Funkcprog 1

Rekurzió és tail rekurzió

Rekurzió: Iteráció helyett használjuk, a rekurzív függvény, olyan függvény, amely végrehajtása során önmagát hívja meg. Intuitívnak kell lennie.

Mindene egyes lépéssel mélyebbre kerülünk a hívási veremben, mivel a hívás mérete véges ezért előfordulhat, hogy betelik és StackOverFlowError-t kapunk.

Tail call optimization

A tail call optimization (TCO) egy olyan optimalizációs technika a programozási nyelvekben, amely segít elkerülni a hívási verem (call stack) túlcsordulását, különösen rekurzív függvények esetén. A lényege, hogy a rekurzív hívásokat úgy alakítsuk ki, hogy azok a "tail position" (végeredmény pozíció) helyzetben legyenek.

A tail position az a helyzet, amikor egy függvény utolsó lépése egy másik függvény hívása, és az eredményét nem kell tovább feldolgozni vagy manipulálni, azaz azonnal vissza lehet adni. Az optimalizáció lényege, hogy a fordító vagy értelmező megpróbálja kiváltani a rekurzív hívást úgy, hogy ne szülessen újabb hívási keret a veremben, hanem egyszerűen visszalépjen az eredeti függvény kezdetére.

Ez az optimalizáció előnyei és hátrányai vannak:

Előnyök:

1. **Stack méret csökkenése:** Mivel a rekurzív hívások nem növelik a hívási verem méretét, csökken a stack túlcsordulásának kockázata.
2. **Teljesítmény javulása:** Mivel nem szükséges újabb és újabb hívási kereteket létrehozni a veremben, a program futása hatékonyabb lehet.

Hátrányok:

1. **Debugolási nehézségek:** A tail call optimization által generált kód nehezebben követhető és debugolható. Mivel a verem keretek helyét felülírják, nehéz lehet nyomon követni a változók értékeit a hívások során.
2. **Nyelvi támogatás:** Nem minden programozási nyelv támogatja automatikusan a tail call optimization-t, és a nyelvi implementációtól függ, hogy a fordító vagy értelmező alkalmazza-e ezt az optimalizációt.

Tehát ahhoz hogy kihasználjuk a farokrekurziót, ahhoz a függvényt tail recursive alakra kell hoznunk.

A tail rekurzió esetén a rekurzió hívás lesz az utolsó művelet a függvényben. Tehát a rekurzív hívás a végeredmény pozícióban van, így nem tölti a hívási vermet.

Scalaban van function overloading azaz lehet ugyanaz a név ha más a paraméter listája.

Jobbá tehetjük a függvényt ha használjuk a tailrec és nesting annotációt.

-tailrec annotáció: nem kötelező a fordító figyelmeztetést ad ha nem sikerült a farokrekurzió.

-nesting annotáció: a scopeon belül deklarált értékek nem láthatóak kívülről, ha több kifejezés van egymás után egy blokkban akkor az utolsó kifejezés értéke lesz az eredmény

Iteratív ciklusok

A Scala nem idiomatikus (nyelvérzékeny), de rekurzióval és if feltételekkel lehet készíteni for ciklus.

Lényegében csak kifejezéseket értékelünk ki, az eredményeket eldobjuk, ezért a ciklus kifejezésnek kell egy típus (pl.: Unit)

Ha nem rakunk else ágat a fordító odagenerál egyet ami vagy jó lesz vagy nem, úgyhogy érdemes nem elfelejteni.

Tuple

A Scála erősen típusos nyelv, vagyis mindennek van valamilyen típusa. Vannak beépített típusok Int, Long vagy String, de létrehozhatunk saját típust is, ilyen lehet például a tuple is amely párban vagy valahanyasban egyben tárol változó típusokat.

Deklarálása:

-explicit típus: val tuple1: (Int, Int) = (3,4)

-inferred típussal: val tuple2 = (5,6)

-vegyes: val tuple3 = („Sándor”, 42, false)

Összefoglalva: a tupleben lehet 1 vagy 2 vagy akár többféle mező is, a sorrend fontos és mindegyik értéknek saját típusa van.

Ha a tuple valamelyik koordinátáját el szeretnénk érni akkor azt a pont operátorral tudjuk elérni, alsó vonal+indexxeléssel: pintln(tuple._2)

Case class:

Ahhoz hogy az általunk létrehozott saját típus intuitív legyen ahhoz használhatunk case class-t amivel el tudjuk nevezni a típusunkat. A kuzuson immutable struktúraként kezeltünk, tehát nem változtathatjuk meg futásidőben az értékét.

pl.:

-case class Pont(x: Int, y: Int)

-explicit: val p: Pont = Pont(1,3)

-inferred: val q = Pont(2,3)

-mezők neveivel: val r = Pont(x=2, y=3)

-nem muszáj tartani a sorrendet: val s = Pont(y=3, x=2)

Gyakran szorzat típusnak is nevezik a case classt, hiszen a mezők értékeinek vesszük a direkt vagy (Descartes) szorzatát.

Hashcode:

== operátor amely a két objektum memóriacímből számított értéke egyenlő

Match kifejezések Scalában

A funkcionális programozásban nagyon gyakori az algebrai adattípusok mintaillesztéssel való feldolgozása a match kulcsszóval.

Fentről lefelé kész van a blokkjában, amelyben ha az illesztett érték a mintaX illeszkedik a mintára akkor az egész kifejezés értéke a kifejezésX érték lesz, különben az illesztő követi a következő case-t.

Hogyan nézhet ki minta:

-azonosító: változó neve, azazonosítóhoz kötjük a változó értékét.

-érték: az illeszkedés akkor történik amikor a bejövő érték megegyezik a konkrét értékkel

-case class: egy case class nevével és annak mezőinek mintájával illeszkedik, az illeszkedés akkor történik meg ha a bejövő érték azonos nevű case class példánya és a mezőit illeszkednek a megadott mintákra

-_(underscore): mindenre illeszkedik, (default) viszont nem bindeli az értéket változóhoz.

- minta után kettőspont + típus: akkor illeszkedik ha a minta illeszkedik és az érték futásidejű típusa megegyezik a kettőspont utáni típussal

-használhatunk if guardot, amivel feltételt adhatunk meg a mintára és tovább tudjuk szűkíteni az illeszkedést

-’(backtick): ha backtickek közé rakunk egy azonosítót akkor az nem új lesz hanem már egy létező azonosító a mintában és ehhez fog megpróbálni illeszkedni.