Introduction

Johan Montelius

KTH

VT24

introduction

Functional programming - what?

variables, literals and builtin operations

- variables: `x`, `y`, `foo`, ..
- integer: `1`, `23`, ..
- float: `1.0`, `23.45`, ..
- bool: `true`, `false`

- binary operations: `+`,`-`,`*`,`/`, ..
- arithmetic functions: `div/2`, `rem/2`, `abs/1`, ..
- comparison: `==`, `!=`, ..
- boolean operators: `and`, `or`, ..

There are more but this is fine for now.

The notation `div/2` *means a function of two arguments.*

functions

- `fn x -> x + 2 end`
- `(fn x -> x + 2 end).(5)`
- `(5 + 2)`
- `7`

- `add = fn(x,y) -> x + y end`
- `add.(5, 4)`
- `(5 + 4)`
- `9`

Parenthesis are optional ... but quite nice to have.

Can we give names to functions?

Modules

```
defmodule Test do

def to_celsius(fahren) do
   (fahren - 32) / 1.8
end

end
```


recursion

```
def fib(n) do
    :
    :
end
```


patterns

```
case <expr> do
  <pattern> -> <expr>
     :
end
```

- $x = 4 + 8$

atoms

```
def roman(number) do
  case number of
    :i -> 1
    :v -> 5
    :x -> 10
  end
end
```

atoms

Atoms are identifiers with a program wide scope.

Used to represent objects or values : `:apple`, `:orange`, `:ok`, `:error`.

The atoms `:true` and `:false` are used to represent boolan *true* and *false*.

syntax allows to omit colon for the atoms `:true`, `:false` *and* `:nil`.

modules and files

```
defmodule Foo do

def fib(n) do
   if (n == 0 or n == 1) do
      1
   else
      fib(n-1) + fib(n-2)
   end
end
```

```
defmodule Bar do

def double_fib(x) do
  2 * Foo.fib(n)
end

end
```

a file "foo.ex" holds the module Foo.

`Foo` *is an alias for an atom* `:"Elixir.Foo"`.

dynamic typing

No type declaration in code.

Elixir

```
def test(x, y) do
  if (x == 0) do
    y
  else
    x + y
  end
end
```

Java

```
public static Integer test(Integ
        if ( x.equals(0) )
            return y;
        else
            return x + y;
}
```

types are checked at run-time

dynamic typing

The development process.

Static typing:

- compile .... error, oh no
- compile .... error, oh no
- compile .... error, oh no
- compile .... yes!
- run .... no problem

Dynamic typing:

- compile .... no problem
- run .... error, oh no
- run .... error, oh no
- run .... error, oh no
- run .... yes!

tuples

A compound data structure.

```
{:orange, 23, "Goda appelsiner från Spanien"}
```

You need to keep track or the order: type, price, sales pitch.

tuples

No declarations of tuples but we have a language to describe them for documentation.

```
@type name() :: :orange | :tomato | :cucumber

@type product() :: {name(), integer(), string()}
```

more on this later

access components

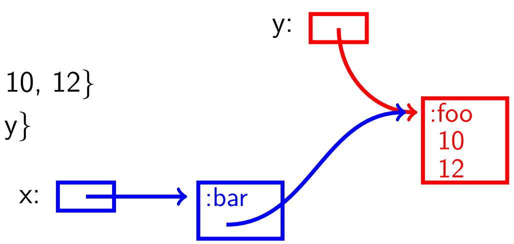
Components of a tuple are extracted using *pattern matching*.

```
x = {:gurka, 123, "Prima gurkor"}


{type, price, _} = x


price = elem(x, 1)    # zero indexed!
```

If you use `elem`*, you most likely don't know how to use pattern matching.*

under the hood

y = {:foo, 10, 12}
x = {:bar, y}



{z, {:foo, n, _}} = x

z: :bar   n: 10

linked list

In functional programming languages linked lists are very useful.

```
[]
```

```
[:one, 2, "three", 4]
```


the cons operator

Creating a new node:

```
foo = 1
rest = [2,3,4,5]
all = [ foo | rest]
```

```
[1,2,3,4,5]
```


list patterns

```
[a, b, c] = [1,2,3]
```

```
[a | rest] = [1,2,3]
```

```
[a, b | rest] = [1,2,3]
```

```
[a | rest] = [1]
```


exercise

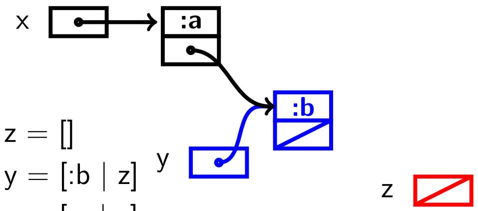
- `[h|t] = [:a,[:b,:c]]`
- `[h1,h2|t] = [:a,:b,:c]`
- `[h1,h2,t] = [:a,:b,:c]`
- `[h1,h2,t] = [:a,:b,:c,:d]`
- `[h1|[h2|t]] = [:a,:b,:c]`
- `[h|t] = [:a|:b]`

list construction

- `h = :a; t = [:b]; [h|t]`
- `h = :a; t = [[:b]]; [h|t]`
- `h = [:a,:b]; t = [:c,:d]; [h|t]`
- `h = [:a,:b]; t = [:c,:d]; [h,t]`
- `h1 = [:a,:b]; h2 = [:c,:d]; t = [:e,:f]; [h1|[h2|t]]`
- `h1 = [:a,:b]; h2 = [:c,:d]; t = [:e,:f]; [h1,[h2|t]]`
- `h = [:a,:b]; t = :c; [h|t]`

cons cells

x = [:a,:b]



z = []
y = [:b | z]
x = [:a | y]

Strings

```
text = "This is a string"
```

```
combined = text <> " that we append to this string"
```

```
answer = 42
message = "The answer is #{answer}, but what is the question?"
```


Strings and binaries

```
<<x, y, z , rest::binary>> = "absdefg"
```

Strings and lists of characters

```
msg = String.to_charlist("abcde")
```

```
[x,y,z | rest] = msg
```

```
msg = [?h, ?e, ?l, ?l, ?o]
```

try in teminal: IEx.configure(inspect: [charlists: :as_lists])

Summary

- functions : and only functions
- modules : one moule one file
- literals : integer, float, boolean, atom
- compund : tuples, lists
- strings: "hello", ?h, ?e, ?l, ?l, ?o, 'hello'