Higher order

Johan Montelius

KTH

VT23

let's play some cards

Properties of a card:

- Suit $\in \{\text{spade}, \text{heart}, \text{diamond}, \text{club}\}$
- Value $\in \{2, 3, \ldots 14\}$
- Card $\in \{\langle s, v\rangle | \quad s \in \text{Suit} \wedge v \in \text{Value}\}$

In Elixir:

```
@type suite :: :spade | :heart | :diamond | :club
@type value :: 2..14
@type card :: {:card, suite, value}
```


order of cards

```
def lt({:card, s, v1}, {:card, s, v2})  do  v1 < v2 end

def lt({:card, :club, _}, _) do  true end

def lt({:card, :diamond, _}, {:card, :heart, _}) do true end

def lt({:card, :diamond, _}, {:card, :spade, _}) do true end

def lt({:card, :heart, _}, {:card, :spade, _}) do true end

def lt({:card, _, _}, {:card, _, _}) do false end
```


sorting cards

```
@spec sort([card]) :: [card]

def sort([]) do [] end
def sort([c]) do [c] end

def sort(deck) do
  {d1, d2} = split(deck)
  s1 = sort(d1)
  s2 = sort(d2)
  merge(s1, s2)
end
```

```
@spec split([card]) :: {[card],[card]}

def split([]) do {[], []} end

def split([c|rest]) do
  {s1, s2} = split(rest)
  { ..., ...}
end
```

tail recursive split

```
def split([]) do
  {[], []}
end

def split([c|rest]) do
  {s1, s2} = split(rest)
  { ..., ...}
end
```

```
def split(deck) do split(deck, [], []) end

@spec split([card], [card], [card]) ::
   {[card],[card]}

def split([], d1, d2) do  {d1, d2} end

def split([c|rest], d1, d2) do
   split(rest, ..., ...)
end
```


sorting cards

```
@spec merge([card], [card]) :: [card]

def merge([], s2) do ... end
def merge(s1, []) do ... end

def merge([c1|r1]=s1, [c2|r2]=s2) do
  case lt(c1, c2) do
    true ->
      [c1| merge(r1, s2)]
    false ->
      [c2| merge(s1, r2)]
  end
end
```


what to do

Implement function that sorts names of people.

Implement function that sorts a frequency table.

Implement function that sorts ....

old friends

Have we seen this before?

sum/1

```
def sum([]) do  0 end
def sum([h|t]) do
    add(h,sum(t))
end
```

prod/1

```
def prod([]) do 1 end
def prod([h|t]) do
   mul(h,prod(t))
end
```

There is no built-in add/2, nor mul/2, but we can pretend that there is.

good to have

We would like to to this:

foldr/3

```
def foldr([], acc, op) do acc end


def foldr([h|t], acc, op) do
   op.(h, foldr(t, acc, op))
end
```

sum/1

```
def sum(l) do
  add = ...
  foldr(l, ..., add)
end
```

prod/1

```
def prod(l) do
  mul = ...
  foldr(l, ..., mul)
end
```


lambda expressions

We introduce a new data structure: a closure

$$\begin{array}{rcl} \textit{Atoms} & = & \{\text{a, b, c}, \ldots\} \\ \textit{Closures} & = & \{<\text{p:s:}\theta> \mid \text{p} \in \text{Parameters} \wedge \text{s} \in \text{Sequences} \wedge \theta \in \text{Environments}\ \} \\ \textit{Structures} & = & \textit{Closures} \cup \textit{Atoms} \cup \{\ \{\text{a, b}\}\ | \text{a} \in \textit{Structures} \wedge \text{b} \in \textit{Structures}\ \} \end{array}$$

A *closure* is a function and an environment.

We have not really defined what *Parameters*, a *Sequences* nor *Environments* are, but let's forget this for a while.

syntax for function expressions

```
  <function>  ::= 'fn' '(' <parameters> ')' '->' <sequence> 'end'
<parameters>  ::= ' ' | <variables>
 <variables>  ::= <variable> | <variable> ',' <variables>


<application>  ::= <expression> '.(' <arguments> ')'
  <arguments>  ::= ' ' | <expressions>
<expressions>  ::= <expression> | <expression> ',' <expressions>


<expression>  ::= <function> | <application> | ...
```


function expressions

We will write:

```
x = 2; f = fn (y) -> x + y end; f.(4)
```

Remember this?

$$\lambda y \rightarrow x + y$$

evaluation of a function expression

$$\frac{\theta = \{v/s \mid v/s \in \sigma \wedge v \text{ free in sequence}\}}{E\sigma(\texttt{fn (parameters) -> sequence end}) \rightarrow \quad \langle \text{parameters} : \text{sequence} : \theta\rangle}$$

```
x = 2; f = fn (y) -> x + y end; f.(4)
```

What is `f`?

evaluation of a function application

$$\frac{E\sigma(f) \rightarrow < v_1, \ldots : \text{seq} : \theta > \qquad E\sigma(e_i) \rightarrow s_i \qquad E\{v_1/s_1, \ldots\} \cup \theta(\text{seq}) \rightarrow s}{E\sigma(f.(e_1, \ldots)) \rightarrow s}$$

```
x = 2; f = fn (y) -> x + y end; f.(4)
```

What is `f.(4)`?

example

```
def foo(x) do
  y = 3
  fn (v) -> v + y  + x
end
```

```
f = foo(2); x = 5;  y = 7; f.(1)
```


case closed

sum/1

```
def sum(l) do
  add = fn (x,y) -> x + y end
  foldr(l, 0, add)
end
```

prod/1

```
def prod(l) do
  mul = fun(x,a) -> x * a end
  foldr(l, 1, mul)
end
```

example

```
def foldr([], acc, op) do acc end
def foldr([h|t], acc, op) do
   op.(h, foldr(t, acc, op))
end
```

What is gurka/1 doing?

```
def gurka(l) do
  f = fn (_, a) -> a + 1 end
  foldr(l, 0, f)
end
```

How about tomat/1?

```
def tomat(l) do
  f = fn (h, a) ->  a ++ [h] end
  foldr(l, [], f)
end
```


example

foldr/3

```
def foldr([], acc, op) do acc end
def foldr([h|t], acc, op) ->
   op.(h, foldr(t, acc, op))
end
```

foldl/3

```
def foldl([], acc, op) do acc end
def foldl([h|t], acc, op) do
  foldl(t, op.(h, acc), op)
end
```


left or right

Which one should you use, *fold-left* or *fold-right*?

append all

Append all lists in a lists.

```
def flatten_r(l) do
  f = fn (e,a) -> e ++ a end
  foldr(l, [], f)
end

def flaten_l(l) do
  f = fn (e,a) -> a ++ e end
  foldl(l, [], f)
end
```

List module

- `foldr(list, acc, f)`: fold from right `f.(x1, .. f.(xn, acc)..)`
- `foldl(list, acc, f)`: fold from left `f.(xn, .. f.(x1, acc)..)`

more patterns

map/2 : apply a function to each element in a list

```
def map([], _) do ... end
def map([h|t], f) do
   [ f.(h) | ... ]
end
```

```
def map(lst, f) do map(lst, f, []) e

def map([], _, sofar) do reverse(sof
def map([h|t], f, sofar) do
   map(t, f, [f.(h)|sofar])
end
```


more patterns

filter/2 : return a list with all elements that meet a criteria

```
def filter([], _) do ... end
def fiter([h|t], f) do
   if f.(h) do
      [ h | ... ]
   else
      ...
   end
end
```


Enum module

- `map(enum, f)`: return the list of `f.(x)` for each element `x` in the enumeration
- `filter(enum, f)`: return a list of all elements `x`, for which `f.(x)` evaluates to true
- `split_with(enum, f)`: partition the enumeration based on the result of `f.(x)`
- `sort(enum, f)`: sort the list given that the function $\hat{f}$ is less than or equal

Higher order

Order of what?

A first order function takes a value, a data structures, as argument and returns a value.

A second order function takes a first order function as argument or returns a first order function.

A third order function ....

Higher order functions takes a higher order ...

Are functions considered to be "first-class citizen"?

truly first-class?

Not really - look at this.

```
f = fn(x) -> x + 1 end
g = fn(y) -> y + 1 end
```

f == g

Summary

Higher order programming:

- closure: a function and an environment
- generic algorithms: separate the recursive pattern from the data it operates over

list comprehension

Generate a list, given some critera.

Compare *set comprehension*:

$$\{x \in \{1, 3, 6, 2, 5, 7, 3\} | x < 4\}$$

```
for x when x < 4 <- [1,3,6,2,5,7,0] do x  end

for {:ok, x} <- [{:ok, 1},{:no, 3},{:ok, 6}] do x  end

for i <- [1,2,3], j <- [:a,:b] do {i,j}  end
```