

Deklaratív programozás 5. előadás

Kabódi László¹ Marussy Kristóf²
BME Számítástudományi és Információelméleti Tanszék
Mesterséges Intelligencia és Rendszertervezés Tanszék

¹`kabodi.laszlo@vik.bme.hu`

²`marussy@mit.bme.hu`

2026. ősz

I. rész

Feltételes kifejezés

- 1 Feltételes kifejezés
- 2 For-jelöléssel, egyszerűen
- 3 Lineáris és elágazó rekurzió

Tartalom

- 1 Feltételes kifejezés
 - FPE-5 – Feltételes kifejezés

Feltételes kifejezés

- A funkcionális nyelvű programok sok kis függvényből állnak, a meghívásuk sorrendjét, a klózok kiválasztását mintaillesztéssel és esetleg örökkel oldjuk meg
- Néha mégis szükség van olyan vezérlési szerkezetekre, amelyekhez hasonlóak gyakoriak az imperatív nyelvekben: *feltételes kifejezésekre*
- A funkcionális nyelvek vezérlési szerkezetei tehát, nem meglepő módon, maguk is *kifejezések*: van értékük, *teljes jogú polgárok*

Feltételes kifejezés

- A funkcionális nyelvű programok sok kis függvényből állnak, a meghívásuk sorrendjét, a klózok kiválasztását mintaillesztéssel és esetleg örökkel oldjuk meg
- Néha mégis szükség van olyan vezérlési szerkezetekre, amelyekhez hasonlók gyakoriak az imperatív nyelvekben: *feltételes kifejezésekre*
- A funkcionális nyelvek vezérlési szerkezetei tehát, nem meglepő módon, maguk is *kifejezések*: van értékük, *teljes jogú polgárok*
- Abból, hogy **a vezérlési szerkezet kifejezés**, következik, hogy (a kivételdobástól eltekintve) bárhogyan is érjen véget a kiértékelése, **valamilyen értéket mindenképpen eredményül kell adnia**
- Az Elixirben a feltételes kifejezést az *if*, *unless*, *cond* és *case* kulcsszavak egyike vezeti be – ezekről lesz szó a következőkben

Feltételes kifejezés

- A funkcionális nyelvű programok sok kis függvényből állnak, a meghívásuk sorrendjét, a klózok kiválasztását mintaillesztéssel és esetleg örökkel oldjuk meg
- Néha mégis szükség van olyan vezérlési szerkezetekre, amelyekhez hasonlók gyakoriak az imperatív nyelvekben: *feltételes kifejezésekre*
- A funkcionális nyelvek vezérlési szerkezetei tehát, nem meglepő módon, maguk is *kifejezések*: van értékük, *teljes jogú polgárok*
- Abból, hogy **a vezérlési szerkezet kifejezés**, következik, hogy (a kivételdobástól eltekintve) bárhogyan is érjen véget a kiértékelése, **valamilyen értéket mindenképpen eredményül kell adnia**
- Az Elixirben a feltételes kifejezést az *if*, *unless*, *cond* és *case* kulcsszavak egyike vezeti be – ezekről lesz szó a következőkben
- Lehetőleg ne vagy csak ritkán, jól megfontoltan használjunk feltételes kifejezést mintaillesztés és örök helyett: a feltételes kifejezéstől a függvény hosszabb, olvashatatlanabb lesz – szükségtelenül
- Ökölszabályként jegyezzük meg, hogy egy 15-20 sornál hosszabb klóz valószínűleg arra utal, valamit nem elég jól kódoltunk

if és unless 1

- Két *paraméterük* van: egy feltétel és egy kulcs-érték lista, amelyben két kulcs van: `do:` és `else:`
- Ha a feltétel igaz, az `if` a `do:`-hoz tartozó értéket, ha hamis, akkor az `else:`-hez tartozót adja eredményül; az `unless` pont fordítva csinálja
- Az `else:` ág, azaz az `{:else, érték}` pár elhagyható, ilyenkor az eredmény `nil`, ha a feltétel `if` esetén hamis, `unless` esetén igaz
- `do: ..., else: ...` helyett `do ... else ... end` is írható

if és unless 1

- Két *paraméterük* van: egy feltétel és egy kulcs-érték lista, amelyben két kulcs van: `do:` és `else:`
- Ha a feltétel igaz, az `if` a `do:`-hoz tartozó értéket, ha hamis, akkor az `else:`-hez tartozót adja eredményül; az `unless` pont fordítva csinálja
- Az `else:` ág, azaz az `{:else, érték}` pár elhagyható, ilyenkor az eredmény `nil`, ha a feltétel `if` esetén hamis, `unless` esetén igaz
- `do: ..., else: ...` helyett `do ... else ... end` is írható
- Példák:

```
iex> if(true && 1, [{:do, "hm"}, {:else, "mh"}]) #függvényszerű
"hm"
iex> if false, do: "hm", else: "mh" #de van egyszerűbb írásmódja is
"mh"
iex> if false || 1, do: "hm" #az else: ág elhagyható
"hm"
iex> if false, do: "hm" #ekkor nil az eredmény, ha a feltétel false
nil
iex> unless false, do: "hm", else: "mh" #unless fordítva csinálja
"hm"
```

if és unless 2

- Az `if` és az `unless` nem nyelvi elem, hanem *makró*
- A makrót fordításkor fejti ki az Elixir. További gyakran használt makrók: `def`, `defp`, `defmodule`, `with`.
- Az `if`-ben és az `unless`-ben a feltétel igazságértékként (*truthy*, *falsy*) értelmezhető, egyébként tetszőleges kifejezés lehet

if és unless 2

- Az if és az unless nem nyelvi elem, hanem *makró*
- A makró fordításkor fejti ki az Elixir. További gyakran használt makrók: `def`, `defp`, `defmodule`, `with`.
- Az if-ben és az unless-ben a feltétel igazságértékként (*truthy*, *falsey*) értelmezhető, egyébként tetszőleges kifejezés lehet
- Az if-ben és az unless-ben definiált változók *lokálisak*
- Példa:

```
iex> x = 1
1
iex> if true, do: (x = x + 1; x)
2
iex> x
1
```

if és unless 2

- Az if és az unless nem nyelvi elem, hanem *makró*
- A makró fordításkor fejti ki az Elixir. További gyakran használt makrók: `def`, `defp`, `defmodule`, `with`.
- Az if-ben és az unless-ben a feltétel igazságértékként (*truthy*, *falsey*) értelmezhető, egyébként tetszőleges kifejezés lehet
- Az if-ben és az unless-ben definiált változók *lokálisak*
- Példa:

```
iex> x = 1
```

```
1
```

```
iex> if true, do: (x = x + 1; x)
```

```
2
```

```
iex> x
```

```
1
```

- Ezért ha értéket akarunk kötni egy változóhoz, akkor ki kell használnunk, hogy if és unless kifejezések, van értékük:

```
iex> x = 1
```

```
1
```

```
iex> x = unless true, do: x + 1, else: x + 2
```

```
3
```

cond 1

- A `cond` feltétel-érték párok sorozatát dolgozza fel
- A `cond`-ban – mint `if`-ben és az `unless`-ben – igazságérték-eredményű, egyébként tetszőleges függvények hívhatók meg a feltételben
- Mint minden kifejezés kiértékelése, a feltételes kifejezésé is balról jobbra, fölülről lefelé halad
- Az **első** teljesülő feltételhez tartozó kifejezés értéke lesz az eredmény
- Ha egyik feltétel sem teljesül, a virtuális gép kivételt dob

cond 1

- A `cond` feltétel-érték párok sorozatát dolgozza fel
- A `cond`-ban – mint `if`-ben és az `unless`-ben – igazságérték-eredményű, egyébként tetszőleges függvények hívhatók meg a feltételben
- Mint minden kifejezés kiértékelése, a feltételes kifejezésé is balról jobbra, fölülről lefelé halad
- Az **első** teljesülő feltételhez tartozó kifejezés értéke lesz az eredmény
- Ha egyik feltétel sem teljesül, a virtuális gép kivételt dob
- `to_decval_cond` eredménye egy hexadecimális számjegy decimális értéke; kivételt dob, ha nem hexadec számjeggyel hívjuk meg

```
def to_decval_cond(hexval) do
  cond do
    ?0 <= hexval && hexval <= ?9 -> hexval - ?0
    ?a <= hexval and hexval <= ?f -> hexval - ?a + 10
  end
end
iex> Fpea.to_decval_cond(?f)
15
```

cond 2

- A kivételdobást elkerülhetjük a `nil` érték visszaadásával:

```
... ?a <= hexval and hexval <= ?f -> hexval - ?a + 10
      true                               -> nil ...
iex> Fpea.to_decval_cond2(0)
nil
```

cond 2

- A kivételdobást elkerülhetjük a `nil` érték visszaadásával:

```
... ?a <= hexval and hexval <= ?f -> hexval - ?a + 10
      true                               -> nil ...

iex> Fpea.to_decval_cond2(0)
nil
```

- vagy az Erlangban szokásos megoldással:

```
... ?0 <= hexval && hexval <= ?9 -> {:ok, hexval - ?0}
      ?a <= hexval && hexval <= ?f -> {:ok, hexval - ?a + 10}
      true                               -> :error ...

iex> Fpea.to_decval_cond3(?z)
:error
```

case

- A case adott **értéket illeszt mintákra**, ezek kiegészíthetők örökifejezéssel

```
def to_decval_case(hexval) do
  case hexval do
    hv when ?0 <= hv and hv <= ?9
      -> hv - ?0
    ?a -> 10
    ?b -> 11
    ?c -> 12
    ?d -> 13
    ?e -> 14
    ?f -> 15
    _   -> nil
  end
end
```

case

- A case adott **értéket illeszt mintákra**, ezek kiegészíthetők örökifejezéssel

```
def to_decval_case(hexval) do
  case hexval do
    hv when ?0 <= hv and hv <= ?9
      -> hv - ?0
    ?a -> 10
    ?b -> 11
    ?c -> 12
    ?d -> 13
    ?e -> 14
    ?f -> 15
    _   -> nil
  end
end
```

- Hangsúlyozzuk, hogy bár vannak feltételes kifejezések az Elixirben (ahogy a többi funkcionális nyelvben is), célszerű helyettük, ahol csak lehet, mintaillesztést használni, esetleg őrral kiegészítve
- A mintaillesztést és esetleg őrt használó klózokkal elegánsabban, átláthatóbban oldható meg az ilyen jellegű feladatok nagy része

cond helyett case?

- A kiértékelendő kifejezés kiválasztására
 - a case kifejezést illeszt mintákra, esetleg örökifejezéssel
 - a cond tetszőleges igazságérték-eredményű függvényt elfogad
- cond helyett tehát akkor jó a case, ha a választást örökkel oldjuk meg

cond helyett case?

- A kiértékelendő kifejezés kiválasztására
 - a case kifejezést illeszt mintákra, esetleg örökifejezéssel
 - a cond tetszőleges igazságérték-eredményű függvényt elfogad
- cond helyett tehát akkor jó a case, ha a választást örökkel oldjuk meg

```
def to_decval_cond(hexval), do:  
  (cond do  
    ?0 <= hexval and hexval <= ?9 -> hexval - ?0  
    ?a <= hexval and hexval <= ?f -> hexval - ?a + 10  
  end)  
  
def to_decval_case2(hexval), do:  
  (case hexval do  
    _ when ?0 <= hexval and hexval <= ?9 -> hexval - ?0  
    _ when ?a <= hexval and hexval <= ?f -> hexval - ?a + 10  
    _ -> nil  
  end)  
  
iex> Fpea.to_decval_case2(?c)  
12
```

- De minek bonyolítsuk a dolgot? ;-). Akkor használjunk case-t, ha mintaillesztéssel gyorsítani tudjuk az esetszétválasztást.

with 1

Egy case-lánc, pl. hibakezelés esetén, nehezen átláthatóvá válhat.

```
def guess_case(val) do
```

```
  integer? = fn(v when is_integer(v)) -> {v, :ok}; _ -> :non_int end
  positive? = fn(v when v > 0) -> {v, :ok}; _ -> :non_pos end
  in_range? = fn(v when 6 <= v and v <= 18) -> {v, :ok}; _ -> :out end
```

```
  case integer?.(val) do
```

```
    {v, :ok} ->
```

```
      case positive?.(v) do
```

```
        {v, :ok} ->
```

```
          case in_range?.(v) do
```

```
            {v, :ok} -> IO.puts("#{v}: bingó!")
```

```
            :out      -> IO.puts("#{v}: out of range.")
```

```
          end
```

```
          :non_pos -> IO.puts("#{v}: wrong number.")
```

```
        end
```

```
        :non_int -> IO.puts("#{val}: wrong value.")
```

```
      end
```

```
    end
```

with 2

Ilyen esetekben hasznos a `with` makró, ami elkülöníti az adatfeldolgozást a hibakezeléstől. (Továbbiak:

<https://dev.to/martinthenth/using-elixirs-with-statement-5e36>

```
def guess_with(val) do

  integer? = fn(v when is_integer(v)) -> {v, :ok}; _ -> :non_int end
  positive? = fn(v when v > 0) -> {v, :ok}; _ -> :non_pos end
  in_range? = fn(v when 6 <= v and v <= 18) -> {v, :ok}; _ -> :out end

  with {v, :ok} <- integer?.(val),
       {v, :ok} <- positive?.(v),
       {v, :ok} <- in_range?.(v) do
    IO.puts("#{v}: bingó!")
  else
    :non_int -> IO.puts("#{val}: wrong value.")
    :non_pos -> IO.puts("#{val}: wrong number.")
    :out      -> IO.puts("#{val}: out of range.")
  end
end
```

Emlékeztető: .ex és .exs

- Az `fpea.ex` fájlt az `iex fpea.ex` paranccsal tölthetjük be, vagy ha már fut az IEx, akkor a `c "fpea.ex"-szel`

```
iex> c "fpea.ex"  
[Fpea]
```

- A betöltött modul(ok) nevét az IEx egy listában írja ki, most: `[Fpea]`.
- Az `.ex` fájlban egy vagy több modulnak kell lennie, az Elixir ezeket a belső alakra fordítja le, és úgy értelmezi. **Függvénydefiníció csak modulban lehet.**
- Ha `elixirc`-vel fordítunk egy fájlt, minden modulja külön `*.beam` fájlt hoz létre.
- Az `iex` az elérhető `.beam` fájlokat automatikusan betölti, ha hivatkozunk rájuk.
- A `.exs` arra utal, hogy a fájlban egy Elixir szkript van, amit az IEx nem fordít le, hanem közvetlenül interpretál.
- Egy Elixir szkriptben nem kell moduldefiníciónak lennie (de lehet benne).
- Az IEx azonnal kiértékeli a betöltött szkriptet.

II. rész

For-jelöléssel, egyszerűen

- 1 Feltételes kifejezés
- 2 For-jelöléssel, egyszerűen
- 3 Lineáris és elágazó rekurzió

Tartalom

- 2 For-jelöléssel, egyszerűen
 - FPE-5 – For-jelölés érdekes példákban (freq és társai, pitag, qsort, perms)
 - FPE-5 – Hatékonyság javítása: küldj több pénzt!

Mire jó a for-jelölés: `freq`

- A for-jelölés jó arra, hogy egy lista elemein végigmenjen.
- Egyszerre egynél több listaelemet csak „trükközve” tud feldolgozni.
- Magasabb rendű függvényekkel együtt „csodákra képes”.
- Nézzük ezt a feladatot: hányszor fordul elő egy adott elem egy listában?

Mire jó a for-jelölés: freq

- A for-jelölés jó arra, hogy egy lista elemein végigmenjen.
- Egyszerre egynél több listaelemet csak „trükközve” tud feldolgozni.
- Magasabb rendű függvényekkel együtt „csodákra képes”.
- Nézzük ezt a feladatot: hányszor fordul elő egy adott elem egy listában?

@spec freq(val::any(), ls::[any()]) :: n::integer()

ls-ben a val értékű elemek száma n

```
def freq(val, ls), do:  
    (for x <- ls, x === val, do: x) |> length()
```

Mire jó a for-jelölés: freq

- A for-jelölés jó arra, hogy egy lista elemein végigmenjen.
- Egyszerre egynél több listaelemet csak „trükközve” tud feldolgozni.
- Magasabb rendű függvényekkel együtt „csodákra képes”.
- Nézzük ezt a feladatot: hányszor fordul elő egy adott elem egy listában?

```
@spec freq(val::any(), ls::[any()]) :: n::integer()
```

```
# ls-ben a val értékű elemek száma n
```

```
def freq(val, ls), do:
```

```
  (for x <- ls, x === val, do: x) |> length()
```

```
iex> Fpea.freq ?a, ~c"almafa"
```

```
3
```

Mire jó a for-jelölés: freq

- A for-jelölés jó arra, hogy egy lista elemein végigmenjen.
- Egyszerre egynél több listaelemet csak „trükközve” tud feldolgozni.
- Magasabb rendű függvényekkel együtt „csodákra képes”.
- Nézzük ezt a feladatot: hányszor fordul elő egy adott elem egy listában?

@spec freq(val::any(), ls::[any()]) :: n::integer()

ls-ben a val értékű elemek száma n

```
def freq(val, ls), do:
```

```
  (for x <- ls, x === val, do: x) |> length()
```

```
iex> Fpea.freq ?a, ~c"almafa"
```

```
3
```

- Ugyanez a reduce: módosítóval:

```
def freq_r(val, ls), do:
```

```
  (for x <- ls, x === val, reduce: 0, do: (acc -> 1 + acc))
```

```
iex> Fpea.freq_r ?a, ~c"almafa"
```

```
3
```

Mire jó a for-jelölés: `smallers`

- Ha két, esetleg több szomszédos listaelemet egyszerre kell kezelni, akkor a szomszédos listaelemekből enneseket rakhatunk össze.

Mire jó a for-jelölés: `smallers`

- Ha két, esetleg több szomszédos listaelemet egyszerre kell kezelni, akkor a a szomszédos listaelemekből enneseket rakhatunk össze.
- Példa: gyűjtsük ki egy listából azokat az elemeket, melyek a közvetlenül előttük álló elemeknél kisebbek!
- Ehhez olyan párokat képezünk az `Enum.zip/2` függvénnyel, amelyek a lista 0. és 1., 1. és 2., 2. és 3., s.í.t. elemeiből állnak.

```
@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: []
```

Mire jó a for-jelölés: `smallers`

- Ha két, esetleg több szomszédos listaelemet egyszerre kell kezelni, akkor a a szomszédos listaelemekből enneseket rakhatunk össze.
- Példa: gyűjtsük ki egy listából azokat az elemeket, melyek a közvetlenül előttük álló elemeknél kisebbek!
- Ehhez olyan párokat képezünk az `Enum.zip/2` függvénnyel, amelyek a lista 0. és 1., 1. és 2., 2. és 3., s.í.t. elemeiből állnak.

```
@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: []
iex> Fpea.smallers [9,8,8,8,6,7,6,5,6,7,2,1]
[8, 6, 6, 5, 2, 1]
```

Mire jó a for-jelölés: peaks

- Következő példánk a lokális csúcsokat gyűjti ki egy listából.
- Egy csúcs akkor lokális, ha nagyobb mindkét szomszédjánál.
- Itt is alkalmazhatjuk az előző trükköt, azaz képezhetünk hármassokat a lista szomszédos elemeiből, de most az Enum modul egy másik függvényét, az Enum.chunk_every/4-et fogjuk alkalmazni.

```
@spec peaks(xs::[integer()]) :: ps::[integer()]
```

```
# az xs-beli lokális csúcsok listája ps
```

```
# egy csúcs lokális, ha mindkét szomszédjánál nagyobb
```

```
def peaks(xs) do
```

```
  triads = Enum.chunk_every(xs, 3, 1, :discard)
```

```
  for [p, c, n] <- triads, p < c and n < c, do: c
```

```
end
```

Mire jó a for-jelölés: peaks

- Következő példánk a lokális csúcsokat gyűjti ki egy listából.
- Egy csúcs akkor lokális, ha nagyobb mindkét szomszédjánál.
- Itt is alkalmazhatjuk az előző trükköt, azaz képezhetünk hármassokat a lista szomszédos elemeiből, de most az Enum modul egy másik függvényét, az Enum.chunk_every/4-et fogjuk alkalmazni.

```
@spec peaks(xs::[integer()]) :: ps::[integer()]
```

```
# az xs-beli lokális csúcsok listája ps
```

```
# egy csúcs lokális, ha mindkét szomszédjánál nagyobb
```

```
def peaks(xs) do
```

```
  triads = Enum.chunk_every(xs, 3, 1, :discard)
```

```
  for [p, c, n] <- triads, p < c and n < c, do: c
```

```
end
```

```
iex> Fpea.peaks [9,8,8,8,6,7,6,5,6,5,2,1]
```

```
[7, 6]
```

Mire jó a for-jelölés: peaks

- Következő példánk a lokális csúcsokat gyűjti ki egy listából.
- Egy csúcs akkor lokális, ha nagyobb mindkét szomszédjánál.
- Itt is alkalmazhatjuk az előző trükköt, azaz képezhetünk hármassokat a lista szomszédos elemeiből, de most az Enum modul egy másik függvényét, az Enum.chunk_every/4-et fogjuk alkalmazni.

```
@spec peaks(xs::[integer()]) :: ps::[integer()]
```

```
# az xs-beli lokális csúcsok listája ps
```

```
# egy csúcs lokális, ha mindkét szomszédjánál nagyobb
```

```
def peaks(xs) do
```

```
  triads = Enum.chunk_every(xs, 3, 1, :discard)
```

```
  for [p, c, n] <- triads, p < c and n < c, do: c
```

```
end
```

```
iex> Fpea.peaks [9,8,8,8,6,7,6,5,6,5,2,1]
```

```
[7, 6]
```

- Enum.chunk_every(list, count, step, :discard)-ban a paraméterek jelentése: list a feldolgozandó lista; count a kigyűjtendő sorozatok hossza; step a következő kiszedendő sorozat kezdete az előző kezdetéhez képest; :discard használata esetén az utolsó sorozatot csak akkor adja vissza, ha az pontosan count hosszúságú.

Lokális csúcsok gyűjtése Enum.at/2-t használó for-jelöléssel

- A lokális csúcsok keresése és a hasonló feladatok megoldhatók a hagyományos tömbös szemlélettel, indexeléssel, de ez idegen a funkcionális stílustól, és nagy listák esetén a hatékonysága is kérdéses.
- Néha – például a take, drop, split, slice és hasonló függvények használatakor – szükség van a for-jelölésben az imperatív nyelvek ciklusváltozójához hasonló futóindexre. Nézzünk most erre egy példát.
- Az alábbi **ellenjavallt** példa az Enum.at/2-t használja csúcskeresésre.

```
def peaks2(xs) do
  for i <- 1..length(xs)-1, curr = Enum.at(xs, i),
    Enum.at(xs, i-1) < curr and Enum.at(xs, i+1) < curr, do: curr
end
```

Lokális csúcsok gyűjtése: hatékonyságvizsgálat benchee-vel

- Hasonlítsuk össze a kétféle megoldás futási idejét a benchee-vel.

```
randlist = for _ <- 0..5000, do: Enum.random(0..9)
Benchee.run(%{
  "peaks chunk_every/4-gyel" => fn -> randlist |> Fpea.peaks() end,
  "peaks at/2-vel"           => fn -> randlist |> Fpea.peaks2() end,
}# , profile_after: true)
```

Name	IPS
peaks chunk_every/4-gyel	1.58 K
peaks at/2-vel	0.0130 K - 121.63x slower +76.55 ms

- Már egy 5000 elemű listára is hatalmas a különbség, több mint 120-szoros!
- 100 elemre még csak 3-szor, 20 000 elemre viszont már jóval több mint 300-szor lassabb az Enum.at/2-t használó változat. Aki nem hiszi, járjon utána! :-)
- Azaz nemcsak a funkcionális stílus, hanem a hatékonyság érdekében is érdemes kerülni az Enum.at/2-t.

Mire jó a for-jelölés: plains

- Következő példánk a maximális hosszúságú, legalább kételemű platókat gyűjti ki egy listából.
- Itt már nem működik a trükk az `Enum.zip`-pel, mert nem egyforma hosszú sorozatokat kell kiszednünk.
- Saját rekurzív függvény írása helyett használjuk az `Enum.chunk_by/2`-t.

```
@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
iex> Fpea.plains [9,0,0,0,6,7,6,5,5,2,2,2,2,1]
[[0, 0, 0], [5, 5], [2, 2, 2, 2]]
```

- `Enum.chunk_by(list, fn)` azoknak a listaelemeknek a listáját adja eredményül, amelyekre az `fn` függvény az őket közvetlenül megelőző listaelemre azonos eredményt adott.
- A példában az `& &1` ún. identitásfüggvényt használjuk; írhatjuk `&(&1)` vagy `fn(x) -> x` end alakban is.

Mire **nem** jó a for-jelölés: `slope_lengths`, `slopes`

- Az előző példában a feladatot valójában az `Enum.chunk_by/2` oldotta meg, a for-jelölést csak a kettőnél rövidebb részlisták kiszűrésére használtuk – mégpedig mintaillesztéssel, hatékonyan.

- Nézzük a következő specifikációt!

```
@type cmp() :: integer(), integer() :: boolean()
```

```
@spec slope_lengths(xs::[integer()], cmp::cmp()) :: ls::[integer()]
```

```
# az xs listából cmp alapján kigyűjtött, maximális hosszúságú, legalább kételemű
```

```
# sorozatok – lejtők, emelkedők vagy platók – hosszának listája ls
```

- Ebben a feladatban bizonyos kritériumok szerinti részlisták hosszát kell meghatározni; ehhez hasonló feladatokban magukat a részlistákat kell előállítani. Ilyen feladatok megoldására a for-jelölés már nem alkalmas.
- Gyakoriak az ilyen jellegű feladatok, ezért az `Enum` modulban van megfelelő függvény, az `Enum.chunk_while/4`; a for-jelölés nem is kell hozzá.
- A megoldást meghagyjuk gyakorló feladatnak. Egy példa a valódi lejtők – `[9,0], [7,6,5], [5,2], [2,1]` – hosszának kiszámítására.

```
iex> Slopes.slope_lengths [9,0,0,0,6,7,6,5,5,2,2,2,2,1], &</2  
[2, 3, 2, 2]
```

Pitagoraszai számhármaskok listája 1

- Könnyen átlátható, de nem hatékony megoldás

```
@spec pitag1(n::integer()) :: ps::{[integer(), integer(), integer()]}  
# az n-nél nem nagyobb összegű pitagoraszai számhármaskok listája ps  
def pitag1(n), do:  
  ls = 1..n  
  for a <- ls,  
    b <- ls,  
    a <= b,  
    c <- ls,  
    a + b + c <= n,  
    a * a + b * b == c * c,  
  do: {a, b, c}  
end
```

Pitagoraszai számhármassok listája 1

- Könnyen átlátható, de nem hatékony megoldás

```
@spec pitag1(n::integer()) :: ps::{[integer(), integer(), integer()]}
```

az n-nél nem nagyobb összegű pitagoraszai számhármassok listája ps

```
def pitag1(n), do:
```

```
  ls = 1..n
```

```
  for a <- ls,
```

```
    b <- ls,
```

```
    a <= b,
```

```
    c <- ls,
```

```
    a + b + c <= n,
```

```
    a * a + b * b === c * c,
```

```
  do: {a, b, c}
```

```
end
```

```
iex> Fpea.pitag1 12
```

```
 [{3, 4, 5}]
```

```
iex> Fpea.pitag1 36
```

```
 [{3, 4, 5}, {5, 12, 13}, {6, 8, 10}, {9, 12, 15}]
```

```
iex> Fpea.pitag 1500
```

```
 [{3, 4, 5}, {5, 12, 13}, ..., {30, 72, 78}, {30, 224, 226}, ...]
```

Pitagoraszi számhármaskok listája 1

- Könnyen átlátható, de nem hatékony megoldás

```
@spec pitag1(n::integer()) :: ps::{integer(), integer(), integer()}}  
# az n-nél nem nagyobb összegű pitagoraszi számhármaskok listája ps
```

```
def pitag1(n), do:
```

```
  ls = 1..n
```

```
  for a <- ls,
```

```
    b <- ls,
```

```
    a <= b,
```

```
    c <- ls,
```

```
    a + b + c <= n,
```

```
    a * a + b * b == c * c,
```

```
  do: {a, b, c}
```

```
end
```

```
iex> Fpea.pitag1 12
```

```
[{3, 4, 5}]
```

```
iex> Fpea.pitag1 36
```

```
[{3, 4, 5}, {5, 12, 13}, {6, 8, 10}, {9, 12, 15}]
```

```
iex> Fpea.pitag 1500
```

```
[{3, 4, 5}, {5, 12, 13}, ..., {30, 72, 78}, {30, 224, 226}, ...]
```

- Hogyan lehetne javítani a hatékonyságán? Leginkább úgy, hogy kevesebb *jelöltet* állítunk elő.

Pitagoraszi számhármassok listája 2

- Hatékonyabb megoldás a tartományok egyszerű szűkítésével

```
def pitag2(n), do:  
  for a <- 1..n,  
    b <- a..n, # b nem lehet kisebb a-nál  
    c <- b..n, # c nem lehet kisebb b-nél  
    a + b + c <= n,  
    a * a + b * b == c * c,  
  do: {a, b, c}  
end
```

Pitagoraszai számhármaskok listája 2

- Hatékonyabb megoldás a tartományok egyszerű szűkítésével

```
def pitag2(n), do:
  for a <- 1..n,
    b <- a..n, # b nem lehet kisebb a-nál
    c <- b..n, # c nem lehet kisebb b-nél
    a + b + c <= n,
    a * a + b * b == c * c,
  do: {a, b, c}
end
iex> Fpea.pitag2 36
[{3, 4, 5}, {5, 12, 13}, {6, 8, 10}, {9, 12, 15}]
iex> Fpea.pitag2 1500
[{3, 4, 5}, {5, 12, 13}, ..., {30, 72, 78}, {30, 224, 226}, ...]
```

Pitagoraszai számhármassok listája 2

- Hatékonyabb megoldás a tartományok egyszerű szűkítésével

```
def pitag2(n), do:
  for a <- 1..n,
    b <- a..n, # b nem lehet kisebb a-nál
    c <- b..n, # c nem lehet kisebb b-nél
    a + b + c <= n,
    a * a + b * b === c * c,
  do: {a, b, c}
end
iex> Fpea.pitag2 36
[{3, 4, 5}, {5, 12, 13}, {6, 8, 10}, {9, 12, 15}]
iex> Fpea.pitag2 1500
[{3, 4, 5}, {5, 12, 13}, ..., {30, 72, 78}, {30, 224, 226}, ...]
```

- A hosszú listák sok helyet foglalnak el a veremben, az összehasonlításuk – mind az eredmény helyességét, mind a futási időt tekintve – nehezebb, ezért a pitagoraszai számhármassok helyett csak a számukat határozzuk meg, adjuk eredményül; lásd a következő diákat. A függvények közös specifikációja:

*@spec pitag*cnt*(n::integer()) :: cnt::integer()*

az n-nél nem nagyobb összegű pitagoraszai számhármassok száma cnt

Pitagoraszi számhármaskok száma 1

Ez a dia egy naív és három javított hatékonyságú megoldást mutat be.

```
def pitag1cnt(n), do:  
  ls = 1..n  
  for a <- ls,  
    b <- ls,  
    a <= b,  
    c <- ls,  
    a + b + c <= n,  
    a * a + b * b == c * c,  
    reduce: 0,  
  do: (acc -> 1 + acc)  
end
```

Pitagoraszi számhármások száma 1

Ez a dia egy naív és három javított hatékonyságú megoldást mutat be.

```
def pitag1cnt(n), do:
  ls = 1..n
  for a <- ls,
    b <- ls,
    a <= b,
    c <- ls,
    a + b + c <= n,
    a * a + b * b === c * c,
    reduce: 0,
  do: (acc -> 1 + acc)
end
def pitag2cnt1(n), do:
  for a <- 1..n,
    b <- a..n, #b >= a
    c <- b..n, #c >= b
    a + b + c <= n,
    a * a + b * b === c * c,
    reduce: 0,
  do: (acc -> 1 + acc)
end
```

Pitagoraszai számhármaskok száma 1

Ez a dia egy naív és három javított hatékonyságú megoldást mutat be.

```
def pitag1cnt(n), do:
  ls = 1..n
  for a <- ls,
    b <- ls,
    a <= b,
    c <- ls,
    a + b + c <= n,
    a * a + b * b === c * c,
    reduce: 0,
  do: (acc -> 1 + acc)
end
def pitag2cnt1(n), do:
  for a <- 1..n,
    b <- a..n, #b >= a
    c <- b..n, #c >= b
    a + b + c <= n,
    a * a + b * b === c * c,
    reduce: 0,
  do: (acc -> 1 + acc)
end
```

```
def pitag2cnt2(n), do:
  for a <- 1..n,
    a2 = a * a, #Javít-e?
    b <- a..n,
    b2 = b * b, #Javít-e?
    c <- b..n,
    a + b + c <= n,
    a2 + b2 === c * c,
    reduce: 0,
  do: (acc -> 1 + acc)
end
```

Pitagoraszai számhármassok száma 1

Ez a dia egy naív és három javított hatékonyságú megoldást mutat be.

```
def pitag1cnt(n), do:
  ls = 1..n
  for a <- ls,
    b <- ls,
    a <= b,
    c <- ls,
    a + b + c <= n,
    a * a + b * b === c * c,
    reduce: 0,
  do: (acc -> 1 + acc)
end
def pitag2cnt1(n), do:
  for a <- 1..n,
    b <- a..n, #b >= a
    c <- b..n, #c >= b
    a + b + c <= n,
    a * a + b * b === c * c,
    reduce: 0,
  do: (acc -> 1 + acc)
end
```

```
def pitag2cnt2(n), do:
  for a <- 1..n,
    a2 = a * a, #Javít-e?
    b <- a..n,
    b2 = b * b, #Javít-e?
    c <- b..n,
    a + b + c <= n,
    a2 + b2 === c * c,
    reduce: 0,
  do: (acc -> 1 + acc)
end
def pitag2cnt3(n), do:
  for a <- 1..n,
    b <- a..n,
    (ab = a+b) < n, #Javít-e?
    c <- b..n,
    ab + c <= n, #Javít-e?
    a * a + b * b === c * c,
    reduce: 0,
  do: (acc -> 1 + acc)
end
```

Pitagoraszi számhármaskok száma 2

Megkérdeztük a *Copilotot*, hogyan csökkenthetném a jelöltek számát. A tartományok további szűkítési lehetőségét mutatta meg. (Egy zárójelhiba azért volt a kódban.) További két verzióval próbáltunk javítani a futási időn.

Pitagoraszi számhármaskok száma 2

Megkérdeztük a *Copilotot*, hogyan csökkenthetném a jelöltek számát. A tartományok további szűkítési lehetőségét mutatta meg. (Egy zárójelhiba azért volt a kódban.) További két verzióval próbáltunk javítani a futási időn.

```
def pitag3cnt1(n), do:
  for a <- 1..(div(n, 3)),
    b <- (a + 1)..(div(n - a, 2)),
    c = :math.sqrt(a * a + b * b),
    c == trunc(c),
    a + b + c <= n
  do {a, b, trunc(c)}
end
|> length()
end
```

Pitagoraszi számhármaskok száma 2

Megkérdeztük a *Copilotot*, hogyan csökkenthetném a jelöltek számát. A tartományok további szűkítési lehetőségét mutatta meg. (Egy zárójelhiba azért volt a kódban.) További két verzióval próbáltunk javítani a futási időn.

```
def pitag3cnt1(n), do:
  for a <- 1..(div(n, 3)),
    b <- (a + 1)..(div(n - a, 2)),
    c = :math.sqrt(a * a + b * b),
    c == trunc(c),
    a + b + c <= n
  do {a, b, trunc(c)}
end
|> length()
end
def pitag3cnt2(n), do:
  for a <- 1..(div(n, 3)),
    b <- (a + 1)..(div(n - a, 2)),
    c = :math.sqrt(a * a + b * b),
    c == trunc(c),
    a + b + c <= n,
    reduce: 0,
  do: (acc -> 1 + acc)
end
```

Pitagoraszai számhármaskok száma 2

Megkérdeztük a *Copilotot*, hogyan csökkenthetném a jelöltek számát. A tartományok további szűkítési lehetőségét mutatta meg. (Egy zárójelhiba azért volt a kódban.) További két verzióval próbáltunk javítani a futási időn.

```
def pitag3cnt1(n), do:
  for a <- 1..(div(n, 3)),
    b <- (a + 1)..(div(n - a, 2)),
    c = :math.sqrt(a * a + b * b),
    c == trunc(c),
    a + b + c <= n
  do {a, b, trunc(c)}
end
|> length()
end
def pitag3cnt2(n), do:
  for a <- 1..(div(n, 3)),
    b <- (a + 1)..(div(n - a, 2)),
    c = :math.sqrt(a * a + b * b),
    c == trunc(c),
    a + b + c <= n,
    reduce: 0,
  do: (acc -> 1 + acc)
end
```

```
def pitag3cnt3(n), do:
  for a <- 1..(div(n, 3)),
    b <- (a + 1)..(div(n - a, 2)),
    c = :math.sqrt(a * a + b * b),
    c == trunc(c),
    a + b + c <= n
  do :any_atom
  end
|> length()
end
```

Pitagoraszai számhármassok száma 2

Megkérdeztük a *Copilotot*, hogyan csökkenthetném a jelöltek számát. A tartományok további szűkítési lehetőségét mutatta meg. (Egy zárójelhiba azért volt a kódban.) További két verzióval próbáltunk javítani a futási időn.

```
def pitag3cnt1(n), do:
  for a <- 1..(div(n, 3)),
    b <- (a + 1)..(div(n - a, 2)),
    c = :math.sqrt(a * a + b * b),
    c == trunc(c),
    a + b + c <= n
  do {a, b, trunc(c)}
end
|> length()
end
def pitag3cnt2(n), do:
  for a <- 1..(div(n, 3)),
    b <- (a + 1)..(div(n - a, 2)),
    c = :math.sqrt(a * a + b * b),
    c == trunc(c),
    a + b + c <= n,
    reduce: 0,
  do: (acc -> 1 + acc)
end
```

```
def pitag3cnt3(n), do:
  for a <- 1..(div(n, 3)),
    b <- (a + 1)..(div(n - a, 2)),
    c = :math.sqrt(a * a + b * b),
    c == trunc(c),
    a + b + c <= n
  do :any_atom
  end
|> length()
end
```

Példák

```
iex> Fpea.pitag1cnt 1000
325
iex> Fpea.pitag2cnt3 1000
325
iex> Fpea.pitag3cnt1 1000
325
```

Pitagoraszai számhármassok száma 3: mérések *Benchee*-vel

```

Benchee.run(
%{"pitag1cnt:  naiv"                => fn -> 200 |> Fpea.pitag1cnt() end,
  "pitag2cnt1: b>=a,c>=b"          => fn -> 200 |> Fpea.pitag2cnt1() end,
  "pitag2cnt2: 2cnt1 + a2,b2"       => fn -> 200 |> Fpea.pitag2cnt2() end,
  "pitag2cnt3: 2cnt1 + a+b<n"       => fn -> 200 |> Fpea.pitag2cnt3() end,
  "pitag3cnt1: számolt c + {}-lista" => fn -> 200 |> Fpea.pitag3cnt1() end,
  "pitag3cnt2: 3cnt1 + reduce"      => fn -> 200 |> Fpea.pitag3cnt2() end,
  "pitag3cnt3: 3cnt1 + atomlista"   => fn -> 200 |> Fpea.pitag3cnt3() end
}# , profile_after: true)

```

Name	IPS
pitag3cnt2: 3cnt1 + reduce	5743.38
pitag3cnt3: 3cnt1 + atomlista	5597.51 - 1.03x slower +0.00454 ms
pitag3cnt1: számolt c + {}-lista	5546.79 - 1.04x slower +0.00617 ms
pitag2cnt3: 2cnt1 + a+b<n	77.52 - 74.09x slower +12.73 ms
pitag2cnt1: b>=a,c>=b	64.59 - 88.93x slower +15.31 ms
pitag2cnt2: 2cnt1 + a2,b2	61.56 - 93.30x slower +16.07 ms
pitag1cnt: naiv	24.30 - 236.39x slower +40.98 ms

Gyorsrendezés (Quicksort)

```
@spec qsort(us::[any()]) :: ss::[any()]  
# Az us lista elemeinek monoton növekedő listája ss  
# Nem helyben rendez, hanem a bemeneti lista másolata jön létre!  
def qsort([]), do: []  
def qsort([pivot|tail]), do:  
  qsort(for x <- tail, x < pivot, do: x) ++  
    [ pivot | qsort(for x <- tail, x >= pivot, do: x) ]
```

Gyorsrendezés (Quicksort)

```
@spec qsort(us::[any()]) :: ss::[any()]
# Az us lista elemeinek monoton növekedő listája ss
# Nem helyben rendez, hanem a bemeneti lista másolata jön létre!
def qsort([]), do: []
def qsort([pivot|tail]), do:
  qsort(for x <- tail, x < pivot, do: x) ++
    [ pivot | qsort(for x <- tail, x >= pivot, do: x) ]
```

Példák:

```
iex> Fpea.qsort [34, 1, 55, 78, 43.2, :math.pi(), :math.exp(1), 31.7]
[1, 2.718281828459045, 3.141592653589793, 31.7, 34, 43.2, 55, 78]
iex> Fpea.qsort [:ab, :acb, :aca, :bca, :bbca, :bac, :abc, :a, :b, :c]
[:a, :ab, :abc, :aca, :acb, :b, :bac, :bbca, :bca, :c]
iex> Fpea.qsort 'the quick brown fox jumps over the lazy dog'
~c"      abcdeefghhijklmnooopqrrsttuuvvwxyz"
iex> Fpea.qsort ["ba", 'ba', :ba, 9.3, 6, &:math.exp/1, [4, 5], [], {2, 3}]
[6, 9.3, :ba, &:math.exp/1, 2, 3, [], [4, 5], ~c"ba", "ba"]
```

Permutáció

@spec perms(xs::[any()]) :: pss::[[any()]]

Az xs lista elemeinek összes permutációját tartalmazó lista pss

```
def perms([], do: [[]])
```

```
def perms(xs), do: (for y <- xs, ys <- perms(xs--[y]), do: [y|ys])
```

Permutáció

```
@spec perms(xs::[any()]) :: pss::[[any()]]
# Az xs lista elemeinek összes permutációját tartalmazó lista pss
def perms([], do: [])
def perms(xs), do: (for y <- xs, ys <- perms(xs--[y]), do: [y|ys])
```

Példa:

```
iex> Fpea.perms [:a, :b]
[ [:a, :b], [:b, :a] ]
iex> Fpea.perms 'tér'
[
  [116, 233, 114],
  [116, 114, 233],
  [233, 116, 114],
  [233, 114, 116],
  [114, 116, 233],
  [114, 233, 116]
]
iex> 'tér'
[116, 233, 114]
```

Tartalom

- 2 For-jelöléssel, egyszerűen
 - FPE-5 – For-jelölés érdekes példákban (`freq` és társai, `pitag`, `qsort`, `perms`)
 - FPE-5 – Hatékonyság javítása: küldj több pénzt!

„Send more money” 1

- Valahol Angliában táviratot kézbesít a postás: SEND + MORE = MONEY.
- A címzett választáviratot küld a fiának: „Küldök, de mennyit?”. És a viszontválasz: „Megírtam! Minden betű más számjegy. Oldd meg!”
- A fiú nyilván nem szerénykedik, ötszámjegyű összeget kér, ezért az M nem lehet 0.
- Az egyszerű *generate-and-test* módszer nem elég jó, mert 10^8 nagyságrendű a variációk száma. Ezért már az első változatban figyelembe vesszük, hogy a betűk különböző számjegyeket jelentenek.
- Először a segédfüggvényt írjuk meg az összeg ellenőrzéséhez az

$$1000s + 100e + 10n + d + 1000m + 100o + 10r + e =$$

$$10000m + 1000o + 100n + 10e + y$$

egyenlet felhasználásával.

„Send more money” 2: Sendmory.check_sum/1

```
@type d() :: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
@type octet() :: d(),d(),d(),d(),d(),d(),d(),d()

@spec check_sum(c::octet()) :: b::boolean()
# b igaz, ha a c jelölt teljesíti az összeadási feltételt
defp check_sum(s,e,n,d,m,o,r,y) do
  send = num([s,e,n,d])
  more = num([m,o,r,e])
  money = num([m,o,n,e,y])
  send+more == money
end
```

A num/1 függvény a számjegyek listáját egész számmá alakítja:

```
@spec num(ns::[d()]) :: n::integer()
# Az ns számjegylista decimális számként n.
def num(ns) do for n <- ns, reduce: 0, do: (acc -> 10*acc + n) end
```

„Send more money” 3: Sendmory.smm1/0

```
@spec smm1() :: [octet()]
# Ellenőrzések generálás közben is.
def smm1() do
  ds = 0..9
  for s <- ds,
    e <- ds, e != s,
    n <- ds, not (n in [s,e]),
    d <- ds, not (d in [s,e,n]),
    m <- 1..9, not (m in [s,e,n,d]),
    o <- ds, not (o in [s,e,n,d,m]),
    r <- ds, not (r in [s,e,n,d,m,o]),
    y <- ds, not (y in [s,e,n,d,m,o,r]),
    check_sum(s,e,n,d,m,o,r,y),
    do: s,e,n,d,m,o,r,y
end
```

Name	ips	average	deviation	median	99th %
smm1	2.09	478.31 ms	±0.72%	477.12 ms	483.46 ms

Mindössze kétszer fut le egy másodperc alatt! Nagyon gyenge! Hol lehetne javítani rajta?

„Send more money” 4: Sendmory.smm1/0

#	CALLS	%	TIME	uS/CALL
Total	77326776	100.0	7263355	0.09
:lists.reverse/1	1	0.00	0	0.00
anonymous fn/0 in :elixir_compiler_2.__FILE__/1	1	0.00	0	0.00
anonymous fn/2 in Sendmory.smm1/0	10	0.00	1	0.10
Sendmory.smm1/0	1	0.00	1	1.00
:erlang.apply/2	1	0.00	3	3.00
anonymous fn/3 in Sendmory.smm1/0	100	0.00	10	0.10
anonymous fn/4 in Sendmory.smm1/0	900	0.00	140	0.16
anonymous fn/5 in Sendmory.smm1/0	7200	0.01	917	0.13
anonymous fn/6 in Sendmory.smm1/0	45360	0.08	5723	0.13
anonymous fn/7 in Sendmory.smm1/0	272160	0.50	36193	0.13
anonymous fn/8 in Sendmory.smm1/0	1360800	2.59	188308	0.14
Sendmory.num/1	4898880	4.30	312577	0.06
Enum.reduce/3	5612357	4.86	352752	0.06
Sendmory.check_sum/1	1632960	5.52	400715	0.25
Enum.reduce_range/5	3567385	8.03	583426	0.16
:lists.member/2	7129620	8.48	616031	0.09
anonymous fn/9 in Sendmory.smm1/0	5443200	11.17	811333	0.15
anonymous fn/2 in Sendmory.num/1	21228480	19.10	1387139	0.07
Enum."-reduce/3-listsfoldl/2-0-"/3	26127360	35.36	2568086	0.10

Nem könnyű kihámozni, hogy a sok (19) függvény közül melyikhez nyúlunk, ráadásul sok köztük a névtelen. De a Sendmory.num/1 gyanús.

„Send more money” 5: Sendmory.check_sum2/1, smm1a/0

A Sendmory.num/1 gyanús, mert sokat hívja, közvetetten, a legtöbb időt felhasználó Enum."-reduce/3-listsfoldl/2-0-"/3-t.

A Sendmory.check_sum/1 is az, mert ő hívja a Sendmory.num/1-ot.

```
defp check_sum(s,e,n,d,m,o,r,y) do
  send  = num([s,e,n,d])
  more  = num([m,o,r,e])
  money = num([m,o,n,e,y])
  send+more == money
end
```

Használjuk ki, hogy a számjegyek száma ismert (4 és 5)!

```
defp check_sum2(s,e,n,d,m,o,r,y) do
  send  = 1000*s + 100*e + 10*n + d
  more  = 1000*m + 100*o + 10*r + e
  money = 10000*m + 1000*o + 100*n + 10*e + y
  send+more == money
end
```

Sendmory.smm1a/0-ban csak annyi a változás, hogy check_sum2/1-t hívja.

Name	ips	average	deviation	median	99th %
smm1a	3.26	306.41 ms	±1.00%	305.36 ms	313.92 ms
smm1	2.07	483.01 ms	±0.85%	482.71 ms	488.42 ms

„Send more money” 6: Sendmory.smm2/0

Kb. 40%-os a javulás. Hol lehetne tovább javítani rajta? A profil egy részlete:

#	CALLS	%	TIME	uS/CALL
Sendmory.check_sum2/1	1632960	6.45	168839	0.10
anonymous fn/8 in Sendmory.smm1a/0	1360800	6.98	182793	0.13
Enum.reduce_range/5	3567385	23.37	612292	0.17
:lists.member/2	7129620	25.56	669607	0.09
anonymous fn/9 in Sendmory.smm1a/0	5443200	32.66	855524	0.16

A check_sum2/1 sokszor és sokat fut, de amíg a varációk számát nem csökkentjük (most 1 632 960), addig ezen nem tudunk változtatni.

A :lists.member/2, azaz a Kernel.in/2 operátor, a tagsági vizsgálat még többet fut, próbáljuk meg listák kivonásával, a Kernel.--/2-vel kiváltani.

```
def smm2() do
  ds = Range.to_list(0..9)
  for s <- ds,
    e <- ds -- [s],
    n <- ds -- [s,e],
    d <- ds -- [s,e,n],
    m <- ds -- [0,s,e,n,d],
    o <- ds -- [s,e,n,d,m],
    r <- ds -- [s,e,n,d,m,o],
    y <- ds -- [s,e,n,d,m,o,r],
    ...
```

„Send more money” 7: Sendmory.smm2/0, smm3/0

Name	ips	average	deviation	median	99th %
smm2	5.92	168.99 ms	±1.68%	169.47 ms	170.75 ms
smm1a	3.26	306.44 ms	±0.55%	305.61 ms	308.81 ms

Majdnem felére csökkent a futási idő, de még mindig nagyon sok. A variációk számát kellene csökkenteni, például úgy, hogy a listák építése közben eldobjuk a rossz variációkat.

Egy (tovább finomítható) lehetőség, hogy *hátról* kezdjük a számjegyek összeadását, és megvizsgáljuk, hogy pl. $d+e$ – ha kétjegyű, akkor a második számjegye – egyenlő-e y -nal (vö. **send** + **more** = **money**).

```
for d <- ds,
  e <- ds -- [d],
  y <- ds -- [d,e],
  rem(d+e, 10) == y,
  n <- ds -- [d,e,y],
  r <- ds -- [d,e,y,n],
  rem(num([n,d]) + num([r,e]), 100) == num([e,y]),
  o <- ds -- [d,e,y,n,r],
  rem(num([e,n,d]) + num([o,r,e]), 1000) == num([n,e,y]),
  s <- ds -- [d,e,y,n,r,o],
  m <- ds -- [0,d,e,y,n,r,o,s],
  check_sum2(s,e,n,d,m,o,r,y), ...
```

„Send more money” 8: Sendmory.smm3/0

Name	ips	average	deviation	median	99th %
smm3	1.42 K	703.67 us	±3.43%	695.03 us	789.63 us
smm2	0.00613 K	163209.60 us	±2.18%	163440.91 us	167366.79 us

Óriási a nyereség, kb. 230-szoros a gyorsulás! Nem olyan meglepő, ha azt is látjuk, hogy variációk száma 1866-ra csökkent az 1 632 960-ról:

#	CALLS	% TIME	uS/CALL
Sendmory.check_sum2/1	1866	1.56 233	0.12
...			
Sendmory.num/1	13422	7.45 1115	0.08
Enum.reduce/3	15304	7.72 1155	0.08
anonymous fn/2 in Sendmory.num/1	31194	18.33 2743	0.09
Enum."-reduce/3-listsfoldl/2-0-"/3	54894	47.97 7180	0.13

smm3/0-ban a num/1-et hívjuk, amiről már láttuk, hogy lassú. Hagyjuk el!

```
... n <- ds -- [d,e,y],
    r <- ds -- [d,e,y,n],
    rem(num([n,d]) + num([r,e]), 100) === num([e,y]),
    o <- ds -- [d,e,y,n,r],
    rem(num([e,n,d]) + num([o,r,e]), 1000) === num([n,e,y]), ...
```

Hat helyen hagytuk el num/1-et. Nézzük, hozott-e ez változást a futási időkhben smm3a/0-ban.

„Send more money” 9: Sendmory.smm3a/0

Name	ips	average	deviation	median	99th %
smm3a	2.40 K	416.94 us	±9.63%	405.19 us	601.81 us
smm3	1.55 K	645.86 us	±6.75%	631.41 us	851.17 us

Most is kb. 40%-os a javulás. Hol lehetne még javítani a futási időkön?

#	CALLS	% TIME	uS/CALL	
...				
anonymous fn/4 in Sendmory.smm3a/0	90	0.42	15	0.17
anonymous fn/5 in Sendmory.smm3a/0	720	2.19	79	0.11
anonymous fn/6 in Sendmory.smm3a/0	504	2.47	89	0.18
anonymous fn/9 in Sendmory.smm3a/0	732	3.77	136	0.19
Enum.reduce/3	1882	4.22	152	0.08
anonymous fn/8 in Sendmory.smm3a/0	1450	5.25	189	0.13
Sendmory.check_sum2/1	1866	6.91	249	0.13
anonymous fn/9 in Sendmory.smm3a/0	1866	9.77	352	0.19
anonymous fn/7 in Sendmory.smm3a/0	3024	10.07	363	0.12
:erlang.--/2	1881	13.10	472	0.25
Enum."-reduce/3-listsfoldl/2-0-"/3	10278	41.66	1501	0.15

További varációk kizárásával, például úgy, hogy a két szám összeadásakor esetleg keletkező átvitelt is felhasználjuk.

Esetleg a for-comprehenzió kiváltásával (most már csak a komprehenzió hívja az Enum."-reduce/3-listsfoldl/2-0-"/3-t).

III. rész

Lineáris és elágazó rekurzió

- 1 Feltételes kifejezés
- 2 For-jelöléssel, egyszerűen
- 3 **Lineáris és elágazó rekurzió**

Tartalom

- 3 Lineáris és elágazó rekurzió
 - FPE-5 – Rekurzív adatstruktúrák

Rekurzív adatstruktúrák

A rekurzív adatstruktúrák (adattípusok) alapvetően kétfélék.

- Lineáris rekurzív adatstruktúra (lista)

A funkcionális nyelvekben: beépített típus

Megvalósítás: láncolt listával

- Elixirben a beépített lista típusa: *@type list() :: [] | [any() | list()]*
- *Ennessel is megvalósíthatjuk*, nevezzük veremnek:
@type lifo() :: :empty | {lifo(), any()} # last-in-first-out

- Elágazó rekurzív adatstruktúra (fa)

Általában nem beépített típus a funkcionális nyelvekben, de többnyire vannak fák használatát segítő modulok (pl. Erlangban a `gb_tree`)

- Bináris fa: *@type btree() :: :lf | {btree(), any(), btree()}*
- Három ágú fa: *@type ttree() :: :lf | {ttree(), ttree(), ttree(), any()}*
- Sok ágú fa (pl. egy listában vannak a csomópontokhoz tartozó fák):
@type ltree() :: :lf | {any(), [ltree()]}

Az ennessben az érték és az ágak sorrendje tetszőleges. Az érték lehetne csak a levélben vagy levélben is.

Verem megvalósítása ennessel 1 (dp_lifo.ex)

- Típusa: *@type lifo() :: :empty | {lifo(), any()} # last-in-first-out*
- Definíciók:
 - verem teteje = a verem tetején lévő elem
 - verem alja = a verem teteje alatti veremrész
- Műveletek:
 - üres verem létrehozása (empty/0)
 - a verem üres voltának vizsgálata (is_empty/1)

@spec empty() :: s::lifo()

s az üres verem

```
def empty(), do: :empty
```

@spec is_empty(s::lifo()) :: b::boolean()

b igaz, ha s üres

```
def is_empty(:empty), do: true
```

```
def is_empty({_s,_x}), do: false
```

Verem megvalósítása ennessel 2 (dp_lifo.ex)

- Típusa: *@type lifo() :: :empty | {lifo(), any()} # last-in-first-out*
- Műveletek:
 - egy elem berakása a verembe (push/2)
 - a verem alja (pop/1)
 - a verem teteje (top/1)

@spec push(s::lifo(), x::any()) :: s_new::lifo()

s_new az x-szel megfejtelt s verem

def push(:empty, x), do: {:empty, x}

def push({_s, _x} = s, x), do: {s, x} # {_s, _x} = s: réteges minta

@spec pop(s::lifo()) :: s_new::(lifo() | nil)

s_new az s verem alja, vagy nil, ha s üres

def pop(:empty), do: nil

def pop({s, _x}), do: s

@spec top(s::lifo()) :: x::(any() | nil)

x az s verem teteje, vagy nil, ha s üres

def top(:empty), do: nil

def top({_s, x}), do: x

Kis példák a verem használatára 1 (dp_lifo.ex)

Elemek berakása egy verembe

```
iex> s1 = Dp.Lifo.push(Dp.Lifo.empty(), 1)
{:empty, 1}
iex> s2 = Dp.Lifo.push(s1, 2)
{{:empty, 1}, 2}
iex> s3 = Dp.Lifo.push(s2, 3)
{{{:empty, 1}, 2}, 3}
```

Ugyanez tömörebben és érthetőbben:

```
iex> push = &Dp.Lifo.push/2; Dp.Lifo.empty() |> push.(1) |> push.(2) |> push.(3)
{{{:empty, 1}, 2}, 3}
```

Kis példák a verem használatára 2 (dp_lifo.ex)

Fordítsunk meg egy karakterlistát!

Emlékeztetőül a `push` specifikációja:

@spec push(s::lifo(), x::any()) :: s_new::lifo().

1. lépés: a verembe betesszük az elemeket

```
iex> ~c"szöveg"  
[115, 122, 246, 118, 101, 103]  
iex> lifo = ~c"szöveg" \  
    |> List.foldl(Dp.Lifo.empty(), &(Dp.Lifo.push(&2,&1)))  
{{{{{{{empty, 115}, 122}, 246}, 118}, 101}, 103}}
```

2. lépés: a verem elemeit sorban kivesszük és listába fűzzük

*@spec to_list(s::lifo()) :: ls::[any()]
ls az s verem elemeit tartalmazó lista LIFO sorrendben*

```
def to_list(:empty), do: []  
def to_list({s, x}), do: [x | to_list(s)]
```

```
iex> Dp.Lifo.to_list(lifo)  
[103, 101, 118, 246, 122, 115]  
iex> Dp.Lifo.to_list(lifo) |> List.to_string  
"gevözs"
```

Rekurzív adatstruktúrák 2

A rekurzív adatstruktúrák (adattípusok) alapvetően kétfélék

- Lineáris rekurzív adatstruktúra (lista)

A funkcionális nyelvekben: beépített típus

Megvalósítás: láncolt listával

- Elixirben a beépített lista típusa: *@type list() :: [] | [any() | list()]*
- Ennessel is megvalósíthatjuk, nevezzük veremnek (last-in-first-out):
@type lifo() :: :empty | {lifo(), any()} # last-in-first-out

- Elágazó rekurzív adatstruktúra (fa)

Általában nem beépített típus a funkcionális nyelvekben, de többnyire vannak fák használatát segítő modulok (pl. Erlangban a `gb_tree`)

- Bináris fa: *@type btree() :: :lf | {btree(), any(), btree()}*
- Három ágú fa: *@type ttree() :: :lf | {ttree(), ttree(), ttree(), any()}*
- Sok ágú fa (pl. egy listában vannak a csomópontokhoz tartozó fák):
@type ltree() :: :lf | {any(), [ltree()]}

Az ennesekben az érték és az ágak sorrendje tetszőleges. Az érték lehetne csak a levélben vagy lehetne levélben is.

Bináris fából lista preorder bejárással (dp_tree.ex)

- Típusdefiníció (egy a sok lehetséges közül)

@type btree() :: :lf | {btree(), any(), btree()}

- Jelölés (példák)

bt1 = { :lf, :a, :lf }

bt2a = { { :lf, :c, :lf }, :b, :lf }

bt2b = { :lf, :e, { :lf, :f, :lf } }

bt2c = { { :lf, :h, :lf }, :g, { :lf, :i, :lf } }

bt2 = { bt2a, :a, { bt2b, :d, bt2c } }

- Bejárás (preorder)

@spec bt_to_list(btr :: btree()) :: xs :: [any()]

A btr fa preorder bejárásának eredménye az xs lista

def bt_to_list(:lf), do: []

def bt_to_list({bt, v, jt}), do:

[v | bt_to_list(bt)] ++ bt_to_list(jt)

- Futtatás

iex> Dp.Tree.bt_to_list(bt1)

[:a]

iex> Dp.Tree.bt_to_list(bt2)

[:a, :b, :c, :d, :e, :f, :g, :h, :i]

Három ágú fából lista preorder bejárással (dp_tree.ex)

- Típusdefiníció (egy a sok lehetséges közül)

@type ttree() :: :lf | {ttree(), ttree(), ttree(), any()}

- Jelölés (példák)

tt1 = { :lf, :lf, :lf, :a }

tt2a = { { :lf, :lf, :lf, :c }, :lf, :lf, :b }

tt2b = { { :lf, :lf, :lf, :d }, { :lf, :lf, :lf, :f }, :lf, :e }

tt2c = { { :lf, :lf, :lf, :h }, :lf, { :lf, :lf, :lf, :i }, :g }

tt2 = { tt2a, tt2b, tt2c, :a }

- Bejárás (preorder)

@spec tt_to_list(ttr :: ttree()) :: xs :: [any()]

A ttr fa preorder bejárásának eredménye az xs lista

def tt_to_list(:lf), do: []

def tt_to_list(bt, mt, jt, v), do:

[v | tt_to_list(bt)] ++ tt_to_list(mt) ++ tt_to_list(jt)

- Futtatás

iex> Dp.Tree.tt_to_list(tt1)

[:a]

iex> Dp.Tree.tt_to_list(tt2)

[:a, :b, :c, :d, :e, :f, :g, :h, :i]

Sok ágú fából lista preorder bejárással (dp_tree.ex)

- Típusdefiníció (egy a sok lehetséges közül)

@type ltree() :: :lf | {any(), [ltree()]}

- Jelölés (példák)

lt1 = {:lt, []}

lt2a = {:b, [{:c, []}, {:d, []}]}

lt2b = {:e, [{:f, []}]}

lt2c = {:g, [{:h, []}, {:i, []}]}

lt2 = {:a, [lt2a, lt2b, lt2c]}

- Bejárás (preorder)

@spec lt_to_list(ltr :: ltree()) :: xs :: [any()]

Az ltr fa preorder bejárásának eredménye az xs lista

```
def lt_to_list(:lf), do: []
```

```
def lt_to_list(v, ts), do:
```

```
  [ v | Enum.concat(Enum.map(ts, &lt_to_list/1)) ]
```

- Futtatás

```
iex> {(Dp.Tree.lt_to_list :lf), Dp.Tree.lt_to_list lt1}
{[], [:a]}
```

```
iex> Dp.Tree.lt_to_list(lt2)
```

```
[:a, :b, :c, :d, :e, :f, :g, :h, :i]
```

Egyszerű műveletek bináris fákon 1 (dp_tree.ex)

- Legyen most ez a bináris fánk típusa:

```
@type btree2() :: :lf | {any(), btree2(), btree2()}
```

- Fa létrehozása

```
@spec empty() :: btr::(:lf)
```

```
# btr az egyetlen levélből álló üres fa
```

```
def empty(), do: :lf
```

```
@spec node(v: any(), lt::btree2(), rt::btree2()) :: btr::btree2()
```

```
# btr a v értékből, az lt és az rt fákból összerakott fa
```

```
node(v, lt, rt), do: {v, lt, rt}
```

- Fa leveleinek és szintjeinek száma

```
@spec leaves_cnt(btr::btree2()) :: n::integer()
```

```
# n a btr fa leveleinek száma
```

```
def leaves_cnt(:lf), do: 1
```

```
def leaves_cnt({_ ,lt,rt}), do: leaves_cnt(lt)+leaves_cnt(rt)
```

```
@spec depth(btr::btree2()) :: d::integer()
```

```
# d a btr fa mélysége, azaz szintjének száma
```

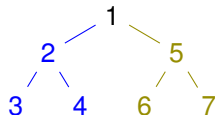
```
def depth(:lf), do: 0
```

```
def depth({_ ,lt,rt}), do: 1 + max(depth(lt), depth(rt))
```

Egyszerű műveletek bináris fákon 2 (dp_tree.ex)

```
e=empty(), t=node(1, node(2, node(3,e,e),
                                node(4,e,e)),
                  node(5, node(6,e,e),
                                node(7,e,e)))
```

~



```
iex> e = &Dp.Tree.empty/0; n = &Dp.Tree.node/3; \
      t = n.(1, n.(2, n.(3, e.(), e.()), n.(4, e.(), e.())), \
            n.(5, n.(6, e.(), e.()), n.(7, e.(), e.())))
{1, {2, {3,:lf,:lf}, {4,:lf,:lf}}, {5, {6,:lf,:lf}, {7,:lf,:lf}}}
```

```
iex> Dp.Tree.leaves_cnt t
```

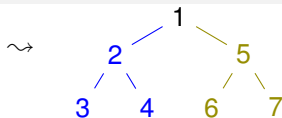
```
8
```

```
iex> Dp.Tree.depth t
```

```
3
```

Műveletek bináris fákon: fából lista (dp_tree.ex)

```
e=empty(), t=node(1, node(2, node(3,e,e),
                             node(4,e,e)),
                  node(5, node(6,e,e),
                             node(7,e,e)))
```



@spec to_list_preord(btr::btree2()) :: xs::[any()]

A btr fa preorder bejárásának eredménye az xs lista

```
def to_list_preord(:lf), do: []
```

```
def to_list_preord({v,lt,rt}), do:
```

```
  [v] ++ to_list_preord(lt) ++ to_list_preord(rt)
```

@spec to_list_inord(btr::btree2()) :: xs::[any()]

A btr fa inorder bejárásának eredménye az xs lista

```
def to_list_inord(:lf), do: []
```

```
def to_list_inord({v,lt,rt}), do:
```

```
  to_list_inord(lt) ++ ([v] ++ to_list_inord(rt))
```

```
iex> Dp.Tree.to_list_preord(t)
```

```
[1, 2, 3, 4, 5, 6, 7]
```

```
iex> Dp.Tree.to_list_inord(t)
```

```
[3, 2, 4, 1, 6, 5, 7]
```

Műveletek bináris fákon: listából fa (dp_tree.ex)

```
t = {1, {2, {3,:lf,:lf}, {4,:lf,:lf}}, {5, {6,:lf,:lf}, {7,:lf,:lf}}}
```

```
@spec from_list_preord(xs::[any()]) :: btr::btree2()
```

```
# btr az xs listából preorder sorrendben felépített fa
```

```
def from_list_preord([]), do: empty()
```

```
def from_list_preord([x|xs]) do
```

```
  {l1s, l2s} = Enum.split(xs, div(length(xs), 2))
```

```
  node(x, from_list_preord(l1s), from_list_preord(l2s))
```

```
end
```

```
@spec from_list_inord(xs::[any()]) :: btr::btree2()
```

```
# btr az xs listából inorder sorrendben felépített fa
```

```
def from_list_inord([]), do: empty()
```

```
def from_list_inord(xs) do
```

```
  {l1s, [x|l2s]} = Enum.split(xs, div(length(xs), 2))
```

```
  node(x, from_list_inord(l1s), from_list_inord(l2s))
```

```
end
```

```
iex> Dp.Tree.from_list_preord([1, 2, 3, 4, 5, 6, 7])
```

```
{1,{2,{3,:lf,:lf},{4,:lf,:lf}},{5,{6,:lf,:lf},{7,:lf,:lf}}}
```

```
iex> Dp.Tree.from_list_inord([3, 2, 4, 1, 6, 5, 7])
```

```
{1,{2,{3,:lf,:lf},{4,:lf,:lf}},{5,{6,:lf,:lf},{7,:lf,:lf}}}
```

5. gyakorlat

- Gyakorlat anyaga:

<https://dp.iit.bme.hu/dp26a/gy/dp26a-fp5gy.livemd>