# Tillämpad programmering

## Erlang III

Johan Montelius

# append/2 - en lista av två

```
append([], Ys) -> ...;
append([H|T], Ys) ->
     Appended = append(T,Ys),
     ....
```

# reverse/1 - vänd en lista

```
reverse([]) -> ...;
reverse([H|T]) ->
     Reversed = reverse(T),
     ....
```

# partition/2 - dela upp

```
partition(_,[]) -> {[], []};
partition(Piv, [H|T]) ->
     {Low, High} = partition(Piv, T),
     if
        H < Piv -> ....;
        true -> ...;
     end.
```

# Rekursiva funktioner

- Många gånger relativt enkelt.
  - Vad skall man rekursera över?
  - Hur ser basfallet ut?
  - Hur skall man bygga upp ett svar från ett svar på det rekuriva anropet?
- Den enkla lösningen kanske inte är den mest effektiva.

# append/2 - en lista av två

```
append([], Ys) -> ...;
append([H|T], Ys) ->
        Appended = append(T,Ys),
        ....
```

Inte svansrekursiv!

# reverse/1 - vänd en lista

```
reverse([]) -> ...;
reverse([H|T]) ->
     Reversed = reverse(T),
     ....
```

O(?)

# reverse/1 - vänd en lista

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

reverse([], R) -> ...;
reverse([H|T], R) ->
        reverse(T, ...).
```

# partition/2 - dela upp

```
partition(Piv, L) ->
     partition(Piv, L, [], []).

partition(_,[], Low, High) ->
     ...;
partition(Piv, [H|T], Low, High) ->
     if
        H < Piv ->
          partition(Piv, T, ..., ...);
        true ->
          partition(Piv, T, ..., ...)
     end.
```

# append/2 - en lista av två

```
append(Xs, Ys) ->
        append(Xs, Ys, []).

append([], Ys, Rs) ->
        ...;
append([H|T], Ys, Rs) ->
        append(T, Ys, [H|Rs]).
```

Svansrekursiv!

Har det någon betydelse?

# sortera en lista

- merge sort
  - dela listan i två lika stora listor
  - sortera listorna
  - skapa en lista som plockar det största elelemtet från vardera lista
- quick sort
  - dela upp en lista i de låga respektive höga elementen
  - sortera listorna
  - sammanfoga de sorterade listorna

# msort/1 - merge sort

```
msort([]) -> [];
msort([H]) -> [H];
msort(L) ->
     {A, B} = split(L),
     Sorted_A = msort(A),
     Sorted_B = msort(B),
     merge(Sorted_A, Sorted_B).
```

# split/1 - dela listan

```
split([]) -> {[], []};

split([H|T]) ->
        {A, B} = split(T),
        { ... , ... }.
```

# split/1 - svansrekursiv

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

split([], A, B) -> ...;
split([H|T], A, B) ->
     split(T, ..., ...).
```

# merge/2

```
merge([], B) -> ...;
merge(A, []) -> ...;
merge([A|Ar], [B|Br]) when A < B ->
         ....;
merge([A|Ar], [B|Br]) when A >= B ->
         .....
```

# merge/2

```
merge([A|Ar], [B|Br]) -> when A < B ->
     [A | merge(Ar, [B|Br]);

merge(As, Bs) ->
     [A|Ar] = As,
     [B|Br] = Bs,
     if
        A < B ->
          [A | merge(Ar, Bs)];
        true ->
          [B | merger(As, Br)
     end.
```

# 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(spade, Suite) ->
     Suit == diamond or Suite == spade;
gr(diamond, Suite) ->
     Suite == club;
gr(club, _) ->
     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} = partition(Card, Rest),
     Sorted_low = sort(Low),
     Sorted_high = sort(High),
     append(.., ...).
```

# sortera kort

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

partition(_, [], Low, High) ->
     {Low, High};

partition(Card, [First|Rest], Low, High) ->
     case card:gr(Card, First) of
        true ->
          partition(Card, Rest [First|Low], High);
        false ->
          partition(Card, Rest, Low, [First|High])
     end.
```

# högre ordningen

```
F = fun(X) -> X + 3 end.
```

```
F(4)
```

# sum/1

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

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

# prod/1

```
prod(L) -> prod(L, 1).

prod([], P) -> P;
prod([H|T], P) ->
     sum(T, H*P).
```

# foldl/1

```
-spec fold(fun(), any(), [any()]) -> any().

foldl(Op, Acc, []) -> Acc;

foldl(Op, Acc, [H|T]) ->
     foldl(Op, Op(H,Acc), T).
```

# sum/1

```
sum(L) ->
      Op = fun(X,Acc) -> X + Acc end,
      Acc = 0,
      foldl(Op, Acc, L).
```

# prod/1

```
prod(L) ->
      Op = fun(X,Acc) -> X * Acc end,
      Acc = 0,
      foldl(Op, Acc, L).
```

# sortera kort

```
partition(Card, _, Deck) ->
     partition(Card, Op, Deck, [], []).

partition(_, _, [], Low, High) ->
     {Low, High};

partition(Card, Gr, [First|Rest], Low, High) ->
     case Gr(Card, First) of
        true ->
          partition(Card,Gr,Rest,[First|Low], High);
        false ->
          partition(Card,Gr,Rest,Low, [First|High])
     end.
```

# que/1

```
que(L) ->
     Op = fun(X,Acc) -> [X|Acc] end,
     Acc = [],
     foldl(Op, Acc, L).
```

# foldr/1

```
foldr(Op, Acc, []) -> Acc;

foldr(Op, Acc, [H|T]) ->
      Op(H, foldr(Op, Acc, T)).
```

# Abstraktioner

- Moduler
  - gömmer detaljer i implementationen:
  - abstrakt datatyp
- Högre ordningens funktioner
  - ger generella algoritmer