KTH Information and
Communication Technology

ID1218
Johan Montelius

Tillämpad Programmering (ID1218)
2013-06-04 09:00-13:00

Förnamn: _____

Efternamn: _____

Personnummer: _____

Regler

Du får inte ha något materiel med dig förutom skrivmateriel. Mobiler etc, skall lämnas till tentamensvakten.

Instruktioner

- Tentamen har totalt 36 poäng och skall skrivas på 4 timmar.
- Läs igenom hela tentamen innan du börjar.
- Svaren skall lämnas på dessa sidor, använd det utrymme som finns under varje uppgift för att skriva ner ditt svar. Inga ytterligare sidor skall lämnas in.
- Du skall lämna in hela denna tentamen.
- Svar skall skrivas på svenska.

Betyg

< 16	F
≥ 16	E
≥ 20	D
≥ 24	C
≥ 28	B
≥ 32	A

Erhållna poäng

Skriv inte här, detta är för rättningen.

Uppgift	1	2	3	4	5	6	Σ
Max	6	6	6	6	6	6	36
Poäng							

Totalt antal poäng:

Betyg:

Namn:_____Personnummer:_____

1 Funktionell Programmering i Erlang [totalt 6 poäng]

1.1 Hitta minsta värdet i ett träd [2 poäng]

Skriv en funktion *smallest(Tree)* som tar ett träd och returnerar det minsta värdet i trädet eller atomen *inf* om det anropas med ett tomt träd. Vi kan representera ett träd med antingen atomen *nil* (det tomma trädet) eller en tupel *{node, Value, Left, Right}* där värdet (Value) är ett heltal. Trädet är inte ordnat.
Till din hjälp kan du använda det faktum att $<$ *operatorn* ordnar alla heltal mindre än atomer: $11 < inf$ är alltså sant.

Svar:

```
smallest(Tree) -> smallest(Tree, inf).

smallest(nil, S) -> S;
smallest({node, Value, Left, Right}, Sofar) ->
   Smallest = if
                Value < Sofar -> Value;
                true -> Sofar
              end,
   smallest(Right, smallest(Left, Smallest)).
```

Namn:_____Personnummer:_____

1.2 K-värdet [2 poäng]

Långdistanslöpare pratar ibland om ett k-värde, som är det högsta tal som man har sprungit minst *k* kilometer minst *k* gånger. Om jag har ett k-värde på 24 så betyder det att jag har sprungit 24 kilometer eller mer minst 24 gånger. Till en början ökar man på sitt k-värde rätt snabbt men sen blir det tuffare.
Skriv en funktion *kvalue(Runs)*, som givet en lista med löppass i kilometer, räknar ut k-värdet. Givet listan

```
[13,5,7,11,6,4,8]
```

är k-värdet 5, eftersom vi sprungit fem lopp på minst fem kilometer Vi har inte k-värde 6 eftersom det bara finns fem pass på minst sex kilometer.

Svar: Om vi antar att vi kan sortera listan med största värden först så har vi denna enkla lösning.

```
kvalue(Runs)  ->
   kvalue(sort(Runs), 0).

kvalue([], K) -> K;
kvalue([R|Runs], K) ->
   if
     R < K -> K
     true -> kvalue(Runs, K+1)
   end.
```

Vi kan göra varianter på detta, som att sortera dem med minsta värdet först med nästan samma lösing, eller något mer komplicerat där man går igenom fall för fall.

Namn:_____Personnummer:_____

1.3 Mod av listor [2 poäng]

Skriv först en funktion *sub(L1, L2)* som tar två listor och returnerar en lista som är de element som är kvar av *L1* när vi har plockat bort lika många element som *L2* är lång. Om vi anropar funktionen:

```
sub([1,2,3,4,5], [a,b])
```

så får vi resultatet:

```
[3,4,5]
```

Om *L1* inte innehåller tillräckligt många element så skall atomen *no* returneras. Du får inte använda aritmetiska uttryck i din lösning.

Svar:

```
sub(Rest, []) ->
    Rest;
sub([], _) ->
    no;
sub([_|T], [_|R]) ->
    sub(T, R).
```

Observera ordningen på de två första klausulerna; detta för att smidigt fånga *sub([],[])*.
Använd nu funktionen *sub/2* och implementera funktionen *mod(List, Mod)* som tar två listor som argument och som returnerar den rest vi får från *List* om vi upprepade gånger plockar bort det antal element som finns i *Mod*. Om listan *Mod* är tom skall funktionen returnera *error*. Om vi anropar funktionen:

```
mod([1,2,3,4,5}, [a,b])
```

så skal vi få resultatet:

```
[5]
```

Svar:

```
mod(_, []) ->
    error;
mod(List, Mod) ->
    case sub(List, Mod) of
        no ->
            List;
        Rest ->
            mod(Rest, Mod)
    end.
```

Namn:_____Personnummer:_____

2 Högre-ordningens funktioner [6 poäng]

2.1 plock ut och summera [2 poäng]

Antag att vi har en lista med spelkort representeras av tupler på formen *{card, Suit, Value}*, där färgen, *Suit*, är en atom och valören, *Value*, ett tal mellan 1 och 13. Skriv en funktion *points(Cards, Suit)* som summerar alla valörer på kort av given färg. Listan av kort skall bara gås igenom en gång, dvs man kan inte först skapa en lista av alla kort av en given färg och sen gå igenom dessa för att summera poängen. Du skall, i möjligaste mån, använda dig av de högre ordningens funktioner: filter/2, map/2, foldl/3 och/eller foldr/3, se appendix för hur de anropas.

Svar:

```
points(Cards, Suite) ->
    foldl(fun({card, S, V}, N) ->
                if
                  S == Suit -> V+N;
                  true -> N
                end,
          0,
          Cards).
```

Man fick alltså inte göra en fold på en filter.

Namn:_____Personnummer:_____

2.2 traversera ett uttryck [2 poäng]

Ett uttryck är antingen ett värde eller en binär operation; exempel på uttryck kan vara: *5*, *4 + 4* och *5 + (3 - 2)*. Vi representerar uttryck med hjälp av en trädstruktur som antingen är ett värde *{value, N}*, eller en operation *{op, Op, Exp1, Exp2}*, där *Op* är en binär funktion. Uttrycket *5 + (3 - 2)* skulle då representeras med följande struktur:

```
{op, '+', {value, 5}, {op, '-', {value, 3}, {value, 2}}}
```

Skriv en högre ordningens funktion *traverse(Fun, Expr)* som traverserar ett uttryck i *post-order*: först vänster gren, sedan höger gren och sedan själva noden. Funktionen *Fun*, som appliceras skall ta ta ett argument som antingen är ett värde *{value, N}* eller en operation *{op, Op, V1, V2}* där *V1* och *V2* är resultatet av de traverserade grenarna.

```
Expr = {op, '+', {value, 5}, {op, '-', {value, 3}, {value, 2}}},
Fun = fun(E) ->
              case E of
                  {value, N} ->
                      integer_to_list(N);
                  {op, Op, V1, V2} ->
                      "(" ++ V1 ++ " " ++
                       atom_to_list(Op) ++
                       " " ++ V2 ++ ")"
              end
      end,
 traverse(Fun, Expr).
```

skall ge resultatet

```
"(5 + (3 - 2))"
```

Svar:

```
traverse(Fun, {value, N}) ->
    Fun({value, N});
traverse(Fun, {op, Op, Left, Right}) ->
      Fun({op, Op, traverse(Fun, Left), traverse(Fun, Right)}).
```

2.3 traversera ett uttryck [2 poäng]

Antag att vi har implementerat funktionen *traverse/2* i den föregående uppgiften men nu vill evaluera ett uttryck och returnera svaret. Vi vill kunna skriva på följande sätt

Namn:_____Personnummer:_____

```
Expr = {op, '+', {value, 5}, {op, '-', {value, 3}, {value, 2}}},
Fun = ???,
traverse(Fun, Expr).
```

och få svaret *6*. Hur skall funktionen *Fun* se ut för att vi skall kunna evaluera uttryck. Antag att vi begränsar oss till operationerna addition och subtraktion.

Svar:

```
Fun = fun(Exr) ->
         case Expr of
           {value, N} ->
                N;
           {op, '+', V1, V2} ->
                V1 + V2;
           {op, '-', V1, V2} ->
                V1 - V2
         end
       end
```

Namn:_____Personnummer:_____

3 Concurrency [totalt 6 poäng]

3.1 glidande medelvärde [2 poäng]

Skriv en funktion *sliding/0* i Erlang som startar en process. Processen skall ha ett tillstånd som gör det möjligt att räkna ut medelvärdet av de tre senaste meddelanden som mottagits. Processen skall startas i ett tillstånd där de tre senaste värden antas vara *0*. Processens tillstånd skall inte växa utan man skall kunna arbeta i ett konstant minne.
Processen skall kunna ta emot följande meddelande:

- **{msg, N}** : ändra tillståndet så att man kan beräkna *(N + N1 + N2)/3* där *N1* och *N2* är de två senaste meddelanden.
- **stop** : terminera processen

Svar:

```
sliding()  -> spawn(fun() -> sliding(0,0,0) end).

sliding(N1, N2, N3) ->
 receive
    {msg, N} ->
        sliding(N, N1, N2);
    stop ->
        ok
 end.
```

Man kan breäkna medel men vi behöver inte ha gjort det om ingen begär det.

3.2 returnera ett svar [2 poäng]

Utöka processen i uppgiften ovan så att vi kan skicka ett meddelande och få tillbaks ett meddelande med medelvärdet för de tre senaste värdena.

Svar:

```
sliding(N1, N2, N3) ->
 receive
    {msg, N} ->
        sliding(N, N1, N2);
    {request, From} ->
       From ! {average, (N1, N2, N3)/3},
       sliding(N1, N2, N3);
    stop ->
        ok
 end.
```

3.3 plocka upp svaret [2 poäng]

Visa hur en klient skulle kunna använda sig av processen, antag att den har en processidentifierare, *Pid*, till en skapad process. Svaret skall vara på formen:

```
  :
Pid ! ...
Average = ...
  :
```

Inga egendefinierade hjälpfunktioner skall användas och variabeln *Average* skall bindas till det returnerade medelvärdet.

Svar:

```
  :
Pid ! {request, self()},
Average = receive
            {average, X} -> X
          end,
  :
```

Namn:_____Personnummer:_____

4 Värden, pekare och arrayer i C++[totalt 6 poäng]

4.1 legala uttryck [2 poäng]

Vilka uttryck är legala uttryck i C++(som följer standarden C++03) ? Svara enbart ja eller nej, alla rätt ger 2 poäng, ett avdrag för varje fel.

1. `int y = 3;  int &x = y;`

   Svar: ja: vi definierar en referens x och låter den referera till y

2. `int y = 7; int *x = &y;`

   Svar: ja: inget problem att ta adressen av y, och det är en int-pekare vilket stämmer bra med x

3. `int &x;`

   Svar: nej: vi kan inte skapa en referens utan att säga vad den skall referera till

4. `int x=42; int** y = &(&x) - 2;`

   Svar: nej: vi kan inte ta adressen av en adress (inte ett l-värde)

5. `int y = 7; int *x = y;`

   Svar: nej: y är av typen int, men x har typen int*

4.2 värdet på x: [2 poäng]

Givet nedanstående uttryck, vad är värdet på variabeln x?

```
int a[5] = {3,2,1,5,4};
int x = a[1] + a[3];
```

Svar: Variabeln x får värdet 7 eftersom arrayer är noll-indexerade.

4.3 värdet på y: [2 poäng]

Givet nedanstående uttryck, vad är värdet på variabeln y?

```
int a[6] = {6,3,2,1,5,4};
int y = *(&a[3] - 2);
```

Svar: Variabeln y får värdet 3 eftersom (&a[3] - 2) är &a[1] och *(&a[1]) är a[1] dvs 3.

Namn:_____Personnummer:_____

5 Klasser och objekt [totalt 6 poäng]

I följande frågor antar vi att vi inkluderar biblioteket *iostream* och använder dess namespace *std*.

5.1 Bytt är bytt [2 poäng]

Nedan finns två versioner av en *swap-procedur* som är tänkt att byta värdet på två variabler. Vilken fungerar och varför fungerar den ena och inte den andra?

```
void swip(int &x, int &y) {
    int tmp;

    tmp = x;
    x = y;
    y = tmp;
}

void swup(int x, int y) {
    int tmp;

    tmp = x;
    x = y;
    y = tmp;
}
```

Svar: Den första fungerar eftersom proceduren har referenser som parametrar. Den kommer alltså att arbeta på *x* och *y* som om dessa var de verkliga variablerna. Den andra proceduren kommer inte att åstadkomma speciellt mycket eftersom de lokala variablerna *x* och *y* endast har kopior av värdena som proceduren anropas med.

5.2 Vilka objekt modifieras [2 poäng]

Givet programmet nedan, vad kommer att skrivas ut på `cout` och varför? Skriv en kort motivering till ditt svar.

```
class IntCell
{
  public:

     IntCell(int initialValue = 0 )
       { storedValue = new int(initialValue); }

     int getValue() const
       { return *storedValue; }

     void setValue(int val)
       { *storedValue = val;}

      const IntCell &operator=(const IntCell &rhs) {
         *storedValue = *(rhs.storedValue);
         return *this;
      }

  private:
       int *storedValue;
};

int main()
{
  IntCell a(2);
  IntCell b = a;
  IntCell c;

  c = b;

  a.setValue(4);

  cout << a.getValue() << endl;
  cout << b.getValue() << endl;
  cout << c.getValue() << endl;
}
```

Svar:
Det kommer att skrivas ut två stycken 4:or följt av en 2:a. Anledningen är att objektet *b* kommer att peka på samma int som *a* gör eftersom den gör en

grund kopia av *a* när den skapas. Här används en default-variant av en "copy constructor" som gör en grund kopia".
Vid den explicita tilldelning $c = b$, kommer däremot den användardefinerade operatorn att användas (som kanske inte är optimalt implementerad) och göra en djup kopia dvs. *c* kommer ha kvar sin int men denna int kommer att ha en kopia av *b:s* värde (dvs 2). När vi gör *a.setValue(4)* så kommer *a:s* int, som också är *b:s* int, att ändra värde. Den int som *c* pekar på kommer dock vara oförändrad.

Namn:_____Personnummer:_____

5.3 En student är också en person, eller? [2 poäng]

Givet följande definitioner:

```
class Person {
  string name;
public:
  Person(const string &nn) : name(nn) {};
  virtual void print() const {
    cout << name;
  };
};

class Student : public Person {
  string prgm;
public:
  Student(const string &nn, const string &cc) : Person(nn), prgm(cc) {}
  void print() const {
    Person::print();
    cout  << "/" << prgm;
  }
};

void print_person(const Person &p) {
  p.print();
};
```

Kommer programmet nedan att kompilera och om så, vad skrivs ut när vi kör programmet nedan?

```
int  main() {
  Student jim("Jim", "ICT");
  Person sue("Sue");

  print_person(jim);
  print_person(sue);
}
```

Svar: Programmet komplierar och kör utan problem. Programmet skriver ut "Jim/ICT" och "Sue" eftersom vi deklarerat metoden print som virtuell. När vi anropar print_person med vår student "Jim" så kommer vi göra en dynamisk dispatch och använda oss av den print metod som finns definerad för *Student*.

Namn:_____ Personnummer:_____

6 Komplexitet [totalt 6 poäng]

6.1 Sökning i o-ordnat träd [2 poäng]

Antag att vi har representerat en mängd nyckel/värde-par i ett o-ordnat träd och har en sökfunktion som plockar fram ett värde givet en nyckel. Vilken tidskomplexitet har en sådan funktion? Svara med den asymptotiska tidskomplexiteten exempelvis *O(n) där n är ....*. Glöm inte att ange vad *n* är.

Svar: Sökning i ett o-ordnat träd är *O(n)* där n är antalet element i trädet. Eftersom trädet är o-ordnat så måste vi göra ett arbeta som är proportionellt till storleken på trädet; vi kan inte som i fallet med ett ordnat träd halvera arbetet i varje steg.

6.2 Sökning i lista [2 poäng]

Antag att vi representerar en mängd atomära element som en lista. Vi kan välja på att hålla listan ordnad eller låta den vara o-ordnad. Vad blir skillnaden i tidkomplexitet för den sökfunktion som vi kan använda. Svara med den asymptotiska tidskomplexiteten exempelvis *O(n) där n är ....* för de två fallen. Glöm inte att ange vad *n* är.

Svar: Sökningen är i båda fallen *O(n)* där *n* är längden på listan. Vi kan inte göra binärsökning i en lista eftersom vi inte kan hoppa fram och tillbaks i listan på konstant tid.

Namn:_____Personnummer:_____

6.3 Fokus [totalt 2 poäng]

Antag att vi har ett program där vi i princip gör två saker: vi läser in en mängd värden från en fil och vi gör ett konstant antal sökningar bland dessa värden. Antalet värden i filerna som vi arbetar på kommer att växa och vi är orolig för att exekveringstiden kommer att bli för stor när antalet värden stiger.
Vi har en väldigt enkel implementation där vi läser in värden och representerar dem i en lista som vi sedan gör sökningar i. Ett förslag är att istället lägga in värdena i ett ordnat träd och därmed få en bättre tidkomplexitet för sökningen. Hur skulle det påverka tidkomplexiteten för hela programmet, dvs både inläsning och sökning?

Svar: Vid en första anblick så är tidskomplexiteten för programmen densamma, vi måste i båda fallen gå igenom filen vilket är en $O(n)$ operation. I det första fallet så går vi dock igenom filen och lägger elementen i en lista (inte ordnar dem i en lista), detta är en totalt sett en $O(n)$ operation eftersom vi kan lägga elementen först i listan vilket är en konstant operation. Vi gör sedan ett antal sökningar där varje sökning är $O(n)$ men vi har fortfarande totalt sett en $O(n)$ operation.
I det andra fallet så måste vi dock betala ett pris för att ordna elementen i trädet så själva inläsningen och skapandet av trädet är en $O(n * log(n))$ operation. Vi har n element och kommer för varje ellement betala $O(log(n))$ för att lägg in det på sin plats. Vi gör sedan ett antal sökningar, där varje sökning visserligen bara har en $O(log(n))$ komlexitet, men det förändrar ju inte den totala komplexiteten. I det andra fallet har vi alltså en sämre tidskomplexitet: $O(n * log(n))$ istället för $O(n)$.

Namn:_____Personnummer:_____

Appendix

Följande Erlang-funktioner kan komma till användning.
map(Fun, List): returnerar en lista där funktionen applicerats på varje element i den givna listan.
filter(Fun, List): returnerar en lista av de element i listan för vilka funktionen returnerar *true*.
foldl(Fun, Acc, List) och foldr(Fun, Acc, List): returnerar en värde som fås genom att applicera funktionen på varje element i listan och ett ackumulerat värde där Acc är det initiala ackumulerade värdet. Funktionen foldr börjar med det sista elementet och foldl med det första elementet