Tartalom

DP25a - 4. FP gyakorlat, 2025-10-14

Listafeldolgozás Enum.chunk_while/4-gyel

A legutóbbi előadáson használtuk az Enum.chunk_every/4 és az Enum.chunk_by/2 függvényeket, és megemlítettük az Enum.chunk_while/4 függvényt is. A következő feladatokban az utóbbi használatát kérjük.

L1. Platók

Az előadáson az Enum.chunk_by/2-vel oldottuk meg az alábbi feladatot.

```
defmodule Plains do
 @spec plains(xs::[integer()]) :: ps::[integer()]
 # az xs lista azonos értékű, maximális hosszúságú,
 # legalább kételemű sorozatainak, azaz platóinak listája ps
 def plains(xs) do
  plains = Enum.chunk\_by(xs, &(&1))
  for ps = [_,_|_] <- plains, do: ps
 end
end
```

Oldja meg ugyanezt Enum.chunk_while/4-gyel. Írjon hatékony, a paraméterként átadott függvényekben is csak mintaillesztést alkalmazó megoldást.

```
defmodule Plains do
 @spec plains(xs::[integer()]) :: ps::[integer()]
 # az xs lista azonos értékű, maximális hosszúságú,
 # legalább kételemű sorozatainak, azaz platóinak listája ps
 def plains(xs) do
  ...
 end
end
```

```
(Plains.plains([9,0,0,0,6,7,6,5,5,2,2,2,2,1]) == [[0, 0, 0], [5, 5], [2, 2, 2, 2]])|> IO.inspect()
```

L2. Kisebbek

Az előadáson az Enum.zip-pel oldottuk meg az alábbi egyszerű feladatot.

```
defmodule Smallers do
 @spec smallers(xs::[integer()]) :: ss::[integer()]
 # ss azoknak a xs-beli egészeknek a listája, melyek a
 # közvetlenül előttük álló listaelemnél kisebbek
 def smallers([_|xs]=xxs) do
  for {p, c} <- Enum.zip(xxs ,xs), p > c, do: c
 end
 def smallers([]), do: []
end
```

Oldja meg ugyanezt Enum.chunk_while/4-gyel. Írjon hatékony, a paraméterként átadott függvényekben is csak mintaillesztést alkalmazó megoldást.

```
defmodule Smallers do
 @spec smallers(xs::[integer()]) :: ss::[integer()]
 # ss azoknak a xs-beli egészeknek a listája, melyek a
 # közvetlenül előttük álló listaelemnél kisebbek
 def smallers(xs) do
  ...
 end
end

(Smallers.smallers([9,8,8,8,6,7,6,5,6,7,2,1]) == [8, 6, 6, 5, 2, 1]) |> IO.inspect()
```

L3. Lejtők, emelkedők, platók hossza

Az előadáson szó volt az alábbi feladatról, de nem oldottuk meg.

```
defmodule Slopes do
 @type cmp() :: integer(), integer() :: boolean()
 @type spec slope_lengths(xs::[integer()], cmp::cmp()) :: ls::[integer()]
 # az xs listából a cmp alapján kigyűjtött, maximális hosszúságú, legalább
 # kételemű sorozatok – lejtők, emelkedők vagy platók – hosszának listája ls
 def slope_lengths(xs, cmp) do
  ...
 end
end

(Slopes.slope_lengths([9,0,0,0,6,7,6,5,5,2,2,2,2,1], &>/2) == [2, 3, 2, 2])
 |> IO.inspect(label: "lejtők")
(Slopes.slope_lengths([9,0,0,0,6,7,6,5,5,2,2,2,2,1], &==/2) == [3, 2, 4])
 |> IO.inspect(label: "platók")
(Slopes.slope_lengths([9,0,0,0,6,7,6,5,5,2,2,2,2,1], &</2) == [3])
 |> IO.inspect(label: "emelkedők")
```

Oldja meg a feladatot Enum.chunk_while/4-gyel.

Binárisok kezelése

B1. Változó hosszúságú bájtsorozat dekódolása egész számmá

A tetszőleges hosszúságú nemnegatív egész számok bájtok formájában történő leírásának egyik módja, hogy minden bájt első bitjével jelezzük, folytatódik-e a bájtsorozat, a maradék 7 bitben pedig a tényleges szám egy 7 bites szegmensét tároljuk. A bájt kezdő 0-s bitje jelzi, hogy vége van a sorozatnak, az ezt követő 7 bit pedig a tárolt szám utolsó 7 bitje; a kezdő 1-es bit pedig azt jelzi, hogy a következő 7 bit a szám következő legmagasabb helyiértékű 7 bitje, és a szám alacsonyabb helyiértékű bitjei a következő bájtban vannak. Néhány példa bites reprezentációja:

00000001 -> 0b0000001 = 1 (a kezdő 0 bit jelzi, hogy a szám 1 bájt hosszú, értéke 0000001, vagyis 1)
00000010 -> 0b0000010 = 2 (szintén 1 bájtos szám, értéke 0000010, vagyis 2)
100000010000001 -> 0b00000010000001 = 129 (az első 7 bit 0000001, a kezdő 1-es bit jelzi,
hogy folytatódik, a második 7 bit 0000001)

A bit shift az Elixirben nem beépített operátor, az import Bitwise paranccsal lehet aktiválni a megfelelő <<< és >>> operátorokat. (A Bitwise modul betöltése eltart egy ideig az első futtatáskor.)

Érdemes segédfüggvényt használni a részeredmények tárolására külön paraméterben. A bit shift operátorok precedenciája alacsony, figyeljen a zárójelezésre!

A binárisok szintaxisáról és használatáról bővebb leírás érhető el a https://elixir-lang.org/getting-started/binaries-strings-and-char-lists.html oldalon.

```
defmodule Varlen do
 import Bitwise
 @spec decode(bin::binary()) :: int::integer()
 # bin decimális értéke int
 def decode(bin) do
  ...
 end
```

```
end

IO.puts(Varlen.decode(<<0x01>>) === 1)
IO.puts(Varlen.decode(<<0x7F>>) === 127)
IO.puts(Varlen.decode(<<0xFF, 0x7F>>) === 16383)
IO.puts(Varlen.decode(<<0x81, 0x80, 0x01>>) === 16385)
```

Elágazó rekurzió

Elágazó rekurzióról akkor beszélünk, ha egy függvény *ugyanabban a klózban legalább kétszer* hívja meg *saját magát.*

Elágazóan rekurzív adatstruktúrák, pl. bináris fák bejárásának, feldolgozásának nyilvánvalóan az a természetes módja, ha elágazóan rekurzív algoritmusokat írunk rájuk, de vannak olyan számítási feladatok is, amelyekre sokkal könnyebb és érthetőbb elágazóan rekurzív algoritmust készíteni. Az utóbiak között vannak olyanok, amelyeket egy kis fejtörés után érdemes sokkal hatékonyabban, azaz lineárisan rekurzív, akkumulátoros segédfüggvény segítségével vagy más módszerrel, pl. dinamikus programozással megoldani, és vannak olyanok, amelyekre ugyan lehetne lineárisan rekurzív algortimust írni, de nem érdemes, mert az adatszerkezet jellege miatt nem lesz hatékonyabb.

Elágazó rekurzióra már több példát láttunk az előadásokon – Fibonacci-számok, kombináció, permutáció, gyorsrendezés –, a ház feladatok között is volt ilyen jellegű, a legutóbbi előadáson pedig bináris és többágú fákról és ezeket kezelő egyszerű algortimusokról is volt szó. Most ilyen, fákon műveleteket végző gyakorló feladatok következnek.

T1. Bináris fák feldolgozása elágazó rekurzióval

A következő feladatokhoz az alábbi tree és inttree adattípusokat definiáljuk:

```
@type tree() :: :leaf | {any(), tree(), tree()}
@type inttree() :: :leaf | {integer(), inttree(), inttree()}
```

Tehát egy tree() típusú Elixir-kifejezés

- vagy egy adatot (a továbbiakban *címkének* nevezzük) tartalmazó *csomópont*, amely további két tree() típusú értéket tartalmaz: az első a bal, a második a jobb részfa;
- vagy egy címke nélküli :leaf (azaz *levél)* atom.

Egy inttree() olyan tree(), amelynek minden címkéje egész szám. A példákban felhasznált változók és értékük:

```
t1 = {4,
      {3,:leaf,:leaf},
      {6,
        {5,:leaf,:leaf},
        {7,:leaf,:leaf}
      }
    }
t2 = {:a,
       {:b, {:v,:leaf,:leaf}, :leaf},
       {:c,
          :leaf,
          {:d,
             {:w, {:x,:leaf,:leaf}, :leaf},
             {:f, {:x,:leaf,:leaf}, {:y,:leaf,:leaf}}
          }
       }
    }
```

A típusokat és a két változó helyett a két függvényt definiáljuk a T modulban, hogy más modulokból hivatkozhassunk rájuk:

```
defmodule T do
 @type tree() :: :leaf | {any(), tree(), tree()}
 @type inttree() :: :leaf | {integer(), inttree(), inttree()}
 def t1() do
   {4, {3, :leaf, :leaf}, {6, {5, :leaf, :leaf}, {7, :leaf, :leaf}}}
 end

 def t2() do
   {:a, {:b, {:v, :leaf, :leaf}, :leaf},
    {:c, :leaf,
```

```
    {:d, {:w, {:x, :leaf, :leaf}, :leaf}, {:f, {:x, :leaf, :leaf}, {:y, :leaf, :leaf}}}}}
  end
end

IO.puts("t1 = " <> inspect(T.t1()))
IO.puts("t2 = " <> inspect(T.t2()))

defmodule T do
  @type tree() :: :leaf | {any(), tree(), tree()}
  @type inttree() :: :leaf | {integer(), inttree(), inttree()}
  def t1() do
    {4, {3, :leaf, :leaf}, {6, {5, :leaf, :leaf}, {7, :leaf, :leaf}}}
  end

  def t2() do
    {:a, {:b, {:v, :leaf, :leaf}, :leaf},
     {:c, :leaf,
      {:d, {:w, {:x, :leaf, :leaf}, :leaf}, {:f, {:x, :leaf, :leaf}, {:y, :leaf, :leaf}}}}}
  end
end

IO.puts("t1 = " <> inspect(T.t1()))
IO.puts("t2 = " <> inspect(T.t2()))
```

```
t1 = {4, {3, :leaf, :leaf}, {6, {5, :leaf, :leaf}, {7, :leaf, :leaf}}}
t2 = {:a, {:b, {:v, :leaf, :leaf}, :leaf}, {:c, :leaf, {:d, {:w, {:x, :leaf, :leaf}, :leaf}, {:f, {:x, :leaf, :leaf}, {:y, :leaf, :leaf}}}}}

:ok
```

T2. Bináris egészfa minden címkéjének megnövelése 1-gyel

```
defmodule IncTree do
  @spec fa_noveltje(f0::T.inttree()) :: f::T.inttree()
  # Az f fa minden címkéje eggyel nagyobb az f0 fa azonos helyen lévő címkéjénél
  def fa_noveltje(f0) do
  ...
  end
end
IncTree.fa_noveltje(T.t1()) === {5,{4,:leaf,:leaf},{7,{6,:leaf,:leaf},{8,:leaf,:leaf}}}
```

T3. Bináris fa tükörképe

```
defmodule Mirtree do
  @spec fa_tukorkepe(f0::T.tree()) :: f::T.tree()
  # f az f0 fa tükörképe
  def fa_tukorkepe(f0) do
  ...
  end
end
Mirtree.fa_tukorkepe(T.t1()) === {4,{6,{7,:leaf,:leaf},{5,:leaf,:leaf}},{3,:level,:level}}
```

T4. Bináris fa inorder, preorder és postorder bejárása

```
defmodule TravTree do
  @spec inorder(f::T.tree()) :: ls::[any()]
  # ls az f fa elemeinek a fa inorder bejárásával létrejövő listája
  def inorder(f) do
  ...
  end
  @spec preorder(f::T.tree()) :: ls::[any()]
  # ls az f fa elemeinek a fa preorder bejárásával létrejövő listája
  def preorder(f) do
  ...
  end
  @spec postorder(f::T.tree()) :: ls::[any()]
  # ls az f fa elemeinek a fa postorder bejárásával létrejövő listája
```

```
  def postorder(f) do
  ...
  end
end
(TravTree.inorder(T.t1())  === [3,4,5,6,7]) |> IO.inspect()
(TravTree.preorder(T.t1())  === [4,3,6,5,7]) |> IO.inspect()
(TravTree.postorder(T.t1()) === [3,5,7,6,4]) |> IO.inspect()
```

T5. Címke előfordulása (rendezetlen) bináris fában

```
defmodule Contains do
  @spec tartalmaz(f::T.tree(), c::any()) :: b::boolean()
  # b igaz, ha c az f fa valamely címkéje
  def tartalmaz(f0) do
  ...
  end
end
(Contains.tartalmaz(T.t1(), :x) === false) |> IO.inspect()
(Contains.tartalmaz(T.t2(), :x) === true) |> IO.inspect()
```

T6. Címke összes előfordulásának száma bináris fában

```
defmodule Occurs do
  @spec elofordul(f::T.tree, c::any()) :: n::integer()
  # A c címke az f fában n-szer fordul elő
  def elofordul(f0) do
  ...
  end
end
(Occurs.elofordul(T.t1(), :x) === 0) |> IO.inspect()
(Occurs.elofordul(T.t2(), :x) === 2) |> IO.inspect()
```

T7. Címkék felsorolása írjon hatékony, lineáris időigényű algoritmust! Ehhez segédfüggvény használatát javasoljuk.

```
defmodule Labels do
  @spec cimkek(f::T.tree()) :: ls::[any()]
  # ls az f fa címkéinek listája inorder sorrendben
  def cimkek(f) do
  ...
  end

  @spec cimkek(f::T.tree(), zs::[any()]) :: ls::[any()]
  # ls az f fa címkéinek listája inorder sorrendben a zs elé fűzve
  defp cimkek(f, zs) do
  ...
  end
end
(Labels.cimkek(T.t1()) === [3,4,5,6,7]) |> IO.inspect()
(Labels.cimkek(T.t2()) === [:v,:b,:a,:c,:x,:w,:d,:x,:f,:y]) |> IO.inspect()
```