In functional programming, a program is a set of functions.

A function takes some arguments and *returns a result* ... it does not change the given arguments.

The returned value of a function is only depending on the given arguments.

Fundamentally different from *imperative programming*!

```
def foo(x) do
  y = bar(x)
  z = zot(x)
  {y, z}
end
```

```
def grk(x) do
  z = zot(x)
  y = bar(x)
  {y, z}
end
```

What is the difference between these two functions?

```
def foo(x, y) do
  res = try do
    bar(x, y)
  rescue
    error ->
      {:caught, error}
  end
  {x, y, res}
end
```
today's topic
Lists and recursion

pattern matching

- \([h|t] = [\langle a, [\langle b, c \rangle] \rangle]
- \([h_1, h_2|t] = [\langle a, b, c \rangle]
- \([h_1, h_2, t] = [\langle a, b, c \rangle]
- \([h_1, h_2, t] = [\langle a, b, c, d \rangle]
- \([h_1|\langle h_2|t \rangle] = [\langle a, b, c \rangle]
- \([h|t] = [\langle a \rangle| b \rangle]

list construction

- \(h = \langle a \rangle; t = [\langle b \rangle]; [h|t]\)
- \(h = \langle a \rangle; t = [\langle b \rangle]; [h|t]\)
- \(h = [\langle a, b \rangle]; t = [\langle c, d \rangle]; [h|t]\)
- \(h = [\langle a, b \rangle]; t = [\langle c, d \rangle]; [h|t]\)
- \(h_1 = [\langle a, b \rangle]; h_2 = [\langle c, d \rangle]; t = [\langle e, f \rangle]; [h_1|h_2|t]\]
- \(h_1 = [\langle a, b \rangle]; h_2 = [\langle c, d \rangle]; t = [\langle e, f \rangle]; [h_1, h_2|t]\]
- \(h = [\langle a, b \rangle]; t = c; [h|t]\)

cons cells

- \(x = [\langle a, b \rangle]\)
- \(z = []\)
- \(y = [\langle b, z \rangle]\)
- \(x = [\langle a, y \rangle]\)
def append([], y) do y end

def append([h|t], y) do
    z = append(t, y)
    [h | z]
end

a = [1,2]; b = [3,4]; c = append(a, b)

What is the asymptotic time complexity of append/2.

The infix operator '++' is append!
x ++ y is not a constant time operation!

Is [x|y] a constant time operation?

<table>
<thead>
<tr>
<th>length of X</th>
<th>run-time in ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>50</td>
</tr>
<tr>
<td>8000</td>
<td>78</td>
</tr>
<tr>
<td>10000</td>
<td>75</td>
</tr>
<tr>
<td>12000</td>
<td>99</td>
</tr>
<tr>
<td>14000</td>
<td>102</td>
</tr>
<tr>
<td>16000</td>
<td>110</td>
</tr>
<tr>
<td>18000</td>
<td>122</td>
</tr>
<tr>
<td>20000</td>
<td>150</td>
</tr>
</tbody>
</table>

How long time does it take to append a list of 40.000 elements?
A *multiset* (or bag) is a set possibly with duplicated elements.

Define a function that returns the union of two multisets.

```elixir
def union([], y) do y end

def union([h|t], y) do
  z = union(t, y)
  [h|z]
end
```

```elixir
def tailr([], y) do y end

def tailr([h|t], y) do
  z = [h|y]
  tailr(t, z)
end
```

Is there a difference?

```elixir
union([:a,:b], [:c])

z = union([:b], [:c])
z' = union([], [:c])

[:b|z']
[:a|z]
[:a,:b,:c]
```

```elixir
tailr([:a,:b], [:c])
tailr([:b], [:a,:c])
tailr([], [b,:a,:c])

[:b,:a,:c]
```

**tail recursion optimization**

When the last expression in a sequence is a function call, the stack frame of the caller can be reused.

We call these functions *tail recursive*.

Possibly more efficient code.

Probably more complicated.

Very important when we will define processes!
def odd([ ]) do ... end
def odd([h|t]) do
  if rem(h,2) == 1 do
    ...
  else
    ...
  end
end
def odd_n_even(l) do
  odd = odd(l),
  even = even(l),
  {odd, even}
end

We're building a tuple that is not needed, its only purpose is to return the two lists.

def odd_n_even(l) do
  odd_n_even(l, [], [])
end
def odd_n_even([], odd, even) do
  ...
end
odd_n_even([h|t], odd, even) do
  if rem(h,2) == 1 do
    odd_n_even(t, ..., ...)
  else
    odd_n_even(t, ..., ...)
  end
end
def sum(l) -> sum(l, ...) end
def sum([], s) do ... end
def sum([n|t], s) do sum(t, ...) end
A function that reverses a list:
\[
\text{rev}([1,2,3,4]) \rightarrow [4,3,2,1]
\]

```elixir
def rev([]) do [] end
def rev([h|t]) do
  rev(t) ++ [h]
end
def rev(l) do rev(l, []) end
def rev([], res) do res end
def rev([h|t], res) do
  rev(t, [h|res])
end
```

A function that flattens a list of list:
\[
\text{flatten}([[1,2],[3,4]]) \rightarrow [1,2,3,4]
\]

```elixir
def flat([]) do [] end
def flat([h|t]) do
  h ++ flat(t)
end
def flat(l) do flat(l, []) end
def flat([], res) do res end
def flat([h|t], res) do
  flat(t, res ++ h)
end
```

Summary

- Pattern matching of lists - learn it by heart
- cons - is a constant time operation
- append - is a $O(n)$ function
- tail recursion - a technique to master
- think about complexity