let’s play some cards

Properties of a card:
- Suit ∈ \{spade, heart, diamond, club\}
- Value ∈ \{2, 3, ..., 14\}
- Card ∈ \{(s, v) | s ∈ Suit ∧ v ∈ Value\}

In Elixir:

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

order of cards

def \text{lt}(\{\text{:card, s, v1}\}, \{\text{:card, s, v2}\}) \text{ do } v1 < v2 \text{ end}
def \text{lt}(\{\text{:card, :club, _}\}, _) \text{ do } \text{true end}
def \text{lt}(\{\text{:card, :diamond, _}\}, \{\text{:card, :heart, _}\}) \text{ do } \text{true end}
def \text{lt}(\{\text{:card, :diamond, _}\}, \{\text{:card, :spade, _}\}) \text{ do } \text{true end}
def \text{lt}(\{\text{:card, :heart, _}\}, \{\text{:card, :spade, _}\}) \text{ do } \text{true end}
def \text{lt}(\{\text{:card, _, _}\}, \{\text{:card, _, _}\}) \text{ do } \text{false end}

sorting cards

@spec sort([\text{card}]) :: [\text{card}]
def sort([]) \text{ do } [] \text{ end}
def sort([c]) \text{ do } [c] \text{ end}
def sort(deck) \text{ do }
\{d1, d2\} = \text{split}(\text{deck})
s1 = \text{sort}(d1)
s2 = \text{sort}(d2)
merge(s1, s2)
end
tail recursive split

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

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

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

sorting cards

```elixir
@spec split([card], [card], [card]) :: [card]
def split([], d1, d2) do {d1, d2} end

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

what to do

Implement function that sorts names of people.
Implement function that sorts a frequency table.
Implement function that sorts ....

old friends

```elixir
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 do 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
```

only problem, ...How do we express the function?

10 / 34

---

lambda expressions

We introduce a new data structure: a closure

\[
\begin{align*}
\text{Atoms} &= \{a, b, c, \ldots\} \\
\text{Closures} &= \{\langle p:s:e \rangle \mid p \in \text{Parameters} \land s \in \text{Sequences} \land e \in \text{Environments}\} \\
\text{Structures} &= \text{Closures} \cup \text{Atoms} \cup \{\{a, b\} \mid a \in \text{Structures} \land b \in \text{Structures}\}
\end{align*}
\]

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.

11 / 34

---

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> | ...
```

12 / 34

---

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

\[
\theta = \{v/s \mid v/s \in \sigma \land v \text{ free in sequence}\}
\]

\[
E\sigma(fn \ (\text{parameters}) \rightarrow \text{sequence end}) \rightarrow (\text{parameters : sequence} : \theta)
\]

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

What is f?

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

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

evaluation of a function application

\[
E\sigma(e) \rightarrow <v_1, \ldots : \text{seq} : \theta > \quad E\sigma(e_i) \rightarrow s_i \quad E\{v_1/s_1, \ldots \} \cup \theta(\text{seq}) \rightarrow s
\]

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

What is f.(4)?

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

def prod(l) do
  mul = fun(x,a) -> x * a end
  foldr(l, 1, mul)
end
What is `gurka/1` doing?

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

How about `tomat/1`?

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

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

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

```erlang
def tomat(l) do
  f = fn (h, a) -> [h|a] end
  foldl(l, [], f)
end
```
append all

Append all lists in a list.

def append_right(l) do
  f = fun(e,a) -> e ++ a end
  foldr(l, [], f)
end

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

an infinite list

inf = infinity(); [0|inf] = inf.(); [1|inf] = inf.()

def infinity() do
  fn() -> infinity(0) end
end

def infinity(n) ->
  [...|...]
end

patterns

In the 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) ..)

In the 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 f is less than or equal

the list of fibonacci

A function that returns an infinite list of Fibonacci numbers.

def fib() do
  fn() -> fib(1,1) end
end

def fib(f1, f2) do
  [f1 | fn() -> fib(f2, f1+f2)]
end
Let's represent a range of integers from 1 to 10 as:

```
{:range, 1, 10}
```

Elixir gives us a syntax for this:

```
1..10
```

This is not exactly how Elixir represents it but it's fine for now.

```
def sum(r) do
    reduce(r, 0, fn(x,a) -> x+a end)
end
```

Our reduce/3 should work as foldl/3 (does it make sense to have a foldr/3).

```
def reduce({:range, from, to}, acc, fun) do
    if from <= to do
        reduce({:range, from+1, to}, fun.(from, acc), fun)
    else
        acc
    end
end
```

Can we, without traversing the full range:

- take the first five elements
- find an element in the range
- return the first element and the rest of the range

We need to control the reduction.

```
def reduce({:range, from, to}, {:cont, acc}, fun) do
    if from <= to do
        reduce({:range, from+1, to}, fun.(from, acc), fun)
    else
        {:done, acc}
    end
end
def sum(r) do
    reduce(r, {:cont, 0}, fn(x,a) -> {:cont, x+a} end)
end
```
stop in the middle

def reduce(_, {:halt, acc}, _fun) do
  {:halted, acc}
end
def take(r, n) do
  reduce(r, {:cont, {:sofar, 0, []}},
    fn(x, {:sofar, s, a}) ->
      if s == n do
        {:halt, Enum.reverse(a)}
      else
        {:cont, {:sofar, s+1, [x|a]}}
      end
    end)
end

Elixir libraries

- List: operates on lists, returns a list or some value.
- Enum: takes an Enumerable as argument, returns a list or value.
- Stream: takes an Enumerable as argument, returns an Enumerable or value.

A datastructure is Enumerable if it implements the reduce protocol. Lists and ranges are Enumerable.

head and tail

: def reduce(range, {:suspend, acc}, fun) do
  {:suspended, acc, fn(cmd) -> reduce(range, cmd, fun) end}
end
def head(r) do
  reduce(r, {:cont, nil},
    fn (x, _) ->
      {:suspend, x}
    end)
end

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”?
Not really - look at this.

```erlang
f = fn() -> :ok end
g = fn() -> :ok end
case f do
  ^g -> :equal
  _ -> :no
end
```

Summary

Higher order programming:

- **closure**: a function and an environment
- **generic algorithms**: separate the recursive pattern from the data it operates over
- **continuations**: powerful technique to handle incomplete information
- **range**: representation of a range of integers
- **streams**: lazy evaluation of ranges resulting in