

Deklaratív programozás 4. 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

Kifejezés, őrkifejezés

- 1 Kifejezés, őrkifejezés
- 2 Függvény, névtelen függvény, magasabb rendű függvény

Tartalom

- 1 Kifejezés, őrkifejezés
 - FPE-4 – Kifejezés, szekvenciális kifejezés, őrkifejezés

Kifejezések

Kifejezések

- Elég alaposan kistafíroztuk magunkat különféle könyvtári függvényekkel (ideértve az infix pozíciójú műveleteket is) ahhoz, hogy összetettebb feladatokat oldjunk meg
- Nézzük most a kifejezés különféle változatait az Elixirben
- Először is szögezzük le *ismét*, hogy az Elixirben *minden* kifejezés – az összes többi funkcionális nyelvhez hasonlóan

Kifejezések

- Elég alaposan kistafíroztuk magunkat különféle könyvtári függvényekkel (ideértve az infix pozíciójú műveleteket is) ahhoz, hogy összetettebb feladatokat oldjunk meg
- Nézzük most a kifejezés különféle változatait az Elixirben
- Először is szögezzük le *ismét*, hogy az Elixirben *minden* kifejezés – az összes többi funkcionális nyelvhez hasonlóan
- A kifejezés lehet:
 - Összetett kifejezés: tetszőleges adatstruktúra, lehetnek benne elvégzendő műveletek/függvényhívások is – sok példát láttunk rá
 - Term: mint az összetett kifejezés, de *tömörnek*, tovább már nem egyszerűsíthetőnek kell lennie – erről is volt már szó
 - Szekvenciális kifejezés – ilyen példák is voltak már, de azért összefoglaljuk a dolgokat
 - Őrkifejezés – ez új téma, bár már láttunk rá példát
 - Feltételes kifejezés – ez is új, bár már láttunk rá példát

Kifejezés kiértékelése

- Funcionális nyelvek esetében mindig *kiértékelésről* (evaluation) beszélünk, sohasem *végrehajtásról*; az utóbbi használatát meghagyjuk az imperatív nyelveknek
- A kifejezés *kiértékelése* alapvetően *mohó* (eager, strict) az Elixirben
- De vannak kivételek, azaz *lusta* (lazy, non-strict) kiértékelésű kifejezések, pl. kétoperandusú logikai műveletek, valamint teljes modul is: `Stream`

Kifejezés kiértékelése

- Funcionális nyelvek esetében mindig *kiértékelésről* (evaluation) beszélünk, sohasem *végrehajtásról*; az utóbbi használatát meghagyjuk az imperatív nyelveknek
- A kifejezés *kiértékelése* alapvetően *mohó* (eager, strict) az Elixirben
- De vannak kivételek, azaz *lusta* (lazy, non-strict) kiértékelésű kifejezések, pl. kétoperandusú logikai műveletek, valamint teljes modul is: `Stream`

Vessük össze a mohó és a lusta kiértékelést az alábbi példák alapján!

```
iex> nevező = 0
0
iex> nevező > 0 && ( 1 / nevező ) < 1
false
iex>
```

Kifejezés kiértékelése

- Funcionális nyelvek esetében mindig *kiértékelésről* (evaluation) beszélünk, sohasem *végrehajtásról*; az utóbbi használatát meghagyjuk az imperatív nyelveknek
- A kifejezés *kiértékelése* alapvetően *mohó* (eager, strict) az Elixirben
- De vannak kivételek, azaz *lusta* (lazy, non-strict) kiértékelésű kifejezések, pl. kétoperandusú logikai műveletek, valamint teljes modul is: Stream

Vessük össze a mohó és a lusta kiértékelést az alábbi példák alapján!

```
iex> nevező = 0
0
iex> nevező > 0 && ( 1 / nevező ) < 1
false
iex> (&Kernel.&&/2).( nevező > 0, ( 1 / nevező ) < 1) # Mi más itt?
** (ArithmeticError) bad argument in arithmetic expression: 1 / 0 ...
iex>
```

Kifejezés kiértékelése

- Funcionális nyelvek esetében mindig *kiértékelésről* (evaluation) beszélünk, sohasem *végrehajtásról*; az utóbbi használatát meghagyjuk az imperatív nyelveknek
- A kifejezés *kiértékelése* alapvetően *mohó* (eager, strict) az Elixirben
- De vannak kivételek, azaz *lusta* (lazy, non-strict) kiértékelésű kifejezések, pl. kétoperandusú logikai műveletek, valamint teljes modul is: Stream

Vessük össze a mohó és a lusta kiértékelést az alábbi példák alapján!

```
iex> nevező = 0
0
iex> nevező > 0 && ( 1 / nevező ) < 1
false
iex> (&Kernel.&&/2).( nevező > 0, ( 1 / nevező ) < 1) # Mi más itt?
** (ArithmeticError) bad argument in arithmetic expression: 1 / 0 ...
iex> Kernel.&&( nevező > 0, ( 1 / nevező ) < 1) # Mi más itt?
false
iex>
```

Kifejezés kiértékelése

- Funcionális nyelvek esetében mindig *kiértékelésről* (evaluation) beszélünk, sohasem *végrehajtásról*; az utóbbi használatát meghagyjuk az imperatív nyelveknek
- A kifejezés *kiértékelése* alapvetően *mohó* (eager, strict) az Elixirben
- De vannak kivételek, azaz *lusta* (lazy, non-strict) kiértékelésű kifejezések, pl. kétoperandusú logikai műveletek, valamint teljes modul is: Stream

Vessük össze a mohó és a lusta kiértékelést az alábbi példák alapján!

```
iex> nevező = 0
0
iex> nevező > 0 && ( 1 / nevező ) < 1
false
iex> (&Kernel.&&/2).( nevező > 0, ( 1 / nevező ) < 1) # Mi más itt?
** (ArithmeticError) bad argument in arithmetic expression: 1 / 0 ...
iex> Kernel.&&( nevező > 0, ( 1 / nevező ) < 1) # Mi más itt?
false
iex> nevező == 0 && ( 1 / nevező ) < 1 # És mi más itt?
** (ArithmeticError) bad argument in arithmetic expression: 1 / 0 ...
```

Szekvenciális kifejezés

- Kifejezések pontosvesszővel elválasztott, opcionálisan *zárójelezett* sorozata: $\text{exp}_1; \text{exp}_2; \dots; \text{exp}_n$ vagy $(\text{exp}_1; \text{exp}_2; \dots; \text{exp}_n)$
- Kell a zárójel, ha az adott helyen egyetlen kifejezésnek kell állnia
- A szekvenciális kifejezés értéke az utolsó részkifejezés – exp_n – értéke
- exp_i -ben ($i < n$)
 - vagy változóhoz kötünk értéket,
 - vagy mellékhatást akarunk elérni (pl. kiírunk valamit),
 - egyébként a kifejezés értéke „elvész”

Szekvenciális kifejezés

- Kifejezések pontosvesszővel elválasztott, opcionálisan *zárójelezett* sorozata: $\text{exp}_1; \text{exp}_2; \dots; \text{exp}_n$ vagy $(\text{exp}_1; \text{exp}_2; \dots; \text{exp}_n)$
- Kell a zárójel, ha az adott helyen egyetlen kifejezésnek kell állnia
- A szekvenciális kifejezés értéke az utolsó részkifejezés – exp_n – értéke
- exp_i -ben ($i < n$)
 - vagy változóhoz kötünk értéket,
 - vagy mellékhatást akarunk elérni (pl. kiírunk valamit),
 - egyébként a kifejezés értéke „elvész”
- Példák:

```
iex> sumsq = fn(x,y) -> sq = fn x -> x*x end;\
                        sq.(x) + sq.(y) end; sumsq.(3,5)
```

```
34
```

```
iex>
```

Szekvenciális kifejezés

- Kifejezések pontosvesszővel elválasztott, opcionálisan *zárójelezett* sorozata: $\text{exp}_1; \text{exp}_2; \dots; \text{exp}_n$ vagy $(\text{exp}_1; \text{exp}_2; \dots; \text{exp}_n)$
- Kell a zárójel, ha az adott helyen egyetlen kifejezésnek kell állnia
- A szekvenciális kifejezés értéke az utolsó részkifejezés – exp_n – értéke
- exp_i -ben ($i < n$)
 - vagy változóhoz kötünk értéket,
 - vagy mellékhatást akarunk elérni (pl. kiírunk valamit),
 - egyébként a kifejezés értéke „elvész”
- Példák:

```
iex> sumsq = fn(x,y) -> sq = fn x -> x*x end;\
                        sq.(x) + sq.(y) end; sumsq.(3,5)
```

```
34
```

```
iex> [x=3; x+x]
```

```
** (SyntaxError) iex:1:5: syntax error before: ';'
```

```
iex> [(x=3; x+x)]
```

```
[6]
```

```
iex> "x értéke"; x
```

```
3
```

```
iex> IO.write "x értéke "; x
```

```
x értéke 3
```

Kifejezések csoportosítása

- A szekvenciális kifejezés egyfajta csoportosítást jelent
- A `do ... end` blokk a csoportosítás egy másik változata
- `do ... end` blokkot használunk pl. függvénydefinícióban, feltételes kifejezésben, `for`-jelölésben
- Egy `do ... end` blokk belseje egy szekvenciális kifejezés, amely állhat egyetlen kifejezésből, vagy részkifejezések pontosvesszővel vagy új sorba írással elválasztott sorozatából

Kifejezések csoportosítása

- A szekvenciális kifejezés egyfajta csoportosítást jelent
- A `do ... end` blokk a csoportosítás egy másik változata
- `do ... end` blokkot használunk pl. függvénydefinícióban, feltételes kifejezésben, `for`-jelölésben
- Egy `do ... end` blokk belseje egy szekvenciális kifejezés, amely állhat egyetlen kifejezésből, vagy részkifejezések pontosvesszővel vagy új sorba írással elválasztott sorozatából
- A `do ... end` blokk valójában egy olyan kulcs-érték pár alternatív írásmódja, amelyben `do`: a kulcs, az érték pedig az esetleg zárójelbe tett szekvenciális kifejezés

Kifejezések csoportosítása

- A szekvenciális kifejezés egyfajta csoportosítást jelent
- A `do ... end` blokk a csoportosítás egy másik változata
- `do ... end` blokkot használunk pl. függvénydefinícióban, feltételes kifejezésben, `for`-jelölésben
- Egy `do ... end` blokk belseje egy szekvenciális kifejezés, amely állhat egyetlen kifejezésből, vagy részkifejezések pontosvesszővel vagy új sorba írással elválasztott sorozatából
- A `do ... end` blokk valójában egy olyan kulcs-érték pár alternatív írásmódja, amelyben `do`: a kulcs, az érték pedig az esetleg zárójelbe tett szekvenciális kifejezés

Példák:

```
def hello_1(msg, name), do: (IO.write msg; IO.puts " , mizújs, #{name}?")
```

Kifejezések csoportosítása

- A szekvenciális kifejezés egyfajta csoportosítást jelent
- A `do ...end` blokk a csoportosítás egy másik változata
- `do ...end` blokkot használunk pl. függvénydefinícióban, feltételes kifejezésben, `for`-jelölésben
- Egy `do ...end` blokk belseje egy szekvenciális kifejezés, amely állhat egyetlen kifejezésből, vagy részkifejezések pontosvesszővel vagy új sorba írással elválasztott sorozatából
- A `do ...end` blokk valójában egy olyan kulcs-érték pár alternatív írásmódja, amelyben `do`: a kulcs, az érték pedig az esetleg zárójelbe tett szekvenciális kifejezés

Példák:

```
def hello_1(msg, name), do: (IO.write msg; IO.puts ", mizújs, #{name}?")
def hello_2(msg, name), do: (      # csoportosítás zárójelezéssel
    IO.write msg                  # pontosvessző helyett új sorba
    IO.puts ", mizújs, #{name}?") )
```

Kifejezések csoportosítása

- A szekvenciális kifejezés egyfajta csoportosítást jelent
- A `do ... end` blokk a csoportosítás egy másik változata
- `do ... end` blokkot használunk pl. függvénydefinícióban, feltételes kifejezésben, `for`-jelölésben
- Egy `do ... end` blokk belseje egy szekvenciális kifejezés, amely állhat egyetlen kifejezésből, vagy részkifejezések pontosvesszővel vagy új sorba írással elválasztott sorozatából
- A `do ... end` blokk valójában egy olyan kulcs-érték pár alternatív írásmódja, amelyben `do`: a kulcs, az érték pedig az esetleg zárójelbe tett szekvenciális kifejezés

Példák:

```
def hello_1(msg, name), do: (IO.write msg; IO.puts ", mizújs, #{name}?")
def hello_2(msg, name), do: (      # csoportosítás zárójelezéssel
  IO.write msg                    # pontosvessző helyett új sorba
  IO.puts ", mizújs, #{name}?" )
def hello_3(msg, name) do        # zárójel helyett do...end
  IO.write msg
  IO.puts ", mizújs, #{name}?" # #{tömör kif.}: ún. interpoláció
end                             # az interpoláció sztringben használható
```

Őrkifejezés, ór 1

- Láttuk, hogy egy függvény definiálásakor a különféle esetekre önálló klózokat írunk, az eseteket pedig mintaillesztéssel ismerjük föl
- De mintaillesztéssel nem lehet megkülönböztetni minden lehetséges esetet, pl. hogy egy érték lista-e vagy atom, negatív-e vagy pozitív
- Ilyenkor `when`-nel kezdődő *őrkifejezést* (guard clause) használhatunk
- Az őrnek ***mellékhatás nélküli, hatékonyan kiértékelhető, kivételt soha nem dobó, igazságérték-eredményű*** kifejezésnek kell lennie

Őrkifejezés, őr 1

- Láttuk, hogy egy függvény definiálásakor a különféle esetekre önálló klózokat írunk, az eseteket pedig mintaillesztéssel ismerjük föl
- De mintaillesztéssel nem lehet megkülönböztetni minden lehetséges esetet, pl. hogy egy érték lista-e vagy atom, negatív-e vagy pozitív
- Ilyenkor `when`-nel kezdődő *őrkifejezést* (guard clause) használhatunk
- Az őrnek ***mellékhatás nélküli, hatékonyan kiértékelhető, kivételt soha nem dobó, igazságérték-eredményű*** kifejezésnek kell lennie
- Az őrkifejezésben lehet:
 - Term (tömör kifejezés)

Őrkifejezés, őr 1

- Láttuk, hogy egy függvény definiálásakor a különféle esetekre önálló klózokat írunk, az eseteket pedig mintaillesztéssel ismerjük föl
- De mintaillesztéssel nem lehet megkülönböztetni minden lehetséges esetet, pl. hogy egy érték lista-e vagy atom, negatív-e vagy pozitív
- Ilyenkor `when`-nel kezdődő *őrkifejezést* (guard clause) használhatunk
- Az őrnek ***mellékhatás nélküli, hatékonyan kiértékelhető, kivételt soha nem dobó, igazságérték-eredményű*** kifejezésnek kell lennie
- Az őrkifejezésben lehet:
 - Term (tömör kifejezés)
 - Egyes modulok *guard*-ként definiált függvényei, például
 - a Kernel modul összes típust vizsgáló predikátuma (`is_TÍPUS`)
 - a Kernel modulban definiált `abs/`, `round/1`, `trunc/1`, `ceil/1`, `floor(/1`,

Őrkifejezés, őr 1

- Láttuk, hogy egy függvény definiálásakor a különféle esetekre önálló klózokat írunk, az eseteket pedig mintaillesztéssel ismerjük föl
- De mintaillesztéssel nem lehet megkülönböztetni minden lehetséges esetet, pl. hogy egy érték lista-e vagy atom, negatív-e vagy pozitív
- Ilyenkor `when`-nel kezdődő *őrkifejezést* (guard clause) használhatunk
- Az őrnek ***mellékhatás nélküli, hatékonyan kiértékelhető, kivételt soha nem dobó, igazságérték-eredményű*** kifejezésnek kell lennie
- Az őrkifejezésben lehet:
 - Term (tömör kifejezés)
 - Egyes modulok *guard*-ként definiált függvényei, például
 - a Kernel modul összes típust vizsgáló predikátuma (`is_TÍPUS`)
 - a Kernel modulban definiált `abs/`, `round/1`, `trunc/1`, `ceil/1`, `floor(/1`, `elem/2`, `tuple_size/1`, `hd/1`, `tl/1`,

Őrkifejezés, őr 1

- Láttuk, hogy egy függvény definiálásakor a különféle esetekre önálló klózokat írunk, az eseteket pedig mintaillesztéssel ismerjük föl
- De mintaillesztéssel nem lehet megkülönböztetni minden lehetséges esetet, pl. hogy egy érték lista-e vagy atom, negatív-e vagy pozitív
- Ilyenkor `when`-nel kezdődő *őrkifejezést* (guard clause) használhatunk
- Az őrnek ***mellékhatás nélküli, hatékonyan kiértékelhető, kivételt soha nem dobó, igazságérték-eredményű*** kifejezésnek kell lennie
- Az őrkifejezésben lehet:
 - Term (tömör kifejezés)
 - Egyes modulok *guard*-ként definiált függvényei, például
 - a Kernel modul összes típust vizsgáló predikátuma (`is_TÍPUS`)
 - a Kernel modulban definiált `abs/`, `round/1`, `trunc/1`, `ceil/1`, `floor(/1`, `elem/2`, `tuple_size/1`, `hd/1`, `tl/1`, `length/1`, `in/2`, `bit_size/1`, `byte_size/1`

Őrkifejezés, ór 1

- Láttuk, hogy egy függvény definiálásakor a különféle esetekre önálló klózokat írunk, az eseteket pedig mintaillesztéssel ismerjük föl
- De mintaillesztéssel nem lehet megkülönböztetni minden lehetséges esetet, pl. hogy egy érték lista-e vagy atom, negatív-e vagy pozitív
- Ilyenkor *when*-nel kezdődő *őrkifejezést* (guard clause) használhatunk
- Az őrnek ***mellékhatás nélküli, hatékonyan kiértékelhető, kivételt soha nem dobó, igazságérték-eredményű*** kifejezésnek kell lennie
- Az őrkifejezésben lehet:
 - Term (tömör kifejezés)
 - Egyes modulok *guard*-ként definiált függvényei, például
 - a Kernel modul összes típust vizsgáló predikátuma (*is_TÍPUS*)
 - a Kernel modulban definiált *abs/*, *round/1*, *trunc/1*, *ceil/1*, *floor(/1*, *elem/2*, *tuple_size/1*, *hd/1*, *tl/1*, *length/1*, *in/2*, *bit_size/1*, *byte_size/1*
 - az Integer modulban definiált *is_even/*, *is_odd/1*
 - Örökből álló olyan kifejezés, mely aritmetikai, összehasonlító, logikai (kivétel: *&&*, *||*, *!*) és konkatenáló, továbbá binárisokra / sztringekre alkalmazható műveletekkel van összerakva

Őrkifejezés, őr 2

- Őrkifejezésben tehát **nem** lehetnek
 - felhasználó által definiált függvények
 - a modulokban *function*-ként definiált függvények,mert
 - mellékhatásuk lehet,
 - a kiértékelésük hatékonysága nem garantálható,
 - kivételt dobhatnak.

Őrkifejezés, őr 2

- Őrkifejezésben tehát **nem** lehetnek
 - felhasználó által definiált függvények
 - a modulokban *function*-ként definiált függvények,mert
 - mellékhatásuk lehet,
 - a kiértékelésük hatékonysága nem garantálható,
 - kivételt dobhatnak.
- Példa: faktoriális számítása

Őrkifejezés, őr 2

- Őrkifejezésben tehát **nem** lehetnek
 - felhasználó által definiált függvények
 - a modulokban *function*-ként definiált függvények,

mert

- mellékhatásuk lehet,
 - a kiértékelésük hatékonysága nem garantálható,
 - kivételt dobhatnak.
- Példa: faktoriális számítása
A rekurzió soha nem áll le, ha n lebegőpontos vagy negatív:

```
def fac(0), do: 1  
def fac(n), do: n * fac(n-1)
```

Örkifejezés, ór 2

- Örkefejezésben tehát **nem** lehetnek
 - felhasználó által definiált függvények
 - a modulokban *function*-ként definiált függvények,

mert

- mellékhatásuk lehet,
 - a kiértékelésük hatékonysága nem garantálható,
 - kivételt dobhatnak.
- Példa: faktoriális számítása
A rekurzió soha nem áll le, ha n lebegőpontos vagy negatív:

```
def fac(0), do: 1
def fac(n), do: n * fac(n-1)
```

fac_2 kivételt dob, ha n nem egész, vagy ha negatív:

```
def fac_2(0), do: 1
def fac_2(n) when (is_integer n) and n > 0, do: n * fac_2(n-1)
```

Örkifejezés, ör 2

- Örkefejezésben tehát **nem** lehetnek
 - felhasználó által definiált függvények
 - a modulokban *function*-ként definiált függvények,

mert

- mellékhatásuk lehet,
 - a kiértékelésük hatékonysága nem garantálható,
 - kivételt dobhatnak.
- Példa: faktoriális számítása
A rekurzió soha nem áll le, ha n lebegőpontos vagy negatív:

```
def fac(0), do: 1
def fac(n), do: n * fac(n-1)
```

fac_2 kivételt dob, ha n nem egész, vagy ha negatív:

```
def fac_2(0), do: 1
def fac_2(n) when (is_integer n) and n > 0, do: n * fac_2(n-1)

iex> Fpea.fac_2 -1
** (FunctionClauseError) no function clause matching in ...
```

II. rész

Függvény, névtelen függvény, magasabb rendű függvény

- 1 Kifejezés, őrkifejezés
- 2 Függvény, névtelen függvény, magasabb rendű függvény

Tartalom

2

Függvény, névtelen függvény, magasabb rendű függvény

- FPE-4 – Függvény, névtelen függvény
- FPE-4 – Magasabb rendű függvény

Függvény definiálása

- Névtelen és neves függvények definiálásával már eddig is sokszor találkoztunk – nehéz lenne úgy beszélni a funkcionális programozásról, hogy kezdettől fogva ne legyen szó függvényekről és listákról.
- Most igyekszünk mindent elmondani előbb a *névtelen*, majd a *neves* függvények definiálásáról, amit csak tudni érdemes.
- A névtelen függvényt lambdának, lambda-függvénynek is nevezik. A λ -jelölést matematikai függvények tulajdonságainak vizsgálatához vezette be Alonzo Church az 1930-as években.

Névtelen függvény 1 (dp_fun.exs)

- A névtelen függvény, másnéven *lambda* definíciója az `fn` kulcsszóval kezdődik, egy vagy több *klózból* áll, és az `end` kulcsszóval fejeződik be:

```
fn
```

```
  klóz
```

```
  klóz ...
```

```
end
```

Névtelen függvény 1 (dp_fun.exs)

- A névtelen függvény, másnéven *lambda* definíciója az `fn` kulcsszóval kezdődik, egy vagy több *klózból* áll, és az `end` kulcsszóval fejeződik be:

```
fn
```

```
  klóz
```

```
  klóz ...
```

```
end
```

- A klóz *fejből* és *törzsből* áll, a kettőt `->` választja el:

```
klózfej -> klóztörzs
```

Névtelen függvény 1 (dp_fun.exs)

- A névtelen függvény, másnéven *lambda* definíciója az `fn` kulcsszóval kezdődik, egy vagy több *klózból* áll, és az `end` kulcsszóval fejeződik be:

```
fn  
  klóz  
  klóz ...  
end
```
- A klóz *fejből* és *törzsből* áll, a kettőt `->` választja el:

```
klózfej -> klóztörzs
```
- A klózfej *paraméterlistából* (mintakifejezésből) és opcionálisan *örkifejezésből* áll (A szögletes zárójelek itt az opcionálisat jelölik):

```
paraméterlista [when örkifejezés]
```
- A klóztörzs tetszőleges szekvenciális kifejezés.

Névtelen függvény 1 (dp_fun.exs)

- A névtelen függvény, másnéven *lambda* definíciója az `fn` kulcsszóval kezdődik, egy vagy több *klózból* áll, és az `end` kulcsszóval fejeződik be:

```
fn
  klóz
  klóz ...
end
```

- A klóz *fejből* és *törzsből* áll, a kettőt `->` választja el:

```
klózfej -> klóztörzs
```

- A klózfej *paraméterlistából* (mintakifejezésből) és opcionálisan *örkifejezésből* áll (A szögletes zárójelek itt az opcionálisat jelölik):

```
paraméterlista [when örkkifejezés]
```

- A klóztörzs tetszőleges szekvenciális kifejezés.

- Példa:

```
stir = # Hogy hivatkozni tudjunk a névtelen fv-re, névhez kell kötni
fn {a, b} when a < b           -> {b, a}
  {a, b} when rem(a,b) == 0 -> d = div(a,b); {b * d, d}
  x                           -> x
end
```

Névtelen függvény 2 (dp_fun.exs)

- A második klóz törzsében egy szekvenciális kifejezés van, amit szabad, de – ellentétben a neves függvénnyel – nem kell zárójelbe tenni:

```
... {a, b} when rem(a,b) == 0 -> (d = div(a,b); {d * b, d})  
    x                          -> x ... # x mindenre illeszkedik
```

Névtelen függvény 2 (dp_fun.exs)

- A második klóz törzsében egy szekvenciális kifejezés van, amit szabad, de – ellentétben a neves függvénnyel – nem kell zárójelbe tenni:

```
... {a, b} when rem(a,b) == 0 -> (d = div(a,b); {d * b, d})  
    x                          -> x ... # x mindenre illeszkedik
```

- A most definiált névtelen függvény túl bonyolult, rendszerint nagyon egyszerűket és rövideket definálunk „menet közben” (on-the-fly).
- Névtelen függvényt tipikusan két esetben használunk: akkor, amikor
 - egy függvénydefinícióban lokális függvényre van szükség,
 - egy függvény paraméterként egy egyszerűbb, saját függvényt vár.

Névtelen függvény 2 (dp_fun.exs)

- A második klóz törzsében egy szekvenciális kifejezés van, amit szabad, de – ellentétben a neves függvénnyel – nem kell zárójelbe tenni:

```
... {a, b} when rem(a,b) == 0 -> (d = div(a,b); {d * b, d})
      x                               -> x ... # x mindenre illeszkedik
```

- A most definiált névtelen függvény túl bonyolult, rendszerint nagyon egyszerűket és rövideket definálunk „menet közben” (on-the-fly).
- Névtelen függvényt tipikusan két esetben használunk: akkor, amikor
 - egy függvénydefinícióban lokális függvényre van szükség,
 - egy függvény paraméterként egy egyszerűbb, saját függvényt vár.
- Példák:

```
def sumsq(x, y), do: ( sq = fn x -> x*x end; sq.(x) + sq.(y) ) # Ott a.!
iex> Dp.Fun.sumsq(3, 5)
34
def sumtr(f, x, y), do: f.(x) + f.(y) # Ott a.!
iex> Dp.Fun.sumtr(fn x -> 2 * x + 1 end, 3, 5)
18
```

Névtelen függvény 3 (dp_fun.exs)

- Amikor változónevet függvényként hívunk meg, **ponttal kell elválasztani** a paraméterektől, amiket **zárójelbe kell rakni**.
- A lokális függvény paraméterei és más változói lokálisak, azokat a befogadó függvény sem látja.
- A lokális függvény használhatja az őt befogadó függvény paramétereit és más változóit.

Névtelen függvény 3 (dp_fun.exs)

- Amikor változónevet függvényként hívunk meg, **ponttal kell elválasztani** a paraméterektől, amiket **zárójelbe kell rakni**.
- A lokális függvény paraméterei és más változói lokálisak, azokat a befogadó függvény sem látja.
- A lokális függvény használhatja az őt befogadó függvény paramétereit és más változóit.
- Példák:

```
def sumx2y2z(x, y, z), do:  
  ( sq = fn x -> x*x end;  
    sq.(x) + sq.(y) + z  
  )  
iex> Dp.Fun.sumx2y2z(3, 5, 6)  
40
```

Névtelen függvény 3 (dp_fun.exs)

- Amikor változónevet függvényként hívunk meg, **ponttal kell elválasztani** a paraméterektől, amiket **zárójelbe kell rakni**.
- A lokális függvény paraméterei és más változói lokálisak, azokat a befogadó függvény sem látja.
- A lokális függvény használhatja az őt befogadó függvény paramétereit és más változóit.
- Példák:

```
def sumx2y2z(x, y, z), do:  
  ( sq = fn x -> x*x end;  
    sq.(x) + sq.(y) + z  
  )  
iex> Dp.Fun.sumx2y2z(3, 5, 6)  
40
```

- A névtelen függvény nem rekurzív, azaz nem hívhatja önmagát.
- A kis névtelen függvények használata annyira gyakori, hogy az Elixirben *rövidítő jelölés* van a definiálásukra.

Névtelen függvény 4 (dp_fun.exs)

- Nézzük az előbbi két példát a *rövidítő jelöléssel*, továbbá a második példa egy változatát, amelyben a névtelen függvénynek három paramétere van:

```
def sumsq(x, y), do: (sq = fn x -> x * x end; sq.(x) + sq.(y))
def sumsq_2(x, y), do: (sq = &(&1 * &1); sq.(x) + sq.(y)) # kell a pont!
iex> Dp.Fun.sumsq_2(3, 5)
34
def sumtr(f, x, y), do: f.(x) + f.(y) # kell a pont!
iex> Dp.Fun.sumtr(fn x -> 2 * x + 1 end, 3, 5)
iex> Dp.Fun.sumtr(&(2 * &1 + 1), 3, 5)
18
```

Névtelen függvény 4 (dp_fun.exs)

- Nézzük az előbbi két példát a *rövidítő jelöléssel*, továbbá a második példa egy változatát, amelyben a névtelen függvénynek három paramétere van:

```
def sumsq(x, y), do: (sq = fn x -> x * x end; sq.(x) + sq.(y))
def sumsq_2(x, y), do: (sq = &(&1 * &1); sq.(x) + sq.(y)) # kell a pont!
iex> Dp.Fun.sumsq_2(3, 5)
34
def sumtr(f, x, y), do: f.(x) + f.(y) # kell a pont!
iex> Dp.Fun.sumtr(fn x -> 2 * x + 1 end, 3, 5)
iex> Dp.Fun.sumtr(&(2 * &1 + 1), 3, 5)
18
iex> sumtr_2 = &(&1.(&2) + &1.(&3)) # kell a pont!
iex> sumtr_2.(&(2 * &1 + 1), 3, 5) # kell a pont!
18
```

Névtelen függvény 4 (dp_fun.exs)

- Nézzük az előbbi két példát a *rövidítő jelöléssel*, továbbá a második példa egy változatát, amelyben a névtelen függvénynek három paramétere van:

```
def sumsq(x, y), do: (sq = fn x -> x * x end; sq.(x) + sq.(y))
def sumsq_2(x, y), do: (sq = &(&1 * &1); sq.(x) + sq.(y)) # kell a pont!
iex> Dp.Fun.sumsq_2(3, 5)
```

34

```
def sumtr(f, x, y), do: f.(x) + f.(y) # kell a pont!
```

```
iex> Dp.Fun.sumtr(fn x -> 2 * x + 1 end, 3, 5)
```

```
iex> Dp.Fun.sumtr(&(2 * &1 + 1), 3, 5)
```

18

```
iex> sumtr_2 = &(&1.(&2) + &1.(&3)) # kell a pont!
```

```
iex> sumtr_2.(&(2 * &1 + 1), 3, 5) # kell a pont!
```

18

- A névtelen függvényt változóhoz sem kell kötni, úgy is alkalmazható:

```
iex> (&(&1.(&2) + &1.(&3))).(&(2 * &1 + 1), 3, 5) # kell a pont!
```

18

Neves függvény 1 (dp_fun.exs)

- Neves függvényt csak modulban lehet definiálni `def`-fel vagy `defp`-vel.
- A `defp`-vel definiált függvény privát (azaz lokális), csak a modul más függvényeiből hívható meg, az adott modulon kívülről nem.
- Amikor egy modul függvényeiből hívunk meg olyan függvényeket, amelyek ugyanabban a modulban vannak definiálva, akkor a modul nevét nem kell – de szabad – kiírni a függvéynév elé.
- Amikor egy modulban függvénydefiníción kívül hivatkozunk egy függvényre, akkor mindig ki kell írni a modulnevet a függvéynév elé, akkor is, ha a függvény ugyanabban a modulban van definiálva. A modulban nem függvénydefiníciók belsejében megadott kifejezéseket az Elixir már a fordítás során futtatja.
- A neves függvény definíciója is egy vagy több *klózból* áll. Minden klóznak vagy `def`-fel, vagy `defp`-vel kell kezdődnie:

```
def klóz  
def klóz ...
```

Neves függvény 2 (dp_fun.exs)

- A klóz itt is *fejből* és *törzsből* áll.
- A klózfej *névből*, *paraméterlistából* – mintakifejezésekből – és opcionálisan *örkifejezésből* áll¹. A paraméterlistát az egyértelműség kedvéért sokszor zárójelbe kell rakni²:

```
név [ ( ) | ( ) paraméterlista ( ) ] [when örkiifejezés]
```

- A paraméterek közül az utolsó opcionális, azaz elhagyható, ha `\\` jelekkel elválasztva megadjuk az alapértelmezett értéket.

```
def inspect(item, options \\ [])
```

¹A `[` és `]` ebben a jelölésben az opcionálisat jelölik, a `|` pedig az alternatívát.

²A *név* és a nyitó zárójel között nem lehet szóköz!

³A pontosvesszővel elválasztott vagy új sorba írt részkifejezéseket zárójelbe kell rakni.

Neves függvény 2 (dp_fun.exs)

- A klóz itt is *fejből* és *törzsből* áll.
- A klózfej *névből*, *paraméterlistából* – mintakifejezésekből – és opcionálisan *örkifejezésből* áll¹. A paraméterlistát az egyértelműség kedvéért sokszor zárójelbe kell rakni²:

```
név [ ( ) | ( [paraméterlista ] ) ] [when örkiyejezés ]
```

- A paraméterek közül az utolsó opcionális, azaz elhagyható, ha \\ jelekkel elválasztva megadjuk az alapértelmezett értéket.

```
def inspect(item, options \\ [ ])
```

- A klóztörzs a *do*:-val bevezetett tetszőleges szekvenciális kifejezés³:

```
do: [ ( [szekvenciális kifejezés ] ) ]
```

- A klóztörzs alternatív leírási módja a *do*...*end* pár használata, ilyenkor a *do* elé *nem szabad* vesszőt rakni, a kerek zárójelek pedig elhagyhatók:

```
do szekvenciális kifejezés end
```

- Egy függvény klózai közé más függvénydefiníciót nem szabad beékelni.

¹A [és] ebben a jelölésben az opcionálisat jelölik, a | pedig az alternatívát.

²A *név* és a *nyitó zárójel között nem lehet szóköz*!

³A pontosvesszővel elválasztott vagy új sorba írt részkifejezéseket zárójelbe kell rakni.

Neves függvény 3 (dp_fun.exs)

- A függvényt a neve és paramétereinek száma (az ún. aritása) azonosítja. Ugyanazzal a névvel több különböző aritású függvény definiálható.
- A neves függvények lehetnek rekurzívak (a névtelek nem).
- Oldjuk meg a következő tanulságos feladatot!
Válogassuk ki egy `xs` listából a *d*-vel *osztható* egészeket, rendre *szorozzuk meg őket d*-vel, *(d+1)*-gyel stb., majd az így kapott elemeket adjuk vissza egy *csökkenő sorrendű* listában.

Neves függvény 3 (dp_fun.exs)

- A függvényt a neve és paramétereinek száma (az ún. aritása) azonosítja. Ugyanazzal a névvel több különböző aritású függvény definiálható.
- A neves függvények lehetnek rekurzívak (a névtelek nem).
- Oldjuk meg a következő tanulságos feladatot!
Válogassuk ki egy `xs` listából a *d*-vel *osztható* egészeket, rendre *szorozzuk meg őket d*-vel, *(d+1)*-gyel stb., majd az így kapott elemeket adjuk vissza egy *csökkenő sorrendű* listában.
- Tudnunk kell, hogy a feltételnek megfelelő elemek közül hányadiknál járunk, mert egyesével növekvő egészekkel kell megszoroznunk.
- Ehhez egy segédfüggvényre és egy plusz paraméterre van szükségünk, hiszen globális változók nincsenek az Elixirben. A segédfüggvényt az új paraméter kezdőértékével, *d*-vel hívjuk meg:

```
def picked(xs, d), do: (picked xs, d, d)
```

Neves függvény 3 (dp_fun.exs)

- A függvényt a neve és paramétereinek száma (az ún. aritása) azonosítja. Ugyanazzal a névvel több különböző aritású függvény definiálható.
- A neves függvények lehetnek rekurzívak (a névtelek nem).
- Oldjuk meg a következő tanulságos feladatot!
Válogassuk ki egy `xs` listából a *d*-vel *osztható* egészeket, rendre *szorozzuk meg őket d*-vel, *(d+1)-gyel stb.*, majd az így kapott elemeket adjuk vissza egy *csökkenő sorrendű* listában.
- Tudnunk kell, hogy a feltételnek megfelelő elemek közül hányadiknál járunk, mert egyesével növekvő egészekkel kell megszoroznunk.
- Ehhez egy segédfüggvényre és egy plusz paraméterre van szükségünk, hiszen globális változók nincsenek az Elixirben. A segédfüggvényt az új paraméter kezdőértékével, *d*-vel hívjuk meg:

```
def picked(xs, d), do: (picked xs, d, d)
```
- Mind a három lehetséges esetre definiálunk egy-egy klózt:
 - 1 a soron következő elem egész szám és *d*-vel osztható,
 - 2 a soron következő elem nem egész szám vagy nem osztható *d*-vel,
 - 3 a lista üres.

Neves függvény 4 (dp_fun.exs)

- A három klózból álló privát függvény:

```
defp picked([x|xs], d, i) when (is_integer x) and (rem x,d) === 0 do  
  [ x * i | picked(xs, d, i+1) ]  
end
```

Neves függvény 4 (dp_fun.exs)

- A három klózból álló privát függvény:

```
defp picked([x|xs], d, i) when (is_integer x) and (rem x,d) === 0 do
  [ x * i | picked(xs, d, i+1) ]
end
defp picked([_x|xs], d, i), do: picked(xs, d, i)
defp picked([], _d, _i), do: []
```

- Az 1. és a 2. klóz sorrendje fontos, fordított sorrendben minden listaelemet eldobna, a d-vel osztható egészeket is.

Neves függvény 4 (dp_fun.exs)

- A három klózból álló privát függvény:

```
defp picked([x|xs], d, i) when (is_integer x) and (rem x,d) === 0 do
  [ x * i | picked(xs, d, i+1) ]
end
defp picked([_x|xs], d, i), do: picked(xs, d, i)
defp picked([], _d, _i), do: []
```

- Az 1. és a 2. klóz sorrendje fontos, fordított sorrendben minden listaelemet eldobna, a d-vel osztható egészeket is.
- Az 1. klózban listát építünk: az eredménylista feje a soron következő elem (x) és a számláló (i) szorzata lesz, az eredménylista farkát pedig úgy kapjuk meg, hogy a picked függvényt rekurzívan meghívjuk a soron következő elem utáni maradéklistára, az i számlálót pedig megnöveljük.

Neves függvény 4 (dp_fun.exs)

- A három klózból álló privát függvény:

```
defp picked([x|xs], d, i) when (is_integer x) and (rem x,d) == 0 do
  [ x * i | picked(xs, d, i+1) ]
end
defp picked([_x|xs], d, i), do: picked(xs, d, i)
defp picked([], _d, _i), do: []
```

- Az 1. és a 2. klóz sorrendje fontos, fordított sorrendben minden listaelemet eldobna, a d-vel osztható egészeket is.
- Az 1. klózban listát építünk: az eredménylista feje a soron következő elem (x) és a számláló (i) szorzata lesz, az eredménylista farkát pedig úgy kapjuk meg, hogy a picked függvényt rekurzívan meghívjuk a soron következő elem utáni maradéklistára, az i számlálót pedig megnöveljük.
- Az 1. és a 2. klózban minden rekurzív lépésben eggyel rövidül a maradéklista. Mivel a lista korlátos, előbb-utóbb eljutunk az üres listáig, ami a rekurzió végét jelenti.

Neves függvény 4 (dp_fun.exs)

- A három klózból álló privát függvény:

```
defp picked([x|xs], d, i) when (is_integer x) and (rem x,d) == 0 do
  [ x * i | picked(xs, d, i+1) ]
end
defp picked([_x|xs], d, i), do: picked(xs, d, i)
defp picked([], _d, _i), do: []
```

- Az 1. és a 2. klóz sorrendje fontos, fordított sorrendben minden listaelemet eldobna, a d-vel osztható egészeket is.
- Az 1. klózban listát építünk: az eredménylista feje a soron következő elem (x) és a számláló (i) szorzata lesz, az eredménylista farkát pedig úgy kapjuk meg, hogy a picked függvényt rekurzívan meghívjuk a soron következő elem utáni maradéklistára, az i számlálót pedig megnöveljük.
- Az 1. és a 2. klózban minden rekurzív lépésben eggyel rövidül a maradéklista. Mivel a lista korlátos, előbb-utóbb eljutunk az üres listáig, ami a rekurzió végét jelenti.
- Az eredménylista nincs rendezve! Rendezzük csökkenő sorrendben:

```
def picked(xs, d), do: Enum.sort((picked xs, d, 0), &>/2)
```

Neves függvény 4 (dp_fun.exs)

- A három klózból álló privát függvény:

```
defp picked([x|xs], d, i) when (is_integer x) and (rem x,d) == 0 do
  [ x * i | picked(xs, d, i+1) ]
end
defp picked([_x|xs], d, i), do: picked(xs, d, i)
defp picked([], _d, _i), do: []
```

- Az 1. és a 2. klóz sorrendje fontos, fordított sorrendben minden listaelemet eldobna, a d-vel osztható egészeket is.
- Az 1. klózban listát építünk: az eredménylista feje a soron következő elem (x) és a számláló (i) szorzata lesz, az eredménylista farkát pedig úgy kapjuk meg, hogy a picked függvényt rekurzívan meghívjuk a soron következő elem utáni maradéklistára, az i számlálót pedig megnöveljük.
- Az 1. és a 2. klózban minden rekurzív lépésben eggyel rövidül a maradéklista. Mivel a lista korlátos, előbb-utóbb eljutunk az üres listáig, ami a rekurzió végét jelenti.
- Az eredménylista nincs rendezve! Rendezzük csökkenő sorrendben:

```
def picked(xs, d), do: Enum.sort((picked xs, d, 0), &>/2)
```

Neves függvény 5 (dp_fun.exs)

- Nézzünk néhány tesztesetet, futási eredményt:

```
iex> Dp.Fun.picked([3, 5, 6, 9, 0], 3)
```

```
[45, 24, 9, 0]
```

```
iex> Dp.Fun.picked('abcdefgh', 2)
```

```
[520, 408, 300, 196]
```

```
iex> Dp.Fun.picked([:b, [], '', "", {}, %{}], 4), 4)
```

```
[16]
```

```
iex> Dp.Fun.picked(Enum.to_list(2..36 // 6), 4)
```

```
[192, 100, 32]
```

Neves függvény 5 (dp_fun.exs)

- Nézzünk néhány tesztet, futási eredményt:

```
iex> Dp.Fun.picked([3, 5, 6, 9, 0], 3)
```

```
[45, 24, 9, 0]
```

```
iex> Dp.Fun.picked('abcdefgh', 2)
```

```
[520, 408, 300, 196]
```

```
iex> Dp.Fun.picked([:b, []], ' ', '"', '{}', %{}, 4), 4)
```

```
[16]
```

```
iex> Dp.Fun.picked(Enum.to_list(2..36 // 6), 4)
```

```
[192, 100, 32]
```

- picked azt csinálja, amit várunk tőle. Jó tulajdonsága, hogy csak egyszer gyalogol végig a listán. A szerkezete azonban nem elég tiszta, nem eléggé érthető.
- A feladat tulajdonképpen három részfeladatból áll:
 - 1 a megfelelő elemek – a d-vel osztható egészek – kigyűjtése a listából,
 - 2 a kigyűjtött elemek listáján a transzformáció elvégzése elemenként,
 - 3 az eredménylista rendezése.
- Az eredménylista rendezése eddig is külön lépés volt, nézzük most a másik kettőt!

Neves függvény 6 (dp_fun.exs)

- A d-vel osztható egészek kigyűjtése a listából picked/3-hoz nagyon hasonló, de eggyel kisebb az aritása, és nem végez transzformációt:

```
defp filt([x|xs], d) when (is_integer x) and (rem x,d) === 0 do
  [ x | filt(xs, d) ]
end
defp filt([_x|xs], d), do: filt(xs, d)
defp filt([], _d), do: []
filt([3, :a, 8, 6, 0], 3) === [3, 6, 0]
```

Neves függvény 6 (dp_fun.exs)

- A d-vel osztható egészek kigyűjtése a listából picked/3-hoz nagyon hasonló, de eggyel kisebb az aritása, és nem végez transzformációt:

```
defp filt([x|xs], d) when (is_integer x) and (rem x,d) === 0 do  
  [ x | filt(xs, d) ]  
end
```

```
defp filt([_x|xs], d), do: filt(xs, d)
```

```
defp filt([], _d), do: []
```

```
filt([3, :a, 8, 6, 0], 3) === [3, 6, 0]
```

- Ha egyszer filt/2-vel megszűrtük a listát, az új lista minden elemét lehet transzformálni, miközben nyilvántartjuk, hányadik elemnél tartunk:

```
defp mult([x|xs], i), do: [x * i | mult(xs, i+1)]
```

```
defp mult([], _i), do: []
```

```
mult([3, 6, 0], 2) === [6, 18, 0]
```

Neves függvény 6 (dp_fun.exs)

- A d-vel osztható egészek kigyűjtése a listából picked/3-hoz nagyon hasonló, de eggyel kisebb az aritása, és nem végez transzformációt:

```
defp filt([x|xs], d) when (is_integer x) and (rem x,d) === 0 do
  [ x | filt(xs, d) ]
end
defp filt([_x|xs], d), do: filt(xs, d)
defp filt([], _d), do: []
filt([3, :a, 8, 6, 0], 3) === [3, 6, 0]
```
- Ha egyszer filt/2-vel megszűrtük a listát, az új lista minden elemét lehet transzformálni, miközben nyilvántartjuk, hányadik elemnél tartunk:

```
defp mult([x|xs], i), do: [x * i | mult(xs, i+1)]
defp mult([], _i), do: []
mult([3, 6, 0], 2) === [6, 18, 0]
```

- Már csak kombinálni kell filt/2-t és mult/2-t Enum.sort/2-vel:

```
def picked_2(xs, d), do: Enum.sort(mult(filt(xs, d), d), &>/2)
iex> Dp.Fun.picked_2(Enum.to_list(2..36 // 6), 4)
[192, 100, 32]
```

Neves függvény 6 (dp_fun.exs)

- A d-vel osztható egészek kigyűjtése a listából picked/3-hoz nagyon hasonló, de eggyel kisebb az aritása, és nem végez transzformációt:

```
defp filt([x|xs], d) when (is_integer x) and (rem x,d) === 0 do
  [ x | filt(xs, d) ]
end
defp filt([_x|xs], d), do: filt(xs, d)
defp filt([], _d), do: []
filt([3, :a, 8, 6, 0], 3) === [3, 6, 0]
```
- Ha egyszer filt/2-vel megszűrtük a listát, az új lista minden elemét lehet transzformálni, miközben nyilvántartjuk, hányadik elemnél tartunk:

```
defp mult([x|xs], i), do: [x * i | mult(xs, i+1)]
defp mult([], _i), do: []
mult([3, 6, 0], 2) === [6, 18, 0]
```

- Már csak kombinálni kell filt/2-t és mult/2-t Enum.sort/2-vel:

```
def picked_2(xs, d), do: Enum.sort(mult(filt(xs, d), d), &>/2)
iex> Dp.Fun.picked_2(Enum.to_list(2..36 // 6), 4)
[192, 100, 32]
```
- Csak három függvényt kombináltunk, mégis nehezen olvasható a kód, mert sok a zárójel, és mert belülről kifelé haladva kell olvasni.

Neves függvény 7: |> operátorral (dp_fun.exs)

- Segédváltozók bevezetésével átláthatóbbá, érthetőbbé tehetjük:

```
def picked_3(xs, d) do
  fs = filt(xs, d)
  ms = mult(fs, d)
  Enum.sort(ms, &>/2)
end
```

Neves függvény 7: |> operátorral (dp_fun.exs)

- Segédváltozók bevezetésével átláthatóbbá, érthetőbbé tehetjük:

```
def picked_3(xs, d) do
  fs = filt(xs, d)
  ms = mult(fs, d)
  Enum.sort(ms, &>/2)
end
```

- *A funkcionális programokra jellemző, hogy sok kis függvény egyszerű transzformációt végez, melyhez adatokat vesz át más függvénytől, majd ad át más függvénynek.*
- A |> (pipe) operátorral elkerülhetjük az egymásba skatulyázást és új változók bevezetését:

```
def picked_4(xs, d) do
  xs
  |> filt(d)
  |> mult(d)
  |> Enum.sort(&>/2)
end

iex> Dp.Fun.picked_4(Enum.to_list(2..36 // 6), 4)
[192, 100, 32]
```

Neves függvény 8: |> operátorral (dp_fun.exs)

- A |> operátorral még akkor is jól olvasható a kód, ha egyetlen sorba írjuk a transzformációsorozatot:

```
def picked_4(xs, d), do: xs |> filt(d) |> mult(d) |> Enum.sort(&>/2)
```

Neves függvény 8: |> operátorral (dp_fun.exs)

- A |> operátorral még akkor is jól olvasható a kód, ha egyetlen sorba írjuk a transzformációsorozatot:

```
def picked_4(xs, d), do: xs |> filt(d) |> mult(d) |> Enum.sort(&>/2)
```

- Az infix |> operátor a bal oldali kifejezés értékét adja át a jobb oldali függvénynek első paraméterként – így azt nem kell, *nem is szabad* kiírni
- A |> operátor jobb oldalán a függvény további paramétereit vagy a teljes függvényhívást zárójelbe kell rakni, mert a |> mohó kiértékelésű. A fenti sor így is zárójelezhető:

```
... do: xs |> (filt d) |> (mult d) |> (Enum.sort &>/2)
```

Neves függvény 8: `|>` operátorral (`dp_fun.exs`)

- A `|>` operátorral még akkor is jól olvasható a kód, ha egyetlen sorba írjuk a transzformációsorozatot:

```
def picked_4(xs, d), do: xs |> filt(d) |> mult(d) |> Enum.sort(&>/2)
```

- Az infix `|>` operátor a bal oldali kifejezés értékét adja át a jobb oldali függvénynek első paraméterként – így azt nem kell, *nem is szabad* kiírni
- A `|>` operátor jobb oldalán a függvény további paramétereit vagy a teljes függvényhívást zárójelbe kell rakni, mert a `|>` mohó kiértékelésű. A fenti sor így is zárójelezhető:

```
... do: xs |> (filt d) |> (mult d) |> (Enum.sort &>/2)
```

- Térjünk vissza még a `filt/2`, majd a `mult/2` függvényhez!
- Mindkettő túl speciális: az elsőbe a szűrőfeltétel, a másodikba az elvégzendő művelet van „gránitszilárdsággal” beépítve. Ha ezeket függvényparaméterként adnánk át, rugalmasabb lenne a megoldásunk.

Neves függvény 9: `filt` újra (`dp_fun.exs`)

- Ez itt a `filt/2` első, specifikus verziója:

```
defp filt([x|xs], d) when (is_integer x) and (rem x,d) === 0 do
  [ x | filt(xs, d) ]
end
defp filt([_x|xs], d), do: filt(xs, d)
defp filt([], _d), do: []
```

Neves függvény 9: `filt` újra (`dp_fun.exs`)

- Ez itt a `filt/2` első, specifikus verziója:

```
defp filt([x|xs], d) when (is_integer x) and (rem x,d) === 0 do
  [ x | filt(xs, d) ]
end
defp filt([_x|xs], d), do: filt(xs, d)
defp filt([], _d), do: []
```

- Az őrt cseréljük le függvényhívásra – a függvényt paraméterként adjuk át:

```
defp filt_2([x|xs], f?) when f?(x), do: [ x | filt_2(xs, f?) ]
defp filt_2([_x|xs], f?), do: filt_2(xs, f?)
defp filt_2([], _f?), do: []
```

Az `f?` névben a kérdőjellel arra utalunk, hogy a függvény igazságértéket ad eredményül – azaz predikátum.

Neves függvény 9: `filt` újra (`dp_fun.exs`)

- Ez itt a `filt`/2 első, specifikus verziója:

```
defp filt([x|xs], d) when (is_integer x) and (rem x,d) === 0 do
  [ x | filt(xs, d) ]
end
defp filt([_x|xs], d), do: filt(xs, d)
defp filt([], _d), do: []
```

- Az őrt cseréljük le függvényhívásra – a függvényt paraméterként adjuk át:

```
defp filt_2([x|xs], f?) when f?(x), do: [ x | filt_2(xs, f?) ]
defp filt_2([_x|xs], f?), do: filt_2(xs, f?)
defp filt_2([], _f?), do: []
```

Az `f?` névben a kérdőjellel arra utalunk, hogy a függvény igazságértéket ad eredményül – azaz predikátum.

- Van itt egy kis probléma: a `filt_2`/2 fordításakor az IEx hibát jelez
**** (CompileError) ...anonymous call is not allowed in guards.**
- Idézzük föl: a `when` őrkifejezést vezet be, ő csak könyvtári függvény lehet, azokból se mind. Az `f?` paraméterről nem tudható, mi lesz az aktuális értéke, ezért viselkedik ilyen elutasítóan az Elixir.

Neves függvény 10: `filt_2` és `oper` (`dp_fun.exs`)

- Örökifejezés helyett tehát őr nélküli *feltételes kifejezést* kell használnunk, összevonva a korábbi 1. és 2. klózt:

```
defp filt_2([x|xs], f?), do:  
  (if f?.(x), do: [ x | filt_2(xs, f?) ], else: filt_2(xs, f?))  
defp filt_2([], _f?), do: []
```

Neves függvény 10: `filt_2` és `oper` (`dp_fun.exs`)

- Örökifejezés helyett tehát ör nélküli *feltételes kifejezést* kell használnunk, összevonva a korábbi 1. és 2. klózt:

```
defp filt_2([x|xs], f?), do:  
  (if f?.(x), do: [ x | filt_2(xs, f?) ], else: filt_2(xs, f?))  
defp filt_2([], _f?), do: []
```

- `mult/2` helyett írjunk egy általánosabb `oper/2`-t:

```
defp oper([x|xs], f), do: [ f.(x) | oper(xs, f) ]  
defp oper([], _f), do: []
```

Neves függvény 10: `filt_2` és `oper` (`dp_fun.exs`)

- Örkiefejezés helyett tehát őr nélküli *feltételes kifejezést* kell használnunk, összevonva a korábbi 1. és 2. klózt:

```
defp filt_2([x|xs], f?), do:  
  (if f?.(x), do: [ x | filt_2(xs, f?) ], else: filt_2(xs, f?))  
defp filt_2([], _f?), do: []
```

- `mult/2` helyett írjunk egy általánosabb `oper/2`-t:

```
defp oper([x|xs], f), do: [ f.(x) | oper(xs, f) ]  
defp oper([], _f), do: []
```

Ez nem jó, mert így nem követjük, hányadik listaelemnél is tartunk, márpedig `x`-szel és folyton változó másik értékkel kell műveletet végezni.

Neves függvény 10: `filt_2` és `oper` (`dp_fun.exs`)

- Örkefejezés helyett tehát ör nélküli *feltételes kifejezést* kell használnunk, összevonva a korábbi 1. és 2. klózt:

```
defp filt_2([x|xs], f?), do:  
  (if f?.(x), do: [ x | filt_2(xs, f?) ], else: filt_2(xs, f?))  
defp filt_2([], _f?), do: []
```

- `mult/2` helyett írjunk egy általánosabb `oper/2`-t:

```
defp oper([x|xs], f), do: [ f.(x) | oper(xs, f) ]  
defp oper([], _f), do: []
```

Ez nem jó, mert így nem követjük, hányadik listaelemnél is tartunk, márpedig `x`-szel és folyton változó másik értékkel kell műveletet végezni.

- `oper/2` második paramétere legyen akkor az `{f, i}` pár:

```
defp oper([x|xs], {f, i}), do: [ f.(x, i) | oper(xs, {f, i+1}) ]  
defp oper([], _fi), do: []
```

Neves függvény 10: `filt_2` és `oper` (`dp_fun.exs`)

- Örktifejezés helyett tehát ör nélküli *feltételes kifejezést* kell használnunk, összevonva a korábbi 1. és 2. klózt:

```
defp filt_2([x|xs], f?), do:
  (if f?.(x), do: [ x | filt_2(xs, f?) ], else: filt_2(xs, f?))
defp filt_2([], _f?), do: []
```

- `mult/2` helyett írjunk egy általánosabb `oper/2`-t:

```
defp oper([x|xs], f), do: [ f.(x) | oper(xs, f) ]
defp oper([], _f), do: []
```

Ez nem jó, mert így nem követjük, hányadik listaelemnél is tartunk, márpedig `x`-szel és folyton változó másik értékkel kell műveletet végezni.

- `oper/2` második paramétere legyen akkor az `{f, i}` pár:

```
defp oper([x|xs], {f, i}), do: [ f.(x, i) | oper(xs, {f, i+1}) ]
defp oper([], _fi), do: []
```

Ez sem elég általános; később megnézzük, lehet-e általánosabb megoldást írni ilyen jellegű feladatokra.

- A következő dián osszerakjuk a teljes megoldást az új részmegoldásokkal.

Neves függvény 11: `filt_2` és `oper` (`dp_fun.exs`)

- A teljes megoldás új verziója az új részmegoldásokkal

```
def picked_5(xs, d) do
  f? = &(is_integer &1) and (rem &1, d) === 0
  g  = &(&1 * &2)
  xs
  |> filt_2(f?)
  |> oper(g, d)
  |> Enum.sort(&>/2)
end
```

Neves függvény 11: `filt_2` és `oper` (`dp_fun.exs`)

- A teljes megoldás új verziója az új részmegoldásokkal

```
def picked_5(xs, d) do
  f? = &(is_integer &1) and (rem &1, d) === 0
  g  = &(&1 * &2)
  xs
  |> filt_2(f?)
  |> oper(g, d)
  |> Enum.sort(&>/2)
end
iex> Dp.Fun.picked_5(Enum.to_list(2..36 // 6), 4)
[192, 100, 32]
```

Neves függvény 11: `filt_2` és `oper` (`dp_fun.exs`)

- A teljes megoldás új verziója az új részmegoldásokkal

```
def picked_5(xs, d) do
  f? = &(is_integer &1) and (rem &1, d) === 0
  g  = &(&1 * &2)
  xs
  |> filt_2(f?)
  |> oper(g, d)
  |> Enum.sort(&>/2)
end
iex> Dp.Fun.picked_5(Enum.to_list(2..36 // 6), 4)
[192, 100, 32]
```

Itt két névtelen két segédfüggvényt definiáltuk lokálisan, az `f?` és az `f` névhez kötve őket, de egy újabb verziójú `picked` függvény paraméterei is lehetnének. Ezzel a megoldásunk még flexibilisebb lenne.

- Az olyan függvényt, amelynek a paraméterei között függvényérték is van, *magasabb rendű* függvénynek hívjuk. Függvény eredménye is lehet függvényérték, ilyenkor is magasabb rendű függvényekről beszélünk.
- Következő témánk a magasabb rendű függvények.

Tartalom

2

Függvény, névtelen függvény, magasabb rendű függvény

- FPE-4 – Függvény, névtelen függvény
- FPE-4 – Magasabb rendű függvény

Magasabb rendű függvények: `map/2` és `filter/2` (`dp_fun.exs`)

- Magasabb rendű függvény: paramétere vagy eredménye függvény.
- A magasabb rendű függvények általánosított eszközök bonyolultabb feladatok egyszerű megoldására.
- *Magasabb rendű függvények használatával a rekurzió többnyire rejtve lesz csak jelen a programjainkban.*

Magasabb rendű függvények: `map/2` és `filter/2` (`dp_fun.exs`)

- Magasabb rendű függvény: paramétere vagy eredménye függvény.
- A magasabb rendű függvények általánosított eszközök bonyolultabb feladatok egyszerű megoldására.
- *Magasabb rendű függvények használatával a rekurzió többnyire rejtve lesz csak jelen a programjainkban.*
- Az előbb definiált `oper/2` és `filt/2` a `map/2`, ill. a `filter/2` „előfutára”.
- `map/2` egy lista elemeit transzformálja, `filter/2` szűréssel kiválogatja az adott feltételnek megfelelő elemeket a listából:

```
def map([x|xs], f), do: [ f.(x) | map(xs, f) ]
def map([], _f),      do: []

def filter([x|xs], p?), do:
  (if p?.(x), do: [ x | filter(xs, p?) ], else: filter(xs, p?))
def filter([], _p?),    do: []
```

- „Hivatalos” verziójuk az Elixirben: `Enum.map/2` és `Enum.filter/2`, tetszőleges felsorolható kollekcióra alkalmazhatók.

Egyszerű példák `map/2`-vel és `filter/2`-vel

```
iex> Enum.map([ "alma", "korte" ], &String.length/1)
[4, 5]
iex>
```

Egyszerű példák map/2-vel és filter/2-vel

```
iex> Enum.map([ "alma", "korte" ], &String.length/1)
[4, 5]
iex> Enum.map([ [10, 20], [10, 20, 30] ], &Enum.sum/1)
[30, 60]
iex>
```

Egyszerű példák map/2-vel és filter/2-vel

```
iex> Enum.map([ "alma", "korte" ], &String.length/1)
[4, 5]
iex> Enum.map([ [10, 20], [10, 20, 30] ], &Enum.sum/1)
[30, 60]
iex> for s <- ["alma", "korte"], do: String.length(s) #map helyett
[4, 5]
```

Egyszerű példák map/2-vel és filter/2-vel

```
iex> Enum.map([ "alma", "korte" ], &String.length/1)
[4, 5]
iex> Enum.map([ [10, 20], [10, 20, 30] ], &Enum.sum/1)
[30, 60]
iex> for s <- ["alma", "korte"], do: String.length(s) # map helyett
[4, 5]
```

Amikor perms/1-et a 'tér' karakterlistára alkalmaztuk, számlistát kaptunk eredményül. Ezt map/2-vel könnyű sztringek listájává konvertálni:

Egyszerű példák map/2-vel és filter/2-vel

```
iex> Enum.map([ "alma", "korte" ], &String.length/1)
[4, 5]
iex> Enum.map([ [10, 20], [10, 20, 30] ], &Enum.sum/1)
[30, 60]
iex> for s <- ["alma", "korte"], do: String.length(s) #map helyett
[4, 5]
```

Amikor perms/1-et a 'tér' karakterlistára alkalmaztuk, számlistát kaptunk eredményül. Ezt map/2-vel könnyű sztringek listájává konvertálni:

```
iex> (Fpea.perms ~c"tér") |> (Enum.map &List.to_string/1)
["tér", "tré", "étr", "ért", "rté", "rét"]

iex>
```

Egyszerű példák map/2-vel és filter/2-vel

```
iex> Enum.map([ "alma", "korte" ], &String.length/1)
[4, 5]
iex> Enum.map([ [10, 20], [10, 20, 30] ], &Enum.sum/1)
[30, 60]
iex> for s <- ["alma", "korte"], do: String.length(s) #map helyett
[4, 5]
```

Amikor perms/1-et a 'tér' karakterlistára alkalmaztuk, számlistát kaptunk eredményül. Ezt map/2-vel könnyű sztringek listájává konvertálni:

```
iex> (Fpea.perms ~c"tér") |> (Enum.map &List.to_string/1)
["tér", "tré", "étr", "ért", "rté", "rét"]

iex> Enum.filter([:x, 10, L, 20, {}], &is_number/1)
[10, 20]
iex> Enum.filter([:x, {7,3}, 10, L, 20, {}], &is_tuple/1)
[{7, 3}, {}]
iex>
```

Egyszerű példák map/2-vel és filter/2-vel

```
iex> Enum.map([ "alma", "korte" ], &String.length/1)
[4, 5]
iex> Enum.map([ [10, 20], [10, 20, 30] ], &Enum.sum/1)
[30, 60]
iex> for s <- ["alma", "korte"], do: String.length(s) #map helyett
[4, 5]
```

Amikor perms/1-et a 'tér' karakterlistára alkalmaztuk, számlistát kaptunk eredményül. Ezt map/2-vel könnyű sztringek listájává konvertálni:

```
iex> (Fpea.perms ~c"tér") |> (Enum.map &List.to_string/1)
["tér", "tré", "étr", "ért", "rté", "rét"]

iex> Enum.filter([:x, 10, L, 20, {}], &is_number/1)
[10, 20]
iex> Enum.filter([:x, {7,3}, 10, L, 20, {}], &is_tuple/1)
[{7, 3}, {}]
iex> for i <- 10..50, rem(i,7) === 0, do: i #filter helyett
[14, 21, 28, 35, 42, 49]
iex>
```

Egyszerű példák map/2-vel és filter/2-vel

```
iex> Enum.map([ "alma", "korte" ], &String.length/1)
[4, 5]
iex> Enum.map([ [10, 20], [10, 20, 30] ], &Enum.sum/1)
[30, 60]
iex> for s <- ["alma", "korte"], do: String.length(s) #map helyett
[4, 5]
```

Amikor perms/1-et a 'tér' karakterlistára alkalmaztuk, számlistát kaptunk eredményül. Ezt map/2-vel könnyű sztringek listájává konvertálni:

```
iex> (Fpea.perms ~c"tér") |> (Enum.map &List.to_string/1)
["tér", "tré", "étr", "ért", "rté", "rét"]

iex> Enum.filter([:x, 10, L, 20, {}], &is_number/1)
[10, 20]
iex> Enum.filter([:x, {7,3}, 10, L, 20, {}], &is_tuple/1)
[{7, 3}, {}]
iex> for i <- 10..50, rem(i,7) === 0, do: i #filter helyett
[14, 21, 28, 35, 42, 49]
iex> Enum.to_list 14..50 // 7 #az előző egyszerűbben, szűrés nélkül
[14, 21, 28, 35, 42, 49]
```

Magasabb rendű függvények: `foldr/3` és `foldl/3`

- Gyakran van szükség listákon vagy más kollektciókon aggregáló, redukáló műveletek elvégzésére, pl. egy lista elemeinek összeadására, összeszorzására, egy lista összes elemének egy másik elé fűzésére.
- Ilyenkor a listaelemeken tipikusan valamilyen kétoperandusú műveletet kell végrehajtani, pl. $1+3+6+10+15+21$
- El lehet végezni balról jobbra vagy jobbról balra haladva:
 $(((((0+1)+3)+6)+10)+15)+21 == 56$, $(((((0+21)+15)+10)+6)+3)+1 == 56$

Magasabb rendű függvények: `foldr/3` és `foldl/3`

- Gyakran van szükség listákon vagy más kollektciókon aggregáló, redukáló műveletek elvégzésére, pl. egy lista elemeinek összeadására, összeszorzására, egy lista összes elemének egy másik elé fűzésére.
- Ilyenkor a listaelemeken tipikusan valamilyen kétoperandusú műveletet kell végrehajtani, pl. $1+3+6+10+15+21$

- El lehet végezni balról jobbra vagy jobbról balra haladva:

$$((((0+1)+3)+6)+10)+15)+21 == 56, \quad (((((0+21)+15)+10)+6)+3)+1 == 56$$

$$((((0-1)-3)-6)-10)-15)-21 == -56, \quad (((((0-21)-15)-10)-6)-3)-1 == -56$$

Magasabb rendű függvények: `foldr/3` és `foldl/3`

- Gyakran van szükség listákon vagy más kollektciókon aggregáló, redukáló műveletek elvégzésére, pl. egy lista elemeinek összeadására, összeszorzására, egy lista összes elemének egy másik elé fűzésére.
- Ilyenkor a listaelemeken tipikusan valamilyen kétoperandusú műveletet kell végrehajtani, pl. $1+3+6+10+15+21$
- El lehet végezni balról jobbra vagy jobbról balra haladva:
 $(((((0+1)+3)+6)+10)+15)+21 == 56$, $(((((0+21)+15)+10)+6)+3)+1 == 56$
 $(((((0-1)-3)-6)-10)-15)-21 == -56$, $(((((0-21)-15)-10)-6)-3)-1 == -56$
- A kétféle sorrendet megvalósító függvényt `foldl`-nek, ill. `foldr`-nek szokás hívni (a másodikat `reduce`-nak is); a specifikációjuk azonos:
 $@spec\ fold^*(xs::[any()],\ acc::any(),\ f::(x::any(),\ y::any())\ \rightarrow\ r::any()) :: rs::[any()]$
- `foldl` balról jobbra, `foldr` jobbról balra halad végig a listán.
- Az `f` függvényt, ahol a 2. paraméternek kell az `acc` akkumulátornak lennie, minden lépésben a soron következő listaelemre és az akkumulátorra alkalmazzák.
- Nem asszociatív műveletek esetén `f`-ben a műveleti sorrend fontos! Pl.
 $fn(x,acc)\rightarrow\ acc-x$ end vagy $fn(x,acc)\rightarrow\ x-acc$ end más lesz a `fold` függvények eredménye.

foldr/3 és foldl/3 ⁴ definíciója és használata (dp_fun.exs)

```
R  def sumr([x|xs], acc), do:
    (&+/2).(x, sumr(xs, acc))
  def sumr([], acc), do: acc

  def foldr([x|xs], acc, fun), do:
    fun.(x, foldr(xs, acc, fun))
  def foldr([], acc, _fun), do: acc
```

⁴Az Elixirben: `List.foldr/3`, `List.foldl/3`, továbbá `Enum.reduce/2`, `Enum.reduce/3`.

foldr/3 és foldl/3 ⁴ definíciója és használata (dp_fun.exs)

R	<pre>def sumr([x xs], acc), do: (&+/2).(x, sumr(xs, acc)) def sumr([], acc), do: acc</pre>	<pre>def foldr([x xs], acc, fun), do: fun.(x, foldr(xs, acc, fun)) def foldr([], acc, _fun), do: acc</pre>
L	<pre>def suml([x xs], acc), do: suml(xs, (&+/2).(x, acc)) def suml([], acc), do: acc</pre>	<pre>def foldl([x xs], acc, fun), do: foldl(xs, fun.(x, acc), fun) def foldl([], acc, _fun), do: acc</pre>

• Példák

⁴Az Elixirben: `List.foldr/3`, `List.foldl/3`, továbbá `Enum.reduce/2`, `Enum.reduce/3`.

foldr/3 és foldl/3 ⁴ definíciója és használata (dp_fun.exs)

R	<pre>def sumr([x xs], acc), do: (&+/2).(x, sumr(xs, acc)) def sumr([], acc), do: acc</pre>		<pre>def foldr([x xs], acc, fun), do: fun.(x, foldr(xs, acc, fun)) def foldr([], acc, _fun), do: acc</pre>
L	<pre>def suml([x xs], acc), do: suml(xs, (&+/2).(x, acc)) def suml([], acc), do: acc</pre>		<pre>def foldl([x xs], acc, fun), do: foldl(xs, fun.(x, acc), fun) def foldl([], acc, _fun), do: acc</pre>

• Példák

```
iex> Dp.Fun.foldr([1,2,3,4], 1, &*/2)
```

⁴Az Elixirben: List.foldr/3, List.foldl/3, továbbá Enum.reduce/2, Enum.reduce/3.

foldr/3 és foldl/3 ⁴ definíciója és használata (dp_fun.exs)

R	<pre>def sumr([x xs], acc), do: (&+/2).(x, sumr(xs, acc)) def sumr([], acc), do: acc</pre>		<pre>def foldr([x xs], acc, fun), do: fun.(x, foldr(xs, acc, fun)) def foldr([], acc, _fun), do: acc</pre>
L	<pre>def suml([x xs], acc), do: suml(xs, (&+/2).(x, acc)) def suml([], acc), do: acc</pre>		<pre>def foldl([x xs], acc, fun), do: foldl(xs, fun.(x, acc), fun) def foldl([], acc, _fun), do: acc</pre>

• Példák

```
iex> Dp.Fun.foldr([1,2,3,4], 1, &*/2) # 1*(2*(3*(4*1)))
24
```

⁴Az Elixirben: List.foldr/3, List.foldl/3, továbbá Enum.reduce/2, Enum.reduce/3.

foldr/3 és foldl/3 ⁴ definíciója és használata (dp_fun.exs)

R	<pre>def sumr([x xs], acc), do: (&+/2).(x, sumr(xs, acc)) def sumr([], acc), do: acc</pre>		<pre>def foldr([x xs], acc, fun), do: fun.(x, foldr(xs, acc, fun)) def foldr([], acc, _fun), do: acc</pre>
L	<pre>def suml([x xs], acc), do: suml(xs, (&+/2).(x, acc)) def suml([], acc), do: acc</pre>		<pre>def foldl([x xs], acc, fun), do: foldl(xs, fun.(x, acc), fun) def foldl([], acc, _fun), do: acc</pre>

• Példák

```
iex> Dp.Fun.foldr([1,2,3,4], 1, &*/2) # 1*(2*(3*(4*1)))
24
iex> Dp.Fun.foldl([1,2,3,4], 1, &*/2)
```

⁴Az Elixirben: List.foldr/3, List.foldl/3, továbbá Enum.reduce/2, Enum.reduce/3.

foldr/3 és foldl/3 ⁴ definíciója és használata (dp_fun.exs)

R	<pre>def sumr([x xs], acc), do: (&+/2).(x, sumr(xs, acc)) def sumr([], acc), do: acc</pre>		<pre>def foldr([x xs], acc, fun), do: fun.(x, foldr(xs, acc, fun)) def foldr([], acc, _fun), do: acc</pre>
L	<pre>def suml([x xs], acc), do: suml(xs, (&+/2).(x, acc)) def suml([], acc), do: acc</pre>		<pre>def foldl([x xs], acc, fun), do: foldl(xs, fun.(x, acc), fun) def foldl([], acc, _fun), do: acc</pre>

• Példák

```
iex> Dp.Fun.foldr([1,2,3,4], 1, &*/2) # 1*(2*(3*(4*1)))
24
iex> Dp.Fun.foldl([1,2,3,4], 1, &*/2) # 4*(3*(2*(1*1)))
24
```

⁴Az Elixirben: List.foldr/3, List.foldl/3, továbbá Enum.reduce/2, Enum.reduce/3.

foldr/3 és foldl/3 ⁴ definíciója és használata (dp_fun.exs)

R	<pre>def sumr([x xs], acc), do: (&+/2).(x, sumr(xs, acc)) def sumr([], acc), do: acc</pre>		<pre>def foldr([x xs], acc, fun), do: fun(x, foldr(xs, acc, fun)) def foldr([], acc, _fun), do: acc</pre>
L	<pre>def suml([x xs], acc), do: suml(xs, (&+/2).(x, acc)) def suml([], acc), do: acc</pre>		<pre>def foldl([x xs], acc, fun), do: foldl(xs, fun.(x, acc), fun) def foldl([], acc, _fun), do: acc</pre>

• Példák

```
iex> Dp.Fun.foldr([1,2,3,4], 1, &*/2) # 1*(2*(3*(4*1)))
24
iex> Dp.Fun.foldl([1,2,3,4], 1, &*/2) # 4*(3*(2*(1*1)))
24
iex> Dp.Fun.foldr([1,2,3,4], 0, &- /2)
```

⁴Az Elixirben: List.foldr/3, List.foldl/3, továbbá Enum.reduce/2, Enum.reduce/3.

foldr/3 és foldl/3 ⁴ definíciója és használata (dp_fun.exs)

R	<pre>def sumr([x xs], acc), do: (&+/2).(x, sumr(xs, acc)) def sumr([], acc), do: acc</pre>		<pre>def foldr([x xs], acc, fun), do: fun(x, foldr(xs, acc, fun)) def foldr([], acc, _fun), do: acc</pre>
L	<pre>def suml([x xs], acc), do: suml(xs, (&+/2).(x, acc)) def suml([], acc), do: acc</pre>		<pre>def foldl([x xs], acc, fun), do: foldl(xs, fun.(x, acc), fun) def foldl([], acc, _fun), do: acc</pre>

• Példák

```
iex> Dp.Fun.foldr([1,2,3,4], 1, &*/2) # 1*(2*(3*(4*1)))
24
iex> Dp.Fun.foldl([1,2,3,4], 1, &*/2) # 4*(3*(2*(1*1)))
24
iex> Dp.Fun.foldr([1,2,3,4], 0, &-/2) # 1-(2-(3-(4-0))), fn(x,acc)-> x-acc end
-2
```

⁴Az Elixirben: List.foldr/3, List.foldl/3, továbbá Enum.reduce/2, Enum.reduce/3.

foldr/3 és foldl/3 ⁴ definíciója és használata (dp_fun.exs)

R	<pre>def sumr([x xs], acc), do: (&+/2).(x, sumr(xs, acc)) def sumr([], acc), do: acc</pre>		<pre>def foldr([x xs], acc, fun), do: fun(x, foldr(xs, acc, fun)) def foldr([], acc, _fun), do: acc</pre>
L	<pre>def suml([x xs], acc), do: suml(xs, (&+/2).(x, acc)) def suml([], acc), do: acc</pre>		<pre>def foldl([x xs], acc, fun), do: foldl(xs, fun.(x, acc), fun) def foldl([], acc, _fun), do: acc</pre>

• Példák

```
iex> Dp.Fun.foldr([1,2,3,4], 1, &*/2) # 1*(2*(3*(4*1)))
24
iex> Dp.Fun.foldl([1,2,3,4], 1, &*/2) # 4*(3*(2*(1*1)))
24
iex> Dp.Fun.foldr([1,2,3,4], 0, &-/2) # 1-(2-(3-(4-0))), fn(x,acc)-> x-acc end
-2
iex> Dp.Fun.foldl([1,2,3,4], 0, &-/2)
```

⁴Az Elixirben: List.foldr/3, List.foldl/3, továbbá Enum.reduce/2, Enum.reduce/3.

foldr/3 és foldl/3 ⁴ definíciója és használata (dp_fun.exs)

R	<pre>def sumr([x xs], acc), do: (&+/2).(x, sumr(xs, acc)) def sumr([], acc), do: acc</pre>		<pre>def foldr([x xs], acc, fun), do: fun(x, foldr(xs, acc, fun)) def foldr([], acc, _fun), do: acc</pre>
L	<pre>def suml([x xs], acc), do: suml(xs, (&+/2).(x, acc)) def suml([], acc), do: acc</pre>		<pre>def foldl([x xs], acc, fun), do: foldl(xs, fun.(x, acc), fun) def foldl([], acc, _fun), do: acc</pre>

• Példák

```
iex> Dp.Fun.foldr([1,2,3,4], 1, &*/2) # 1*(2*(3*(4*1)))
24
iex> Dp.Fun.foldl([1,2,3,4], 1, &*/2) # 4*(3*(2*(1*1)))
24
iex> Dp.Fun.foldr([1,2,3,4], 0, &-/2) # 1-(2-(3-(4-0))), fn(x,acc)-> x-acc end
-2
iex> Dp.Fun.foldl([1,2,3,4], 0, &-/2) # 4-(3-(2-(1-0))), fn(x,acc)-> x-acc end
2
```

⁴Az Elixirben: List.foldr/3, List.foldl/3, továbbá Enum.reduce/2, Enum.reduce/3.

foldr/3 és foldl/3 ⁴ definíciója és használata (dp_fun.exs)

R	<pre>def sumr([x xs], acc), do: (&+/2).(x, sumr(xs, acc)) def sumr([], acc), do: acc</pre>	def foldr([x xs], acc, fun), do:	<pre> fun.(x, foldr(xs, acc, fun)) def foldr([], acc, _fun), do: acc</pre>
L	<pre>def suml([x xs], acc), do: suml(xs, (&+/2).(x, acc)) def suml([], acc), do: acc</pre>	def foldl([x xs], acc, fun), do:	<pre> foldl(xs, fun.(x, acc), fun) def foldl([], acc, _fun), do: acc</pre>

• Példák

```
iex> Dp.Fun.foldr([1,2,3,4], 1, &*/2) # 1*(2*(3*(4*1)))
24
iex> Dp.Fun.foldl([1,2,3,4], 1, &*/2) # 4*(3*(2*(1*1)))
24
iex> Dp.Fun.foldr([1,2,3,4], 0, &-/2) # 1-(2-(3-(4-0))), fn(x,acc)-> x-acc end
-2
iex> Dp.Fun.foldl([1,2,3,4], 0, &-/2) # 4-(3-(2-(1-0))), fn(x,acc)-> x-acc end
2
iex> Dp.Fun.foldr([1,2,3,4], 0, fn(x,acc)-> acc-x end)
```

⁴Az Elixirben: List.foldr/3, List.foldl/3, továbbá Enum.reduce/2, Enum.reduce/3.

foldr/3 és foldl/3 ⁴ definíciója és használata (dp_fun.exs)

R	<pre>def sumr([x xs], acc), do: (&+/2).(x, sumr(xs, acc)) def sumr([], acc), do: acc</pre>		<pre>def foldr([x xs], acc, fun), do: fun(x, foldr(xs, acc, fun)) def foldr([], acc, _fun), do: acc</pre>
L	<pre>def suml([x xs], acc), do: suml(xs, (&+/2).(x, acc)) def suml([], acc), do: acc</pre>		<pre>def foldl([x xs], acc, fun), do: foldl(xs, fun.(x, acc), fun) def foldl([], acc, _fun), do: acc</pre>

• Példák

```
iex> Dp.Fun.foldr([1,2,3,4], 1, &*/2) # 1*(2*(3*(4*1)))
24
iex> Dp.Fun.foldl([1,2,3,4], 1, &*/2) # 4*(3*(2*(1*1)))
24
iex> Dp.Fun.foldr([1,2,3,4], 0, &-/2) # 1-(2-(3-(4-0))), fn(x,acc)-> x-acc end
-2
iex> Dp.Fun.foldl([1,2,3,4], 0, &-/2) # 4-(3-(2-(1-0))), fn(x,acc)-> x-acc end
2
iex> Dp.Fun.foldr([1,2,3,4], 0, fn(x,acc)-> acc-x end) # (((0-4)-3)-2)-1
-10
```

⁴Az Elixirben: List.foldr/3, List.foldl/3, továbbá Enum.reduce/2, Enum.reduce/3.

foldr/3 és foldl/3 ⁴ definíciója és használata (dp_fun.exs)

R	<pre>def sumr([x xs], acc), do: (&+/2).(x, sumr(xs, acc)) def sumr([], acc), do: acc</pre>		<pre>def foldr([x xs], acc, fun), do: fun(x, foldr(xs, acc, fun)) def foldr([], acc, _fun), do: acc</pre>
L	<pre>def suml([x xs], acc), do: suml(xs, (&+/2).(x, acc)) def suml([], acc), do: acc</pre>		<pre>def foldl([x xs], acc, fun), do: foldl(xs, fun.(x, acc), fun) def foldl([], acc, _fun), do: acc</pre>

• Példák

```
iex> Dp.Fun.foldr([1,2,3,4], 1, &*/2) # 1*(2*(3*(4*1)))
24
iex> Dp.Fun.foldl([1,2,3,4], 1, &*/2) # 4*(3*(2*(1*1)))
24
iex> Dp.Fun.foldr([1,2,3,4], 0, &-/2) # 1-(2-(3-(4-0))), fn(x,acc)-> x-acc end
-2
iex> Dp.Fun.foldl([1,2,3,4], 0, &-/2) # 4-(3-(2-(1-0))), fn(x,acc)-> x-acc end
2
iex> Dp.Fun.foldr([1,2,3,4], 0, fn(x,acc)-> acc-x end) # (((0-4)-3)-2)-1
-10
iex> Dp.Fun.foldl([1,2,3,4], 0, fn(x,acc)-> acc-x end)
```

⁴Az Elixirben: List.foldr/3, List.foldl/3, továbbá Enum.reduce/2, Enum.reduce/3.

foldr/3 és foldl/3 ⁴ definíciója és használata (dp_fun.exs)

R	<pre>def sumr([x xs], acc), do: (&+/2).(x, sumr(xs, acc)) def sumr([], acc), do: acc</pre>		<pre>def foldr([x xs], acc, fun), do: fun(x, foldr(xs, acc, fun)) def foldr([], acc, _fun), do: acc</pre>
L	<pre>def suml([x xs], acc), do: suml(xs, (&+/2).(x, acc)) def suml([], acc), do: acc</pre>		<pre>def foldl([x xs], acc, fun), do: foldl(xs, fun.(x, acc), fun) def foldl([], acc, _fun), do: acc</pre>

• Példák

```
iex> Dp.Fun.foldr([1,2,3,4], 1, &*/2) # 1*(2*(3*(4*1)))
24
iex> Dp.Fun.foldl([1,2,3,4], 1, &*/2) # 4*(3*(2*(1*1)))
24
iex> Dp.Fun.foldr([1,2,3,4], 0, &-/2) # 1-(2-(3-(4-0))), fn(x,acc)-> x-acc end
-2
iex> Dp.Fun.foldl([1,2,3,4], 0, &-/2) # 4-(3-(2-(1-0))), fn(x,acc)-> x-acc end
2
iex> Dp.Fun.foldr([1,2,3,4], 0, fn(x,acc)-> acc-x end) # (((0-4)-3)-2)-1
-10
iex> Dp.Fun.foldl([1,2,3,4], 0, fn(x,acc)-> acc-x end) # (((0-1)-2)-3)-4
-10
```

⁴Az Elixirben: List.foldr/3, List.foldl/3, továbbá Enum.reduce/2, Enum.reduce/3.

További példák `foldr/3` és `foldl/3` használatára

Példák listák összefűzésére

```
iex> {is, as} = {~c"indul", ~c"aludni"}  
{~c"indul", ~c"aludni"}  
iex> List.foldr(is, as, &([&1|&2]))
```

További példák foldr/3 és foldl/3 használatára

Példák listák összefűzésére

```
iex> {is, as} = {~c"indul", ~c"aludni"}  
{~c"indul", ~c"aludni"}  
iex> List.foldr(is, as, &([&1|&2])) # [?i|[?n|[?d|[?u|[?l|~c"aludni"]]]]]  
~c"indulaludni"
```

További példák foldr/3 és foldl/3 használatára

Példák listák összefűzésére

```
iex> {is, as} = {~c"indul", ~c"aludni"}  
{~c"indul", ~c"aludni"}  
iex> List.foldr(is, as, &([&1|&2])) # [?i|[?n|[?d|[?u|[?l|~c"aludni"]]]]]  
~c"indulaludni"  
iex> List.foldl(is, as, &([&1|&2]))
```

További példák foldr/3 és foldl/3 használatára

Példák listák összefűzésére

```
iex> {is, as} = {~c"indul", ~c"aludni"}  
{~c"indul", ~c"aludni"}  
iex> List.foldr(is, as, &([&1|&2])) # [?i|[?n|[?d|[?u|[?l|~c"aludni"]]]]]  
~c"indulaludni"  
iex> List.foldl(is, as, &([&1|&2])) # [?l|[?u|[?d|[?n|[?i|~c"aludni"]]]]]  
~c"ludnialudni"
```

További példák foldr/3 és foldl/3 használatára

Példák listák összefűzésére

```
iex> {is, as} = {~c"indul", ~c"aludni"}  
{~c"indul", ~c"aludni"}  
iex> List.foldr(is, as, &([&1|&2])) # [?i|[?n|[?d|[?u|[?l|~c"aludni"]]]]]  
~c"indulaludni"  
iex> List.foldl(is, as, &([&1|&2])) # [?l|[?u|[?d|[?n|[?i|~c"aludni"]]]]]  
~c"ludnialudni"  
iex> List.foldl(is, [], &([&1|&2]))
```

További példák foldr/3 és foldl/3 használatára

Példák listák összefűzésére

```
iex> {is, as} = {~c"indul", ~c"aludni"}  
{~c"indul", ~c"aludni"}  
iex> List.foldr(is, as, &([&1|&2])) # [?i|[?n|[?d|[?u|[?l|~c"aludni"]]]]]  
~c"indulaludni"  
iex> List.foldl(is, as, &([&1|&2])) # [?l|[?u|[?d|[?n|[?i|~c"aludni"]]]]]  
~c"ludnialudni"  
iex> List.foldl(is, [], &([&1|&2])) # [?l|[?u|[?d|[?n|[?i|[]]]]]  
~c"ludni"
```

További példák foldr/3 és foldl/3 használatára

Példák listák összefűzésére

```
iex> {is, as} = {~c"indul", ~c"aludni"}
{~c"indul", ~c"aludni"}
iex> List.foldr(is, as, &([&1|&2])) # [?i|[?n|[?d|[?u|[?l|~c"aludni"]]]]]
~c"indulaludni"
iex> List.foldl(is, as, &([&1|&2])) # [?l|[?u|[?d|[?n|[?i|~c"aludni"]]]]]
~c"ludnialudni"
iex> List.foldl(is, [], &([&1|&2])) # [?l|[?u|[?d|[?n|[?i|[]]]]]]
~c"ludni"
```

- `&([&1|&2])` a *listakonstruktorból* csinál névtelen függvényt: a második (lista-)paraméter elé fűzi az elsőt.
- Mivel `foldr` az első listaparaméter utolsó karakterével kezd, az 1. listát az *eredeti sorrendben* fűzi a 2. elé: eredménye azonos `append/2`-ével.
- `foldl` viszont az első listaparaméter első karakterével kezd, ezért az 1. listát *fordított sorrendben* fűzi a 2. elé: `revapp/2` a szokásos neve, az Elixirben `Enum.reverse/2` néven található meg.
- Ha `foldl` második paramétere az üres lista, akkor megfordítja az első listát: eredménye azonos `Enum.reverse/1`-ével.

Záró megjegyzések a magasabb rendű függvényekről

- Amikor funkcionális nyelven egy listán pl. valamilyen összegzést végzünk, az imperatív nyelvekben megszokott *ciklus* helyett *rekurziót* használunk.
- Az alábbi jól ismert példában a rekurzió *explicit*, azaz megjelenik a programszövegben, a `sum/1` meghívását a futtatórendszer *rekurzív processzként* futtatja – hacsak a fordítóprogram át nem alakítja ciklussá:

```
def sum([]), do: 0
def sum([x|xs]), do: x + sum xs
iex> Dp.Fun.sum [1,2,3,4,5]
15
```

Záró megjegyzések a magasabb rendű függvényekről

- Amikor funkcionális nyelven egy listán pl. valamilyen összegzést végzünk, az imperatív nyelvekben megszokott *ciklus* helyett *rekurziót* használunk.
- Az alábbi jól ismert példában a rekurzió *explicit*, azaz megjelenik a programszövegben, a `sum/1` meghívását a futtatórendszer *rekurzív processzként* futtatja – hacsak a fordítóprogram át nem alakítja ciklussá:

```
def sum([]), do: 0
def sum([x|xs]), do: x + sum xs
iex> Dp.Fun.sum [1,2,3,4,5]
15
```

- Ha ugyanezt a feladatot pl. `List.foldl/3`-mal oldjuk meg, a programszöveg nem tartalmaz rekurziót:

```
iex> List.foldl([1,2,3,4,5], 0, &(&1 + &2))
15
```

- Lehet, hogy a futtatott kód rekurzív, lehet, hogy nem, nem tudhatjuk.
- Mindenesetre a kódot sokkal könnyebb megérteni, sem programozóként, sem kódolvasóként nem kell a rekurzív megoldás megfejtésével bajlódni.