KTH Information and
Communication Technology

ID1019
Johan Montelius

Programmering II (ID1019)
2014-03-10 14:00-17:00

Förnamn: _____

Efternamn: _____

Personnummer: _____

Instruktioner

- Du får inte ha något materiel med dig förutom skrivmateriel. Mobiler etc, skall lämnas till tentamensvakten.
- Svaren skall lämnas på dessa sidor, använd det utrymme som finns under varje uppgift för att skriva ner ditt svar.
- Svar skall skrivas på svenska.
- Du skall lämna in hela denna tentamen.
- Inga ytterligare sidor skall lämnas in.

Betyg

Tentamen har ett antal uppgifter där några är lite svårare än andra. De svårare uppgifterna är markerade med en stjärna, *poäng**, och ger poäng för de högre betygen. Vi delar alltså upp tentamen in grundpoäng och högre poäng. Se först och främst till att klara de normala poängen innan du ger dig i kast med de högre poängen. I svaren på frågorna i de högre poängen skall du visa att du behärskar området.
Skriv först lösningar på ett kladdpapper och skriv sedan ett väl formulerat svar i denna tentamen. Ett slarvigt skrivet svar ökar inte chanserna för högre poäng eller betyg.

Notera att det av de 24 grundpoängen räknas bara som högst 22 och, att högre poäng inte kompenserar för avsaknad av grundpoäng. Gränserna för betyg är enligt nedan.

- E: 14 grundpoäng
- D: 18 grundpoäng
- C: 22 grundpoäng
- B: 22 grundpoäng och 8 högre poäng
- A: 22 grundpoäng och 12 högre poäng

Kursens slutbetyg kan blir högre om man ligger nära en gräns och har skrivit mycket bra rapporter.

Erhållna poäng

Skriv inte här, detta är för rättningen.

Uppgift	1	2	3	4	5	Σ
Max G/H	4/-	8/4	4/4	4/4	4/4	24/16
G/H						

Totalt antal poäng:

Betyg:

Namn:_____Personnummer:_____

1 Datastrukturer och möstermatchning

1.1 en korthög [2 poäng]

Ge en representation av en hög med spelkort. Tänk på att vi skall kunna representera godtyckligt stora högar och att det skall var enkelt att avgöra färg (dvs spader, hjärter, ..) och valör på enskilda kort.

Svar:

```
-type color() :: heart|spade|club|diamond.
-type value() :: 1..13.
-type card() :: {color(), value()}.
-type dec() :: [card()].
```

1.2 vad är Y [2 poäng]

Vad är bindningen för `Y` i följande mönstermatchningar (var för sig), i de fall där matchningen lyckas (ett poängs avdrag för varje felaktigt eller uteblivet svar, svar skall ges i sin enklaste form):

- `[X,Y] = [1,2,3]`
- `[X,Z|Y] = [1,2,3,4]`
- `X = [], Y = [1|X]`
- `Y = [1|[2,3]]`
- `H = gurka, T = banan, Y = [H|T]`
- `X = {f, 42}, {G, Y} = X`
- `X = {f, 42}, Z = {g, X}, {_,{Y,_}} = Z`

Svar:

- misslyckas
- `Y = [3,4]`
- `Y = [1]`
- `Y = [1,2,3]`
- `[gurka|banan]`
- `Y = 42`
- `Y = f`

Namn:_____Personnummer:_____

2 Funktionell programmering

2.1 representera aritmetiska uttryck [2 poäng]

Hur skulle vi representera aritmetiska *uttryck*, som till exempel $2 + 3$ och $2+3*6$, om vi sedan skall arbeta med dessa och göra till exempel förenklingar mm.
Beskriv representationen för uttryck, där dess kan vara:

- heltal: 1,2,3 ...
- summa av uttryck: till exempel $2 + 3$ ...
- produkt av uttryck: till exempel $2 * 4$ ...

Notera att representationen skall vara så att man kan representera uttryck som till exempel $2+3*(4+6)$ (som naturligtvis är lika med $(2+(3*(4+6)))$). Använd med fördel Erlangs typ-notation i din beskrivning.

Svar:

```
-type const() :: {const, integer()}.
-type sum() :: {add, expr(), expr()}.
-type prod() :: {mul, expr(), expr()}.

-type expr() :: sum()|prod()|const().
```

2.2 evaluering av uttryck [2 poäng]

Med antagandet om representationen i föregående uppgift, implementera en funktion `eval/1` som tar ett uttryck och returnerar dess *resultat*. Om vi anropar funktionen med representationen för $2 + (3 * 4)$ skall vi returnera 14.

Svar:

```
eval({const, C}) -> C;
eval({add, E1, E2}) -> eval(E1) + eval(E2);
eval({mul, E1, E2}) -> eval(E1) * eval(E2);
```

2.3 varierande variabler [2 poäng]

Utöka representationen i uppgift **??** så att vi även kan hantera uttryck med variabler. Vi skall till exempel kunna representera uttryck så som: $2 + x$ och $2 + (3 * y)$. Beskriv hur variabler representeras och hur detta påverkar vår representation av uttryck. Använd gärna Erlangs typ-notation.

Svar:

```
-type var() :: {var, atom()}.

-type expr() :: sum()|prod()|const()| var().
```

2.4 variablers värde [2 poäng]

Beskriv representationen av en mängd *variabelbindningar*, dvs mappning från variabler till värden, och en funktion `lookup/2` som tar en variabel och en mängd variabelbindningar som argument och returnerar ett värde för den sökta variabeln. Vi antar att den sökta variabeln finns i mappningen.

Svar: Eftersom vi vet att vi skall mapa variabler till värden kan vi välja en enklare representation. Vi kan skapa en mappning mellan atomerna som identifierar variablerna till godtyckliga värden.

```
-type env() :: [{atom(), any()}].

lookup(Var,[{Var, Value}|_]) -> Value;
lookup(Var,[_|Rest]) -> lookup(Var,Rest).
```

2.5 evaluera med variabler [2 poäng*]

Implementera en funktion `eval/2` som tar ett uttryck och en en mängd variabelbindningar och returnerar värdet på det evaluerade uttrycket. Använd de representationer av variabler och variabelbindningar samt funktionen `lookup/2` i de föregående uppgifterna.

Svar:

```
eval({const, C}, _) -> C;
eval({var, V}, Env) ->  lookup(V, Env);
eval({add, E1, E2}, Env) -> eval(E1, Env) + eval(E2, Env);
eval({mul, E1, E2}, Env) -> eval(E1, Env) * eval(E2, Env);
```

2.6 derivering av uttryck [2 poäng*]

Implementera en funktion `deriv/2` som tar ett uttryck, representerat som i uppgifterna ovan, och en variabel, och returnerar derivatan av uttrycket med avseende på variabeln. Deriveringsreglerna är som bekant som följer:

- $c' = 0$ givet att c är en konstant eller en variabel skild från den variabel vi deriverar över
- $x' = 1$ givet att x är den variabel vi deriverar över
- $(f(x) + g(x))' = f'(x) + g'(x)$
- $(f(x) * g(x))' = f'(x) * g(x) + f(x) * g'(x)$

Svar: Detta kan tyckas komplicerat men det är naturligtvis mycket enkelt.

```
deriv({const, _}, _) -> 0
deriv({var, V}, V) -> 1;
deriv({var, Y}, _) -> {var, Y};
deriv({add, F, G}, V) -> {add, deriv(F, V), deriv(G, V)};
deriv({mul, F, G}, V) ->
       {add, {mul, deriv(F, V), G}
             {mul, F, deriv(G, V)}}.
```

3 Högre ordningens funktioner

3.1 foldin på ett träd [2 poäng]

Antag att vi har ett binärt träd representerat enligt formen nedan.

```
-type tree() :: empty | node().
-type node() :: {node, any(), tree(), tree()}.
```

Implementera en funktion `inorder/3` som traverserar ett träd i *in-order* (dvs från först vänstra grenen sen roten och sen högra grenen) och tillämpar en funktion på nodernas värde och en ackumulerande parameter.
Om vi har trädet `T` nedan:

```
T = {node, b, {node, a, empty, empty}, {node, c, empty, empty}}
```

och gör anropet:

```
inorder(F, gurka, T)
```

där `F` är en funktion. Så skall vi få resultatet:

```
F(c, F(b, F(a, gurka)))
```

Svar:

```
inorder(_, Acc, empty) -> Acc;
inorder(F, Acc, {node, V, L, R}) ->
    inorder(F, F(V, inorder(F, Acc, L)), R).
```

3.2 summan av ett träd [2 poäng]

Antag att vi har implementerat funktionen `inorder/3` i föregående uppgift. Implementera en funktion `sum/1` som använder sig av den funktionen och returnerar summan av alla värden i trädet. Om vi har trädet `T`:

```
T = {node, 7, {node, 5, empty, empty}, {node, 2, empty, empty}}
```

så så skall `sum(T)` returnera `14`.

Svar:

```
sum(T) ->
    F = fun(E,A) -> E+A end,
    inorder(F, 0, T).
```

3.3 funktionen reverse [4 poäng*]

Implementera funktionen `reverse/1` (som returnerar den omvända listan) enbart med hjälp av: `foldl/3`, `foldr/3` och/eller `map/2`. Du får inte använda dig av *++-operatorn* och skal inte heller skriva en egen rekursiv funktion utan ta hjälp av lämpliga högre ordningens funktioner.
Definitionen av de högre ordningens funktioner som du kan använda dig av återfinns i appendix.

Svar: reverse(X) -> foldl(fun(E,A) -> [E|A] end, [], X).

4 Komplexitet

4.1 är det så lönt [4 poäng]

Antag att vi skall arbeta på stora mängder av ord och att vi skall implementera en funktion som plockar ut ord som även är ord om de skrivs baklänges. Alla palindrom skall, som "apa", skall naturligtvis vara med men även ord som "galna" eftersom "anlag" är ett ord (dvs om det är med i den mängd ord vi arbetar med).
En lösning är att läsa in alla ord i en lista och sen plocka ord för ord, vända på det och se om det finns med i listan; enkelt att implementera.

En annan lösning är att skapa ett sorterat träd av alla orden och sen använda sig av trädet när man skall söka efter omvända ord. Det kostar ju en del att skapa ett sorterat träd så frågan är ju om det lönar sig.
Hur är det, skulle det löna sig att skapa ett sorterat träd? Vad kostar det och vad tjänar man? Vad måste man se upp med?

Svar: Det kommer att löna sig även för ganska små exempel. Att skapa ett sorterat träd har tidskomplexiteten $O(n\log(n))$, vi betalar alltså en faktor $\log(n)$ jämfört med att bara lägga in alla ord i en lista. När vi sen har vårt sorterade träd så kostar själva sökningen bara $O(\log(n))$ jämfört med $O(n)$ för en lista. Eftersom vi skall göra n sökningar så blir komplexiteten för sökningen $O(n\log(n))$ jämfört med $O(n^2)$.
För att få en uppskattning på hur mycket som vi tjänar kan vi ta $n = 8.000$, då får vi i ena fallet $13 * 8.000 = 104.000$ och i andra fallet 64.000.000. Att man tar en engångskostnad att sortera ordern torde i detta fall vara försumbart.
Det vi måste se upp med är att vi skapar ett så balanserat träd som möjligt. Om vi gör en enkel lösning och har otur (orden är redan sorterade när vi läser in dem) så kan vi få ett sorterat träd som i praktiken är en lista.

4.2 binärsökning i lista [4 poäng*]

Observera : detta är en något annorlunda lösning på ett sökproblem och ingenting som kan rekommenderas
Antag att vi har en ordnad lista av nyckel/värde-par där nycklarna är atomer och värdena heltal. Ett exempel är listan `L` nedan:

```
L = [{a,13},{b,42},{d,17},{f,32},{n,14}]
```

Vi har även en funktion `elem/2`, som hittar ett element i en lista givet en position, definerad enligt nedan.

```
elem(1,[H|_]) -> H;
elem(N,[_|T]) -> elem(N-1,T);
```

Vi antar att den nyckel som vi letar efter finns i listan så om vi anropar `elem(3, L)` där L är listan ovan så får vi resultatet `{d,17}`.
Vi definierar nu en sökfunktion `search/4` som använder binärsökning och hittar värdet för en nyckel mellan två positioner; vi antar att nyckeln finns i listan.

```
search(Key, First, Last, List) ->
  N = (Last - First) div 2 + First,
  {K, V} = elem(N, List),
  if
```

```
    Key == K -> V;
    Key > K  -> search(Key, N+1, Last, List);
    Key < K  -> search(Key, First, N-1, List)
  end.
```

Vi antar att vi vet hur lång listan är och anropar därför alltid funktionen *search(Key, 1, length_ of_ list, L)*. Om vi gör anropet `search(b, 1, 5, L)`, där L är listan ovan får vi resultatet 42.
Frågan är vilken asymptotisk tidskomplexitet funktionen `search/4` har.

Svar: Funktionen `search/4` har asymptotisk tidskomplexitet $O(n * \log(n))$ där n är längden på listan. Vi måste göra $\log(n)$ sökoperationer eftersom vi använder binärsökning men varje sökoperation är beroende av listans längd n. Om vi skulle göra en enkel linjär sökning genom listan så skulle vi ha en tidskomplexitet på $O(n)$, dvs bättre.

5 Concurrency

5.1 bin som samlar honung [2 poäng]

Vi skall modellera ett system där vi har ett antal bin som samlar honung i en skål. Skålen, björnen och varje bi representeras av olika processer som interagerar genom att skicka meddelanden till varandra.
Bin kan bara lämna honung om det finns plats i skålen. Skålen rymmer naturligtvis inte obegränsat med honung så när den är full, skall en björn väckas som kan sleva i sig all honung.
Implementera en process som beskriver skålen. Processen skall ha följande egenskaper:

- den vet vilken björn som sover
- den skall innehålla att antal *honungsenheter* som inte får överstiga ett max antal
- om skålen inte är full kan bin lämna honung i skålen (en enhet åt gången)
- om skålen blir full skall skålen *väcka* björnen
- när björnen har ätit upp honungen är skålen tom

Skålen vet naturligtvis vilken björn som skall väckas och vi kan anta att björnen vet vilken skål den skall äta ur när den vaknar. Vi har dock ett antal bin som levererar honung så skålen måste få reda på vilket bi det är som vill lämna honung. Du behöver inte skriva hur processen startas eller hur den får reda på vilken björn den skall väcka utan bara implementera den rekursiva funktion som beskriver skålens beteende.

Namn:_____Personnummer:_____

Svar:

```
open(Max, Max, Bear) ->
   Bear ! wakiwaki,
   full(Max, Bear);
open(N, Max, Bear) ->
   receive
       {honey, Bee} ->
           Bee ! ok,
           open(N+1, Max, Bear)
   end.

full(Max, Bear) ->
   receive
      hmrghh ->
         open(0, Max, Bear)
   end.
```

5.2 bin och björn [2 poäng]

Implementera en process för ett bi och en björn som kan arbeta med skålen i föregående uppgift. Du behöver inte ange hur processerna startas.

Svar:

```
bee(Dish) ->
  Dish ! {honey, self()},
  receive
     ok ->
        bee(Dish)
  end.

bear(Dish) ->
  receive
     waikwaki ->
        Dish ! hmrghh,
        bear(Dish)
  end.
```

5.3 start me up [4 poäng*]

Om vi skall starta upp våra processer så har vi nu ett dilemma, om vi startar skålen först så vet den inte vilken björn den skall väcka men om vi startar björnen först så vet den inte vilken skål den skall äta ur. Det finns lite olika lösningar på det hör problemet, beskriv två lösningar och ge exempel på de förändringar som vi kanske måste göra i vår implementation.

Svar: Vi kan starta björnen först och sen låta skålen skicka med sitt process-id när den väcker björnen. Björnen får då reda på vilken skål den skall äta ur. Detta ger oss möjlighet att ha flera skålar men björnen vet inte vilken skål som den skall äta ur om den inte blir väckt.

```
   Bear ! {wakiwaki, self()},
```

Vi kan starta björnen först och sen låta skålen presentera sig för björnen i ett inledande skede.

```
init(Max, Bear) ->
   Bear ! {hello, self()},
   open(0, Max, Bear).

 :

init() ->
   receive
```

```
      {hello, Dish} -> bear(Dish)
   end.
```

Vi kan starta björnen som i sin tur startar skålen och alla bin. Detta kan vara en bra metod men inte så bra om bina eller skålen skall vara kända för andra än björnen.

```
init(Max) ->
   Me = self(),
   Dish = spawn(fun() -> open(0,Max, Me) end),
   start_al_bees(Dish),
   bear(Dish).
```

Namn:_____Personnummer:_____

Appendix

- `map(F, L)`: returnerar listan av $F(E)$, där E är element i listan `L`
- `foldr(F, A, L)`: returnerar A_0 där A_k = `A`, $A_{i-1} = F(E_i, A_i)$, k är längden på listan `L` och E_i det *i:te* elementet i listan
- `foldl(F, A, L)`: returnerar A_k där A_0 = `A`, $A_i = F(E_i, A_{(i-1)})$, k är längden på listan `L`, och E_i det *i:te* elementet i listan
- `filter(F, L)`: returnerar listan av element E_i, där E_i är element i listan `L`, för vilka $F(E_i)$ returnerar *true*