KTH Information and
Communication Technology

ID1019
Johan Montelius

Programmering II (ID1019)
2017-03-13 08:00-12:00

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:

De som skriver 4.5hp versionen så skall bara svara på frågorna 1-5. Gränsen för E, D och C är då 14, 16 och 20 poäng. Gränserna för B och A är 8 respektive 12 poäng.

- 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 = 3`
- `[X,Y,Z] = [1|[2,3]]` **Svar:** `Y = 2`
- `[Y|_] = []` **Svar:** *misslyckas*
- `Z = 42, X = {Z, foo}, {Y, _} = X` **Svar:** `Y = 42`
- `X = 32, Z = [52], Y = [X|Y]` **Svar:** *misslyckas*

1.2 Ett telefonnummer [2 poäng]

Vi kan representera telefonnummer med nummer, strängar eller listor av enskilda nummer. För vart och ett av dessa alternativ, vad är fördelen?

- `737652065` **Svar:** enkel att jämföra, fungerar som nyckel
- `"737652063"` **Svar:** enkel att skriva ut, kan använda nummer med bokstäver "+46.."
- `[7,3,7,6,5,2,0,6,5]` **Svar:** bra om vi behöver veta de individuella talen

Svar: Skillanden mellan de två sista är inte så stor båda är listor av tal.

2 Funktionell programmering

2.1 räkna upp en frekvens [2 poäng]

Skriv en funktion `freq(Key, Freq)` som tar en nyckel (en atom) och en lista med frekvenser representerade med tupler `{Key, F}` av nycklar och

Namn:_____

dess frekvens. Funktionen skall returnera en uppdaterad lista där frekvensen för den givna nyckeln har räknats upp med ett.

Svar:

```
freq(Key, []) -> [{Key, 1}];
freq(Key, [{Key, F}|Rest]) -> [{Key, F+1}|Rest];
freq(Key, [{K, F}|Rest]) -> [{K, F}| freq(Key, Rest)];
```

2.2 skapa en frekvenstabell [2 poäng]

Använd funktionen `freq/2` och implementera en funktion `fraq(Keys)` som tar en lista av nycklar och returnerar en lista av frekvenser på formen ovan. Du får bara gå igenom listan en gång och får inte använda några inbyggda eller biblioteksfunktioner.

Svar:

```
fraq([]) -> [];
fraq([K|Rest]) ->
   freq(Key, fraq(Rest)).
```

Svar: Vi kan skriva den svansrekursiv med varför.

2.3 en effektiv kö [2 poäng]

Antag att vi vill implementera en kö i Erlang. Kön skall vara sådan att vi, i de flesta fall kan ta det första elementet i kön på konstant tid och alltid lägga till ett element i slutet av kön i konstant tid. Det trick vi skall använda oss av är att representera kön med hjälp av en tupel med två listor `{queue, First, Last}`. En kö med elementen 1 till 6 skulle kunna se ut som följer:

```
{queue, [1,2], [6,5,4,3]}
```

Vi har de första elementen (hur många kommer att visa sig) i den första listan och resten - i omvänd ordning - i den andra lista. Om i nu plockar bort det första elementet i kön så får vi kön:

```
{queue, [2], [6,5,4,3]}
```

Om vi sedan lägger till ett element, 7, så får vi följande kö:

```
{queue, [2], [7,6,5,4,3]}
```

Problemet uppstår naturligtvis då vi försöker plocka ett element från kön när första listan är tom. Då tar vi den andra listan, vänder på den och lägger

Namn:_____

den som första lista. Detta gör att det ibland tar längre tid att plocka det första elementet i kön.

Implementerar funktionerna `new()`, `enqueue(Elem, Queue)` och `dequeue(Queue)`. Funktionen `new/0` skall returnera en tom kö, `enqueue/2` returnerar en uppdaterad kö och `dequeu/1` returnerar antingen `{ok, Elem, Updated}`, där `Updated` är kön när vi tagit bort ett element `Elem`, eller `fail` om kön är helt tom. Du får använda dig av en funktion `reverse/1` som vänder på en lista.

Svar:

```
new() -> {queue, [], []}.

enqueue(Elem, {queue, First, Last}) -> {queue, First, [Elem|Last]};

deque({queue, [], []}) -> fail;
deque({queue, [Elem|Rest], Last}) -> {ok, Elem, {queue, Rest, Last}};
deque({queue, [], Last}) ->
    [Elem|Rest] = reverse(Last),
    {ok, Elem, {queue, Rest, []}}.
```

2.4 slå ihop två köer [2 poäng*]

Lite knepigare är att skriva en funktion som slår ihop två köer. Alla element i den första kön skall naturligtvis komma före alla element i den andra kön och alla element i köerna skall naturligtvis behålla sin inbördes ordning. Skriv funktionen `app_queue/2` som tar två köer på formen ovan och returnerar en ihopslagen kö. Du får använda dig av en funktion `append/2` som slår ihop två listor och `reverse/1` som vänder på en lista.

Svar:

ett alternativ

```
app_queue({queue, F1, L1}, {queue, F2, L2}) ->
   {queue, append(F1, reverse(L1)), append(L2, reverse(F2))}.
```

2.5 en bättre sträng [2 poäng]

Eftersom Erlang representerar strängar som listor av tecken så blir det lite kostsamt att slå ihop två strängar. Vi skulle kunna göra det enklare för oss genom att representera en sträng som en tuple på följande sätt:

```
-type str() :: {str, list(char())} | {str, str(), str()}.
```

Namn:_____

En `str()` är alltså antingen en tuple som har en vanlig sträng som sitt andra element eller, en tuple som innehåller två `str()`. Vi kan nu implementera vår egen `str_append/2` som tar två `str()` och returnerar en ihopslagen `str()` på konstant tid; definiera funktionen.

Svar:

```
str_append(A, B) -> {str, A, B}.
```

Namn:_____

2.6 räta ut mig [2 poäng]

Om vi nu har vår egen representation av strängar, `str()`, så kan vi behöva en funktion som tar en `str()` och returnerar en vanlig sträng dvs en lista av tecken. Skriv funktionen `str_to_list/1`, du får använda dig av en funktion `append/2` som slår ihop två listor.

Svar:

```
str_to_list({str, L}) -> L;
str_to_list({str, S1, S2}) ->
   append(str_to_lst(S1), str_to_lst(S2)).
```

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 mönstermatchning [2 poäng]

Utför mönstermatchningen nedan, givet att: $\sigma = \{X/a, Y/\{a, b\}\}$.

- $P\sigma(\{Z, b\}, Y) \rightarrow$ **Svar:** $\{Z/a\} \cup \sigma$
- $P\sigma(Z, \{a, X\}) \rightarrow$ **Svar:** $\{Z/\{a, a\}\} \cup \sigma$
- $P\sigma(X, Y) \rightarrow$ **Svar:** fail

3.2 plus och minus [2 poäng*]

Det vore väl rätt så bra om vi i språket kunde ha aritmetiska operationer så som addition och subtraktion av heltal. För att hantera detta detta skall vi först utöka språket och sen även definiera vilka regler som skall gälla vid evaluering.

Namn:_____

För enkelhetens skull så skriver vi alla aritmetiska uttryck med parenteser så att vi har associationen helt klar. Vi vill kunna skriva uttryck som:

```
((10 - Y) + (8 + 3))
```

Antag att argumenten till ett aritmetiskt uttryck antingen är: ett aritmetiska uttryck, ett heltal eller en variabel. Hur kan detta beskrivas med hjälp av ett BNF-uttryck? Antag att vi har definierat uttrycken "<integer>" som beskriver heltalen och "<var>" som beskriver variabler.

Svar:

$\langle arithm \rangle$ ::= $\langle integer \rangle$ | $\langle var \rangle$ | '(' $\langle arithm \rangle$ '+' $\langle arithm \rangle$ ')' | '(' $\langle arithm \rangle$ '-' $\langle arithm \rangle$ ')'

$\langle expression \rangle$ ::= ... | $\langle arithm \rangle$

Vi måste även ha regler som beskriver vad som skall göras när vi skall evaluera ett aritmetiskt uttryck. Hur skall vi skriva reglerna för evalueringsfunktionen E?

Svar:

- $E\sigma((e_1 + e_2) \rightarrow E\sigma(e_1) + E\sigma(e_2)$
- $E\sigma((e_1 - e_2) \rightarrow E\sigma(e_1) - E\sigma(e_2)$

3.3 destruktiv tilldelning [4 poäng*]

Det språk som vi arbetat med är ett funktionellt språk där vi inte kan tilldela variabler nya värden. Antag nu som övning att vi vill kunna ge variabler nya värden, och tillåta en konstruktion som `X := 4`, dvs att vi ger variabeln `X` värdet 4 oavsett vad det har för värde innan. Vilka tillägg i språket skulle vi behöva och hur skulle vi kunna beskriva evalueringen av en sekvens där vi nu istället för ett mönstermatchningsuttryck kan se ett tilldelningsuttryck?

Svar:

Vi behöver en syntax för tilldelning och vi tillåter den enbart i en sekvens.

$\langle assignment \rangle$ ::= $\langle var \rangle$ ':=' $\langle expression \rangle$

$\langle sequence \rangle$ ::= ... | $\langle assignment \rangle$ ',' $\langle sequence \rangle$

Sen får vi modifiera vår evaluering av sekvenser för att ta hand om tilldelningarna. Vi kan göra detta genom att beskriva att den omgivning vi fortsätter med, θ, är en om omgivning där vi tagit bort eventuella bindningar för variabeln, $\{v/u\}$ och sen lagt till den nya bindningen $\{v/t\}$.

Namn:_____

$$\frac{E\sigma(e) \to t \qquad \theta = \{v/t\} \cup (\sigma \setminus \{v/u\}) \qquad E\theta(\text{sequence}) \to s}{E\sigma(v := e, \text{sequence}) \to s}$$

4 Komplexitet

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

4.1 en vanlig kö [2 poäng]

Antag att vi implementerar en kö som en lista där första elementet i kön är först i listan. Vi kan då implementera `enqueue/2` som föjer:

```
enqueue(Elem, []) -> [Elem];
enqueue(Elem, [H|T]) -> [H|enqueue(Elem, T)].
```

Vilken asymptotisk tidskomplexitet har funktionen?

Svar: Funktionen har den asymptotiska tidskomplexiteten $O(n)$ där n är längden på kön. Detta eftersom vi måste gå igenom hela kön när vi skall lägga till elementet sist.

4.2 lite bättre [2 poäng]

Funktionen i föregående uppgift är inte svansrekursiv men det kan vi fixa med en ackumulerande parameter och funktionen `reverse/1`.

```
enqueue(Elem, Queue) -> enqueue(Elem, Queue, []).

enqueue(Elem, [], Sofar) -> reverse([Elem|Sofar]).
enqueue(Elem, [H|T], Sofar) -> enqueue(Elem, T, [H|Sofar]).
```

Givet att `reverse/1` har tidskomplexitet $O(n)$ där n är längden på listan, vad har då `enqueue/2` för tidskomplexitet?

Svar: Ingen skillnad, funktionen är fortfarande $O(n)$. Notera att vi nu går igenom listan två gånger men komplexitetten är den samma.

Namn:_____

4.3 flatten hit eller dit [2 poäng*]

Antag att vi har en lista av listor som i sin tur kan bestå av listor och vi vill skapa en enkel lista av alla element. Vi vill alltså ha följande beteende:

```
> flatten([[1,2], [[3], [4,5,6]]])
[1,2,3,4,5,6]
```

Detta kan vi enkelt göra genom att anropa `flatten/1` rekursivt och sen använda `append/2` för att slå ihop resultaten.

```
flatten([]) -> [];
flatten([H|T]) ->
   append(flatten(H), flatten(T));
flatten(E) -> [E].
```

Detta är dock inte svansrekursivt så vi kanske vill implementera det som följer:

```
flatten(L) -> flatten(L, []).

flatten([], Done) -> Done;
flatten([H|T], Sofar) ->
   flatten(T, append(Sofar, flatten(H, [])));
flatten(E, Sofar) -> [E|Sofar].
```

Lite krångligare men nu är det ju svansrekursivt. Vad är nackdelen med att göra på detta sätt?

Svar: Den första lösningen har komplexitet $O(n)$, om vi inte har en patologisk lista. Om vi har n element som är regelbundet utspridda bland l listor i den yttre listan så kommer varje inre lista ha ca n/l element. Vi kommer då göra l anrop till append och ha arbete som är $O(n/l)$ i varje. Det ger totalt $O(n)$. Om vi däremot har en patologisk lista av listor där första elementet innehåller $n-1$ element så kommer append göra $n-1$ operationer. Om den listan i sig är patologisk så kommer append i den att göra $n-2$ operationer osv. Då får vi faktiskt $O(n^2)$ i komplexitet.

Den andra lösningen har komplexitet $O(n^2)$ eftersom vi där även i det generella fallet kommer att bygga upp `Sofar` till att innehålla fler och fler element. Varje gång vi gör ett rekursionssteg så blir `Sofar` längre och dess längd är generellt $O(n)$. Vi gör $O(n)$ antal append-operationer så den totala komplexiteten blir $O(n^2)$.

Namn:_____

5 Concurrency

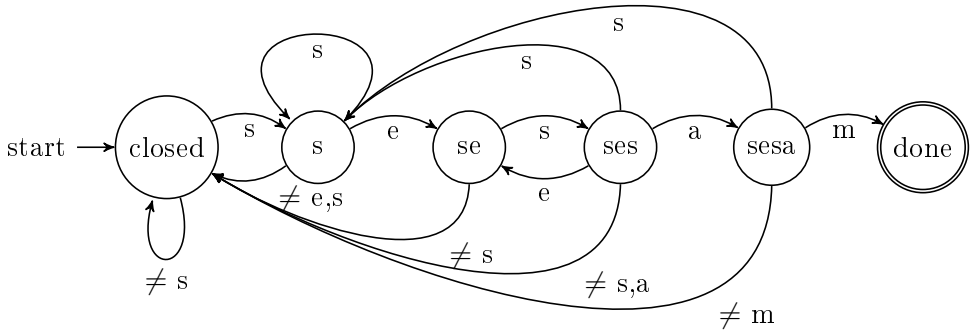
5.1 Sesam öppna dig [2 poäng]

Implementera en process som fungerar som ett kombinationslås. Processen skall börja i ett låst läge och skall sen ta emot meddelanden, bokstäver, ett åt gången. Om den får en sekvensen "s e s a m" så skall den dö (vilket gör den till en rätt meningslös process men det här är en tenta).

Vi skall kunna skicka den vilken sekvens av meddelanden som helst men endast då vi skickar den hemliga sekvensen skall vi nå det öppna läget (där den nu dör). Nedan finns början på ett tillståndsdiagram för processen, men det saknas en hel del övergångar. Komplettera diagrammet och implementera sedan processen. Implementera även en procedur `sesam/0` som startar processen.

Notera att processen skall komma till det öppna läget efter en sekvens "f o o s e s e s a m".

Svar:



Svar:

```
semsam() -> spawn(fun() -> closed() end).
```

Namn:_____

```
closed() ->
  receive
    s -> s();
    _ -> closed();
  end.

s() ->
  receive
    e -> se();
    s -> s();
    _ -> closed();
  end.

se() ->
  receive
    s -> ses();
    _ -> closed();
  end.
```

```
ses() ->
  receive
    a -> sesa();
    e -> se();
    s -> s();
    _ -> closed();
  end.

sesa() ->
  receive
    m -> open();
    s -> s();
    _ -> closed();
  end.

open() -> done.
```

5.2 Baba [2 poäng]

Antag att vårt kassaskåp som skyddas av mekanismen i föregående uppgift skall gå in i ett *öppet* tillstånd när man har skickat rätt sekvens. I det öppna tillståndet skall man kunna skicka ett meddelande och få ett svar tillbaks. Man skall också kunna skicka ett meddelande som får processen att gå till sitt stängda läge. Hur skulle det öppna tillståndet implementeras, hur kan det meddelandet se ut som vi skall skicka och hur kan svaret leveras?

Svar: Ett förslag:

```
open() ->
  receive
     {ali, From} ->
        From ! baba,
        open();
      close ->
       closed()
      _ ->
       open()
   end
```

Namn:_____

5.3 parallel matrismultiplikation [4 poäng*]

Antag att vi har implementerat matrismultiplikation med hjälp av funktionen `mult/2` nedan. Hur skulle vi kunna skriva om funktionen så att den gjorde multiplikationen parallellt? Du får använda inbyggda funktioner och biblioteksfunktioner som till exempel `lists:map/2`, `make_ref/0` mm.

```
mult(A,B) ->
  rows(A, transpose(B)).

rows([], _) -> [];
rows([Row|Rows], Cols) ->
  [cols(Row, Cols) | rows(Rows, Cols)].

cols(R, Cols) -> lists:map(fun(C) -> dot(R,C) end, Cols).

dot([], []) -> 0;
dot([A|R], [B|C]) -> A*B + dot(R,C).

transpose([]) -> [];
transpose([[]|_]) -> [];
transpose(M) ->
     [[H || [H |_] <- M ] |  transpose([ T || [_| T] <- M ])].
```

Svar:

```
rows(Rows, Cols) ->
    Self = self(),
    Calc = fun(R) ->
              Ref = make_ref(),
              spawn(fun() -> C = cols(R, Cols), Self ! {Ref, C}, Ref end),
              Ref
             end,
    Refs = lists:map(Calc, Rows),
    Collect = fun(Ref) ->
                receive
                   {Ref, C} -> C
                end
               end,
    lists:map(Collect, Refs).
```

Namn:_____

6 Programmering

6.1 Huffman

6.1.1 Något bättre än en lista [6 poäng]

Om vi implementerar en avkodare för en huffmankodad sekvens så kan vi naturligtvis ha en lista med en mappning från sekvenser till tecken. Detta är dock inte särskilt effektivt och det finns ett betydligt bättre sätt att representera mappningen. Hur kan vi representera kodtabellen så att vår avkodning blir betydligt effektivare. Förklar i ord och rita en bild, du behöver inte skriva någon kod.

Svar: Man representerar tabellen som ett träd där löven är tecken och varje väg från roten till ett löv är en unik kod.

6.1.2 Är det någon skillnad? [2 poäng*]

Antag att vi skall implementera ett program som avkodar en Huffmankodad fil. Själva avkodaren för inget problem men vi har två alternativ att implementera filhanteringen. Den ena lösningen ser ut som följer:

```
one(Encoded, Result, Table) ->
    {ok, Seq} = file:read_file(Encoded),
    Decoded = decode(Seq, Table),
    {ok, Out} = file:open(Result, [write]),
    file:write(Out, Decoded).

decode([], _Table) ->
     [];
decode(Seq, Table) ->
     {Char, Rest} = decode_char(Seq, 1, Table),
     [Char|decode(Rest, Table)].
```

Den andra lösningen ser snarlik ut:

```
two(Encoded, Result, Table) ->
    {ok, Seq} = file:read_file(Encoded),
    {ok, Out} = file:open(Result, [write]),
    decode(Sequence, Out, Table).

decode([], _Out, _Table) ->
     ok;
decode(Seq, Out, Table) ->
```

Namn:_____

```
    {Char, Rest} = decode_char(Seq, 1, Table),
    file:write(Out, [Char]),
    decode(Rest, Table).
```

Finns det någon fördel att göra på det ena eller andra sättet?

Svar: I den andra lösningen så har vi en svansrekursiv lösning eftersom vi hela tiden skriver de tecken vi hittar. Om filen är väldigt stor så kan det vara en fördel. Det är dock säkert mer effektivt att först avkoda hela filen och sen gör utskriften i ett svep så, om vi inte är oroliga för stacken så är den första lösningen att föredra.

Namn:_____

6.2 Meta-interpretator

6.2.1 eval_match/3 [6 poäng]

Koden nedan är saxat ur den meta-interpretator som vi implementerade. Förklara vad de olika parametrarna till funktionen är och varför vi returnerar de värden vi gör.

```
eval_match({var, Id}, Str, Env) ->
    case env:lookup(Id, Env) of
        false ->
            {ok, env:add(Id, Str, Env)};
        {Id, Str} ->
            {ok, Env};
        {Id, _} ->
            fail
    end;
```

6.2.2 apply [2 poäng*]

Antag att vi har utökat vår interpretator till att även hantera funktioner. Vi kan skapa s.k. *closures* som representeras av en sekvens av variabelidentifierare, en *sekvens* och en omgivning som ger de fria variablerna värden. Vi kan sen ta denna konstruktion och applicera den på en sekvens av argument. Koden nedan är den del i vår interpretator som skulle ansvara för att just den operationen.

```
eval_expr({apply, Expr, Args}, Env) ->
    case eval_expr(Expr, Env) of
        {ok, {closure, Par, Seq, Theta}} ->
            case eval_args(Args, Env) of
                error ->
                    error;
                EvaluatedArgs ->
```

Namn:_____

```
                    < What goes here? >

            end;
        error ->
            error
    end;
```

Om vi skall evaluera uttrycket `F(X,42)` så skall vi först evaluera `F` och förhoppningsvis då få en *closure*. Sen evaluerar vi argumenten `X` och `42` som förhoppningsvis går bra. Därefter skall vi göra något, beskriv vad vi måste göra?

Svar: We skall skapa en ny omgivning givet omgivningen `Theta` och bindningarna av variabelidentifierarna i `Par` och resultatet av de evaluerade argumenten `EvaluatedArgs`.

```
                EvaluatedArgs ->
                    New = env:args(Par, EvaluatedArgs, Theta),
                    eval_seq(Seq, New)
```

6.3 En server

6.3.1 en server och en klient [4 poäng]

När vi bygger en tjänst som använder sig av TCP så kommer servern och klienten använda sig av olika API:er för att skapa en förbindelse. Vem använder vad och hur används följande funktioner:

- `gen_tcp:connect(IP,Port,Opt)` **Svar:** Används av klienten när den kopplar upp sig mot en server. Kommer att returnera en *kommunikationskanal*.
- `gen_tcp:accept(Socket)` **Svar:** Används av servern när den har skapat en *listening Socket*. Kommer returnera en *kommunikationskanal* när en klient kopplar upp sig.
- `gen_tcp:listen(Port, Opt)` **Svar:** Används av servern för att skapa en *listening Socket*. Servern har registrerat sig som ägare till en port.

Namn:_____

6.3.2 snabbt men kanske för snabbt [4 poäng*]

Om vi implementerar en server så har vi nedan en lösning för att parallelisera implementationen. Vi skapar en Erlang-process för varje inkommande förfrågan och hanterar den separat samtidigt som vi omedelbart kan ta emot nästa.

```
handler(Listen) ->
    case gen_tcp:accept(Listen) of
  {ok, Client} ->
      spawn(fun() -> request(Client) end),
    handler(Listen);
  {error, Error} ->
    io:format("rudy: error ~w~n", [Error])
     end.
```

Vad är nackdelen med denna lösning?

Svar: Vi riskerar skapa fler Erlang processer än vad vi kan hantera. Vi har ingen kontroll på hur många vi skapar.

6.3.3 binary[2 poäng*]

Antag att vi läser in ett meddelande från en datagram-socket som vi har öppnat som `binary`. Det vi läser in är ett datagram som har följande struktur:

```
                                1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      ID                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|QR|   Opcode  |AA|TC|RD|RA|   Z    |   RCODE   |
:                                                :
```

Skriv en funktion `head/1`, som tar en binär och returnerar en struktur som innehåller identifieraren och de olika flaggorna kodade som heltal och resten av binären som en binär. Vi skall alltså kunna göra följande:

```
> head(<<"aabbfoobar">>).
{24929,0,12,0,1,0,0,6,2,<<"foobar">>}
```

Svar:

```
head(<<Id:16, QR:1, Op:4, AA:1, TC:1, RD:1, RA:1, Z:3, RC:4, Rest/binary>>) ->
   {Id, QR, Op, AA, TC, RD, RA, Z, RC, Rest}.
```