Programmering II (ID1019)
2017-06-07 1400-18:00

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.

- 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.

De som skriver 4.5hp versionen så skall bara svara på frågorna 1-6. Gränsen för E, D och C är då 14, 16 och 20 poäng. Gränserna för B och A är 8 respektive 12 poäng*. Om du inte har labmomentet avklarat så kan du även svara på fråga 7 där gränsen för godkänt är 10 grundpoäng.

Erhållna poäng

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 listoperationer [2 poäng]

Evaluerar utrycken nedan, vad blir X? Skriv svaret på så enkel form som möjligt dvs inte på den form som det står nu.

- `X = [[1,2] | [3,4]]` **Svar:** `X = [[1,2], 3, 4]`
- `X = [1] ++ [2,3,4]` **Svar:** `X = [1, 2, 3, 4]`
- `X = [1,2] ++ [[3,4]]` **Svar:** `X = [1, 2, [3 4]]`
- `X = [1 | [2,[3,4]]]` **Svar:** `X = [1, 2, [3, 4]]`

1.3 en korthög [2 poäng]

Om vi skall representera en korthög så kan vi antingen representerade den som en tuple, där varje element är ett kort, eller lista av kort. Det finns fördelar med båda representationerna, vilka?

Svar: Om vi representerar den som en tuple kan vi på konsant tid hitta det n:te kortet i högen. Om vi representerar det som en lista så kan vi på konsant tid plocka bort det första kortet och representera resten som en hög

Namn:_____

2 Rekursion

2.1 enkel rekursion [2 poäng]

Skriv en funktion, `transf(X,Y,L)`, som tar två värden, X och Y, och en lista av tal och returnerar en lista där alla värden lika med X har plockats bort och övriga multplicertas med Y. Om vi evaluerar `transf(5, 2, [2,5,4,3,5])` skall resultatet vara `[4,8,6]`.

Svar:

```
transf(_, _, []) -> [];
transf(X, Y, [X|R]) -> transf(X, Y, R);
transf(X, Y, [H|R]) -> [H*Y|transf(X, Y, R)].
```

2.2 svansrekursion [2 poäng]

Nedan finns definitionen av funktionen `sum/1` som summerar elementen i en lists. Skriv om fuktionen så att den är svansrekursiv.

```
sum([]) -> 0;
sum([H|T]) -> H + sum(T).
```

Svar:

```
sum(L) -> sum(L, 0);

sum([], S) -> S.
sum([H|T], S) -> sum(T, H+S).
```

Namn:_____

2.3 ett träd [2 poäng]

Antag att ett träd är representerat med atomen `nil` för en tom gren och strukturen `{node, Value, Left, Right}` för en nod i trädet där värdet i en nod är ett heltal. Skriv en funktion `min/1` som hittar det minsta värdet i ett träd eller returnerar `inf` om trädet är tomt. Du kan använda dig av det faktum att alla tal är *mindre* än atomer dvs `3 < inf` är sant.

Svar:

```
min(L) -> min(L, inf).

min(nil, M) -> M;
min({node, V, L, R}, M) ->
  if V < M -> min(L, min(R, V));
     true ->  min(L, min(R, M))
  end.
```

3 Högre ordningen

3.1 filter och map [2 poäng]

Givet definitionerna nedan, vad returnerar ett anrop till `foo([1,2,3,4,5])`?

- `filter(F, L)`: returnerar en lista av alla element, E, i L för vilka $F(E)$ evalueras till *true*.
- `map(F, L)`: returnerar en lista av $F(E)$ för varje element E i L.

```
foo(L) ->
  map(fun(X) -> X*3 end, filter(fun(X) -> X =/= 3 end, L)).
```

Svar: `[3,6,12,15]`

3.2 vänster eller höger [2 poäng*]

Givet definitionerna nedan, vad returnerar ett anrop till `bar([1,2,3,4])`?

- `foldl(F, A, L)`: returnerar värdet $F(E_n, F(...F(E_1, A)))$ där E_k är elementen i L.

Namn:_____

```
bar(L) ->
   foldl(fun(X, A) -> [X|A] end, [], L).
```

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

4 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.

4.1 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:_____

4.2 pattern matching [2 poäng*]

De fyra 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 $v/s \in \sigma$
- $P\sigma(v, s) \rightarrow \{v/s\} \cup \sigma$ if $v/t \notin \sigma$

Det är ett mycket viktigt fall vi inte täcker med dessa, vilket fall, hur ser regeln ut och när har vi nytta av den?

Svar: Fallet är där vi försöker göra en mönstermatching med en datastruktur (eller variabel bunden till en datastruktur) som inte är identisk med den datstruktur vi matcher mot. I detta fall är resultatet *fail* och det är viktigt att veta då vi implementerar t.ex. *case-uttryck*.

Namn:_____

4.3 konstigt uttryck [4 poäng*]

Antag att vi har utökat vårt språk till att även kunna hantera foo-uttryck. Vi har en syntax för dessa och evaluerar dem enligt följande regel där *param* en sekvens av variabler och *seq* en sekvens av uttryck.

- $E\sigma$((foo(param) -> sequence end) $\rightarrow foo(param, seq)$

Vi kan sedan ta denna konstruktion och applicera den på en sekvens av uttryck med hjälp av följande regel:

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

Detta gäller om *param* är en sekvens av i variabler $p_1, ..p_i$, $args$ är en sekvens av uttryck $a_1, ...a_i$ och $E\sigma(a_i)- > s_i$. Omgivningen θ är unionen av bindningar $\{p_1/s_1...\}$ och σ. Vi kan dock inte bara slå ihop dessa rakt av utan att hamna i problem. Vilket är problemet och hur skulle vi kunna hantera det?

Svar: Om p_k finns i σ så skulle vi få dubbla bindningar för en variabel. Vi måste antingen ta bort alla element p_k/t ur σ innan vi tar unionen eller på något sätt döpa om parametrarna så att de är unika.

Om vi hade en *foo-konstruktion* i Erlang, vad skulle då evalueringen av `test(2)` returnera? Motivera sitt svar.

```
test(Y) ->
  F = bar(6),
  F(3).

bar(Y) ->
  foo(X) -> X + Y end.
```

Svar: Evalueringen skulle returnera 5 eftersom vi applicerar F på 3 i en omgivning där Y är 2. F är i detta skede en foo-struktur som har kroppen X + Y, variablen X får sin bindning till 3 men foo har inte med sig en omgivning (det är inte en closure) där Y har ett värde. Y får sitt värde i den omgivning där vi gör appliceringen.

Namn:_____

5 Komplexitet

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

5.1 hitta värdet [2 poäng]

Funktionen nedan hittar värdet för en nyckel i ett träd. Vad är den assymptotiska tidskomplexiteten för funktionen, antag att trädet är balanserat.

```
lookup(_, nil) -> false;
lookup(K, {node, K, V, _, _}) -> {ok, V};
lookup(K, {node, _, _, L, R}) ->
      case lookup(K, L) of
         false -> lookup(K, R);
         Found -> Found
      end.
```

Svar: Funktionen har tidskomplexitet $O(n)$ där n är antalet element i trädet. Trädet är inte ordnat och vi söker igenom trädet tills vi hittar värdet.

5.2 svansrekursivt [2 poäng]

Funktionen `sentence/1` nedan tar en lista av ord och konstruerar en mening. Listan `["hej", "på", "dig"]` ger meningen `"Hej på dig."`. Funktionen `sentence/2` är svansrekursiv men vi gör detta till ett pris. Vad är den asymptotiska tidskomplexiteten för funktionen (orden kan antas vara av konstant längd k).

```
sentence([]) -> []
sentence([[W|Ord] | Words ]) ->
      sentence(Words, [string:to_capital(W)|Ord]).

sentence([], Sofar) -> append(Sofar, [$\.]);
sentence([Word|Words], Sofar) ->
       sentence(Words, append(Sofar, [$\ | Word])).
```

Svar: Tidskomplexiteten är n^2, där n är antalet ord i listan, eftersom vi kommer utföra allt längre append-operationer; variablen `Sofar` kommer ha en längd som är proportionell till n.

Namn:_____

5.3 traverser det här [2 poäng*]

Nedan har du en funktion som traverserar ett ordnat träd och returnerar alla värden som en ordnad lista. Vad är den asymptotiska tidskomplexiteten för funktionen, antag att trädet är balanserat. Motivera ditt svar.

```
trav(nil) -> [];
trav({node, V, L, R}) -> trav(L) ++ [V|trav(R)].
```

Svar: Funktionen har tidskomplexitet $O(n*lg(n))$ där n är antalet element i trädet. Djupet på trädet är $lg(n)$ och på varja djup så görs att antal append-operationer som har komplexitet n

Namn:_____

6 Concurrency

6.1 en summa [2 poäng]

Implementera en procedur som startar upp en process som har ett värde som tillstånd. Processen skall kunna hantera följande meddelanden:

- `{add, N}` : addera N till värdet
- `{sub, N}` : subtrahera N från värdet
- `{req, Pid}` : skicka `{total, V}`, där V är värdet, till Pid

Svar:

```
start(N) -> spawn(fun() -> sum(N) end).

sum(S) ->
  receive
    {add, N} -> sum(S + N);
    {sub, N} -> sum(S - N);
    {req, Pid} -> Pid ! {total, S}, sum(S)
  end.
```

Namn:_____

6.2 kardemumma [2 poäng]

Antag att vi har nedanstående definition av procedurerna `cardamom/1` och `momcarda/1`. Antag också att vi har startat en process med anropet `spawn(fun() -> cardamom(10) end)`. Antag seden att vi skickar följande sekvens av meddelanden till processen: `{sub, 4}, {add, 10}, {mul, 2}, {sub, 10}`. Vad är nu processens tillstånd?

```
cardamom(S) ->
    receive
        {add, X} -> momcarda(S + X);
        {sub, X} -> momcarda(S - X)
    end.

momcarda(S) ->
    receive
        {add, X} -> cardamom(S + X);
        {sub, X} -> cardamom(S - X);
        {mul, X} -> cardamom(S * X)
    end.
```

Svar: Processen kommer hantera meddelanden i den ordning de kommer men endast kunna hantera `mul` i tillståndet `momcard`. Efter sekvensen kommer processen att vara i tillståndet `cardamom(12)`. Det är viktigt att ange att vi är i `cardamom` eftersom vi då är i ett tillstånd där vi inte kan ta emot meddelanden på formen `{mul, N}`.

Namn:_____

6.3 ping-ping [4 poäng*]

Nedan finns en procedur som implementerar en process. Skriv en procedur som startar två sådan processer. Varje process skall ha ett unikt namn och den andra processens process-identifierare som sitt tillstånd. Du får naturligtvis skriva procedured som iniatliserar dessa tillstånd.

```
ping(Name, Pid) ->
  receive
    {frw, Msg} ->
           Pid ! {msg, Msg},
           ping(Name, Pid);
    {msg, Msg} ->
           io:format("~w received ~w~n", [Name, Msg]),
           ping(Name, Pid);
    terminate ->
           ok
  end.
```

Svar:

```
start() ->
   Foo = spawn(fun() -> init(foo) end),
   Bar = spawn(fun() -> ping(bar, Foo) end),
   Foo ! {connect, Bar}.

init(Name) ->
   receive
     {connect, Pid} -> ping(Name, Pid)
   end.
```

Namn:_____

7 Programmering

7.1 aritmetiska uttryck

7.1.1 representera aritmetiska uttryck [2 poäng]

Hur skulle vi kunna representera aritmetiska *uttryck*, som till exempel $2+3$ och $2+3*6$, om vi sedan skall arbeta med dessa och göra till exempel förenklingar mm.

Beskriv representationen för uttryck, där dess kan vara:

- heltal: 1, 2, 3 ...
- summa av uttryck: $2+3$ ...
- produkt av uttryck: $2*4$ ...

Notera att representationen skall vara så att man kan representera uttryck som till exempel $2+3*(4+6)$. Använd med fördel Erlangs typ-notation i din beskrivning.

Svar:

```
-type const() :: {const, integer()}.
-type sum() :: {add, expr(), expr()}.
-type prod() :: {mul, expr(), expr()}.
-type expr() :: sum()|prod()|const().
```

7.1.2 evaluering av uttryck [2 poäng]

Med antagandet om representationen i föregående uppgift, implementera en funktion `eval/1` som tar ett uttryck och returnerar dess *resultat*. Om vi anropar funktionen med representationen för $2+(3*4)$ skall vi returnera 14.

Svar:

```
eval({const, C}) -> C;
eval({add, E1, E2}) -> eval(E1) + eval(E2);
eval({mul, E1, E2}) -> eval(E1) * eval(E2);
```

7.1.3 varierande variabler [2 poäng]

Utöka representationen så att vi även kan hantera uttryck med variabler. Vi skall till exempel kunna representera uttryck så som: $2+x$ och $2+(3*y)$.

Namn:_____

Beskriv hur variabler representeras och hur detta påverkar vår representation av uttryck. Använd gärna Erlangs typ-notation.

Svar:

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

-type expr() :: sum()|prod()|const()| var().
```

7.1.4 variablers värde [2 poäng]

Beskriv representationen av en mängd *variabelbindningar*, dvs mappning från variabler till värden, och en funktion `lookup/2` som tar en variabel och en mängd bindningar som argument och returnerar ett värde för den sökta variabeln. Vi antar att den sökta variabeln finns i mappningen.

Svar: Eftersom vi vet att vi skall mappa variabler till värden kan vi välja en enklare representation. Vi kan skapa en mappning mellan atomerna som identifierar variablerna till godtyckliga värden.

```
-type env() :: [{atom(), any()}].

lookup(Var,[{Var, Value}|_]) -> Value;
lookup(Var,[_|Rest]) -> lookup(Var,Rest).
```

7.1.5 evaluera med variabler [2 poäng*]

Implementera en funktion `eval/2` som tar ett uttryck och en mängd variabelbindningar och returnerar värdet på det evaluerade uttrycket. Använd de representationer av variabler och variabelbindningar samt funktionen `lookup/2` i de föregående uppgifterna.

Svar:

```
eval({const, C}, _) -> C;
eval({var, V}, Env) ->  lookup(V, Env);
eval({add, E1, E2}, Env) -> eval(E1, Env) + eval(E2, Env);
eval({mul, E1, E2}, Env) -> eval(E1, Env) * eval(E2, Env);
```

Namn:_____

7.1.6 derivering av uttryck [2 poäng*]

Implementera en funktion `deriv/2` som tar ett uttryck, representerat som i uppgifterna ovan, och en variabel, och returnerar derivatan av uttrycket med avseende på variabeln. Deriveringsreglerna är som bekant som följer:

- $c' = 0$ givet att c är en konstant eller en variabel skild från den variabel vi deriverar över
- $x' = 1$ givet att x är den variabel vi deriverar över
- $(f(x) + g(x))' = f'(x) + g'(x)$
- $(f(x) * g(x))' = f'(x) * g(x) + f(x) * g'(x)$

Svar: Detta kan tyckas komplicerat men det är naturligtvis mycket enkelt.

```
deriv({const, _}, _) -> 0
deriv({var, V}, V) -> 1;
deriv({var, _}, _) -> 0;
deriv({add, F, G}, V) -> {add, deriv(F, V), deriv(G, V)};
deriv({mul, F, G}, V) ->
       {add, {mul, deriv(F, V), G}
             {mul, F, deriv(G, V)}}.
```

Namn:_____

7.2 ett adresserbart minne

7.2.1 en cell [2 poäng]

I modulen `cell`, implementera en process som håller ett värde som sitt tillstånd. Processen skall kunna svara på en sekvens av meddelanden. Följande meddelanden skall hanteras:

- `{set, V}`: sätt värdet till V
- `{get, Pid}`: sicka `{ok, V}` där V är värdet till Pid
- `{free, Pid}`: sicka `{ok, V}` där V är värdet till Pid, terminera processen

Modulen skall exportera en funktion `start/1` som tar ett värde och returnerar en processidentifierare för en cell-process.

Svar:

```
-module(cell).
-export([start/1]).

start(N) ->
    spawn(fun() ->  cell(N) end).

cell(N) ->
    receive
        {set, V} ->
            cell(V);
        {get, Pid} ->
            Pid ! {ok, N},
            cell(N)
        {free, Pid} ->
            Pid ! {ok, N},
            ok
    end.
```

Namn:_____

7.2.2 ett funktionellt gränssnitt [2 poäng]

I samma modul, implementera och exportera tre funktioner:

- `set(Pid, V)`: givet en pid för en cell så sätts cellens värde till V, returnera ok
- `get(Pid)`: givet en pid för en cell så returneras `{ok, V}` där V är cellens värde
- `free(Pid)`: givet en pid för en cell så returneras `{ok, V}` där V är cellens värde, cellen terminerar

Svar:

```
-export([set/2, get/1]).

set(Pid, V) ->
    Pid ! {set, V},
    ok.

get(Pid) ->
    Pid ! {get, self()},
    receive
        {ok, V} ->
            {ok, V}
    end.

free(Pid) ->
    Pid ! {free, self()},
    receive
        {ok, V} ->
            {ok, V}
    end.
```

Namn:_____

7.2.3 uppdatera [2 poäng]

Givet definitionen nedan vad blir resultatet när vi gör anropet `test(1000)`. Motivera.

```
test(N) ->
     Cell = cell:start(0),
     Self = self(),
     spawn(fun() -> add(N, Cell, Self) end),
     spawn(fun() -> add(N, Cell, Self) end),
     receive
        done ->
            receive
               done ->
                    cell:free(Cell)
            end
     end.

add(0, _, Master) ->
    Master ! done;
add(N, Cell, Master) ->
    {ok, V}= cell:get(Cell),
    cell:set(Cell, V+1),
    add(N-1, Cell, Master).
```

Svar: Resultatet är allt ifrån `{ok, 2}` till `{ok, 2000}` beroende på i vilken ordning de olika processernas meddelanden tas emot. Om båda processerna läser värdet X så kan den ene hinna uppdatera cellens värde flera gånger innan den andre hinner skriva X+1. Den andra processen kommer då att läsa X+1 även om den själv har utfört flera uppdateringar däremellan.

Namn:_____

7.2.4 ett adresserbart minne [2 poäng]

I en modul `mem`, implementera en funktion `new/1` som tar en lista av värden och returnerar en tuple av process-identifierarare för celler som initialiserats med värdena från listan. Du får använda dig av en inbyggs funktion `list_to_tuple/1` som tar en lista av element och returnerar motsvarande tupel.

Svar:

```
-module(mem).

-export([new/1]).

new(Values) ->
    Pids = lists:map(fun(V) -> cell:start(V) end, Values),
    list_to_tuple(Pids).
```

7.2.5 funktionellt interface [2 poäng*]

Implementera funktioneran `get/2` och `set/3` i modulen `mem`. Använd den inbyggda functionen `element/2` som tar en position och en tupel och returnerar elementet för den poistionen. Funktionerna skall använda sig av de exporterade funktionerna från `cell` och inte känna till vilka meddelande som används internt av cell-processen.

Svar:

```
set(Array, N, V) ->
    cell:set(element(N, Array), V).

get(Array, N) ->
    cell:get(element(N, Array)).
```

Namn:_____

7.2.6 asynkront anrop [2 poäng*]

Vårt gränssnitt för att hämta ett värde är nu synkront men vi skulle vilja ha ett asynkront gränssnitt. Lägg till de funktioner som behövs i modulen `cell` så att vi får en funktion `get_asyn/2` som initierar en läsning av en cell men som returnerar en unik referens (använd `make_ref/0`). Referensen kan sedan användas för att ta emot svaret genom att anropa `get_answ/1` som då returnerar `{ok, V}`.

Svar:

```
get_asyn(Pid) ->
    Ref = make_ref(),
    Pid ! {get, Ref, self()},
    Ref.

get_answ(Ref) ->
    receive
        {ok, Ref, V} ->
            {ok, V}
    end.

cell(N) ->
    receive
        {set, V} ->
            cell(V);
        {get, Pid} ->
            Pid ! {ok, N},
            cell(N);
        {get, Ref, Pid} ->
            Pid ! {ok, Ref, N},
            cell(N)
    end.
```

7.2.7 summera värden [2 poäng*]

Använda det asynkrona gränssnittet och implementera en funktion `sum/1` som summerar alla talen i ett minne. Du skall använda dig av de högre ordningens funktionerna `lists:foldl(Fun, Acc, List)` och `lists:map(Fun, List)`. Du kan använda de inbyggda funktionerna `size/1` som returnerar storleken på en tupel och `lists:seq(From, To)` som returnerar en lista med heltal från From till To.

Svar:

Namn:_____

```
sum(Array) ->
    Ids = lists:seq(1, size(Array)),
    Refs = lists:map(fun(I) -> get_asyn(Array, I) end, Ids),
    lists:foldl(fun(R, A) -> {ok, V} = get_answ(R), A+V end, 0, Refs).
```