Funkcionális programozás a gyakorlatban

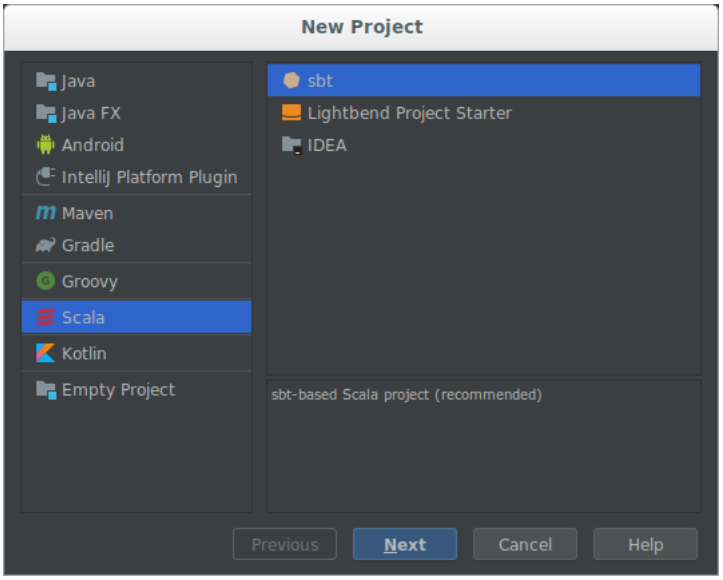
work in progress

0.1 Programozási környezet és egy Hello Scala

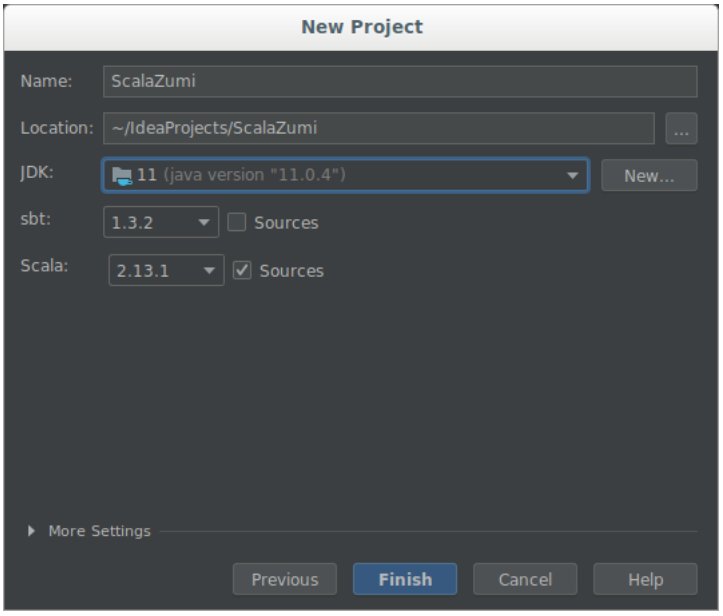
Scala fejlesztéshez sok opció van, van hozzá Eclipse plugin is, IntelliJ IDEA plugin is. Ha pl. az IntelliJ IDEA plugint választjuk, akkor szerezzük be az IntelliJ friss változatát a https://www.jetbrains.com/idea/download oldalról, rakjuk fel. Elég hozzá a Community. Ha frissen telepítjük, akkor közben a Download featured plugins lapon kérjük tőle, hogy tegye fel a Scala plugint is, egyébként a Settings/Plugins menüben lehet lehúzni később, ha ezt a lépést kihagytuk.

A jegyzetben a screenshotok, menük stb. az IntelliJ IDEA 2019.2.3 verziója alatt készültek.

Hozzunk létre egy új Scala projektet, az sbt (Scala Build Tool) használatával:



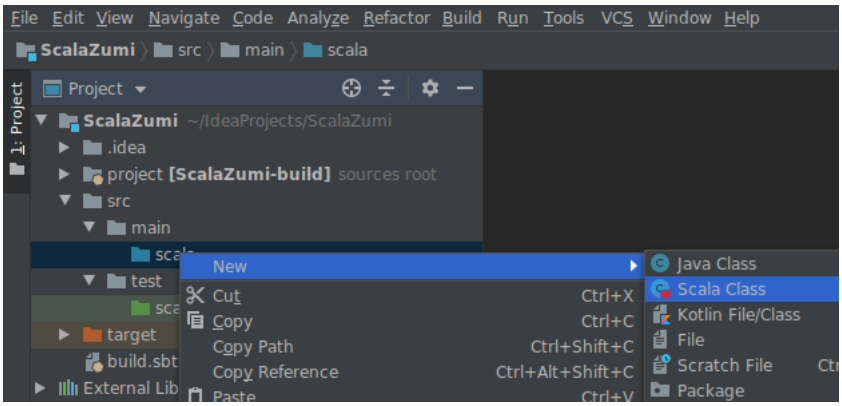
Nálam a JDK 11.0.4-es, az sbt 1.3.2-es, a Scala pedig 2.13.1-es. A Scalának a forrását is betikkeltem, az sbt-nek nem, a projekt neve pedig ScalaZumi lett:

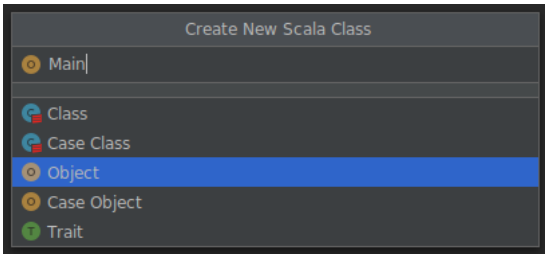


Új projekt létrehozásakor (legalábbis legelső alkalommal) eltart egy rövid ideig, amíg az IntelliJ lehúzza az összes szükséges jart, ezt szépen kivárjuk.

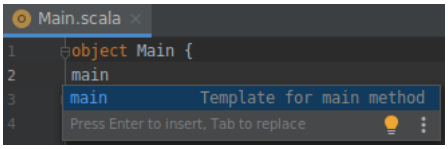
A bal oldali projektstruktúrában három érdekes dolgot találunk, amiket fogunk piszkálni: src/-main/scala, ide kerülnek a forráskódjaink, src/test/scala, ide kerülnek a tesztjeink, és a build.sbt file, ide írjuk majd be a fordításhoz szükséges egyéb csomagokat stb.

Hozzunk létre egy új Scala osztályt a src/main/scala helyen, mégpedig egy Object-et. Én itt Main-nek neveztem el.

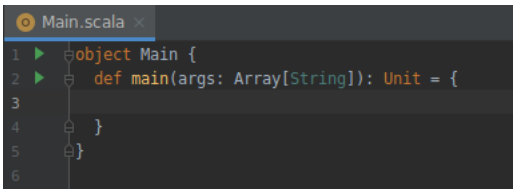




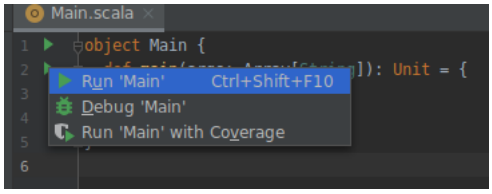
Az újonnan létrehozott Main.scala fileunkban, az object Main belsejébe készítsünk egy main függvényt. Ennek a legegyszerűbb módja, ha (mondjuk) a main begépelése után ütünk ctrl-space-t és elfogadjuk a felkínált main method automatikus kiegészítést:



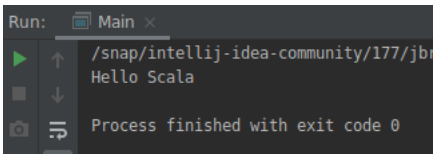
Ezt kéne kapjuk (persze ha typing fétisünk van, be is gépelhetjük kézzel az egészet – ügyelve a kis- és nagybetűkre):



Az eleinte írt programjaink belépési pontja mindig egy objecten belül deklarált, main nevű, Array[String] típusú argumentumot váró, Unit visszatérési típusú metódusa lesz. Ha ennek belsejébe írunk kódot, azt futtathatjuk:



Miután a scalac fordító lebuildeli nekünk a kódunkat, lent a konzol ablakban láthatjuk is az eredményt:



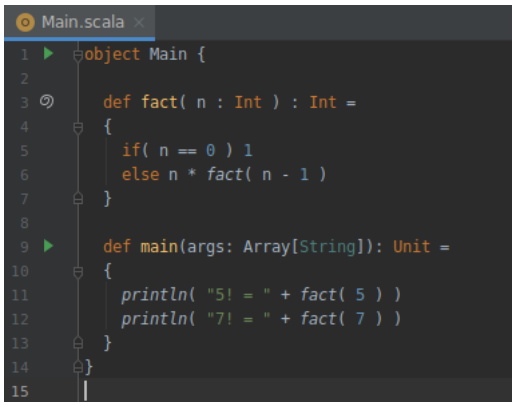
Easy.

Metódusok, típusok, if és rekurzió

A Scala egy, az objektum-orientáltságot és a funkcionális programozást ötvöző nyelv. A kurzus első részében a funkcionális paradigmára helyezzük a hangsúlyt – amíg ez így van, addig mindent az egy darab object Main-ünk belsejébe fogunk kódolni és ezen belül lesz a main metódus is, ami pedig a programunk belépési pontja lesz.

A nyelvi elemek megismeréséhez lekódolunk pár „matekos” függvényt – előbb-utóbb ebből lesz majd app is, most csak a szintaxis megértéséhez tesszük ezt.

Első körben írunk egy faktoriálist számító rekurzív[1] függvényt, és kiíratjuk az 5! és 7! értékeket:



Ahogy progalapból is hopefully megtanultuk, ha természetes számot vár a függvényünk, önmagát hívja (azaz rekurzív) kisebb természetes számra, akkor azt „könnyű” (lehet) belátni, hogy tényleg helyes outputot számít ki: megnézzük, hogy 0-ra helyes-e az érték (most igen: ha a híváskor $n = 0$, akkor az érték 1 lesz), és utána feltesszük, hogy az input $n > 0$-nál kisebbekre már tudjuk, hogy helyes az output, abból „valahogy” kiókumuláljuk, hogy n-re is jó lesz (most igen: ha $\mathsf{fact}(n-1)$ értéke $(n-1)!$, akkor ha n-re hívjuk, akkor az érték $n * \mathsf{fact}(n-1) = n * (n-1)! = n!$ lesz, tehát a kód helyes).

Azt kapjuk, hogy $5! = 120$ és $7! = 5040$. Lássuk a programunk részeit külön-külön (abból azt is látni fogjuk, hogy $n = 0$-ra miért 1 lesz az érték és $n > 0$-ra miért $n * \mathsf{fact}(n-1)$ értéke).

Metódus (kb: „függvény”) deklarálását a def kulcsszóval jelöljük, ezután jön a metódus neve (most ez fact), utána nyitó-csukójelek közt az argumentumlistája, majd kettőspont, a visszatérési érték típusa, egy egyenlőségjel, utána maga a metódus törzse, ebben a példában az olvashatóság kedvéért kapcsos zárójelbe rakva. Az argumentumlista pedig név, kettőspont, típus párok, vesszővel elválasztva.

Azt már a példakódból is láthatjuk, hogy Scalában ezek szerint beépített típusok az Int, Unit, Array[String], és egyébként a String is. Egyelőre a Stringeket tartalmazó tömbbel (ez az Array[String]) nem foglalkozunk. Az Int 32-bites előjeles egész számokat tároló típus (tehát

[1] lásd: rekurzív

kb. ± kétmilliárdig tudunk bele számokat rakni), megfelel a Java int alaptípusának, a String pedig unicode szöveget tárol (mint pl. "sanyi", "5!= ", vagy "asdfjklé"), megfelel a Java String típusának.

A Unit típusról egyelőre annyit érdemes tudnunk, hogy míg az előző típusokba sokféle értéket rakhattunk (az Int típusbapl. kb 4 milliárd-féle érték tartozik), a Unit típusba csak egyféle érték tartozik, mégpedig a (). Ennek a típusnak a létjogosultságát az adja, hogy Scalában szinte minden kifejezés lesz, és minden kifejezésnek kell legyen típusa; ha valaminek „nincs visszatérési értéke", azt pl. C-ben vagy Javában void-nak deklaráljuk, és amit Scalában Unit-nak deklarálunk, az a JVM számára a legtöbb esetben void lesz.

A substitution model, call-by-name és call-by-value

Most már tehát tudjuk, hogy a fenti kódban a fact nevű metódus inputként egy Int-et kap, és Int-et ad vissza. Ha megnézzük az implementációt, az első, ami feltűnhet, hogy nincs return utasítás. Ugyan maga a kulcsszó létezik a nyelvben és ebben az esetben akár oda is írhatnánk, idiomatikus Scala kódban nem szokás használni. Ennek az oka: a Scala nyelv elvi működése (az „operatív szemantika") azt mondja, hogy ha meghívunk egy metódust (pl. fact(3)), akkor ezt a kifejezést úgy értékeljük ki, hogy a metódus formális paraméterei helyébe az aktuális paramétereket behelyettesítjük a metódus kódjában[2]. Azaz pl. most:

$$\mathsf{fact}(3) \ \triangleright\ \mathsf{if}(3 == 0)\ 1\ \mathsf{else}\ 3 * \mathsf{fact}(3-1)$$

Itt és később a $\triangleright$ jelentése: a bal oldali kifejezés a kiértékelés („futtatás") egy lépésében átíródik a jobb oldalivá.

A feltételes elágazás ebben a nyelvben is if..else konstruktummal implementálható: a kifejezés formája $\mathsf{if}(B)\ E\ \mathsf{else}\ F$ alakú, ahol B egy Boolean típusú kifejezés, E és F pedig bármilyen típusú kifejezés lehet – akár Unit is, sőt: ha nem írunk else ágat, akkor a fordító generál is automatikusan egy else () befejezést a kifejezésnek, ami láttuk, hogy tényleg egy Unit típusú érték.

A Boolean típus megfelel a Java boolean alaptípusának és két értéke van: a true és a false.

A feltételes elágazásra vonatkozó operatív szemantika szabályai:

$$\begin{aligned}
\mathsf{if}(\mathsf{true})\ E\ \mathsf{else}\ F &\ \triangleright\ E \\
\mathsf{if}(\mathsf{false})\ E\ \mathsf{else}\ F &\ \triangleright\ F \\
\mathsf{if}(B)\ E\ \mathsf{else}\ F &\ \triangleright\ \mathsf{if}(B')\ E\ \mathsf{else}\ F, \text{ ha } B \triangleright B'
\end{aligned}$$

Azaz: ha a kifejezés feltételét már kiértékeltük, akkor annak megfelelően átírjuk vagy az if, vagy az else ágban szereplő kifejezésre, ha meg még nem, akkor a feltételt értékeljük tovább.

Jelen esetben a $3 == 0$ még nem egy Boolean érték, ezt értékeljük tovább, persze ez az egyenlőség hamis (false) lesz, tehát:

$$\begin{aligned}
\mathsf{fact}(3) &\triangleright \mathsf{if}(3 == 0)\ 1\ \mathsf{else}\ 3 * \mathsf{fact}(3-1) \\
&\triangleright \mathsf{if}(\ \mathsf{false}\ )\ 1\ \mathsf{else}\ 3 * \mathsf{fact}(3-1) \\
&\triangleright 3 * \mathsf{fact}(3-1)
\end{aligned}$$

Ezen a ponton, amikor is a függvényhívás argumentuma még nem egy érték (most egy összetett kifejezés, az $3-1$ az argumentum), két lehetőség van a funkcionális programozási nyelveknél.

[2] egyelőre legalábbis így lesz. Stay tuned.

Az első a call-by-name konvenció: amikor is ezt az összetett kifejezést ahogy van, úgy helyettesítjük be a metódus törzsébe. Ekkor az átírás folytatása a következőképp alakul:

$$
\begin{aligned}
& 3 * \mathsf{fact}(3-1) \\
\triangleright\ & 3 * \Big(\mathsf{if}(3-1==0)\ 1\ \mathsf{else}\ (3-1) * \mathsf{fact}(3-1-1)\Big) \\
\triangleright\ & 3 * \Big(\mathsf{if}(2==0)\ 1\ \mathsf{else}\ (3-1) * \mathsf{fact}(3-1-1)\Big) \\
\triangleright\ & 3 * \Big(\mathsf{if}(\mathsf{false})\ 1\ \mathsf{else}\ (3-1) * \mathsf{fact}(3-1-1)\Big) \\
\triangleright\ & 3 * \Big((3-1) * \mathsf{fact}(3-1-1)\Big) \\
\triangleright\ & 3 * \Big((3-1) * (\mathsf{if}(3-1-1==0)\ 1\ \mathsf{else}\ (3-1-1) * \mathsf{fact}(3-1-1-1))\Big) \\
\triangleright\ & 3 * \Big((3-1) * (\mathsf{if}(2-1==0)\ 1\ \mathsf{else}\ (3-1-1) * \mathsf{fact}(3-1-1-1))\Big) \\
\triangleright\ & 3 * \Big((3-1) * (\mathsf{if}(1==0)\ 1\ \mathsf{else}\ (3-1-1) * \mathsf{fact}(3-1-1-1))\Big) \\
\triangleright\ & 3 * \Big((3-1) * (\mathsf{if}(\mathsf{false})\ 1\ \mathsf{else}\ (3-1-1) * \mathsf{fact}(3-1-1-1))\Big) \\
\triangleright\ & 3 * \Big((3-1) * ((3-1-1) * \mathsf{fact}(3-1-1-1))\Big) \\
\triangleright\ & 3 * \Big((3-1) * ((3-1-1) * \mathsf{if}(3-1-1-1==0)\ 1\ \mathsf{else}\ (3-1-1-1) * \mathsf{fact}(3-1-1-1-1))\Big) \\
\triangleright\ & 3 * \Big((3-1) * ((3-1-1) * \mathsf{if}(2-1-1==0)\ 1\ \mathsf{else}\ (3-1-1-1) * \mathsf{fact}(3-1-1-1-1))\Big) \\
\triangleright\ & 3 * \Big((3-1) * ((3-1-1) * \mathsf{if}(1-1==0)\ 1\ \mathsf{else}\ (3-1-1-1) * \mathsf{fact}(3-1-1-1-1))\Big) \\
\triangleright\ & 3 * \Big((3-1) * ((3-1-1) * \mathsf{if}(0==0)\ 1\ \mathsf{else}\ (3-1-1-1) * \mathsf{fact}(3-1-1-1-1))\Big) \\
\triangleright\ & 3 * \Big((3-1) * ((3-1-1) * \mathsf{if}(\mathsf{true})\ 1\ \mathsf{else}\ (3-1-1-1) * \mathsf{fact}(3-1-1-1-1))\Big) \\
\triangleright\ & 3 * \Big((3-1) * ((3-1-1) * 1)\Big) \\
\triangleright\ & 3 * \Big((3-1) * ((2-1) * 1)\Big) \\
\triangleright\ & 3 * \Big((3-1) * (1 * 1)\Big) \\
\triangleright\ & 3 * \Big((3-1) * 1\Big) \\
\triangleright\ & 3 * \Big(2 * 1\Big) \\
\triangleright\ & 3 * 2 \\
\triangleright\ & 6
\end{aligned}
$$

és ez lesz a $\mathsf{fact}(3)$ kifejezésünk értéke[3].

A második a call-by-value konvenció: amikor is először kiértékeljük az összes argumentumot,

[3]Valójában nem **teljesen** így történik, mert a szorzás is egy metódus, és az interpreter előbb a szorzás bal oldalán álló argumentumokat értékelné ki, azaz pl. a $(3-1)$-et írná át 2-re a szorzásjel előtt, és csak aztán fejtené ki a $\mathsf{fact}(3-1-1)$ hívást.

és ezután helyettesítjük be a függvény törzsébe a kapott értékeket. Ekkor az átírás folyamata:

$$
\begin{aligned}
3 * \mathsf{fact}(3-1) &\triangleright 3 * \mathsf{fact}(2) \\
&\triangleright 3 * \Big(\mathsf{if}(2 == 0)\ 1\ \mathsf{else}\ 2 * \mathsf{fact}(2-1)\Big) \\
&\triangleright 3 * \Big(\mathsf{if}(\mathsf{false})\ 1\ \mathsf{else}\ 2 * \mathsf{fact}(2-1)\Big) \\
&\triangleright 3 * 2 * \mathsf{fact}(2-1) \\
&\triangleright 3 * 2 * \mathsf{fact}(1) \\
&\triangleright 3 * 2 * (\mathsf{if}(1 == 0)\ 1\ \mathsf{else}\ 1 * \mathsf{fact}(1-1)) \\
&\triangleright 3 * 2 * (\mathsf{if}(\mathsf{false})\ 1\ \mathsf{else}\ 1 * \mathsf{fact}(1-1)) \\
&\triangleright 3 * 2 * 1 * \mathsf{fact}(1-1) \\
&\triangleright 3 * 2 * 1 * \mathsf{fact}(0) \\
&\triangleright 3 * 2 * 1 * (\mathsf{if}(0 == 0)\ 1\ \mathsf{else}\ 0 * \mathsf{fact}(0-1) \\
&\triangleright 3 * 2 * 1 * (\mathsf{if}(\mathsf{true})\ 1\ \mathsf{else}\ 0 * \mathsf{fact}(0-1) \\
&\triangleright 3 * 2 * 1 * 1 \\
&\triangleright 3 * 2 * 1 \\
&\triangleright 3 * 2 \\
&\triangleright 6
\end{aligned}
$$

és így is 6-ot kaptunk a kifejezés értékének.

Scalában a default a call-by-value, tehát a példánk fact(3) hívásában az utóbbi módon zajlik a számítás. Azonban ha egy argumentumot call-by-name akarunk átadni egy függvénynek, arra is van mód: a fenti példában egyszerűen $n :$ $\Rightarrow$ Int-et kell írjunk $n :$ Int helyébe (a $\Rightarrow$ jelet persze =>-ként visszük be) és máris call-by-name értékelődik ki a bejövő argumentum:

```
def fact( n : ⇒ Int ) : Int = if ( n == 0 ) 1 else n * fact( n - 1 )
```

Arra is van lehetőségünk, hogy egy több argumentummal rendelkező metódusban némelyik argumentumokat call-by-name, másokat meg call-by-value értékeljünk ki.

Igen, a kapcsos – ahogy láthatjuk – elhagyható: a kapcsos zárójel alapvetően arra való, hogy több kifejezést szervezzen egy blokkba, de mivel itt csak egy darab kifejezésünk van, nincs rá szükség. Ha több kifejezésünk van egy blokkban, akkor ezek egymás után kiértékelődnek, és az egész blokk értéke az utolsó kifejezés értéke lesz. Ezért gondoljuk át, mit teszünk, mielőtt pl. elhagyjuk az else kulcsszót és külön sorba írjuk a benne deklarált függvényhívást így:

```
def fact( n : Int ) : Int = {
  if ( n == 0 ) 1
  n * fact( n - 1 )
}
```

Ekkor ugyanis végtelen ciklusba esünk (ha kipróbáljuk erre a változatra meghívni pl. a fact(2)-t, vagy 2 helyett bármi mást szállítunk argumentumnak, kapunk is egy StackOverflowErrort). Miért is? Mert ez nem egy imperatív programozási nyelv, itt nem szakad meg a kiértékelés egy beékelt return miatt. Ha pl. meghívjuk a fact(0)-t, akkor ez történik (ha egy sorba több kifejezést írunk, akkor azokat pontosvesszővel választhatjuk el, ezt a példában így írom – ettől

még a kapcsos kell köré):

$$
\begin{aligned}
\mathsf{fact}(0) &\triangleright \{\ \mathsf{if}(0 == 0)\ 1\ \mathsf{else}\ ();\ 0 * \mathsf{fact}(0-1)\ \} \\
&\triangleright \left\{\ \mathsf{if}(\mathsf{true})\ 1\ \mathsf{else}\ ();\ 0 * \mathsf{fact}(0-1)\ \right\} \\
&\triangleright \left\{1;\ 0 * \mathsf{fact}(0-1)\ \right\} \\
&\triangleright \left\{0 * \mathsf{fact}(0-1)\ \right\} \\
&\triangleright \left\{0 * \mathsf{fact}(-1)\ \right\} \\
&\triangleright \left\{0 * \{\mathsf{if}(-1 == 0)\ 1\ \mathsf{else}\ (); -1 * \mathsf{fact}(-1-1)\ \}\right\} \\
&\triangleright \left\{0 * \{\mathsf{if}(\mathsf{false})\ 1\ \mathsf{else}\ (); -1 * \mathsf{fact}(-1-1)\ \}\right\} \\
&\triangleright \left\{0 * \{(); -1 * \mathsf{fact}(-1-1)\ \}\right\} \\
&\triangleright \left\{0 * \{-1 * \mathsf{fact}(-1-1)\ \}\right\} \\
&\triangleright \left\{0 * \{-1 * \mathsf{fact}(-2)\ \}\right\} \\
&\triangleright \ldots
\end{aligned}
$$

(emlékszünk? Ha az if hez nincs else, akkor a fordító berak egy else () ágat) és így a rekurzió nem fog megállni.

Pontosabban, a StackOverflowError amikor jön, akkor megáll és kilövi a programunkat... ha jobban megnézzük, akkor a folyamat során ahogy hívogatjuk a fact függvényt, egyre hosszabb lesz a teljes kifejezésünk: előbb $\mathsf{fact}(0)$, aztán $0 * \mathsf{fact}(-1)$, majd $0 * (-1 * \mathsf{fact}(-2))$, majd $0*(-1*(-2*\mathsf{fact}(-3)))$, és így tovább. Amint a kifejezés „hossza” (ezt nemsokára pontosítom, inkább „mélysége”) túllő egy előre beállított korlátot (azaz „betelik a hívási verem”), a JVM dob nekünk egy ilyen errort. A hívási verem méretét jellemzően akkora értékre szokta defaultból beállítani a JVM, hogy egy StackOverflowError szinte mindig tényleg megbízható jele annak, hogy itt bizony egy olyan (rekurzív, azaz önmagát újabb és újabb paraméterekkel hívó függvény) számítás zajlik, ami végtelen ciklusba esett.

A hívási verem és a tail recursion

Hogy a hívási verem fogalmát jobban megértsük és megismerkedjünk az ezt kiküszöbölő tail recursionnal (farokrekurzió), nézzük meg a másik klasszik rekurzió állatorvosi lovat, az n. Fibonacci számot kiszámító függvényt:

```
object Main {

  def fact( n : Int ) : Int = if( n == 0 ) 1 else n * fact( n - 1 )

  def fib( n : Int ) : Int = if( n <= 1 ) 1 else fib( n - 1 ) + fib( n - 2 )

  def main(args: Array[String]): Unit =
  {
    println( fib( 4 ) )
    println( fib( 5 ) )
    println( fib( 6 ) )
  }
}
```

Mondjuk ki akarjuk számítani a $\mathsf{fib}(4)$ hívást. Az operatív szemantika szerint haladunk a kiértékeléssel:

$$\begin{aligned}\mathsf{fib}(4) &\triangleright \mathsf{if}(4 \leq 1)\ 1\ \mathsf{else}\ \mathsf{fib}(4-1) + \mathsf{fib}(4-2)\\ &\triangleright \mathsf{if}(\mathsf{false})\ 1\ \mathsf{else}\ \mathsf{fib}(4-1) + \mathsf{fib}(4-2)\\ &\triangleright \mathsf{fib}(4-1) + \mathsf{fib}(4-2)\end{aligned}$$

Na most ezen a ponton van „kívül” egy függvényünk, mégpedig az *összeadás* az (őt hajtanánk végre utoljára). Ennek a függvénynek az argumentumait még nem értékeltük ki mindet, call-by-value üzemelünk, ezért először ki kell értékeljük a $\mathsf{fib}(4-1)$-et, majd a $\mathsf{fib}(4-2)$-t, és utána összeadni a két értéket. Eddig ezeket egy sorba írtuk, most kiszámolnánk a $4-1$-et, ami 3 lesz, aztán az így kapott $\mathsf{fib}(3)$ átírásával folytatnánk tovább, ami rendben is van, de ennél összetettebb egy kicsit a kiértékelési mechanizmus, nem egy stringként vagy egy nagy kifejezésfaként tartja nyilván a programunk aktuális állapotát a JVM, hanem[4] ilyenkor lenyomja egy verembe a függvényt az argumentum-kifejezésekkel együtt, majd szépen sorban végighalad az argumentumokon, jelzi magának, hogy épp melyiket értékeli ki és csak azt tartja az aktuális „munkamemóriájában”. Rajzban most kérdőjellel fogom jelölni az aktuálisan kiértékelt argumentumot. Tehát a hívás a következő módon folytatódik tovább[5]:

$\mathsf{fib}(4-1) + \mathsf{fib}(4-2)$

$\triangleright$ | + | $\mathsf{fib}(4-1)$ | $\mathsf{fib}(4-2)$ |

$\triangleright$ $\mathsf{fib}(4-1)$

| + | ? | $\mathsf{fib}(4-2)$ |

$\triangleright$ $\mathsf{fib}(3)$

| + | ? | $\mathsf{fib}(4-2)$ |

$\triangleright$ $\mathsf{if}(3 \leq 1)\ 1\ \mathsf{else}\ \mathsf{fib}(3-1) + \mathsf{fib}(3-2)$

| + | ? | $\mathsf{fib}(4-2)$ |

$\triangleright$ $\mathsf{if}(\mathsf{false})\ 1\ \mathsf{else}\ \mathsf{fib}(3-1) + \mathsf{fib}(3-2)$

| + | ? | $\mathsf{fib}(4-2)$ |

$\triangleright$ $\mathsf{fib}(3-1) + \mathsf{fib}(3-2)$

| + | ? | $\mathsf{fib}(4-2)$ |

és itt megint egy olyan függvényhívásunk van, aminek az argumentumait nem értékeltük még ki, tehát ezt megint lerakjuk a hívási verembe és folytatjuk a kiértékelését az argumentumainak:

$\mathsf{fib}(3-1) + \mathsf{fib}(3-2)$

| + | ? | $\mathsf{fib}(4-2)$ |

$\triangleright$

+	$\mathsf{fib}(3-1)$	$\mathsf{fib}(3-2)$
	+ ? $\mathsf{fib}(4-2)$	

$\triangleright$ $\mathsf{fib}(3-1)$

+	?	$\mathsf{fib}(3-2)$
+	?	$\mathsf{fib}(4-2)$

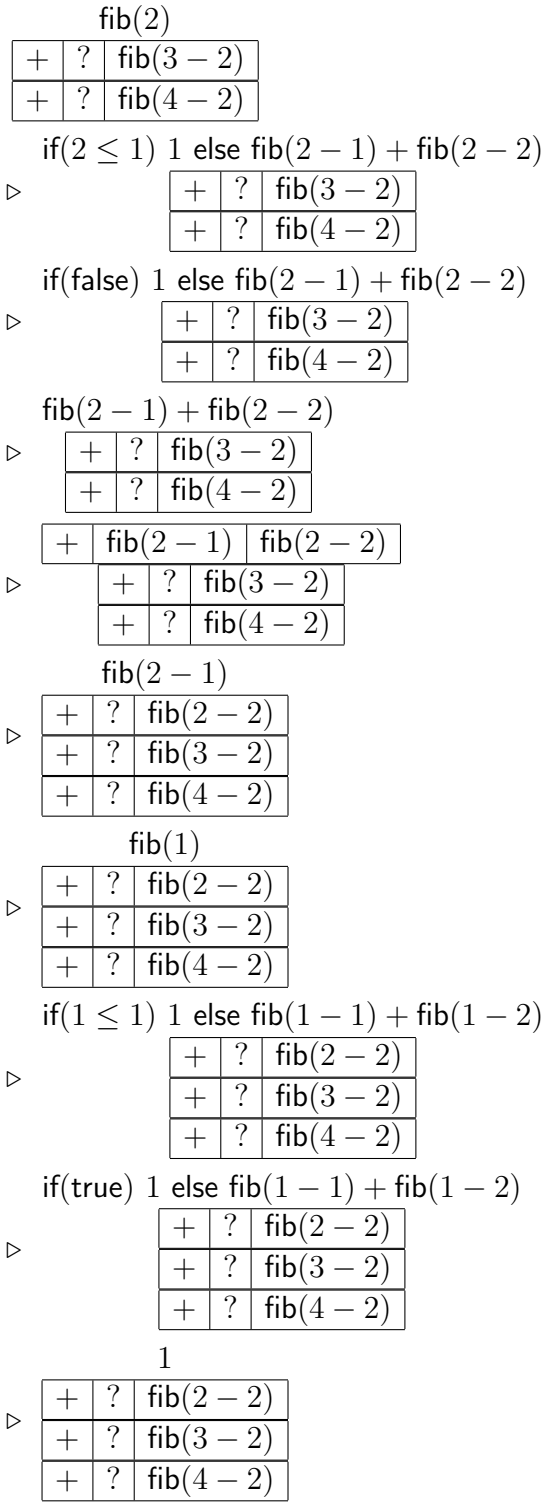
$\triangleright$ $\mathsf{fib}(2)$

+	?	$\mathsf{fib}(3-2)$
+	?	$\mathsf{fib}(4-2)$

[4] a következőkben leírt működés még mindig egyszerűsített, de nekünk most megfelelő lesz

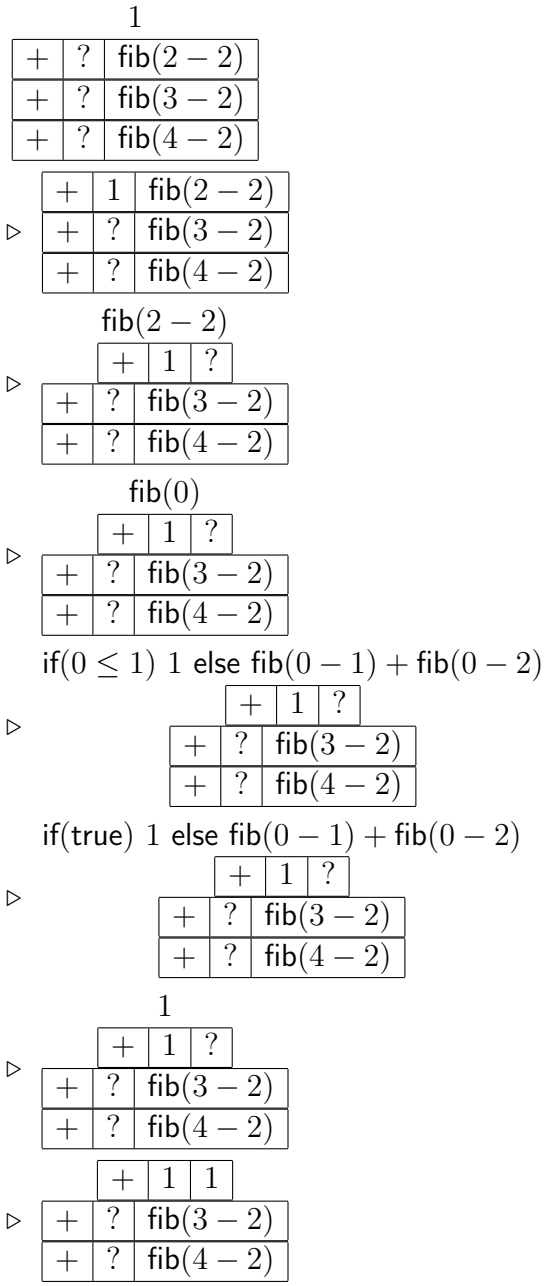
[5] TODO: fix ezt az ugly elrendezést vhogy

Telik a stack, már két elem van benne... nemsokára fogunk belőle kipakolni is.

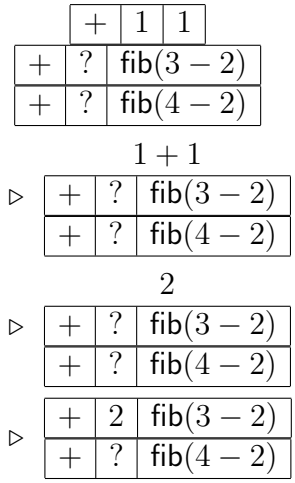


Hop, most nincs rekurzív hívás! Kaptunk egy értéket. Ezt visszaadhatjuk a hívási verembe, most már tudjuk, hogy a *legfölső* elem ?-ének a helyére 1 kerül és elkezdhetjük kiértékelni a

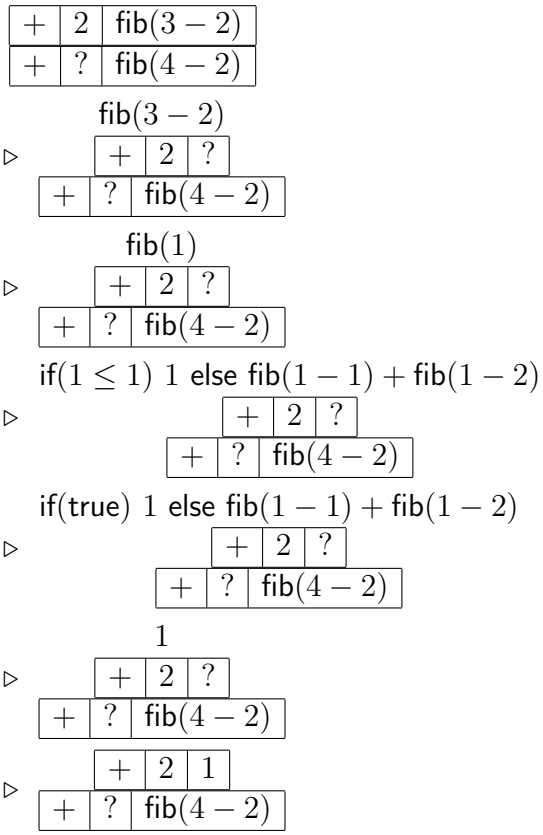
következőt:


1
+ ? fib(2 − 2)
+ ? fib(3 − 2)
+ ? fib(4 − 2)
+ 1 fib(2 − 2)
+ ? fib(3 − 2)
+ ? fib(4 − 2)
fib(2 − 2)
+ 1 ?
+ ? fib(3 − 2)
+ ? fib(4 − 2)
fib(0)
+ 1 ?
+ ? fib(3 − 2)
+ ? fib(4 − 2)
if(0 ≤ 1) 1 else fib(0 − 1) + fib(0 − 2)
+ 1 ?
+ ? fib(3 − 2)
+ ? fib(4 − 2)
if(true) 1 else fib(0 − 1) + fib(0 − 2)
+ 1 ?
+ ? fib(3 − 2)
+ ? fib(4 − 2)
1
+ 1 ?
+ ? fib(3 − 2)
+ ? fib(4 − 2)
+ 1 1
+ ? fib(3 − 2)
+ ? fib(4 − 2)

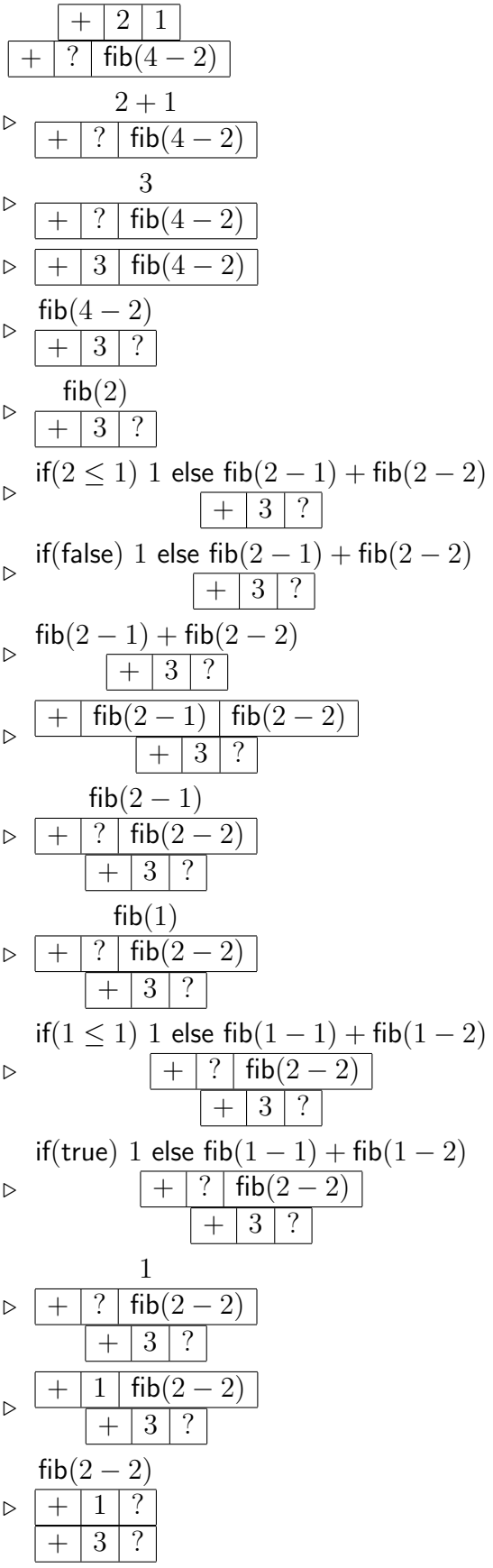
Ezen a ponton a legfölső hívásnak minden argumentumát sikerült kiértékelni, meghívhatjuk az összeadás függvényt rájuk:



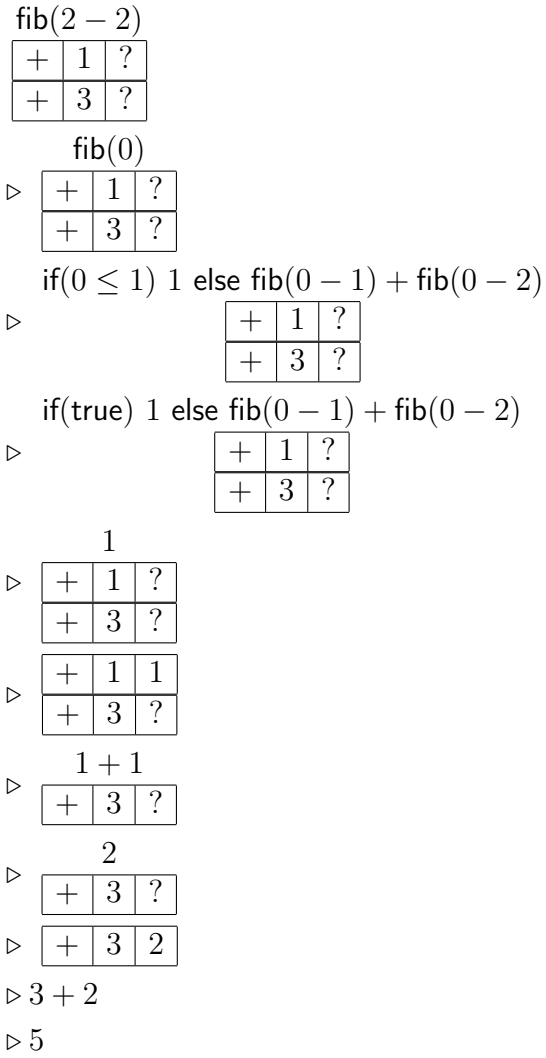
De a verem tetején lévő hívásnak még mindig csak az első argumentumát értékeltük ki. . . nézzük a másodikat is:



Alakul. . . csak lassan.



A célegyenesbe értünk talán, mindenkinek az utolsó argumentumát számoljuk már...



És ez az értéke a fib(4) kifejezésnek. A negyedik Fibonacci szám az 5. (és tényleg: $1, 1, 2, 3, 5$, nullánál kezdve az indexelést. easy)

A StackOverflowError akkor jön és szakítja meg a programunk futását, ha az itt rajzolt hívási verem mélysége (aminek a legmélyebb ponton most kb 3 volt, attól függően, hogy a tetején levő kifejezést most verem-elemnek számoljuk vagy nem) túllép egy előre beállított korlátot. Ez a korlát néhány ezres mélységet tesz lehetővé alapból.

A funkcionális paradigma alapeleme a rekurzió és a rekurzióban gondolkodás, sokszor pl. listákat is rekurzívan járunk be és dolgozunk fel (forráskódi szinten – a fordító van, hogy hatékonyabb kódot fordít belőle). Ez viszont azt jelenti, hogy egy néhány ezres lista feldolgozásakor is megtelne a hívási verem és kapnánk egy errort. Ennek a problémának a kiküszöbölése az ún. tail recursion alkalmazása (ahol lehet – nem mindig lehet), ha a programozási nyelvünk támogatja a tail recursion optimizationt. A Scala támogatja. A Java nem. Lássuk, mi is ez.

Vegyük megint a faktoriálist számító kódunkat:

```
def fact( n : Int ) : Int = if ( n == 0 ) 1 else n * fact( n - 1 )
```

Ez így ebben a formában az $n!$ kiszámításakor n mélységig tölti is a hívási vermet. Igaz, hogy ez ennél a kódnál valójában nem probléma, mert ha 1000!-t akarjuk kiszámítani, az se fog elférni a 32 bites Intünkben, de ezen most lendüljünk túl.

Nézzük a következő kódot ehelyett:

```
def fact2( n : Int , acc : Int ) : Int =
  if ( n == 0 ) acc else fact2( n - 1, n * acc )
```

Először is gondoljuk végig, mit számolhat ez így ki. Azt mondom erre, hogy az output $\mathsf{n}! * \mathsf{acc}$. Egy ilyet hogy látunk be? Megint indukcióval: az n értéke csökken és nemnegatív[6], szóval megint ér n szerinti indukciót használni. Ha $\mathsf{n} == 0$, akkor ACC jön vissza, ami tényleg épp $0! * \mathsf{acc} = 1 * \text{ACC}$, erre tehát jó a képlet. Na most ha $\mathsf{n} > 0$ és ez a képlet igaz az N-nél kisebb számokra, akkor ha n-re hívjuk a kiértékelést, a kifejezés értéke indukció szerint $(\mathsf{n}-1)! * \mathsf{n} * \mathsf{acc}$ lesz, ami meg pont $n! * \mathsf{acc}$. Vagyis, ha a $\mathsf{fact}(\mathsf{n}, 1)$ kifejezést kiértékeljük, akkor szintén $\mathsf{n}!$-t kapunk.

De mégis miért jó ez?

Próbáljuk csak ki ezt $\mathsf{n} = 5$-re:

$$
\begin{aligned}
\mathsf{fact2}(5,1) &\triangleright \mathsf{if}(5 == 0)\ 1\ \mathsf{else}\ \mathsf{fact2}(5-1, 5*1)\\
&\triangleright \mathsf{if}(\mathsf{false})\ 1\ \mathsf{else}\ \mathsf{fact2}(5-1, 5*1)\\
&\triangleright \mathsf{fact2}(5-1, 5*1)\\
&\triangleright \mathsf{fact2}(4, 5*1)\\
&\triangleright \mathsf{fact2}(4, 5)\\
&\triangleright \mathsf{if}(4 == 0)\ 5\ \mathsf{else}\ \mathsf{fact2}(4-1, 4*5)\\
&\triangleright \mathsf{if}(\mathsf{false})\ 5\ \mathsf{else}\ \mathsf{fact2}(4-1, 4*5)\\
&\triangleright \mathsf{fact2}(4-1, 4*5)\\
&\triangleright \mathsf{fact2}(3, 4*5)\\
&\triangleright \mathsf{fact2}(3, 20)\\
&\triangleright \mathsf{if}(3 == 0)\ 20\ \mathsf{else}\ \mathsf{fact2}(3-1, 3*20)\\
&\triangleright \mathsf{if}(\mathsf{false})\ 20\ \mathsf{else}\ \mathsf{fact2}(3-1, 3*20)\\
&\triangleright \mathsf{fact2}(3-1, 3*20)\\
&\triangleright \mathsf{fact2}(2, 3*20)\\
&\triangleright \mathsf{fact2}(2, 60)\\
&\triangleright \mathsf{if}(2 == 0)\ 60\ \mathsf{else}\ \mathsf{fact2}(2-1, 2*60)\\
&\triangleright \mathsf{if}(\mathsf{false})\ 60\ \mathsf{else}\ \mathsf{fact2}(2-1, 2*60)\\
&\triangleright \mathsf{fact2}(2-1, 2*60)\\
&\triangleright \mathsf{fact2}(1, 2*60)\\
&\triangleright \mathsf{fact2}(1, 120)\\
&\triangleright \mathsf{if}(1 == 0)\ 120\ \mathsf{else}\ \mathsf{fact2}(1-1, 1*120)\\
&\triangleright \mathsf{if}(\mathsf{false})\ 120\ \mathsf{else}\ \mathsf{fact2}(1-1, 1*120)\\
&\triangleright \mathsf{fact2}(1-1, 1*120)\\
&\triangleright \mathsf{fact2}(0, 1*120)\\
&\triangleright \mathsf{fact2}(0, 120)\\
&\triangleright \mathsf{if}(0 == 0)\ 120\ \mathsf{else}\ \mathsf{fact2}(0-1, 0*120) \triangleright 120
\end{aligned}
$$

Kijött a 120 érték – és közben nem telt a hívási verem!

[6] ok, ezt persze illene tesztelni, true

Amiért ez így történik: a rekurzív kódban minden egyes rekurzív hívás már „végső érték”, azzal már a továbbiakban nem csinálunk semmit. A faktoriális első implementációjában nem így történt: ott még az eredményre rászorzunk n-nel és ennek a szorzásnak az eredménye lesz az érték, de itt a fact2$(\mathsf{n}-1, \mathsf{acc})$ kifejezés már konkrétan az eredményt fogja adni.

Imperatív kódban ezt úgy vennénk észre, hogy minden rekurzív hívás eredményét azonnal returnölnénk.

Ha egy rekurzív kifejezés így épül fel, akkor tail rekurzívnak (vagy farokrekuzívnak) nevezzük és ahogy a fenti példa is mutatja, az ilyen rekurzív kifejezések nem terhelik a hívási vermet – tehát pl. listafeldolgozó műveleteket is írhatunk rekurzióval, amennyiben i) az tail rekurzió és ii) a használt programozási nyelvünk támogatja a tail call optimizationt, azaz ha ilyen esetekben tényleg nem is nyomja le a hívási verembe a kiszámítandó értéket (azaz nem nyom le minden egyes ilyen esetben egy egyedül álló ?-et az előző rész reprezentációjában a verembe). A Scala alkalmaz tail call optimizationt, tehát tail recursive függvények nem fogják felenni a stacket. A Java nem alkalmazza ezt az optimalizálást – ott a tail recursive függvények ugyanúgy stack overflowot fognak dobni.

Már csak két szépséghibája van a dolognak:

- Most egy kétváltozós függvényünk van a faktoriális kiszámítására és a második paramétert 1-re kell állítsuk, ha faktoriálist akarunk számítani. Ezt kivédjük persze egy fact(n) = fact2$(n, 1)$ deffel.
- A fact2 függvényt valójában nem akarjuk, hogy bárki meghívja, ez a függvény csak a fact számára kéne látható legyen. Scalában ez nem probléma: megtehetjük azt, hogy egy függvény scope-jában belül deklarálunk egy másik függvényt. Ekkor ez a másik függvény csak ebben a scope-ban lesz látható.

A végső faktoriális kódunk így néz ki:

```
import scala.annotation.tailrec

object Main {

   def fact( n : Int ) =
   {
      @tailrec
      def fact2( n : Int , acc: Int ) : Int =
        if( n == 0 ) acc else fact2( n - 1 , n * acc )
      fact2( n, 1 )
    }

   def main(args: Array[String]): Unit =
   {
      println( fact( 5 ) )
   }
}
```

Még egyszer: a fact-on belül van deklarálva a fact2, így kintről nem látszik. A fact implementációjában két kifejezés van: az első defeli a fact függvényt, a második pedig egy fact2$(\mathsf{n}, 1)$ hívás, és ennek az eredménye lesz a fact(n) hívás eredménye. A mainben csak kiszámoltatjuk és kiíratjuk 5!-t (és ha fact2-t próbálnánk hívni, fordítási hibát kapunk).

Két dolog van még ebben a kódban, amiről eddig nem volt szó:

- A fact2 előtt szereplő @tailrec sor. Ez egy *annotáció* (ami kukaccal kezdődik, az annotáció), ctrl-space-re kiegészítette az IntelliJ IDEA és beírta a szükséges importot is a file elejére (használd az automatikus kiegészítést, hasznos). Ha ezt beírjuk egy függvény elé, akkor azt mondjuk a fordítónak, hogy ez itt egy tail rekurzív függvény lesz, és legyen szíves optimalizálni. Optimalizálná enélkül is, ennek az annotációnak a lényege az, hogy ha az implementáció mégsem tail rekurzív, akkor a fordító hibát fog dobni. Így ha a @tailrecet beírjuk egy, szándékunk szerint tail rekurzív függvény elé és lefordul, biztosak lehetünk benne, hogy tényleg sikerült tail rekurzív függvényt írnunk és nem fog telni a hívási verem.

- A fact függvénynél az egyenlőségjel elé nem írtuk be a visszatérési típust. Nincs ott, hogy : Int. Általában nem is kell: ha a Scala fordító valaminek a típusát ki tudja következtetni, akkor nem kell odaírjuk. Ezt rekurzív függvények esetében soha nem fogja tudni megtenni, de a fact függvény nem rekurzív: az a fact2-t fogja hívni, arról a fordító pedig tudja (hiszen megmondtuk), hogy Int lesz, ezért tudja azt is, hogy a fact is Intet ad vissza.

  Ettől persze ha akarjuk, kiírhatjuk, hogy mi a típusa a függvénynek, de fölösleges.

Példa: logikai műveletek gyorsított kiértékeléssel

Sok programozási nyelven ha egy olyan kifejezést látunk, mint a false && f(0, 1), ahol f valami függvény, akkor ennek a kiértékelésekor a függvény meg se hívódik, hiszen a logikai éselés eredménye mindenképpen hamis lesz. (Ezt érdemes észben tartani, ha az f függvény valamiféle mellékhatására számítunk). Ezt hívják gyorsított kiértékelésnek.

A feladat: írjunk olyan my_and és my_or függvényeket, melyek két-két Boolean kifejezést várnak és gyorsított kiértékeléssel számítják ki a logikai és ill. vagy értéküket!

A megoldás:

```
object Main {
  def my_and( x : Boolean, y : ⇒ Boolean ) = if ( x ) y else false
  def my_or ( x : Boolean, y : ⇒ Boolean ) = if ( x ) true else y
}
```

Amire itt érdemes figyelni: ha a második argumentumot is call-by-value kérjük, akkor az nem lesz gyorsított kiértékelés, mindenképpen kiszámítaná mindkét argumentumot, mielőtt egyáltalán kifejtené a függvény törzsét. Ezért a másodikat call-by-name kell deklaráljuk. Az elsőt mindenképp ki kell számoljuk, ezt pedig call-by-name érdemes kérjük. A másik, amire felfigyelhetünk: nem írtam ki a visszatérési típusát egyik függvénynek sem, hiszen a Scala fordító ki tudja következtetni a típust: az első esetben pl. az if-else szerkezet egyik szálán y jön vissza, ami egy Boolean, a másikon false, ami szintén egy Boolean, tehát mindenképp Boolean lesz, amire kiértékelődik a kifejezés. Kapcsosba se tettem őket, hiszen az if-else szerkezet egyetlen kifejezést alkot mindkét esetben.

Hogy ez így valóban call-by-name lesz a második argumentumon, könnyen tesztelhetjük:

```
object Main {
  def loop : Boolean = loop
  def my_and( x : Boolean, y : => Boolean ) = if ( x ) y else false
  def my_or ( x : Boolean, y : => Boolean ) = if ( x ) true else y

  def main(args: Array[String]): Unit = {
    println( my_and( false, loop ))
```

```
  }
}
```

A fenti kód kiírja, hogy false. A loop metódus definíciója lehet, hogy furcsának tetszik: eddigi tudásunk alapján def loop() : Boolean = loop()-ként írtuk volna meg. Scalában ha egy metódus nem kér egyáltalán paramétert, akkor a () rész elhagyható. A konvenció az, hogy olyankor, amikor a metódus hívásának van mellékhatása (pl. kiír valamit, turkál egy adatbázisban, ilyenek), akkor kirakjuk a nyitócsukójeleket, ha pedig nincs, hanem ténylegesen egy mellékhatás nélküli, azaz pure kifejezés-kiértékelés, akkor nem szoktuk.

Ha a loop kifejezést megpróbáljuk kiértékelni, akkor végtelen (tail) rekurzióba esünk: ez azért „rosszabb" egyébként, mint a „sima" végtelen rekurzió, mert ott legalább egy idő után kidob minket az érkező StackOverflowError, itt meg a tail call optimization miatt

$$\mathsf{loop} \triangleright \mathsf{loop} \triangleright \mathsf{loop} \triangleright \mathsf{loop} \triangleright \mathsf{loop} \triangleright \dots$$

nem telik a stack, kézzel kell kilőjük a programot. (Kipróbálhatjuk mindezt pl. egy my_and(loop,false) hívással.)

Példa: tail recursive Fibonacci

A feladat: adjunk egy tail recursive implementációt a Fibonacci sorozatra.

Egy megoldás:

```
object Main {
  def fib( n : Int ) =
  {
    @tailrec
    def fib_tail( n : Int , prev : Int , curr : Int ) : Int =
    {
      if( n == 0 ) curr
      else fib_tail( n - 1 , curr, prev + curr )
    }
    if( n <= 1 ) 1 else fib_tail( n - 2, 1, 2 )
  }
}
```

Érdemes megfigyelnünk, hogy ebben az esetben nem csak a call stackkel szemben vagyunk nagyon barátságosak, de a kiszámítás ideje is sokat csökkent (kb. 1.6^n-ről n-nel arányosra). Ha végiggondoljuk, hogy itt mi történik: visszük magunkkal a sorozat utolsó két kiszámolt értékét a prev és curr argumentumokban, az n pedig azt mondja meg, hogy még milyen messze van az a tag, akit le kell kérdeznünk. Azaz: mindig $\mathsf{fib\_tail}(k, \mathsf{fib}(n-k-1), \mathsf{fib}(n-k))$ alakú hívást végzünk, k pedig a leszállási feltétel. Indukcióval megint kijön: alapból a $\mathsf{fib\_tail}(n-2, 1, 2)$ hívás pont ez $k = n-2$-re, hiszen $\mathsf{fib}(n-(n-2)-1) = \mathsf{fib}(1) = 1$ és $\mathsf{fib}(n-(n-2)) = \mathsf{fib}(2) = 2$, tehát pont így hívjuk; a következő hívásnál a $(k, \mathsf{fib}(n-k-1), \mathsf{fib}(n-k))$-ból pedig $(k-1, \mathsf{fib}(n-k), \mathsf{fib}(n-k-1)+\mathsf{fib}(n-k)) = (k-1, \mathsf{fib}(n-k), \mathsf{fib}(n-(k-1)))$ lesz, tehát az indukciós feltevés alapján rendben van.

Értékek: def, val, lazy val

Immutable (konstans) értékeket is létrehozhatunk a val kulcsszóval, a szintaxis: val név kettőspont típus egyenlő kifejezés. Itt is, ha a fordító ki tudja következtetni a kifejezés típusát, és az nekünk megfelel, akkor nem kell kiírnunk a kettőspont típus részt. Pl:

```
object Main {

  val my_pi : Double = 22.0/7

  def main(args: Array[String]): Unit =
  {
    println( "A kor kerulete osztva az atmerojevel sztem kb " + my_pi )
  }
}
```

A Double (teljes nevén scala.Double) pedig a Java double alaptípusának megfelelő osztály, pontosan ugyanazzal a számábrázolási módszerrel és értéktartománnyal. Mivel a 22.0 egy Double típusú értékké fordul, azt osztva az Int 7-tel még mindig a Double típusban vagyunk, a fenti kódban elhagyható a típus kiírása.

Felmerülhet a kérdés, hogy ezt minek, hiszen deffel is megtehetjük ugyanezt. Sőt, van egy érdekes kulcsszó kombináció, a lazy val, ami szintén látszatra ugyanezt csinálja:

```
object Main {

  val      my_pi_val     : Double = 22.0/7
  def      my_pi_def     : Double = 22.0/7
  lazy val my_pi_lazyval : Double = 22.0/7

  def main(args: Array[String]): Unit =
  {
    println( "A kor kerulete osztva az atmerojevel sztem kb " +
             my_pi_val + " vagy " + my_pi_def + " vagy + " + my_pi_lazyval )
  }
}
```

A három közti különbség akkor válik láthatóvá, ha teszünk egy mellékhatást, mondjuk egy println-t is a kifejezésbe (ezért kapcsosba teszem őket és a 22.0/7 érték lesz a hátsó, mert azt akarom értékként visszaadni) és többször is kiíratjuk őket:

```
object Main {

  val      my_pi_val     : Double = { println("val"); 22.0/7 }
  def      my_pi_def     : Double = { println("def"); 22.0/7 }
  lazy val my_pi_lazyval : Double = { println("lazy val"); 22.0/7 }

  def main(args: Array[String]): Unit =
  {
    println("Hello, starting")
    println( "A kor kerulete osztva az atmerojevel sztem kb " +
             my_pi_val + " vagy " + my_pi_def + " vagy " + my_pi_lazyval )
    println( "A kor kerulete osztva az atmerojevel sztem kb " +
```

```
            my_pi_val + " vagy " + my_pi_def + " vagy " + my_pi_lazyval )
    println( "A kor kerulete osztva az atmerojevel sztem kb " +
            my_pi_val + " vagy " + my_pi_def + " vagy " + my_pi_lazyval )
    println("End.")
  }
}
```

Az eredmény:

```
val
Hello, starting
def
lazy val
A kor kerulete osztva az atmerojevel sztem kb 3.142857142857143 vagy
    3.142857142857143 vagy 3.142857142857143
def
A kor kerulete osztva az atmerojevel sztem kb 3.142857142857143 vagy
    3.142857142857143 vagy 3.142857142857143
def
A kor kerulete osztva az atmerojevel sztem kb 3.142857142857143 vagy
    3.142857142857143 vagy 3.142857142857143
End.
```

Ebből már láthatjuk:

- a val pontosan egyszer kerül kiértékelésre, még a main-be való belépés előtt (technikailag: mindig akkor, amikor létrehozzuk az adott value-t tartalmazó scope-ot, jelen esetben a Main objektum létrehozásakor, ami persze megelőzi a Main objektum main metódusába való belépést)
- a def (ahogy láttuk eddig is a függvények kiértékelésekor) minden egyes hivatkozáskor újra és újra kiértékelődik
- a lazy val a kettő keveréke: az *első* ráhivatkozáskor értékelődik ki, aztán megtartja az értéket és a következő hívásokkor már az eltárolt értéket adja vissza.

Csábítónak hangzik minden val helyett lazy valt használni, de ezt inkább ne tegyük, az extra bookmarking miatt azokban az esetekben, amikor úgyis ki lesz értékelve az a kifejezés legalább egyszer, időt vesztünk a lazy val használatával. A def pedig akkor lehet jó választás, ha valami mellékhatást (mint fentebb a println volt) szeretnénk, ha megtörténne minden egyes kiértékeléskor, vagy ha valami időben változó értéktől (ilyet eddig nem láttunk) is függhet a kifejezés értéke.

Összetett típusok, referenciák

Láttunk és használtunk már néhány Scala „alaptípust”: ilyenek az Int, a Long (ez 64-bites előjeles egész, mint a Java long típusa), a String, a Double, a Float (ez a két lebegőtípus megfelel a Java double és float típusainak), a Char (ez a Java Characterének), vagy a String.

Nagyon gyakran egybetartozóan szeretnénk kezelni adatokat, ilyen lehet pl. egy név – cím páros, amikor is két string kéne alkosson egy logikailag egybetartozó egységet, vagy pl. ha kétdimenziós

vektorokat akarunk tárolni, aminek egy x és egy y koordinátája van. C-ben ilyeneket a struct kulcsszóval hozhattunk létre. Scalában erre a class kulcsszót fogjuk használni[7]. A példakódunk:

```
//mehet ugyanabba a fileba, mint amiben a Main object van, nem gond
class Vektor( val x : Int , val y : Int )

object Main
{
  def main(args: Array[String]): Unit =
  {
    val p = new Vektor( 1, 2 )
    val q = new Vektor( 3, 4 )

    println( p.x + "," + p.y ) // 1,2
    println( q.x + "," + q.y ) // 3,4

  }
}
```

Lássuk, mi történik itt. Egyrészt deklaráltunk egy Vektor nevű osztályt. Ennek az osztálynak két adattagja lesz, a val x és a val y paraméterek a kerek zárójelben azt mondják, hogy ha létrehozunk egy Vektort, mondjuk p néven később, akkor annak lesz egy p.x és egy p.y nevű adattagja, mindkettő Int. Ez kb megfelel egy C-beli typedef struct int x,y; Vektor;-nak, azzal a különbséggel, hogy egy ilyen Vektor létrehozáskor Scalában kötelesek vagyunk inicializálni az x,y adattagokat és (mivel a val kulcsszóval deklaráltuk őket) az objektum élettartama során az x,y értéke nem változhat.

Valamilyen módon át kell adjuk létrehozáskor az osztály egy újonnan létrehozott *példányának* azokat az adatokat, amik alapján kitölti az adattagok értékeit, erre való a *konstruktor*. Scalában a konstruktor[8] fejlécét az osztály neve mellett írjuk ki: tehát a class Vektor(val x : Int , val y : Int) rész azt mondja, hogy egy Vektor létrehozásakor a létrehozandó példánynak átadunk két Int értéket, erre a két átadott értékre lentebb x és y néven hivatkozhatunk.

A class egy példányának létrehozására a new kulcsszót használjuk: a new Vektor(1, 2) kifejezés értéke egy újonnan létrehozott („new”) Vektor típusú objektum lesz, mely konstruktáranak átadtuk az 1,2 értékeket, azaz x = 1, y = 2 paraméterekkel hozzuk létre.

Ami ilyenkor történik, az teljesen ugyanolyan, mint egy függvényhívásnál: a substitution modelnek megfelelően, call-by-value hívjuk a konstruktort és a new Vektor(1, 2) kifejezésből lesz egy Vektor{ val x = 1; val y = 2 } egység, egy objektum, az osztálynak egy példánya. Ez a példány egyébként valahol a heap memóriában, dinamikusan jön létre. A kifejezésnek az értéke az erre a példányra mutató *referencia* lesz (ami lényegében egy memóriacím), így a val p = new Vektor(1, 2) híváskor a létrehozott új Vektor példányt nem fogjuk egy-az-egyben bitenként átmásolni, hogy p megkapja ezt az értéket, hanem a p konkrétan erre az újonnan lefoglalt memóriaterületre mutató referenciát fog tárolni.

A class kulcsszóval deklarált „összetett típusok”, ha csak val adattagokat teszünk beléjük, akkor azon kívül, hogy „egyben tartanak” valamiféle logikailag összetartozó adatokat, nem sok mindenre jók. Még az == operátor viselkedését is ha teszteljük:

```
class Vektor( val x : Int , val y : Int )
```

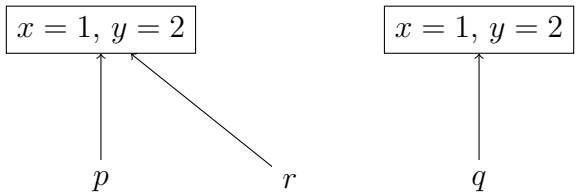
[7] egyelőre. stay tuned

[8] lehet többféle konstruktort is készíteni, stay tuned. Az alapértelmezett konstruktort így adjuk meg

```
object Main
{
  def main(args: Array[String]): Unit =
  {
    val p = new Vektor( 1, 2 )
    val q = new Vektor( 1, 2 )
    val r = p

    println( p == q ) // false
    println( p == r ) // true
    println( q == r ) // false
    println( p == p ) // true
  }
}
```

Abból, hogy a p == q hamis lesz, látjuk, hogy a Scala alapból *nem* tartja egyenlőnek egy osztály két példányát csak azért, mert az adattagjaik egyenlőek. Alapból két objektum közt akkor lesz igaz az egyenlőség, ha ez tényleg *ugyanaz* az objektum, vagyis a memóriának ugyanazt a részét foglalja el (megegyezik a kezdőcíme, a referenciája). A p == p ezért igaz; és ha megnézzük, a p == r azért lesz igaz, mert a val r = p értékadással nem azt kérjük a Scalától, hogy másolja le a p által mutatott Vektor objektum teljes tartalmát valahova a memóriának egy újonnan foglalt területére és annak a kezdőcímét rakja r-be, hanem azt, hogy az r mutasson ugyanoda, mint a p. Ezért lesz p és r egyenlőek. Rajzban:



A két new hívás a memóriában egy-egy újonnan foglalt helyen hozza létre az egyforma tartalmú, de ettől még alapból nem „egyenlő” objektumot, a p referencia az elsőre, a q a másodikra mutat, az r pedig megkapja ugyanazt, mint p a fenti kódban.

Metódusok osztályokban, operator overloading

Bővítsük az előző osztálydefiníciónkat: szeretnénk egy olyan metódust, ami két Vektort összead (a szokásos módon: az összegben az x koordináta a két x koordináta összege legyen, és ugyanez az y-ra is).

Egyfajta megoldás ez is:

```
class Vektor( val x : Int , val y : Int )

object Main
{
   def plus( u: Vektor, v: Vektor ) = new Vektor( u.x + v.x, u.y + v.y )

   def main(args: Array[String]): Unit =
   {
      val p = new Vektor( 1, 2 )
      val q = new Vektor( 3, 4 )
      val r = plus( p, q )
```

```
    println( r.x + "," + r.y ) // 4,6
  }
}
```

. . . de szemléletét tekintve ez nem igazán egy objektumorientált gondolkodás. Ennek a megközelítésnek akkor lenne létjogosultsága, ha a Vektor osztály definíciója előre adott lenne, mi mint az osztály felhasználói ahhoz nem nyúlhatnánk, és az összeadás valami olyan feature lenne, ami nem „szerves része” az osztály által biztosított funkcionalitásoknak.

Két Vektor összeadásának lehetősége azonban egy olyan „alap” funkció, amitől elvárhatja az ember, hogy ezt ne egy, a Vektortól függetlenül, pl. a Main objektumban külön implementált metódus valósítsa meg, hanem (ahogy pl. az Intek esetében is) „zárja egységbe” az osztály az adatot (ami most a két adattag) és a működést (ami most a két vektor összeadása lehet). Erre van mód: a Vektor osztálydefinícióján belül is definiálhatunk metódusokat. Ha egy C osztályon belül deklarálunk pl. egy f(x : D, y : E) metódust, akkor ezt c.f(d,e) formában hívhatjuk meg, ahol c egy C osztálypéldány, d egy D típusú kifejezés, e pedig egy E típusú kifejezés.

Tehát: osztályon belül, *tagfüggvényként* deklarált n-változós metódus kb. megfelel egy $(n+1)$-változós függvénynek, melynek az első változója mindig az osztály egy példánya lesz. Ezt a példányt, „akinek” a metódusát hívjuk, a metóduson belül a this kulcsszóval érjük el.

Mivel tehát a plus metódus két Vektorból számít ki egy újabbat, és az egyik input Vektor a this lesz, így ha az összeadást tagfüggvényként deklaráljuk, akkor *egy* argumentumot, a másik (a jobb oldali) Vektort kell várjon. Lássuk:

```
class Vektor( val x : Int , val y : Int )
{
  def plus( that: Vektor ) = new Vektor( this.x + that.x , this.y + that.y )
}

object Main
{
  def main(args: Array[String]): Unit =
  {
    val p = new Vektor( 1, 2 )
    val q = new Vektor( 3, 4 )
    val r = p.plus( q )
    val r2 = p plus( q )
    val r3 = p plus q

    println( r.x + "," + r.y ) // 4,6
    println( r2.x + "," + r2.y ) // 4,6
    println( r3.x + "," + r3.y ) // 4,6
  }
}
```

Láthatjuk, hogy a p.plus(q) hívás az, amire számíthattunk: vegyük a p Vektor példányt, neki a plus metódusát hívjuk meg a q argumentummal. A plus metódus implementációjáról annyit, hogy ha egy C osztály egy tagfüggvénye egy argumentumot vár, mely szintén C típusú (mint amilyen most a plus), akkor egy scalás kódolási konvenció ennek az argumentumnak a that nevet adni. De persze nem kötelező, bárhogy nevezhetnénk a metódus formális paraméterét.

Másik két hívási szintaxisra láthatunk példát az r2 és r3 valuek értékadásánál: egyrészt, a

pontot nem feltétlenül muszáj kirakni, a legtöbb esetben[9] whitespace is megfelel helyette.

Az r3-nál meg azt láthatjuk, hogy ha a metódus egyetlen argumentumot vár, akkor *sok esetben* nem szükséges az sem, hogy zárójelek közé rakjuk az argumentumot.

Láthatjuk, hogy ez a p plus q ez már egészen hasonlít arra, mintha egy bináris operátort írtunk volna meg, amit infix is használhatunk... ha a metódust elnevezhetnénk +-nak, akkor a scope pont és a metódus hívó zárójelek elhagyásával egy ember számára egészen könnyen olvasható és írható kódot kapnánk...

...és hát Scalában elnevezhetjük a metódusainkat *szinte* akárhogy. Lehet a neve + is:

```
class Vektor( val x : Int , val y : Int )
{
   def +(that: Vektor ) = new Vektor( this.x + that.x , this.y + that.y )
}

object Main
{
   def main(args: Array[String]): Unit =
   {
      val p = new Vektor( 1, 2 )
      val q = new Vektor( 3, 4 )
      val r = p + q

      println( r.x + "," + r.y ) // 4,6
   }
}
```

A helyzet az, hogy az így implementált Vektor összeadás többek közt épp azért hasonlít ennyire ahhoz, mint ha mondjuk két Intet adnánk össze, mert az Intek összeadása is pont így készül: valójában az Int is egy olyan osztály, melynek van egy def +(that: Int) = metódusa. Próbáljuk ki: az 5-öt és a 7-et úgy is összeadhatjuk, hogy 5.+(7), mert az 5-ből egy Int példány készül, aminek van egy + metódusa, ami argumentumnak megkaphat egy másik Intet.

A kód hatékonysága miatt nem kell most aggódni: végeredményben a fordító ezekből az Intekből a jól ismert plain old int elemi adattípust fogja készíteni és rajta a „rendes" összeadást meghívni, forráskódi szinten azonban még ez az összeadás is egy osztálynak tagfüggvényeként van reprezentálva.

Ha egy programozási nyelv támogatja a „szokásos" műveleti jelek, mint a +, - (ebből van a „kivonás" és a „negatív előjel" változat is), stb., vagy akár a függvényhívás „újradefiniálását" is a saját magunk által készített osztályokban, akkor azt mondjuk, hogy támogatja az operator overloadingot. A Scala tehát támogatja. (A Java nem. A C++ igen.)

Példa: Vektor osztály operátorokkal

A feladat: bővítsük a Vektor osztályt kivonással, ellentéttel, skaláris szorzattal, számmal való szorzással és koordináta-kijelöléssel (kerek zárójelben megadva a koordinátát, pl. u(0) adja vissza az u.x mező, u(1) pedig az u.y mező értékét).

```
class Vektor( val x : Int , val y : Int )
```

[9] Fogunk látni olyan példát, ahol ez nem így van; amíg kétségeink vannak, hogy fordul-e a kód nélküle, rakjuk ki a pontot bátran.

```
{
   def +( that: Vektor ) = new Vektor( this.x + that.x , this.y + that.y )
   def -( that: Vektor ) = new Vektor( this.x - that.x , this.y - that.y )
   def unary_-() = new Vektor( -this.x, -this.y )
   def *( that: Vektor ) = this.x * that.x + this.y * that.y
   def *( n : Int ) = new Vektor( this.x * n, this.y * n )
   def apply( i : Int ) = if ( i == 0 ) this.x else if( i == 1 ) this.y
      else throw new IllegalArgumentException("No coordinate " + i + " in a
         Vector!")
   def ==( that: Vektor ) = ( this.x == that.x )&&( this.y == that.y )

   override def toString: String = "(" + x + "," + y + ")"
}

object Main
{
   def main(args: Array[String]): Unit =
   {
      val p = new Vektor( 1, 2 )
      val q = new Vektor( 3, 4 )
      println( p+q ) // (4,6)
      println( p+q == q+p ) // true
      println( p-q ) // (-2,-2)
      println( p*q ) // 11
      println( p*2 ) // (2,4)
      println( p(0) ) // 1
      println( (p+q)(1) ) // 6
      println( -p ) // (-1,-2)
   }
}
```

A fenti implementációban láthatunk pár újdonságot:

- ha egy unary_- nevű, paraméter nélküli metódust deklarálunk, az fog lefutni, amikor az osztály egy példánya *elé* írjuk a mínusz előjelet (ez történik a println(-p) sorban)
- az apply metódus a kerek zárójeleket mint „függvényhívást” terheli túl: ha deklarálunk egy apply(i: Int) metódust, akkor az osztály példányait szintaktikailag „meghívhatjuk, mint egy függvényt”, mégpedig egy egy darab Intet váró függvényt. Persze lehet az applynak többváltozós változata is, egyszerre többet is deklarálhatunk az osztályon belül.
- a kódban ha az apply metódus nem a 0 és nem is az 1 értékek valamelyikét kapja, akkor dob egy *kivételt*. Erre még visszatérünk később, de ha nem kezeljük le (most nincs kivételkezelés a kódban), akkor ki is lövi a program futását (hasonló történt a túl mély rekurzió esetében előjövő StackOverflowError esetében is; Exception és Error közt az az egyik fő különbség, hogy az Error mindenképp ki fogja lőni a programunkat, ha előjön, nem helyreállítható hiba, az Exception pedig ugyan kivételes viselkedésre utal, de a kivétel „elkapható” a fentebbi szinteken, és lekezelhetjük valami megfelelő módon). Sokféle kivétel osztály van, ha azt akarjuk jelezni, hogy a metódus nem megfelelő paramétert kapott, akkor az IllegalArgumentException kivételosztály egy új példányát hozzuk létre newval és konstruktorba a hibaüzivel, majd azt dobjuk throwval.
- a toString metódus: ha létrehozunk egy osztályt, annak lesz – egyebek közt – egy toString

metódusa is, mely alapértelmezetten általában valami nem túl hasznos kimenetet produkál: alapértelmezetten kiírja a létrehozott objektum osztályát, kukac, a memóriacímet, ahova az objektum létre lett hozva (pl. Vektor@5c0369c4). A println metódus ha egy nem-String objektumot kap, akkor a kapott objektumnak meghívja a toString metódusát, és az így kapott Stringet írja ki.

Ezt az alapértelmezetten megkapott metódust írjuk felül egy új, a céljainknak jobban megfelelő toString metódussal. Mivel nem változtatja meg az objektumot, nem jár mellékhatással, így a konvenció, hogy nem írjuk ki utána a () jeleket. Az override kulcsszó pedig azt jelzi a fordítónak, hogy itt egy, az osztályban már létező[10] metódus implementációját áll szándékunkban felülírni; ha ilyen nevű metódus nem volt amúgy az osztályban (mert pl. elgépeltük a nevét), a fordító ránk fog kiáltani, hogy ez nem override.

- Túlterheltük az == operátort is: így már két vektort úgy hasonlítunk össze, hogy pontosan akkor lesznek egyenlőek, ha x és y koordinátájuk is megegyezik. Ezért a p+q == q+p kifejezés értéke true lesz. (A metódusok hívásának sorrendjére is létezik precedencia: itt pl. ahogy számítunk is rá, előbb a két összeadás, majd az eredmények közt az egyenlőség tesztelés fut le. Erre még visszatérünk.)

- a * operátornak is adtunk két overrideot: az egyik, amikor is egy másik vektorral szorozzuk össze (skalárisan) a vektorunkat, a másik, amikor is egy számmal. Az a metódus hívódik meg, amelyiknek megfelelő objektum áll a jobb oldalon.

Egyelőre ez a kód nem ad nekünk megoldást arra, hogy egy vektort egy számmal szorozzunk meg *balról*. Jelen tudásunk szerint ha pl. egy 3 * u alakú kifejezést szeretnénk értelemmel megtölteni, akkor az Int osztályban kéne egy def *(u : Vektor) : Vektor metódust deklarálnunk, de persze a beépített Int osztályhoz nem nyúlhatunk, tehát ez nem opció. Másrészt, az, hogy egy vektort megszorozzunk balról egy számmal, és az eredmény legyen egy vektor, az a két osztály közül inkább a Vektor osztály tulajdonságának látszik, mintsem az Intének, ezért lehetőleg egy olyan megoldást kell keressünk, amivel mindezt a Vektor osztályon belül (vagy legalábbis közel hozzá) kellene implementáljuk.

Erre még visszatérünk, amikor a companion objecteket és az implicit konverziókat nézzük.

Leszármaztatás, traitek

A funkcionális programozásban a leggyakrabban használt adattípus a *lista*. A nyelv lehetőségeinek és szintaxisának egyre mélyebb megismerésével egyre jobb lista implementációkat fogunk készíteni. Először egy Intekből álló lista adatszerkezetet implementálunk.

Amit elvárunk ettől az adatszerkezettől: egy lista vagy üres (Scalában az üres listát Nil fogja jelölni), vagy nemüres, amikor is van neki egy feje head : Int, és egy „maradék része”, ami szintén egy Int lista (vagy üres, vagy nem). Azt látjuk tehát, hogy kellene egy „alap” típus, ami Intek egy listája, üres vagy nemüres, és ezen belül „speciálisabb” típusok, az üres ill. nemüres listára. Ha valami metódus egy listát vár, akkor adhassunk oda neki akár üres listát, akár nemürest, el kéne fogadnia.

Az objektumorientált szemléletben ezt úgy is mondhatjuk, hogy a lista egy osztály, aminek leszármazott osztálya az (adattag nélküli) üres lista osztály, és leszármazott osztálya a nemüres lista osztály is. Erre kódot Scalában így írhatunk:

[10] örökölt, stay tuned

```
class Lista
class Nil extends Lista
class NemuresLista( val head : Int, val tail : Lista ) extends Lista

object Main
{
  def main(args: Array[String]): Unit =
  {
    val list = new NemuresLista( 1, new NemuresLista( 4, new NemuresLista( 2, new
        Nil() ) ) )
    println( list ) // NemuresLista@42f93a98
  }
}
```

Amit látunk ebben a kódban: leszármazott osztályt az extends kulcsszóval deklarálunk. Három osztályunk van: a Lista, a Nil és a NemuresLista. Az utóbbi kettő leszármazik a Lista osztályból, ez pl. azt jelenti, hogy ami metódus paraméterként Lista objektumot vár, annak adhatunk Nil vagy NemuresLista példányt is. Így például a NemuresLista konstruktorának is adhatunk egy Int head elem mellé akár üres, akár nemüres listát, ez lesz a lista „vége”, a fejet követő rész. A main metódusban láthatjuk, hogy hogyan is hozhatjuk létre az $(1, 4, 2)$ listát: ez egy nemüres lista, aminek a feje az 1, a farka a $(4, 2)$ lista, ami egy nemüres lista, aminek a feje a 4, a farka a (2) lista, ami egy nemüres lista, aminek a feje a 2, a farka meg az $()$ üres lista.

A println metódus meg egyelőre az alapértelmezett: kiírja a futásidejű típust (hogy ténylegesen melyik osztály példányaként hoztuk létre ezt az értéket), meg hogy a memóriában épp hol van.

Mármost, ez elég kényelmetlen így. Először is, a Nil osztály minden példánya „ugyanaz” a Nil objektum kéne legyen (legalábbis nincs sok értelme megkülönböztetni két üres listát egymástól), most meg a szokásos módon még két new Nil() sem lesz egyenlő egymással, amíg az == operátort felül nem definiáljuk. (Az üres nyitó-csukójelek most is elhagyhatók, new Nil-lal is működne a kód.)

Ha egy osztálynak egy darab „kitüntetett” példányát akarjuk létrehozni és elnevezni, arra való az object kulcsszó, melyet a következőképp használunk:

```
class Lista
object Nil extends Lista
class NemuresLista( val head : Int, val tail : Lista ) extends Lista

object Main
{
   def main(args: Array[String]): Unit =
   {
      val list = new NemuresLista( 1, new NemuresLista( 4, new NemuresLista( 2, Nil
          ) ) )
      println( Nil == Nil ) //true
      println( list ) // NemuresLista@3d921e20
   }
}
```

Ennek a kódnak az előzőhöz képesti előnyei: i) a Nil egy fix objektum, a mainbe való belépéskor már létezik, és nem kell (nem is lehet) újra meg újra létrehozni: a new Nil helyett elég Nil-t írni és ii) két üres lista mindig egyenlő lesz, mert tényleg ugyanazt az objektumot használjuk üres

lista létrehozásához.

A konstruktorhívogatást a kódból is jobb volna megúszni. Erre megoldás pl. az, ha a listáink kapnak egy olyan metódust, ami eléjük fűz egy elemet és visszaadja ezt az új listát (figyelem: az eredeti lista objektum nem változik! Csak készül egy másik lista, akinek az eredeti lista a farka):

```
class Lista
{
   def prepend( head : Int ) = new NemuresLista( head, this )
}
object Nil extends Lista
class NemuresLista( val head : Int, val tail : Lista ) extends Lista

object Main
{
   def main(args: Array[String]): Unit =
   {
      val list = Nil.prepend(2).prepend(4).prepend(1) // list = (1,4,2)
   }
}
```

Világos: a Nil az üres lista objektum, a Nil.prepend(2)-t ha kifejtjük, épp a new NemuresLista(2, Nil)-t kapjuk, ha ezen hívjuk a prepend(4)-et, akkor a new NemuresLista(4, new NemuresLista(2, Nil))-t, végül is ugyanazt az (1,4,2) listát, mint az előbb, de a kód tisztább.

Eltekintve attól, hogy a listát így felépítve „végigolvasni” pont fordított sorrendben kell, ami kevésbé teszi olvashatóvá a kódot. Erre Scalaban azt a szintaktikai cukrot tudjuk használni, hogy ha egy metódus neve :ra (kettőspontra) végződik, akkor őt *balról* írjuk az objektum mellé. Ha pl. a metódusunkat (a Scala konvenció szerint) a :: névre kereszteljük append helyett:

```
class Lista
{
   def ::( head : Int ) = new NemuresLista( head, this )
}
object Nil extends Lista
class NemuresLista( val head : Int, val tail : Lista ) extends Lista

object Main
{
   def main(args: Array[String]): Unit =
   {
      val list = 1 :: 4 :: 2 :: Nil // list = (1,4,2)
   }
}
```

Ezzel a szintaxissal már lehet együtt élni. Ami történik: ugyanúgy a Nil objektumból indulunk ki és először a 2 kerül az üres lista elé új fejelemnek, de ezt Nil.::(2) helyett (2)::Nil-ként tudjuk írni, mert a metódus nevének a végén ott a :. (Pontot ilyenkor nem teszünk sehova.) Mivel pedig unáris, a zárójeleket is elhagyhatjuk. (Ezt egyébként az előbb a prependnél is megtehettük volna, Nil prepend 2 prepend 4 prepend 1 is OK.)

Ha egy osztálynak (mint amilyen most a Lista) nincsenek adattagjai (így pl. a konstruktora is üres), akkor megfontolhatjuk, hogy ne a class, hanem a trait kulcsszóval vezessük be. Ezzel a

következőt nyerjük: egy class egyszerre csak legfeljebb egy másik classt extendelhet, de traitet tetszőlegesen sokat, így nagyobb rugalmasságunk van az osztályhierarchiánk megtervezésekor. Most a Lista osztályból készíthetünk így inkább traitet, a Nilből nem (mert nem osztály, hanem objektum) és a NemuresListaból sem (mert van adattagja):

```
trait Lista
{
   def ::( head : Int ) = new NemuresLista( head, this )
}
object Nil extends Lista
class NemuresLista( val head : Int, val tail : Lista ) extends Lista

object Main
{
   def main(args: Array[String]): Unit =
   {
      val list = 1 :: 4 :: 2 :: Nil // list = (1,4,2)
   }
}
```

A trait tehát olyasmiként alkalmazható, mint Javában egy interface (nincs adattag, egy osztály több interfacet is implementálhat, interfacek leszármazhatnak más interfacekből), azzal, hogy lehetnek benne implementált metódusok is (mint a mienkben a ::), de lehetnek benne nem implementált metódusok is (melyeket a leszármazott osztályokban, amiket majd ténylegesen példányosítani akarunk, definiálnunk kell), ebből meg olyan, mint Javában egy (adattag nélküli) abstract class.

Hogy lássunk egy példát absztrakt metódusra, lesz egy length műveletünk, ami minden listának lesz (ezért a Lista traiten belül kell deklaráljuk – de ott még nem tudunk neki „jelentést” adni), az üres lista hossza 0 lesz, a nemüresé pedig eggyel több, mint a farkáé (rekurzió!):

```
trait Lista
{
  def ::( head : Int ) = new NemuresLista( head, this )
  def length : Int
}
object Nil extends Lista
{
  override val length = 0
}
class NemuresLista( val head : Int, val tail : Lista ) extends Lista
{
  override val length = 1 + tail.length
}

object Main
{
  def main(args: Array[String]): Unit =
  {
    val list = 1 :: 4 :: 2 :: Nil // list = (1,4,2)
    println( list.length ) // 3
  }
}
```

Itt azt láthatjuk, hogy a Lista traiten belül azt mondjuk, hogy minden Lista-t extendelő objektumban / példányosítható osztályban kell legyen egy Int típusú „metódus”, aminek length a neve. Rögtön a Nil objektumban meg azt láthatjuk, hogy ott ezt a defet valként deklaráltuk – ezt megengedi a Scala szintaxis, az osztály felhasználójának csak annyit elég tudnia, hogy ha van egy list : Lista objektuma, akkor a list.length neki egy Intet ad értékül, de hogy ez egy def vagy val, az számára mindegy. Amiért a Nilben valként érdemes deklarálni: ezzel megúszunk egy fölösleges metódushívást. (Igaz, cserébe a Nil objektum maga egy kicsit nagyobb lesz[11].) A NemuresLista esetében is megtehetnénk ezt, itt is lehet def, val vagy lazy val a length „adattag”, hiszen – mivel a lista immutable, nem változik létrehozás után – a lista teljes élettartama során ugyanaz lesz az érték. Ha defet írunk, abból metódus lesz és minden híváskor újra kiszámoljuk – ennek talán az a hátulütője, hogy a metódushíváskor a futásidő a lista hosszával lesz arányos és mindig újraszámoljuk, plusz az implementációnál figyelnünk illik arra is, hogy tail rekurzív legyen, amit készítünk (tehát pl. az 1+tail.length nem lesz jó), különben egy pár ezer hosszú lista hosszának lekérdezésekor errort kapunk. Ha ehelyett valként deklaráljuk, akkor a nemüres lista osztályunkba bekerül egy adattag, ami a hosszát tárolja, és már példányosításkor ki is számítjuk. Lehetne lazy val is, azt akkor lenne indokolt használni, ha összetettebb lenne a művelet (nem pedig egy egyszerű növelés eggyel), és nem biztos, hogy hozzá akarnánk férni a programban.

Case classok és algebrai adattípusok

A jelenlegi lista implementációnk (ahogy egy funkcionális nyelvben egy lista általában is az) immutable, az adattagjai nem „változók”. Ha egy classban minden adattag val (egyelőre mást nem is láttunk), megfontolandó class helyett a case class használata. A kettő közti különbségek:

1. Szintaxis: a case class konstruktorában nem kell kiírnunk a val kulcsszót, minden deklarált adattag alapból immutable és látható lesz.
2. Szintaxis: objektum létrehozásakor nem használjuk a new kulcsszót. (Ennek oka, hogy a companion objectben készül egy túlterhelt apply metódus, lásd később.)
3. ==: a fordító készít hozzá egy egyenlőség implementációt, amiben az adattagok egyenlőségét ellenőrzi végig, nem pedig az alapértelmezett, referencia szerinti egyenlőség lesz a mérvadó.
4. mivel az egyenlőség felülíródik, ezért egy új hashcode metódust is készít a fordító, ami az adattagok hashcode-ja alapján számítódik. (Ennek fontosságát lásd később.)
5. Az osztály objektumain lehet match/case mintaillesztést végezni. (Erre hamarosan látunk példát is.)

Mivel a nemüres lista implementációnk tisztán val mezőket tartalmaz, ezért módosíthatjuk case classra:

```
sealed trait Lista
{
   def ::( head : Int ) = NemuresLista( head, this )
   def length : Int
}
object Nil extends Lista
```

[11] TODO ennek a mértékét ellenőrizni

```
{
   override val length = 0
}
case class NemuresLista( head : Int, tail : Lista ) extends Lista
{
   override val length = 1 + tail.length
}

object Main
{
   def main(args: Array[String]): Unit =
   {
      val list = 1 :: 4 :: 2 :: Nil // list = (1,4,2)
      val list2 = 4 :: 2 :: Nil
      println( 1 :: list2 == list ) // true
   }
}
```

Tehát: a case classunk konstruktorában nem írjuk ki a valt, nem írunk newot új példány gyártásakor és kapunk egy „érték szerinti” összehasonlítás operátort. Amit még ezen a kódon láthatunk, az a sealed kulcsszó a trait előtt: ezzel arról tájékoztatjuk a fordítót, hogy kizárólag ebben az egy forrásfile-ban szabad leszármaztatni a Lista traitből, máshonnan nem. Ezzel lehetővé válik a mintaillesztés (lásd később) automatikus ellenőrzése, hogy minden lehetőséget figyelembe vettünk-e.

Amit most létrehoztunk, ha csak az adattagokat nézzük, egy *algebrai adattípus*[12].

Mintaillesztés

Mondjuk, szeretnénk egy olyan metódust, mely a paraméterként kapott int listánkból készít egy másikat, melyben minden eredeti elem meg van duplázva. Ezt a metódust nem a Lista osztály tagfüggvényeként implementáljuk (feladat: készítsünk egy ezt megvalósító implementációt tagfüggvényekkel), mert nem a Lista osztályunk szerves része, hanem mondjuk a main objektum (amiről most már tudjuk, hogy miért object: mert egy példány, egy létező objektum, aminek a main metódusát futtatjuk) egy tagfüggvénye:

```
object Main
{
   def doubleList( list : Lista ) : Lista = list match {
      case Nil => Nil
      case NemuresLista( head, tail ) => (head * 2) :: doubleList( tail )
   }

   def main(args: Array[String]): Unit =
   {
      val list = 1 :: 4 :: 2 :: Nil // list = (1,4,2)
      val list2 = doubleList( list )
   }
}
```

[12] TODO írni az algebrai adattípusokról

Amit itt látunk, az a (tail) rekurzió és az immutable objektumok mellett funkcionális nyelvek egy újabb közös vonása: a mintaillesztés. A MATCH kulcsszó bal oldalán egy illesztendő kifejezés szerepel, a jobb oldalán pedig egy case $E \Rightarrow F$ esetekből álló lista, a kifejezés értéke pedig: az első olyan case jobb oldala, melynek a bal oldala illeszkedik.

Intuitíve világos, hogy mit látunk a doubleList metódusban: vegyük a kapott list listát és ezt illesszük. Ha list maga a Nil objektum (vagyis: ha üres listát kaptunk), akkor a kifejezés értéke szintén Nil lesz. Ha pedig nem, hanem egy NemuresLista jött be, akkor nevezzük el a fejét headnek, a farkát tailnek és a kifejezés értékét úgy kapjuk, hogy a tailon rekurzívan hívjuk a doubleList metódust, ez visszaad nekünk egy Listat, ami elé még betesszük a fejelem dupláját.

A Scala mintaillesztő motorja elég erős, sok mindent lehet használni mintának a case kulcsszó után:

- Lehet írni konstanst, mint 3, 5 (ha mondjuk épp egy Intet illesztünk) vagy "Sanyi" (ha épp egy Stringet). Ez akkor illeszkedik, ha az illesztendő kifejezés megegyezik (==) a konstans literállal.
- Lehet írni egy új kisbetűs változónevet. Ez bármire illeszkedik, ekkor ennek a változónak az értéke az illesztendő kifejezés lesz a CASE jobb oldalában, mikor azt kiértékeljük.

Pl. ezek szerint egy mintaillesztés alapú rekurzív megvalósítás az első n négyzetszám összegére:

```
def sumSquares( n : Int ) : Int = n match {
  case 0 => 0
  case n => n*n + sumSquares( n - 1 )
}
```

Ugyanez tail rekurzívan:

```
def sumSquares( n : Int ) = {
  def sumSquares( n : Int , acc : Int ) : Int = n match {
    case 0 => acc
    case n => sumSquares( n - 1, n*n )
  }
  sumSquares( n, 0 )
}
```

Itt pl. a mintaillesztést az n Intre használtuk, ha egyenlő 0-val, visszaadjuk a nullát (vagy az akkumulátort a második megvalósításban), egyébként meg rekurzívan összeadjuk az első $n-1$ négyzetszámot és az összeghez hozzáadjuk az n^2-et.

- Lehet írni P:T alakú kifejezést, ahol P egy minta, T pedig egy típus. Az illesztendő kifejezés akkor illik erre a mintára, ha T típusú és illeszkedik a P mintára.
- Lehet írni C(P1,...,Pk) alakú kifejezést, ahol C egy case class, melynek van k-változós konstruktora[13]. Az illesztendő kifejezés akkor illik erre a mintára, ha (futásidejű) típusa épp a C osztály, és ami konstruktorral létrehoztuk, abban a paraméterek rendre illeszkednek a P1, . . . , Pk mintákra.

Ez történt előbb a duplázás második case esetében.

[13] később, amikor több konstruktorunk is lehet, itt elképzelhető, hogy meg kell adjuk explicit a belső patternek típusát Pi : Ti alakban

Lehet továbbá ún. case guardokat is használni:

- Lehet írni P if B alakú kifejezést, ahol P egy minta, B pedig egy Boolean kifejezés. Az illesztendő kifejezés illik erre a mintára, ha illeszkedik P-re és a B guard kifejezés értéke is igaz lesz.
- Lehet írni _ (underscore) jelet is, ez mindenre illeszkedik. A különbség változónév és e közt, hogy ez a jel nem boundolja az illesztendő kifejezés értékét, így ha azt nem használnánk, ahelyett ezzel illeszteni gyorsabb.

Feladat: Lista.toString

Overridoljuk a Lista toString metódusát: zárójelek közt írjuk ki vesszővel szeparálva a lista tartalmát. Ehhez használhatunk egy innerToString metódust, mely (pl. rekurzívan) nyitójel nélkül írja ki a listánkat, és a toString ez elé tegye a nyitójelet. (Gondoljuk meg: miért csináljuk így?)

```
sealed trait Lista {
  def ::(head: Int) = NemuresLista(head, this)
  protected def innerToString : String
  override def toString = "(" + innerToString
}
object Nil extends Lista {
  override val innerToString = ")"
}
case class NemuresLista( head : Int, tail : Lista ) extends Lista
{
  override val length = 1 + tail.length
  override def innerToString : String = tail match {
    case Nil => head + Nil.innerToString
    case tail : NemuresLista => head + "," + tail.innerToString
  }
}

object Main
{
  def doubleList( list : Lista ) : Lista = list match {
    case Nil => Nil
    case NemuresLista( head, tail ) => (head * 2) :: doubleList( tail )
  }
  def main(args: Array[String]): Unit =
  {
    val list = 1 :: 4 :: 2 :: Nil // list = (1,4,2)
    val list2 = doubleList( list ) // (2,8,4)
    println( list2 )
  }
}
```

Amit ebben a megoldásban pluszban látunk: a protected kulcsszó azt eredményezi, hogy az innerToString metódust csak a Lista traitet extendelő osztályokban látjuk, „kívülről” (pl. a mainből) nem. Így pl. a mainben egy list.innerToString kifejezés fordítási hibát okoz. Általában is jó gyakorlat csak azt engedni, hogy az osztályunk felhasználója lássa, amit szeretnénk en-

gedni neki, hogy meghívjon, és a csak „belső” működéshez kellő metódusokat elrakjuk a szeme elől, protectedre állítva annak láthatóságát (visibility). Egyéb láthatóságok még: public (ez a default, ekkor látszik mindenki számára) és private (ami traitben / absztrakt osztályban nem megengedett: ezt még a leszármazott osztályokból sem látni, csak és kizárólag abban az osztályban látszik, melyben deklaráljuk).

Feladat: mergesort

Implementáljunk egy mergesort(list : Lista) : Lista metódust, mely összefésülő rendezi az input listát (természetesen mivel a listánk immutable, egy új listában adja vissza a rendezett értékeket).

```
def split( list : Lista ) : (Lista, Lista) = list match {
  case Nil => (Nil, Nil)
  case NemuresLista( _, Nil ) => ( list, Nil )
  case NemuresLista( head1, NemuresLista( head2, tail )) => {
    val (list1,list2) = split( tail )
    ( head1 :: list1, head2 :: list2 )
  }
}
def merge( list1 : Lista, list2 : Lista ) : Lista = (list1, list2) match {
  case (Nil,_) => list2
  case (_,Nil) => list1
  case ( NemuresLista(head1, tail1), NemuresLista( head2, _ )) if head1 < head2 =>
     head1 :: merge( tail1, list2 )
  case ( _, NemuresLista( head2, tail2 )) => head2 :: merge( list1, tail2 )
}
def mergesort( list : Lista ) : Lista = list match {
  case Nil => Nil
  case NemuresLista( _, Nil ) => list
  case _ => {
    val (list1, list2) = split( list )
    val sortedList1 = mergesort( list1 )
    val sortedList2 = mergesort( list2 )
    merge( sortedList1, sortedList2 )
  }
}
```