KTH Information and
Communication Technology

ID1019
Johan Montelius

Programmering II (ID1019)
2016-03-19 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.
- 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:

- 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:

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

1.2 strängar med ihopslagning på konstant tid [2 poäng]

Om vi representerar strängar som listor av ASCII-värden så är det enkelt att "läsa" från en sträng eftersom det första elementet kan nås på konstant tid (det ligger ju först i listan). Det är dock rätt jobbigt att slå ihop två strängar eftersom detta måste göras med en append-operation. Vi skulle kunna representera strängar som antingen 1/ en lista av ASCII-värden eller 2/ en struktur som i sig innehåller två strängar. Fördelen skulle vara att man då kunde sätta ihop två strängar på konstant tid. Nackdelen är naturligtvis att det ibland är lite jobbigt att hitta första bokstaven.
Beskriv denna form av representation. Du får själv bestämma vilka strukturer som skall användas; använd med fördel s.k. typ-notation.
Implementera funktionen `concat/2` som tar två strängar och returerar den ihopslagna strängen.

Svar:

```
-type string() ::  [char()] | {string, string(), string()}.

concat(S1, S2) -> {string, S1, S2}.
```

2 Rekursiva funktioner

2.1 ta bort alla sekvenser av upprepningar [2 poäng]

Antag att vi har en lista av element och vill skapa en en lista som består av samma element i samma ordning men där vi har plockat bort alla upprepningar av element som kommer efter varandra. Om vi har listan:

Namn:_____

```
[1,2,2,3,1,2,4,4,4,2,3,3,1]
```

Så skall vi skapa listan:

```
[1,2,3,1,2,4,2,3,1]
```

Notera att vi fortfarande har dubbletter av element, som `2` i exemplet, men att dessa inte kommer direkt efter varandra.
Hur implementeras funktionen `reduce/1` som har ovan givna egenskaper?

Svar:

```
reduce([]) ->
    [];
reduce([A, A |Rest]) ->
    reduce([A|Rest]);
reduce([A|Rest]) ->
    [A|reduce(Rest)].
```

2.2 Caesarchiffer [2 poäng]

Ett Caesarchiffer är kanske den enklaste formen av kryptering och görs genom att man byter ut varje bokstav i ett meddelande mot den bokstav som är tre positioner tidigare i alfabetet. Ordet "hej" skulle alltså kodas "ebg". Antag att vi endast använder bokstäverna 'a' till 'z' och att vi hanterar alfabetet som en ring; dvs 'a' kodas som 'x', 'b' som 'y' och 'c' som 'z'. Mellanslag kodas som mellanslag så meningen "encoded you are" blir "bkzlaba vlr xob". ASCII-värdet för mellanslag är 32 (vi kan även skriva `$ ` men det är lite svårt att se mellanslaget). Värdet för 'a' är 97 och 'z' har värdet 122. Skriv funktionen `encode/1` som tar en sträng och returnerar den kodade versionen.

Svar:

```
encode([]) ->
    [];
encode([32|Rest]) ->
    [32|encode(Rest)];
encode([X|Rest]) when X < 100 ->
    [X-3+26|encode(Rest)];
encode([X|Rest])  ->
    [X-3|encode(Rest)].
```

2.3 En triss i poker [2 poäng]

Antag att vi vill implementerat ett program som spelar poker. Vi har valt att representera en "hand" som en lista av fem oordnade kort.

Namn:_____

Skriv en funktion `triss/1` som kan avgöra om vi har en triss på hand (tre kort av samma valör). Funktionen skall returnera `true` om vi har ett tretal och annars `false`. Du får själv välja hur kort skall representeras.
Funktionen kan returnera `true` om om vi har ett fyrtal eller en så kallad "kåk" (en triss och ett par) eftersom båda dessa händer inehåller en triss.
Till din hjälp får du använda biblioteksfunktionen `lists:filter/2` som tar en funktion och en lista med element och returnerar en lista med de element för vilket funktionen returnerar `true`.
Som exempel kommer anropet:

```
> lists:filter(fun(X) -> X > 3 end, [8,2,6,3]).
```

att returnerar `[8,6]`.

Svar:

```
triss([]) ->
    false;
triss([{card, _, V}|Hand]) ->
    case lists:filter(fun({card, _, N}) -> N == V end, Hand) of
        [_, _ | _] ->
            true;
        _ ->
            triss(Hand)
    end.
```

2.4 *merg sort* bättre än *quick sort* [2 poäng]

Algoritmen *merge sort* bygger på att man först delar upp en lista i två lika stora delar, sorterar delarna och sedan sammanfogar de två sorterade listorna. Algoritmen kan implementeras som följer:

```
msort([]) -> [];
msort(A) ->
   {L1, L2} = split(A),
   merge(msort(L1), msort(L2)).
```

Antag att listan består av heltal. Hur implementerar du funktionen `merge/2`?

Svar:

```
merge([], R) ->
    R;
merge(L, []) ->
    L;
merge([H1|T1]=L, [H2|T2]=R) ->
```

Namn:_____

```
    if
      H1 < H2 -> [H1|merge(T1,R)];
      true -> [H2|merge(L,T2)]
    end.
```

2.5 från hög till lista [2 poäng]

Antag att vi har en så kallad *hög* (på engelska *heap*), som är representerad som ett binärt träd. En hög är antingen tom, som vi representerar med atomen `nil`, eller en nod som består av ett element och två grenar som också är högar, `{heap, Element, Left, Right}`.
En viktig egenskap hos en hög är att det minsta elementet ligger i roten. Det näst minsta elementet ligger antingen i dess högra eller vänstra gren men eftersom båda dessa är högar så vet vi att det ligger överst i någon av dessa.
Implementera en funktion `heap_to_list/1` som tar en hög som argument och returnerar en ordnad lista av alla element. Du får inte använda dig av några biblioteksfunktioner men du kan använda dig av någonting du just skrivit ;-).

Svar:

```
heap_to_list(nil) -> [];
heap_to_list({heap, E, Left, Right}) ->
    L = heap_to_list(Left),
    R = heap_to_list(Right),
    [E | merge(L, R)].
```

2.6 ta bort det minsta [2 poäng*]

Den stora fördelen med en hög (*heap*) är ju att vi alltid hittar det minsta värdet i trädets rot. Det är kanske inte helt enkelt att plocka bort det minsta och ordna om så att man får en ny hög. Lösningen blir dock ganska enkel om man tänker rekursivt.
Skriv en funktion `pop/1`, som tar en hög och returnerar `{ok, Value, Heap}` där `Value` är högens minsta värde och `Heap` en ny hög där värdet har tagits bort. Om man försöker göra pop på en tom hög skall funktionen returnerar `false`.

Svar:

```
pop(nil) -> false;
pop({heap, V, L, nil}) ->
   {ok, V, L};
pop({heap, V, nil, R}) ->
   {ok, V, R};
pop({heap, V, L, R}) ->
```

Namn:_____

```
{heap, VL, _, _} = L,
{heap, VR, _, _} = R,
if
   VL < VR ->
      {ok, VL, Rest} = pop(L),
      {ok, V, {heap, VL, Rest, R}};
   true ->
      {ok, VR, Rest} = pop(R),
      {ok, V, {heap, VR, L, Rest}}
end.
```

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.

3.1 evaluera ett uttryck [2 poäng]

Evaluera följande uttryck, antag att:

$$\sigma = \{X/a, Y/\{a, b\}\}$$

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

3.2 and, or och xor [2 poäng*]

Det vore väl rätt så bra om vi i språket kunde ha de boolska operatorer inbyggda i språket. För att hantera detta detta skall vi först utöka språket och sen även definiera vilka regler som skall gälla vid evaluering.
För enkelhetens skull så skriver vi alla boolska uttryck med parenteser så att vi har associationen helt klar. Operanderna är '&' för and, '|' för or och 'x' för xor. Vi vill naturligtvis även ha de två boolska värdena `true` och `false`. Vi vill kunna skriva sekvenser som:

```
A = true, B = false, ((A & B) | (A x B))
```

Namn:_____

Eftersom vi vill utöka de uttryck som vi kan hantera så ändrar vi beskrivningen av <*expr*> så att det även innefattar boolska uttryck.

$\langle expr \rangle \ ::= \ ... \mid \langle bool \rangle$

Nu behöver vi bara med en BNF-grammatik beskriva hur de boolska uttrycken, <*bool*>, ser ut; hur ser en sådan beskrivning ut?

Svar:

$\langle bool \rangle \ ::= \ \text{true} \mid \text{false} \mid$
$\text{'('} \ \langle expr \rangle \ \text{'\&'} \ \langle expr \rangle \ \text{')'} \mid$
$\text{'('} \ \langle expr \rangle \ \text{'|'} \ \langle expr \rangle \ \text{')'} \mid$
$\text{'('} \ \langle expr \rangle \ \text{'x'} \ \langle expr \rangle \ \text{')'}$

Vi måste även ha en regel som beskriver vad som skall göras när vi skall evaluera ett boolskt uttryck. Hur skall vi skriva en ny regel för evalueringsfunktionen E? I din beskrivning så skall du använda dig av notationen $\wedge$, $\vee$ och $\oplus$ för att beskriva vad som skall göras.

Svar:

- $E\sigma((e_1 \ \& \ e_2) \rightarrow E\sigma(e_1) \wedge E\sigma(e_2)$
- $E\sigma((e_1 \mid e_2) \rightarrow E\sigma(e_1) \vee E\sigma(e_2)$
- $E\sigma((e_1 \ \text{x} \ e_2) \rightarrow E\sigma(e_1) \oplus E\sigma(e_2)$

3.3 lambda [4 poäng*]

Antag att vi har utökat vårt språk till att även kunna hantera lambdauttryck. Vi har en syntax för dessa och evaluerar dem enligt följande regel:

- $E\sigma((\text{fun(vars) -> sequence end}) \rightarrow closure(param, sequence, \theta)$

Här är *param* en sekvens av variabelidentifierare och θ den delmängd av σ som har variabelbindningarna för de fria variablerna i *sequence*.
Frågan är nu vad vi skall göra när vi skall evaluera en tillämpning av vår *closure* på en sekvens av argument.

- $E\sigma(closure(param, sequence, \theta)(\text{args}))- > ...?$

Svar: Vi skall evaluera en sekvens av uttryck som vi har i vår förslutning men det gäller naturligtvis att göra det i rätt omgivning.

$$E\sigma(closure(param, sequence, \theta)(\text{args}))- > E\theta'(sequence)$$

Namn:_____

Detta gäller om *param* är en sekvens av i parametrar $p_1, ..p_i$, args är en sekvens av uttryck $a_1, ...a_i$ och

$$E\sigma(a_i) -> s_i$$

Omgivningen θ' är unionen av bindningar p_i/s_i och θ,

Svar:

4 Complexity

I svaren till nedanstående frågor så var noga med att ange vad till exempel n är och motivera ditt svar.

4.1 traversera ett träd [2 poäng]

Antag att vi har ett binärt träd där en nod är representerat som antingen `{leaf, E}` eller `{tree, Left, Right}`. Antag att trädet är balanserat - vad är den asymptotiska tidskomplexiteten för att traversera trädet med följande funktion:

```
traverse({leaf, E}) -> [E];
traverse({node, Left, Right}) ->
    traverse(Left) ++ traverse(Right).
```

Svar: Antag att vi har ett träd med n löv. I varje rekursion delar vi trädet i två delar vilket gör att vi har ett rekursionsdjup på $\lg(n)$. I första rekursionen skall vi göra append på de två genererade listorna vilket är en operation som tar $n/2$ steg. I nivån under har vi två append-operationer vilka är $n/4$ var dvs sammantaget $n/2$ operationer. Varje nivå i rekursionen halveras arbetet för operationen men vi dubblerar antalet append-operationer. Varje nivå i trädet genererar därför ett arbete som är $O(n)$. Resultatet är:

$$O(n \lg(n))$$

4.2 kanske lite bättre ... eller [2 poäng]

Är det någon skillnad om vi istället skriver vår funktion så här:

```
traverse(Tree) -> traverse(Tree, []).
traverse({leaf, E}, Sofar) -> [E|Sofar];
traverse({node, Left, Right}, Sofar) ->
    traverse(Left, traverse(Right, Sofar)).
```

Namn:_____

Svar:
Vi kommer att besöka varje nod i trädet en gång på samma sätt som i föregående uppgift. Skillnaden är att vi nu har ett konstant arbete i varje nod. I noderna noderna gör vi en mönstermatchning och två funktionsanrop och i löven gör vi en cons-operation. Om n är antalet noder i trädet så är tidskomplexiteten:

$$O(n)$$

4.3 traversera en hög [2 poäng*]

I uppgift 2.5 så gjorde vi om en hög till en lista. Vad är den asymptotiska tidskomplexiteten för denna operation? Antag att högen är balanserad så vi är inte intresserade av den extrema situation där vi har alla element i ena grenen.

Svar:
Rekursionsdjupet är $O(\lg(n))$ där n är antalet element i högen eftersom vi i varje nivå kommer att dela högen i två. I varje rekursion måste vi dock göra en en merge-operation som är $O(l)$ där l är de element i de delhögar som skall slås ihop. Storleken på delhögarna halveras visserligen men eftersom vi har två rekursiva anrop på varje nivå så förblir arbetet per nivå $O(n)$. Summa summarum har vi:

$$O(n \lg(n))$$

5 Concurrency

5.1 atomic swap [2 poäng]

Implementera en procedur `new/1` som skapar en process som skall fungera som en minnescell. Argumentet till proceduren är cellens initiala värde. Processen skall hanter två meddelande: `{swap, New, From}` och `{set, New}`. I det första fallet skall den process som begärt uppdateringen, `From`, få ett meddelande tillbaks `{ok, Old}`, där `Old` är det värde som cellen hade innan uppdateringen. I det andra fallet skickas det inte något svar i retur.

Svar:

```
new(Value) ->
   spawn(fun() -> cell(Value) end).

cell(Old) ->
  receive
```

Namn:_____

```
    {swap, New, From} ->
       From ! {ok, Old},
       cell(New);
    {set, New} ->
       cell(New)
  end.
```

5.2 spin-lock [2 poäng]

En inte helt effektivt sätt att implementera ett lås är ett så kallat *spin-lock*. Idén är att prova att ta ta låset och fortsätta att prova tills man får låset. Antag att vi har implementerat proceduren `new/1` i föregående uppgift och vi vill använda en cell för att implementera ett spin-lock.
Implementera tre procedurer: `create/0`, `lock/1` och `release/1`. Proceduren `create/0` skall returnera ett lås som `lock/1` och `release/1` kan använda. Procedurerna `lock/1` och `release/1` skall returner `ok`.

Svar:

```
create() ->  new(open).

lock(Cell) ->
   Cell ! {swap, taken, self()},
   receive
     {ok, open} ->
         ok;
     {ok, taken} ->
         lock(Cell)
   end.

release(Cell) ->
   Cell ! {set, open}, ok.
```

5.3 en semafor [4 poäng*]

En spin-lock i all ära men det är mer behändigt att ha ett lås i form av en så kallad *semafor*. En semafor är en konstruktion där man kan anmäla intresse för att få ett lås och man får det när det är ledigt. Om låset för närvarande inte är ledigt så får man vänta men man behöver inte göra någonting själv eller ens vara medveten om att man väntar. En semafor är också mer generell i det att den kan tillåta flera processer att ta låset samtidigt men den har ett maximalt antal som den släpper in. Om antalet är 1 så kallas semaforen för binär.
Hur skulle vi implementera en semafor i Erlang? Vi kan implementera den som en process som kan ta två meddelanden: `{request, From}` och `release`.

Namn:_____

En process som skickar ett request-meddelande kommer om/när det finns lås kvar, få ett meddelande `granted` tillbaks. När processen är färdig i den kritiska sektorn skall den skicka ett release-meddelande till semaforen som då kan ge resursen till nästa process om det finns någon som väntar.
Implementerar proceduren `new/1` som skapar en semafor som har den givna funktionaliteten. Argumentet till proceduren är de antalet resurser som semaforen har.

Svar:

```
new(N) -> spawn(fun() -> semphore(N) end).

semphore(0) ->
  receive
    release ->
       semphore(1)
  end;
semphore(N) when N > 0 ->
  receive
    {request, From} ->
       From ! granted,
       semphore(N-1);
    release ->
       semphore(N+1)
  end.
```

6 Programmering

6.1 en liten kalkylator [totalt 6 + 2 poäng]

Du skall implementera en liten kalkylator som skall kunna hantera aritmetiska uttryck med addition och subtraktion över heltal.

6.1.1 representera och evaluerar uttryck [6 poäng]

Du skall först beskriva hur aritmetiska uttryck skall representeras och sedan implementera en funktion `eval/1` som tar ett uttryck och returnerar resultatet av det evaluerade uttrycket.

Svar:
Antag att heltal representeras som `{int, I}`, där `I` är ett heltal och att de aritmetiska operationeran representeras som `{add, A, B}` och `{sub, A, B}`.
Vi skulle kunna beskriva detta genom typdeklarationerna:

```
-type expr() :: {int, int()}  | {add, expr(), expr()} | {sub, expr(), expr()}.
```

Namn:_____

Funktionen `eval/1` kan nu definieras som följer:

```
eval({int, I}) -> I;
eval({add, A, B}) -> eval(A) + eval(B);
eval({sub, A, B}) -> eval(A) - eval(B).
```

6.1.2 lägg till en omgivning [2 poäng*]

Antag att vi även vill kunna hantera variabler i våra aritmetiska uttryck. Vi måste utöka vår representation till att även kunna beskriva variabler och sen implementera en evalueringsfunktion `eval/2` som tar ett uttryck och en omgivning och returnerar ett svar.
För enkelhetens skull så antar vi att vi redan har implementerat en modul, `env`, som exporterar en funktion `lookup/2` som tar en identifierare och en omgivning och returnerar antingen `{Id, Val}` om det finns en bindning för variabeln, eller `false` om det inte finns någon bindning.
Funktionen `eval/2` skall returnera antingen `{ok, Res}` om evalueringen lyckas, eller `error` om vi till exempel försöker evaluera ett uttryck med en variabel som inte har en bindning. Implementera funktionen `eval/2`.

Svar:
Vi utökar representation av uttryck genom att lägga till en representation av variabler.

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

Funktionen `eval/2` kan nu definieras som följer:

```
eval({int, I}, _) -> {ok, I};
eval({var, Id}, Env) ->
    case env:lookup(Id, Env) of
       {Id, Val} -> {ok, Val};
       false -> error
    end;
eval({add, A, B}, Env) ->
    case eval(A, Env) of
       {ok, Aval} ->
           case eval(B, Env) of
             {ok, Bval} ->
                 {ok, Aval + Bval};
             error ->
                 error;
       error ->
           error
    end;
```

Namn:_____

```
  :
  same for {sub, A, B}
```

6.2 en barberare [totalt 6 + 2 poäng]

Ett klassiskt exempel på ett synkroniseringsproblem är att hantera kunder som kommer in till en barberare. Antagandet är att vi har en barberare som naturligtvis bara kan raka en kund åt gången. Vi har dock tre väntstolar i salongen så kunder som kommer in kan slå sig ner och vänta om det finns plats. En kund som kommer in i salongen då alla stolar är upptagna kommer vända i dörren och ta en promenad innan han gör ett nytt försök. Om man vänder i dörren så kan man inte räkna med att bli rakad men om man får sitta ner och vänta så skall man garanterat bli rakad och ingen skall kunna smita före i kön.

6.2.1 hur implementerar ett väntrum [6 poäng]

När vi implementerar detta i Erlang så är det naturligt att dela upp problemet i två olika processer: en som hanterar väntrummet och en som agerar barberare. Barberaren kan fråga väntrummet efter nästa kund och får då en *pid* på näste man till rakning. Kunden ombeds ta plats i frisörsstolen och efter ett tag får denne ett meddelande att rakningen är klar.
Själva proceduren som implementerar barberare skulle kunna skrivas som följer:

```
barber(Waiting) ->
   Waiting ! {next, self()},
   receive
     {ok, Customer} ->
          Customer ! have_a_seat,
          cut_cut_cut_shave_hot_towel_talk(),
          Customer ! thatz_it,
          barber(Waiting)
   end.
```

Väntrummet svara på välkomsthälsningar från kunder som kommer in i salongen, och ber dem antingen vänta eller meddelar att det är fullt. Hur kan den rekursiva procedur som beskriver väntrummet implementeras?

Svar:

```
waiting([]) ->
   receive
     {hello, Customer} ->
        Customer ! please_wait,
```

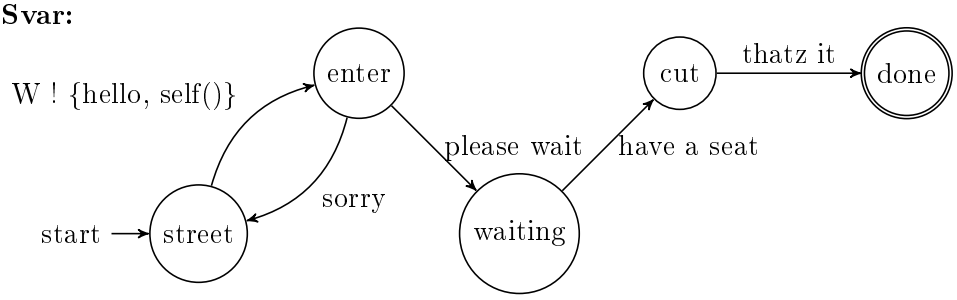
Namn:_____

```
        waiting([Customer])
   end;
waiting([Next|Rest]=Waiting) ->
   receive
     {hello, Customer} ->
        if length(Waiting) >= 3 -> Customer ! sorry;
           true -> waiting(Waiting++[Customer])
        end;
     {next, Barber} ->
        Barber ! {next, Next},
        waiting(Rest)
   end.
```

6.2.2 en kund som ett tillståndsdiagram [2 poäng*]

En kund ges när den skapas tillgång till *pid:en* till ett väntrum. Kunden skall meddela sin ankomst när han kliver innanför dörren. Om han ombeds sitta ner, väntar han på sin tur och blir sen om allt fungerar rakad. När rakningen är klar så är han fri att göra andra saker, men vi kommer låta kunden terminera. Om alla väntstolar är upptagna skall kunden ta en tur runt kvarteret och sedan återkomma till salongen.
Hur skulle vi kunna beskriva en kund som ett tillståndsdiagram. Visa vilka tillstånd som kunden befinner sig i och vilka meddelanden som får den att byta tillstånd. Beskriv även vilka meddelanden den skickar när den kommer till ett nytt tillstånd.

Svar:



6.3 en LDAP server [totalt 4 + 6 poäng]

LDAP, Lightweight Directory Access Protocol, är ett protokoll över vilket man kan göra förfrågningar mot en katalogtjänst. Protkollet är definerat över TCP/IP och har ett otal olika meddelanden för att göra förfrågningar, addera, modifiera eller ta bort information i en katalog. LDAP används till exempel av email-klienter när de skall hämta adresser från ett delat adressregister.

Namn:_____

6.3.1 en enkel lösning [4 poäng]

Om vi antar att vi har en modul, `ldap`, som kan hantera en förfrågan från en klient så skulle en LDAP-server kunna ha följande struktur:

```
-define(Port, 67).

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

init(Port) ->
    case gen_tcp:listen(Port, [binary, {active, true}]) of
        {ok, Listen} ->
            handler(Listen),
            gen_tcp:close(Listen);
        {error, Error} ->
            io:format("ldap server: initialization failed: ~w~n", [Error]),
            error
    end.

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

Denna struktur har dock en begränsning i hur snabbt den kan hantera förfrågningar. Beskriv den begränsning som strukturen har och föreslå en enkel förändring som skulle göra att att vi skulle kunna hantera flera förfrågningar per tidsenhet.

Svar: Strukturen kommer att hantera förfrågningar sekventiellt. Eftersom varje förfrågan troligtvis innebär uppslagningar i en databas så kommer det finns mycket tid där servern väntar. För att utnyttja tiden bättre så kan man hantera flera förfrågningar parallellt. Detta kan åstadkommas genom att starta en process för varje inkommande förfrågan.

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

Namn:_____

```
        {error, Error} ->
            error
    end.
```

6.3.2 två flugor i en smäll [3 poäng*]

Vi antar att du har löst föregående fråga på ett tillfredställande sätt. Det skulle betyda att servern skulle kunna hantera betydligt fler förfrågningar per tidsenhet men det har även ytterligare en fördel. Lösningen ger oss säkert en annan stor fördel som har att göra med att modulen `ldap` är rätt komplex och kanske inte helt tillförlitlig. Vad är problemet och varför är den lösning som du ger i föregående fråga även till viss del en lösning på detta problem?

Svar:
I lösningen ovan så kommer servern att överleva även om en enskild förfrågan crashar. När vi hanterar varje förfrågan i en egen process så är de skyddade från varandra. Den klient som skickar en förfrågan om genererar ett fel kommer visserligen aldrig att få ett svar men övriga klienter kan arbeta på som vanligt.

6.3.3 ett mindre problem, kanske [3 poäng*]

Den lösning som du föreslagit i föregående fråga är nog mycket bra i de flesta fall. Den kanske dock inte har ideal egenskaper om vår server får ett myket stort antal förfrågningar under en mycket kort tid. Var blir problemet och hur skulle man kunna kontrollera situationen?
Du behöver inte implementera en lösning men beskriva hur en lösning skulle kunna se ut och om lösningen har några nackdelar i sig.

Svar: Om vi har en lösning där vi hela tiden skapar nya processer för att ta hand om förfrågningar så kan vi få ett problem med att för många förfrågningar skall hanteras sammtidigt. Servern kan bli överbelastad vilket skulle leda till att svarstiderna försämrade.
För att hantera detta skulle vi behöva ha en gräns på hur många förfrågningar som kan hanteras på samma gång. Man skulle kunn aimplemnetera det med hjälp av en enkel räknare men det skulle kräva att en process som avslutat hanteringen av en förfrågan rapporterade tillbaks.
Räknaren skulle även vara tvungen att hantera de fall där hanterare crashar och därmed aldrig rapporterar tillbaks. Vi skulle med andra ord få en mer komplex lösning för att skydda oss mot någonting som kanske aldrig är ett problem.