    Tile(test, _) when (test = color) -> [position + 8]
    | _ -> [];;

let beatTopRight position board color =
  match position mod 8 with
    7 -> []
    | _ -> (
      match getTile (position + 9) board with
        Tile(test, _) when (test = color) -> [position + 9]
        | _ -> []
    );;

let beatRight position board color =
  match position mod 8 with
```

```
    7 -> []
    | _ -> (
      match getTile (position + 1) board with
        Tile(test, _) when (test = color) -> [position + 1]
        | _ -> []
    );;

let beatBottomRight position board color =
  match position mod 8 with
    7 -> []
    | _ -> (
      match getTile (position - 7) board with
        Tile(test, _) when (test = color) -> [position - 7]
        | _ -> []
    );;

let beatBottom position board color =
  match getTile (position - 8) board with
    Tile(test, _) when (test = color) -> [position - 8]
    | _ -> [];;

let beatBottomLeft position board color =
  match position mod 8 with
    0 -> []
    | _ -> (
      match getTile (position - 9) board with
        Tile(test, _) when (test = color) -> [position - 9]
        | _ -> []
    );;

let beatLeft position board color =
  match position mod 8 with
    0 -> []
    | _ -> (
      match getTile (position - 1) board with
        Tile(test, _) when (test = color) -> [position - 1]
        | _ -> []
    );;

let beatTopLeft position board color =
  match position mod 8 with
    0 -> []
    | _ -> (
      match getTile (position + 7) board with
        Tile(Black, _) -> [position + 7]
        | _ -> []
    );;
```

Aufgabe 21 Balancierte Bäume in OCaml (Lösungsvorschlag)

a) Suche und Rückgabe des Niveaus eines Knotens:

```
# let rec niveau_recursive x level btree =
  match btree with
    Empty -> -1
  | Node(y, left, right)  when x = y -> level
  | Node(y, left, right)  when x < y ->
        niveau_recursive x (level+1) left
  | Node(y, left, right)             ->
        niveau_recursive x (level+1) right;;
val niveau_recursive : 'a -> int -> 'a btree -> int = <fun>
```

Und die Einbettungsfunktion:

```
# let niveau x btree =
  niveau_recursive x 0 btree;;
val niveau : 'a -> 'a btree -> int = <fun>
# niveau 15 sample_tree1;;
- : int = 2
```

Die definierte rekursive Funktion `niveau_recursive` verwendet mit `level` den Abstand des Knotens von der Wurzel. Zum Einstieg setzt die Funktion `niveau` diesen auf 0. Ein leerer Baum wird mit -1 gezählt.

b)
```
# let rec height btree =
  match btree with
    Empty -> 0
  | Node(_, left, right) -> 1 + max (height left) (height right);;
val height : 'a btree -> int = <fun>
# height sample_tree1;;
- : int = 4
```

c)
```
# let rec is_balanced btree =
        match btree with
        | Empty -> true
        | Node (_,left, right) ->
             if (abs((height left) - (height right))) < 2
                then true && is_balanced(left) && is_balanced(right)
                else false;;
val is_balanced : 'a btree -> bool = <fun>
# is_balanced sample_tree1;;
- : bool = true
```

Aufgabe 22 Schach: Bewegung der Bauern in OCaml (Lösungsvorschlag)

Eine mögliche Implementierung könnte folgendermaßen aussehen:

a) `getPossibleMoves position board lastMove`

```
let possibleMoves position board lastMove =
  match tile with
```

```
    Empty -> []
    | Tile(White, Pawn) -> []
    | Tile(Black, Pawn) -> []
    | _ -> failwith "not implemented";;
```

b) `color` Die geeignetste Stelle hierfür ist direkt nach der Definition der Funktion.

```
let color = match tile with
  Tile(White, _) -> Black
  | Tile(Black, _) -> White
  | _ -> None
in
```

c) `doubleForward`

- Tile(White, Pawn)

```
let forward = moveTop position board in
let doubleForward =
  match forward with
    [a] when (position > 7 && position < 16) ->
      moveTop (position + 8) board
    | _ -> []
in
```

- Tile(Black, Pawn)

```
let forward = moveBottom position board in
let doubleForward =
  match forward with
    [a] when (position > 49 && position < 57) ->
      moveBottom (position - 8) board
    | _ -> []
in
```

d) `passant Tile(White, Pawn)` Die inline Funktion `passant` muss im Block der Fallunterscheidung Tile(White, Pawn) plaziert werden.

```
let passant =
  match lastMove with
    Move(_, startRow, column, row) when (getRow position = 4) -> (
      match getTile (getPosition column row) board with
        Tile(Black, Pawn) -> (
          match (startRow - row) with
            2 -> (
              match ((getColumn position) - column) with
                1 -> [position + 7]
                | -1 -> [position + 9]
                | _ -> []
```

```
              )
            | _ -> []
          )
        | _ -> []
      )
    | _ -> []
in
```

e) `passant Tile(Black, Pawn)` Die inline Funktion muss im Block der Fallunterscheidung Tile(Black, Pawn) plaziert werden.

```
let passant =
  match lastMove with
    Move(_, startRow, column, row) when (getRow position = 4) -> (
      match getTile (getPosition column row) board with
        Tile(Black, Pawn) -> (
          match (startRow - row) with
            2 -> (
              match ((getColumn position) - column) with
                1 -> [position + 7]
                | -1 -> [position + 9]
                | _ -> []
              )
            | _ -> []
          )
        | _ -> []
      )
    | _ -> []
in
```

f) Zusammensetzen Hier sind nur die fehlenden Teile aufgeführt.

- Tile(White, Pawn)

```
let beatLeft = beatTopLeft position board color in
let beatRight = beatTopRight position board color in
beatLeft @ forward @ doubleForward @ beatRight @ passant
```

- Tile(Black, Pawn)

```
let beatLeft = beatBottomLeft position board color in
let beatRight = beatBottomRight position board color in
beatLeft @ forward @ doubleForward @ beatRight @ passant
```