Norsk

Universitetet i Bergen
Det matematisk-naturvitenskapelige fakultet

Eksamen i	:	INF-121A Funksjonell programmering
Dato	:	20 Februar 2014
Tid	:	9:00 – 12:00
Antall sider	:	2
Tillatte hjelpemidler	:	ingen

- Løsninger av delproblemer som du ikke har besvart kan antas gitt dersom de trenges i andre delproblemer.
- Forklar kort funksjoner som du selv innfører og angi deres type.
- Haskell-kode skal ikke bruke noen andre moduler enn *Prelude*.
- Prosentsatsene ved hver oppgave angir *kun omtrentlig* vekting ved sensur og forventet tidsforbruk/vanskelighetsgrad ved løsning.

1 Noen Haskell funksjoner (30%)

1.1. Definer en funksjon `maxsum::[[Int]]->Int` som tar en liste av tall-lister som input og velger ett tall fra hver liste slik at summen av valgte tall blir maksimal – denne summen er resultatet som returneres, f.eks.:
`maxsum [[1,2],[4,2],[3,6,5,9]] = 15` (dvs., summen av valgene: 2+4+9)
`maxsum [[1,0],[4,2,6],[3,7,5]] = 14` (dvs., summen av valgene: 1+6+7)

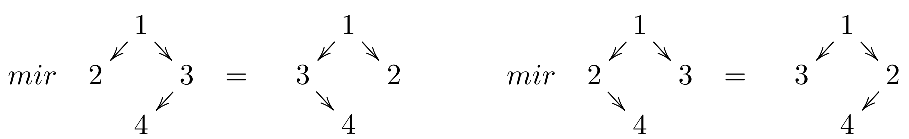
1.2. Definer en funksjon pack::`EQ(t) => [t]->[[t]]`, som pakker påfølgende duplikater fra inputlisten i separate sublister, f.eks.:
`pack "aabbbcdaaa" = ["aa","bbb","c","d","aaa"]`
`pack [1,2,2,1,1,3] = [[1],[2,2],[1,1],[3]]`

1.3. Bruk så `pack` til å definere en funksjon `lcode::EQ(t) => [t]->[(Int,t)]`, som komprimerer inputlisten ved å bytte ut hver (maksimal) delsekvens av påfølgende duplikater av `x::t` med et par `(i,x)`, der `i` er antallet duplikater av `x` i denne delsekvensen. For eksempel:
`lcode "aabbbcdaaa" = [(2,'a'),(3,'b'),(1,'c'),(1,'d'),(3,'a')]`.

2 Binære trær (30%)

2.1. Definer en Haskell datatype av binære trær, `BT a`, over en vilkårlig `EQ`-type `a`, der én av konstruktorene er `Empty`, og der hver node som ikke er `Empty` har 2 subtrær. Et blad er en node med begge subtrærne `Empty`.

2.2. Definer en funksjon `mir::(BT a)->(BT a)` som returnerer et speilbilde av argument-tre f.eks. (manglende subtrær betegner `Empty`),



2.3. Definer en funksjon `sym::(BT a)->Bool` som returner `True` hvis input-treet er symmetrisk (dvs. er sitt eget speilbilde), og `False` ellers.

2.4. Et binært søketre lagrer verdier av `Ord` type ved alle noder (som ikke er `Empty`) slikt at hver node holder verdien større enn alle verdier lagret i nodens venstre subtre og ikke større enn alle verdier lagret i nodens høyre subtre. F.eks., treet R til venstre er et søketre, mens treet G til høyre er det ikke:

$$\begin{tikzcd}[row sep=small, column sep=small] R & 2 \arrow[dl] \arrow[dr] & & \\ 1 & & 3 \arrow[dl] \arrow[dr] & \\ & 2 & & 4 \end{tikzcd} \qquad \begin{tikzcd}[row sep=small, column sep=small] & 2 \arrow[dl] \arrow[dr] & & G \\ 1 \arrow[dr] & & 3 \arrow[dr] & \\ & 2 & & 4 \end{tikzcd}$$

Definer en funkjson `cons::Ord(t) => [t]->(BT t)` som lager et binært søketre med alle elementer fra inputlisten. Avhengig av implementasjon, kan treet R over være resultat, f.eks., av kall `cons [4,2,3,1,2]` og/eller `cons [1,2,4,3,2]`.

3 Grammatikk og enkel parsing (40%)

3.1. Skriv en entydig grammatikk for et språk L over alfabetet $\{a, b, c\}$ som inneholder alle (og kun) strenger med minst to substrenger "ab". F.eks., følgende strenger er med i L:
 abab, ccbbabbbacaaacabba, abcabc, ...
mens følgende er ikke med i L:
 aba, abdab, aabb, ...

3.2. Tegn parsetre for strengen "aabab".

3.3. Definer en Haskell funksjon `aksept::String->Bool` som returnerer `True` hvis og bare hvis argumentstrengen er med i L fra forrige deloppgave, og `False` ellers. (Jo kortere definisjon av funksjonen og jo færre hjelpefunksjoner, desto bedre.)

3.4. Definer en Haskell funksjon `trans::String->String` som returnerer:
– en streng inneholdende "FEIL!" dersom input ikke er med i språket L, og ellers
– inpustrengen med alle forekomster av "ab" erstattet med "d".

Lykke til!
Michał Walicki

Universitetet i Bergen

Det matematisk-naturvitenskapelige fakultet

Norsk

Eksamen i	:	INF-121 Programmeringsparadigmer
Dato	:	20 Februar 2014
Tid	:	9:00 – 12:00
Antall sider	:	2
Tillatte hjelpemidler	:	ingen

- Løsninger av delproblemer som du ikke har besvart kan antas gitt dersom de trenges i andre delproblemer.
- Forklar kort funksjoner/predikater som du selv innfører – i Haskell, angi deres type.
- Haskell-kode skal ikke bruke noen andre moduler enn *Prelude*.
- Prosentsatsene ved hver oppgave angir *kun omtrentlig* vekting ved sensur og forventet tidsforbruk/vanskelighetsgrad ved løsning.

1 Haskell (25%)

1.1. Definer en funksjon `maxsum::[[Int]]->Int` som tar en liste av tall-lister som input og velger ett tall fra hver liste slik at summen av valgte tall blir maksimal – denne summen er resultatet som returneres, f.eks.:

`maxsum [[1,2],[4,2],[3,6,5,9]] = 15` (dvs., summen av valgene: 2+4+9)

`maxsum [[1,0],[4,2,6],[3,7,5]] = 14` (dvs., summen av valgene: 1+6+7)

1.2. Definer en funksjon pack::`EQ(t) => [t]->[[t]]`, som pakker påfølgende duplikater fra inputlisten i separate sublister, f.eks.:

`pack "aabbbcdaaa" = ["aa","bbb","c","d","aaa"]`

`pack [1,2,2,1,1,3] = [[1],[2,2],[1,1],[3]]`

1.3. Bruk så `pack` til å definere en funksjon `lcode::EQ(t) => [t]->[(Int,t)]`, som komprimerer inputlisten ved å bytte ut hver (maksimal) delsekvens av påfølgende duplikater av `x::t` med et par `(i,x)`, der `i` er antallet duplikater av `x` i denne delsekvensen. For eksempel:

`lcode "aabbbcdaaa" = [(2,'a'),(3,'b'),(1,'c'),(1,'d'),(3,'a')].`

2 Binære trær (25%)

2.1. Definer en Haskell datatype av binære trær, `BT a`, over en vilkårlig `EQ`-type `a`, der én av konstruktorene er `Empty`, og der hver node som ikke er `Empty` har 2 subtrær. Et blad er en node med begge subtrærne `Empty`.

2.2. Definer en funksjon `mir::(BT a)->(BT a)` som returnerer et speilbilde av argument-tre f.eks. (manglende subtrær betegner `Empty`),

$$mir\ \begin{tikzcd} & 1 \arrow[dl] \arrow[dr] & \\ 2 & & 3 \arrow[dl] \\ & 4 & \end{tikzcd} = \begin{tikzcd} & 1 \arrow[dl] \arrow[dr] & \\ 3 \arrow[dr] & & 2 \\ & 4 & \end{tikzcd} \qquad mir\ \begin{tikzcd} & 1 \arrow[dl] \arrow[dr] & \\ 2 \arrow[dr] & & 3 \\ & 4 & \end{tikzcd} = \begin{tikzcd} & 1 \arrow[dl] \arrow[dr] & \\ 3 & & 2 \arrow[dl] \\ & 4 & \end{tikzcd}$$

2.3. Definer en funksjon `sym::(BT a)->Bool` som returner `True` hvis input-treet er symmetrisk (dvs. er sitt eget speilbilde), og `False` ellers.

3 Prolog (25%)

"Definer predikat" betyr å programmere et Prolog predikat samt alle hjelpepredikater. Du kan bruke alle innebyggede predikater fra standard SWI-Prolog. Notasjon `pred(..+A..)` betyr at argumentet `A` antas instansiert ved kall av `pred`, mens mangel på en pluss, `pred(..B..)`, at argumentet må også kunne genereres ved et kall til `pred`.

Oppgaven er å programmere i Prolog predikater tislvarende funksjoner fra oppgave 1. Du kan anta at inputparametre ikke inneholder noen variabler ved kall.

3.1. Definer et predikat `maxsum(+L,R)` som holder hvis `R` er et resultat av funksjonen `maxsum` fra oppgave 1.1 anvendt på listen (av lister av tall) `L` (dvs., maksimal sum av enkelte tall valgt fra hver liste i `L`.)

3.2. Definer et predikat `pack(+L,R)` som holder hvis `R` er en liste av lister, med påfølgende duplikater fra listen `L`, dvs. er et resultat av funksjonen `pack` fra oppgave 1.2.

3.3. Definer predikat `lcode(+L,R)` som holder hvis `R` tilsvarer resultat av funksjonen `lcode` fra oppgave 1.3 anvendt på listen `L`.

4 Søketrær og unifikasjon (25%)

Vi betrakter to følgende Prolog relasjoner, `max1` og `max2`:

```
max1(X,Y,Y) :- X =< Y.          max2(X,Y,Y) :- X =< Y, !.
max1(X,_,X).                    max2(X,_,X).
```

4.1. Hva blir Prologs svar til følgende spørringene (når man trykker ; gjentatte ganger):

a) `max1(1,3,1).` c) `max1(X,3,3).` d) `max1(3,Y,3).`
b) `max2(1,3,1).` e) `max2(3,Y,3).` ?

4.2. Tegn søketrær for spørringer `max1(1,3,Z)` og `max2(1,3,Z)`. Husk å angi mgu på relevante grener. Skriv så resultater for begge spørringene når man trykker ; gjentatte ganger.

4.3. Gi alle stegene av unifikasjon med occurs_check og spesifiser resultatet på input:

`g(A,[H|T]) = g([a|T],[A,b]).`

Hva blir Prologs svar til denne spørringen?

Lykke til!
Michał Walicki

INF-121A (løsningsforslag)

```
-- 121/haskell/eks14.hs
-- 1.1
maxsum ls = sum [maximum l | l <- ls]

-- 1.2
pack [] = [[]]
pack [x] = [[x]]
pack (x:y:ls) = let (r:rs) = pack (y:ls) in
                 if (x==y) then (x:r):rs
                 else ([x]:r:rs)
-- 1.3
lcode [] = []
lcode ls = map (\x -> (length x, head x)) (pack ls)

-- 2.1
data Tree a = Em | Br a (Tree a) (Tree a)
              deriving (Show, Eq)

-- 2.2
mir Em = Em
mir (Br a l r) = Br a (mir r) (mir l)

-- 2.3
-- full score for use of EQ here - from the definition of BT
sym t = mir t == t

-- 2.4
cons [x] = Br x Em Em
cons (x:xs) = ins x (cons xs)

ins x Em = Br x Em Em
ins x (Br y l r) = if (x<y) then Br y (ins x l) r
                   else Br y l (ins x r)

-- 3.1
-- S := aB | bS | cS
-- B := aB | bC | cS
-- C := aD | bC | cC
-- D := aD | bF | cC
-- F := aF | bF | cF | e

--3.2

--3.3
aksept('a':rs) = aksB(rs)        aksB ('a':rs) = aksB(rs)
aksept('b':rs) = aksept(rs)      aksB ('b':rs) = aksC(rs)
```

```
aksept('c':rs) = aksept(rs)
aks x = False

aksB ('c':rs) = aksept(rs)
aksB x = False

aksC ('a':rs) = aksD(rs)
aksC ('b':rs) = aksC(rs)
aksC ('c':rs) = aksC(rs)
aksC x = False

aksD ('a':rs) = aksD(rs)
aksD ('b':rs) = aksF(rs)
aksD ('c':rs) = aksC(rs)
aksD x = False

aksF [] = True
aksF ('a':rs) = aksF(rs)
aksF ('b':rs) = aksF(rs)
aksF ('c':rs) = aksF(rs)
aksF x = False

-- For a full score: better, since simpler and shorter:
aksept(xs) = aksS(xs,0)
aksS('a':'b':xs,n) = aksS(xs,n+1)
aksS(x:xs,n) = if (elem x ['a','b','c']) then aksS(xs,n)
               else False
aksS([],n) = n > 1

--3.4
trans xs = tr (xs,0)
tr('a':'b':xs,n) = 'd':tr(xs,n+1)
tr('a':xs,n) = 'a':tr(xs,n)
tr('b':xs,n) = 'b':tr(xs,n)
tr('c':xs,n) = 'c':tr(xs,n)
tr(x:xs,n) = ' ':x:"-FEIL!"
tr([],n) = if (n>1) then "" else "-FEIL!"
```

INF-121

Problem 1 – solution

(For both 1 and 2, see solution to 121A)

Problem 3 – solution

3.1. `mexEl(+L,E)` gives the maximal element E from the list L, `maxs(+LL,L)` gives the list L with such maximal elements from each list in LL, and then `maxaux(+L,R)` gives in R the sum of all elements from L.

```
mexEl([X],X).
mexEl([H,Y|T],Z):- H =< Y, !, mexEl([Y|T],Z).
mexEl([H,Y|T],Z):- H > Y, mexEl([H|T],Z).
maxs([X],[Z]) :- mexEl(X,Z).
maxs([H,G|T],[Z,Y|Tm]) :- mexEl(H,Z), maxs([G|T],[Y|Tm]).
maxsum(L,X) :- maxs(L,M), maxaux(M,X).
maxaux([X],X).
maxaux([H|T],Y) :- maxaux(T,Z), Y is H+Z.
```

3.2.

```
pack([X],[[X]]).
pack([X,Y|T], [[X|R]|Z]):- X==Y, !, pack([Y|T],[R|Z]).
pack([X,Y|T], [[X]|Z]):- not(X==Y), pack([Y|T],Z).
```

3.3. If needed, we can define `head([H|_],H)`, and the rest as follows:

```
lcode(X,L) :- pack(X,R), conv(R,L).
conv([],[]).
conv([X|R],[(A,N)|L]):- head(X,A), length(X,N), conv(R,L).
```

Problem 4 – solution

4.1.

a) ?– max1(1,3,1). true .
b) ?– max2(1,3,1). true .
c) ?– max1(X,3,3). ERROR: =</2: Arguments are not sufficiently instantiated.
d) ?– max1(3,Y,3). Y=3 ; true.
e) ?– max2(3,Y,3). Y=3 .

4.2.

?– max1(1,3,Z). Z=3 ; Z=1 .
?– max2(1,3,Z). Z=3 .

4.3. One should rewrite it first to the term notaton to avoid ambiguities. Then:

g(A, .(H,T)) = g(.(a,T), .(A, .(b, [])))
A = .(a,T), .(H,T) = .(A, .(b,[]))
A = .(a,T), H = A, T = .(b,[])
A = .(a,.(b,[])), H = A, T = .(b,[])
A = .(a,.(b,[])), H = .(a,.(b,[])), T = .(b,[])

and rewriting back to the sugared syntax:

A = [a,b], H = [a,b], T = [b],

which is the resulting mgu.

Prolog will give the same answer, since the only possible difference concerns occurs_check, which Prolog does not apply, but which does not appear in this example.