Lists and recursion

Johan Montelius

KTH

VT23

first a recap

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*!

any difference?

```
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?

catch this

```
def foo(x, y) do
  try do
    {:ok, bar(x, y)}
  rescue
    error ->
      {:error, error}
  end
end
```

```
def test(x, y) do
  case foo(x, y) do
   {:error, msg} ->
      # state of x and y?
        :
   {:ok, res} ->
        :
  end
end
```

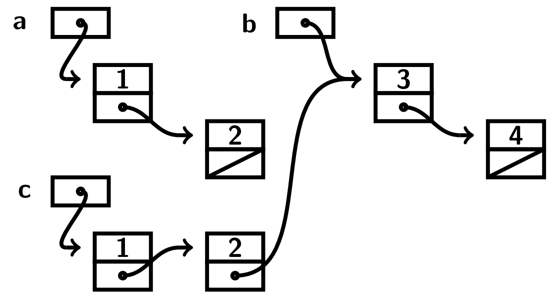
today's topic

Recursion over lists

append

```
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)
```



append

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

What is the *asymptotic time complexity* of append/2.

append

length of X	run-time in ms
4000	50
8000	78
10000	75
12000	99
14000	102
16000	110
18000	122
20000	150

How long time does it take to append a list of 40.000 elements?

warning

The infix operator '++' is append!

x ++ y is not a constant time operation!

Is [x|y] a constant time operation?

union of multisets

A *multiset* (or bag) is a set possibly with duplicated elements.

Define a function that returns the union of two multisets.

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

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

Is there a difference?

evaluation union vs tailr

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

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

tail recursive?

```
def sum([]) do 0 end
def sum([n|t]) do
  n + sum(t)
end
```

```
def sum([]) do 0 end
def sum([n|t]) do
  s = sum(t)
  n + s
end
```


accumulators

```
def odd([]) do  ... end
def odd([h|t]) do
   if rem(h,2) == 1 do
       ...
   else
       ...
   end
end
```

```
def split(l) do
   odd = odd(l)
   even = even(l)
   {odd, even}
end
```


accumulators

```
do odd_n_even([]) do
   {..., ...}
end
```

```
def odd_n_even([h|t]) do
  {odd, even} = odd_n_even(t)
  if rem(h,2) == 1 do
    ...
  else
     ...
  end
end
```

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

accumulators

```
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
```

tail recursive?

```
def sum(l) -> sum(l, ...) end

def sum([], s) do ... end
def sum([n|t], s) do sum(t, ...) end
```


n-reverse

A function that reverses a list:

```
rev([1,2,3,4]) -> [4,3,2,1]
```

```
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
```


n-flatten

A function that flattens a list of list:

```
flatten([[1,2],[3,4]]) -> [1,2,3,4]
```

```
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

- recursion over lists is very common
- tail recursion - a technique to master
- think about complexity
- cons - is a constant time operation
- append - is a $O(n)$ function