KTH Information and
Communication Technology

ID1019
Johan Montelius

Programmering II (ID1019)
2016-06-10 09:00-13:00
7.5 hp

Namn: _____

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 eller engelska.
- 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 40 grundpoängen räknas bara som högst 34 och, att högre poäng inte kompenserar för avsaknad av grundpoäng. Gränserna för betyg är som följer:

- Fx: 22 grundpoäng
- E: 24 grundpoäng

- D: 30 grundpoäng
- C: 34 grundpoäng
- B: 34 grundpoäng och 14 högre poäng
- A: 34 grundpoäng och 20 högre poäng

Gränserna kan komma att justeras nedåt men inte uppåt.

Erhållna poäng

Skriv inte här, detta är för rättningen.

Uppgift	1	2	3	4	5	6	Σ
Max G/H	4/-	10/2	2/6	4/2	4/4	16/10	40/24
G/H							

Totalt antal poäng:

Betyg:

Namn:_____

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:

- `[Y|_] = [1,2,3]` **Svar:** `Y = 1`
- `[X,_ |Y] = [1,2,3]` **Svar:** `Y = [3]`
- `Y = [1|Y]` **Svar:** *misslyckas*
- `[X,Y,Z] = [1|[2,3]]` **Svar:** `Y = 2`
- `Z = 32, X = {foo, Z}, {Y, _} = X` **Svar:** `Y = foo`

1.2 Konstruktor, selektor och sprutlackning [2 poäng]

Antag att vi representerar bilar med en tupel `{car, Id, Brand, Color}`, där `Id` är en unik identifierare, `Brand` en sträng som beskriver bilmärket och `Color` en färg representerad som en atom t.ex. `green`, `black` mm.

Implementerar en funktion `new/3` som tar en identifierare, ett märke och en färg och returnerar en bil.

Svar:

```
new(Id, Brand, Color) -> {car, Id, Brand, Color}.
```

Implementerar en funktion `color/1` som tar en bil och returnerar dess färg.

Svar:

```
color({car, _, _, Color}) -> Color.
```

Implementerar en funktion `paint/2` som tar en bil och en färg och returnerar en likadan bil men med den nya färgen.

Svar:

```
paint({car, Id, Brand, _}, Color) -> {car, Id, Brand, Color}.
```

Namn:_____

2 Rekursiva funktioner

2.1 Summa och längd i ett svep [2 poäng]

Antag att vi har en lista av heltal och vi vill beräkna deras summa och antal men vi vill göra det i en genomgång av listan.

Implementera en funktion `once/1` som tar en lista och returnerar en tupel `{Sum, Length}` med summan av talen och längden på listan.

Du får inte ge en lösning där du först går igenom listan för att beräkna summan och sen går igenom den en gång till för att räkna ut längden.

Svar:

```
once([]) -> {0,0};
once([H|T]) ->
   {S,L} = once(T),
   {S+H,L+1}.
```

2.2 Ackermannfunktionen [2 poäng]

Alla de funktioner som vi har arbetat med under kursen har varit så kallade *primitivt rekursiva funktioner*. Det finns dock rekursiva funktioner som inte är primitiva och växer snabbare än de exponentiella funktioner som vi varnat för. Den mest kända rekursiva funktion som inte är primitivt rekursiv är *Ackermannfunktionen* och den definieras som följer:

$$Ack(m,n)\begin{cases} n+1 & \text{if } m = 0 \\ Ack(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ Ack(m-1, Ack(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Implementera en funktion `ack/2` enligt definitionen på Ackermannfunktionen.

Svar:

```
ack(0,N) -> N+1;
ack(M,0) -> ack(M-1, 1);
ack(M,N) -> ack(M-1, ack(M, N-1)).
```

Ack(4,1) är lika med 65533 och det tar ca 30 sekunder att räkna ut på min laptop. Ack(4,2) är ..... och tar .....

Namn:_____

- Ja, hur lång tid skulle det ta? Ett extra bonuspoäng till de som ens är i närheten av de rätta svaret eller körtiden.

OBS - använd inte resten av tentamenstiden till att försöka räkna ut svaret!

Svar: (Ack(4,2) is in the order of $2^{2^{16}}$, a number that has 20.000 digits if we write it in decimal notation. It would take aprx $4^{2^{16}}$ seconds to calculate. This is close to $2^{2^{15}}$ years or, the age of the universe is 14 billion years, is $2^{2^{10}}$ times the age of the universe..... did I get it right?

2.3 ett aritmetiskt uttryck [2 poäng]

Antag att vi av outgrundlig anledning väljer att representera ett aritmetiskt uttryck som en lista av tal och operatorer. För att inte hamna i tvetydigheter så begränsar vi oss till operationerna '+' och '-'. En representation av uttrycket $5 + 2 - 3 + 10$ representeras alltså med listan:

```
[5, '+', 2, '-', 3, '+', 10]
```

Implementerar en funktion `eval/1` som tar ett aritmetiskt uttryck och returnerar det beräknade värdet. För exemplet ovan skall vi alltså returnera `14`. Vi antar att uttrycket är korrekt.

Svar:

Vi vill inte evaluera $5 + 2 - 3 + 10$ som $5 + (2 - (3 + 10)) = -6$ utan som $((5 + 2) - 3) + 10 = 14$.

```
eval([]) -> 0;
eval([N|T]) -> eval(T, N).

eval([], Sum) -> Sum;
eval(['+', N2 | L], Sum) -> eval(L, Sum + N);
eval(['-', N2 | L], Sum) -> eval(L, Sum - N).
```

2.4 isomorfa träd [2 poäng]

Antag att vi representerar tomma träd med `nil` och noder i trädet med tupel `{tree, Value, Left, Right}` där `Left` och `Right` är träd.

Två träd är *isomorfa* om de båda är tomma eller om de båda är noder där deras högra grenar är isomorfa och deras vänstra grenar är isomorfa. Det har alltså ingen betydelse vilket värde själva noden har utan det är formen på träden som vi är ute efter.

Namn:_____

Implementera en funktion `isomorfic/2` som returnerar `true` eller `false` beroende på om dess två argument är isomorfa eller inte.

Svar:

```
isomorfic(nil, nil) -> true;
isomorfic({tree, _, L1, R1}, {tree, _, L2, R2}) ->
   case isomorfic(L1, L2) of
       true -> isomorfic(R1, R2);
       false -> false
   end;
isomorfic(_, _) -> false.
```

2.5 spegelbild [2 poäng]

Antag att vi representerar träd enligt beskrivingen nedan. En spegelvändning av ett träd är naturligtvis ett träd där varje nods vänstergren har byts ut mot orginalets spegelvända högergren och vice versa.

Implementera en funktion `mirror/1` som tar ett träd och returnerar det spegelvända trädet.

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

Svar:

```
mirror(nil) -> nil;
mirror({tree, Value, Left, Right}) ->
   {tree, Value, mirror(Right), mirror(Left)}.
```

2.6 beräkning av polynom [2 poäng*]

Antag att vi representerar ett polynom som en lista av koefficienter. Polynomet $2x^3 + 5x^2 + 7$ skulle representeras med listan `[2,5,0,7]`; den tredje koefficienten är 0 eftersom vi inte har någon term ax.

Implementera en funktion `calc/2` som tar ett polynom och ett värde på polynomets variabel, och beräknar polynomets värde. Notera att polynomet

Namn:_____

kan vara av godtycklig grad även av grad 0 då det bara består av en lista med ett enda värde, värdet på konstanten. Vi kan för enkelhetens skull anta att det alltid finns minst en koefficient, den tomma listan är inget polynom.

Svar:

```
calc(Poly, X) ->  calcp(Poly, X, 0).

calcp([], _, Sum) ->  Sum;
calcp([K|Poly], X, Sum) ->
    calcp(Poly, X, Sum*X + K).
```

3 Evaluera 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.

3.1 fria variabler [2 poäng]

Vilka är den/de fria variabeln/variablerna i följande sekvens?

```
Z = 42, Y, F = fun(X) -> foo(X, Y) end, F(Z)
```

Svar: `Y`

Vad evaluerar sekvensen nedan till?

```
{X,Y} = {a, b}, F = fun(X) -> {X, Y} end, F(c)
```

Svar: `{c,b}`

Namn:_____

3.2 mönstermatchning [2 poäng*]

De tre första reglerna för möstermatchning är som följer:

- $P\sigma(a, s) \rightarrow \sigma$ if $a \equiv s$
- $P\sigma(\_, s) \rightarrow \sigma$
- $P\sigma(v, s) \rightarrow \sigma$ if $v/s \in \sigma$

Vad skall vi göra med $P\sigma(v, s)$ när $v/t \notin \sigma$? Skriv ner en regel som beskriver hur mönstermatchningen skall evalueras.

Svar: $P\sigma(v, s) \rightarrow \{v/s\} \cup \sigma$ if $v/t \notin \sigma$

3.3 λ-uttryck [4 poäng*]

Antag att vi vill utöka vårt språk med lambda-uttryck. Dessa ges en syntax på formen:

⟨*lambda*⟩ ::= 'fun' '(' ⟨*param*⟩ ')' '->' ⟨*sequence*⟩ 'end'

Uttryckets *parametrar*, `<param>`, är en sekvens av variabler och dess *kropp*, `<sequence>`, en sekvens av uttryck.

Ett lambda-uttryck evalueras till en så kallad *closure* som består av dess *parametrar*, *kropp* och en omgivning θ. Vad består denna omgivning θ av?

Svar: När ett lambda-uttryck evalueras i en omgivning σ så skapas omgivningen θ som består av bindningarna i σ för de fria variabler som finns i lambda-uttryckets kropp.

4 Komplexitet

I de följande frågorna är det viktigt att du beskriver vad till exempel n är och att du motiverar ditt svar.

4.1 spegling av träd [2 poäng]

I 2.5 så spegelvända du ett träd. Vad har den funktionen för asymptotisk tidskomplexitet?

Svar: Tidskomplexiteten är $O(n)$ där n är antalet element i trädet eftersom vi gör ett konstant arbete i varje nod.

Namn:_____

4.2 komplexitet för `foo/2` [2 poäng]

Vad är asymptotiska tidskomplexiteten för funktionen `foo/2`? Motivera dina antaganden och alternativa svar.

```
foo(_, nil) -> false;
foo(K, {node, K, V, L, R}) -> {ok, V};
foo(K, {node, E, _, L, R}) when K < E -> foo(K, L);
foo(K, {node, E, _, L, R}) -> foo(K, R).
```

Svar: Funktionen söker efter en nyckel i ett ordnat träd. Tidskomplexiteten är $O(lg(n))$ där n är antalet element i trädet - om vi antar att trädet är balanserat, I värsta fall är tidskomplexiteten $O(n)$.

4.3 hitta en väg [2 poäng*]

Antag att vi har en riktad acyklisk graf som är representerad men hjälp av tupler `{node, Id, Links}` där `Links` är en lista av noder som en nod har förbindelser till. Vi är aningen naiva och implementerar en sök-funktion som följer:

```
find(To, From) ->
   search(To, [From]).

search(_, []) -> fail;
search(To, [{node, To, _}| _]) -> {ok, [To]};
search(To, [{node, Through, Links}| Alt]) ->
   case search(To, Links) of
       {ok, Path} -> {ok, [Through|Path]};
       fail -> search(To, Alt)
   end.
```

Funktionen kommer inte hitta den kortaste vägen men den kommer hitta en väg om det finns en. Vad är den asymptotiska tidskomplexiteten för funktionen?

Svar: Den asymptotiska tidskomplexiteten är $O(2^n)$. Om n är antalet noder i grafen. Om förgreningsfaktorn är 2 så har funktionen samma komplexitet som Fibonacci - om vi har n noder att utgå från så kommer vi att undersöka två vägar, den ena med $n-1$ noder och den andra med $n-2$ noder. Om förgreningsfaktorn är större så har vi mer jobb att utföra och som mest är det när fögreningsfaktorn är n. Hur skulle den grafen se ut? Vad blir komplexiteten för n lika ned 1, 2, 3, ...

Namn:_____

5 Concurrency

5.1 [2 poäng]

Implementerat en procedur `collect/0` som tar emot en sekvens med meddelanden som avslutas med meddelandet `done`. Proceduren skall returnera alla meddelanden (förutom `done`) som en lista där meddelanden är ordnade i den ordningen de togs emot (första meddelandet som togs emot först i listan).

```
-spec collect() -> [any()].
```

Svar:

```
collect() ->
  receive
     done -> [];
     X - > [X| collect()]
  end.
```

5.2 tic-tac-toe [2 poäng]

Antag att vi har följande definition av procedurerna `first/1`, `second/1` och `last/1`.

```
first(P) ->
  receive
     tic -> P ! tic, second(P)
     tac -> P ! tac, second(P)
  end.

second(P) ->
  receive
    tic -> P ! tic, last(P)
    tac -> P ! tac, last(P);
    toe -> P ! toe, last(P)
  end.

last(P) ->
  receive
     X -> P ! X, P ! done
  end.
```

Antag också att vi har implemnterat proceduren `collect/0` i föregående uppgift. Vad blir resultatet när vi exekverar anropet `test()`?

Namn:_____

```
test() ->
   Self = self()
   P = spawn(fun()-> first(Self) end),
   P ! toe,
   P ! tac,
   P ! tic,
   collect().
```

Svar: `[tac, toe, tic]\`

5.3 en monitor [4 poäng*]

En så kallad *monitor* är en konstruktion som skyddar en kritisk sektion i ett program. Exekverande trådar som vill utföra den kritiska operationen kan bara göra detta genom att använda sig av det API som monitorn erbjuder. Bara en tråd skall ges tillgång till den kritiska sektionen och trådar som har begärt tillgång skall få utföra den kritiska operationen i den ordning de begärde access.

Hur skulle man kunna implementera en *monitor* i Erlang? Ge ett exempel med kod som förklarar hur en monitor skulle kunna implementeras.

Svar: Enklaste sättet är att skydda den kritiska sektorn i en process. De processer som vill utföra den kritiska operationen måste skicka ett medelande till monitorn som kommer att hantera dessa en och en i den ordning de tas emot.

```
new() -> spawn(fun() -> monitor() end.

monitor() ->
   receive
     {do, This} ->
         critical(This),
         monitor()
     end.
```

Namn:_____

6 Programmering

6.1 Huffman-kodning

6.1.1 avkodning [6 poäng]

Antag att vi har ett träd för att avkoda en Huffman-kodad text där den kodade texten representeras som en lista av ettor och nollor. Noder i trädet är representerat med en tupel `{huf, Zero, One}` och löven med en tupel `{char, Char}`. När man skall avkoda en kodad text så går man ner i trädet och väljer gren beroende på om man läser en etta eller nolla. Om man kommer till ett löv så har man hitta den bokstav man sökte efter och kan börja från trädets rot igen.

Implementera en funktion `decode/2` som tar en kodad text och ett avkodningträd och returnerar den avkodade textsträngen.

Svar:

```
decode(Seq, Tree) -> decode(Seq, Tree, Tree).

decode([], _, _) -> [];
decode([0|T], {huf, Zero, _}, Tree) -> decode(T, Zero, Tree);
decode([1|T], {huf, _, One}, Tree) -> decode(T, One, Tree);
decode(Seq, {char, Char}, Tree) -> [Char|decode(Seq, Tree, Tree)].
```

Namn:_____

6.1.2 bygg trädet [2 poäng*]

Antag att vi har en frekvenstabell representerad som en lista av tupler `{freq, {char, Char}, F}` och vill bygga ett Huffmanträd på formen given i föregående uppgift. Tabellen är ordnad med de bokstäver med minst frekvens först. Din uppgift är att implementera en funktion `huffman/1` som tar frekvenstabellen och returnerar ett Huffman-träd. Till din hjälp har du en funktion `insert/2` som tar ett element, `{freq, any(), integer()}`, och en sorterad tabell och returnerar en uppdaterad tabell där elementet har lagts in in på rätt position.

Svar:

```
huffman([{freq, Tree, _}]) -> Tree;
huffman([{freq, A, Fa}, {freq, B, Fb} | Rest]) ->
       huffman(insert({freq, {huf, A, B}, Fa+Fb}, Rest)).
```

Namn:_____

6.2 Mandelbrot

6.2.1 en beräkningsprocess [6 poäng]

Antag att vi har en funktion `mandel/2` som kan beräkna "djupet" på en punkt i det komplexa talplanet. Funktionen tar ett komplext tal och ett maximalt djup och returnerar antingen `0` eller det djup för vilket beräkningen kan avgör att punkten inte tillhör Mandelbrot-mängden. Vi har även en funktion `color/1` som tar ett djup och returnerar en färg.

Implementera en process som när den startas kopplar upp sig mot en koordinator och sedan hjälper till med att beräkna färger för de punkter som koordinatorn skickar. Processen skall startats med proceduren `start/1` som tar processidentifieraren för koordinatorn som argument. Den skall skicka ett meddelande `{request, Pid}` till koordinatorn för att hjälpa till med beräkningen.

Följande två meddelanden skall hanteras:

- `{calc, Pixle, Point, Max, Collector}`: processen skall skicka `{color, Pixle, Color}` till processen `Collector`. Här är `Pixle` en representation av den pixel som beräknas och `Point` dess motsvarande punkt i det komplexa talplanet.
- `done` : då processen skall terminera.

Svar: Lite beroende på hur man tolkar uppgiften:

```
start(Cord) ->
  spawn(fun() -> init(Cord) end).

init(Cord) ->
  client(Cord).

client(Cord) ->
   Cord ! {request, self()}
   receive
      {calc, Pixle, Point, Max, Collector} ->
          Collector ! {color, Pixle, color(mandel(Point, Max))},
          client(Cord);
      done -> ok
   end.
```

Namn:_____

6.2.2 en koordinator [2 poäng*]

Implementera en koordinator som kan beräkna en Mandelbrot-bild med hjälp av en eller flera beräkningsprocesser. Koordinatorn skall när den startas: starta en process som skall samla in alla de beräknade punkterna och en process som är ansvarig för att dela ut uppgifter. När insamlaren har skapat hela bilden skall bilden skickas till en process som anges vid uppstart av koordinatorn.

Antag att vi har insamlaren, `collect/2`, och utdelaren, `server/7`, givna. Dessa har följande egenskaper:

- `collect(Width, Height)` : vänta på färger från beräkningsprocesser och returnera den färdiga bilden.
- `server(Width, Height, X, Y, K, Max, Collector)` : distribuera arbete till beräkningsprocesser för att räkna ut en bild med måtten `Width`, `Height`, vänstra övre hörnet i postion `X`, `Y`, steglängden `K` och maxdjup `Max`. Beräkningsprocesserna skall skicka sina svar till `Collector`.

Hur implementerar vi uppstartsproceduren? Proceduren `start/7` skall ta följande argument:

- `Width` och `Height` : bildens bredd och höjd.
- `X`, `Y` och `K`: positionen på vänstra övre hörnet och steglängd.
- `Max` : maximala beräkningsdjupet.
- `Ctrl` : den process som skall ha den färdiga bilden.

Svar:

```
start(Width, Height, X, Y, K, Depth, Cntr) ->
   spawn(fun() -> init(Width, Height, X, Y, K, Depth) end).

init(Width, Height, X, Y, K, Depth) ->
    Collector = spawn(fun() -> init_collector(Width, Height, Ctrl) end),
    server(Width, Height, X, Y, K, Depth, Collector).

init_collector(Width, Height, Ctrl) ->
   Image = collect(Width, Height),
   Ctrl ! {image, Image}.
```

Namn:_____

6.3 en räknare [totalt 4 + 6 poäng]

I denna uppgift skall vi implementera en server som håller reda på en summa. Den nås via HTTP och adderar tal till en intern räknare så att vi kan fråga hur stor summan är.

Vi börjar med en enkel server som kommer svara på anrop på port 8080. Proceduren `request/1` tar hand om ett anrop och returnerar ett svar till klienten. Idén är att vi skall kunna skicka skicka två olika kommandon till servern `{add, N}`, där `N` är ett heltal, och `sum` som är en begäran om att veta hur mycket som har adderats. Vi använder HTTP för att koda frågorna och svaren men det är inte någonting som vi behöver bry oss om för att lösa uppgifterna.

OBS - koden som följer är inte en lösning.

```
-define(Port, 8080).

start() ->
    spawn(fun() -> init() end).

init() ->
    case gen_tcp:listen(?Port, [list, {active, false}, {reuseaddr, true}]) of
        {ok, Listen} ->
            handler(Listen),
            gen_tcp:close(Listen);
        {error, Error} ->
            io:format("oh no: ~w~n", [Error])
    end.

handler(Listen) ->
    case gen_tcp:accept(Listen) of
        {ok, Client} ->
            request(Client),
            handler(Listen);
        {error, _} ->
            error
     end.
```

Namn:_____

6.3.1 en enkel lösning [4 poäng]

Den givna koden är inte en lösning eftersom den bara returnerar ett eko på frågan som kommer in. Din första uppgift är att skriva om servern så att den håller reda på hur mycket som har adderats och returnerar summan till den som frågar. Du behöver inte skriva om hela servern utan kan ändra i den givna koden.

```
request(Client) ->
    case gen_tcp:recv(Client, 0) of
        {ok, Str} ->
            Response = case parse(Str) of
                           {add, N} ->
                               reply("received " ++ integer_to_list(N));
                           sum ->
                               reply("you want to know the sum");
                           error ->
                               notfound()
                       end,
            gen_tcp:send(Client, Response);
        {error, Error} ->
            error
    end,
    gen_tcp:close(Client).
```

Svar:

```
               :
            handler(Listen, 0),
               :


handler(Listen, N) ->
    case gen_tcp:accept(Listen) of
        {ok, Client} ->
            N1 = request(Client, N),
            handler(Listen, N1);
        {error, _} ->
            error
     end.

request(Client, Sum) ->
    {_, Updated} = case gen_tcp:recv(Client, 0) of
        {ok, Str} ->
```

Namn:_____

```
            {Response, Added} = case parse(Str) of
                                    {add, N} ->
                                        {reply("received " ++ integer_to_list(N)), Sum+N};
                                    sum ->
                                        {reply("the sum is " ++ integer_to_list(Sum)), Sum});
                                    error ->
                                        {notfound(), Sum}
                                end,
            gen_tcp:send(Client, Response),
            {ok, Added};
        {error, Error} ->
            {error, Sum}
    end,
    gen_tcp:close(Client),
    Updated.
```

Namn:_____

6.3.2 bättre prestanda [3 poäng*]

I lösningen till föregående fråga så kan du (om du gjort den enkla lösningen) bara svara på en förfrågan åt gången. Antag att vi vill kunna hantera flera förfrågningar parallellt, hur skulle lösningen då se ut? Skriv inte om hela server men beskriv, ned kod, vilka förändringar som måste göras.

Svar:

Vi startar en process som håller en räknare.

```
            Counter = spawn(fun() -> counter(0) end),
            handler(Listen, Counter),
               :
```

Dess process-id skickas som argument till varje ny fråga.

```
handler(Listen, Counter) ->
    case gen_tcp:accept(Listen) of
        {ok, Client} ->
            spawn(fun() -> request(Client, Counter) end),
            handler(Listen, Counter);
        {error, _} ->
            error
     end.
```

Nu kan vi delegera add-operationer och göra förfrågningar om summan.

```
request(Client, Counter) ->
    case gen_tcp:recv(Client, 0) of
        {ok, Str} ->
            Response = case parse(Str) of
                           {add, N} ->
                               Counter ! {add, N},
                               reply("received " ++ integer_to_list(N));
                           sum ->
                               Counter ! {req, self()},
                               receive
                                 {sum, Sum} ->
                                       reply("the sum is " ++ integer_to_list(Sum))
                               end;
                           error ->
                               notfound()
                       end,
            gen_tcp:send(Client, Response);
        {error, Error} ->
```

Namn:_____

```
            error
    end,
    gen_tcp:close(Client).
```

Processen som håller summan är mycket enkel.

```
counter(Sum) ->
  receive
    {add, N} ->
       counter(Sum+N);
    {req, Pid} ->
       Pid ! {sum, Sum},
       counter(Sum)
  end.
```

Namn:_____

6.3.3 dubblera summan om under 20 [3 poäng*]

Antag att vi har en lösning på föregående uppgift, hur skulle vi lösa följande problem?

Vi vill kunna erbjuda en *atomär operation* som dubblar summan om den är under 20. Man tror kanske att det bara är för en klient att först skicka in en fråga om vad summan är, kontrollera om den är under 20 och i så fall skicka in en begäran om att addera lika mycket. Problemet med denna lösning är att den inte är atomär, om två klienter utför operationen samtidigt så kommer de .... - ja, vad kommer att hända? Vad är problemet och vad menas med atomär?

Svar:

Atomär i det hör sammanhanget betyder att operationerna skall genomföras i sin helhet och i en sekvensiell ordning. Om vi har summan 15 och båda klienterna först gör en förfrågan om summan och sen skickar in en addering på 15 så kommer vi att ha summan 45. Om operationerna var atomära så skulle den ena operationen utföras och ge summan 30 så att den andra dubbleringen aldrig skulle ske.

Hur skulle man med en mycket liten förändring utöka implementationen så att den skulle kunna utföra an villkorad atomär fördubbling. Du kan anta att `parse/1` returnerar vad du vill så det är enbart de interna förändringarna vi är intresserade av.

Svar: Lösningen är mycket enkel, vi behöver bara addera ett nytt medelande till vår räknare.

```
    :
double ->
    Counter ! double_if_less_than_20,
    reply("doubled if less then 20");
    :

    :
double_if_less_than_20 ->
    if
      Sum < 20 ->
         counter(2*Sum);
      true ->
         counter(Sum)
    end;
 :
```