Tillämpad programmering

Erlang VI

Johan Montelius

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 foldl(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).
```

sum/1

```
sum([1,2,3])
foldl(Op, 0, [1,2,3]) {Op = fun(X,Axx) -> X+Acc end}
foldl(Op, (1 + 0), [2,3])
foldl(Op, (2 +(1 + 0)), [3])
foldl(Op, (3 + (2 + (1 + 0))), [])
(3 + (2 + (1 + 0)))
```

prod/1

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

prod/1

```
prod([1,2,3])
foldl(Op, 0, [1,2,3]) {Op = fun(X,Axx) -> X*Acc end}
foldl(Op, (1 * 0), [2,3])
foldl(Op, (2 * (1 * 0)), [3])
foldl(Op, (3 * (2 * (1 * 0))), [])
(3 * (2 * (1 * 0)))
```

sortera kort

```
partition(Card, Op, 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.
```

konstigt/1

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


konstigt([1,2,3])
```

foldl/1

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

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

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

foldr/1

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

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

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

sum/1

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

sum([1,2,3])
foldr(Op, 0, [1,2,3]) {Op = ...}
(1 + foldr(Op, 0, [2,3]))
(1 + (2 + foldr(Op, 0, [3])))
(1 + (2 + (3 + foldr(Op, 0, []))))
(1 + (2 + (3 + 0)))
```

map/2


```
map(Op, []) ->
      ...;

map(Op, [H|T]) ->
      [... | ...].

add(I, L) ->
      Op = fun(X) -> ... end,
      map(Op, L).

add(5, [1,2,3])
```

filter/2


```
filter(Op, []) ->
      ...;

filter(Op, [H|T]) ->
      case ... of
            true ->
                  ....;
            false ->
                  ....
      end.
```

Högre ordningens funktioner

- Abstraherar algoritmer:
  - generella functioner som kan parametriseras för sin domän
- Jämför *interfaces* i Java
  - ger oss möjlighet att skriva generella procedurer som
- Jämför *polymorfism* i C++
  - som vi kommer att titta på senare

closure - continuation

- Ge mig en funktion som tar ett argument och returnera:
  - {done, N} om argumentate är **nil**
  - {more, F} annars
- N är summan av ***talen som vi sett***
- F är en funktion som vi kan applicera på ***nästa tal***

closure - continuation

```
count(nil, N) ->
      {done, N};

count(I, S) ->
      {more, ...}


counter() ->
      fun(X) -> ... end.
```

lite roligt med listor


```
[ X*2 || X <- [1,2,3]]

[2,4,6]

[{X, Y} || X <- [1,2],3 Y <- [3,4]]

[{1,3}, {1,4}, {2,3}, {2,4}]

[X || X <- [1,2,3,4], X > 2]

[3,4]
```

qsort/1

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

qsort([H|T]) ->
      Low  = [X || X <- ... , ...],
      High = [X || X <- ... , ...]
      qsort(Low) ++ [H] ++ qsort(High).
```