dp23a 1. FP-előadás, 2023-09-05

Két lista rekurzív összefűzése (app)

A bevezető DP-előadáson a deklaratív stílus érzékeltetésére két lista összefűzését mutattuk be egy Elixir-függvényel és egy Prolog-eljárással. Most, az első FP-előadáson az app/2 Elixir-függvény kapcsán fogok sok részletet elmondani, bemutatni a Livebook használatáról, az Elixir nyelvről, a rekurzióról, a rekurzió hatékonyságáról.

```
# app(l1, l2): l1 és l2 listák összefűzöttje (l1⊕l2)
# [] ⊕ b = b
def app([], b) do
 b
end

# [x|a] ⊕ b = [x|a⊕b]
def app([x | a], b) do
 [x | app(a, b)]
end
```

```
defmodule App do
 # app(xs, ys): xs és ys listák összefűzöttje zs == (xs⊕ys)
 # [] ⊕ ys == ys
 def app([], ys) do
  ys
 end

 # [x|xs] ⊕ ys = [x|xs⊕ys]
 def app([x | xs], ys) do
  [x | app(xs, ys)]
 end
end
```

```
defmodule App2 do
 @spec app(xs :: [integer()], ys :: [integer()]) :: zs :: [integer()]
 # xs és ys listák összefűzöttje zs == (xs⊕ys)
 # [] ⊕ ys == ys
 def app([], ys) do
  ys
 end

 # [x|xs] ⊕ ys = [x|xs⊕ys]
 def app([x | xs], ys) do
  [x | app(xs, ys)]
 end
end
```

```elixir
defmodule App3 do
 @spec app(xs :: [integer()], ys :: [integer()]) :: zs :: [integer()]
 # xs és ys listák összefűzöttje zs == (xs⊕ys)
 # [] ⊕ ys == ys
 def app([], ys) do
  ys
 end

 # [x|xs] ⊕ ys = [x|xs⊕ys]
 def app([x | xs], ys) do
  [x | app(xs, ys)]
 end
end

App3.app([], [])

App3.app([], [])
App3.app([], [1, 2, 3])

IO.inspect(App3.app([], []))
IO.inspect(App3.app([], [1, 2, 3]))

IO.inspect(App3.app([], []))
IO.inspect(App3.app([5, 6, 7], [1, 2, 3]))

IO.inspect(App3.app([], []))
IO.inspect(App3.app([5, 6, 7], [1, 2, 3]))
IO.inspect(App3.app([7, 10, 12], [97, 98, 99]))

IO.inspect(App3.app([], []))
IO.inspect(App3.app([5, 6, 7], [1, 2, 3]))
IO.inspect(App3.app([7, 10, 12], [97, 98, 99]), charlists: :as_lists)

App3.app([7, 10, 12], [97, 98, 99]) |> IO.inspect(charlists: :as_lists)

App3.app([7, 10, 12], [97, 98, 99]) |> inspect(charlists: :as_lists) |> IO.puts()

App3.app([7, 10, 12], [97, 98, 99]) |> inspect(charlists: :as_charlists) |> IO.puts()

1.0 === 1

~c"\a\n\fabc" == ~c"\a\n\fabc"

1..13_200_000 |> Range.to_list() |> App3.app([]) |> length() |> IO.puts()

1..13_200_000 |> Range.to_list() |> Enum.reverse() |> length() |> IO.puts()

((1..13_200_000 |> Range.to_list()) ++ []) |> length() |> IO.puts()

for(i <- 1..13_200, do: i) |> App3.app([]) |> length() |> IO.puts()
```

^^^^^ for-komprehenzió, ~-jelölés: nagyon hasznos! Még találkozunk vele.

Faktoriális (jobbrekurzív is)

Ugyancsak a bevezető DP-előadáson szerepelt az $n!$ kiszámítása C nyelven, a rekurzív fact függvénnyel. „Igen, de ez lasssúúú! ⇝ Ún. jobbrekurzív (farokrekurzív, tail recursive) változata a ciklussal azonos hatékonyságú kóddá fordul." - olvashatják az előadásdián. Most megírjuk a függvényt Elixir nyelven.

Az első változat a matematikai definíciót másoló, rekurzív függvény. Az ilyen rekurziót angolul *body recursion*-nek, magyarul talán törzsrekurziónak szokták nevezni, ha hangsúlyozni akarják, hogy a rekurzív hívás eredményével még el kell végezni valamilyen műveletet a függvény törzsében.

A második, jobbrekurzív, angolul: *tail recursive* változat kevésbé szigorúan követi a matematikai definíciót, emiatt nehezebb is megérteni, nehezebb hozzá kifejező, pontos fejkommentet írni. A jobbrekurziót magyarul szokták még terminális (ritkábban farok-) rekurziónak is nevezni, mert a rekurzív hívás az adott függvénytörzsben az utolsó – befejező, lezáró – hívás, az eredményét egy-az-egyben vissza kell adni, már semmiféle műveletet nem végzünk vele.

fac(100_000)

```
defmodule Fac do
 # Típusspecifikáció
 @spec fac(n :: integer()) :: f :: integer()
 # f = n! (azaz f az n faktoriálisa) # Fejkomment
 # ha az n=0 mintaillesztés sikeres
 def fac(0), do: 1
 # ha az n=0 mintaillesztés sikertelen
 def fac(n), do: n * fac(n - 1)
end

Fac.fac(5) |> IO.puts()
Fac.fac(0) |> IO.puts()
Fac.fac(1) |> IO.puts()
Fac.fac(100_000) |> IO.puts()

defmodule Fac1 do
 # Típusspecifikáció
 @spec fac(n :: integer()) :: f :: integer()
 # f = n! (azaz f az n faktoriálisa)
 def fac(n), do: fac(n, 1)

 @spec fac(n :: integer(), a :: integer()) :: f :: integer()
 defp fac(0, a), do: a
 defp fac(n, a), do: fac(n - 1, n * a)
end

Fac1.fac(5) |> IO.puts()
```

```
Fac1.fac(0) |> IO.puts()
Fac1.fac(1) |> IO.puts()
# Fac1.fac(45000) |> IO.puts()
Fac1.fac(100_000)
```

Tapasztalt-e különbséget a törzsrekurzív és a jobbrekurzív függvény futási ideje között?

Két lista jobbrekurzív összefűzése

Most a listák összefűzésére jobbrekurzív függvényt írunk. Ha az eredeti sorrendet meg akarjuk tartani, akkor az első listát az összefűzés előtt meg kellene fordítanunk (miért is? magyarázza el!). Most megelégszünk azzal a verzióval, amelyik az első lista megfordítottját fűzi a második lista elé: ezt csinálja a revapp/2 függvény.

```
defmodule App4 do
 @spec revapp(xs :: [integer()], ys :: [integer()]) :: zs :: [integer()]
 # megfordított xs és ys listák összefűzöttje zs == (rev(xs)⊕ys)
 # [] ⊕ ys == ys
 def revapp([], ys) do
  ys
 end

 # [x|xs] ⊕ ys = xs⊕[x|ys]
 def revapp([x | xs], ys) do
  revapp(xs, [x | ys])
 end
end

App4.revapp([], []) |> length()
1..13_600_000 |> Range.to_list() |> App4.revapp([]) |> length()
```

A tanulság:

Nowadays compilers and virtual machines are too smart to predict their behavior.

This is to ensure that no system resources, for example, call stack, are consumed.

When the call is not tail-recursive, the stack must be preserved across calls. Consider the following example.

```
 defmodule NTC do
  def inf do
   inf()
   IO.puts(".")
   DateTime.utc_now()
```

```
  end
 end
```

Here we need to preserve stack to make it possible to continue execution of the caller when the recursion would return. It won't because this recursion is infinite. The compiler is unable to optimize it and here is what we get:

```
NTC.inf
[1]    351729 killed     iex
```

Please note, that no output has happened, which means that the function was recursively calling itself until the stack blew up.

Please not too, that with Tail-Call Optimization, infinite recursion is possible – and it is used widely in message handling.

Source: https://stackoverflow.com/questions/61626928/elixir-why-tail-recursive-use-more-memory-than-a-body-recursive-function

See also: https://www.erlang.org/doc/efficiency_guide/myths.html, 2.1 Myth: Tail-Recursive Functions are Much Faster Than Recursive Functions

```
defmodule NTC do
 def inf do
  # inf()
  IO.puts(".")
  DateTime.utc_now()
 end
end

NTC.inf()
```

Az inf/0 függvény rekurzívan hívja saját magát, méghozzá úgy, hogy előtte semmilyen más műveletet nem végez: az ilyen rekurzív hívást balrekurzívnak nevezzük. Ez a rekurzió ráadásul végtelen rekurzió, nincs leállási feltétele. Ha futtatni akarja az Elixir-cella tartalmát, törölje a kommentjelet (#), majd pár másodperc futás után kattintson a megjelenő *Stop* gombra.

A végtelen **jobb**rekurziónak (de nem a balrekurziónak) pl. olyan esetekben van jogosultsága, amikor két vagy több processz üzenetet küld egymásnak, és végtelen rekurzív hívásban várnak a válaszüzetenetre.

Egyszerű műveletek listákon

Zárásul nézzünk néhány egyszerű műveletet listákon.

```
xs = [10, 20, 30]
x = hd(xs)
rs = tl(xs)
{zs, xs} = {xs, [5, 6]}
xs = [5, 7]
```

```
IO.inspect(tl(xs), charlists: :as_lists)
xs = [7, 8, 9]
hd(tl(xs))
```

```
for i <- 7..13, do: [i]
```

```
for i <- 7..13, do: [i] |> inspect(charlists: :as_lists)
```

```
for(i <- 7..13, do: [i]) |> inspect(charlists: :as_lists)
```

```
for i <- 7..13, do: [i] |> IO.inspect(charlists: :as_lists)
```

```
for(i <- 7..13, do: [i]) |> IO.inspect(charlists: :as_lists)
```

```
for(i <- 30..36, do: [i]) |> IO.inspect(charlists: :as_lists)
```

```
for(i <- 32..95, do: [i]) |> inspect(charlists: :as_lists)
```

```
for(i <- 32..95, do: [i]) |> inspect(charlists: :as_lists, limit: :infinity)
```

```
for(i <- 32..95, do: [i]) |> inspect(limit: :infinity)
```