KTH Information and
Communication Technology

ID1019
Johan Montelius

Programmering II (ID1019)
2015-03-13 09:00-12:00

Namn: _____

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.
Notera att det av de 24 grundpoängen räknas bara som högst 20 och, att högre poäng inte kompenserar för avsaknad av grundpoäng. Gränserna för betyg är enligt nedan.

- E: 14 grundpoäng
- D: 18 grundpoäng
- C: 20 grundpoäng
- B: 20 grundpoäng och 8 högre poäng
- A: 20 grundpoäng och 12 högre poäng

Gränserna kan komma att justeras nedåt men inte uppåt. 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/-	10/2	2/6	4/2	4/4	24/14
G/H						

Totalt antal poäng:

Betyg:

Namn:_____ Personnummer:_____

1 Datastrukturer och mönstermatchning

1.1 vad är Y [2 poäng]

Vad är bindningen för `Y` i följande mönstermatchningar (var för sig), i de fall där matchningen lyckas:

- `[X|Y] = [1,2,3]`
- `[X,Y] = [1,2,3]`
- `[X,Z|Y] = [1,2,3]`
- `X = {f, 42}, {G, Y} = X`
- `H = head, T = tail, Y = [H|T]`

Svar:

- `Y = [2,3]`
- ***misslyckas***
- `Y = [3]`
- `Y = 42`
- `Y = [head|tail]` **(inte** `[head, tail]`**, som många skrev)**

1.2 tal, lista, sträng eller tuple [2 poäng]

Man kan välja att representerar information med tal, listor, strängar eller tupler mm, lite beroende på hur informationen skall användas. För varje exempel nedan skriv ner hur du skulle representera informationen (du behöver inte använda type-notationen utan kan skriva exempel):

- **ett spelkort, t.ex. spader sju: Svar:** `{card, spade, 7}`
- **en student med namn och kth-id: Svar:** `{student, "Ola Salo", osalo}`
- **en mindre grupp studenter: Svar:** `en lista av studenter`
- **alla studenter registrerade på en större kurs där vi snabbt vill kunna söka efter studenter givet ett kth-id:**

  Svar: `Ett ordnat (efter kth-id) träd av studenter, att använda en lista ger oss inte en snabb sökning.`

Namn:_____ Personnummer:_____

2 Funktionell programmering

2.1 rövarspråket [4 poäng]

När jag var liten kunde man tala så att ingen utom de invigda förstod vad man sa:

> **"hoheloltot obobegogroripoploligogtot"**

Reglerna för det så kallade *rövarspråket* är enkla. Varje konsonant i ett ord byts ut mot: konsonanten, ett "o" och samma konsonant igen. Vokalena i ett ord lämnas som de är. Ordet "hej" blir "hohejoj" och "kanelbulle" skulle alltså kodas som "kokanonelolbobulollole". Jag har dessvärre förlorat förmågan att prata rövarspråket men du kanske kan skriva ett litet program som översätter en mening till rövarspråket.
Antag att:

- **strängar representeras som vanligt av en lista av ascii-värden**
- **meningar representerade som strängar, skrivna med gemener och utan punkt eller komma**
- **mellanslag finns i meningen och representeras av ascii-värdet 32**
- **bokastaven "o" kan skrivas `$o` eller med ascii-värdet 111**
- **vi har en funktion `sound/1` som returnerar `cons` eller `vowel` beroende på om en bokstav är en konsonant eller vokal**

Din uppgift är att skriva en funktion `rovare/1` som tar en mening och returnerar meningen skriven på rövarspråket.

Svar:

```
rovare([]) ->
    [];
rovare([32|Text]) ->
    [32|rovare(Text)];
rovare([Char|Text]) ->
    case sound(Char) of
cons ->
    [Char, $o, Char|rovare(Text)];
        vowel ->
    [Char|rovare(Text)]
    end.
```

2.2 en hög [2 poäng]

En "hög" (heap) kallas ett träd där alla del-träd (inklusive trädet själv) har sitt största element överst. Det betyder att man har direkt åtkomst åt högens största element men att det blir lite knepigare att ta bort det största elementet eller lägga till ett nytt. Algoritmen för att ta bort det största elementet (eller rättare sagt skapa en ny hög som innehåller alla element utom just det största) är som följer:

- **Om trädet har en tom höger- eller västergren så är det rätt så enkelt, om inte så får vi ta och titta lite på de två grenarna.**
- **Antag att högergrenen är den gren som har det största värdet i sin rot.**
- **Skapa ett nytt träd som har en rot med elementet taget från roten från den högra grenen**
- **Det nya trädets vänstergren är identisk med det ursprungliga trädets vänstergren.**
- **Det nya trädets högergren är det träd man får om man tar bort roten från den ursprungliga högergrenen.**

Man gör naturligtvis på samma sätt om det är den vänstra grenen som har det största värdet. Om man tillämpar dessa regler rekursivt så kommer det skapade trädet ha egenskaperna av en "hög". Beskriv hur trädet är representerat och definiera en funktion `pop/1` som givet en hög returnerar en tupel `{Elemen, Rest}`, där `Element` är det största elementet och `Rest` den hög som ges om vi tar bort elementet, eller `false` om det inte finns något element (trädet är tomt).

Svar:

```
pop(nil) ->
    false;
pop({node, E, nil, RH}) ->
    {E, RH};
pop({node, E, LH, nil}) ->
    {E, LH};
pop({node, E, LH, RH}) ->
    {node, L, _, _}=LH,
    {node, R, _, _}=RH,
    if
       L > R ->
```

```
        {L, Poped} = pop(LH),
        {E, {node, L, Poped, RH}};
      true ->
        {R, Poped} = pop(RH),
        {E, {node, R, LH, Poped}}
  end.
```

2.3 lägg till ett element [2 poäng]

Om vi skall lägga till ett element i högen så kan vi göra det enkelt för oss. Om det nya elementet är mindre än trädets rot så lägger vi till det i trädets vänstra gren. Om det är större en dess nuvarande rot så skapar vi ett nytt träd med det nya elementet som rot och lägger sedan den gamla roten i dess vänstra träd. Metoden har sina nackdelar men vi förbiser dem för tillfället. Definiera en funktion `insert/2` som tar ett element och en hög, och returnerar en ny hög där elementet har blivit tillagt.

Svar:

```
insert(N, nil) ->
    {node, N, nil, nil};
insert(Nn, {node, E, LH, RH}) when N > E ->
    {node, N, insert(E, LH), RH};
insert(N, {node, E, LH, RH})  ->
    {node, E, insert(N, LH), RH}.
```

2.4 lite väl vänstervridet [2 poäng]

Metoden ovan har ju problemet med att trädet kommer att vara ruskigt obalanserat, vi lägger ju aldrig till någonting i högergrenen. Gör en liten förändring som medför att vi kommer att lägga till element i omväxlande den vänstra och den högra grenen. n

Svar:

```
insert(N, nil) ->
    {node, N, nil, nil};
insert(N, {node, E, LH, RH}) when N > E ->
    {node, N,  RH, insert(E, LH)};
insert(N, {node, E, LH, RH})  ->
    {node, E,  RH, insert(N, LH)}.
```

2.5 lite mer eftertanke [2poäng*]

Metoden ovan är väl inte helt optimal, även om det fungerar bra givet den lilla insats vi behövde göra. Om vi pop:ar några element

från högen så kanske det hela tiden plockas upp element från dess vänstra gren, trädet skulle då bli lite obalanserat - något som vi inte kompenserar för när vi sen lägger till nya element. Om vi istället skall bemöda oss att försöka balansera trädet när vi lägger till nya element så måste vi hålla rätt på lite mer information.
Antag att varje nod innehåller inte bara ett element och de två delträden utan även en räknare som anger hur många element trädet har totalt. När vi skall lägga till ett nytt element så kan vi då välja att gå ner i den högra eller vänstra grenen beroende på vilken som är minst.
Implementera `insert/2` där du håller rätt på grenarnas storlekar och försöker hålla högen balanserad.

Svar:

```
insert(N, nil) ->
    {node, N, 1, nil, nil};
insert(N, {node, E, K, LH, RH}) ->
    LK = size_heap(LH),
    RK = size_heap(RH),
    if
        (N > E) and (LK < RK) ->
            {node, N, K+1, insert(E, LH), RH};
        (N > E) ->
            {node, N, K+1, LH, insert(E, RH)};
        (LK < RK) ->
            {node, E, K+1, insert(N, LH), RH};
        true ->
            {node, E, K+1, LH, insert(N, RH)}
    end.

size_heap(nil) -> 0;
size_heap({node, _, K, _, _}) -> K.
```

3 Evaluering av uttryck

Vi har under kursen arbetat med att beskriva hur ett språk kan definieras genom att formellt beskriva vilka termer, uttryck och datastrukturer vi har och hur vi med hjälp av regler kan beskriva vad som skall hända när vi evaluerar uttryck. De följande frågorna antar att vi har definierat ett litet funktionellt språk enligt de riktlinjer vi gått igenom.

Namn:_____ Personnummer:_____

3.1 evaluera ett uttryck [2 poäng]

Evaluera följande uttryck, antag att $\sigma = \{X/\{a, b\}, Y/c\}$.

- $E\sigma(c) \rightarrow$ **Svar:** c
- $E\sigma(X) \rightarrow$ **Svar:** $\{a, b\}$
- $E\sigma(\{X, Y\}) \rightarrow$ **Svar:** $\{\{a, b\}, c\}$

3.2 eq [2 poäng*]

Det vore väl rätt så bra om vi i språket kunde ha en inbyggd funktion som kunde avgöra om två termer vara lika. Vi vill kunna skriva sekvenser som:

```
X = a, Y = b, eq([X,b],[a,Y])
```

Detta skall evalueras till datastrukturen *true* (och om de inte var lika till *false*). Det betyder att vi utökar mängden datastrukturer med två värden: *true* och *false*, men vi måste även utöka mänden uttryck (expressions) och ha en regel som beskriver vad som skall göras när vi stöter på vår nya konstruktion. Våra uttryck kan vi utöka genom att lägga till följande regler:

⟨*eq*⟩ ::= **'eq('** ⟨*expression*⟩ **','** ⟨*expression*⟩ **')'**

⟨*expression*⟩ ::= ... | ⟨*eq*⟩

Vi måste även ha en regel som beskriver vad som skall göras när vi skall evaluera ett *eq-uttryck*. Vi tar för givet att vi kan avgöra om två *datastrukturer* är lika i.e. $s \equiv t$ **är antingen sant eller falskt. Hur skall vi skriva en regel för evalueringsfunktionen** E**?**

Svar:

- $E\sigma(eq(e_1, e_2) \rightarrow$ true **om** $s_1 \equiv s_2$ **eller**
- $E\sigma(eq(e_1, e_2) \rightarrow$ false **om** $s_1 \not\equiv s_2$ **där**
- $E\sigma(e_i) \rightarrow s_i$

3.3 if then else [4 poäng*]

Antag att vi har löst uppgiften ovan (uttryck kan evalueras till *true* eller *false*) men att vi nu vill utöka språket till att kunna hantera *if-then-else*. Vi vill kunna skriva sekvenser som:

```
X = b, if eq(X, a) then b else [X,c]
```

Hur skall vi utöka mängden av uttryck i språket så att det inkluderar dessa? Beskriv med hjälp av BNF-notation de utökningar som vi måste göra för uttryck.

Svar:

⟨*if-then-else*⟩ **::=** **'if'** ⟨*expression*⟩ **'then'** ⟨*expression*⟩ **'else'** ⟨*expression*⟩

⟨*expression*⟩ **::=** ... | ⟨*if-then-else*⟩

Nu till den knepigare delen, hur skall vi utöka evalueringsfunktionen så att vi kan hantera våra nya uttryck?

Svar:

- $E\sigma(if\ e_1\ then\ e_2\ else\ e_3) \rightarrow E\sigma(e_2)$ **om** $s_1 \equiv$ true **eller**
- $E\sigma(if\ e_1\ then\ e_2\ else\ e_3) \rightarrow E\sigma(e_3)$ **om** $s_1 \equiv$ false **där**
- $E\sigma(e_1) \rightarrow s_1$

4 Komplexitet

4.1 pop ur en hög [2 poäng]

Eftersom vi börjat med högar så kan vi väl fortsätta - vad är den asymptotiska tidkomplexiteten för funktionen `pop/1` som vi definierade förut?

Svar: Funktionen har tidskomplexitet $O(lg(n))$ där n är antalet element i högen. Varje operation måste, förutom att plocka bort översta elementet, låta element från en gren propagera uppåt. Längden på grenarna är naturligtvis $O(lg(n))$.

4.2 insert i en hög [2 poäng]

Vad är den asymptotiska tidskomplexiteten för funktionen `insert/2` som vi definierade förut?

Svar: Även `insert/2` har tidskomplexitet $O(lg(n))$ där n är antalet element i högen. Vi går ner i en gren för att hitta en plats att

lägga till elementet men förstätter sen att uppdatera grenen till dess slut i.e. $O(lg(n))$ steg.
(*, - ok, om vi tittar på vår naiva* `insert/2` *så kommer ju "trädet" att vara en lista och varje operation då ha en komplexitet på* $O(n)$*.)*

4.3 är detta vettigt [2 poäng*]

Varför skall vi bemöda oss med att implementera en hög, vi skulle ju lika bra kunna lösa problemet med en ordnad lista. Det största elementet ligger först och skall vi lägga till ett så får vi väl jobba lite.
Vad är den asymptotiska tidskomplexiteten för funktionerna `pop/1` **och** `insert/2` **om vi använder en ordnad lista för att representera en hög? Är det vettigt att använda en trädrepresentation, vinner vi egentligen någonting?**

Svar:Om vi använder en lista så skulle `pop/1` **ha tidskomplexitet** $O(1)$ **och** `insert/2` $O(n)$ **- det vi vinner på gungorna förlorar vi på karusellerna. Eftersom vi inte kan göra fler pop-operationer än vi har gjort insert-operationer (varje element som vi pop:ar har vi någon gång lagt till) så är komplexiteten för att lägga till, och senare ta bort, ett element $O(n)$. I fallet där vi har en trädstruktur så är motsvarande operationer $O(lg(n))$ i.e. betydligt bättre.**

5 Concurrency

5.1 ätande filosofer [2 poäng]

De klassiska problemet med de ätande filosoferna visar hur lätt vi kan hamna i en "dead-lock". Nedan är ett exempel på hur vi skull kunna starta upp fem filosofer; proceduren `chop:start/0` **startar en process som hanterar en pinne, proceduren** `filosopher:start/3` **startar en process om givet två pinnar försöker äta sig mätt. Alla filosofer implementerar samma algoritm: vakna, ta högra pinnen, ta vänstra pinnen, ät, lägg tillbaks pinnarna, filosofera.**

```
dinner() ->
   A = chop:start(),
   B = chop:start(),
   C = chop:start(),
   D = chop:start(),
   E = chop:start(),
   filospher:start(sokrates, A, B),
   filospher:start(plato, B, C),
   filospher:start(aristotoles, C, D),
```

```
   filospher:start(euklides, D, E),
   filospher:start(hippias, E, A).
```

Varför kommer dessa filosofer troligtvis att förr eller senare hamna i ett dead-lock? Gör en liten ändring i programmet ovan så att vi undviker problemet. Hur lyder en regel som man ibland kan följa för att undvika dead-lock?

Svar:
Vi får en dead-lock om alla filosoferna vaknar och tar sin högra pinne. Vi kan undvika problemet genom att låta till exempel Hippias startas `filospher:start(hippias, A, E)` dvs han lyfter sin vänstra pinne först.
Den generella regeln är att om alla tar "låsen" i samma ordning så undviker vi en situation där processer väntar på varandra. Det är inte alltid vi vet vilka lås som skall tas i förväg men väldigt ofta så fungerar denna enkla strategi.

5.2 parallell map [2 poäng]

Antag att vi har en lista med element och att vi skall göra två saker; dels skall vi tillämpa en funktion på varje element (map) och sedan skall vi samla ihop alla resultaten (reduce). Båda operationerna är tunga och vi skulle tjäna på att parallellisera beräkningarna.
Vi kan parallellisera map-fasen med följande implementation - där allt är givet utom funktionen `collect/3`. Argumentet `Map` är funktionen som skall tillämpa på varje element, `Red` är funktionen som skall reducera svaret, `Acc` är den initiala ackumulatorn och `Data` den lista av element som vi skall arbetet med.

```
pmap(Map, Red, Acc, Data) ->
   Self = self(),
   F = fun(D) ->
         Ref = make_ref(),
         spawn(fun() -> Self ! {res, Ref, Map(D)} end),
         Ref end,
   Refs = lists:map(F, Data),
   collect(Refs, Red, Acc).
```

Hur skulle du definiera funktionen `collect/3` om den skall samla in alla svaren och sedan "reducera" dem (vänster till höger) med hjälp av funktionen `Red` som tar två argument: ett resultat och det ackumulerade värdet.

Svar:

```
collect([], _, Acc) ->
   Acc;
collect([Ref|Refs], Red, Acc) ->
   receive
     {res, Ref, Res} ->
        collect(Refs, Red, Red(Res, Acc))
   end.
```

5.3 parallell reduce [4 poäng*]

Lösning ovan har nackdelen att alla reduceringar kommer att göras sekvensiellt. Vi kanske inte har så mycket val om reduceringarna måste göras i den specificerade ordningen men om reduktionsoperationen var associativ så skulle vi kunna utföra operationerna parallellt (associativitet ger att $((A + B) + C) + D$ kan utföras $(A + B) + (C + D)$).
Det kräver kanske att vi jobbar lite mer när vi sparkar igång mapprocesserna men vi antar att själva beräkningarna är så tunga att det inte gör så mycket om vi lägger lite extra jobb på att få strukturen rätt.
Definiera `ppmap/4` där vi plockat bort den initiala operatorn i.e. att reducera ett element är elementet självt. Det tredje argumentet är istället en process-id till vilken vi skall skicka resultatet av vår beräkningen (map och reduce). Vi struntar i fallet där listan av element är tom. Nedan är skelettkod som kan lösa problemet, fortsätt på den eller skriv en egen och antar att vi kan dela en lista på mitten med hjälp av `lists:split/1`.

```
ppmap(Map, _Red, Ctrl, [Elem]) ->


ppmap(Map, Red, Ctrl, Data) ->

    :

   {Data1, Data2} = lists:split(Data),

    :

   receive
     {res, Res1} ->
        receive
          {res, Res2} ->
```

```
        end
   end.
```

Svar:

```
ppmap(Map, _Red, Ctrl, [Elem]) ->
   Ctrl ! {res, Map(Elem)};
ppmap(Map, Red, Ctrl, Data) ->
   Self = self(),
   {Data1, Data2} = lists:split(Data),
   spawn(fun() -> ppmap(Map, Red, Self, Data1) end),
   spawn(fun() -> ppmap(Map, Red, Self, Data2) end),
   receive
     {res, Res1} ->
        receive
          {res, Res2} ->
              Ctrl ! {res, Red(Res1, Res2)}
        end
   end.
```