Funktionell programmering

Laboration #2: ***Åttaspel***

Deadline 1: 211004 klockan 13:00

Syftet med den här laborationen är att ge er fortsatt träning i att stegvis utveckla ett program bestående av en mängd funktioner. Ni kommer i den här labben även få träning i hur omfattande och komplexa funktioner kan delas upp i många mindre problem och funktioner. Ni kommer även i denna laboration få hjälp med vilka funktioner som skall tillverkas och i vilken ordning. I ett skarpt fall är så klart uppdelningen i funktioner, och hur de bygger på varandra, en mycket viktig del av utvecklingen.

Bakgrund – Åttaspel

Laborationen behandlar ett åttaspel. Ett åttaspel är en förenklad variant av ett femtonspel, som ni kan se i figuren nedan. Skillnaden mot ett femtonspel är att ett åttaspel består av ett 3x3 bräde med åtta flyttbara brickor medan ett femtonspel består av ett 4x4 bräde med femton flyttbara brickor.

Figur 1: Ett femtonspel.

Målet i ett åttaspel är att flytta runt de 8 brickorna så att de blir sorterade från 1 till 8, följt av den tomma platsen. Ett sätt att representera omflyttningen av brickorna är att fokusera på hur den tomma rutan förflyttar sig. Vi kan på det här sättet definiera en sekvens av olika brickförflyttningar genom att beskriva hur den tomma platsen rör sig under spelets gång.

Uppgifter

I den här laborationen så ska ni skapa en representation för ett åttaspel och utveckla en funktion som tar en osorterad bricka och returnerar hur den tomma platsen ska flyttas runt för att en given spelkonfiguration ska lösas. Utför varje uppgift nedan i rätt ordning så kommer du att ha en sådan lösning när du är klar.

Uppgift 1

För att göra vår kod så snygg som möjligt så börjar vi med att definiera olika typalias för våra olika konstruktioner. Vi kommer att behöva 3 nya typer. En typ som representerar ett spelbräde, som förslagsvis är en lista av listor. En typ för att beskriva en position i listan, ett par av heltalsvärden. Till sist behöver vi också en typ som beskriver ett speltillstånd som innehåller både ett spelbräde och historiken över tidigare handlingar. Kalla dessa nya typer för **Board**, **Position** och **State**. Vänta med att definiera klart vad en **State** är tills att du gjort klart uppgift 2.

Uppgift 2

Vi kommer att behöva en datatyp för de olika handlingarna som går att genomföra. En sådan klass är given i kodstycket nedan. Den saknar dock dokumentation, så din uppgift är att föra in koden nedan i ditt program, förstå vad den gör, samt att skriva en kortare förklaring av vad koden gör.

```haskell
--- Skriv din förklaring här!
data Action = L | U | R | D deriving Eq
instance Show Action where
  show L = "Left"
  show U = "Up"
  show R = "Right"
  show D = "Down"
```

Uppgift 3

Definiera en konstant med namnet **solution** som innehåller en löst bricka. Ett tips är att representera det tomma hålet på brickan med ett värde som inte finns med på brickan, till exempel -1. Definiera sedan en funktion **isSolved** som testar om en given bricka är löst eller inte.

Uppgift 4

Definiera en funktion **findEmpty** som tar ett spelbräde och returnerar positionen på den tomma brickan. Funktionen behöver inte hantera fallet när den tomma brickan inte finns med på spelbrädet.

Uppgift 5

Definiera en funktion **replace** som tar en position, ett värde och ett spelbräde och byter ut värdet på den angivna platsen i spelbrädet mot det nya värdet.

Uppgift 6

Definiera en funktion **swap** som tar två positioner och ett spelbräde och returnerar ett nytt spelbräde där brickorna på de två positionerna har bytt plats med varandra. Använd gärna **replace** som du nyss definierade för att lösa uppgiften.

Uppgift 7

Vi ska nu definiera en funktion som heter **makeMove** tar ett spelbräde och en handling (av typen Action) och flyttar platsen på den tomma brickan enligt handlingen. För att uppnå detta behöver vi ha en viss felhantering då vissa handlingar inte kommer att vara tillåtna för vissa spelbräden. Vi väljer därför att returnera en lista av spelbräden. När vi kan utföra handlingen så returnerar vi en lista som innehåller den ny konfigurationen på spelbrädet och när vi inte kan utföra en handling så returnerar vi den tomma listan. Detta är ett vanligt sätt att hantera felhantering i Haskell och vi kan se det som att funktionen returnerar listan av möjliga lösningar. Det kan vara en god idé att definiera en eller flera hjälpfunktioner som utför de olika delstegen (som till exempel att matcha de olika handlingarna till hur brädet ska ändras).

Uppgift 8

Definiera en funktion som heter **allFutures** som tar ett tillstånd (av typen State) och returnerar alla möjliga tillstånd en handling framåt. Glöm inte bort att du måsta uppdatera historiken i tillstånden också.

Uppgift 9

Definiera en funktion som heter **possibleSolutions** som tar ett spelbräde (startkonfigurationen) och returnerar en oändlig lista av listor med alla möjliga framtida tillstånd. Listan på plats 0 i **possibleSolutions** ska innehålla alla möjliga tillstånd efter 0 handlingar. Listan på plats 1 i **possibleSolutions** ska innehålla alla möjliga tillstånd efter 1 handling framåt, och så vidare.

Uppgift 10

Definiera en funktion som heter **solve** som tar in ett spelbräde och returnerar den bästa lösningen på den givna brickan. Om du har definierat alla tidigare funktioner så blir den här funktionen inte så svår att definiera.

Uppgift 11

Definiera flera testfall för att testa om lösaren fungerar. Obs! det är inte alla konfigurationer som går att lösa så försäkra dig om att era testexempel går att lösa.

Organisation

Laborationen skall lösas i par. Skapa en grupp bestående av två personer i Canvas och lämna in laborationen från den gruppen. Inlämningen skall bestå av två filer:

1. Källkoden.
2. Väl valda körningsexempel.

Ni har tillgång till två handledningar per grupp. Dessa genomförs online och bokas via Canvas. På Canvas finns även ett diskussionsforum kopplat till laborationen där ni kan ställa generella frågor så att alla får samma information. Notera att ingen extra handledning kommer ges, exempelvis via e-mail. Slutligen, kom ihåg att samarbeten mellan grupper är förbjudet då laborationen utgör del av examinationen.