Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Lars Noschinski, Lars Hupel, Dr. Jasmin Blanchette

WS 2013/14
14.01.2014
Abgabe: 21.01.2014, 15:30

Einführung in die Informatik 2
12. Übung

Aufgabe G12.1 Huffman-Kodierung

In der Vorlesung wurde Huffmans Algorithmus zur Komprimierung von Daten vorgestellt.

Schreiben Sie ein Programm, das diesen Algorithmus verwendet, um Dateien zu komprimieren. Das heißt, implementieren Sie die folgenden Funktionen:

```
compress :: String -> FilePath -> IO ()
decompress :: FilePath -> IO (Maybe String)
```

`compress` xs $file$ soll den String xs komprimieren und das Ergebnis in der Datei $file$`.huff` ablegen. Der zur Dekomprimierung benötigte Baum soll in der Datei $file$`.code` landen. Umgekehrt soll `decompress` $file$ die beiden Dateien $file$`.huff` und $file$`.code` lesen und den dekomprimierten String zurückgeben.

Hinweis: Verwenden Sie das Modul `Data.ByteString` um die von Huffman ausgegebene Bitfolge platzsparend abzulegen. Hilfreich ist es dabei, Funktionen `toWord8` und `fromWord8` zu schreiben, die eine Liste von 8 Bits in ein `Word8` und zurück konvertieren.

Wir haben das Modul `Huffman` um Funktionen

```
serializeTree :: Tree -> ByteString
deserializeTree :: ByteString -> Tree
```

erweitert, die sie zum Ablegen des Kodierungsbaums verwenden können.

Aufgabe G12.2 HTML

Gegeben sei ein Datentyp zur Darstellung von einfachen HTML-Dokumenten:

```
data Html =
  Text String |
  Block String [Html]
```

Ein HTML-Dokument ist also

- entweder ein `Text`-Element mit einem `String`-Inhalt
- oder ein `Block`-Element mit einem Namen und einer Liste von geschachtelten HTML-Dokumenten.

Beispiele:

```
Text "Every string should learn to swim"
Block "head" []
Block "body" [Block "p" [Text "My cat"], Text "is not a float"]
Text "Sei Epsilon < 0"
```

Gesucht wird eine Funktion `plainHtml :: Html -> String` die solche Dokumente eine textuelle Darstellung umwandelt.

Ein Block mit Namen *name* und Inhalt $[h_1, h_2, \ldots, h_n]$ wird eingeleitet durch den String `<name>`, und beendet durch den String `</name>`. Zwischen diesen Markern steht dann die Zeichenkette $h_1^\star \cdots h_n^\star$, wobei $h_i^\star$ die textuelle Darstellung von h_i ist.

Ein Text-Element wird durch seinen Inhalt dargestellt. Um die Verwechslung mit Block-Elementen auszuschließen, müssen jedoch einige Ersetzungen vorgenommen werden:

- Die Zeichen `<` und `>` werden durch `<` und `gt;` dargestellt.
- Das Zeichen `&` wird durch `&` dargestellt.
- Aus historischen Gründen sollen Zeichen, die nicht im ASCII-Zeichensatz vorkommen (d.h. Zeichen c, für die $C =$ `Data.Char.ord` $c \geq 2^7$) ebenfalls ersetzt werden und zwar durch einen numerische HTML-Entität. Diese werden als `&#`C`;` dargestellt.

  Beispiel: `č` hat den Code 269 und wird deshalb als `&#269;` kodiert.
- Abweichend vom vorigen Punkt sollen die deutsche Umlaute wie in der Tabelle angegeben dargestellt werden.

Eingabe	Ausgabe
ä	`&auml;`
Ä	`&Auml;`
ö	`&ouml;`
Ö	`&Ouml;`
ü	`&uuml;`
Ü	`&Uuml;`
ß	`&szlig;`

Aufgabe H12.1 QWxsIHlvdXIgYmFzZSBhcmUgYmVsb25nIHRvIHVz (10 Punkte)

Damit E-Mail-Anhänge nicht durch Zeichensatz-Probleme auf dem Weg durchs Internet zerstört werden, nutzt der MIME-Standard das *Base64*-Verfahren[1], um Binärdaten in eine spezielle ASCII-Repräsentation umzuwandeln.

Man fasst dafür die Eingabe als eine Folge von Bits auf (in Haskell: `[Bool]`). Jeweils 3 aufeinanderfolgende Byte (entspricht 24 Bit) werden zusammengefasst und in vier Blöcke zu je 6 Bit aufgeteilt. Jeder dieser Teilblöcke wird dann gemäß einer Lookup Table[2] in ein einzelnes ASCII-Zeichen konvertiert (in Haskell: `[Char]`).

Wenn die Länge der Eingabeliste nicht durch 3 Byte teilbar ist, dann werden Nullbytes aufgefüllt. Vollständig aus Füllbytes bestehende Teilblöcke werden am Ende der Ausgabe als `=`-Zeichen ausgegeben.

[1] siehe auch `https://de.wikipedia.org/wiki/Base64`
[2] siehe `http://www.ietf.org/rfc/rfc2045.txt`, § 6.8

Eingabe	Ausgabe
(leer)	(leer)
`f`	`Zg==`
`fo`	`Zm8=`
`foo`	`Zm9v`
`foob`	`Zm9vYg==`
`fooba`	`Zm9vYmE=`
`foobar`	`Zm9vYmFy`

Ihre Aufgabe ist es, einen Base64-Encoder zu programmieren. Als Zwischenrepräsentation für eine Folge von Bytes eignet sich die Datenstruktur `ByteString` aus dem Paket `Data.ByteString`. Informationen zur Handhabung von `ByteString` finden Sie in der Übungsschablone.

Schreiben Sie zunächst eine Funktion `base64 :: ByteString -> [Char]`, die die eigentliche Codierung durchführt. Implementieren Sie zusätzlich eine IO-Aktion `main :: IO ()`, die den gesamten Text von der Standardeingabe liest und anschließend die den Base64-kodierten Text wieder ausgibt.

Um Ihr Programm selbst zu testen (unter Unix-Systemen), können Sie Ihre Ausgabe mit der von dem Kommandozeilenprogramm `base64` vergleichen. Dieses Tool steht auch auf `fp.in.tum.de` zur Verfügung.

Aufgabe H12.2 Des chiffres et pas des lettres (10 Punkte)

Gegeben sei eine Liste von Ganzzahlen (die Eingabezahlen) sowie eine zusätzliche Ganzzahl (die Zielzahl). Gesucht ist eine Funktion, die prüft, ob sich mittels der Rechenoperationen Addition, Subtraktion und Multiplikation aus den Eingabezahlen die Zielzahl errechnen lässt. Die Liste der Eingabezahlen darf Duplikate enthalten, allerdings darf jede Eingabezahl nur höchstens einmal verwendet werden.

Das Resultat Ihrer Funktion soll natürlich nachprüfbar sein. Deswegen ist in der Lösungsschablone eine Datentyp definiert:

```
data Expr = Input Integer |
            Add Expr Expr |
            Sub Expr Expr |
            Mul Expr Expr
```

1. Schreiben Sie eine Funktion `checkExpr :: [Integer] -> Integer -> Maybe Expr`, die, falls eine Lösung existiert, den Lösungsweg zurückgibt, andernfalls `Nothing`.
2. Oft sind auch näherungsweise Lösungen erwünscht. Schreiben Sie eine weitere Funktion `approxExpr :: [Integer] -> Integer -> Expr`, die stets ein Resultat liefert. Der zurückgegebene Lösungsweg soll eine der besten verfügbaren Lösungen liefern, wobei die Güte einer Lösung als Absolutbetrag der Differenz zwischen Ziel und tatsächlichem Wert definiert ist.

Aufgabe H12.3 Othello (zweiwöchige Wettbewerbsaufgabe, 0 Punkte)

Diese vorletzte, zweiwöchige Aufgabe besteht darin, künstliche Intelligenz (KI) für das Spiel *Othello* zu programmieren. Der Master of Competition hat die Idee von seinem Kollegen Dr.

Tjark Weber übernommen. Teile der Erklärungen unten sind mit leichten textuellen Änderungen dem Wikipedia-Artikel über das Spiel[3] entnommen.

Othello ist ein Brettspiel für zwei Personen, die *schwarze* bzw. *weiße* Spielsteine benutzen. Das Brett besteht aus 8×8 Feldern. Jedes Feld kann entweder frei oder mit einem (schwarzen oder weißen) Stein besetzt sein. Die Felder sind als (*Zeile*, *Spalte*) indiziert, wobei die Nummerierung bei 0 beginnt. Zu Beginn des Spiels sind die vier mittleren Felder besetzt: $(3, 3)$ und $(4, 4)$ sind weiß; $(4, 3)$ und $(3, 4)$ sind schwarz. Alle anderen Felder sind frei.

Die Spieler sind abwechselnd an der Reihe. Schwarz fängt an. Ein Zug besteht darin, dass der Spieler einen Stein seiner Farbe auf ein freies Feld setzt. Der Spieler darf nur so setzen, dass ausgehend von dem gesetzten Stein in beliebiger Richtung (senkrecht, waagerecht oder diagonal) ein oder mehrere gegnerische Steine anschließen und danach wieder ein eigener Stein liegt. Es muss also mindestens ein gegnerischer Stein von dem gesetzten Stein und einem anderen eigenen Stein in gerader Linie eingeschlossen werden. Dabei müssen alle Felder zwischen den beiden eigenen Steinen von gegnerischen Steinen besetzt sein. Alle gegnerischen Steine, die so eingeschlossen werden, wechseln die Farbe, indem sie umgedreht werden. Ein Zug kann mehrere Reihen gegnerischer Steine gleichzeitig einschließen, die dann alle umgedreht werden.

Der Spieler, der an der Reihe ist, muss einen Stein setzen. Hat er keine Möglichkeit, einen Stein regelkonform zu setzen, muss er passen. Müssen beide Spieler unmittelbar nacheinander passen (z.B. wenn das Brett voll ist), ist das Spiel beendet. Der Spieler, der am Ende die meisten Steine seiner Farbe auf dem Brett hat, gewinnt. Bei gleicher Zahl ist das Spiel unentschieden.

Jetzt konkreter zu Ihrer Aufgabe. Sie besteht darin, eine Datei namens `Othello_Player.hs` einzureichen. Diese muss auf dem folgenden Modul basieren:

```
module Othello_Base where

data Color = Black | White
  deriving (Eq, Show)

data Move = Pass | Play (Int, Int)
  deriving (Eq, Show)
```

Ihr Modul `Othello_Player` sollte wie folgt aussehen:

```
{-WETT
  ...
-}

module Othello_Player (State, startState, nextState) where
import Othello_Base

data State = ...
```

[3] `http://de.wikipedia.org/wiki/Othello_(Spiel)`

```
startState :: Color -> State
startState whoAmI = ...

nextState :: State -> Move -> (Move, State)
nextState opponentMove oldState = ...
```

Der Typ `State` stellt den internen Zustand Ihres Programmes dar.

Die Funktion `startState` nimmt die Farbe des KI-Spielers als Argument und gibt den Startzustand zurück.

`nextState` nimmt zwei Argumente: der alte Zustand und der aktuelle Zug des Gegners. Am Spielanfang ist der „aktuelle“ Zug `Pass`. Die Funktion gibt ein Paar (*Zug*, *Zustand*) zurück, in dem *Zug* der von der KI gewünschte Zug und *Zustand* der neue Zustand nach diesem Zug (und nach dem vorherigen Zug des Gegners) darstellt. Dieser Zustand wird (zusammen mit der Antwort des Gegners auf *Zug*) im nächsten `nextState`-Aufruf als Argument angegeben. *Zug* darf dann und nur dann `Pass` sein, wenn kein Stein gesetzt werden kann.

Falls die KI schwarz spielt, sieht die Sequenz der Aufrufe wie folgt aus:

```
let state0 = startState Black
let (myMove1, state1) = nextState state0 Pass
let (myMove2, state2) = nextState state1 opponentMove1
let (myMove3, state3) = nextState state2 opponentMove2
...
```

Falls die KI weiß spielt, sieht die Sequenz der Aufrufe wie folgt aus:

```
let state0 = startState White
let (myMove1, state1) = nextState state0 opponentMove1
let (myMove2, state2) = nextState state1 opponentMove2
...
```

Ganz oben in der `Othello_Player.hs`-Datei sollte ein Kommentar der Form

```
{-WETT
  ...
-}
```

stehen. Dieser sollte Ihre Implementierung beschreiben und relevante Quellenangaben enthalten.

Um Ihre Implementierung zu testen, können Sie den einfachen Treiber `Othello_Driver` nutzen (siehe Übungswebseite). Er setzt voraus, dass `State` die Typklasse `Show` unterstützt.

Die (Co(ntra)-)MCs werden ein Turnier mit allen eingereichten KI-Programmen organisieren. Jede KI wird gegen jede KI spielen, einmal als schwarz und einmal als weiß. Falls Ihre Implementierung einen falschen Zug zurückgibt oder sonst Fehler aufweist, werden Sie disqualifiziert.

Für jedes Spiel wird Ihre KI eine Punktzahl zwischen $+64$ und -64 erhalten:

$$\textit{Punktzahl} = \langle \textit{Anzahl Felder Ihrer Farbe} \rangle - \langle \textit{Anzahl Felder der anderen Farbe} \rangle$$

Jedem Programm werden fünf Minuten pro Spiel zur Verfügung gestellt. Es darf keine neuen Threads oder Prozesse starten. Bei Timeout oder anderen technischen Problemen bekommt der Gegner +64 und Sie −64 Punkte. Die Hardware des MCs ist fantastisch, trotzdem sollten Sie es nicht übertreiben.

Hinweis 1: Es existiert eine Fülle von Verfahren, um dieses Problem zu lösen. Statt das Rad neu zu erfinden, könnten Sie überlegen, einen etablierten Ansatz zu implementieren (Stichwort *Computer Othello*[4]). In diesem Fall sollten Sie nicht vergessen, Ihre Lösung mit einer Quellenangabe zu vorsehen. Es wird aber erwartet, dass Sie den Code selbst (eventuell als Mitglied eines Teams) geschrieben haben.

Hinweis 2: Dr. Weber rät: Do not go overboard. We understand that one could spend a lifetime perfecting an Othello AI, while you only have a few days.

Wichtig: Wenn Sie diese Aufgabe als Wettbewerbsaufgabe abgeben, stimmen Sie zu, dass Ihr Name ggf. auf der Ergebnisliste auf unserer Internetseite veröffentlicht wird. Sie können diese Einwilligung jederzeit widerrufen, indem Sie eine Email an `fp@fp.in.tum.de` schicken.

[4] z. B. `http://en.wikipedia.org/wiki/Computer_Othello`