Tillämpad programmering

Erlang II

Johan Montelius

Datastrukturer

- literaler
  - atomer: foo, gurka, 'ett o två'
  - nummer: 1,2,.. 3.14 ...
- samansatta (compound)
  - tupler: {foo, a, b, 2}
  - cons-cell: [gurka | tomat]

Tupler

```
{A, 42, C} = {3, 42, gurka}
```

{A = 3 , C = gurka }

```
{A, 25, C} = {13, 23, 33}
```

bad match

```
{A, 23, A} = {13, 23, 33}
```

bad match

cons-celler

```
[H | T] = [a,b,c,d]
```

$\{H = a\ , T = [b,c,d]\ \}$

```
[X, Y | T] = [1,2,3,4,5]
```

$\{X = 1, Y = 2\ \ , T = [3, 4, 5]\ \}$

```
[H | T] = [foo]
```

$\{H = foo,\ \ T = [\,]\ \}$

vad är en lista?

En <u>lista</u> är antingen:

en tom lista (nil) eller,

en cons-cell där svansen är en <u>lista</u>.

```
[]

[1,2,3]          [1 | [2,3]]

[1 | X]
```

proper/1 - är det en riktig lista

```
-spec proper([any()]) -> bool.

proper(L) ->
    case L of
       [] -> ...;
       [_|T] -> ...;
       _  -> false
    end.
```

proper/1 - är det en riktig lista

```
-spec proper([any()]) -> bool.

proper(L) ->
     case L of
        [] -> true;
        [_|T] -> dont_know;
         _ -> false
     end.
```

proper/1 - är det en riktig lista

```
-spec proper([any()]) -> bool.

proper(L) ->
      case L of
         [] -> true;
         [_|T] -> magic(T);
          _  -> false
      end.
```

proper/1 - är det en riktig lista

```
-spec proper([any()]) -> bool.

proper(L) ->
      case L of
         [] -> true;
         [_|T] -> proper(T);
         _  -> false
      end.
```

append/2 - en lista av två

```
-spec append([_], [_]) -> [_].

append(Xs, Ys) ->
    case Xs of
       [] -> ...;
       [H|T] ->
          [ H | ... ]
    end.
```

member/2 - finns i listan

```
-spec member(_, [_]) -> bool.

member(X, L) ->
    case L of
        [] -> ...;
        [X | _] -> ...;
        [_ | T] -> ...
    end.
```

length/1 - längden av en lista

```
length(L) ->
      case L of
          [] -> ...;
          [_|T] -> ....
      end.
```

hmmm...

```
length([1,2,3,4])
    1 + length([2,3,4])
        1 + length([3,4])
            1 + length([4])
                1 + length([])
                    0
                1
            2
        3
    4
```

length/2 - svansrekursiv

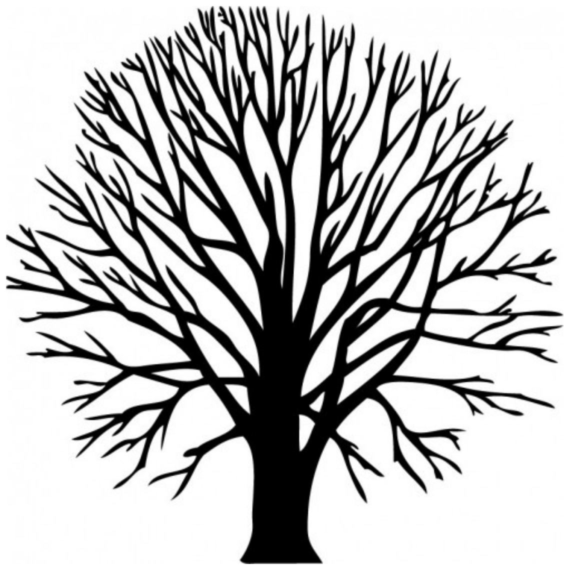
```
length(L) ->
        length(L,0).

length(L, S) ->
        case L of
              [] -> S;
              [_|T] ->
                  S1 = S + 1,
                  length(T, S1)
        end.
```

ingen stack!

```
length([1,2,3,4])
length([1,2,3,4], 0)
length([2,3,4], 1)
length([3,4], 2)
length([4], 3)
length([], 4)
4
```

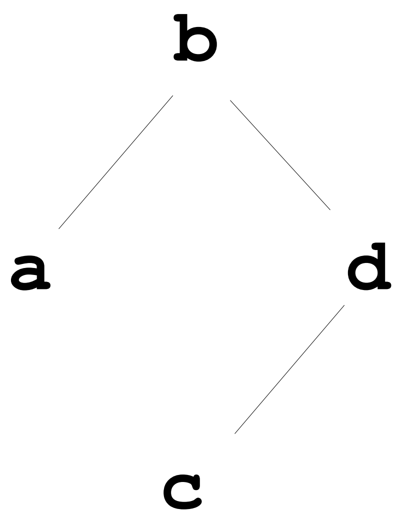
trädstrukturer



nil

{node, Val, Left, Right}

```
{node, b,
     {node, a, nil, nil},
     {node, d,
          {node, c, nil, nil},
          nil}}
```



is_tree/1 - är det ett träd?

```
is_tree(Tree) ->
    case Tree of
        nil -> ...;
        {node, _, Left, Right} ->
          ... and ...;
          _ -> false
    end.
```

insert/3 - lägg in i träd

```
insert(Val, nil) ->
     {node, Val, nil,nil};

insert(Val, {node, V, L, R}) when Val < V ->
     Inserted = insert(Val, L),
     {node, V, Inserted, R};

insert(Val, {node, V, L, R}) when Val >= V ->
     Inserted = insert(Val,R),
     {node, V, L, Inserted}.
```

same-same-but-different

```
insert(Val, Tree) ->
    case Tree of
        nil -> {node, Val, nil,nil};
        {node, V, L, R} ->
            if
                Val < V ->
                    Inserted = insert(Val, L),
                    {node, V, Inserted, R};
                Val >= V ->
                    Inserted = insert(Val, L),
                    {node, V, L, Inserted}
            end
    end.
```

member/2

```
member(Val, nil) -> false;

member(Val, {node, Val, _, _}) -> true;

member(Val, {node, V, L, _}) when Val < V ->
        member(Val, L);

member(Val, {node, V, _, R}) when Val > V ->
        member(Val, R);
```

key-value store (som lista)

[]

[{Key, Val}, ..]

```
[{foo, 13}, {bar, 42}]
```

key-value store

new() -> Store

is_store(Store) -> bool

add(Key, Value, Store) -> Store

lookup(Key, Store) -> {ok, Value} , fail

delete(Key, Stor) -> Store

-module(store)

```
-module(store).

-export([new/0,
         is_store/1,
         add/3,
         lookup/2,
         delete/2]).
```

-module(store)

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

-spec new() -> store().

-spec is_store(any()).

-spec add(atom(), any(), store()) ->
          store().

-spec lookup(atom(), store()) ->
          {ok, any()} | fail.

-spec delete(atom(), store()) -> store().
```

new/0 add/3

```
new() -> ??.

add(Key, Value, Store) ->
        ??.
```

is_store/1

```
is_store([]) -> true;

is_store([{Key,_}| Rest]) when is_atom(Key) ->
      is_store(Rest);

is_store(_) -> false.
```

lookup/3

```
lookup(_, []) -> fail;

lookup(Key, [{Key, Val}|_]) ->
     {ok, Val};

lookup(Key, [_|Rest]) ->
     lookup(Key, Rest).
```

delete/2

```
delete(_, []) ->
     [];

delete(Key, [{Key, _}|Rest]) ->
     Rest;

delete(Key, [Elem|Rest]) ->
     ... .
```

modulen - abstraktion

```
test() ->
        S0 = store:new(),
        S1 = store:add(foo, 13, S0),
        S2 = store:add(bar, 12, S1),
        S3 = store:delete(foo, S2),
        store:lookup(bar, S3).
```

key-value store (som ordnat träd)

nil

{node, Key, Value, Left, Right}

```
{node, foo, 14,
     {node, bar, 12, nil, nil},
     {node, zot, 34, nil, nil}}
```

add/3

```
add(Key, Val, nil) ->
     {node, Key, Val, nil, nil}.

add(Key, Val, {node,K,V,L,R}) when Key =< K ->
     Added = add(Key, Val, L),
     {node, K, V, Added, R};

add(Key, Val, {node,K,V,L,R}) when Key > K ->
     Added = add(Key, Val, R),
     {node, K, V, L, Added}.
```

lookup/2

```
lookup(Key, nil) -> ...;

lookup(Key,{node,Key,Val,_,_}) -> ...;

lookup(Key,{node,K,_,L,_}) when Key < K ->
     ...;

lookup(Key,{node,K,_,_,R}) when Key > K ->
     ....
```

delete/2

```
delete(_, nil) -> ...;

delete(Key,{node,Key,_,L,nil}) -> ...;

delete(Key,{node,Key,_,nil,R}) -> ...;

delete(Key,{node,K,V,L,R}) when Key < K ->
     Deleted = ...
     ...;

delete(Key,{node,K,V,L,R}) when Key > K ->
     Deleted = ...
     ...;
```

delete/2

```
delete(Key,{node,Key,_,L,R}) ->
     {Leftmost, Value} = leftmost(R),
     Removed = ...,
     {node, .., .., .., ..}
```

letmost/2

```
leftmost({node, Kel, Val, nil, _}) ->
    ...;
leftmost({node,Key,Val,L,_}) ->
    leftmost(...).
```

beskriv ett kort

{card , Suit, Value}

suit = (heart, club, spade, diamond)

value = (ace, king, queen, knight, 10, ... 2)

```
{card, heart, king}
```

```
{card, club, 10}
```

```
{card, spade, 2}
```

jämföra färg

```
-module(suite).

-export([gr/2]).

gr(heart, Suite) ->
     Suite =/= heart;
gr(dimond, Suite) ->
     Suit == club or Suite == spade;
gr(club, Suite) ->
     Suite == spade;
gr(spade, _) ->
     false.
```

jämföra valör

```
-module(value).
-export([gr/2]).

gr(ace, Value) ->
     Value =/= ace;
gr(king, Value) ->
     (Value == queen) or gr(queen, Value);
gr(queen, Value) ->
     (Value == knight) or gr(knight, Value);
gr(knight, Value) ->
     is_integer(Value);
gr(Value1, Value2)  ->
     is_integer(Value1) and
     is_integer(Value2) and
     Value1 > Value2.
```

jämföra kort

```
-module(card).

gr({card, Suit, Val1}, {card, Suit, Val2}) ->
     value:gr(Val1, Val2).

gr({card, Suit1, _},{card, Suit2, _}) ->
     suit:gr(Suit1, Suit2);
```

```
card:gr({card, heart,5},{card, spade, 2}).
```

sortera kort

```
sort([]) -> [];

sort([Card|Rest]) ->
     {Low, High} = split(Card, Rest),
     Sorted_low = sort(Low),
     Sorted_high = sort(High),
     append(Sorted_low, [Card|Sorted_high]).
```

sortera kort

```
split(_, []) -> [];

split(Card, [First|Rest]) ->
     case card_gr(Card, First) of
        true ->
          {Low, High} = split(Card, Rest},
          {[First|Low], High};
        false ->
          {Low, High} = split(Card, Rest},
          {Low, [First|High]}
     end.
```

sortera kort

```
split(Card, Deck) ->
     split(Card, Deck, [], []).

split(_, [], Low, High) ->
     {Low, High};
split(Card, [First|Rest], Low, High) ->
     case card_gr(Card, First) of
        true ->
          split(Card, Rest,[First|Low],High);
        false ->
          split(Card, Rest,Low,[First|High])
     end.
```

modulen som abstraktion

- Gömmer detaljer i implementationen:
  - abstrakt datatyp
- De exporterade funktionerna ger API:
  - konstruktorer
  - dekonstruktorer
  - jämföra
  - transformera
  - etc