Lösningsförslag för Tenta i Programmeringsparadigm 2017-01-13 14.00-17.00

Funktionell Programmering

1. (a) Två lösningsförslag följer. Det första använder två hjälpfunktioner. *(6p)*

```
hofstadter::Int->[Int]
hofstadter 0 = []
hofstadter n = helper n [1] [2]

helper::Int->[Int]->[Int]->[Int]
helper 1 r _ = r
helper n r s =
  let nextr = (last r)+(last s)
      newr  = r++[nextr]
      nexts = counter ((last s)+1) newr in
    helper (n-1) newr (s++[nexts])

counter::Int->[Int]->Int
counter n forbidden
  |notElem n forbidden = n
  |otherwise = counter (n+1) forbidden
```

Nedan följer en alternativ lösning med bara en hjälpfunktion.

```
hofstadterSequence::Integer->[Integer]
hofstadterSequence n =
    hoffHelper (n-1) [1] [] 3

hoffHelper::Integer->[Integer]->[Integer]->Integer->[Integer]
hoffHelper 0 answer _ _ = answer
hoffHelper n answer diffs candidate =
    let newdiff = candidate - (last answer)
        used = answer ++ diffs
        in  if candidate `elem` used || newdiff `elem` used then
                hoffHelper n answer diffs (candidate + 1)
                else hoffHelper (n - 1) (answer ++ [candidate])
                        (diffs ++ [newdiff]) (candidate + 1)
```

(b) Denna lösning använder det andra lösningsförslaget ovanför: *(2p)*

```
hofstadterSequence 3
hoffHelper 2 [1] [] 3
hoffHelper 1 [1 3] [2] 4
hoffHelper 1 [1 3] [2] 5
hoffHelper 1 [1 3] [2] 6
hoffHelper 1 [1 3] [2] 7
hoffHelper 0 [1 3 7] [2 4] 8
```

2. (a) Tre lösningsförslag. Observera att Haskell är nollindexerat. *(2p)*

```
nthHoff::Int->Integer
nthHoff n =
    hofstadterSequence!!(n-1)

longerHoff::Int->Integer
longerHoff n =
    last (take n hofstadterSequence)

evenLongerHoff::Int->Integer
evenLongerHoff n =
    first (reverse (take n hofstadterSequence))
```

   (b) Lat evaluering. *(1p)*

3. Uppgiften kan lösas på flera olika sätt och vi listar tre av dem. Den första var den vi förväntade oss. Många frågade om nub och den eliminerar dubbletter i en lista. Den fungerar även om listan inte är sorterad så det går bra att kasta om funktionsanropens ordning. *(5p)*

```
import Data.List

squareCube::Int->[Int]
squareCube n =
    (nub . filter (<=n) . sort) [a^2+b^3|a<-[1..n],b<-[1..n]]
```

   Uppgiften kan också lösas med en enda listomfattning.

```
squareCube::Int->[Int]
squareCube n =
    [a|a<-[1..n],x<-[1..n],y<-[1..n],x^2+y^3==a]
```

   Om a står efter x och y i listomfattningen så behövs en sortering:

```
import Data.List

squareCube::Int->[Int]
squareCube n =
    sort [a|x<-[1..n],y<-[1..n],a<-[1..n],x^2+y^3==a]
```

4. (a) `Integral a => a -> Bool` *(2p)*

   (b) Lambdakalkyl. Vi godkänner också lambda calculus. *(1p)*

   (c) `[1,3,5,7,9]` *(1p)*

Logikprogrammering

5 (a) Mål: `reverse(L1, L2)` *(6p)*

1. Skapar instans av första regeln: `reverse([], [])`
2. Unifierar `L1=[], L2=[]`.

Första svaret: `L1=[], L2=[]`. För nästa svar, fortsätter:

3. Backtrack till senaste branch point (steg 1)
4. Skapar instans av andra regeln:
   `reverse([X1|Y1], Z1) :- reverse(Y1, Yrev1), append(Yrev1, [X1], Z1).`
   Unifierar: `L1=[X1|Y1], L2=Z1`.

4.1. Första delmålet: `reverse(Y1, Yrev1).`

4.2. Unifierar `Y1=[], Yrev1=[]` (på samma sätt som steg 1-2 ovan, eftersom både `Y1` och `Yrev1` är oinstansierade).

4.3. Andra delmålet: `append([], [X1], L2).`

4.4. Unifierar `L2=[X1]`

Andra svaret: `L1=[X1], L2=[X1]` för en ny oinstansierad variabel `X1`

(b) Mål: `reverse(X, [a, b]).` *(4p)*

1. Skapar instans av första regeln: `reverse([], []).`
   Kan inte unifiera `[a, b]` med `[]`.
   Går till nästa regel.
2. Skapar instans av andra regeln:
   `reverse([X1|Y1], Z1) :- reverse(Y1, Yrev1), append(Yrev1, [X1], Z1).`
   Unifierar: `X=[X1|Y1], Z1=[a, b]`.

2.1. Första delmålet: `reverse(Y1, Yrev1).`

2.1.1. Första lösning (enligt a-uppgiften):
`Y1=[], Yrev1=[]`

2.2. Andra delmålet: `append([], [X1], [a,b]).`

2.2.1. Misslyckas, backtrack till första delmålet.

2.1.1. Andra lösning (enligt a-uppgiften):
`Y1=[X2], Yrev1=[X2]` (för en ny variabel `X2`)

2.2. Andra delmålet: `append([X2], [X1], [a,b]).`

2.2.1. Unifierar: `X2=a, X1=b`

Första svaret: `X = [b | [a | []]]`, som utformateras som `X = [b, a]`.

6. (a) Möjlig lösning: *(5p)*

```
candVotes(_, [], 0).
candVotes(C, [C | Vs], N) :- !,
   candVotes(C, Vs, N1),
   N is N1+1.
candVotes(C, [_ | Vs], N) :-
   candVotes(C, Vs, N).
```

Utan snittet kommer lösningen även att säga att t.ex. `candVotes([a], [a], 0)` är sant, vilket är felaktigt, men inget avdrag ges för lösningar som har detta fel.

(b) Möjlig lösning: *(5p)*

```
electResult([], _, []).
electResult([C | Cs], Vs, [res(C, N) | Rs]) :-
   candVotes(C, Vs, N),
   electResult(Cs, Vs, Rs).
```

Formella språk och syntaxanalys

7 *(9p)*

(a) Nej, språket är inte reguljärt.
Bevis:

- Om språket (låt oss kalla det L) vore reguljärt så skulle också $\overline{L}$ också vara reguljärt. ($\overline{L}$ är komplementet till L, som innehåller alla strängar som *inte* inte är palindrom, dvs alla strängar som är palindrom.)
  Detta är ett grundläggande faktum som ingår i kursen och får användas utan motivering, men det är också enkelt att bevisa: om L är reguljärt finns en DFA för språket. Ta denna DFA och byt alla accepterande tillstånd till icke-accepterande, och vice versa (inklusive ett eventuellt fail-tillstånd, om automaten har ett sådant – detta blir nu ett "success-tillstånd"). Det är lätt att övertyga sig om att resulterande automat då känner igen $\overline{L}$.
- Men $\overline{L}$ är inte reguljärt, språket med alla palindrom var ett av de återkommande exemplen på icke-reguljära språk i kursen.
- Alltså är inte heller L reguljärt.

(b) Ja, språket är reguljärt.

Ett balanserat parentesuttryck har ett jämnt antal tecken. Med andra ord måste längden på strängen vara ett jämnt primtal. Det enda jämna primtalet är 2. Följaktligen har alla strängar i språket längd 2, och språket är alltså ändligt och därmed reguljärt. (Mer konkret innehåller det bara strängen "`()`").

(c) Ja, språket är reguljärt.

Vi kan konstruera en automat med 42 tillstånd, numrerade från 0 upp till 41, där 0 är start-tillstånd. Om vi befinner oss i tillstånd i och läser en siffra d (mellan 0 och 9) så går vi till tillståndet med nummer $(10 \cdot i + d) \bmod 42$. När vi läst ett tal n kommer då det tillstånd vi befinner oss i vara $n \bmod 42$. Ett tal n är delbart med 42 om $n \bmod 42 = 0$, dvs om vi befinner oss i tillstånd 0, och detta är därför det (enda) accepterande tillståndet.

Exempel: på indatasträngen 714 kommer vi gå som följer:

- Från tillstånd 0 på tecken 7 gå till $(10 \cdot 0 + 7) \bmod 42 = 7$
- Från tillstånd 7 på tecken 1 gå till $(10 \cdot 7 + 1) \bmod 42 = 71 \bmod 42 = 29$
- Från tillstånd 29 på tecken 4 gå till $(10 \cdot 29 + 4) \bmod 42 = 294 \bmod 42 = 0$

Slutar i tillstånd 0, alltså är 714 delbart med 42 (och mycket riktigt, $714/42 = 17$).

(Denna lösning, som den beskrivits, accepterar även tom sträng, vilket kanske inte är önskvärt, men denna detalj är enkel att fixa och det var inte nödvändigt att ha gjort det för att få full poäng.)

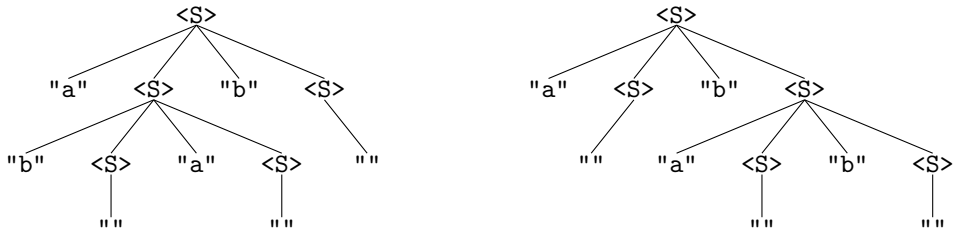
8 *(7p)*

(a) Här är några exempel (bara två behövde anges):

```
aaaabbbb
abababab
abbaabba
bbaabbaa
baaaabbb
```

(b) Språket består av alla strängar över tecknen 'a' och 'b' som har exakt lika många a:n som b:n.

(c) Exempelsträng: `abab`
Denna sträng kan härledas på följande två olika sätt:



9 *(4p)*

Möjlig lösning, i BNF-notation:

```
<A> ::= "0" <A> | "1" <B> | ""
<B> ::= "0" <C> | "1" <A>
<C> ::= "0" <B> | "1" <C>
```

Språket är det som genereras av icke-slutsymbolen `<A>`.
