Funkcionális nyelvek

Király, Roland

Funkcionális nyelvek

Király, Roland

Publication date 2011

A tananyag a TÁMOP-4.1.2-08/1/A-2009-0046 számú Kelet-magyarországi Informatika Tananyag Tárház projekt keretében készült.

A tananyagfejlesztés az Európai Unió támogatásával és az Európai Szociális Alap társfinanszírozásával valósult meg.

Nemzeti Fejlesztési Ügynökség http://ujszechenyiterv.gov.hu/ *06 40 638-638*

Tartalom

1. fejezet - Funkcionális nyelvek

1. A funkcionális programozási nyelvek világa

1.1. A funkcionális nyelvekről

A funkcionális programozási nyelvek világa még a programozók között sem igazán közismert. Legtöbbjük az objektum orientált, valamint az imperatív nyelvek használatában jártas, és egyáltalán nem rendelkezik ismeretekkel az előbbiekről. Sokszor azt is nehéz elmagyarázni, hogy egy nyelv mitől funkcionális...

Ennek számos oka van, többek között az, hogy ezek a nyelvek vagy speciális célokra készültek, és ezáltal nem terjedhettek el széles körben, vagy olyan bonyolult őket használni, hogy az átlag programozó hozzá sem kezd, vagy ha igen, akkor sem képes felnőni a feladathoz. Az oktatásban - néhány követendő kivételtől eltekintve - sem igazán találkozhatunk ezzel a programozási paradigmával. Az oktatási intézmények nagy részében szintén az imperatív és az OO nyelvek terjedtek el, és a jól bevált módszereket nehezen váltják fel újakkal. Mindezek ellenére érdemes komolyabban foglalkozni az olyan funkcionális programozási nyelvekkel, mint a Haskell, a Clean, és az Erlang. A felsorolt nyelvek széles körben elterjedtek, jól elsajátíthatók, logikus felépítésűek, és az iparban is alkalmazzák némelyiket.

Mindezen tulajdonságaikból kifolyólag ebben a jegyzetben is a felsorolt nyelvek közül választottunk ki kettőt, de nem feledkeztünk el a jelenleg feltörekvőben lévő nyelvekről sem, mint az F#, ezért minden fejezetben e három nyelven mutatjuk be a nyelvi elemeket, és a hozzájuk tartozó példaprogramokat.

A fentiek alapján, és abból kiindulva, hogy a kedves olvasó ezt a könyvet a kezében tartja, feltételezhetjük, hogy a funkcionális nyelvekhez csak kis mértékben, vagy egyáltalán nem ért, de szeretné elsajátítani a használatukat.

Ebből az okból kifolyólag azt a módszert alkalmazzuk, hogy a különleges funkcionális nyelvi elemeket az imperatív és az OO program konstrukciókból vett példákkal magyarázzuk, valamint megpróbálunk párhuzamokat vonni a paradigmák között.

Abban az esetben, ha nem tudunk olyan " hagyományos” programozási nyelvekből vett elemeket találni, amelyek az adott fogalom analóg megfelelői, akkor a mindennapi életből, vagy az informatika egyéb területeiről merítünk. Ezzel a módszerrel bizonyosan elérjük a célunkat.

A jegyzet tehát a gyakorlati képzést, és az említett programozási nyelvek gyakorlati oldalról való megközelítését tűzte ki célul. Természetesen nagy hangsúlyt fektetünk a nyelvek tanulásának elméleti kérdéseire is, de nem ez az elsődleges célunk. A bemutatásra kerülő példaprogramok elkészítését a programozási stílus elsajátítása érdekében javasoljuk, s, hogy ez a feladat ne okozzon problémát, a programok forráskódjait ahol csak lehet, lépésről-lépésre bemutatjuk.

1.2. A funkcionális nyelvekről általában

A funkcionális nyelvek tanulását érdemes az elmélet megismerése mellett a paradigma filozófiai hátterének vizsgálatával kezdeni. Érdemes továbbá megvizsgálni a funkcionális nyelvek legfőbb jellemzőit.

A paradigma megismerése során kitérünk arra is, hogy miért, és mikor érdemes ezeket a nyelveket használni. A funkcionális nyelvek egyik előnyős tulajdonsága a kifejezőerő, ami azt jelenti, hogy viszonylag kevés forráskóddal sok mindent le tudunk írni. Ez a gyakorlatban annyit tesz, hogy bonyolult problémákat tudunk megoldani viszonylag rövid idő alatt, a lehető legkisebb energia befektetésével.

A funkcionális programok nyelvezete közel áll a matematika nyelvéhez. A matematikai formulák szinte egy az egyben átírhatók funkcionális nyelvi elemekre. Ez megint nagyon hasznos, ha figyelembe vesszük azt a tényt, hogy a programozás nem a fejlesztőeszköz elindításával és a program megírásával kezdődik, hanem a tervezési fázissal. Elsőként a program matematikai modelljét kell megalkotni, majd el kell készíteni a specifikációját, és csak ezután jön az a munkafolyamat, amely során a program szövegét begépeljük a szövegszerkesztőbe.

Nézzünk meg a funkcionális nyelvek használatára egy konkrét példát úgy, hogy az elsajátítani kívánt nyelvek egyikét sem ismerjük. Legyen az első példa az `n!` kiszámítása bármely `n` érték mellett. Ez a probléma annyira általános, hogy szinte minden programozási tankönyvben megtaláljuk egy változatát.

1.1. programlista. Faktoriális függvény - Erlang

```
factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).
```

Láthatjuk, hogy a forráskód, ami Erlang nyelven íródott igen egyszerű, és nem sokban különbözik a matematikai formulákkal definiált függvénytől, ami az alábbi módon írható le:

1.2. programlista. Faktoriális függvény

```
fakt n = n*fakt n-1
```

Ez a leírás, ha Clean nyelven írjuk le, még inkább hasonlít a matematikai formára.

1.3. programlista. Faktoriális függvény – Clean

```
fakt n = if (n==0) 1
          (n * fakt (n-1))
```

A következő hasznos tulajdonság a kifejező erő mellett az, hogy a rekurzió megvalósítása nagyon hatékony. Természetesen készíthetünk rekurzív programokat imperatív nyelveken is, de ezeknek a vezérlési szerkezeteknek nagy hátránya, hogy a rekurzív hívások száma korlátozott, ami azt jelenti, hogy több-kevesebb lépés után mindenképpen megállnak.

Ha a bázisfeltétel nem állítja meg a rekurziót, akkor a program-verem telítődése mindenképpen meg fogja. Funkcionális nyelvekben lehetőség van a rekurzió egy speciális változatának a használatára. Ez a konstrukció a `tail-recursion`, amely futása nem függ a program-veremtől, vagyis ha úgy akarjuk, soha nem áll meg.

1.4. programlista. Farok-rekurzív hívások

```
f()->
    ...
    f().
g1() ->
    g1().
```

A konstrukció lényege, hogy a rekurzív hívások nem használják a vermet (legalábbis nem a megszokott módon). A verem kiértékelő és gráf átíró rendszerek mindig a legutolsó rekurzív hívás eredményével térnek vissza. A farok-rekurzió megvalósulásának az a feltétele, hogy a rekurzív függvény utolsó utasítása a függvény önmagára vonatkozó hívása legyen, és ez a hívás ne szerepeljen kifejezésben (1.5. programlista).

1.5. programlista. Farok-rekurzív faktoriális függvény - Erlang

```
fact(N) -> factr(N, 1).

factr(0, X) -> X;
factr(N, X) -> factr(N-1, N*X).
```

1.6. programlista. Farok-rekurzív faktoriális függvény – Clean

```
fact n = factr n 1

factr 0 x = x
factr n x = factr (n-1) (n*x)
```

1.7. programlista. Farok-rekurzív faktoriális függvény – F#

```
let rec fakt n =
      match n with
      | 0 -> 1
      | n -> n * fakt(n-1)

let rec fakt n = if n=0 then 1 else n * fakt (n-1)
```

A rekurzív programokat látva és azt a tényt figyelembe véve, hogy a rekurzió erősen korlátozott futással bír, felmerülhet bennünk az a kérdés, hogy egyáltalán mi szükség van rá, hiszen a rekurzív programokhoz mindig találhatunk velük ekvivalens iteratív megoldást. Ez az állítás igaz, ha nem egy funkcionális nyelvekről szóló tankönyvben állítjuk. A funkcionáls nyelvekben nem, vagy csak nagyon ritka esetekben találunk olyan - iterációs lépések megvalósítására szolgáló - vezérlő szerkezeteket, mint a `for`, vagy a `while`. Az iteráció egyetlen eszköze a rekurzió és így már érthető, hogy miért van szükség a farok-rekurzív változatra.

1.8. programlista. Rekurzív szerver - Erlang

```
loop(Data) ->
   receive
     {From, stop} ->
            From ! stop;
       {From, {Fun, Data}} ->
           From ! Fun(Data),
            loop(Data);
  end.
```

Az Erlang nyelvben - és más funkcionális nyelvek esetében is - nem ritka az sem, hogy a kliens-szerver programok szerver része rekurziót használ a tevékeny várakozás megvalósítására. A kliens-szerver alkalmazások szerver oldali része minden egyes kérés kiszolgálása után új állapotba kerül, vagyis rekurzívan meghívja önmagát az aktuálisan kiszámolt adatokkal. Amennyiben az 1.8. programlistában látható ismétlést a rekurzió hagyományos formájával szeretnénk megoldani, a szerver alkalmazás viszonylag hamar leállna a `stack owerflow` üzenet valamely, az adott szerveren használatos változatával. Az itt felsorolt néhány jellemző mellett a funkcionális nyelvek még számos, csak rájuk jellemző tulajdonsággal rendelkeznek, de ezekre a jegyzet más részeiben térünk ki. Részletesen bemutatjuk tehát a funkcionális nyelvek speciális elemeit, a rekurzió hagyományos és farok-rekurzív változatát, a halmazkifejezések használatát, valamint a lusta és a szigorú kifejezés-kiértékelés problémáit.

Ahogy említettük, a fejezetek határozottan gyakorlat-centrikusak, és nem csak a "tisztán" funkcionális nyelvi elemek használatát ismertetik, hanem a funkcionális nyelvek ipari felhasználásánál alkalmazott üzenetküldésre, vagy a konkurens programok írására is kitérnek.

A jegyzet három ismert funkcionális nyelven mutatja be a programozási paradigmát. Az egyik az Erlang, a másik a Clean, a harmadik az F#. Az Erlangot fő nyelvként használjuk, mivel az ipari alkalmazása nagyon jelentős, és rendelkezik minden olyan tulajdonsággal, mely a funkcionális paradigma előnyeinek kihasználása mellett alkalmassá teszi az elosztott és hálózati programok írására.

A Clean-re második nyelvként azért esett a választás, mert a funkcionális nyelvek közül talán a legszemléletesebb, és nagyon hasonlít a Haskell-re, ami a funkcionális nyelvek klasszikus darabjának számít. Az F# a harmadik nyelv, melynek használata jelenleg rohamosan terjed, ezért nekünk sem szabad elfeledkezni róla. A példaprogramok magyarázata főként az Erlang forrásokra vonatkozik, de ahol lehetséges, a Clean és F# nyelvű programrészek magyarázata is helyet kapott.

1.3. Erlang

Az Erlang nyelvet az Ericsson és az Ellemtel Computer Science Laboratories fejlesztette ki. Olyan programozási nyelv, mely lehetővé teszi valós idejű elosztott, és különösen hibatűrő rendszerek fejlesztését. Az Ericsson az Erlang Open Telecom Platform kiterjesztését használja telekommunikációs rendszerek kifejlesztésére. Az OTP használata mellett megoldható a megosztott memória nélküli adatcsere az alkalmazások között. A nyelv

támogatja a különböző programozási nyelveken írt komponensek integrálását. Nem "tisztán" funkcionális nyelv, sajátos típusrendszere van, valamint kitűnően használható elosztott rendszerek készítésére.

1.4. Clean

A Clean funkcionális nyelv, amely számos platformon és operációs rendszeren hozzáférhető, és ipari környezetben is használható. Rugalmas és stabil integrált fejlesztői környezettel rendelkezik, ami egy szerkesztőből és egy projekt manager-ből áll. A Clean IDE az egyetlen fejlesztőeszköz, amelyet teljes egészében tiszta funkcionális nyelven írtak. A fordítóprogram a rendszer legösszetettebb része. A nyelv moduláris, definíciós modulokból (`dcl`), valamint implementációs fájlokból (`icl`) áll. A fordító a forráskódot platform független ABC kódra fordítja (.abc fájlok).

1.5. FSharp

A .NET keretrendszerben meghatározó szerepet fog betölteni a funkcionális programozási paradigma. A funkcionális paradigma új nyelve az F#. A nyelvet a Microsoft a .NET keretrendszerhez fejleszti. Az F# nem tisztán funkcionális nyelv, támogatja az objektum orientált programozást, a .NET könyvtárak elérését, valamint az adatbázis kezelést. F#-ban lehetőség van SQL-lekérdezéseket leírni (meta nyelvtan segítségével), és ezeket SQL-re fordítani külső értelmező használatával. Az F# nyelven írt kódokat fel lehet használni C# programokhoz, mivel azok szintén hozzáférnek az F# típusokhoz.

2. Funkcionális programok általános jellemzői

2.1. Tiszta és nem tiszta nyelvek

Mielőtt elkezdenénk a nyelvi elemek részletes tárgyalását, néhány fogalmat mindenképpen tisztáznunk kell, és meg kell ismerkednünk a funkcionális nyelvek alapjaival.

A funkcionális nyelvek között léteznek tiszta, és nem tisztán funkcionális nyelvek. A LISP, az Erlang, az F#, és még néhány ismert funkcionális nyelv nem tisztán funkcionálisak, mivel a függvényeik tartalmaznak mellékhatásokat (valamint néhány nyelv esetében destruktív értékadást).

A mellékhatások, valamint a destruktív értékadás fogalmára később visszatérünk...

Tiszta nyelv a Haskell, ami az input, és az output kezelésére a `Monad` technikát használja. A Haskell mellett tiszta nyelvnek számít még a Clean és a Miranda. A funkcionális nyelvek alapja a lambda-kalkulus, és a funkcionális programok általában függvény definíciók sorozatából és egy kezdeti kifejezésből állnak. A program futtatása során a kezdeti kifejezés kiértékelésével juthatunk el a program végeredményéhez. A futtatáshoz legtöbbször egy redukciós, vagy gráf-átíró rendszert használunk, amely a programszövegből felépített gráfot behelyettesítések sorozatával redukálja.

A matematikai logikában, és a számítástudományban a Lambda-kalkulus a függvény definíciók, függvény alkalmazások, és a rekurzió formális eszköze. Alonzo Church vezette be az 1930-as években a matematika alapjai kutatásának részeként. A Lambda-kalkulus alkalmas kiszámítható függvények formális leírására. Ezen tulajdonsága miatt válhatott az első funkcionális nyelvek alapjává.

Mivel minden Lambda-kalkulussal definiálható függvény Turing-kiszámítható, minden *kiszámítható* függvény leírható lambda-kalkulussal. Később ezekre az alapokra támaszkodva kifejlesztették a kibővített Lambda-kalkulust, amely már tartalmaz az adatokra vonatkozóan típus, operátor és literál fogalmat. A programok fordítása során a kibővített Lambda-kalkulus programjait először az eredeti Lambda-kalkulusra alakítják át, és ezt implementálják. A kibővített Lambda-kalkuluson alapuló nyelvek például az ML, Miranda és a Haskell...

2.2. Term újraíró rendszerek

A Lambda-kalkulus egy alternatív változata a Term Újraíró Rendszer (TRS). Ebben a modellben a funkcionális program függvényei újraírási szabályoknak, a kezdő kifejezése pedig egy redukálható termnek felel meg. A TRS általánosabb modell a Lambda-kalkulusnál. Nem determinisztikus működést is le lehet írni vele. A Lambda-kalkulusban a legbaloldalibb, legkülső levezetések sorozata vezet el a normál formához, a TRS-ben a párhuzamos-legkülső redukciókkal lehet biztosítani a normál formaelérését. Felhasználható funkcionális

nyelvek implementálására, de figyelembe kell venni azt a problémát, hogy nem biztosítja az azonos kifejezések egyszeri kiszámítása.

2.3. Gráf újraíró rendszerek

A TRS rendszerek általánosítása a gráf újraíró rendszer (GRS), mely konstans szimbólumokon, valamint gráf csúcsok nevein értelmezett újraírási szabályokból áll. A funkcionális program kezdő kifejezésének egy speciális kezdő gráf, a függvényei pedig a gráf újraírási szabályoknak felelnek meg. A leglényegesebb különbség a két újraíró rendszer között az, hogy a GRS-ben az azonos kifejezések kiszámítása csak egyszer történik meg. Ez a tulajdonsága sokkal inkább alkalmassá teszi funkcionális nyelvek implementációjára. Az implementáció általános módszere, hogy a programot TRS formára alakítják, majd GRS-t készítenek belőle, végül gráf redukcióval normál formára hozzák.

A funkcionális nyelvek mintaillesztése, és az ebben használt prioritás alkalmazható a gráf újraírási szabályainak érvényesítése során is. Az ilyen redukciós módszert alkalmazó rendszereket funkcionális GRS-nek (FGRS) nevezik.

Funkcionális programnyelvek implementációiban FGRS-ket használva könnyen leírhatók a logikai és imperatív nyelvek olyan nem funkcionális tulajdonságai is, mint a mellékhatás.

2.4. Alapvető nyelvi konstrukciók

A funkcionális nyelvek - legyenek tisztán, vagy nem tisztán funkcionálisak - tartalmaznak számos olyan konstrukciót, melyeket az OO nyelveknél nem, vagy csak korlátozott funkcionalitás mellett találhatunk meg, valamint rendelkeznek olyan tulajdonságokkal, amelyek szintén csak a funkcionális nyelveket jellemzik.

2.5. Függvények és rekurzió

A funkcionális programok írása során függvény definíciókat készítünk, majd a kezdeti kifejezés kiértékelésével - ez is egy függvény - elindítjuk a program futását. Egy függvény meghívhatja önmagát, funkcionális nyelvek esetén farok-rekurzió alkalmazása mellett akárhányszor.

2.6. Hivatkozási helyfüggetlenség

A kifejezések értéke nem függ attól, hogy a program szövegében hol fordulnak elő, vagyis bárhol helyezzük el ugyanazt a kifejezést, az eredménye ugyanaz marad. Ez a tisztán funkcionális nyelvekre igaz leginkább, ahol a függvényeknek nincs mellékhatása, így a kiértékelésük során sem változtatják meg az adott kifejezést.

2.7. Nem frissíthető változók

A funkcionális nyelvekre jellemző, de az OO programozók körében legkevésbé kedvelt technika a nem frissíthető változók használata. A funkcionális programozási nyelvek nem engedik meg a destruktív értékadást (a változók többszöri kötését), vagyis az `I = I +1` típusú értékadásokat.

2.8. Lusta és mohó kiértékelés

A funkcionális nyelvek kifejezés kiértékelése lehet lusta (*lazy*), vagy mohó (*strict*). A lusta kiértékelés során a függvények argumentumai csak akkor értékelődnek ki, ha azokra feltétlenül szükség van. A Clean ilyen lusta kiértékelést alkalmaz, míg az Erlang inkább a szigorú nyelvek közé tartozik. A mohó kiértékelés minden esetben kiértékeli a kifejezéseket, mégpedig olyan hamar, ahogyan az lehetséges. A lusta kiértékelés a `inc 4+1` kifejezést elsőként `(4+1)+1` kifejezésként interpretálná, míg a mohó rendszer azonnal `5 + 1` formára hozná.

2.9. Mintaillesztés

A mintaillesztés függvényekre alkalmazva azt jelenti, hogy a függvény több ággal rendelkezhet, és az egyes ágakhoz (*clause*) a formális paraméterek különböző mintáit rendelhetjük hozzá. A függvény hívásakor mindig az az ág fut le, amelynek a formális paramétereire a híváskor megadott aktuális paraméterek illeszkednek. Az OO nyelvekben az ilyen függvényeket *overload* függvénynek nevezzük. A mintaillesztés alkalmazható változók, és listák mintaillesztése esetén is, valamint elágazások ágainak (*branch*) a kiválasztására.

2.10. Magasabb rendű függvények

Funkcionális nyelvekben lambda-kifejezéseket, vagyis speciálisan leírt függvényeket adhatunk át más függvények paraméter listájában. Természetesen nem a függvényhívás kerül a paraméterek közé, hanem a függvény prototípusa.

Számos programozási nyelv lehetőséget ad arra, hogy változókba kössük a lambda-függvényeket, majd az így megkonstruált változókat később függvényként használjuk. A lambda-kifejezéseket elhelyezhetjük lista-kifejezésekben is. A függvénnyel való paraméterezés segítségünkre van abban, hogy teljesen általános függvényeket, vagy programokat hozzunk létre, ami kifejezetten hasznos szerver alkalmazások készítésekor.

2.11. Curry módszer

A Curry módszer a részleges függvény alkalmazás, ami azt jelenti, hogy a több paraméteres függvények visszatérési értéke lehet egy függvény és a maradék paraméterek. Ez alapján minden függvény tekinthető egyváltozós függvénynek. Amennyiben a függvénynek két változója van, akkor csak az egyiket tekintjük a függvényhez tartozó paraméternek. A paraméter megadása egy új egyváltozós függvényt hoz létre, amit alkalmazhatunk a második változóra. A módszer több változó esetén is működőképes. Erlangban és F#-ban nincs ilyen nyelvi elem, de a lehetősége adott. Készítéséhez függvény kifejezést használhatunk.

2.12. Statikus típusrendszer

A statikus típusrendszerekben nem kötelező a deklarációk megadása, de követelmény, hogy a kifejezés legáltalánosabb típusát a fordítóprogram minden esetben ki tudja következtetni. A statikus rendszer alapja a Hindley-Milner polimorfikus típusrendszer.

Természetesen a típusok megadásának az elhagyhatósága nem azt jelenti, hogy azok nincsenek jelen az adott nyelvben. Igenis vannak típusok, csak a kezelésükről, valamint a kifejezések helyes kiértékeléséről a típuslevezető rendszer és a fordítóprogram gondoskodik.

2.13. Halmazkifejezések

A különböző nyelvi konstrukciók mellett a funkcionális nyelvekben számos különleges adattípust is találunk, mint a rendezett `n`-es, vagy ismertebb nevén a `tuple`, valamint a halmazkifejezések, és az ezekhez konstruálható lista generátorok. A lista adattípus a Zermelo-Fraenkel féle halmazkifejezésen alapul. Tartalmaz egy generátort, amely az elemek listába tartozásának feltételét írja le, valamint azt, hogy a lista egyes elemeit hogyan, és milyen számban kell előállítani. Ezzel a technológiával elvben végtelen listát is képesek vagyunk leírni az adott programozási nyelven, mivel nem a lista elemeit, hanem a lista definícióját írjuk le.

3. Alapvető Input-Output

3.1. Programozási környezetek használata

A fejezetekben szereplő példák megértéséhez, valamint a gyakorlás érdekében meg kell tanulnunk azt, hogy miként készítsük el az Erlang, a Clean és az F# nyelvű programokat, valamint azt is, hogyan kell a futtatás során a programok ki és bemenetét kezelni. A programok elkészítéséhez felhasznált eszközök és futtató rendszerek mindegyike ingyenesen hozzáférhető, de a fejlesztő eszközök tekintetében a kedves olvasó eltérhet az itt szereplő szoftverektől és operációs rendszertől, és választhatja a szívének kedveset.

3.2. Erlang

Erlang programokról lévén szó, a rugalmasság fontos szempont mind a fejlesztés, mind a programok felhasználása tekintetében. Az Erlang nem rendelkezik kizárólag a nyelvhez készített fejlesztő eszközzel.

Programjainkat elkészíthetjük szövegszerkesztővel, vagy más nyelvekhez készített, esetleg általános felhasználású grafikus fejlesztő eszközzel. A Linux rendszereken az egyik legelterjedtebb szerkesztő program az Emacs. Ezt a programot nem kezelhetjük szövegszerkesztőként, mivel alkalmas szinte minden ismert nyelv, még a Tex alapú nyelvek kezelésére is.

Egy másik érdekes rendszer az Eclipse, ami inkább Java nyelvű programok fejlesztésére készült, de rendelkezik Erlang nyelvű programok írására és futtatására használható beépülő modullal.

Amennyiben nem kívánunk fejlesztő eszközt használni, programozhatunk közvetlenül a parancssorban, de ebben az esetben nehezebb dolgunk lesz az összetett, több modulból felépülő szoftverek kivitelezésével.

3.3. Clean

A Clean nyelvű példaprogramok kevesebb fejtörést okoznak, mivel a nyelv rendelkezik saját - és igen sajátos - integrált fejlesztői eszközzel. Az eszköz a szöveg szerkesztési, valamint a program futtatási szolgáltatások mellett rendelkezik beépített hibakeresővel (debugger), ami megkönnyíti a programozó munkáját.

Természetesen az Erlang is rendelkezik hibakeresővel, ami szintén jól használható, de a hibaüzenetei első látásra igen ijesztőek...

3.4. FSharp

Az F# programokat hagyományosan a MS Visual Studio legújabb verzióival, vagy más programírásra alkalmas editorral is elkészíthetjük. A nyelv futtató rendszere és a *debuggere* kényelmes, valamint könnyen tanulható. Az F# alkalmas nagyobb projektek készítésére, és az elkészített projektek integrálhatóak a C# és C++ nyelvű programokhoz.

3.5. Kezdeti lépések Erlang programok futtatásához

Az első program készítésének a lépéseit a kód megírásától a futtatásáig elemezni fogjuk. Ez a program egyszerű összeadást valósít meg két változóban tárolt számon. A parancssori verzió elkészítését követően megírjuk a függvényt használó változatot, amit egy modulban helyezünk el, hogy azt később is futtatni tudjuk. A parancssori megoldásnak az a gyakorlati haszna, hogy nem kell konfigurálnunk semmilyen fejlesztő eszközt, vagy futtató környezetet.

Természetesen a program futtatásához rendelkeznünk kell a nyelv fordító programjával és futtató rendszerével...

Az Erlang program elkészítéséhez gépeljük a parancssorba az `erl` szót, ami elindítja az Erlang rendszert parancssori üzemmódban. Amennyiben elindult a rendszer, gépeljük be az alábbi program sorokat!

3.1. programlista. Változók összeadása parancssori programma

```
> A = 10.
> B = 20.
> A + B.
> 30
```

A 3.1. programlista első sorában értéket adunk az `A` változónak. Ezt a műveletet valójában kötésnek kellene neveznünk, mivel - mint azt korábban már leszögeztük - a funkcionális nyelvekben a változók csak egyszer kapnak értéket, vagyis egyszer köthető hozzájuk adat. Az imperatív nyelvekben megszokott `destruktív` értékadás (`A = A + 1`) a funkcionális nyelvekben nincs jelen. Ez a gyakorlatban annyit jelent, hogy az értékadást megismételve hibaüzenetet kapunk eredményül, ami tájékoztat minket a változó korábbi kötéséről.

A második sorban a `B` változó értékét kötjük, majd az `A + B` kifejezéssel összeadjuk a két változó tartalmát. Az utolsó sorban a kifejezés eredményét láthatjuk a parancssorba írva. Amennyiben meg szeretnénk ismételni ezt a néhány utasítást, a hibák elkerülése érdekében az `f` nevű könyvtári függvényt kell meghívnunk, ami felszabadítja a kötött változókat.

Az `f()` (3.2. példa) paraméterek nélküli meghívása az összes változót felszabadítja, és minden, a változókban tárolt adat elveszik...

3.2. programlista. Változók felszabadítása

```
> f(A).
```

```
> f(B).
> f().
```

Ahogy a példaprogramokban láthatjuk, az utasítások végét . (pont) zárja. Ez a modulok esetén úgy módosul, hogy a függvények végén pont, az utasításaik végén pedig vessző áll. A ; arra használható, hogy a függvények, vagy az elágazások ágait elválasszák egymástól.

Amennyiben hiba nélkül lefuttattuk a programot, készítsük el a tárolt változatát is, hogy ne kelljen minden futtatás előtt újra és újra begépelnünk a sorait. Az összetartozó függvényeket modulokba, a modulokat fájlokba szokás menteni, mert így bármikor felhasználhatjuk a tartalmukat. Minden modul rendelkezik azonosítóval, vagyis van neve, és egy `export` listával, ami a modul függvényeit láthatóvá teszi a modulon kívüli világ számára.

3.3. programlista. Összadó függvény modulja

```
-module(mod).
-export([sum/2]).

sum(A, B) ->
         A + B.
```

A függvények modulon belüli sorrendje teljesen lényegtelen. A sorrend semmilyen hatást nem gyakorolhat a fordításra és a futtatásra. A modul nevét a `-module()` zárójelei között kell elhelyezni, és a névnek meg kell egyezni a modult tartalmazó fájl nevével. Az `export`-lista, ahogy a neve is mutatja egy lista, melybe a publikus függvényeket kell feltüntetni a nevükből és az `aritás`-ukból képzett párossal (`f/1, g/2, h/0`).

Az aritás a függvény paramétereinek a száma, tehát egy két paraméteres függvény aritása kettő, a paraméterekkel nem rendelkező függvényé pedig nulla.

A függvényekre a dokumentációkban, valamint a szöveges leírásokban is a név és aritás párossal hivatkozunk, mert így egyértelműen azonosítani lehet azokat. Több modul használata esetén a függvény moduljának a neve is része az azonosítónak. A modul nevéből, és a függvény azonosítójából álló nevet minősített névnek nevezzük (`modul:fuggveny/aritas`, vagy `mod:sum/2`)...

A függvények neve mindig kisbetűvel kezdődik, a változóké nagybetűvel. A függvényeket a nevük vezeti be, majd a paraméterlista következik, melyet különböző őrfeltételek követhetnek. A feltételek után áll a függvény törzse, melyet egy ág esetén egy pont zár.

A függvények készítését és használatát később részletesen megvizsgáljuk...

Ha elkészítettük a modult, mentsük el arra a névre, amit a modulban használtunk azonosítónak, ezután következhet a fordítás, majd a futtatás.

3.4. programlista. Parancssori fordítás

```
> c(mod)
> {ok, sum}
> mod:sum(10,20).
> 30
```

A modul lefordítását a parancssorban végezhetjük a legegyszerűbben a `c(sum)` formulával, ahol a `c` a `compile` szóra utal, a zárójelek között pedig a modul neve található (3.4. programszöveg). A fordítás sikerességéről, vagy az elkövetett hibákról, a fordítóprogram azonnal tájékoztat minket. Siker esetén rögtön meghívhatjuk a modulban szereplő, és az export listában feltüntetett függvényeket.

A futtatáshoz a modul nevét kell használnunk a minősítéshez. A modul nevét : után a függvény neve követi, majd a formális paraméter lista alapján az aktuális paraméterek következnek. Amennyiben mindent jól csináltunk, a képernyőn megjelenik a `sum/2` függvény eredménye, vagyis a megadott két szám összege.

3.6. Clean kezdetek

Clean programok készítése az integrált fejlesztőeszköz segítségével lényegesen egyszerűbb, mint az IDE-vel nem rendelkező funkcionális nyelveké. Az eszköz elindítása után a kód szerkesztőben megírhatjuk a programok forráskódját, majd a hibajavítást követően le tudjuk futtatni azokat. A Clean kód megírásához a következő lépéseket kell végrehajtanunk:

- Indítsuk el a Clean IDE-t.
- A File menüben készítsünk egy új .icl kiterjesztésű fájlt (*New File...*) tetszőleges névvel. Később ez lesz az implementációs modulunk.
- Ezután hozzunk létre ugyancsak a File menüben egy új projektet (*New Project...*), melynek a neve legyen a modulunk neve. Ezáltal létrejön az a projekt, amely tartalmazni fog egy hivatkozást az előzőleg létrehozott .icl fájlra. Ekkor két ablakot kell, hogy lássunk: az egyik a project manager ablak, a másik pedig az .icl fájl, amelybe a forráskód kerül. Ha az előző lépéseket fordítva próbáljuk csinálni, akkor az *"Unable to create new project. There's no active module window."* hibaüzenettel találjuk szembe magunkat.
- A következő lépésben az .icl fájl tartalmát mutató ablakba írjuk be az *import modul neve* sort. A *modul neve* helyére kötelezően az .icl fájl nevét kell megadnunk kiterjesztés nélkül! Ezzel tudatjuk a fordítóprogrammal, hogy mi a modulunk neve, és mi a neve a fájlnak amiben a tartalmát leírjuk.
- Ha mindent helyesen csináltunk, akkor elkezdhetünk forráskódot írni. Ahhoz azonban hogy ezen jegyzetben szereplő példakódokat használni tudjuk (beleértve az alapvető operátorokat, listákat, típusokat, stb.) szükségünk lesz még a *Standard Enviroment library*-re. Ezt úgy tudjuk elérni, hogy a modulunk neve után beírjuk: *import StdEnv*.

3.6. programlista. Clean példaprogram

```
module sajatmodul
import StdEnv

//ide kerül a saját kód
...
..
.
```

3.7. F# programok írása és futtatása

Az egyik, és egyben egyszerűbb lehetőség, a Visual Studio 2010 használata. Ez az első Visual Studio ami már tartalmazza az F# nyelvet.

- Indítsuk el az eszközt, majd a *File* menüben hozzunk létre egy új *F# project*-et.
- Nyissuk meg a *File/New/Project* párbeszédablakot, majd válasszuk ki az *Other Languages/Visual F#* kategóriából az *F# Application* pontot.
- Jelenítsük meg az *Interactive* ablakot, a *View/Other Windows/F# Interactive* menüpont segítségével (használhatjuk a `Ctrl+Alt+F` billentyű kombinációt is).
- A parancsokat gépelhetjük közvetlenül az interaktív ablakba, vagy az automatikusan készített `Program.fs` állományba a szerkesztőn keresztül. Utóbbi esetben a kódban a futtatni kívánt sorra - vagy kijelölésre - jobb egérgombbal kattintva, és a *Send Line to Interactive* - esetleg *Send to Interactive* - menüpontot választva futtathatjuk le a kívánt kódrészletet.
- A program egészét a *Debug/Start Debugging* menüpont kiválasztásával futtathatjuk (gyorsbillentyű: F5). Próbaképpen gépeljük be a 3.1 listában található sorokat az interaktív ablakba.

3.7. programlista. F# példaprogram

```
let A = 10;;
let B = 20;;
A + B;;
```

Az interaktív ablakban a 3.8. listában látható sor jelenik meg.

3.8. programlista. Interaktív ablak

```
val it : int = 30
```

Az F# programok írására a fentieknél egy sokkal egyszerűbb megoldás is kínálkozik. Lehetőségünk van a nyelvhez kiadott parancssoros interpreter használatára, amely ingyenesen letölthető a www.fsharp.net weboldalról.

3.8. programlista. F# module

```
module sajatModul

let a = 10
let b = 20
let sum a b = a + b
```

A parancssori fordító használata mellett a programot egy fájlban kell elhelyeznünk, és azt a 3.8. listában található minta alapján kell elkészítenünk (több modul használata esetén a *module* kulcsszó kötelező).

3.8. Mellékhatások kezelése

Vizsgáljuk meg újra az Erlang nyelven készült `sum/2` nevű függvényt (3.9. programlista). Láthatjuk, hogy a függvény azon kívül, hogy összeadja a paraméterlistájában kapott két értéket, semmi mást nem csinál.

A meghívás helyére visszatér az összeggel, de nem ír semmit a képernyőre, nem küld és nem kap üzenetet, vagyis csak a paramétereiben fogad el adatot.

Az ilyen függvényekre azt mondjuk, hogy nincs mellékhatásuk. Számos nyelvben a mellékhatással rendelkező függvényeket a `dirty` jelzővel illetik, ami arra utal, hogy a függvény nem teljesen szabályos működésű. Az imperatív nyelvekben a mellékhatásos függvényeket eljárásnak (void), a mellékhatás nélküli változatokat függvénynek nevezzük. Ezekben a nyelvekben a mellékhatással nem rendelkező függvényeknek van típusa, az eljárásoknak nincs, vagy void típusúak.

Az Erlang és az F# nyelvek nem tisztán funkcionálisak, mint a Haskell, aminek az IO műveletei úgynevezett `Monad`-okra épülnek.

Azért, hogy láthassuk a példaprogramok kimenetét, néha el kell térnünk attól az elvtől, hogy nem készítünk `dirty` függvényeket. A mellékhatásokat azért sem mellőzhetjük, mert a kiíró utasítások eleve mellékhatást eredményeznek. Az ilyen függvények nem csak az eredményüket adják vissza, hanem elvégzik a kiírás műveletét is. Az Erlang nyelv egyik kiíró utasítása az `io` könyvtári modulban található `format` függvény. Segítségével a standard inputon - vagy a programozó által meghatározott inputon - formázottan jeleníthetjük meg az adatokat. Második próbálkozásnak készítsük el az összeadást végző program azon változatát, amely üzeneteket ír a standard inputra, valamint vissza is tér az eredménnyel. Ez a függvény tipikus példája a mellékhatásoknak, melyek a funkcionális programok függvényeit nehezen használhatóvá tehetik. Ez az állítás egyébként minden más paradigma esetén is igaz.

3.9. programlista. Mellékhatásos függvény - Erlang

```
-module(mod).
-export([sum/2]).
```

```
sum(A, B) ->
     io:format("szerintem ~w~n",
                    [random:uniform(100)]),
     A + B.
```

A 3.9 példában a `sum/2` függvény elsőként kiírja a képernyőre, hogy szerinte mi lesz az összeadás értéke. A `random` modul `uniform/1` függvénye segítségével előállít egy véletlen számot, ezt használva tippként, nyilván elég csekély találati eséllyel. A kiírás után visszatér a valódi eredménnyel. A függvénynek ebben a formában nem sok értelme van, de jól példázza a mellékhatások, valamint a `format` működését.

Az IO műveleteket a programozók gyakran hibakeresésre is használják. Olyan programozási környezetben, ahol a hibakeresés nehézkes, vagy nincs rá más lehetőség, az adott nyelv kiíró utasításai alkalmasak a változók, listák és egyéb adatok megjelenítésére, valamint a hibák megtalálására.

4. Az adatok kezelése

4.1. Változók

A funkcionális nyelvekben a változók kezelése nagyon eltér az imperatív és OO környezetben megszokottaktól. A változók csak egyszer kaphatnak értéket, vagyis egyszer lehet kötni hozzájuk értéket. A destruktív értékadás hiánya számos olyan konstrukció használatát kizárja, amelyek az egymás utáni, többszörös értékadáson alapulnak (`I = I + 1`).

Természetesen minden, az OO és egyéb imperatív nyelveknél megszokott formáknak van funkcionális megfelelője. A destruktív értékadást kiválthatjuk egy újabb változó bevezetésével `A = A0 + 1`. Az ismétléseket (ciklusok, mint a *for*, a *while* és a *do-while*) rekurzióval és több ággal rendelkező függvények írásával oldjuk meg, de erről a függvényekről szóló részben ejtünk szót.

Funkcionális nyelvek használata mellett az adatainkat tárolhatjuk változókban, a változókat listákban, vagy rendezett n-esekben (`tuple`), és természetesen fájlokban, valamint adatbázis kezelők tábláiban. Számos funkcionális nyelvi környezetben találunk az adott nyelvhez optimalizált adatbázis kezelő modulokat, vagy önálló rendszereket. Az Erlang ismert adatbázis kezelője a `MNESIA`, és a hozzá tartozó lekérdező nyelv a `QLC`...

A fájl és adatbázis alapú tárolással nem foglalkozunk részletesen, de az említett három adatszerkezetet mindenképpen meg kell vizsgálnunk ahhoz, hogy képesek legyünk a hatékony adatkezelésre. A változók, a tuple és a listák jellemzik leginkább a funkcionális nyelvek adattárolását, de számos nyelvben van még egy különleges elem, az `atom`, amelyet legtöbbször azonosítóként használunk. A függvények neve is atom, valamint a paraméter listákban is gyakran találhatunk atomokat, amelyek a függvények egyes ágait azonosítják (4.1 programlista).

A változókban tárolhatunk egész és valós számokat, valamint karaktereket és karakter sorozatokat. A karakterláncok a legtöbb funkcionális nyelvben a listákhoz készített " szintaktikus cukorkák".

A " szintaktikus cukorka" kifejezés arra utal, hogy a `string` nem valódi adattípus, hanem lista, amely a karakterek kódjait tárolja, de készült hozzá egy mechanizmus és néhány függvény, valamint operátor, ami a felhasználást, kiírást és a formázást egyszerűbbé teszi...

4.1. programlista. Erlang változók

```
 A = 10, %integer
 B = 3.14, %real
 C = alma, %változóban tárolt atom
 L = [1, 2, A, B, a, b], %vegyes tartalmú lista
 N = {A, B, C = 2}, %három elemű tuple
LN = [N, A, B, {a, b}] %lista négy elemmel
```

4.2. programlista. F# változók

```
let A = 10 //int
```

```
 let B = 3.14 //float
let L1 = [1, 2, A, B, 'a', 'b']
                   //vegyes tartalmú lista
let L2 = [1; 2; 3; 4] //int lista
 let N = (A, B, C) //három elemű tuple
let LN = [N, A, B, ('a', 'b')] //lista négy elemme
```

5. Kifejezések

5.1. Műveleti aritmetika

A funkcionális nyelvek is rendelkeznek azokkal az egyszerű, numerikus, és logikai operátorokkal, mint az imperatív, és OO társaik. A kifejezések kiértékelésében is csak annyiban térnek el, hogy a funkcionális nyelvekben a korábban már említett lusta, valamint a mohó kiértékelés is (vagy mindkettő egyszerre) szerepelhet. A különbségek ellenére az egyszerű műveletek írásában, és a kifejezések szerkezetében nagy a hasonlóság. A 5.1. lista bemutatja a teljesség igénye nélkül az operátorok használatát mind a numerikus, mind logikai érékekre alkalmazva.

5.1. programlista. Operátorok használata - Erlang

```
> hello.
hello
> 2 + 3 * 5.
17
> (2 + 3) * 5.
25
> X = 22.
22
> Y = 20.
20
> X + Y.
42
> Z = X + Y.
42
> Z == 42.
true
> 4 / 3.
1.3333333333333333
> {5 rem 3,  4 div 2}.
{2, 2}
```

Az 5.1., 5.2., és 5.3. programlisták azokat az operátorokat tartalmazzák, melyeket a különböző kifejezések írásakor használhatunk.

Ha jobban megvizsgáljuk a listát, rájöhetünk, hogy nem csak az operátorokat, hanem azok precedenciáját is tartalmazza. Az első sorban találjuk a legerősebb műveleteket, és lefelé haladva a listában a " gyengébbeket”...

5.2. programlista. Operátor precedencia - Erlang

```
(unáris) +, (unaris) -, bnot, not
/, *, div, rem, band, and (bal asszociatívak)
+, -, bor, bxor, bsl, bsr, or, xor
                                (bal asszociatívak)
++, -- (jobb asszociatívak)
==, /=, =003C
    003E, 003C=, 003C, =:=, =/=
andalso
orelse
```

5.3. programlista. Operátor precedencia – Clean

```
!
lefoglalva függvény alkalmazás számára
o !! %
^
* / mod rem
+ - bitor bitand bitxor
++ +++
== 003C
    003C
    003C= 003E
    003E=
0026
    0026
||
:=
`bind`
```

5.3. programlista. Operátor precedencia – F#

```
-, +, %, %%, 0026, 0026
    0026, !, ~
        (prefix operátorok, bal asszociatívak)
*, /, % (bal asszociatívak)
-, + (bináris, bal asszociatívak)
003C, 003E, =, |, 0026 (bal asszociatívak)
0026, 0026
    0026 (bal asszociatívak)
or, || (bal asszociatívak)
```

A numerikus, és a logikai típusok mellett a `string` típusra is találunk néhány operátort. Ezek a `++` és a `--`. Ezek a műveletek valójában lista kezelő operátorok. Tudjuk, hogy a két típus lényegében azonos...

5.2. Mintaillesztés

A minták használata, és a mintaillesztés a funkcionális nyelvekben ismert, és gyakran alkalmazott művelet. Imperatív programozási nyelvekben, valamint az OO programokban is találunk hasonló fogalmakat, úgy mint az operátor és függvény túlterhelés. Mielőtt rátérnénk a funkcionális nyelvek mintaillesztésére, ismétlésként vizsgáljuk meg az OO osztályokban gyakorta alkalmazott `overload` metódus működését. A technika lényege az, hogy egy osztályon belül (`class`) ugyanazzal a névvel, de más paraméterezéssel állíthatunk elő metódusokat.

5.4. programlista. Overloading OO nyelvekben

```
class cppclass
{
 v  double sum(int x, int y)
     {
       return x + y;
    }
     string sum(string x, string y)
   {
     return toint(x) + toint(y);
   }
}
```

Az 5.4. forrásszövegben definiált osztály példányosítása után a metódusait a példány minősített nevével érhetjük el (`PName -> sum(10, 20)`). Ezáltal a metódus két különböző verzióját is meghívhatjuk. Amennyiben a `sum/2` metódust két egész számmal hívjuk meg, az első, `int`-ekkel dolgozó változat fut le, `string` típusú paraméterek esetén a második (5.5. programlista).

5.5. programlista. Túlterhelt metódus alkalmazása

```
cppclass PName = new PName();
int result1 = PName -> sum(10,20);
int result2 = PName -> sum("10","20");
```

Az `overload` metódus(ok) meghívásakor mindig az az ág fut, amelyik formális paramétereire "illeszkedik" a megadott aktuális paraméterlista.

Az OO programokban a mintaillesztést nem ugyanazon elvek mentén alkalmazzuk, mint a funkcionális nyelvekben. Ez a különbség abból ered, hogy a funkcionális paradigma használata merőben más gondolkodásmódot igényel a különböző problémák megoldása során...

A mintaillesztés, az `overload` technikához hasonló, de eltérő módon működik mint a funkcionális nyelvek mintákat használó függvényei. Ezekben a nyelvekben bármely függvény rendelkezhet több ággal (`clause`), és mindig a paraméterektől függően fut le valamely ága.

5.6. programlista. Mintaillesztés case-ben - Erlang

```
patterns(A)->
  [P1, P2| Pn] = [1, 2, 3, 4],
  {T1, T2, T3} = {data1, data2, data3},
  case A of
       {add, A, B} -> A + B;
       {mul, A, B} -> A * B;
       {inc, A} -> A + 1;
       {dec, B} -> A - 1;
        _ -> {error, A}
  end.
```

5.7. programlista. Mintaillesztés case-ben – F#

```
let patterns a =
    match a with
    | ("add", A, B) -> A + B
    | ("mul", A, B) -> A * B
    | ("inc", A) -> A + 1 //HIBA!
          //Sajnos ezt a részt nem lehet F#-ban
          //ez a nyelv nem ennyire megengedő...
    | _ -> ("error", A) //Itt szintén baj van...
    …
```

A mintaillesztést használja ki a `case` elágazás (5.6. programlista), az `if`, valamint az üzenet küldések során is minták alapján dönthető el, hogy melyik üzenet hatására mit kell reagálnia a fogadó rutinnak. Az üzenetküldésre később kitérünk.

5.8. programlista. Üzenetek fogadása – Erlang

```
receive
     {time, Pid} -> Pid ! morning;
     {hello, Pid} -> Pid ! hello;
     _ -> io:format("~s~n", ["Wrong message format"])
end
```

A mintaillesztést tehát nem csak függvényekben, elágazásokban, vagy üzenetek fogadására, hanem változók, tuple, lista, és rekord adatszerkezetek kezelésére is felhasználhatjuk. A 5.9. programlista a minták további alkalmazását bemutató példákat, és a példákhoz tartozó magyarázatokat tartalmazza.

5.9. programlista. Mintaillesztések - Erlang

```
{A, abc} = {123, abc}
              %A = 123,
             %abc illeszkedik az abc atomra
{A, B, C} = {123, abc, bca}
              %A = 123, B = abc, C = bca
              %a mintaillesztés sikeres
{A, B} = {123, abc, bca}
              %Hibás illeszkedés,
              %mert nincs elegendő elem
              %a bal oldalon
 A = true
              %A-ba kötjük a true értéket
{A, B, C} = {{abc, 123}, 42, {abc,123}}
              %Megfelelő illeszkedés, a
              %változók sorba megkapják az
              %{abc, 123}, a 42, és az {abc, 123}
              %elemeket
[H|T] = [1,2,3,4,5]
              %A H változó felveszi az 1 értéket,
              %a T a lista végét: [2,3,4,5]
[A,B,C|T] = [a,b,c,d,e,f]
              %Az A változóba bekerül az a atom,
              %B-be a b atom, C-be a c atom, és T
              %megkapja a lista végét: [d,e,f]
```

5.3. Őr feltételek használata

A függvények ágait, az `if`, a `case` kifejezések ágait, a `try` blokkok, valamint a különböző üzenetküldő kifejezések ágait kiegészíthetjük őr feltételekkel, így szabályozva az ágak lefutását. A `guard` feltételek használata nem szükséges, de jó lehetőség arra, hogy a programok szövegét olvashatóbbá tegyük.

5.10. programlista. Őr feltétel függvény ágaihoz (clause) - Erlang

```
max(X, Y) when X > Y -> X;
max(X, Y) -> Y.
```

5.11. programlista. Őr feltétel függvény ágaihoz (clause) – Clean

```
maximum x y
| x > y = x
        = y
```

5.12. programlista. Őr feltétel függvény ágaihoz (clause) - F#

```
let max x y =
      match x, y with
      | a, b when a 003E= b -003E a
      | a, b when a 003C b -003E b
```

Az 5.10 példában a `max/2` függvény az első ág őr feltételét használja fel arra, hogy kiválassza a paraméterei közül a nagyobbat. Ezt a problémát természetesen megoldhattuk volna egy `if`, vagy egy `case` kifejezés használatával is, de ez a megoldás sokkal érdekesebb.

5.13. programlista. Összetett őr feltételek - Erlang

```
f(X) when (X == 0) or (1/X > 2) ->
...
g(X) when (X == 0) orelse (1/X > 2) ->
```

```
...
```

5.14. programlista. Címe

```
f x
| x == 0 || 1/x > 2 =
...
```

Az őr feltételekben tetszés szerint használhatjuk az operátorokat, de arra mindenképpen ügyelnünk kell, hogy az őr eredménye logikai érték legyen (`true/false`). A feltétel lehet összetett, ekkor a részeit logikai operátorokkal tudjuk elválasztani egymástól (5.13. programlista).

5.4. If kifejezés

Mint azt tudjuk, minden algoritmikus probléma megoldható szekvencia, szelekció, és iteráció segítségével. Nincs ez másként akkor sem, mikor funkcionális nyelvekkel dolgozunk. A szekvencia kézenfekvő, mivel a függvényekben szereplő utasítások sorban, egymás után hajtódnak végre. Az iterációs lépések rekurzióval valósulnak meg, az elágazások pedig több ággal rendelkező függvények, vagy az `if`, és a `case` vezérlő szerkezetek formájában realizálódnak a programokban. Az `if` őr feltételeket használ arra, hogy a megfelelő ága kiválasztásra kerüljön. Működése ha nem sokban, de kissé mégis eltér a megszokottól, mert nem csak egy igaz, valamint egy hamis ággal rendelkezik, hanem annyival, ahány őr feltételt szervezünk. Minden egyes ágban egy kifejezéseket tartalmazó szekvencia helyezhető el, amely az adott őr teljesülésekor végrehajtódik 5.15., 5.16., és 17. programlista).

5.15. programlista. Az if kifejezés általános formája - Erlang

```
if
  Guard1 ->
     Expr_seq1;
  Guard2 ->
     Expr_seq2;
 ...
end
```

5.16. programlista. Az if kifejezés általános formája Clean nyelven

```
if Guard1 Expr_seq1 if Guard2 Expr_seq2
 …
```

5.17. programlista. F# példa if kifejezésre (legnagyobb közös osztó)

```
let rec hcf a b =
      if a=0 then b
      elif a 003C b then hcf a (b-a)
      else hcf (a-b) b
```

A funkcionális programok nagy részében inkább a `case` kifejezéseket használják a programozók, és sok nyelv is inkább a több ággal rendelkező elágazásokat támogatja.

5.5. Case kifejezés

Az elágazások egy másik változata szintén több ággal rendelkezik. A `C` alapú programozási nyelveknél megszokhattuk, hogy a több ágú szelekciós utasítások kulcsszava általában a `switch` (5.18. programlista), és az egyes ágakat vezeti be a `case`.

5.18. programlista. A case kifejezés a C alapú nyelvekben

```
switch (a)
{
    case 1:  ... break;
    case 2:  ... break;
    ...
    default: ... break;
}
```

Az Erlang, valamit a Clean nyelvekben az elágazás a `case` kulcsszót követően tartalmaz egy kifejezést, majd az `of` szó után egy felsorolást, ahol a kifejezés lehetséges kimenetei alapján biztosan eldönthető, hogy melyik ág kerüljön kiválasztásra.

Az egyes ágak tartalmaznak egy-egy mintát, amelyre a kifejezés (az `Expr` a 5.19., 5.20., 5.21. listákban) egy várható eredményének illeszkednie kell (legalábbis illik valamelyik ágban szereplő mintára illeszkednie), ezután egy `->` szimbólumot találunk, majd azok az utasítások következnek, amelyeket abban az ágban végre kell hajtani. Az elágazásban szervezhetünk "egy" alapértelmezett ágat is, melynek a sorát a `_ ->` formula vezeti be. Az _ jelre bármely érték illeszkedik. Ez azt jelenti, hogy a `case` utáni kifejezés értéke - bármi legyen az, akár hiba is - minden esetben illeszkedni fog erre az ágra.

Amennyiben nem akarjuk, hogy az elágazást tartalmazó függvény parciális működést mutasson, és gondosan szeretnénk megírni a programjainkat, ügyelve a kifejezés kiértékelése során felmerülő hibákra, akkor mindenképp készítsünk alapértelmezett ágat az elágazások végére...

5.19. programlista. A case kifejezés általános formája Erlang nyelven

```
case Expr of
    Pattern1 -> Expr_seq1;
    Pattern2 -> Expr_seq2;
     _ -> Default_seq
end.
```

5.20. programlista. A case kifejezés általános formája Clean nyelven

```
patterns Expr
| Expr==Pattern1 = Expr_seq1
| Expr==Pattern2 = Expr_seq2
| otherwise = Expr_seq3
```

5.21. programlista. A case kifejezés általános formája F# nyelven

```
match Expr with
| Pattern1 -> Expr_seq1
| Pattern2 -> Expr_seq2
```

Az ágakat a `;` zárja, kivéve az utolsót, vagyis az `end` szócska előttit. Ide nem teszünk semilyen jelet, jelezve ezzel a fordító számára, hogy ez lesz az utolsó ág. Ennek a szabálynak a betartása az egymásba ágyazott `case` kifejezések használata mellett válik igazán fontossá, mivel az egymásba ágyazás során a `case` ágai helyett sokszor az `end` szó után kell (vagy nem kell) alkalmazni a `;` jelet (5.22. példaprogram).

5.22. programlista. Egymásba ágyazott case kifejezések Erlang nyelven

```
...
case Expr1 of
  Pattern1_1 -> Expr_seq1_1;
  Pattern1_2 -> case Expr2 of
```

```
                      Pattern2_1 -> Expr_seq2_1;
                      Pattern2_2 -> Expr_seq2_2s
                   end
end,
…
```

5.23. programlista. Egymásba ágyazott case kifejezések Clean nyelven

```
module modul47
import StdEnv

patterns a
| a==10 = a
| a>10 = patterns2 a

patterns2 b
| b==11 =1
| b>11 =2

Start = patterns 11
```

5.24. programlista. Egymásba ágyazott case kifejezések F# nyelven

```
match Expr with
| Pattern1_1 -> Expr_seq1_1
| Pattern1_2 -> match Expr2 with
                    | Pattern2 1 -> Expr seq2 1
                    | Pattern2_2 -> Expr_seq2_2
```

Az 5.24. listában látható példa használható, de nem egyezik meg az *Erlang* és a *Clean* változatokkal. Az ebben a példában szereplő beágyazott `case` kifejezések esetén az *F#* fordító a behúzások alapján kezeli, hogy melyik minta, melyik *match*-hez tartozik...

A 5.25. példában a `case` kifejezést egy két ággal (`clause`) rendelkező függvénnyel kombináltuk. A `sum/2` a bemenetére érkező lista elemeit aszerint válogatja szét, hogy azok milyen formátumban helyezkednek el a listában. Ha az adott elem `tuple`, akkor az abban szereplő első elemet adja hozzá az összeghez (`Sum`). Abban az esetben, ha az elem nem egy `tuple`, hanem egy "szimpla" változó, akkor egyszerűen csak hozzáadja az eddigi futások során összegzett eredményhez. Minden, az előzőektől eltérő esetben, vagyis, ha a lista soron következő eleme nem illeszkedik a `case` első két ágára (`branche`), akor azt nem vesszük figyelembe, vagyis nem adjuk hozzá az összeghez.

Az angol terminológia a függvény ágakra a *clause*, amíg a `case` ágaira a *branche* kifejezést használja. Ebben a bekezdésben ezt jelöltük, de a továbbiakban csak ott teszünk erre vonatkozóan megjegyzést, ahol feltétlenül szükséges...

A függvény két ágát arra használjuk, hogy az összegzés a lista utolsó elemének feldolgozása után mindenképpen megálljon, majd visszaadja az eredményt.

5.25. programlista. Elágazások használata - Erlang

```
sum(Sum, [Head|Tail]) ->
     case Head of
            {A, _} when is_integer(A) ->
                                  sum(Sum + A, Tail);
            B when is_integer(B) ->
                                 sum(Sum + B, Tail);
            _ -> sum(Sum, Tail)
     end;
sum(Sum, []) ->
     Sum.
```

5.6. Kivételkezelés

Bármely más paradigmához hasonlóan, a funkcionális nyelvekben is nagyon fontos szerepet kap a hibák, és a hibás működésből adódó kivételek kezelése. Mielőtt azonban a hibakezelés elsajátításába fognánk, érdemes elgondolkodni azon, hogy mit nevezünk hibának. A hiba a program egy olyan nem kívánatos működése, amire a program írásakor nem számítunk, de annak futása alatt jelentkezhet (és legtöbbször szokott is, természetesen nem a tesztelés, hanem a program bemutatása során)...

A szintaktikai és szemantikai hibákat a fordítóprogram már fordítási időben kiszűri, így ezek nem is okozhatnak kivételt. A kivételeket legtöbbször az IO műveletek, a file és memória kezelése, valamint a felhasználói adatcserék idézik elő...

Amennyiben felkészülünk a hibára, és előfordulásakor kezeljük, az már nem is hiba, hanem egy kivétel, amivel a programunk meg tud birkózni.

5.26. programlista. Try-catch blokk - Erlang

```
try függvény/kifejezés of

    Minta [when Guard1] -> blokk;
    Minta [when Guard2] -> blokk;
  ...
catch
    Exceptiontípus:Minta [when ExGuard1] -> utasítások;
    Exceptiontípus:Minta [when ExGuard2] -> utasítások;
...
after
     utasítások
end
```

5.27. programlista. Kivételkezelés, try-finally blokk F# nyelven

```
 let függvény [paraméter,...] =
       try
            try
                 utasítások
            with
                 | :? Exceptiontípus as ex
                           [when Guard1] -> utasítások
                 | :? Exceptiontípus as ex
                          [when Guard2] -> utasítások
       finally
            utasítások
```

A hibák kezelésének egyik ismert módja a több ággal (`clause`) rendelkező függvények használata, a másik az adott nyelvben rendelkezésre álló kivétel kezelők alkalmazása (5.26., 5.27. programlisták). A több ággal rendelkező függvények a hibák egy kis részét megoldják, ahogy ezt a 5.28. programlistában láthatjuk, de lehetőségeik korlátozottak. Az `osszeg/1` függvény össze tud adni két számot `tuple`, vagy lista formában érkező adatokból. Minden más esetben 0-át ad vissza, de hibával leáll, ha a beérkező adatok típusa nem megfelelő (pl.:string, vagy atom).

Ezt a problémát kezelhetnénk őr feltételek bevezetésével, vagy újabb függvény ágak hozzáadásával, de nem a végtelenségig. Lehet, hogy előbb-utóbb megtalálnánk az összes olyan inputot, melyek hatására a programunk helytelen működést produkál, de sokkal valószínűbb az, hogy nem gondolnánk minden lehetőségre, és a legnagyobb körültekintés mellett is maradnának hibák a programban...

A függvény ágak készítése mellett sokkal biztonságosabb, és ráadásul érdekesebb megoldást kínál a kivételkezelés. A technika lényege, hogy a hibák lehetőségét magában hordozó programrészeket kivételkezelő blokkban helyezzük el, majd a kivétel létrejöttekor kezeljük azokat, a blokk második részében elhelyezett utasításokkal.

A kivételek feldobását mi is kikényszeríthetjük a `throw` kulcsszó használatával, de mindenképpen kezelni kell azokat, különben a futtatórendszer, vagy rosszabb esetben az operációs rendszer fogja kezelni őket, és ez a lehetőség nem kimondottan elegáns...

5.28. programlista. Több féle paraméter kezelése - Erlang

```
osszeg({A, B}) ->
      A + B;
osszeg([A, B]) ->
      A + B;
osszeg(_)->
         0.
```

A hibát természetesen nem elég csak elkapni, kezelni is kell tudni. Sokszor az is elegendő, ha megpróbáljuk a rendszer által adott üzeneteket elfogni, majd formázottan kiírni a képernyőre, hogy a programozó ki tudja azt javítani, vagy a felhasználó értesüljön róla, és ne próbálja ugyanazt újra és újra elkövetni (5.29. programlista).

5.29. programlista. A kivétel okának felderítése a catch blokkban - Erlang

```
kiir(Data) ->
     try
          io:format("~s~n",[Data])
    catch
        H1:H2 ->
           io:format("H1:~w~n H2:~w~n Data:~w~n",
                                    [H1,H2,Data])
   end.
```

Az 5.29. programban a `kiir/1` függvény a bemenetére érkező szöveget kiírja a képernyőre. A paraméter típusa nincs meghatározva, de csak szöveges adatot (`string`) képes kiírni a `format` függvényben elhelyezett `"~s"` miatt. A kivételkezelő alkalmazása mellett a függvény nem megfelelő paraméter esetén sem áll meg, hanem megmutatja a rendszertől érkező hibaüzeneteket, melyek a `VAR1:VAR2` mintára illeszkednek.

Amennyiben nem érdekes a rendszer üzeneteinek tartalma, használhatjuk a `_:_` mintát is az elkapásukra, így a fordítóprogram nem hívja fel a figyelmünket minden fordításkor arra a tényre, hogy a mintában szereplő változókat nem használjuk, csak értéket adtunk nekik...

Az 5.29. programban látható kivételkezelő ekvivalens az 5.30. programban található változattal. Ebben a verzióban a `try` után áll a kifejezés, vagy az utasítás, ami hibát okozhat, majd az `of` kulcsszó. Az `of` után a `case`-hez hasonlóan mintákat heyezhetünk el, melyek segítségével feldolgozhatjuk a try-ban elhelyezett kifejezés értékét.

5.30. programlista. Kivételek kezelése mintákkal - Erlang

```
try funct(Data) of
     Value -> Value;
      _ -> error1
catch  _:_-> error2
end
```

5.31. programlista. Példa kivételkezelésre F# nyelven

```
let divide x y =
    try
       Some( x / y )
   with
       | :? System.DivideByZeroException as ex ->
         printfn "Exception! %s " (ex.Message); None
```

Az `after` kulcsszót akkor használhatjuk, ha a program egy adott részét mindenképpen le szeretnénk futtatni, a kivételek keletkezésétől függetlenül. Az `after` blokkjában elhelyezett részekre a `try` blokkban lévő utasítások hibás, valamint hibátlan futása után is sor kerül.

5.32. programlista. Kivételek dobása F# nyelven

```
try
       raise InvalidProcess("Raising Exn")
with
       | InvalidProcess(str) -> printf "%s\n" str
```

A mechanizmus jól használható olyan esetekben, ahol az erőforrásokat (file, adatbázis, process, memória) hiba esetén is be kell zárni, vagy le kell állítani.

6. Összetett adatok

6.1. Rendezett n-esek

Az első szokatlan adatszerkezet a rendezett `n-es`, vagy más nevén a `tuple`. Az n-esek tetszőleges számú, nem homogén elemet tartalmazhatnak (kifejezést, függvényt, gyakorlatilag bármit). A `tuple` elemeinek a sorrendje, és elemszáma létrehozás után kötött, viszont tetszőleges számú elemet tartalmazhatnak.

A tuple elemszámára vonatkozó megkötés nagyon hasonlít a statikus tömbök elemszámánál megismert megkötésekhez. Egyszerűen arról van szó, hogy tetszőleges, de előre meghatározott számú elemet tartalmazhatnak...

Olyan esetekben érdemes `tuple` adatszerkezetet használnunk, mikor egy függvénynek nem csak egy adattal kell visszatérnie, vagy, ha egy üzenetben egynél több adatot kell elküldenünk. Az ilyen, és ehhez hasonló esetekben az adatokat egyszerűen be tudjuk csomagolni (6.1. programlista).

- {A, B, A + B} egy rendezett n-es, ahol a két első elem tartalmazza egy összeadó kifejezés két operandusát, a harmadik elem pedig maga a kifejezés.
- {query, Function, Parameters} egy tipikus üzenet elosztott programok esetén, ahol az első elem az üzenet típusa, a második a függvény, amelyet futtatni kell, a harmadik elem pedig a függvény paramétereit tartalmazza.
- Arra hogy az átadott függvény látszólag csak egy változó, a lambda-kalkulusról és a magasabb rendű függvényekről szóló rész magyarázatot ad. Ez a technika gyakori a funkcionális programozási nyelvek esetében, és mindennapos a mobil kódok használatát támogató rendszerekben...
- {lista, [Head|Tail]} egy olyan `tuple`, mely egy lista szétbontását teszi lehetővé. Igazság szerint nem a `tuple` az, ami a listát szétszedi első elemre és egy listára, mely a lista végét tartalmazza. A `tuple` csak leírja az adatszerkezetet, mely a lista mintaillesztő kifejezést tartalmazza.

Vizsgáljuk meg a 6.1. programot, amely egy rendezett n-est használ az adatok megjelenítésére.

6.1. programlista. Tuple használata Erlang nyelven

```
-module(osszeg).
-export([osszeg/2, teszt/0]).
 osszeg(A, B) ->
     {A, B, A + B}.

 teszt() ->
     io:format("~w~n",[osszeg(10, 20)]).
```

6.2. programlista. Tuple használata Clean nyelven

```
module osszeg
import StdEnv
osszeg a b = (a, b, a+b)
teszt = osszeg 10 20
Start = teszt
```

6.3. programlista. Tuple készítése F# nyelven

```
let osszeg a b = (a, b, a + b)
```

Az `osszeg/2` függvény nem csak az eredményt adja vissza, hanem a kapott paramétereket is, melyet a `teszt/0` függvény kiír a képernyőre. Figyeljük meg, hogy az `osszeg/2` függvény nem a megszokott egy elemű visszatérési értékkel rendelkezik. Természetesen más programozási technológiák esetén is definiálhatunk olyan függvényeket, melyek listákkal, vagy más összetett adattípussal térnek vissza, de a `tuple`-höz hasonló adatszerkezet, melyben nem típusosak az elemek, nem sok van, talán csak a halmaz, de ott az elemek sorrendje és száma nem kötött, és a használata is bonyolultabb a rendezett n-esekénél. Egy másik felhasználási módja a `tuple` szerkezetnek, amikor több ággal rendelkező függvények ágaiban - valamilyen praktikus okból - szeretnénk megtartani az aritást, vagyis a paraméterek számát. Ilyen ok lehet a hibakezelés, valamint az általánosabban felhasználható függvények írásának a lehetősége.

6.4. programlista. Tuple több függvény ágban - Erlang

```
-module(osszeg).
-export([osszeg/1]).
 osszeg({A, B}) ->
     A + B;
 osszeg([A, B]) ->
     A + B;
 osszeg(_) ->
         0.
 teszt() ->
     io:format("~w~n~w~n",[osszeg({10, 20}),
                                  osszeg([10, 20])]).
```

A6.4. programlistában, az `osszeg/1` függvény ezen változatában tuple, valamint lista formátumban kapott adatokat is fel tudunk dolgozni. A függvény harmadik ága kezeli a hibás paramétereket, a `(_)` jelentése az, hogy a függvény az előző ágaktól eltérő, bármilyen egyéb adatot kaphat. Ennél a verziónál csak abban az esetben lehet a függvénynek több ága, ha az aritása nem változik. Fordítsuk le a programot, majd hívjuk meg különböző tesztadatokkal. Láthatjuk, hogy a függvény az első két ágnak megfelelő adatok esetén a helyes eredménnyel tér vissza, egyéb adatokkal való paraméterezéskor viszont 0-át ad eredményül. Ahogy a kivételkezelésről szóló részben már említettük, ez egy rendkívül egyszerű, és bevált módja a hibakezelésnek. Több ággal rendelkező függvények esetén gyakran használják az utolsó ágat, és az "_" paramétert arra, hogy a függvény minden körülmények között megfelelően működjön.

6.2. Rekord

A rekord a tuple adatszerkezet névvel ellátható változata olyan könyvtári függvényekkel "felvértezve", melyek a mezők, vagyis a rekord adatainak elérését segítik.

6.5. programlista. Nevesített tuple - Erlang

```
{szemely, Nev, Eletkor, Cim}
```

Az 6.5. programlistában egy tipikus rekord szerkezetet láthatunk, melynek a neve mellett van három mezője. A rekord hasonlít egy olyan tuple struktúrára, ahol a tuple első eleme egy atom. Amennyiben ebben a tuple-ben az

első elem atom, és megegyezik egy rekord nevével, a rekord illeszthető a `tuple`-ra, és fordítva. Ez azt jelenti, hogy függvények paraméter listájában, az egyik típus formális, a másik aktuális paraméterként illeszthető egymásra. A rekordokat deklarálni kell (6.6. programlista), vagyis a modulok elején be kell vezetni a típust. Itt meg kell adni a rekord nevét, valamint a mezőinek a neveit. A mezők sorrendje nem kötött, mint a `tuple`-ben. A rekord mezőire irányuló értékadás során a rekord egyes mezői külön-külön is kaphatnak értéket. Ezt a mechanizmust rekord update-nek nevezzük (6.14 programlista).

6.6. programlista. Rekord - Erlang

```
-record(nev, {mezo1, mezo2, ..., mezo}).
-record(ingatlan, {ar = 120000,
                   megye,
                   hely = "Eger"}).
```

6.7. programlista. Rekord Clean nyelven

```
:: Ingatlan = { ar :: Int,
                megye :: String,
                 hely :: String
                }
ingatlan1::Ingatlan
ingatlan1 = { ar = 120000,
              megye = "Heves",
               hely = "Eger"
              }
```

6.8. programlista. F# rekord definíció

```
type nev = {mezo1 : típus ;
            mezo2 : típus ;
          ... mezo : típus}

type ingatlan = {ar : int;
                 megye : string;
                  hely : string}
```

A 6.6. programban a második rekord definíció első mezője, az `ar` mező alapértelmezett értékkel rendelkezik, míg a `megye` nevű mező nem kapott értéket. Ez megszokott eljárás rekordok definíciója esetén, mert segítségével azok a mezők, melyek kötelező kitöltésűek, mindenképpen kapnak értéket. Természetesen az alapértelmezett érték bármikor felülírható. A rekordokat köthetjük változókba ahogy azt a 6.9. programlistában láthatjuk. A `rec1/0` függvényben az `X` változóba kötöttük az ingatlan rekordot. A rekord alapértelmezett értékkel rendelkező mezői megtartják az értéküket, azok a mezők viszont, amelyek nem kaptak a definíció során kezdőértéket, üresek maradnak.

Az értékkel nem rendelkező mezők hasonlóak az OO nyelvek listáinál, és rekord mezőinél használt `NULL` értékű elemekhez...

A `rec2/0` függvényben található értékadásban frissítettük a rekord `ar` mezőjét, valamint értéket adtunk a `megye` nevű mezőnek, így ez a függvény az értékadás, és a rekord frissítés műveletét is bemutatja. A `rec3/3` függvény az előzőek általánosítása, ahol a paraméterlistában kapott adatokkal lehet a rekord mezőit kitölteni.

6.9. programlista. Tuple és rekord - Erlang

```
-module(rectest).
-export([rec1/0, rec2/0, rec3/3]).
-record(ingatlan,{ar=120000, megye, hely="Eger"}).

rec1() -> R1 = #ingatlan{}.
```

```
rec2() ->
   R2 = #ingatlan{ar = 110000, megye ="Heves"}.

rec3(Ar, Megye, Hely) ->
   R3 = #ingatlan{ar = Ar, megye = Megye, hely = Hely}.
```

6.10. programlista. Tuple és rekord - F#

```
type ingatlan =
      { ar : int; megye : string; hely: string }

let rec1 =
       let R1 =
      { ar = 12000; megye = "Heves"; hely = "Eger"} R1

let rec2 ar megye hely =
      let R2 = { ar = ar;
                 megye = megye;
                  hely = hely } R2
```

A rekordok sokkal általánosabb felhasználásúvá teszik a függvényeket és az adattárolást, mivel segítségükkel a függvény paraméterek száma bővíthetővé válik. Mikor a függvény paramétere egy rekord, a függvény paraméterszáma sem kötött. A rekordot a definícióban kiegészíthetjük újabb mezőkkel, ezzel bővítve a függvény paramétereinek a számát...

6.11. programlista. Rekord update - Erlang

```
-module(rectest).
-export([writerec/1]).
-record(kedvenc, {nev = "Bloki", tulaj}).

writerec(R) ->
      io:format("~s", [R#kedvenc.nev]).
```

6.12. programlista. Rekord update – Clean

```
module rectest
import StdEnv

::Tulaj = { tnev    ::String,
              eletkor  ::Int }
::Kedvenc = { knev  ::String,
                 tulaj ::Tulaj}

tulaj1::Tulaj
tulaj1 = {tnev = "Egy Tulaj", eletkor=10}

kedvenc1::Kedvenc
kedvenc1 = {knev = "Bodri", tulaj=tulaj1 }

rectest r = r.tnev
Start = rectest kedvenc1.tulaj
```

6.13. programlista. Rekord update – F#

```
type kedvenc = { nev : string; tulaj : string }

let writerec (r : kedvenc) = printfn "%s" r.nev
```

Természetesen a rekord mezőire egyesével is lehet hivatkozni, ahogy azt a 6.11. programlistában láthatjuk. A `writerec/1` függvény egy rekordot kap paraméterként, majd az első mezőjét kiírja a képernyőre.

6.14. programlista. Rekord update - Erlang

```
-module(rectest).
-export([set/2, caller/0]).
-record(ingatlan, {ar, megye, hely}).

set(#ingatlan{ar = Ar , megye = Megye} = R, Hely) ->
     R#ingatlan{hely = Hely}.

caller() ->
     set({1200000, "BAZ"}, "Miskolc").
```

A 6.14. programlista már a funkcionális nyelvek világában is a "varázslat" kategóriába tartozik. A `set/1` függvény a paraméterlistájában látszólag egy értékadást tartalmaz, de ez valójában egy mintaillesztés, amely során a `caller/0` függvényben egy rekordot hozunk létre, mjd ellátjuk adatokkal. A `set/2` függvény második paramétere arra szolgál, hogy a függvény törzsében a `hely` mező is értéket kaphasson egy egyszerű értékadás során.

7. Függvények és rekurzió

7.1. Függvények készítése

Ahogy azt korábban említettük, a funkcionális programok moduljaiban függvényeket és egy kezdeti kifejezést definiálunk, majd egy kezdeti kifejezés segítségével elindítjuk a kiértékelést. Természetesen ez a kijelentés nem vonatkozik a könyvtári modulokra, ahol több, vagy akár az összes függvény meghívását szeretnénk lehetővé tenni a modul külvilága számára. Mivel minden funkcionális nyelvi elem visszavezethető a függvényekre, - egyes nézetek szerint minden függvény - meg kell ismerkednünk a különböző változatokkal, és azok használatával. A függvénynek rendelkeznie kell névvel, paraméterlistával és függvény törzzsel.

7.1. programlista. Erlang függvények általános definíciója

```
funName(Param1_1,...,Param1_N)
                    when Guard1_1,...,Guard1_N ->
     FunBody1;
funName(Param2_1,...,Param2_N)
                    when Guard2_1,...,Guard2_N ->
     FunBody2;
...
funName(ParamK_1, ParamK_2, ..., ParamK_N) ->
    FunBodyK.
```

7.2. programlista. Clean függvények általános definíciója

```
function args
| guard1 = expression1
| guard2 = expression2
where
       function args = expression
```

Ezek a részek elengedhetetlenek, de kiegészíthetjük őket újabbakkal. A függvénynek lehet őr feltétele, valamint több ággal is rendelkezhet. Az ágakat `clause`-oknak nevezzük. Természetesen az ágak is tartalmazhatnak őr feltételeket is.

7.3. programlista. F# függvények általános definíciója

```
// Nem-rekurzív
let funName Param1_1 ...
            Param1_N [: return-type] = FunBody1

// Rekurzív
let rec funName Param2_1 ...
            Param2_N [: return-type] = FunBody2
```

A neve azonosítja a függvényt. A név atom típusú, kis betűvel kezdődik, valamint nem tartalmazhat szóközt. A függvénynek lehet több aktuális paramétere, de írhatunk paraméterrel nem rendelkező függvényeket is. Erlang esetén a paraméterlistát határoló zárójelekre ekkor is szükség van, a Clean és az F# nyelvekben nincs ilyen megkötés. A törzs írja le azt, hogy mit kell a függvény meghívásakor végrehajtani. A törzs tartalmazhat egy vagy több utasítást, melyeket az adott nyelvre jellemző szeparátor segítségével választunk el egymástól. A szeparátor általában vessző (`,`), vagy pontosvessző (`;`).

Erlang-ban a függvény törzset a `.` zárja le, több ág esetén az ágakat `;` | választja el egymástól.

7.4. programlista. Erlang függvények

```
f(P1, P2) ->
       P1 + P2.

g(X, Y)->
       f(X, Y).

h()->
       ok.
```

7.5. programlista. Clean függvények

```
f p1 p2 = p1 + p2.

g x y = f x y

h = "ok"
```

7.6. programlista. F# függvények

```
let f p1 p2 = p1 + p2

let g x y = f x y

let h = "ok"
```

7.2. Rekurzív függvények

A függvények természetesen meghívhatnak más függvényeket, vagy saját magukat. Ezt a jelenséget, mikor egy függvény magát hívja, rekurziónak nevezzük. A rekurzió imperatív és OO nyelvekben is jelen van, de használata számos problémát okozhat, mivel a rekurzív hívások száma erősen korlátozott. A funkcionális nyelvekben a függvények futtatása, vagyis a verem kiértékelő rendszer teljesen eltérő a nem funkcionális nyelvek fordító programjainál megismerttől. Abban az esetben, ha a függvényben szereplő utolsó utasítás a függvény saját magára vonatkozó hívása - és ez a hívás nem szerepel kifejezésben - a rekurzív hívások száma nem korlátozott. Ha számos egymást hívó rekurzív függvény létezik, és minden rekurzív hívás farok-rekurzív, akkor a hívások nem "fogyasztják" a verem területet. A legtöbb általános célú funkcionális programozási nyelv megengedi a korlátlan rekurziót, amellett, hogy Turing-teljes marad a kiértékelése.

7.7. programlista. Imperatív do-while ciklus

```
int n = 10;
do
{
    Console.WriteLine("{0}",n)
     n--;
}
while(n >= 0)
```

A rekurzió funkcionális nyelvekben azért nagyon fontos eszköz, mert más lehetőség nem nagyon kínálkozik az ismétlések megvalósítására, mivel a ciklusok teljes mértékben hiányoznak.

7.8. programlista. F# do-while ciklus

```
let f =
        let mutable n = 10
         while (n >= 0) do
           printfn "%i" n
           n 03C- n - 1
```

7.3. Rekurzív ismétlések

A 7.9. programban láthatjuk, hogy miként lehet a 7.7. programban bemutatott C# nyelvű `do-while` ciklushoz hasonló ismétlés programját előállítani. A `dowhile/1` függvény két ággal rendelkezik. Az első ág minden lefutáskor csökkenti a paraméterként kapott szám értékét. Ezzel közelítünk a nulla felé, mikor is a második ág fut le, ami visszaadja a nulla értéket. Az első ágban a kiíró utasítást azért vezettük be, hogy láthassuk az egyes futások eredményét. Ilyen ismétléseket nem szoktunk használni, a példa is csak arra jó, hogy megmutassa a párhuzamot a ciklusok és a rekurzív ismétlések között. A gyakorlatban inkább listák feldolgozására, vagy szerverek tevékeny várakozásának a megvalósítására használjuk ezt a fajta ismétlő szerkezetet. A listák kezelésére, és egyszerű szerver alkalmazások készítésére láthatunk példákat a későbbi fejezetekben.

7.9. programlista. Ismétlés rekurzióval - Erlang

```
-module(rekism).
-export([dowhile/1]).

dowhile(N) ->
      N0 = N - 1,
      io:format("~w~n",[N0]),
      dowhile(N0);
dowhile(0) ->
      0.
```

A rekurziót használhatjuk az imperatív nyelvekben alkalmazott vezérlő szerkezetek és alapvető programozási tételek implementálására.

7.10. programlista. Ismétlés rekurzióval – F#

```
let rec dowhile n =
       match (n - 1) with
       | 0 -> printfn "0"
       | a -> printfn "%i" a
              dowhile (n - 1)
```

Ez egy igen sajátos megközelítése a rekurziónak, de nem példa nélküli, ezért szerepeljen itt is egy megvalósítása (7.11. programlista).

7.11. programlista. Összegzés C nyelven

```
osszeg = 0;
for(i = 0; i 003C max; i++)
{
   osszeg += i;
}
```

7.12. programlista. Összegzés Erlang nyelven

```
for(I, Max, Sum)
          when I 003C Max ->
         for(I + 1, Max, Sum + I);
for(I, Max, Sum) ->
       Sum.
```

7.13. programlista. Összegzés F#-ban

```
let sum i max =
    let mutable sum0 = 0
    for j = i to max do
            sum0 003C- sum0 + j
    sum0
```

7.14. programlista. Összegzés F#-ban rekurzívan

```
let rec sum i max =
       match i with
       | i when i 003C max -> (sum (i + 1) max) + i
       | _ -> i
```

7.4. Magasabb rendű függvények

A magasabb rendű függvények talán a legérdekesebb konstrukciók a funkcionális nyelvek eszköztárában. Segítségükkel függvény prototípusokat - igazság szerint lambda-kifejezéseket - adhatunk paraméterként függvények formális paraméterlistájában. Erlang nyelvi környezetben arra is lehetőségünk van, hogy függvényeket adjunk át változókba kötve, majd az így megkonstruált változókat függvényként alkalmazzuk.

A magasabb rendű függvények, és egyáltalán a funkcionális programozási nyelvek alapja a Lambda-kalkulus...

A magasabb rendű függvények üzenetekbe csomagolása, és elküldése távoli rendszerek számára olyan mértékű dinamizmust biztosít az üzenetküldéssel rendelkező funkcionális nyelveknek, melyeket az imperatív rendszerek esetében ritkán találunk meg. Lehetőségünk van a feldolgozásra szánt adatokat, és az azok feldolgozását végző függvényeket is egy üzenetbe csomagolva elküldeni a másik fél, vagy program számára hálózaton, vagy közös memória használata mellett. Így a kliens-szerver alkalmazásokat teljes mértékben általánosíthatjuk, ami lehetővé teszi a funkcióval való paraméterezésüket. A funkcionális nyelvekben a "Minden függvény..." kezdetű mondat a magasabb rendű függvények által értelmet nyerhet azok számára is, akik ezidáig csak OO és imperatív nyelvi környezetben dolgoztak...

7.17. programlista. Függvény kifejezések - Erlang

```
caller() ->
   F1 = fun(X) -> X + 1 end

   F2 = fun(X, inc) -> X + 1;
            (X, dec) X - 1 end

   F2(F1(10), inc).
```

7.18. programlista. Függvény kifejezések – F#

```
let caller() =
    let f1 = (fun x -> x + 1)
    let f2 = (fun x exp ->
                    match exp with
                    | "inc" -> x + 1
                    | "dec" -> x - 1)

    f2 (f1 10) "inc"
```

A 7.17. példában az első függvényt az `F1` nevű változóhoz rendeltük hozzá. Az `F2` változóba az előzőhöz hasonló módon egy több ággal rendelkező függvényt kötöttünk, ami vagy megnöveli, vagy lecsökkenti eggyel az első paraméterében kapott értéket, attól függően, hogy `inc`, vagy `dec` az első paraméterként megadott atom.

A magasabb rendű függvények rendelkezhetnek több ággal, mely ágak attól függően futnak le, hogy melyik ágra illeszkedő paraméterrel hívtuk meg a függvényt. Az ágakat `;`-vel választjuk el egymástól, és az ágak kiválasztása mintaillesztéssel történik...

Vizsgáljuk meg a 7.17. programlistát. A listában szereplő modul `caller/0` függvényét meghívva, a futás eredménye 12 lesz, mivel az `F1(10)` megemeli az értéket 11-re, majd behelyettesíti a 11-et az `F2(11, inc)` hívásnál, ami megnöveli a kapott értéket 12-re.

Természetesen ez a program nem indokolja a magasabb rendű függvények alkalamazását, igazság szerint a "hagyományos" változat sokkal átláthatóbb. Ismertetésének az egyetlen oka, hogy szemléletesen bemutatja ennek a különleges nyelvi konstrukciónak a használatát...

A magasabb rendű függvények lehetnek más függvények paraméterei - itt a függvény prototípussal paraméterezünk - , vagy elhelyezhetjük őket lista kifejezésekben, ami szintén egy érdekes felhasználási terület. Természetesen a lista kifejezések használatáról később szót ejtünk. A függvénnyel való paraméterezés segítségünkre van abban, hogy teljesen általános függvényeket, vagy akár modulokat hozzunk létre. Erre a felhasználási módra egyszerű példát láthatunk a 7.19. programban.

7.19. programlista. Magasabb rendű függvények modulja - Erlang

```
-module(lmd).
-export([use/0, funct/2]).

funct(Fun, Data) ->
      Fun(Data);
funct(Fun, DataList) when is_list(DataList)->
      [Fun(Data) || Data 003C- DataList].

use() ->
      List = [1, 2, 3, 4],
      funct(fun(X) -> X + 1 end, List),
      funct(fun(X) -> X - 1, 2).
```

7.20. programlista. Magasabb rendű függvények modulja – F#

```
//a "use" foglalt kulcsszó, helyette van "funct"

let funct1 fn data = fn data

let funct2 fn dataList =
       match dataList with
       | h :: t -> List.map fn dataList

let funct() =
       let List = [1; 2; 3; 4]
       (funct2 (fun x ->  x + 1) List),
       (funct1 (fun x -> x + 1) 2)
```

```
//alternatív megoldás:
let funct() =
       let List = [1; 2; 3; 4]
      funct2 (fun x ->  x + 1) List
       funct1 (fun x -> x + 1) 2
```

A magasabb rendű függvényeket - az "egyszerű" függvények mintájára - egymásba ágyazhatjuk (7.20. programlista), valamint ezek is rendelkezhetnek több ággal, ahogy ezt az egyszerű függvényeknél láthattuk.

7.21. programlista. Egymásba ágyazott lambda függvények - Erlang

```
fun(fun(X) -> X - 1 end) -> X + 1 end.
```

7.22 ábra. Egymásba ágyazott lambda függvények - F#

```
let caller = (fun x -> (fun x -> x - 1) x + 1)
```

7.5. Függvény kifejezések

A függvényekre hivatkozhatunk a nevükből, és az `aritásukból` képzett párossal is. Az ilyen függvény kifejezésekre (7.21. programlista) tekinthetünk úgy, mint egy függvényre mutató referenciára.

7.23. programlista. Függvény referencia - Erlang

```
-module(funcall).
-export([caller/0, sum/2]).

sum(A, B) ->
        A + B.

caller() ->
        F = fun sum/2,
        F(10, 20).
```

7.24 ábra. Függvény referencia - Clean

```
summa a b = a+b

caller = f 10 20
      where f x y = summa y x / 2
```

A 7.21 példaprogramban a `sum/2` függvényre a `caller/0` függvényben úgy hivatkozunk, hogy egy változóba kötjük, és a változó nevét felhasználva hívjuk meg a megfelelő paraméterekkel.

7.25. programlista. Függvény referencia - F#

```
let sum a b = a + b

let caller =
        let f = sum
        f 10 20
```

Ezt a megoldást alkalmazhatjuk függvények paraméter listájában is, valamint olyan esetekben, ahol függvények egy halmazából kell kiválasztani azt, amit éppen meg szeretnénk hívni. Ahhoz, hogy teljes mértékben megértsük

a párhuzamot a két változat között, írjuk át az előző (7.21. programlista), függvény kifejezést használó változatot úgy, hogy a `caller/0` függvényben szereplő `F = sum/2` kifejezést kibontjuk.

7.26. programlista. Kibontott függvény kifejezés - Erlang

```
-module(funexp).
-export([caller1/0, caller2/0]).

sum(A, B) ->
       A + B.

caller1()->
       F = fun(V1, V2) ->
                   f(V1, V2) end,
       F(10, 20).

caller2() ->
       sum(10, 20).
```

A 7.26. példában szereplő függvény első változatában a függvény kifejezés kibontását egy magasabb rendű függvénnyel oldottuk meg.

A második változatban egyszerűen csak meghívtuk a `sum/2` függvényt. Ennél a megoldásnál egyszerűbb nem nagyon létezik, de az eredményt, és a működést tekintve ez is azonos az eredeti, és az átalakítás utáni első verzióval. A szerver alkalmazásokban gyakori, hogy a szerver nem tartalmazza az elvégzendő feladatok programját. A függvényeket, és az adatokat is üzenetek formájában kapja meg, elvégzi a kapott feladatot, majd visszaküldi az eredményt a kliens számára. Az ilyen, általánosítás elvén alapuló rendszerek nagy előnye, hogy a szerver-alkalmazás erőforrást takarít meg a kliensek számára, másrészt pedig teljesen általánosan tud működni. Mivel az elvégzendő feladat kódját is a kliensektől kapja magasabb rendű függvények formájában, a forráskódja is viszonylag egyszerű tud maradni, bármilyen bonyolult feladat elvégzése mellett is (7.27. programlista).

7.27. programlista. Függvény hívás üzenetküldéssel – Erlang

```
loop(D) ->
      receive
            {exec, From, Fun, Data} ->
               From ! Fun(Data);
            ...
    end
```

Máskor a szerver tartalmazza a futtatni kívánt függvényeket, de a kliens választja ki, hogy aktuálisan melyiket kell futtatni. Ekkor a szervert név-aritás párossal paraméterezzük, és csak a számításokhoz szükséges adatokat bocsájtjuk a rendelkezésére. Ezeknél az alkalmazásoknál igazán hasznosak a függvény kifejezések, és általában a magasabb rendű függvények, mivel kevés egyéb lehetőség adódik a mobil kódok egyszerű, üzenetekben történő küldésére, és futtatására.

8. Listák és halmazkifejezések

8.1. Lista adatszerkezet

A lista adatszerkezet lényegesen bonyolultabb mint a `tuple`, de cserébe a bonyolultságáért olyan lehetőségeket ad a programozó kezébe, mely más adattípusok használatával nem érhető el. A listákhoz szorosan kapcsolódik az előállításukra alkalmas konstrukció, a lista-kifejezés. Alapja a Zermelo-Fraenkel féle halmazkifejezés. A lista-kifejezés valójában egy generátor, mely furcsa mód nem a lista elemeit, hanem a listába tartozás feltételét írja le, valamint azt, hogy a lista egyes elemeit hogyan és milyen számban kell előállítani.

Ezt a konstrukciót hívják listagenerátornak, másnéven halmazkifejezésnek. A magyar listagenerátor elnevezés megtévesztő lehet, mivel angolul a konstrukció neve list comprehension, és a list-comprehension első eleme a `generator`. A magyar nyelvben gyakran a teljes kifejezésre alkalmazzák a listagenerátor elnevezést. Mi a könnyebb megértés érdekében használjuk a lista-kifejezés nevet...

Ezzel a technológiával végtelen darabszámú listákat is definiálhatunk, mivel nem a lista elemeit, hanem a lista definícióját írjuk le. A listák használhatóak mintaillesztésben, vagy függvények visszatérési értékeként, és szerepelhetnek a programok bármely részében, ahol egyébként az adatok is. A listák szétbonthatóak (mintaillesztéssel) egy fej, valamint egy farok részre, ahol a fej az első elem, a farok a lista további elemeit tartalmazó lista, melyben a fej elem nem szerepel. A listák a 8.1. programlistában látható általános szintaxissal írhatók le.

8.1. programlista. Lista szintaxisa - Erlang

```
[E || Qualifier_1, Qualifier_2, ...]
```

A 8.2., és 8.3. programszöveg bemutatja a statikus listák definiálásának, valamint kezelésének a módját. A programrészlet listákat köt változókba, majd azokat más listákban újra felhasználja.

8.2. programlista. Listák kötése változóba - Erlang

```
Adatok = [{10,20}, {6,4}, {5,2}],
Lista = [A, B, C, D],
L = [Adatok, Lista]...
```

8.3. programlista. Listák kötése változóba F#

```
let Adatok = [(10, 20); (6, 4); (5, 2)]
let Lista = [A, B, C, D]
let L = [Adatok, Lista]
```

A listák előállítására (generálására) számos eszköz áll rendelkezésre a funkcionális nyelvekben. Használhatunk erre a célra rekurzív függvényeket, lista-kifejezést, vagy igénybe vehetjük az adott nyelvben rendelkezésre álló könyvtári függvényeket.

8.2. Statikus listák kezelése

Minden *L* lista felbontható egy fej (`Head`) elemre, és a lista maradék részére (`Tail`). Ez a felbontás, és a felbontás rekurzív ismétlése a lista mindenkori második részére (*[Head|Tail'] = Tail*) lehetővé teszi a lista rekurzív bejárását.

A 8.4. példában szereplő mintaillesztés csak egy elemet vesz le a lista elejéről, de ha többször alkalmaznánk, akkor mindig az aktuális első elemet venné ki a listából, így előbb-utóbb a teljes listát feldolgozná, viszont az ismétlésekhez, és ezzel együtt a teljes feldolgozáshoz egy rekurzív függvényre lenne szüksége.

8.4. programlista. Mintaillesztés listákra - Erlang

```
L = [1,2,3,4,5,6],
[Head| Tail] = L...
```

8.5. programlista. Mintaillesztés listákra – F#

```
let L = [1; 2; 3; 4; 5; 6]
match L with
| Head :: Tail -> …
```

Vegyük észre, hogy a 8.4. programban a `Head` valtozó egy adatot tartalmaz (egy számot), a `Tail` viszont egy listát. Ez a további feldolgozás szempontjából lényeges momentum, mivel a lista elejét és a végét másképp kell kezelnünk...

Tehát a lista feje mindig az aktuális első elem, a vége pedig a maradék elemek listája, melyeket mintaillesztéssel el tudunk különíteni az elejétől, és további feldolgozásra átadni a függvény következő rekurzív futásakor.

8.6. programlista. Lista bejárása rekurzív függvénnyel - Erlang

```
listabejaras(Acc, [H|T]) ->
      Acc0 = Acc + H,
      listabejarass(Acc0, T);
listabejaras(Acc, []) ->
      Acc.
```

8.7. programlista. Lista bejárása rekurzív függvénnyel – Clean

```
listabejaras [h:t] = h + listabejaras t
listabejaras [] = 0
```

8.8. programlista. Lista bejárása rekurzív függvénnyel – F#

```
let rec listabejaras acc list =
    match list with
    | h :: t -> listabejaras (acc + h) t
    | [] -> acc
```

Az egy elemű lista elérésekor a `Head` megkaphatja a lista egyetlen elemét, a `Tail` pedig a maradékot, vagyis az üres listát. Az üres listát a rekurzív függvény második ága kezeli úgy, hogy megállítja a rekurzív futást. Igazság szerint az üres lista a bázis feltétele a rekurziónak. A lista elemeit a rekurzív futás során tetszőleges módon feldolgozhatjuk, összegezhetjük, vagy éppen kiírhatjuk a képernyőre. A megfelelően megírt rekurzív függvények funkcionális nyelvek esetén nem fenyegetnek leállással, ezért bármilyen hosszú listát képesek vagyunk a segítségükkel feldolgozni, legyen szó listák előállításáról, vagy bejárásról. Hogy jobban megértsük a rekurzív feldolgozás lényegét, készítsük el azt a függvényt, ami egy tetszőleges, számokat tartalmazó lista elemeit összegzi.

8.9. programlista. Lista elemeinek összegzése - Erlang

```
-module(list1).
-export([osszeg/2]).

sum(Acc, [Head|Tail]) ->
        Acc0 = Acc + Head,
        sum(Acc0, Tail);
sum(Acc, []) ->
        Acc.
```

8.10. programlista. Listafeldolgozás – Clean

```
module list1
import StdEnv

sum [head:tail] = head + sum tail
sum [] = 0
```

8.11. programlista. Listafeldolgozás – F#

```
let rec sum acc list =
       match list with
        | h :: t ->
```

```
            let mutable acc0 = acc + h
            sum acc0 t
        | [] -> acc
```

A 8.9. programban a `sum/2` függvény szerkezetét tekintve egy több ággal rendelkező függvény. Az első ág feladata, hogy a paraméterként kapott lista első elemét leválassza a listáról, és kösse a `Head` változóba, majd meghívja önmagát az aktuális összeggel, és a lista maradékával. A második ág megállítja a rekurziót amennyiben elfogytak az elemek a listából, vagyis, ha az aktuális (második) paramétere üres lista. Az üres listát a `[]` formulával írjuk le. A függvény visszatérési értéke a listában szereplő számok összege, melyet a rekurzív futás alatt az `Acc`, és az `Acc0` változókban tárolunk.

Az `Acc0` változóra azért van szükség, mert mint tudjuk, az `Acc = Acc + H` destruktív értékadásnak számít, így funkcionális nyelvekben nem alkalmazhatjuk...

Amennyiben a paraméterként megadott lista nem számokat tartalmaz, a függvény akkor is működőképes, de mindenképpen olyan típusú elemeket kell tartalmaznia, amelyekre értelmezhető a + operátor...

Készítsük el a `sum/2` függvényt implementáló modult, fordítsuk le, majd hívjuk meg parancssorból (8.12. programlista).

8.12. programlista. Összegzés programjának futtatása - Erlang

```
> c(lista1).
> {ok, lista1}
> List = [1, 2, 3, 4].
> List.
> 1, 2, 3, 4
> lista1:osszeg(0, List).
> 10
```

8.13. programlista. Összegzés programjának futtatása - Clean

```
modul lista1
import StdEnv

osszeg [head:tail] = head + osszeg tail
osszeg [] = 0

L=[1,2,3,4]
Start = osszeg L
```

8.14. programlista. Összegzés futtatása az F# interaktív ablakban

```
val sum : int -> int list -> int
> let List = [1; 2; 3; 4];;

val List : int list = [1; 2; 3; 4]
> sum 0 List;;
val it : int = 10
```

Az összegzés programja egyszerű műveletet implementál. A rekurzió egy ágon fut, és a visszatérési értéke is egy elem. Annak érdekében, hogy jobban megértsük a rekurzív feldolgozást, készítsünk még néhány, kicsit összetettebb listakezelő alkalmazást.

8.15. programlista. Listák kezelése – Erlang

```
-module(functions).
-export([osszeg/2, max/1, avg/1]).
```

```
osszeg(Acc, [Head|Tail]) ->
        Acc0 = Acc + Head,
       osszeg(Acc0, Tail);
 osszeg(Acc, []) ->
        Acc.

max(List)->
       lists:max(List).

avg(List) ->
       LList = [Num || Num 003C- List,
            is_integer(Num), Num =/= 0],
       NData = lists:foldl(
          fun(X, Sum) -> X + Sum end, 0, List),
       NData/length(LList).
```

8.16. programlista. Listák kezelése - Clean

```
module functions
import StdEnv
osszeg [head:tail] = head + osszeg tail
osszeg [] = 0
maximum [x] = x
maximum [head:tail]
      | head > res = head
      | otherwise = res
       where res = maximum tail

average lst = toReal(foldl sum 0 lst)
            / toReal(length lst)
   where
    sum x s = x + s
```

8.17. programlista. Listák kezelése – F#

```
let rec osszeg acc list =
       match list with
       | h :: t ->
            let mutable acc0 = acc + h
            osszeg acc0 t
       | [] -> acc

let max list = List.max list

let avg (list: float list) =
                       List.average list
```

Rekurzív függvények segítségével elő is tudunk állítani új listákat, vagy a régieket át tudjuk alakítani egy újabb változóba kötve. Az ilyen feladatot ellátó függvények paramétere lehet egy lista (vagy olyan konstrukció, amiből " elemek jönnek ki"), a visszatérési értéke pedig egy másik, ami a generált elemeket tartalmazza.

8.18. programlista. Listák előállítása és összefűzése - Erlang

```
-module(functions).
-export([comp1/1, comp2/2, comp3/3]).

 comp1({A, B , C}) ->
        [A, B ,C].

 comp2(List, [Head| Tail]) ->
       List1 = List ++ add(Head, 1),
       comp2(List1, Tail);
 comp2(List, []) ->
       List.
```

```
 comp3(Fun, List, [Head| Tail]) ->
        List1 = List ++ Fun(Head),
        comp3(Fun, List1, Tail);
 comp3(_, List, []) ->
        List.
```

8.19. programlista. Listák előállítása és összefűzése - Clean

```
module functions
import StdEnv

comp1 a b c = [a, b, c]

comp2 [head:tail] = [head+1] ++ comp2 tail
comp2 [] = []

comp3 fun [head:tail] = fun [head] ++ comp3 fun tail
comp3 fun [] = []
```

A 8.18. példamodul három függvényt tartalmaz. A `comp1/1` egyetlen sora a paraméterként érkező hármas elemeit egy listába helyezi.

8.20. programlista. Listák előállítása és összefűzése – F#

```
let comp1 (a, b, c) = [a; b; c]

let rec comp2 list1 list2 =
       match list2 with
       | head :: tail ->
              comp2 (list1 @ [head]) tail
       | [] -> list1

let rec comp3 fn list1 list2 =
       match list2 with
       | head :: tail -> comp3 fn
                      (list1 @ [fn head]) tail
       | [] -> list1
```

A `comp2/2` tipikus rekurzív listafeldolgozás, ami több ággal rendelkezik. Az első ág a harmadik paraméterben érkező lista első eleméhez hozzáad egyet, majd elhelyezi egy új listában, amit átad a következő hívásnak. Üres lista esetén megáll és visszatér az eredmény listával. A `comp3/3` hasonlóan működik, mint a `comp2/2`. Tekinthetünk erre a függvényre úgy, mintha a `comp3/3` általános változata lenne, mivel az első paramétere egy függvény kifejezés, amit meghívunk a lista minden elemére. Ez a változat azért általánosabb az előzőnél, mert nem csak egy adott függvényt képes meghívni a lista elemeire, hanem bármilyet. A program teljesen általánosított változatának implementációját a 8.21. programlista tartalmazza. A példa azért kapott helyet a fejezetben, hogy megmutassuk a magasabb rendű függvények egyik gyakori felhasználását.

8.21. programlista. Függvény általánosítása – Erlang

```
-module(list3).
-export([comp3/3, use/0]).

 comp3(Fun, List, [Head| Tail]) ->
        List1 = List ++ Fun(Head),
        comp3(Fun, List1, Tail);
 comp3(_, List, []) ->
        List.

 use() ->
        List = [1, 2, 3, 4],
```

```
        comp3(fun(X) -> X + 1 end, [], List).
```

8.22. programlista. Lista magasabb rendű függvénnyel – Clean

```
import StdEnv

comp3 funct [head:tail] = funct head
                              ++ comp3 funct tail
comp3 funct [] = []

use = comp3 plus lista
    lista = [1,2,3,4]
    where plus n = [n+1]
```

Láthatjuk, hogy a példaprogramban a `use/0` függvénynek csak annyi szerepe van, hogy meghívja a `comp3/3` függvényt, és előállítsa a listát.

8.23. programlista. Lista magasabb rendű függvénnyel – F#

```
let rec comp3 fn list1 list2 =
       match list2, fn with
       | head :: tail, fn -> comp3
             fn (list1 @ [fn head]) tail
       | [], _ -> list1

let funct =
       let List = [1; 2; 3; 4]
       comp3 (fun x -> x + 1) [] List
```

Amennyiben futtatni szeretnénk ezt a programot, fordítsuk le, majd hívjuk meg a `use/0` függvényt a modul-minősítővel (8.24. programlista).

8.24. programlista. A use/0 meghívása - Erlang

```
> c(list3).
> {list3, ok}
> list3:use().
> […]
```

8.25. programlista. Use meghívása – Clean

```
module list3
import StdEnv

comp3 funct [head:tail] = funct head
++ comp3 funct tail
comp3 funct [] = []

use = comp3 plus lista
    lista = [1,2,3,4]
    where plus n = [n+1]

Start = use
```

Ha jobban megnézzük a 8.21. és a 8.18. programokat, láthatjuk, hogy a `comp2/3` és a `comp3/3` függvények ugyanazt az eredményt produkálják, ha az add függvényben ugyanaz "történik", ami a függvény kifejezésben, de a `comp3/3` többcélú felhasználásra is alkalmas, mivel az első paraméterben megadott függvényt tetszés szerint változtathatjuk.

A függvények általánosítása hatékonyabbá, és széles körben alkalmazhatóvá teszi a programjainkat. Amikor csak lehetőségünk van rá, készítsünk minél általánosabban felhasználható függvényeket...

8.3. Lista kifejezések

A funkcionális nyelvek eszköztárában számos olyan érdekes konstrukciót találunk, melyeket a hagyományos imperatív, vagy OO nyelvekben nem. Ilyen eszköz a lista-kifejezés, melyet listák feldolgozására, és előállítására egyaránt használhatunk. A lista-kifejezések a listákat, vagy ha úgy tetszik, a halmazkifejezéseket dinamikusan, vagyis a program futása közben állítják elő. Ez a gyakorlatban azt jelenti, hogy nem a lista elemeit írjuk le, hanem azt, hogy a listát hogyan, és milyen feltételek mellett kell előállítani. Ez a dinamizmus elméletben végtelen hosszú listák definiálását is lehetővé teszi. Mindezek ellenére a listagenerátorok legtöbbször listákból állítanak elő újabb listákat (8.26. programlista).

A lista kifejezésekben függvényeket alkalmazhatunk a lista tetszőleges elemére úgy, hogy nem kell rekurzív függvényeket készítenünk, valamint összefésülhetünk, vagy szétválogathatunk listákat akár összetett feltételek alapján.

8.26. programlista. Listafeldolgozás lista-kifejezéssel – Erlang

```
FList = [1, 2, 3, 4, 5, 6],
Lista = [A| A 003C- FList, A /= 0]...
```

A 8.27. példában szereplő listagenerátor azokat az elemeket válogatja ki egy listából, melyek értéke nem nulla. A lista kifejezés két részből áll. Az első elem a listát állítja elő, vagyis ez a generátor rész. A második azoknak az elemeknek a sorozatát adja, melyből az új listát állítjuk elő (ezt a listát nevezhetjük forrás listának). Van egy harmadik, opcionális rész, ahol feltételeket adhatunk a lista elemek feldolgozásához. A feltételekből több is lehet, és minden feltétel hatással van a forrás lista feldolgozására. A sorban megadott feltételek úgy érvényesülnek, mintha `AND` operátor lenne közöttük.

8.27. programlista. Listafeldolgozás lista-kifejezéssel – F#

```
let fLista = [1; 2; 0; 3; 4; 5; 6]
let lista = List.filter (fun a -> a 003C
    003E 0) fLista
```

Amennyiben a generátor részben több függvényhívást, vagy kifejezést is el szeretnénk helyezni, a `begin-end` kulcsszavakat kell használnunk a csoportok képzésére. A `begin` és az `end` kulcsszavak blokkot képeznek, melyben az utasítások szekvenciálisan hajtódnak végre (8.28. programlista).

8.28. programlista. Begin-end blokk használata lista kifejezésben - Erlang

```
Lista = [begin f1(Elem), f2(Elem) end
                || Elem 003C- KLista, Elem >= 0, Elem 003C 10]
```

8.29. programlista. Begin-end blokk - F# " legközelebbi megoldás”

```
let lista =
       let kLista = List.filter
            (fun elem -> elem >= 0 0026
         0026 elem 003C 10) kLista
       let kLista = List.map f1 kLista
       let kLista = List.map f2 kLista
        kLista
```

A 8.28. példában a lista kifejezés eredményét egy változóba kötöttük. A `[]` között az első elem az új lista aktuális elemét definiálja `begin-end` blokkba zárva. Ebben a részben azt a kifejezést kell elhelyeznünk, ami

megadja az elem létrehozásának a módját. A `||` jelek után található az eredeti lista aktuális eleme, vagy az a kifejezés, ami bonyolultabb adattípus esetén "kicsomagolja”" az elemet. A sort a `003C-` követően az eredeti lista, majd újabb vessző után a feltételek zárják, melyek az elemek listába kerülését szabályozzák. A feltétel valójában egy szűrő mechanizmus, ami a megfelelő elemeket átengedi az új listába, a többit pedig " eldobja”.

8.30. programlista. Összetett feltételek lista kifejezésben - Erlang

```
List2 = [begin filter(Elem), add(Elem) end ||
                    Elem 003C- List1, Elem > 0, Elem 003C100]
```

A 8.30. példaprogramban elhelyeztünk két feltételt is, ami 1 és 100 között szűri a bemeneti lista elemeit. A felsorolt feltételek sorban érvényesülnek, közöttük `ÉS` kapcsolat van.

8.31. programlista. Összetett feltételek lista kifejezésben - F#

```
let list2 = List.filter
             (fun elem -> elem > 0 0026
    0026 elem 003C 100)
```

A feltételek alkalmazása mellett a kiinduló listák elemszáma nem mindig egyezik meg az eredmény lista hosszával...

A konstrukció felépítése alapján könnyedén el tudjuk képzelni, hogyan működik a lista generálása. Az eredeti lista elemeit sorra vesszük, majd a feltétel alapján hozzáadjuk az új listához a generátor részben leírt kifejezés alkalmazása mellett.

8.32. programlista. Lista előállítása függvénnyel - Erlang

```
listmaker() ->
  List1 = [{2, elem1}, 3, 0, {4, {pos, 2}}],
  List2 = [Add(Elem, 1) || Elem 003C- List1],
  List2.

add(Elem, Value) - >
        Elem + Value;
add({Elem, _}, Value) ->
       Data = {Elem, _},
       Data + Value;
add(_, _ ) ->
    0.
```

8.33. programlista. Listák feldolgozása függvénnyel - F# tömör változat

```
let add elem value = elem + value

let listmaker =
       let list1 = [1; 2; 3; 4; 5]
       let list2 = List.map
             (fun elem -> add elem 1) list1
       list2

//kimenet:
val listmaker : int list = [2; 3; 4; 5; 6]
```

Ha a generálás során nem szeretnénk `begin-end` blokkot használni, az ide szánt kifejezéseket elhelyezhetjük egy függvényben, majd a függvényt meghívhatjuk a generátorban minden elemre, ahogy ezt a 8.33. programban is láthatjuk. A `List1` elemei adottak, de a formátumuk nem megfelelő (inhomogén). A feldolgozás során ezekből az elemekből állítjuk elő az új listát úgy, hogy minden elemhez hozzáadunk egy tetszőleges értéket, jelen esetben egyet az `add/2` függvény használatával. Az `add/2` függvénybe beépítettünk egy apró trükköt. A

függvény két ággal rendelkezik, ami abban az esetben hasznos, ha a kiindulási listánk felváltva tartalmaz rendezett n-eseket és számokat.

A gyakorlati munka során nem ritka, hogy nem homogén adatokat kapunk feldolgozásra. Ekkor alkalmazhatjuk az OO nyelvekben is ismert `overload` technológiát, vagy használhatunk mintaillesztést az elemek szelektálására...

Az `add` függvény tehát rendezett n-es paramétert kapva "kicsomagolja" az adott elemet, majd hozzáadja a `Value` változóban kapott számot, egyszerű érték-paraméterre meghívva viszont " simán”' elvégzi az összeadást. A függvény harmadik ága nulla értéket ad vissza, ha rossz paraméterekkel hívták meg. Amennyiben a megfelelő módon elkészítjük és futtatjuk a programot, az a `Lista2`-ben szereplő paraméterek alapján a szövegben látható sorozatot állítja elő, mivel kiindulási lista első eleme egy n-es, ami egy számot és egy atomot tartalmaz. Az `add/2` második ága "kicsomagolja" a számot és hozzáad egyet, így az eredmény a hármas szám lesz.

8.34. programlista. A 8.33. program kimenete

```
[3, 4, 5]
```

A második elem úgy jön létre, hogy a generátorban szereplő függvény első ága egyszerűen csak megnöveli a kapott paraméter értékét eggyel. Az eredeti lista harmadik eleme nulla, ezért a generálásban elhelyezett feltétel miatt kimarad. Végül az új lista harmadik eleme az eredeti negyedik eleméből jön létre, és az értéke öt lesz.

8.4. Összetett és beágyazott listák

A lista-kifejezéseket úgy, mint a listákat egymásba is ágyazhatjuk tetszőleges mélységben, de vigyázzunk, mert a túl mély beágyazás olvashatatlanná teszi a programjainkat, esetenként le is lassíthatja a működésüket.

8.35. programlista. Beágyazott lista - Erlang

```
Lista1 = [{egy, 1}, {ketto, 2}, ...,
        {sok, 1231214124}],
Lista2 = [Elem + 1 || Elem
      003C- [E || {E, _ } 003C- Lista1]]
```

A 8.35. programban az egymásba ágyazás során elsőként a legbelső lista-kifejezés érvényesül. Ezután a külső kifejezésre kerül a sor, ami a kapott elemeket úgy kezeli, mintha azok egy egyszerű lista elemei lennének.

A funkcionális nyelvek egyik alapvető építő eleme a lista, valamint a lista-kifejezés, ezért nagy valószínűséggel minden ilyen nyelv rendelkezik könyvtári modullal, amely lehetővé teszi az ismert listakezelő eljárások alkalmazását a nyelv függvényein keresztül... Annak az eldöntése viszont, hogy a listák készítését, feldolgozását és kezelését milyen módon végzi el, mindig a programozótól függ. Sokan használják a lista-kifejezéseket, de van aki a listakezelő könyvtári függvényekre esküszik. Nem jobb egyik sem a másiknál, és a legtöbb esetben az adott probléma alapján kell eldönteni, hogy melyik a járható, és főként a könnyebben járható út.

9. Funkcionális nyelvek ipari felhasználása

9.1. Funckionális nyelvek az iparban

A funkcionális nyelvek az iparban, valamint a telekommunikáció területén sem elhanyagolható mértékben váltak ismertté. Az Erlang-ot használják kommunikációs, és egyéb nagy hibatűrést igénylő rendszerek készítésére, és a Clean nyelv is számos ipari projektben megjelenik. Funkcionális nyelveken programoznak matematikai számításokat végző rendszereket, és más tudományágak, mint a fizika és a kémia művelői számára is lehetőséget biztosítanak a gyors és hatékony munkavégzéshez, mivel a matematikai formulák egyszerűen átírhatóak az adott programozási nyelv dialektusára.

9.2. Kliens-szerver alkalmazások készítése

Azért, hogy bizonyítsuk korábbi, szerverekkel kapcsolatos állításainkat, és azért is, hogy bemutassuk az üzenetküldések, és a konkurens programok készítésének a lehetőségeit, a 9.1. listában implementáltunk egy üzenetek küldésére és fogadására alkalmas szervert.

9.1. programlista. Kliens-szerver alkalmazás kódja - Erlang

```
-module(test_server).
-export([loop/0, call/2, start/0]).

start()->
     spawn(fun loop/0).

call(Pid, Request) ->
   Pid ! {self(), Request},
   receive
      Data ->
             Data
 end.

loop() ->
     receive
       {From, hello} ->
          From ! hello,
           loop();
       {From, {Fun, Data}} ->
         From ! Fun(Data),
           loop();
       {From, stop} ->
           From ! stopped;
       {From, Other} ->
           From ! {error, Other},
           loop()
  end.
```

Mielőtt azonban a kód elemzésébe kezdenénk, meg kell ismernünk a szerverek futtatásához szükséges hátteret. A különböző szerver alkalmazások úgynevezett `node()`-okon futnak. Nagyon leegyszerűsítve a dolgot, a *node* egy futó alkalmazás, mely a nevén keresztül szólítható meg. Az egymással kommunikáló *node*-ok futhatnak különálló gépeken, de egy számítógépen is. Ha ismerjük az adott *node* azonosítóját, üzeneteket küldhetünk neki, pontosabban meghívhatjuk a kódjában implementált függvényeket. Az 9.2. programlistában szereplő hívások a *node1@localhost* azonosítóval ellátott *node*-ot szólítják meg. A `Mod` változó tartalmazza a *node* moduljának a nevét, a `Fun` a modulban szereplő függvényt, amit meg akarunk hívni, és a `Params` a függvény paramétereit. Ha nincs paramétere a függvénynek, akkor üres listát kell a paraméterek helyett megadni. Természetesen a szerver alkalmazás használata előtt el kell indítani a `spawn(Mod, Fun, Params)` hívással, ahol a változók szintén a modul nevét, az indításhoz használható függvényt, valamint a megfelelő paramétereket tartalmazzák.

A `spawn` *node*-okat indít el külön szálakon, melyeket névvel láthatunk el. Ez a név lesz a *node* azonosítója. Ha nem akarunk nevet rendelni hozzá, a processz ID-jét is használhatjuk, ami a *spawn* visszatérési értéke, és szintén azonosítja a futó alkalmazást.

9.2. programlista. Távoli eljárás hívás parancssorból – Erlang

```
> rpc:call(node1@localhost, Mod, Fun, []).
> rpc:call(node1(@localhost, Mod, Fun, Params).
```

A 9.2. példaprogram bemutatja az üzenetküldés, és a processzek indításának a lehető legegyszerűbb módját. A bemutatásra kerülő szerver alkalmas arra, hogy fogadja a kliensektől érkező adatokat, és az adatok feldolgozására küldött függvényeket futtassa, majd a számítások elvégzése után visszaküldje az eredményt annak a processznek, akitől kapta a kérést.

A szerver `loop/0` függvénye végzi a tevékeny várakozást, és kiszolgálja az érkező kéréseket, majd rekurzívan meghívja saját magát, hogy újabb kéréseket tudjon fogadni. A `start/0` függvény indítja el a szervert a `spawn` függvény meghívásával, ami " fellövi” a szerver processzt.

Ennek a függvénynek a visszatérési értéke szolgáltatja a kliensek számára a szerver processz azonosítóját, vagyis az elérhetőségét. Mikor elindítjuk a szervert, ezt az értéket érdemes egy változóban tárolni, hogy bármikor megszólíthassuk az alkalmazást (9.3. programlista).

9.3. programlista. Szerver indítása - Erlang

```
> c(test_server).
> {ok,test_server}
> Pid = test_server:start().
> 003C0.58.0003E
```

A `loop/0` négy dologra alkalmas, megszólítani pedig a szintén a modulban szereplő `call/2` függvény meghívásával lehet (9.4 programlista).

Ebben a példában a szervert egy függvény kifejezéssel és egy számmal paramétereztük. A szerver a hívás hatására a függvény kifejezést alkalmazza a számra, majd a kérést küldő processz számára visszaküldi az eredményt. Ez a processz a kérést küldő Erlang *node* (amiből a hívást intéztük), és az azonosítóját a `call/2` függvény állítja elő a `self/0` függvény meghívásával. Egyébként a választ is ez a függvény küldi vissza.

9.4. programlista. Kérés a szerverhez - Erlang

```
> test_server:call(Pid, {fun(X) -> X + 2 end, 40}).
> 42
```

A `loop/0` funkciói a kérések során küldött paraméterek alapján a szerver elindítása után:

- A `{From, hello}` *tuple* paraméter küldése esetén - ahol a `From` változó tartalmazza a küldő processz azonosítóját - a válasz a *hello* atom lesz, amit a küldő azonnal vissza is kap.
- Amennyiben a kérésben a `{From, Fun, Params}` mintára illeszkedő adatot találunk, a `From` a küldő azonosítója lesz, a `Fun` a végrehajtásra kerülő függvény kifejezés, és a `Params` a paramétereként szolgáló adat. Az eredmény ebben az esetben is visszakerül a küldőhöz.
- A `{From, stop}` kérés azt eredményezi, hogy a szerver nem hívja meg önmagát, vagyis leáll, de előtte visszaküldi a *stopped* atomot, tájékoztatva ezzel a küldőt a fejleményekről (9.5. programlista).

Az ismétlés utolsó ága felelős a futás során felmerülő hibák kezeléséért. Ez a munkája abban merül ki, hogy a nem megfelelő, vagyis a minták egyikére sem illeszkedő kérések esetén értesíti a küldőt a hibáról, majd újra meghívja önmagát (9.6 programlista).

9.5. programlista. Szerver leállítása – Erlang

```
> test_server:call(Pid, stop).
> stopped
```

9.6. programlista. Hibás kérés kezelése - Erlang

```
> test_server:call(Pid, abc123).
> {error,abc123}
```

A `loop/0` függvényben a várakozást a `receive` vezérlő szerkezet végzi, az üzenetküldéseket pedig a `!` operátor, melynek a bal oldalán a címzett azonosítója, a jobb oldalán pedig az üzenetek tartalma található. Az üzenet kizárólag egy elemű lehet, ezért alkalmaztunk *tuple* adatszerkezetet a több adat becsomagolására.

Az egyes ágakban szereplő mintákat felfoghatjuk úgy, hogy ez a szerver alkalmazás protokollja, mely segítségével kommunikálhatunk vele. A szerver csak az ezekre a mintákra illeszkedő adatokat érti meg.

A szerver alkalmazás és a kliensek kódja ugyanabba a modulba kerül. Ez a technika az elosztott programok esetén nem ritka, mivel így alkalom adtán mind a kliens, mind a szervergépek el tudják látni a másik feladatait.

A kliens-szerver programot csak Erlang nyelven ismertettük, mivel az F# változat hosszú, és az elosztottságot megvalósító része nem is annyira funkcionális, mint azt várnánk, és a Clean nyelvet sem igazán az elosztott programok készítésére fejlesztették ki...

A program megírását és kipróbálását követően gondolkodhatunk azon is, hogy ezen az nyelven bármilyen bonyolult kliens-szerver alkalmazást, FTP szervert, vagy éppen *chat* programot megírhatunk néhány sor kóddal, és futtathatjuk ezeket a lehető legkisebb erőforrás igény mellett. Alapja lehet ez a program bonyolult, nagy erőforrás igényű számításokat végző programok elosztott futtatásának is, de kihasználhatjuk az erősebb számítógépek nyújtotta számítási kapacitást úgy is, ha az ismertetett módon "küldözgetjük" a függvényeket és az adatokat az erősebb gépeknek, és a kis teljesítményű társaikon csak a visszakapott eredményt dolgozzuk fel. Ahogy a fejezetek feldolgozása során láthattuk, a funkcionális nyelveket alkalmazhatjuk a programozás szinte minden ismert területén, legyen szó adatbázis kezelésről, szerverek írásáról, vagy egyszerű számításokat végző, felhasználói programok készítéséről.

Ezeknek a nyelveknek viszonylag nagy a kifejező ereje, és a különböző nyelvi elemek nagy alkotói szabadságot biztosítanak a programozók számára.

10. Funkcionális nyelvek a gyakorlatban

10.1. Programfejlesztés Erlangban - a fejlesztőeszköz beállításai

Az imperativ nyelveknél megszokott library modulokban, vagy az OO programok névtereibe rendezett, funkcionalitás ugyanúgy típus szerint összegyűjtött függvények halmazát jelenti, mint a funkcionális nyelvek moduljaiba rendezett függvények esetében.

Az Erlang nyelvben a függvényeket modulokba rendezzük, a függvények moduljait exportáljuk, hogy azok a modulon kívül is elérhetőek legyenek. Ez a technológia hasonlít az OO nyelvekben megismert osztályokra, és azok védelmi szintjeire, mint a private, public, protected. (Az Erlang nyelvben a nem exportált függvények lokálisak a modulra. Ha globális adatokat szeretnénk definiálni, akkor rekordokat készítünk.)

Mindezek mellett a funkcionális programozási nyelvek, különösen az Erlang számos olyan különleges programkonstrukciót is tartalmaz, amelyeket az OO és az imperatív nyelveknél nem találhatunk meg.

1. Lista kifejezések használata
2. Lusta vagy Mohó kifejezés kiértékelés
3. Farok rekurzió
4. Változók kötése (destruktív értékadás hiánya)
5. Ciklus típusú iterációk hiánya
6. Függvények hívási helyfüggetlensége
7. Mintaillesztés
8. Currying
9. Magasabb rendű függvények

A felsorolásban szereplő nyelvi elemek teszik a funkcionális nyelveket mássá, és ezektől a tulajdonságoktól válnak érdekessé, vagy különlegessé. A következő néhány oldalon szinte minden fejezethez találhatunk feladatokat, és az említett tulajdonságok szinte mindegyikét megismerhetjük a megoldások elkészítésén keresztül. A feladatok megoldásai csak azokon a pontokon találhatóak meg, ahol ez feltétlenül szükséges. A feladatok igazság szerint a megoldást is tartalmazzák, de csak szöveges formában (inkább csak elmesélik azt, hogy a megoldásban rejlő lehetőségeket az olvasó aknázza ki, és minél többet profitáljon belőle…).

Ezekre a leírásokra tekinthetünk úgy, mint a formális specifikációt megelőző szöveges leírásra, amely nagyobb programok tervezésénél elengedhetetlen eszköz. Amennyiben nem akarunk a fejlesztés során zsákutcába kerülni. Az azért van így, mert a kezdő programozónak meg kell tanulnia analizálni a felhasználóktól, vagy a program egyéb megrendelőjétől származó szöveges leírásokat, vagy, ahogy ez a legtöbb fejlesztésnél lenni szokott a programozók egymás közt folytatott megbeszéléseinek a dokumentációiból. Ezen módszer bevezetése miatt a feladatok szövege hosszú, de minden esetben kimerítő magyarázatot és segítséget nyújt a programok elkészítéséhez.

10.2. Programfejlesztés Erlangban - a fejlesztőeszköz beállításai

Ahhoz, hogy a most következő programokat el tudjuk készíteni, mindhárom nyelvhez telepítenünk, és konfigurálnunk kell a fejlesztőeszközöket. Az ad-hoc fejlesztések kisméretű programoknál jól működhetnek, még akkor is, ha tetszőleges szövegszerkesztő programmal megírjuk a kódot, majd egy parancssori fordítóval futtatjuk, és az operációs rendszer parancssorában futtatjuk a programokat. Jól átgondolt fejlesztéseknél viszont, ahol készül programterv, specifikáció, megvalósíthatósági tanulmány, és még számos fontos eleme a programnak, a jól, és jól használhatóra kialakított programozási környezet elengedhetetlen.

Clean programok írásához a nyelv rendelkezik a saját nyelvén készített fejlesztői eszközzel (IDE – Integrated Development Environment), amely számos kiegészítő, valamint debug funkciójával megkönnyítheti a programozó munkáját. z F# esetén is hasonló a helyzet, mivel az F# nyelv része a Visual Studio nevű igen komoly eszközben használható programozási nyelvek csoportjának, így az F# programozók igénybe vehetik az IDE összes magas szintű szolgáltatását. A Clean programok írásakor arra kell ügyelnünk, hogy a nyelv fejlesztő, szövegszerkesztő és futtató rendszeres sajátos módon működik. Mikor új projektbe kezdünk, elsőként a modul fájlt kell megnyitnunk, vagy létrehoznunk, és csak ezután kérhetünk hozzá egy korábban már elkészített, vagy egy teljesen új projektet. Ha nem így járunk el, a fordítás és a futtatás közben rengeteg hibaüzenetet fogunk kapni, és a programok emiatt nem futnak majd le. A sorrendre tehát ügyelnünk kell, de ezen kívül túl sok meglepetés nem fog érni bennünket.

A Visual Studióban, és a Clean IDE-ben a megírt programok futtatásához elég aktiválnunk a fordítás és a futtatás menüpontjait, és az eszköz automatikusan elmenti a programot, majd fordítja és futtatja, hibák esetén meg figyelmezteti a programozót, és megmutatja a azok helyét és típusát.

Az Erlang programozási nyelv használata esetén már egészen más a helyzet. Az Erlang programokat megírhatjuk tetszőleges szövegszerkesztő programmal, és a fordítás után nagyon sok lehetőségünk lesz a kész programok futtatására. Szövegszerkesztő alatt itt nem az MS Word szövegszerkesztőhöz hasonló szoftvereket értjük, és még a Wordpad típusú programokat sem használjuk a fejlesztésre, mivel ezek nem csupán a szöveget mentik az erre a célra kiválasztott fájlokba, hanem rendelkeznek saját formátummal, és a formátum leíró tegjeiket, valamint jeleiket mind a szövegbe mentik. E miatt a metódus miatt válnak problémássá, ha programfejlesztésről van szó. Egyébként a túl sok, inkább szövegszerkesztésre alkalmas funkció is csak hátráltatná a programozó munkáját.

Amennyiben ezzel az egyszerű módszerrel fejlesztjük az Erlang programokat, mindenképpen válasszunk valami egyszerűbb, könnyen kezelhető, és különleges formátumjelölőkkel nem rendelkező eszközt. A NotePad jó választás lehet. Az így elkészített programokat mentsük olyan helyre, ahol az Erlang fordító eléri azokat, vagy a fordítás során adjuk meg a teljes elérési utakat a fájlokig, és gondoskodjunk arról is, hogy a fordító rendelkezzen írási joggal arra a könyvtárra, ahova a (.beam) fájlokat létre kell hoznia. Az Erlang (.beam) kiterjesztésű állományokba menti a lefordított programszöveget. Fontos momentuma a fordításnak az is, hogy a fájlok karakterkódolása megfelelő legyen. Unix alapú operációs rendszeren latin1 kódolást, Windows operációs rendszeren pedig az ANSII kódolást használjunk, különben a fordító szintaktikai hibákat fog jelezni az amúgy teljesen helyesre elkészített programszövegben.

Ennél a módszernél sokkal hatékonyabb, ha választunk egy olyan szerkesztő programot, amely rendelkezik úgynevezett szkriptelési lehetőségekkel, vagyis minden egyes kiterjesztéshez (fájltípushoz), amely futtatható, be tudjuk állítani egy tetszőleges billentyűparancs hatására lefutó programokat. Ez lehetőséget biztosít arra, hogy pl.: Erlang programok esetén az egyik funkció billentyű lenyomására elinduljon az Erlang parancssori fordítója, és lefordítsa az éppen betöltött, és fókuszban lévő fájlt. Sajnos a legtöbb szerkesztő program egy menü, vagy gomb aktiválásakor csak egy parancsot képes végrehajtani, de ezt a problémát könnyedén áthidalhatjuk egy jól megirt szkripttel, vagy batch fájl használatával. Kézenfekvő tehát, hogy olyan eszközt válasszunk, ami jól konfigurálható, és sokoldalú. Egy nagyon jó választás lehet az Erlang programozók számára az Emacs nevű eszköz, amely Unix, és Windows alapú operációs rendszereke is futtatható, és számos olyan lehetőséggel rendelkezik, amik alapján bármely ismert programozási nyelvhez fejlesztőeszközként használhatjuk. Az Emacs

különböző módokat használ az egyes fájltípusok kezelése során, amely módokra meg tudjuk tanítani előre elkészített plugin-ok letöltésével és konfigurálásával, vagy saját, Elisp nyelven írt pluginok elkészítésével. Az Emacs-ben találhatunk C, C++, és akár Latex fájlok szerkesztéséhez és fordításához is beépülő elemeket, menüket, és egyéb eszközöket, és ami fontos a számunkra, az Erlang-hoz is.

Töltsük le a szoftver operációs rendszerünkön futtatható verzióját, majd telepítsük. Ha készen vagyunk, indítsuk el, ér azonnal láthatjuk, hogy az erl kiterjesztésű fájlokra egyáltalán nem reagál a program.

Ahhoz, hogy az Emacs erlang módba kerüljön, néhány apró beállítást meg kell tennünk. Az Emacs a beállításokat a (.emacs) nevű fájlban tárolja. Ha ilyennel nem rendelkezünk, akkor nyugodtan hozzuk létre vagy a felhasználói home könyvtárunkba, vagy Windows operációs rendszeren a C meghajtó gyökérkönyvtárába (a jogosultságokra ügyeljünk). Ha már rendelkezünk Emacs szerkesztővel, akkor a létező (.emacs) fájlt használjuk. Egyszerűen csak írjuk bele a következő néhány Lisp nyelven írt programsort:

((setq load-path (cons "C:/Program Files/erl5.7.5/lib/tools-2.6.5.1/emacs"

load-path))

(setq erlang-root-dir "C:/Program Files/erl5.7.5")

(setq exec-path (cons "C:/Program Files/erl5.7.5/bin" exec-path))

(require 'erlang-start)).

Az első setq rész a load-path változóban megmondja az Emacs-nek, hogy hol van az Erlangos Emacs plugin, a második az Erlang könyvtárait mutatja meg, a harmadik a bináris állományok és a futtatáshoz szükséges útvonalat definiálja. Ezen beállítások mindenképpen szükségesek a fejlesztőeszköz kialakításához.

Ha még nem rendelkezünk Erlang futtató rendszerrel, akkor töltsük le az erlang.org weboldalról, és telepítsük az operációs rendszerünkön telepítésre használható módszerrel. Unix-okon a (/usr/lib) könyvtár környékén keresgéljünk, Windows alatt a Program Files könyvtárban (erlangx.y.z) néven, ahol az x.y.z a verziót jelöli, ami fontos, mivel ez kell a beállításokhoz az Emacs-ben. Ezen a könyvtáron belül van egy olyan tool könyvtár, ahol megtaláljuk az emacs kiterjesztését az Erlangnak, vagyis azt a plugint, ami az Erlang programokhoz kell. A Lisp kódban erre kell lecserélni az ott szereplő elérési utakat.

Mikor készen vagyunk, mentsük el a (.emacs) fájlt, és nyissunk meg egy erl kiterjesztésű fájlt. Ha mindent jól beállítottunk, a menüsornak ki kell egészülnie egy új, erlang feliratú menüponttal, ahol számos, nagyon hasznos funkciót találhatunk, mint a syntax highlight beállítása akár több szinten is, vagy a skelenton-ok beszúrása. Ez a funkció előre elkészített, de a lehető leg általánosabban paraméterezett vezérlő szerkezeteket szúr be a kódba segítséget adva ezzel a kezdő, és a haladó programozóknak is. A kezdők esetében nem kell gondolkozni a nyelvi elemek szintaxisán, haladóknál meg a sok gépeléstől kímél meg bennünket a menü. Találunk itt elágazást, egyszerű utasítások kódját, de komplett kliens-szerver alkalmazást is, amit a beszúrást követően kedvünkre alakíthatunk.

Számunkra mégsem ez a legfontosabb pontja azt új menünek. Találunk a beszúrások alatt egy (start a new shell), és egy (view shell) feliratút (a run menüpontból nyílnak alapértelmezés szerint). Elsőként aktiváljuk az elsőt. Ekkor a szerkesztő két részre osztja fel az ablakot. Az ablakokat itt egyébként buffer-nek hívjuk, és külön menüt találhatunk ezek kezelésére. A felső buffer-be kerül a forráskódunk (ha megírtuk…), az alsóba a parancssor, ami a Linux rendszerek parancssori módjára emlékeztethet bennünket. Ide írhatjuk a fordításhoz szükséges utasításokat (c(erlangfile)), ahol az erlangfile az általunk készített fájl neve, de az (.erl) kiterjesztést el kell hagynunk. Mivel az Emacs ismeri az elérési utakat a fájlokhoz, ezért ezekkel nem is kel foglalkoznunk a fordítás és futtatás során. Ahogy lefordítottuk a szintaktikailag helyes kódot, a fordító ezt jelzi számunkra a parancssorban: ({ok, erlangfile}). Ezután már futtathatjuk is programot a modul nevének, a kettőspontnak és a függvény paraméterekkel ellátott hívásának a begépelésével (pl.: erlfile:f(10).), ahol az f az erlfile nevű modulban található (exportált) egy paraméteres függvény. Az exportálásra azért van szükség, hogy a függvényeket a modulokon kívül is lehessen hívni.

A program lefutása jelzi számunkra, hogy jó munkát végeztünk, és ezután, ha Erlang fájlokkal nyitjuk meg az Emacs eszközt, mindig az Erlang fejlesztőeszközként fog működni.

Az Emacs használatának elsajátítása az első alkalmakkor okozhat némi fejtörést, mivel a megszokott Ctrl X/C/V billentyűparancsok itt nem működnek, vagy egészen más hatást váltanak ki, mint amit elvárunk tőlük. Emacs-

ben az Alt X, vagy a Ctrl és egyéb billentyűk egyszerre történő lenyomása aktiválja a funkciókat. Például a fájlok megnyitása Ctrl + X + F billentyűk lenyomásával eszközölhető. A másoláshoz viszont elég csak kijelölni a másolandó szöveget, majd a beillesztéshez a Ctrl + Y billentyűpárt használhatjuk. A kivágáshoz nincs külön funkció, azt menüből használhatjuk. Az Erlang programok azonnali fordításához a Ctrl + C + K billentyűjét használjuk úgy, hogy a Ctrl és az X egyszerre legyen lenyomva, majd ha ezeket felengedtük, akkor nyomjuk le a B gombot a billentyűzeten. Ekkor ugyanaz történik, mintha az erlang menüben aktiválnánk a shell-t és begépelnénk a (c(modulneve).) szöveget, a végére egy pontot tennénk, majd lenyomnánk az entert. Ha ezt a módszert túlságosan bonyolultnak találjuk, használjuk a menüket, vagy keressünk egy hasonló funkciókkal rendelkező, de számunkra könnyebben kezelhető szoftvert. A bufferek közti váltás a Ctrl + X + B.

Sok programozó használja az Eclipse nevű, és nagyon sokoldalú IDE-t, ami alkalmas többek között Java, C, C++, és Erlang programok fejlesztésére az adott nyelvhez készült pluginok alkalmazása mellett. Az Eclipsehez hasonló a Netbeans, ami szintén megfelelő Erlang programozók számára, és ez az utóbbi két szoftver használatát tekintve inkább a megszokott (Windows operációs rendszeren is standard) módon működtethető (működik a Ctrl X/C/V).

Igazság szerint teljesen mindegy, hogy milyen eszközt választunk a fejlesztéshez, a lényeg az, hogy minél hatékonyabban, és gyorsabban tudjunk dolgozni vele, és a lehető legkevesebb emberi, időbeli, és egyéb erőforrást keljen elhasználni még a nagyobb projektek és fejlesztések céljainak eléréséhez is.

Az itt leírt konfigurációs és telepítést végző műveletek elvégzésének a megkönnyítésére a bemutatott lépések közül néhányat videóra is vettünk, hogy a kedves olvasónak segítséget nyújtsunk a saját eszközkészletének a kialakításában. Ha még ezek után sem sikerül eredményhez jutnia, akkor a legjobb módszer az Interneten található leírásokban, vagy fórumokon megkeresni a megoldást. Szoftverfejlesztők, és internetes közösségek tagjai általában nyitottak, barátságosak, és hajlandóak segítséget nyújtani kezdők számár is. Bátran kérdezzünk!

10.3. Az első feladat elkészítése

Amennyiben az Emacs fejlesztő eszköz működőképes, vágjunk is bele egy egyszerű modul megszerkesztésébe, fordításába és futtatásába. Ha kétségeink vannak a felől, hogy elsőre is menni fog a dolog, megnézhetjük az erről a mozzanatról készült videó mellékletet, amelyben minden lépés dokumentálva van.

Az első feladat, hogy indítsuk el az Emacs-ot. Ezt megtehetjük a program bin könyvtárában lévő emacs.exe, vagy runemacs.exe fájlok valamelyikének az aktiválásával. A későbbi munkára való tekintettel készítsünk az indító egy parancsikont (shortcut), amelyet elhelyezhetünk az asztalon. Ha megoldottuk a problémát, az Emacs menüjével, vagy a Ctrl + X + F billentyűkombinációval nyissunk meg egy új fájlt, és mentsük el (.erl) kiterjesztéssel (menü, vagy Ctrl + X, Ctrl + S).

A fájl elejére írjuk oda a (-module()) előtagot, és bele a fájl nevét. Fontos momentum, hogy a fájl nevének meg kell egyeznie a modul nevével, különben a fordításnál hibaüzenetet kapunk. Egyébként, ha nem írunk oda semmit, az Emacs megkérdezi, hogy szeretnénk-e, ha ő odaírná a nevet. Mondjunk igent. Most következzen az exportlista a (-export([])) formulával. Ebbe kell írnunk az exportálásra szánt függvényeket, amelyeket a modulon kívülről is szeretnénk elérhetővé tenni. Ebbe a listába a függvények nevei és aritásuk (paraméterek száma) kerül az alábbi formában: (név/aritás, vagy f/1), ahol f a függvény neve, az 1 pedig a paramétereinek a száma. Mivel az export egy listát tartalmaz, ne felejtsük el a listazárójeleket.

Az export listát egyelőre hagyjuk üresen, és koncentráljunk a függvényeink elkészítésére. Az első függvény, amit megírunk, az f nevet kapja, és legyen egy darab paramétere, amit a függvény azonnal vissza is ad visszatérési értékként: (f(A) -> A.). Ügyeljünk arra is, hogy a függvények neve atom, vagyis nem kezdődhet nagybetűvel, a változók viszont nagybetűvel kezdődnek, mivel az Erlang nyelv gyengén típusos, és e szabály alapján azonosítjuk a címkéket, függvény, rekord, és rekord mező neveket, és a változókat. A függvény nevét a paraméter listája követi, amely zárójeleit akkor is ki kell tennünk (a C alapú nyelvekben megszokott formában), ha nincs formális paraméterlista. A változó neve tehát (A). A paraméter listát a (->) kettős jel követi, ami a függvény törzsét vezeti be. A törzs kizárólag a formális paraméter listában megadott változó azonosítóját tartalmazza, és mivel ez a függvény utolsó utasítása, ez lesz a visszatérési értéke is. A függvény törzsét ponttal zárjuk.

Ha készen vagyunk, akkor nyomjuk meg a Ctrl + C, Ctrl + K billentyűket (vagy az erlang menüben keressük meg start a new shell menüt, és a shellbe gépeljük be a (c(moduleneve)) függvényhívást). Ekkor a rendszerünk lefordítja a modult. Amennyiben hibát talál, ezt jelezni fogja a parancssorban, ha hibátlan a forrásszöveg, akkor a következő rendezett pár jelenik meg: ({ok, modulneve}).

Most már nyugodtan futtathatjuk a programot. Gépeljük a parancssorba a következő függvényhívást: (modulneve:f(10).), ahol a modulneve a lefordított modulunk neve, a kettőspont (:) után pedig a függvény neve következik, majd zárójelek között a formális paraméter listára teljesen illeszkedő aktuális paraméter lista. A sort a pont zárja, majd entert kell ütnünk, és kiíródik az eredmény.

Most térjünk vissza a forráskódhoz. Ha nincs fókuszban, akkor keressük meg a buffer listában (menü), vagy a Ctrl + X + B billentyűkombinációval és a nyilakkal. Készítsünk a modulba egy másik függvényt is, amely meghívja az elsőt, és hozzáad a visszatérési értékéhez egy konstans értéket. A függvényt az előzőhöz hasonlóan szintén exportálnunk kell. Ehhez csak annyit kell tennünk, hogy az export listát kiegészítjük az új függvény nevéből, és aritásából képzett, perjellel (/) elválasztott formájával. Készíthetünk új export listát is, de erre jelenleg semmi szükségünk nincs. Az új függvényt nevezzük el g-re. A g-nek is van egy paramétere, és a törzsében tartalmazza az (f(B) + B) kifejezést. Ahhoz, hogy a függvényen belül elhelyezett több utasítás írását is ki tudjuk próbálni, az f/1 hívásának a visszatérési értékét kössük egy tetszőleges nevű változóba, majd ennek a változónak a tartalmát adjuk hozzá a g/1 aktuális paraméterében kapott értékhez. Ekkor a függvénytörzs a következő képen alakul: (C = f(B), F + B). A ”C” változóba tehát kötjük az f/1 visszatérési értékét, amelyet az a g/1 paraméteréből állít elő, majd a kapott eredményt hozzáadjuk a g/1 paraméterében kapott értékhez. Mivel az összeadás a függvény utolsó kifejezése, ezért ennek az eredménye lesz a végeredmény. Most, hogy megvagyunk a program módosításával, ismét le kell fordítanunk a forrásszöveget, hogy a változtatás a (.beam) fájlba is belekerüljön (pontosan úgy van ez, mint a más programozási nyelvek forráskódjából fordított (.exe) állományoknál). Az új függvény futtatása: (modulneve:g(10).), amely hívás eredménye a húsz, lesz az összeadások miatt.

M.1. programlista. Az első teszt module

```
f()->
    ...
    f().
g1() ->
    g1().
```

Az itt bemutatott módszerrel bármilyen egyszerű, vagy éppen bonyolult Erlang programot elkészíthetünk. A lépések hogyanja, és sorrendje ugyan ez, annyi különbséggel, hogy a programok forrásszövege, modul, és függvénynevei változnak. Ha programozás közben elakadunk, akkor használjuk az Erlang futtatókörnyezethez mellékelt help fájlokat (a manuált megtaláljuk az Erlag könyvtárainak egyikében, általában verzióról függően egy (doc) nevű könyvtárban), vagy navigáljunk el az Erlang hivatalos weboldalára, ahol teljes körű referenciát kaphatunk a nyelvről.

10.4. Média alapú segítség a megoldáshoz

A jegyzetben, és különösen ebben a fejezetben található feladatok megoldásához, és a különböző telepítési és konfigurációs eljárások kivitelezéséhez készültek videó, és kép mellékletek.

A videók tartalma: a mellékelt videó anyag bemutatja a jegyzetben használt eszközök letöltését, az Erlang futtató környezet telepítését (video/video1.avi, és video4.avi), az Emacs telepítését és konfigurálást (video/video3.avi és video4.avi), a Clen nyelv programjainak futtatását (video/video2.avi)

A mellékletben található képek a jegyzet fejezeteiben, és a mellékelt feladatsorban található feladatok és példaprogramok elkészítésének módját mutatják be úgy, hogy az első kép minden esetben a forráskódot mutatja a szövegszerkesztőben, a második pedig az adott program kimenetét a futtatást közben, vagy azt követően, hogy a kedves olvasó a problémamegoldás minden pontján segítséget kaphasson. Ezek a képek mellékletként kaptak helyet a jegyzetben, így egymás után, és nem mindig sorrendben közöljük őket.

Képek

10.5. Gyakorló feladatok

1. feladat: Készítsünk Erlang programot, amely kiírja a képernyőre a Hello Világ szöveget. A megoldáshoz használjuk az io:format függvényt, amelynek a formátum sztringjébe helyezzük el a következő elemeket: ”~s~n”. A~s a sztring formátuma, a ~n pedig a sortörés jele. Ahogy azt korábban már láthattuk, az io:format függvény listát vár paraméterként, és a listában megkapott elemeket írja ki a képernyőre, de csak azokat az

elemeket, amelyekhez a formátum sztringben tartozik megfelelő formátum jelölés. Rossz paraméterszám esetén már fordításkor hibát kapunk. Az IO modul még számos más függvényt is tartalmaz az input és az output kezelésére. Erlang programoknál egyébként a konzolos input nem jellemző, mivel az Erlang nem GUI (grafikus felhasználói felület) készítésére tervezett nyelv. Az alacsony szintű Erlang rutinok szinte minden esetben rendelkeznek interfész függvényekkel, amelyek meghívása egy más nyelven íródott, grafikus felhasználói felület feladata, és a két nyelv közti helyes adatcsere, és adatkonverzió megvalósítására szokás használni az IO modul szolgáltatásait.

A FORMAT függvény (szándékosan nagybetűs, hogy elkülönüljön a szöveg többi részétől, de természetesen a programokban a kisbetűs írásmód használandó) számos olyan szolgáltatással rendelkezik, amelyek az adatok szöveggé konvertálására alkalmazhatóak. Átalakíthatjuk a listákat, rendezett n-eseket, atomokat és számokat a megfelelő formátumú, szöveges magyarázattal ellátott formára. A feladat megoldása során kétféleképpen járhatunk el. Az első lehetőség, hogy a kiíró utasítást nem látjuk el formátum sztringgel, mivel a ”Hello Világ” karaktersorozat maga is szöveg típus, és a format alkalmas a kiírására formázás nélkül is. A másik lehetőség, ha a szöveget egy változóba kötjük, majd a formátumban a korábban már bemutatott módon, a ~s formátummal formázzuk, és elhelyezzük második paraméterként a kiírást végző függvényben. Bármelyik lehetőséget használjuk, a sor végén érdemes elhelyezni a ~n formázót, hogy a sortörés a következő kiírást már új sorba tudja kezdeni. A sorvége elhagyásával a kiírások sajnos egy sorban kerülnek a képernyőre.

Amennyiben a kiírás során nem sztring, hanem atom típust használunk, csak a formátumot kell a megfelelő módon átalakítani. Ennél a típusnál nem használható a format egy paraméteres változata, mivel az csak szöveg típus esetén működik. Ha a kiírás során nem tudjuk eldönteni a típust, akkor sem kell más kiíró utasítás után nézni. Ez az eset akkor fordulhat elő, ha a függvényben menet közben derül ki a paraméter típusa. Mivel az Erlang nyelv gyengén típusos, így sokszor az adott függvény paramétere csak a meghíváskor, a formális paraméterlistába való aktuális paraméterek behelyettesítésekor derül csak ki a konkrét típus. Ha ilyen a paraméterezése a függvénynek, és a paramétereket, vagy a velük végzett műveletek eredményét ki alkarjuk íratni, használhatjuk a format első paramétereként a ~w formátumot, amely minden egyes típusnál működőképes, de a szöveg típus szinte olvashatatlan lesz. Ez azért van, mert a szöveg típus valójában egy lista, amely a szöveg karaktereinek a kódját tartalmazza, és a kiírás során a függvény ezt az adatot valódi listának értelmezi, így a kiírásnál számok listáját kapjuk. Amennyiben szöveges adatok karakterré alakítását szeretnénk elvégezni ez a megoldás éppen jó is lesz.

Figyeljünk arra is, hogy az io:format a függvény egyetlen, vagy a legutolsó paramétere lesz, ilyenkor a függvény mellékhatásos, vagyis a kiírás a mellékhatás, a visszatérési értéke pedig az ok atom lesz, mivel a format függvénynek meg ez a visszatérési értéke. Az Erlang nyelvű változat mintájára készítsük el a Clean és az F# nyelvű változatokat is. Természetesen az IO:format nyelvi megfelelőit is meg kell találnunk a feladatok megoldásához. Ezeknél a megoldásoknál tehát más kiíró utasítást kell keresnünk más formázási lehetőségekkel. Az F# esetén ez nem lesz túl nehéz azok számára, akik a C# nyelvet ismerik, mivel a konzolos képernyő F# programokban is hasonlóan működik, mint az OO változuatoknál. A Clean változatban a nyelv, mivel saját fejlesztőeszközzel rendelkezik, számos lehetőséget kínál az adatok megjelenítésére, de annyit bizonyosan, mint az előző kettő.

2. feladat: Az alábbi néhány pontban olyan feladatok kaptak helyet, amelyek a konzolos képernyőre való kiíratást gyakoroltatják, de bevezetik a programozót az összetett adatok kezelésébe és kiírásába a konzolos képernyőn. A feladat részeit egy modulban érdemes implementálni, minden feladathoz egy, vagy több függvény készítésével. Az első feladat az, hogy készítsünk programot, mely egy tetszőleges sorozatról eldönti, hogy melyik a legkisebb, és a második legkisebb eleme. A megoldáshoz használjunk lista adatszerkezetet, és akkor a bemenő adatok előállításáról nem kell külön gondoskodnunk. Ha szépen szeretnénk dolgozni, akkor készítsünk egy olyan függvényt, amely egy listát ad vissza eredményül, és ezt a listát adjuk paraméterként a modul további függvényeinek. Az így elkészített listát bármikor lecserélhetjük a függvény törzsének módosításával. Amennyiben nem szeretnénk ilyen bonyolult megoldást alkalmazni, a modul elkészítése után a futtatáskor egy tetszőleges nevű változóba köthetjük a teszteléshez használt listát. A konkrét feladat tehát az, hogy a paraméterként kapott listát be kell járni egy rekurzív függvény segítségével, majd az aktuálisan legkisebbnek vélt elemet kötni kell egy erre a célra készített változóba és tovább kell adni a következő hívásnak.

Amennyiben a rekurzió futása során az aktuálisan átadott legkisebb elemnél kisebbet találunk, az eddig legkisebbnek vélt elemet a második legkisebb kategóriába kell sorolnunk. Ez a gyakorlatban azt jelenti, hogy ezt az elemet is továbbadjuk a következő hívásnak. Ezt a műveletsort a lista minden eleménél meg kell ismételnünk. Mikor a mintaillesztéssel bejárt lista kiürül, vagyis az a függvény ág kerül sorra, amelyik formális paraméterlistája az üres listát tartalmazza, a legkisebb, és a második legkisebb elemet tároló paramétere változókat ki kell írni, vagy egyszerűen csak vissza kell adni a függvény visszatérési értékeként.

Készítsünk programot, mely kiírja a szorzótáblát a képernyőre. Nyilvánvalóan a képernyő itt az Erlang konzolos ablaka, ahol a formázást csak az io:format függvényben elkövetett trükkökkel tudjuk megoldani. Ahhoz, hogy megkapjuk a szorzótáblát, a számok egy olyan szekvenciáját kell előállítanunk, ahol az aktuális elem két szám szorzata. Az első szorzó tényező a képzeletbeli sorindex, amit minden függvény újrahíváskor, vagyis ”ciklikus” lefutáskor eggyel növelünk, és a másik tényező az oszlopindex, amelyet egy sorindex létrejötte mellett egytől tízig növelni kell. Tehát az első elem az 1*1, a második elem a 1*2, majd az 1*3, és így tovább. Mikor a második tényező eléri a tízet, az elsőt eggyel növelni kell, majd jöhet a következő sorozat. Ezt a műveletsort imperatív nyelvekben nem nehéz megvalósítani, mivel csak két egymásba ágyazott for ciklusra van szükség a megírásához, de funkcionális nyelvek esetében a ciklusok hiánya ezt a megoldást teljesen kizárja.

Az ismétlések megvalósítására az egyetlen eszköz a rekurzió (amennyiben a listagenerátorokat nem vesszük számításba). A rekurzív megoldást úgy kell kialakítanunk, hogy a rekurzió több ágon fusson, vagy az egyes sorokat/oszlopokat más-más függvénnyel kell megvalósítanunk (kezdő programozóként ne ezt a megoldást válasszuk). A listagenerátoros megoldás valamivel egyszerűbb, mert ennél két lista adatszerkezet egymásba ágyazásával megoldható a feladat. Míg az első (külső) listagenerátor ad egy elemet, a belső egytől tízig generálja az elemeket. Igazság szerint a két megoldás, vagyis a függvényhívásos, és a listagenerátoros változat összeolvasztásával kapjuk a legjobb megoldást. Ekkor a listagenerátorban egy olyan függvényt alkalmazunk, amely rekurzívan elszámol egytől tízig, majd egy listában, vagy egyesével visszaadja az elemeket, amelyeket a hívó listagenerátor felhasznál a szorzatok előállításához. Ahhoz, hogy a kimeneti képernyőn ténylegesen a szorzótáblára jellemző mátrixos formát kapjuk, minden sorozat (1*1, 1*2, … n*n+1) végére egy sortörést kell elhelyeznünk, ami feltételezi az IO:FORMAT függvény használatát az adatok megjelenítéséhez.

Készítsünk a modulba egy függvényt, amely paraméterként kap egy egész számot (integer), majd kiírja a hét azonos sorszámú napját a képernyőre. Az 1-es érték jelenti a hétfőt, a 2-es a keddet, a 7-es a vasárnapot. A függvény megírásához a case vezérlő szerkezetre lesz szükségünk, ami egy adott kifejezés eredményétől függően több ágra tudja szétválasztani a függvényt. A megoldás egyszerű. A függvény aktuális paramétere egy és hét közé eső szám (nevezzük N-nek, ahol N = (1..7), és N típusa integer). A számot a case szerkezet kifejezéseként használva hét és még egy ágat kell készítenünk. Az első hét ág az adott sorszámhoz tartozó nap nevét írja ki a format valamely változatát felhasználva. Az utolsó ágra azért van szükség, mert a függvény kaphat N-től különböző értéket is, és ebben az esetben a hibát jelezni kell a felhasználó, vagyis a felhasználói felület irányába. A függvényt még általánosabbá, és hibatűrőbbé tehetjük, ha a fejlécében egy guard utasítással meghatározzuk a típust (when is_integer(N)), és az intervallumot (and N > 0, N< 8). Másik megoldás lehet a hibakezelésre a kivételkezelő blokk használata, amely komoly védelmet jelent bármilyen jellegű hiba keletkezése esetére.

A modul soron következő függvényének egy tetszőleges sorozatról kell tudnia eldönteni azt, hogy annak melyik a legkisebb, vagy akár a legnagyobb eleme, majd az eredményt ki kell írja a konzol képernyőre. A megoldás a korábbi feladatok ismeretében viszonylag egyszerű. A feladat a klasszikus minimum, és maximum kiválasztástól annyiban különbözik, hogy itt a függvény paraméterként kapja meg azt, hogy a min, vagy a max elemet kell visszaadnia. Így tehát az első paramétere a min, vagy a max atom, amely címkeként jelzi a függvény számára az elvégzendő műveletet (igazség szerint csak a feltételben alkalmazott reláció irányát). A függvény megvalósítására használhatunk case vezérlő szerkezetet, vagy egy több ággal rendelkező függvényt, ahol a mintaillesztés, és az overload típusú függvényhívás tulajdonságait kihasználva a relációt, vagyis a műveletet egyszerűen váltogathatjuk. A függvény első ága egy mintaillesztést tartalmaz, amely a listát egy első elemre, és a további elemeire bontja szét. Az aktuális elemről eldöntjük, hogy az kisebb, vagy nagyobb az előzőleg tárolt elemtől (kezdetben az első elemet is összehasonlítjuk saját magával, de ezt a hatékonysági problémát könnyedén kiküszöbölhetjük, ha az első elemet nem vizsgáljuk, hanem azt feltételezzük, hogy ez az elem a legkisebb, és az összehasonlítást a második elemtől kezdjük). Amennyiben az aktuális legkisebb elemnél kisebbet találunk, cseréljük azt a legkisebbre (átadjuk a következő rekurzív hívásnak).

A függvény első paramétere alapján a legnagyobb elem keresésénél is ugyanígy kell eljárnunk. A függvény második ágának első paramétere mindegy, hogy mi, mivel itt már nem használjuk az értékét (Ha a kiírásnál jelezni szeretnénk, hogy minimumot, vagy maximumot kerestünk, ez nem teljesen igaz). Itt tehát használhatjuk az aláhúzás jelet, ami a mindegy megfelelője az Erlang programokban. A második paraméter egy üres lista, ami azt hivatott jelezni a függvényhívások során, hogy a paraméterként megadott lista minden első elemét leválasztottuk, és így üres listához jutottunk. Ekkor meg lehet állni, és közölni kell az eredményt a felhasználóval. ez eredmény lehet a min, vagy a max elem abban az esetben, ha egy általános működésű, mellékhatásoktól mentes függvényt készítettünk, de lehet az utolsó utasítás az eredményének a kiírása is (ekkor nem valódi függvényről, hanem az imperatív nyelveknél használt eljárásról beszélhetünk).

A feladatban felsorolt függvények mindegyikét ki kell próbálni, és ehhez természetesen a modult le kell fordítani, és futtatni az Erlang parancssorban. Ahol szükséges, használjunk konstans paramétereket, vagy készítsük el a megfelelő adatokat a parancssorban, mivel a konzolban az adatok bekérése nem túl egyszerű feladat.

3.. feladat: Írjunk olyan programot, amely addig állít elő egész számokat, amíg azok összege meg nem haladja a 100-at. A beolvasás végén írjuk ki azt, hogy a bekért számok közül hány volt páros, és hány volt páratlan. A feladat megoldásához lista generátort, vagy olyan adattárat kell használnunk, amely szolgáltatja az egész számokat. Mint minden programozási nyelvben, az Erlang-ban is lehetőségünk van véletlen számok előállítására a rand könyvtári modul segítségével. A random értéket tárolhatjuk változóban, de azonnal hozzá is adhatjuk egy akkumulátornak használt változóhoz, amelyet a rekurzív hívások során minden lépésben továbbadunk. A feladat megoldása rekurzióval valósítható meg a leghatékonyabban. A rekurzív függvénynek két ága lesz. Az első ág minden száznál kisebb értékre lefut, és meghívja önmagát úgy, hogy az előzőleg kapott összeghez (kezdetben nulla értékű az aktuális paramétere) hozzáadja az éppen generált véletlen számot, majd az így kapott értékkel hívja meg sajátmagát.

A második ág esetében kissé bonyolultabb a helyzet, mert a mintaillesztésben a 100-as értéket kellene használnunk, és ez akkor lenne igaz, ha az összeg nem mehetne 100 fölé. Ebben az esetben sajnos végtelen rekurzióhoz juthatnánk (funkcionális, farok rekurzív függvényeknél ez majdnem lehetséges). Igaz, hogy a megoldás nem triviális, de nem is lehetetlen. Nem kell mindenképpen integer típusú paramétert használnunk. Jó megoldás az is, ha a véletlen szám generálást és az összegzést végző ág 100-nál kisebb számok esetén meghívja magát azzal az atommal, hogy 'kisebb', egyébként meg a 'nagyobb' atomot adja tovább a következő hívásnak, ami a második ág lefutását eredményezi, így egy egyszerű feltételes elágazás bevezetésével megoldhatóvá válik a mintaillesztés helyes működése, és kiküszöbölhető a százas érték átlépésével járó összes kellemetlenség. A mintaillesztés mellett használhatjuk a case vezérlő szerkezetet is arra, hogy a rekurziót megállítsuk, és az eredményt kiírjuk a képernyőre. Ennek a megoldásnak az elkészítését a kedves olvasóra bízzuk.

4. feladat: Készítsük el az ismert n faktoriális értékét kiszámító feladatot Erlang, Clean és F# nyelven. A program kimenete mindhárom esetben a konzolos képernyőn jelenjen meg, és ne tartalmazzon egyéb kiírást, szöveget, csak az adott, a feladat alapján kiszámított értéket. A faktoriális kiszámításához sorra kell vennünk a számokat, és összeszoroznunk őket. A feladat egyértelműen rekurzív, és ha ebben kételkednénk, csak vizsgáljuk meg a faktoriális függvény matematikában használt definícióját, ami szintén rekurzív. Nyilvánvalóan létezik a problémának iteratív megoldása is, de ez egy funkcionális nyelv esetében nem jöhet számításba az iterációs nyelvi primitívek teljes hiánya miatt. A megoldás egy két ággal rendelkező függvény, ahol az ágak abban különböznek, hogy míg az első ág nulla érték esetén fut, a második tetszőleges, nullától különböző N értékre úgy, hogy megszorozza azt az eggyel csökkentett N értékével (n-1). Igazság szerint az N * fakt(N-1) kifejezést értékeli ki, ahol a fakt(N - 1) a függvény önmagára vonatkozó rekurzív hívása. Ahogy az érték minden újrahíváskor eggyel csökken, hamarosan elérjük a nulla aktuális paramétert, mikor az első ág fog meghívódni, és megkapjuk az eredményt.

A probléma megoldható ezzel a módszerrel, aminek megvan az a hibája, hogy a rekurzív hívás kifejezésben szerepel, így a függvény nem farok-rekurzív, ami azért baj, mert a futása során veremre van szüksége az éppen kiszámolt adatok tárolására. A jó megoldás az, ha a kifejezést (N * …) valahogyan bevisszük a függvény paraméterlistájába, és egy újabb ággal, vagy egy másik függvénnyel kiegészítjük a programot.

5. feladat: Tételezzük fel, hogy létezik teljesen szeparált, L-egység hosszúságú csatorna, és mindkét végénél egy-egy egér. Egy indító jelre az egyik egér U, a másik V sebességgel kezd rohanni a csatorna ellenkező vége felé. Amikor odaérnek, visszafordulnak és újra egymással szemben haladnak (faltól falig rohangálnak, amennyiben nem szeretjük az egereket, a futtatásra alkalmazhatunk rovarokat, vagy kutyákat is, de ekkor ne csőben futtassuk őket, mert beszorulhatnak). A futtatott lények három módon találkozhatnak, néha szemből, néha a gyorsabb utoléri a lassabbat, néha pedig egyszerre érnek egy falhoz. Készítsünk olyan programot, ami tetszőleges (konstansként megadott) adatok mellett kiírja, hogy az adott időtartam alatt az állatok hányszor találkoztak. A programot egy modul függvényeként implementáljuk, hogy ki is lehessen próbálni. A megoldás egyik lehetséges módja, ha listagenerátorokat használunk a futás szimulálására, de egyszerű kifejezésként is megírhatjuk a függvény törzsét. A találkozások pillanatait, vagyis azokat a konstansokat, amelyek elérésekor "találkozás következik be" szintén egy listában tárolhatjuk, amelyet a függvény akkor ad vissza, ha a képzeletbeli pálya hossza, és az idő intervallum ezt lehetővé teszi. Ha ügyesek vagyunk, akkor a listagenerátort kiválthatjuk egy rekurzív függvény két ágával is. a két ágra azért van szükség, hogy a rekurzió megálljon, és eredményt is kapjunk.

6. feladat: Készítsünk programot, mely tetszőleges, tízes számrendszerbeli, egész számot átvált kettes számrendszerbe. Az átváltandó számot a billentyűzetről olvassuk be, majd az átváltás eredményét ugyanide írjuk ki. Az átváltás a legkönnyebben úgy végezhetjük el, ha az átváltandó számot osztjuk a számrendszer alapszámával, vagyis a kettővel.

Az osztás maradéka a kettes számrendszerbeli szám lesz, az egészrészt pedig tovább kell osztanunk mindaddig, amíg el nem érjük a nullát. A programot (sr) nevű modulba készítsük el, és az (atvalt) nevű függvényét exportáljuk, hogy a fordítást követően meg lehessen hívni. Az atvalt-nak két paramétert kell adnunk. Az első a számrendszer alapszáma, a második az átváltandó szám. Az átváltásokat rekurzív hívások sorozatával oldhatjuk meg úgy, hogy minden egyes hívás során képezzük az osztás maradékát és az egészrészét. A maradékot eltároljuk egy erre a célra készített listában, vagy szöveggé alakítva konkatenáljuk egy sztringhez. Abban az esetben, ha nagyobb számrendszerekbe váltunk át, ügyelni kell arra is, hogy a számokat ezeknél már betűk helyettesítik. A megfelelő szám-karakter páros konvertálásához használjunk case vezérlő szerkezetben elhelyezett konverziós utasításokat, vagy készítsünk egy olyan függvényt, amelyet az átváltások elvégzésére szintén a modulban implementálunk. A rekurziónak akkor kell megállnia, ha az osztások eredményeképpen a nulla értéket elérjük az átvitt egészrész esetén. Ahol megálltunk, nincs más feladatunk, mint visszaadni, vagy kiírni az eredményt.

7. feladat: Készítsünk Erlang modult, amely néhány ismert matematikai művelet mellet a min, max, sum, avg, műveleteket implementálja. A műveletek, ahol ez lehetséges működjenek rendezett n-esre, listára és két változóra is. Az alábbi példa megmutatja a sum függvény egy lehetséges, több ággal rendelkező változatát.

M.2. programlista. A sum függvény

```
sum({A, B}) ->
        A + B;
sum([A| B]) ->
        sum(A, B);
sum(Acc, [A| B]) ->
        sum(Acc + A, B);
sum(Acc, []) ->
        Acc.
```

A sum/1 függvény, ahol az 1 a függvény paramétereinek a száma (aritása) egy. Ez azért van így, hogy a listákat kezelő, és a rendezett n-esre használható ágainak a paraméterszáma ne különbözzön. Az azonos nevű, de más paraméterszámú függvények nem egy függvény ágainak számítanak. A sum első ága egy rendezett n-esben, kap két számot, és eredményként visszaadja azok összegét. Ebben a példában láthatjuk, hogy a több ággal rendelkező függvényeket nem csak rekurzív hívások esetén használjuk, hanem a ”hagyományos” overload működéshez is. A második ág egy mintaillesztést tartalmaz formális paraméterként, amely minta az aktuális paraméterként érkező listát egy első elemre, és a lista maradékára bontja szét, ahol az első elem skalár típus, a második viszont lista, így az illeszthető továbbra is aktuális paraméterként a függvény valamely ágára. Ez a második ág egy ugyanolyan nevű, de két paraméterrel rendelkező függvényt hív meg a törzsében. Ennek a függvénynek is sum a neve, de az aritása kettő, így ő egy másik függvény, és az export listában is külön kell szerepeltetni abban az esetben, ha a modulján kívülről is el szeretnénk érni. A függvény megkapja az előzőleg szétbontott lista első elemét, és a maradék listát, amit szintén egy mintaillesztéssel további elemekre bont, de közben meghívja magát az eredeti első elem, és az általa leválasztott első elem összegével. Mikor a lista kiürül, vagyis elfogynak az összegzésre szánt elemek, a sum/2 második ága hívódik meg, ami visszaadja az eredményt.

A sum függvény mintájára megírhatjuk az avg függvényt is, ami az összesített elemek átlagát adja vissza, de felhasználhatjuk a sum/1-et az összegzés elvégzésére, majd az avg-ben egy osztást elvégezve megkapjuk az eredményt. Az osztáshoz viszont szükségünk van a lista darabszámára, amely adathoz kétféleképpen juthatunk hozzá. Az első megoldás az, hogy számoljuk a rekurzív hívásokat, de ehhez át kell alakítanunk a sum/1 függvényt, ami nem túl szerencsés, ha annak az eredeti funkcionalitását meg akarjuk őrizni. A másik verzió az, hogy az avg a length/1 függvény meghívásával megállapítja a paraméterként kapott lista számosságát, majd a sum/1-től kapott eredményt ezzel az értékkel elosztja. Az avg/1 tehát nem túlságosan bonyolult. A szép megoldáshoz készítsük el a length függvény egy lokális, a modulban általunk megírt változatát. Ehhez csak annyit kell tennünk, hogy a megvizsgálandó listát rekurzívan bejárjuk, és minden lépésben növeljük az átadott számláló értékét, amit a függvény indításakor nullára állítunk. A megszámlálásnál vegyük figyelembe, hogy funkcionális nyelvekben a destruktív értékadás a ciklushoz hasonlóan hiányzik. Alkalmazzuk helyette a sum/1-nél látott paraméterátadásra használt akkumulátoros megoldást, csak itt ne az összeget, hanem a darabszámot

tároljuk. A min és a max függvények megoldását az olvasóra bízzuk. A modul kipróbálásához készítsünk egy olyan függvényt is (exportálni se felejtsük el), amely egy számokat tartalmazó listát ad vissza, hogy a tesztelés ne vegyen el túl sok időt.

8. feladat: Írjunk Erlang programot, amely az alábbi listából kiválogatja az integer típusú elemeket, majd ezeket összegzi egy akkumulátorként használt változóban. A lista felváltva tartalmazhat integer típusú adatokat rendezett n-eseket, amelyeknek az első eleme atom, a második int, de nem tartalmazhat beágyazott listákat, egy mélységűeket viszont igen. Ezeknél a listáknál alkalmazzuk a korábban már elkészített, és listák összegének kiszámítására használatos függvényünket, így a listák esetén is egy integer típusú adatot adhatunk hozzá az aktuális összeghez. A listában található szublisták is csak integer adatokat tartalmazhatnak. A lista tartalmazhat az előzőektől különböző elemeket is, mint a harmadik elem, ami atom típusú. Ezek az elemek természetesen nem adhatóak hozzá az összeghez a típusuk különbözősége miatt.

[{alma, 1}, 2, alma, {alma, alma}, {korte, 3}, 4,5, 3, {dio, 6}, [1,2,3,4], 5].

A bemutatott, és természetesen tesztelésre is használható lista összegzése során az eredmény 39 lesz. Az összegzést végző függvénynek több ággal kell rendelkeznie, mivel a listát rekurzívan be kell járnia. A bejárás során, az első ágban egy case vezérlő szerkezetet helyezhezzünk el, amely mintaillesztéssel ki tudja válogatni a megfelelő elemeket, és a ki tudja emelni az összetett adatszerkezetek belsejéből az integer típusú elemeket. A mintaillesztés a következő mintákat tartalmazza: ({E1, E2} when is_integer(E2)), amely minta a kételemű tupple adatokból veszi ki a második, de kizárólag integer típusú adatokat. A mintára az {alma, alma} rendezett n-es nem illeszkedik. A következő minta az: (A when is_integer(A)) amelyre az integer típusú listaelemek fognak illeszkedni. Természetesen az őrfeltétel nem része egyik mintának sem, de tovább szigorítja azok hatását. A következő minta a (nem beágyazott) lista típusú adatokat szűri ki, amelyekre meg kell hívnunk a sum/1 függvényt, hogy az eredményt hozzá tudjuk adni az összeghez. Ez az ág a következőképpen néz ki: (L when is_list(L) -> Acc1 = Acc + sum(L);), ahol az Acc0 az összegzésre használt változó az adott ágban, az Acc pedig az előző hívástól kapott összeg értékét tartalmazz. A két változóra a destruktív értékadás hiánya miatt van szükség. Ez az ág beágyazott lista esetén hibával megállítja a függvény futását. Hogy ez ne következzen be, valahogyan meg kell akadályoznunk a szublisták paraméterként való átadását a sum/2 számára, vagy a keletkező hibát el kell kapnunk egy kivételkezelővel. talán ez a második megoldás valamivel egyszerűbb. A case kifejezésben el kell még helyeznünk egy default ágat, vagyis egy olyan univerzális mintát, amelyre minden, a fentiektől különböző típusú, vagy formájú adat fennakad. Az előző ágaktól eltérően itt nem alkalmazunk őr feltételt, és a mintába az aláhúzás jelet írjuk, amelyre minden illeszkedik. Tekinthetünk rá úgy, mint a Joker lapra a kártyajátékokban, tehát az aláhúzás jel mindent visz. Ebben az ágban (_ -> …) a függvény meghívja önmagát, de az összeget nem növeli semmilyen értékkel. Ez az ág megfelel a SKIP (üres) utasításnak. A függvény második ágában, amely üres lista esetén fut le, már csak ki kell írnunk az összegzés eredményét. Ez a feladat nagyon érdekes, és rengeteg dolgot megtanulhatunk a megírása során. A megoldás magában foglalja a listakezelést, bemutatja a mintaillesztést függvények és case vezérlőszerkezetek alkalmazásánál, ismerteti a rekurziót, és a több függvény ág használatát is. Megírását mindenképpen javasoljuk.

9. feladat: Készítsünk alkalmazást, amely a K pozitív egész konstanshoz kiszámolja a következő összeget: 1*2 + 2*3 + 3*4 + … + K* K+1. A programot készítsük el Erlang, Clean és F# nyelven is. A megoldás nagyon hasonlít a faktoriális függvényhez, csak itt a szorzások mellett összeadásokat is kell végeznünk. A hasonlóság azt sugallja, hogy ezt a problémát is rekurzív függvényekkel valósítsuk meg. Listát nem kell használnunk, így a mintaillesztésben a megállás feltétele nem lehet az üres lista. A függvény első ágának a paramétere a K szám, amelyet folyamatosan csökkentve minden lépésben elő tudjuk állítani a kívánt szorzatot, majd hozzá tudjuk adni az aktuális részösszeghez. Mivel a számítási műveletek elvégzése során nem kell kiírnunk a részeredményeket, és a K aktuális értékét sem, az teljesen mindegy, hogy pl.: egytől tízig, vagy tíztől egyig haladunk. Amennyiben a részeredmények is kellenek, tárolhatjuk azokat listában, vagy az aktuális K és az eredeti K ismeretében minden lépésben kiszámíthatjuk a kívánt értékeket (K - K'). A függvény első ága tehát megkapja az aktuális K-t, és az eddigi összeget (Acc), majd a (K * K + 1) éréket előállítja és hozzáadja azt az Acc-hoz. A destruktív értékadások hiánya miatt egyszerűbb, ha a következő hívás aktuális paraméter listájában szerepelteti a szorzást és az összeadást tartalmazó kifejezést (func(Acc + (K* (K+1))…).A második ágnak, ezután, már csak annyi feladata marad, hogy az üres lista paraméter elérését követően kiírja, vagy visszaadja az eredményt. Itt a második megoldás, vagyis az eredmény visszatérési értékként való közlése jobb, mivel a funkcionális nyelvekben lehetőség szerint kerüljük a mellékhatásokat. A mellékhatás az az esemény, mikor az adott függvény nem csak az eredményét adja vissza, hanem egyén IO műveleteket is végez.

Az imperatív nyelvekben a mellékhatásos függvényeket eljárásnak, a mellékhatások nélkülieket pedig függvénynek nevezzük. A visszatérési érték formázott kiírására használhatunk a modulban implementált, és az

export listában elhelyezett kiíró függvényt, amely az io:format megfelelő paraméterezése mellett ízlésesen megjeleníti a számításokat végző függvények eredményeit.

10. feladat: Készítsük el a faktoriális értékét kiszámító feladat tail-recursive változatát. Az eredeti programot úgy tudjuk átalakít, hogy a faktoriális program függvényeit átírjuk, és készítünk hozzá egy új ágat is. A farok rekurzió feltétele, hogy a függvény önmagára vonatkozó hívása legyen a függvényben az utolsó utasítás, és ez a hívás ne szerepeljen olyan kifejezésben, mint a (N * fakt(N – 1)). Helyette a (fakt(N-1, N*X)) formulát kell használnunk, ahol az X az előző részösszeg. Ebben az esetben a programunk nagy értékek esetén is jól fog működni, mivel a funkcionális nyelvek sajátos futtató rendszere ezt lehetővé teszi számára nagy N értékek esetén is. igazság szerint itt a változókban tárolható legnagyobb érték fogja a határt, vagyis a rekurzív lefutások számát meghatározni. A faktoriális kiszámítása a fenti változtatások bevezetése mellett innentől ugyanazzal a módszerrel történik, amelyet korábban is alkalmaztunk, és amely módszer a matematikaórákról ismerős lehet mindenki számára. Az N értékét folyamatosan csökkentjük, a szorzat értékét folyamatosan növeljük mindaddig, amíg N értéke nulla nem lesz. Ekkor megállunk, és kiírjuk az eredményt, vagyis az n! értékét.

11. feladat: A rekurzió fontos eleme a funkcionális programozási nyelveknek. A gyakorlása elengedhetetlen, ezért készítsünk olyan modult, amely az ismert programozási tételeket valósítja meg rekurzívan. A programozási tételeket lista adatszerkezetre alkalmazzuk, mivel sorozatok tárolására ez a megoldás kézenfekvő, vagyis pl.: a rendezést, vagy maximum kiválasztást nem a tömb, hanem a lista adatszerkezetre értelmezzük. A modulban az ismertebb tételek közül azokat helyezzük el, amelyeket korábban, az előző feladatok elkészítése során még nem programoztuk le, vagyis a minimum, és a maximum kiválasztást eleve hagyjuk ki. Elsőként talán elkészíthetjük az eldöntés tételét. Amennyiben már tanultunk ilyesmiről, akkor gondoljunk vissza arra, hogyan is kellett megoldani az eldöntés problémáját imperatív nyelveken, ha viszont még soha nem hallottunk ilyesmiről, akkor figyeljünk nagyon a megoldás részleteire. Az eldöntés tétele egy tetszőleges, N elemű sorozatot vizsgál meg, és eldönti róla, hogy tartalmaz-e adott tulajdonságú elemet (elemeket). Az eredménye minden esetben egy igen/nem döntés. Ez a klasszikus megoldás, amelyet az Erlang nyelv sajátosságai miatt egy kicsit meg kell változtatnunk. Az elv marad, de a sorozatot listára cseréljük. A lista bejárását rekurzióra cseréljük, és azt a hatékonysági tényezőt egyelőre nem vesszük figyelembe, hogy az első T tulajdonságú elem megtalálásakor meg lehet állni, vagyis abba lehet hagyni a lista további elemzését. Ezt az elvet, és az implementációját később hozzáadhatjuk a programunkhoz.

Első lépésként tehát készítenünk kell egy két ággal rendelkező, rekurzív függvényt, amely bejárja a listát, majd üres lista esetén kiírja a döntés eredményét. A listát a szokásos módon, mintaillesztéssel szedjük szét első elemre és a további elemeket tartalmazó listára. Az első elemre nézve megállapítjuk, hogy az T tulajdonságú-e. Ennél a pontnál rögtön két problémába is ütközhetünk. Az első probléma az, hogy milyen nyelvi elemekkel írjuk le a T tulajdonság meglétét. A második pedig az, hogy milyen módon adjuk tovább az aktuális döntés eredményét, és a végeredmény szempontjából, a döntések sorozatában ez az aktuális döntés mit jelent, és hogyan befolyásolja a végső döntés eredményét, és egyáltalán hogyan írjuk ezt le Erlang, vagy egyéb funkcionális programozási nyelven.

Az első probléma megoldása lehet, ha úgymond bedrótozzuk a döntést, vagyis írunk egy if-et egy konstans tulajdonságra, de ebben az esetben minden további tulajdonság hozzáadásakor egy újabb függvényt kell készítenünk. Ez a megoldás tehát nem túl általános. Az általános megoldáshoz magasabb rendű függvényt kell alkalmaznunk, amelyben leírjuk a kívánt tulajdonság vizsgálatát, és azt a függvényt paraméterként oda kell adnunk az eldöntést implementáló, és az így már eggyel több paraméterrel rendelkező függvénynek. Ez a megoldás megfelelő, és csak az adott tulajdonságot leíró függvény kifejezést kell megírnunk az aktuális probléma megoldásához.

A függvény tehát három paraméterrel kell, hogy rendelkezzen, amelyekből egy paraméter a lista (a vizsgált sorozat elemeivel), a következő paraméter a D döntési változónk, és az utolsó a sorban a vizsgált tulajdonságot leíró függvényt tartalmazó elem. A függvény első ága tehát megvizsgálja a tulajdonság teljesülését az abban a hívásban leválasztott első elemre, majd a lista maradék részére meghívja önmagát úgy, hogy a D-ben elhelyezi a tulajdonság vizsgálat igaz, vagy hamis eredményét (true/false). A függvény második ága az üres listára hívódik meg, és a végső döntés eredményét adja vissza. Mindezek alapján a második probléma megoldása már nagyon egyszerű, mivel a függvény első futásakor a D változóba a hamis értéket kell csak elhelyeznünk (false), és a második ágban meg kell néznünk, hogy az érték hamisra változott-e. Nyilván, ha már egyszer igazra állítottuk az értékét, többé nem szabad hamisra váltani. Ezen a ponton lehet bevezetni a hatékonyságra vonatkozó megszorítást. Erre szintén több lehetőség is kínálkozik. Pl.: ha egyszer már hamisra változtattuk a D értékét, nem kell további vizsgálatokat végeznünk, de egyszerűen a második ágat kiegészíthetjük egy új ággal úgy, hogy a true esetén mindenképpen ez az ág fusson le, akárhány eleme is maradt a listának. A megoldás a kedves olvasóra van bízva. A többi programozási tétellel is hasonlóan járjunk el. Készítsünk szöveges leírást, esetleg

specifikációt az erre a célra kitalált módszerek valamelyikével, és csak a jól átgondolt tervezést követően kezdjünk bele az implementálásba.

12. feladat: A több ággal rendelkező függvényeknél bevezethetünk egy un.: default ágat, amely bármilyen aktuális paraméter esetén jó, vagy legalábbis elfogadható eredményt produkál. Az a default értékű ágat tartalmazó megoldás nem mindig kivitelezhető, és a mellékhatásokkal rendelkező függvényeknél nem működik. Ennek a problémának a kivédésére, és a technológia elsajátítása érdekében készítsünk egy olyan összeadó függvényt, amely egy, kettő, és három paraméter esetén is megfelelően működik, majd vezessünk be ehhez a függvényhez egy olyan ágat, amely akárhány paraméter esetén is működik. Az N paraméter használata esetén (ahol (N > 3)). A függvény azt az atomot adja vissza, hogy: tul_sok_parameter. A függvény eredményének a formázásához készítsünk egy olyan függvényt is, amely minden egyes ág teljesülésekor megfelelően formázza a kimenet felé az eredményt. A feladat megoldásához ne használjunk case kifejezést. Sajnos, ha nem alkalmazunk összetett adattípust, mint a tuple (rendezett n-es), a függvény paraméterek száma nem fog megegyezni az egyes ágakban, így a fordító program hibaüzenettel leáll. A probléma kiküszöbölése érdekében az egy, a két, és a több paraméterrel rendelkező ágakban is csomagoljuk rendezett n-esbe az formális paraméterlistát. Az első ága paramétere ekkor: (osszeg({A}) ->). Az összes többié, kivéve az N paraméterest, hasonlóan fog kinézni (osszeg({A, B}), osszeg({A, B, C})). A mindent vivő, vagyis a default paraméterrel rendelkező ágat egy egyszerű változó paraméterrel kell ellátni. Ezt az ágat mindenképpen a többi után kell elhelyeznünk, különben a paramétere lenyelné a többi ágat, mivel erre az ágra bármely aktuális paraméter lista illeszkedne (osszeg(A)).

Ez a megoldás sajnos a lista típusú adatoknál nem működik megfelelően, mivel azok összegzése a skalár típusúakkal nem lehetséges. Ha erre az esetre is fel szeretnénk készülni, akkor alkalmazzunk kivételkezelő blokkot, vagy eleve lista adatszerkezetet tervezzünk a megoldáshoz. A feladat második felében a kiírást kell megterveznünk, méghozzá úgy, hogy az atom, valamint egész szám visszatérési értékkel is jól működjön. Ez a feladat is egy több ággal rendelkező függvény megírását igényli. az első, de akár mindkét ágat elláthatjuk őr feltéttelekkel, amelyek szűrik a kapott paraméterlistát a mintaillesztéssel nem, vagy csak nehezen megállapítható tulajdonságok alapján is. Az első ág ez alapján (kiir(A) when is_integer(A)), és a második a (kiir(A) when is_atom(A)) formájú lehet. Kétféle adatra elég lehetne egy őrfeltétel alkalmazása is, de nem baj, ha már eleve többféle kimenetre készülünk, és minden típushoz készítünk egy megfelelő függvényágat, majd elhelyezünk a lista végén egy default ágat is. Ha kiválasztás funkcionalitása kész, az egyes ágakban már csak a kiíró utasítást kell a megfelelő módon formázni, és az adattípushoz előállítani a formátum sztringet.

13. feladat: A felhasználói interfésszel rendelkező programoknak fontos része a hibák kezelése, és ezzel párhuzamosan a hibaüzenetek megjelenítése a képernyőn. A hibakezelés gyakorlásához készítsünk olyan, több ággal rendelkező függvényt, amely tetszőleges modul különleges hibaüzeneteit képes kezelni, és a hibaüzeneteket megjeleníteni a képernyőn a hiba típusának és formátumának megfelelően. A megoldáshoz készítsünk egy több ággal rendelkező, mintaillesztést használó függvényt, amelynek első paramétere a hiba oka atom típusban, majd ezt követően a második paramétere a hibához tartozó üzenet, amely szöveg típusú, és végül a harmadik paraméter a kiírást kezdeményező függvény neve.

pl.: error_message(errorType, Message, FunctionName) -> …

A függvény rendelkezzen egy default ággal, amely rossz paraméterezés esetén is kezeli a kimenetet, és egy általános hibaüzenetet jelenít meg a képernyőn. A hibakezelő rutinunk kipróbálásához készítsünk egy Erlang modult, amelynek három függvénye van. Ez első függvény tetszőleges hosszúságú lista elemeit számolja meg, a második összegzi a paraméterként kapott lista elemeit, a harmadik tükrözi a paraméterre érkezett listát. Mindhárom rekurzív függvény használja az imént elkészített hibakezelő rutint úgy, hogy a függvények törzse kivételkezelést tartalmazzon, és a kivételkezelő a megfelelő paraméterezés mellett hívja meg a hibakezelő, üzenetmegjelenítő függvényt.

A megfelelő működés eléréséhez minden függvény exportáljunk, és a modult fordítsuk le, majd futtassuk a függvényeket hibátlan, és hibás paraméterezés mellett is. A kivételek kezeléséhez használhatjuk a try legegyszerűbb formáját, ahol a hibákat az alábbi minta segítségével kaphatjuk el: (Error:Mesg -> …), ahol az Error változó tartalmazni fogja a hiba okát, a Mesg változó pedig azt az üzenetet, amelyet az Erlang futtatórendszer szolgáltat. Az összes kivételkezelős hibaüzenet illeszkedni fog erre a mintára, de sajnos csak az Erlang esetében. Clean és F# programoknál másképpen kell eljárnunk. A hibakezelő rutinnak ezt a két változót kell megjelenítenie, valamint a kivételkezelést tartalmazó függvény nevét, amelyet minden függvényben konstansként odaírhatunk. Ezek alapján a hibakezelés meghívása a következő: (error_message(Error, Mesg, fuggvenyneve)), ahol a fuggvenyneve atom típus. Az error_message/3 függvényt ez után úgy kell elkészítenünk, hogy a lehető legtöbb hiba elfogására rendelkezzen egy ággal, amely ág akkor hívódik meg, ha a megfelelő, és rá jellemző hibatípust kapja meg paraméterként. Az egyes ágakban a kiíró utasítást a hibához tartozó üzenetek

alapján kell formázni. Az Erlang nyelvű megoldás mellett elkészíthetjük a Clean és az F# nyelvű változatokat is. Természetesen ezen megoldásoknál az adott nyelv lehetőségeihez mérten alakítsuk át a függvényeket, és a kiíratást, valamint a hibakezelést.

14. feladat: Implementáljunk egy olyan Erlang nyelvű modult, amely egy magasabb rendű függvények futtatására alkalmas szerver-alkalmazást valósít meg. A szerver magja egy loop nevű, paraméter nélküli függvény legyen, amely rekurzívan meghívja önmagát. A rekurziót természetesen farok –rekurzióval kell elkészítenünk, hogy a szerver ne álljon le idő előtt. Ahhoz, hogy a szerver a felhasználói számára könnyedén működtethető legyen, dolgozzuk ki a megfelelő kommunikációs protokollt. A protokoll egy rendezett n-esben minden esetben tartalmazza a hívó fél azonosítóját (process ID, vagy node azonosító), az elvégzendő feladat függvény kifejezését (magasabb rendű függvény), valamint azt az adatsort, amelyen a függvény kifejezést futtatni kell. A szerverben alkalmazott protokoll tartalmazzon még egy megállító hármast ({Pid, stop, []}), és egy start ping-hez hasonló műveletet, amellyel ellenőrizni lehet a szerver státuszát ({Pid, ping, []}). A stop üzenetre a szerver küldje vissza a stopped üzenetet a hívónak, a státusz üzenetre pedig a válasz az legyen, hogy running. (Természetesen, ha a szerver nem fut, akkor nem kapunk megfelelő választ.) A szervert szinkron módon kell megszólítani, és a választ a klienseknek is meg kell várnia. Az üzenetküldés lebonyolításához használjunk remote procedure call típusú függvényt, vagyis a rpc modul call függvényét. A feladat megoldásához segítséget találunk a kliens szerver-alkalmazásokról szóló fejezetben. A szervernek tartalmaznia kell a kliens oldali függvényeket is (ami Erlang programoknál megszokott dolog), és a következőképpen kell működnie:

```
rpc:call(server, remote, {self(), fun(X) -> X + 1 end, [12]})
```

Ahol az (rpc:call) függvény könyvtári függvénye az rpc modulnak, a remote, az a szerver alkalmazás interfész függvényének a neve, a (fun(X) …) függvény az a magasabb rendű függvény, amelyet a szervernek futtatnia kell. A függvény természetesen tetszőleges magasabb rendű függvénnyel helyettesíthető. A [12], egyelemű lista a függvény paramétere lesz, amelyet a szerver a következőképpen helyettesít be: Ha a kérést a {Pid, F, Params} tupple mintával kapjuk el a megfelelő receive szerkezettel, akkor az F(Params) függvényhívás minden esetben működik, és a Pid ! F(Params) formulával vissza is lehet azt a hívó felé küldeni. A szerver leállításához annyit kell tenni, hogy a loop függvény megfelelő ágában a rekurzív hívást elhagyjuk.

Mielőtt belekezdenénk a szerver elkészítésébe, próbáljuk ki a magasabb rendű függvények futtatását egy teszt modulban, hogy a jóval bonyolultabb szerver implementálásakor már ne kelljen az alapvető funkcionalitás hibáinak a kijavításával foglalatoskodnunk. A teszt modul függvényét kétparaméteresre kell elkészítenünk. Az első paraméter a futtatásra szánt függvény, a második a paraméter lista. Ennél az egyszerű függvénynél arra az egy dologra kell csak ügyelnünk, hogy a második paraméter mindenképpen lista legyen (hivo(F, Params) when is_list(Params) ->). A függvény törzse a következő egyszerű utasítást tartalmazza: (F(Params)), mivel az F változóban egy magasabb rendű függvény, vagyis egy függvény prototípus található. Ekkor a paraméterként átadott függvényekben kezelni kell a listát, de a bonyolultsága ellenére se használjunk más adatszerkezetet, mivel így biztosíthatjuk a változó paraméterszámot. Inkább a paraméter függvényekben gondoskodjunk a listák kezeléséről. Ez a módszer funkcionális nyelvek esetén megszokott dolog. Az Erlang nyelv tartalmaz is hasonló funkcióval rendelkező könyvtári függvényeket apply néven. A hivo/2 függvényt a fejléce alapján a következőképpen lehet paraméterezni: (hivo(fun(X) -> X * X end, [1,2])). Vegyük észre, hogy a magasabb rendű függvényünk nem foglalkozik a lista adatszerkezettel. Ahhoz tehát, hogy a hivo/2 függvény megfelelően működjön, a törzsében alkalmazzuk az apply/3 függvényt, amely könyvtári függvény automatikusan lefuttatja a kapott függvényt akármilyen listára (apply(mod, F, Params)). Ez a megoldás már megfelelő a szerveres változat implementálására is. A szerverbe integráláshoz már csak annyit kell tennünk, hogy a loop (a szerver belső ciklusát implementáló függvény) megfelelő ágában elhelyezzük a hivo/2-t, majd az eredményét visszaküldjük a műveletek elvégzését kezdeményező kliens felé.

15. feladat: Implementáljunk egy olyan Erlang modult, amely segítségével a listagenerátorok és a rekurzív listabejárások használatát vizsgálhatjuk meg és hasonlítjuk össze. A modul tartalmazza néhány ismert listakezelő könyvtári függvény rekurzív, és listagenerátoros megoldását. A könyvtári függvények, amelyeket implementálnunk kell a következők: lists:members/2 amely függvény egy tetszőleges listáról eldönti, hogy egy adott elem szerepel-e a listában (a megoldása hasonlít az eldöntés tételéhez, de itt a tulajdonság az, hogy az elem szerepel-e a listában). A rekurzív megoldáshoz egy olyan rekurzív függvényt kell készítenünk, amely a lista minden elemét sorra veszi, majd az adott elemet összehasonlítja a többivel. Az egyezést megjegyzi a rekurzív futás alatt, és ha végzett, közli azt a világ számára.

16. feladat: Készítsük el a listagenerátoros megoldását is a feladatnak. Ez a megoldás néhány utasítással rövidebb, mivel minden egyes elemet át tudunk helyezni egy másik listába azzal a feltétellel, hogy az

megegyezik a keresett elemmel. Amennyiben a generátor lefutását követően üres listát kaptunk, a keresett elemet a lista nem tartalmazza, ellenkező esetben igen. Az egyezés lehet többszörös is, mivel a unic feltételt nem kötöttük ki (Lista = [Elem|| Elem <- EList, Elem == Keresett]), ahol az Elista változó tartalmazza a vizsgálandó listát, a Lista a z eredményt tartalmazza, és a Keresett változóban a függvény paraméteréül kapott keresett elem foglal helyet. Ha a Lista üres marad, az elem nem szerepel a sorozatban, különben igen. Ezt a tényt a length(Lista) függvényhívás értékének elemzésével meg is tudjuk állapítani úgy, hogy a members/2 függvény visszatérési értéke a következő kifejezés eredménye lesz: (length(Lista) == 0).

A második könyvtári függvény az előző feladat frappáns megoldásához lehet a length/1, amely a paraméterként kapott lista számosságát adja vissza eredményként. Ezt a rekurzív függvényt megírhatjuk egy olyan függvénytörzzsel, amely számolja az elemeket, de ha ügyesek vagyunk, a mintaillesztés tulajdonságait kihasználva is megírhatjuk az így jóval rövidebb változatot. A listagenerátoros megoldás kicsit komplikáltabb lehet a members/2-nél látottaknál, de itt is van jobb megoldás, amit természetesen a kedves olvasóra bízunk. A megoldás lényegi eleme mindhárom esetben az, hogy a lista elemeit valahogyan sorra vegyük, és számoljuk, mint az összegzést végző feladatok esetében.

A harmadik, és egyben az utolsó implementálásra váró függvény a map, amely tetszőleges lista elemeire lefuttatja a paramétereként megadott függvény kifejezést. Ennek a rekurzív változatában a paraméterként kapott listát elemenként be kell járnunk, majd a szintén paraméterként kapott magasabb rendű függvényt futtatnunk kell minden elemre. A listagenerátor esetében nagyon egyszerű a megoldás, mivel a generátor első részében az elemekre egyszerűen csak meghívjuk a kapott függvény kifejezést ([F(Elem) || Elem <- Lista]), ahol az F, a paraméter függvény. A nem listagenerátort használó változatban rekurzívan be kell járnunk a függvényt, de ennél a megoldásnál. ha szükség van a lista elemeire, el kell a teljes listát tárolnunk, vagy el kell helyeznünk azt, egy erre a célra kitalált adattárban (ETS, MNESIA). A magasabb rendű függvények meghívása a lista elemekre itt sem okoz különösebb problémát, mivel alkalmazhatjuk a szervernél használt műveleteket. Amennyiben a listát minden függvényhívás felbontja az első eleme és a maradék listára, és a lista elemek típusa egységes, és ismert, a következő függvényhívás megoldja a problémát: (F(Elso)), ahol az F a paraméter függvény, az Első pedig a lista aktuálisan leválasztott első eleme.

17. feladat: Készítsünk olyan Erlang nyelvű modult, amely listákat tud összefuttatni. A feladatot semmiképpen ne keverjük össze az ismert összefuttatás tételével, amelyben rendezett adatokat tartalmazó listákból készítünk a két eredeti lista elemszámának összegével megegyező listát úgy, hogy a rendezettséget megőrizzük. Most szintén két listát kell összefésülnünk, de itt a feladat az, hogy az első. atomokat tartalmazó lista és a második, számokat tartalmazó lista adott elemiből készítsünk párokat, és ezeket helyezzük el az új listában. A feladatot csavarjuk meg azzal, hogy amennyiben a függvény egy párokat tartalmazó listát kap, azt bontsa fel két listára, amelyek egyike az atomokat, vagyis a párok első elemeit, a másik a számokat, a párok második elemeit tartalmazza. Ezt a két listát rendezett n-esbe foglalva kell visszaadni, mivel a függvénynek egyszerre csak egy visszatérési értéke lehet.

Készítenünk kell tehát egy minimum két ággal rendelkező függvényt az oda és a visszaalakításhoz. az egyszerűség kedvéért elsőként készítsük el az összefésülést végző változatot. A függvény első ága megkapja az összefésülni kívánt két listát (listakezelo(L1, L2)). Az első lista atomokat tartalmaz, a második számokat. mindkét listát fel kell bontanunk első elemre és a lista maradékára ([E1| M1], [E2| M2]). Elő kell állítanunk az ({E1, E2}) rendezett n-est, majd hozzáadnunk ezt az eredményt tartalmazó listához (…, E ++ [{E1, E2}]). Vegyük észre, hogy a listák összefűzése operátort használtuk az eredmény előállításához, így a rendezett n-est is listává kellett alakítanunk. A következő hívásnak tovább kell adnunk a két lista maradékát (M1, M2, …). Amennyiben elfogynak a listákból az elemek, a második ágnak kell átvennie a vezérlést, és az eredményt vissza kell adnia. A feladat megoldása során érdemes ügyelni arra is, hogy a két lista azonos elemszámú legyen. Ezt megoldhatjuk egy őr feltétel bevezetésével: (when length(L1 == length(L2)), de ezzel a megoldással problémákba ütközhetünk a mintaillesztés miatt (próbáljuk megoldani…).

A második pont a listák szétválasztása kell, hogy legyen. Az egyszerűség kedvéért erre a feladatra használhatunk egy azonos nevű, de más paraméterszámú függvényt, ami egy listát kap, és két listát ad vissza, de alkalmazhatunk atomokat is a függvényágak címkézésére. Mindkét megoldás esetén a lényeg az, hogy a függvény fejrészében, a listák mintaillesztésénél vizsgáljuk meg, hogy azok megfelelő formájúak-e. Csak ebben az esetben futtassuk a függvényt. (listakezelo([{A, Int}, M], E1, E2)), ahol a ([{A, Int}]) minta azért érdekes, mert segítségével csak az olyan listákra fut le az ág, amelyek rendezett párokat tartalmaznak. Amennyiben tovább szeretnénk finomítani a megoldást, helyezzünk el a függvényben egy őrfeltételt is az adatok típusának megállapításához (when is_atom(A) and is_integer(Int)). Az elkészített modult a fordítást követően kipróbálhatjuk konstans listák alkalmazásával (listakezelo(lista1(), Lista2(), []).), amelyekhez a modulon belül elkészítjük a függvényeket: (lista1() -> [a, b, c, d]. lista2() -> [1, 2, 3, 4].). Ezekből a tesztadatokból a következő

listát kell kapnunk: ([{a, 1}, {b, 2}, {c, 3}, {d, 4}]). A második ágat futtatva pedig vissza kell kapnunk az eredeti listákat tuple formában: ({[a, b, c, d], [1, 2, 3, 4]}).

18. feladat: Készítsünk olyan alkalmazást, amely kap egy konstans értéket, majd létrehoz egy ilyen hosszúságú listát. A listát feltölti véletlen számokkal, majd formázottan kiírja a képernyőre, ezután pedig megállapítja, hogy hányféle értéket kapott!

19. feladat: Készítsünk konzol programot, mely kiszámítja egy kifejezés értékét. A program a kifejezést szöveges formában kapja meg egy függvénytől, vagy konstansként tartalmazza.

A megadott kifejezésre az egyszerűség kedvéért a következő megkötések vonatkoznak:

Nem tartalmazhat szóközt, vagy egyéb white-space karaktereket.

Nem lehet benne csak egy operátor és két operandus.

Az operandusok kizárólag egész számok lehetnek, és az operátor két oldalán foglalnak helyet.

A megoldáshoz a kapott szöveget szét kell bontanunk, ami imperatív nyelveknél nem bonyolult, de Erlang-ban, ahol a sztring típus csak un.: szintaktikus cukorka, a lista kifejezések használata nyújthat segítséget a számunkra. A szöveg elemekre bontásához használhatjuk a megfelelő könyvtári modulok konverziós függvényeit. Ha ezzel a megoldással nem boldogulunk, akkor kössük ki, hogy az operátorok, valamint az operandus listában kerüljön a bemenetre, valamint az operátor atom, az operandusok integer típusúak legyenek. Ha a paramétereket a megfelelő módon szétbontottuk, egy egyszerű case kifejezés használatával eldönthetjük, hogy az operátornak milyen valódi művelet felel meg. Pl.: a '+' atom esetén összeadást kell végeznünk, és így tovább. A case-ben szükség lehet egy default ágra is, hogy nem értelmezhető operátor esetén se generáljunk hibás kimenetet, és nem árt, ha egy kivételkezelő blokkot is elhelyezünk az esetlegesen előforduló, végzetes hibák kezelésére. Ha teljesen biztosra akarunk menni, a függvény fejrészében helyezzünk el egy összetett őrfeltételt, amely a típus kompatibilitásért fog felelni: (kif([A, B, C]) when is_integer(A), is_atom(B), is_integer(c)). Ahogy láthatjuk, az összetett őrfeltétel egyes részeit vesszővel választottuk el egymástól, ami egyenértékű azzal, mintha az and operátort alkalmaztuk volna. A függvény törzsében egy mintaillesztés segítségével kiemeljük a három elemet a listából, de ebben az esetben a lista paraméter el is hagyható, vagy rendezett n-esre cserélhető. Ekkor a paraméterlista a következő: (kif({A, B, C}) when …). A függvényhez készíthetünk több ágat is, amelyek a különböző adattípusok használata mellett is megfelelő eredményt szolgáltatnak, pl.: listákat is képesek konkatenálni. a modult egészítsük ki egy olyan függvénnyel is, ami a következő feladatot implementálja.

Vegyünk egy másik, de az előzőhöz hasonló példát: Állítsunk elő véletlen egész számokat, majd határozzuk meg azt, hogy melyik a leghosszabb nullákat tartalmazó sorozat az előállított listában. A megoldás itt is egyszerűnek tűnik, de ha elgondolkozunk egy kicsit, rájövünk, hogy a nullák megszámlálásával itt is bajunk lehet. Az előállított értékek mind számok, és a nullák számát csak abban az esetben tudjuk megállapítani, ha konvertáljuk őket valami más típusra. A szöveges konverzió az, ami elsőként szóba kerülhet. A szövegek hosszát már meg tudjuk állapítani, és a nullákhoz tartozó karakter kódokat meg lehet találni a sorozatokban. A második konverziós típus az atom, de ennél a verziónál az a baj, hogy az atom, ahogy ezt a neve is sugallja, nem bontható elemekre és nem számolgathatjuk benne a nullákat. Elindulhatunk abba az irányba is, hogy a nullák számát matematikai műveletek alkalmazásával adjuk meg. Bármelyik megoldást választhatjuk. A feladat befejezése után ennek a függvénynek a kimenetét is a korábban alkalmazott hibakezelő, és megjelenítést végző függvény segítségével írjuk ki a kimeneti képernyőre.

20. feladat: Készítsünk a lists:flatten/1 függvény mintájára olyan modult, amely többszörösen beágyazott listákat úgymond: kilapít, vagyis a beágyazott listákat kicsomagolja mindaddig, amíg listától különböző adathoz nem jut. Ebben a feladatban a listák mintaillesztését sajátos módon kell alkalmaznunk. A listát most is a lista első elemére, és a lista maradék elemeit tartalmazó listára kell bontanunk, de a kicsomagoláshoz a lista első elemét is további vizsgálatoknak kell alávetnünk, és ha ismét lista adatszerkezethez jutunk, a mintaillesztést ismételten el kell végezni az első elemre is. Kicsit olyan az egész, mintha a lista elejére, és a végére is futtatnánk egy mintaillesztős függvényágat.

A flatten/1 függvény a következő listára : ([[1, 2], {a, 1}, [{b, 2}, [c, 1]]]) meghívva az alábbi eredményt adja: ([1, 2, {a, 1}, {b, 2}, c, 1]). Ahhoz, hogy a mi függvényünk is ezt az eredményt szolgáltassa, mindenképpen egy több ágból álló függvényt kell készítenünk. Az ágakban a szokásos mintaillesztés során le kell választanunk az első elemet, majd egy másik függvénnyel (vagy egy újabb ág bevezetésével) addig kell ismételni a

mintaillesztést, amíg lista a beágyazott elemekre rendre lista adatszerkezetet kapunk vissza (vagy üres listát). Amennyiben megtaláltuk a legbelső elem azt vissza kell adni, és hozzáfűzni egy listához, amelyben a kicsomagolt elemeket gyűjtjük. Ezt a lépést követi a lista további elemeinek a hasonló módon történő vizsgálata mindaddig, amíg üres listához nem jutunk. A feladatot természetesen itt is megoldhatjuk listagenerátorok alkalmazásával, amelyek segítségével a rekurzív hívásokat válthatjuk ki. A listák kilapítása egyébként az Erlang változatot adattípusai, és a sokféle visszatérési érték használata miatt nagyon hasznos művelet.

21. feladat: Az Erlang nyelv alkalmas párhuzamos, és elosztott programok készítésére. A több szál egyidejű futtatását a rendszer automatikusan levezényli. Nekünk csak azt a függvényt kell megírnunk, ami a konkrét számításokat elvégzi, és ezt paraméterként odaadni a szálkezelést végző rutinoknak. a szálkezelő elindításához az: (Eredmény1= [rpc:async_call(node(), ?MODULE, function, [Params] || <- Lista])) kifejezést kell kiértékeltetni. Az rpc könyvtári modul, amelynek az async_call függvénye aszinkron módon elindítja a paraméterként kapott függvényt annyi szálon, ahány elemű a listagenerátorban megadott lista (Lista). Ez a lista nem egyezik meg a paraméterként átadottal, amelynek annyi eleme van, ahány paramétert vár a párhuzamosan futtatott függvény. A következő kifejezés a ([_ = rpc:yield(K) || K <- Eredmény1]), amely az Eredmény1 lista elemeit, vagyis az előző kifejezés kimenetére érkező adatokat dolgozza fel. A két kifejezés mellé természetesen meg kell írnunk a tényleges számításokat végző függvényt, amelyet a példában function névvel illettünk, és amely egy paraméterrel rendelkezik.

Most, hogy tudunk elosztott programokat készíteni, írjunk egy olyan modult, amelynek interfész függvénye tetszőleges függvényt képes párhuzamosan futtatni egy szintén paraméterként megadott lista elemeire. A feladat megoldásához használnunk kell a magasabb rendű függvényekről szerzett ismereteinket, és a párhuzamosítás funkcionalitását. A modul egyetlen exportált függvénye két paraméterrel kell, hogy rendelkezzen. Az első egy olyan függvény, amelyet párhuzamosan futtatnunk kell, a második pedig egy olyan lista, amely tartalmazza függvény aktuális paramétereit, minden futtatott szálhoz egyet, és minden ilyen elem szintén egy lista, amely annyi számú paramétert tartalmaz, amennyit a függvény vár. Következzen most egy példa a paraméterezésre. Amennyiben egy olyan összeadó függvényt szeretnénk párhuzamosan futtatni, amelynek két paramétere van (A, B), és az elosztást 10 elemre kívánjuk elvégezni, egy dupla mélységű listát kell készítenünk, ahol minden beágyazott lista egy lehetséges paraméterlistája az összeadásnak ([[1,3], [6, 8], …, [n, m]]).

A futtatásra szánt függvény a sum(A, B) -> A + B, amely függvény nevét csak meg kell adnunk a függvénynek (sum). Az interfész ekkor a következő formát ölti: (sum, [[1, 2], [4, 6]…])). A formális paraméter lista az aktuális alapján nagyon egyszerű, mivel csak két változót kell tartalmaznia. Ha szép programot szeretnénk írni, akkor a második paramétert megvizsgálhatjuk egy őr feltétellel, ami a lista adatszerkezetet szűri (inerf(F, Params) when is_list(params)). A függvény törzsében a következő kifejezést kell elhelyeznünk: (KRes = [rpc:async_call(node(), ?MODULE, F, Params]), ahol a ?MODULE makró az éppen használatban lévő modult jelenti. A KRes változóban kapott eredményt ezután már csak fel kell dolgoznunk a (Res = [_ = rpc:yield(K) || K <- KRes]) kifejezéssel (a listagenerátorban azért szerepel az aláhúzás jel, mert a yield függvény ok atomjára, amelyet futása végén ad vissza, nem vagyunk kíváncsiak).

22. feladat: A proplist azErlang programozási nyelv egy érdekes eleme, melynek segítségével az imperatív nyelvekben is megszokott dictionary, vagy az enumeration típusokhoz hasonló adatszerkezeteket hozhatunk létre. A proplist valójában egy olyan lista, amely rendezett párok rendezetlen sorozatát tartalmazza. A párok első eleme minden esetben kulcsként funkcionál, a második elem a kulcshoz tartozó érték. Kicsit hasonlít ez a telefonkönyvhöz, ahol minden névhez tartozik egy telefonszám.

A proplisteknek az a nagy előnye, hogy készült a kezelésükhöz egy könyvtári modul, amely rengeteg listakezelő (proplist kezelő) függvényt implementál, amelyek közül néhányat mi is alkalmazni fogunk, esetleg meg is írjuk ezek némelyikét.

Az első ilyen hasznos függvény a proplists:get_value/3, amely függvénynek az első paramétere a párokban keresett kulcs, a második eleme az a proplist, amelyben keressük az értéket, a harmadik paraméter pedig opcionálisan tartalmazza azt az alapértelmezett értéket, amelyet a sikertelen kereséskor vissza kell adni. Erre azért van szükség, hogy minden esetben legyen eredménye a keresésnek, és a hívó rutint ne kelljen hibakezelőkkel kiegészíteni. A get_value egyetlen értéket ad vissza, ha a kulcs szerepel a listaelemek között, akkor a hozzá tartozó értéket kapjuk eredményül, ha nem szerepel, akkor az alapértelmezett paramétert (amennyiben adtunk meg ilyet). A következő példaprogram egy háromelemű listából választ ki egy olyan értéket, aminek a kulcs az alma atom, és amennyiben nem talál ilyen kulcsot, üres listával tér vissza: (proplists:get_value(alma, Lista, [])). A paraméterként kapott lista a következő: (ista = [{alma, 1}, {korte, 2}, {dio, 3}]). Most, hogy kipróbáltuk a get_value függvény működését, írjuk meg a sajátunkat, ami ugyanezzel a funkcióval rendelkezik, mint az eredeti, de mi implementáljuk. Hasonlót már készítettünk a listák

összefésülésekor annyi különbséggel, hogy ott egy új listát kellett létrehozni, itt pedig egy elemre kell keresnünk.

Az első megoldás a listagenerátorokat veszi alapul, és azok segítségével keresi meg az adott kulcshoz tartozó elemet. A paraméter lista a korábban létrehozott Lista változóban foglal helyet. Ezt a listát kell végigjárnunk, és minden benne található pár első elemét össze kell hasonlítanunk a keresett elemmel, és egyezés esetén beletenni azt egy másik listába. Több találat esetén sincs probléma, mivel a lista első elemét gond nélkül le tudjuk választani a mintaillesztés segítségével. A generátor ekkor a következő képen alakul: (E = [Elem || {Kulcs, Elem} <- Lista, Elem = KeresettElem]). A KeresettElem a függvény paramétere, és azt a kulcsot tartalmazza, amit keresünk. A Lista is paraméter, és alkalmazhatunk egy harmadik, alapértelmezett paramétert is, ahogy ezt a get_value függvénynél láthattuk. A generátor minden listában található párt kettébont egy tupple mintával, majd megvizsgálja a feltételében, hogy annak első eleme (Kulcs) megegyezik-e a keresett elemmel (KeresettElem). Egyébként itt is mintaillesztést használ a vizsgálatra, és azt a tényt használja ki, hogy minden elem illeszkedik önmagára. Ha igen, akkor az Elem változóban található értéket kell beletenni az eredmény listába, és levenni az első elemet, hogy ez legyen a függvény eredménye ([KElem| _Vege]). Ha nincs találat, akkor egyszerűen csak visszaadjuk a default paramétert. A függvény fejrésze ez alapján a következő: (getvalue(KeresettElem, Lista, []) when is_list(Lista) -> …). A törzse pedig a következő utasításokat tartalmazza: (E = [Elem || {Kulcs, Elem} <- Lista, Elem = KeresettElem, [KElem| _Vege] = E, KElem.). A függvény végére elhelyezhetünk egy case kifejezést, amely megvizsgálja, hogy az E tartalmaz-e elemeket, mert ha nem, az előző mintaillesztés hibával leáll. Tartalom esetén mintát illesztünk, egyébként az alapértelmezést adjuk vissza.

23. feladat: A proplist működési elvéhez nagyon hasonlatos az ETS táblák működése. Az ETS egy olyan memóriában működő adatbázis kezelő alkalmazás, amely segít abban, hogy az Erlang programokban globális adatokat használhassunk. Az Erlang modulok függvényei közötti adatcsere a paraméter átadáson kívül máshogy nem nagyon oldható meg, mivel a globális változók eleve mellékhatást jelentenek a függvényekben. A szerverekben használt közös adatok kezelése is üzenetküldésekkel valósul meg, így a globális változók ebben az esetben sem használhatóak. Tehát marad az ETS tábla. Az ets táblákat létre kell hoznunk, és azok mindaddig a memóriában lesznek, amíg az őket létrehozó program fut, vagy hiba nem történik a tábla kezelése közben. Hiba esetén azt mondjuk, hogy a tábla elszállt. A táblákat létrehozhatjuk az ets:new(…) függvénnyel, ahol a táblának adhatunk nevet (atom típusban), és beállíthatjuk, hogy a táblát a nevén keresztül is el lehessen érni. Erre az ért van szükség, hogy a tábla azonosítóját ne kelljen szintén globális változóként kezelni, ha a különböző (nem egymást hívó) függvényekben el szeretnénk érni azt (named_table). Ezen kívül az egyezőségek kiküszöbölésére beállíthatjuk még a set tulajdonságot is, így biztosítva az un.: unic működést. A tábla létrehozása ekkor a következőképpen alakul: (ets:new(table1, [set, named_table])), amely utasítás létrehozza a table1 nevű táblát, amely alapértelmezés szerint kulcs – érték párokat tartalmaz a proplisekthez hasonlóan. ez a tény arra enged következtetni, hogy az ets tábla kitűnő a proplistek tárolására és globális adatcserék lebonyolítására az adott modul függvényei között. A tábla megszüntetésére az (ets:delete(table1)) függvényhívást használhatjuk, amely nem csak a tartalmat, hanem a táblát is megszünteti. Amennyiben elemeket szeretnénk beszúrni a táblába az (ets:insert/2) függvényt használjuk (ets:insert(table1, {kulcs, 1})). A beszúrásban az első paraméter a tábla neve, természetesen ez csak a named_table tulajdonság megléte esetén működik, különben a tábla azonosítóját kell használnunk, amit a létrehozásakor kapunk meg (Azonosito = ets:new(…)). A második paraméter az a páros, aminek első eleme a kulcs, mai azonosítja az adatot, a második pedig az adat. Ezeket rendezett párok formájában adhatjuk meg, de természetesen az n-esek egymásba ágyazásával, vagy listák beágyazásával összetett adatokat is létrehozhatunk.

A keresése is a kulcsok alapján történik, a proplisteknél látottakhoz hasonlóan, csak itt az ets táblák kezelésére alkalmas függvények felhasználásával. A legegyszerűbb módszer az (ets:lookup(table, key)) hívás, ahol a key a keresett kulcs, és a függvény visszatérési értéke az adat lesz. Az ets táblákat egyszerűen listává alakíthatjuk az (ets:tab2list(table1)) függvényhívással.

Tegyünk próbát az ets táblák használatára. Az erlang parancssorban hozzunk létre egy tetszőleges nevű táblát, szúrjunk bele adatokat, majd írassuk ki azokat, és végül töröljük. hA mindez sikerült, akkor készítsük el a saját adatbázis kezelő modulunkat úgy, hogy az a lehető legjobban megvalósítsa a platform függetlenség követelményeit. A függetlenség azt jelenti, hogy a modul függvényei szabványos interfésszel rendelkeznek, amely interfész akkor sem változik meg, ha az adatbázis kezelő programot kicseréljük, vagyis a modult használó programok észre sem veszik a változást. Ez a technika nagyon hasznos minden adattároló program megírásakor, mivel soha nem tudhatjuk, hogy mikor kell kicserélni akár hatékonysági okokból az adatbázis kezelőnket. Ilyenkor jó, ha nem kell a teljes programot átalakítanunk, csak az adatbázis kezelő modulunk függvényeinek a törzsét. Sokszor az is előfordul, hogy a modult használó programok többféle paraméterezéssel rendelkeznek, vagy az új verzió kívánja meg az új paraméterezést, de a régi módszert használó változatok is futnak a programunkból. Ilyen esetekben be kell vezetni egy kompatibilitást megőrző, régi paraméterekkel rendelkező

függvényhívásokat kompenzáló, konverziós függvényt is, de ezt is csak akkor tudjuk megtenni, ha platform függetlenre készítjük el a modult.

Az elkészítendő modul interfésze tehát tartalmazza az ets-ben használható függvényhívások megfelelőit, és azok egyszerűen csak hívja meg a kellő átalakítások mellett. A terv a következő: szükségünk lesz egy create függvényre az ets:new szinonimájaként. ennek a paramétereit jól kell megválasztani, de a legjobb az, ha az eredeti paraméterezést alkalmazzuk: (create(TableName, Params) when is_list(Params)), ahol a TableName a tábla nevét, a Params pedig a beállításokat tartalmazza ([set, named_table]). A függvény törzse a következő: (ets:new(TableName, Params)). A when feltételét azért vezettük be, hogy a nem lista paramétereket szűrjük, esetleg listába rakjuk, így segítve a felhasználóinkat. A függvény visszatérési értéke a tábla azonosítója kell, hogy legyen, mivel a nem nevesített táblákkal is tudnunk kell dolgozni. Ez akkor is jól fog jönni, ha az ets helyett MNESIA táblákat használunk, vagy esetleg SQL-t. A delete függvény elkészítése a create ismeretében már nagyin egyszerű. A paramétere a tábla neve, az ets:delete/1 függvényt hívja meg és a visszatérési értékét választhatjuk logikai típusúra, de visszaadhatjuk az ets:delete/1 visszatérési értékét is.

Az insert, és a lookup is hasonlóan működik. Választhatunk beszédesebb neveket is, pl.: a find, vagy a keres megfelelő lesz. A paraméterezését hagyjuk meg, vagyis ugyanolyanra alakítsuk, mint az eredeti függvényé: (find(Table, Key)), ahol a Table a tábla neve, és a Key a keresendő kulcs, a visszatérési érték pedig a kulcshoz tartozó érték. Most, hogy a paraméterezést mi alakítjuk, lehetőségünk adódik a proplist-nél látott alapértelmezett érték bevezetésére (find(Table, Key, [])), amelyet a függvény harmadik paramétereként kell megadnunk (az üres lista [] megfelelő lehet, de rábízhatjuk a felhasználóra is a választást). Ilyenkor a függvény törzsébe egy case kifejezéssel megvizsgálhatjuk, hogy van-e visszatérési érték, és ha nincs, a default értéket adjuk vissza. Az insert az ets:insetr-et hívja, és ugyanúgy jár el, mint a delete.

A delete függvényt implementáló saját változatunkat az SQL után szabadon hívhatjuk drop-nak is (drop(Table)), és ebben csak az ets:delete/1 függvényt kell meghívni, és annak a visszatérési értékével kell visszatérni. A funkcionalitás nagy részét ezzel a függvénnyel le is tudtuk, már csak egy funkció hiányzik, az adat törlése a táblából, vagy ami még jobb a teljes tartalom törlésével kombinálható törlés függvény, amit a korábbi, drop elnevezés használata miatt nevezhetünk delete-nek. A delete, amennyiben nincs más paramétere, csak a a tábla neve, minden adatot töröl a táblából. Ennek a legegyszerűbb módja, ha a táblát eldobjuk, majd újra létrehozzuk. Ezután készíthetünk egy ugyanilyen nevű, de más paraméterszámmal rendelkező változatot is, aminek az első paramétere a tábla neve, a második pedig a törlendő pároknak a kulcs, vagyis az első eleme. Az a verzió az ets:tab2list függvény segítségével listává alakítja a tábla tartalmát, és egy listagenerátor segítségével előállítja a keresett elemeket nem tartalmazó új listát. Ha ezzel megvan, akkor törli a táblát, majd ismét létrehozza, és az új lista elemeivel feltölti azt (feltöltéshez használhatja a modul erre célra készített interfész függvényét).

Ha megvagyunk a modullal, akkor próbáljuk ki. Ehhez csak annyit kell tennünk, hogy az imént a parancssorba gépelt függvényhívásokat megismételjük a saját függvényhívásainkra cserélve. Amennyiben az eredmény azonos az előzővel, vagy legalábbis ekvivalens (az átalakítások miatt) akkor készen is vagyunk a feladat megoldásával.

24. feladat: Készítsük el a jól ismert quicksort program Erlang nyelvű implementációját, amelyet az alábbi példaprogramban be is mutatunk.

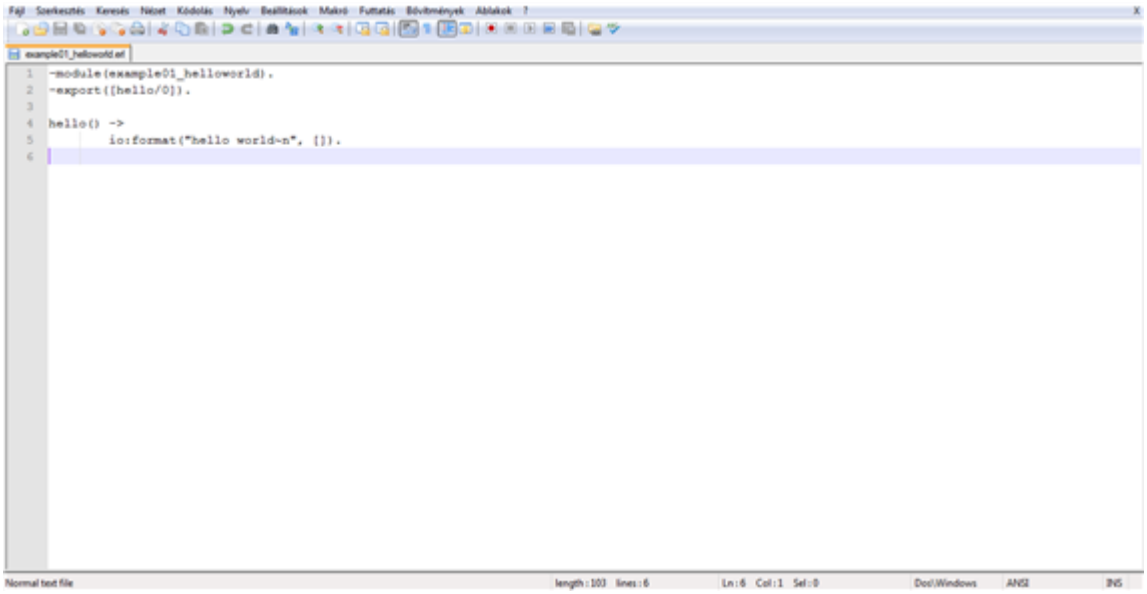
M.3. programlista. Gyorsrendezés

```
quicksort([H|T]) ->
           {Smaller_Ones,Larger_Ones} = split(H,T,{[],[]}),
            lists:append( quicksort(Smaller_Ones),
            [H | quicksort(Larger_Ones)]
);
quicksort([]) -> [].
        split(Pivot, [H|T], {Acc_S, Acc_L}) ->
         if Pivot > H -> New_Acc = { [H|Acc_S] , Acc_L };
              true -> New_Acc = { Acc_S , [H|Acc_L] }
         end,
split(Pivot,T,New_Acc);
split(_,[],Acc) -> Acc.
```

A programot egy saját modulba implementáljuk, vagy integráljuk abba a modulba, amelyben az alapvető programozási tételeket készítettük el. Ehhez a modulhoz készíthetünk egy interfész függvényt, amelynek az első paramétere a futtatandó programozási tétel, a második paramétere pedig az a sorozat, vagy skalár típusú változó, amelyre az adott tételt értelmezni kell. A modulhoz, és az exportált függvényeihez készítsünk hibakezelő rutinokat. Ezen a ponton használhatjuk a korábban már szintén elkészített hibakezelő függvényt, amely a hibaüzenetek mellett azt is megmutatja, hogy mely függvényhívás okozta a hibát. A hibakezelőt érdemes kiegészítenünk egy olyan ággal is, ami nem a hibákat, hanem az egyes függvények eredményét jeleníti meg formázottan, az adott visszatérési érték típusához igazodva.

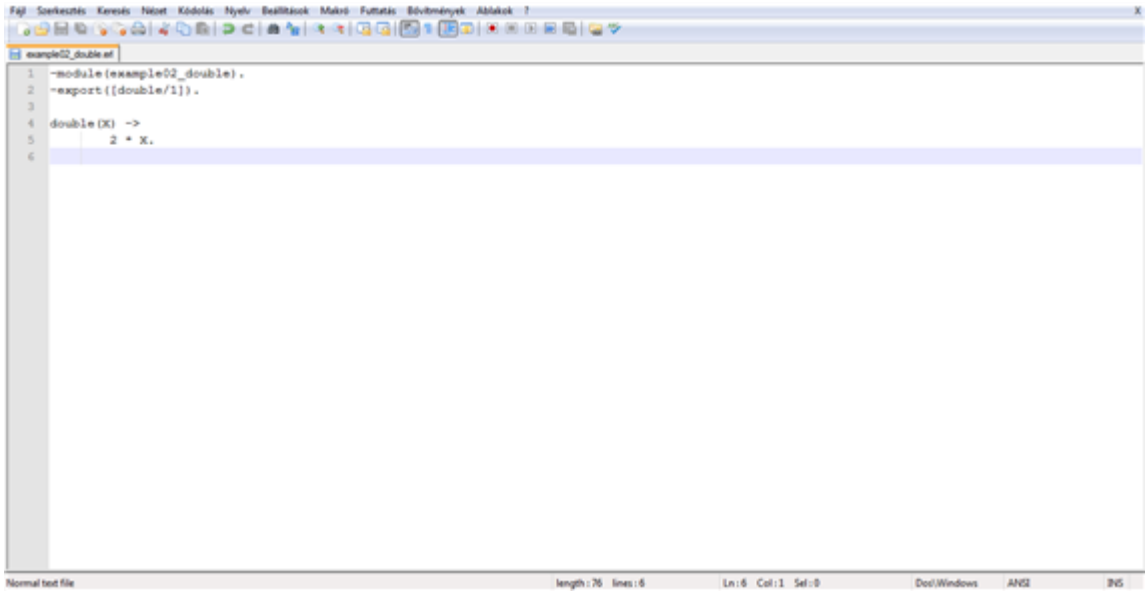
10.6. A fejezetekhez tartozó képek (Feladatok szerkesztés közben és a kimeneti képernyők)

Hello world



A program kimeneti képernyője

Szám szorzása

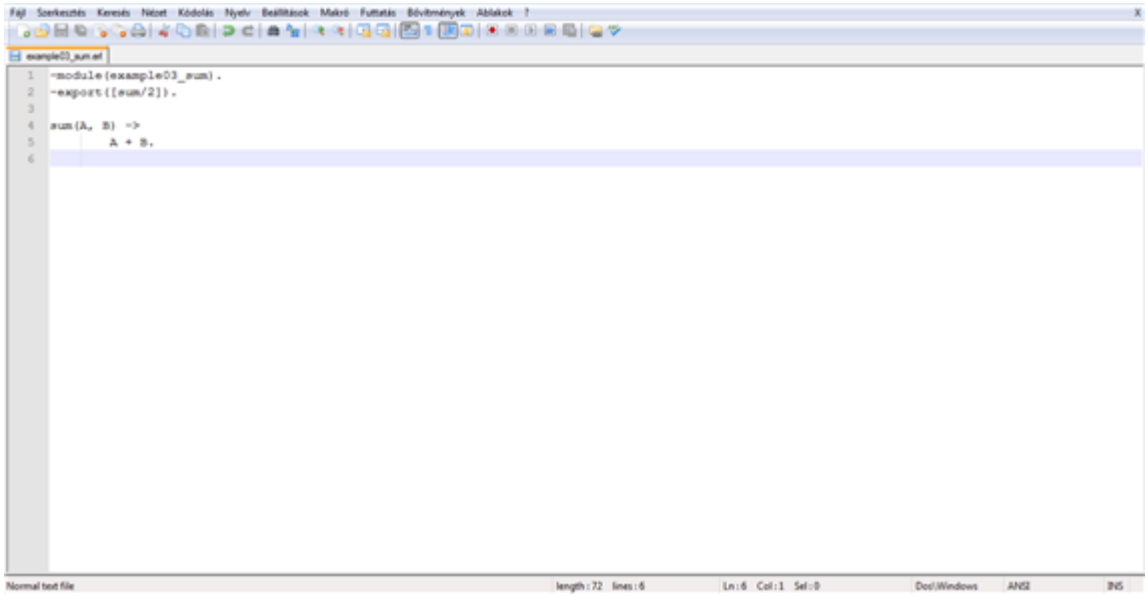


```
-module(example02_double).
-export([double/1]).

double(X) ->
      2 * X.
```

A program kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example02_double:double(4).
8
2> example02_double:double(19).
38
```

Összegzés

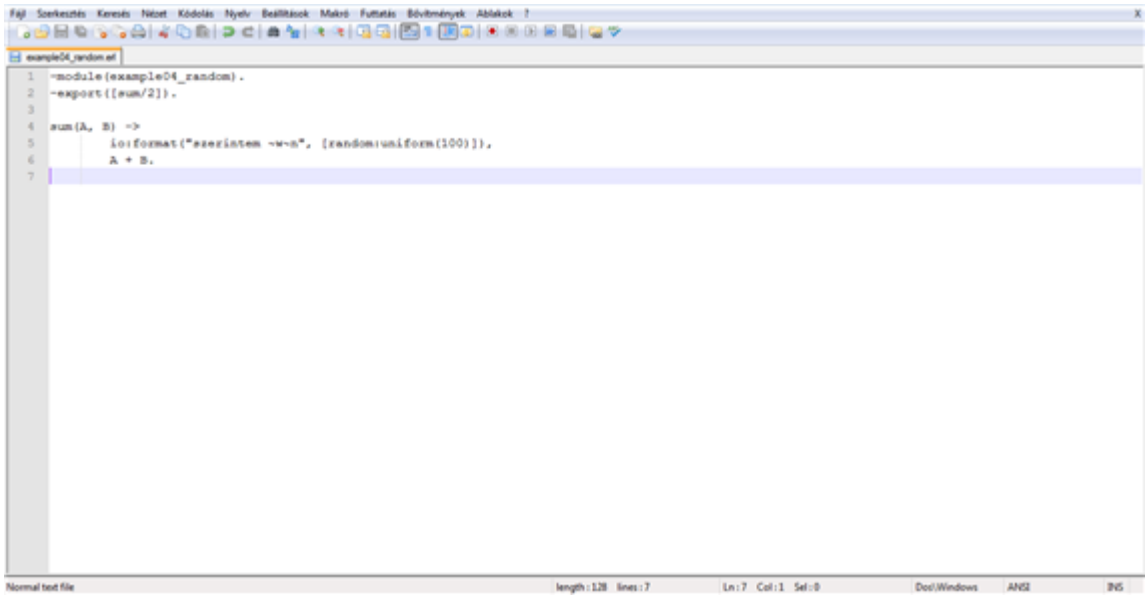


```
-module(example03_sum).
-export([sum/2]).

sum(A, B) ->
      A + B.
```

A program kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example03_sum:sum(5,12).
17
2> example03_sum:sum(16,74).
90
```

Véletlen szám generálás

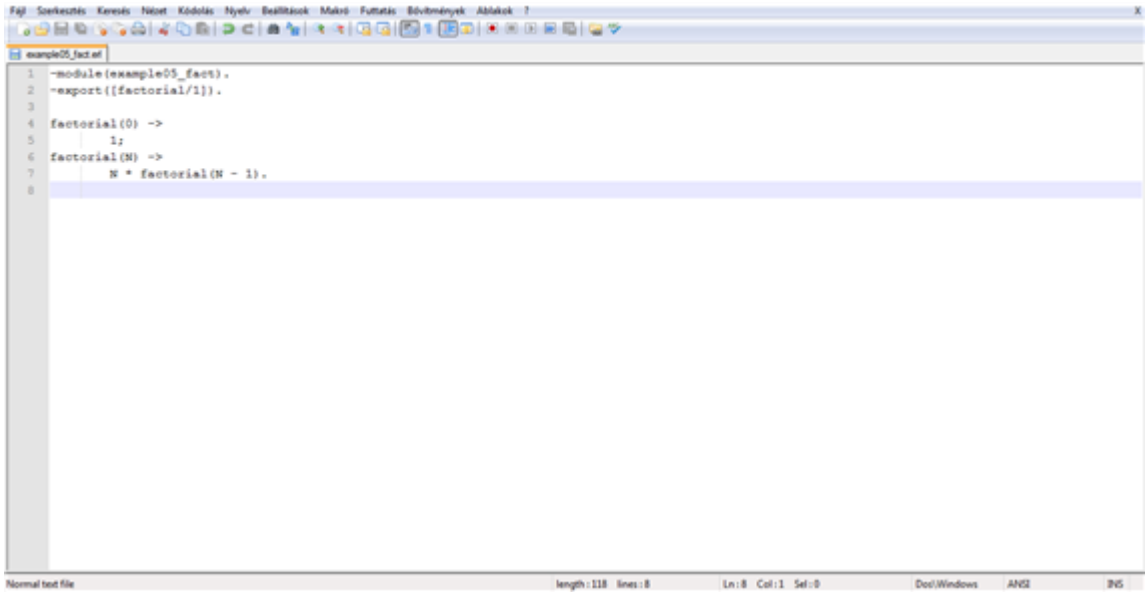


```
-module(example04_random).
-export([sum/2]).

sum(A, B) ->
        io:format("szerintem ~w~n", [random:uniform(100)]),
        A + B.
```

A program kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example04_random:sum(12,39).
szerintem 45
51
2> example04_random:sum(64,20).
szerintem 73
84
```

Faktoriális program

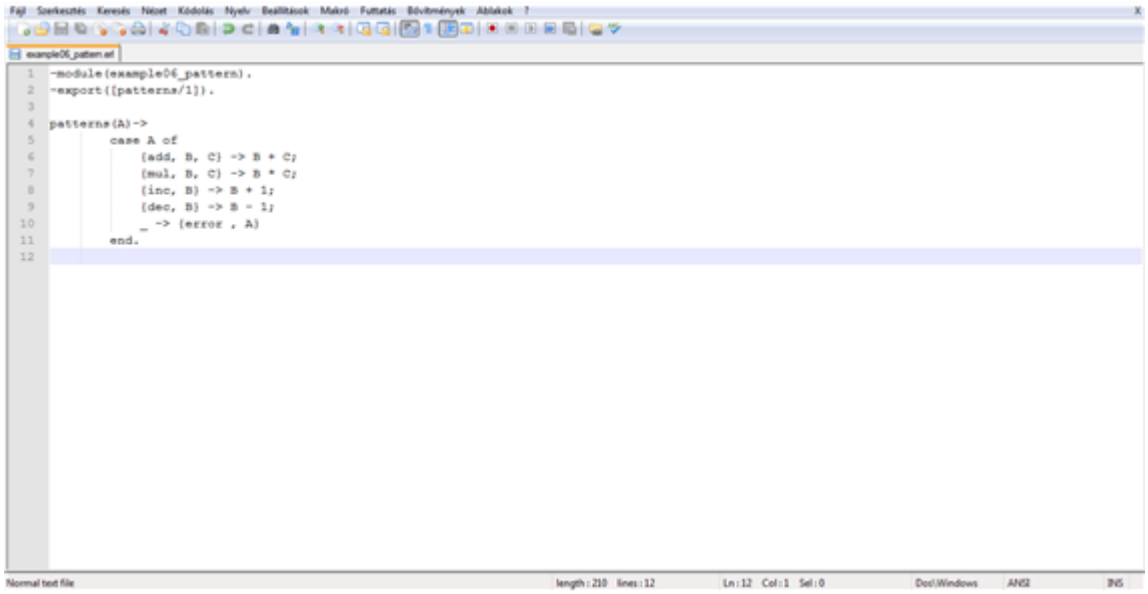


```
-module(example05_fact).
-export([factorial/1]).

factorial(0) ->
        1;
factorial(N) ->
        N * factorial(N - 1).
```

A program kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example05_fact:factorial(5).
120
2> example05_fact:factorial(8).
40320
```

Mintaillesztés

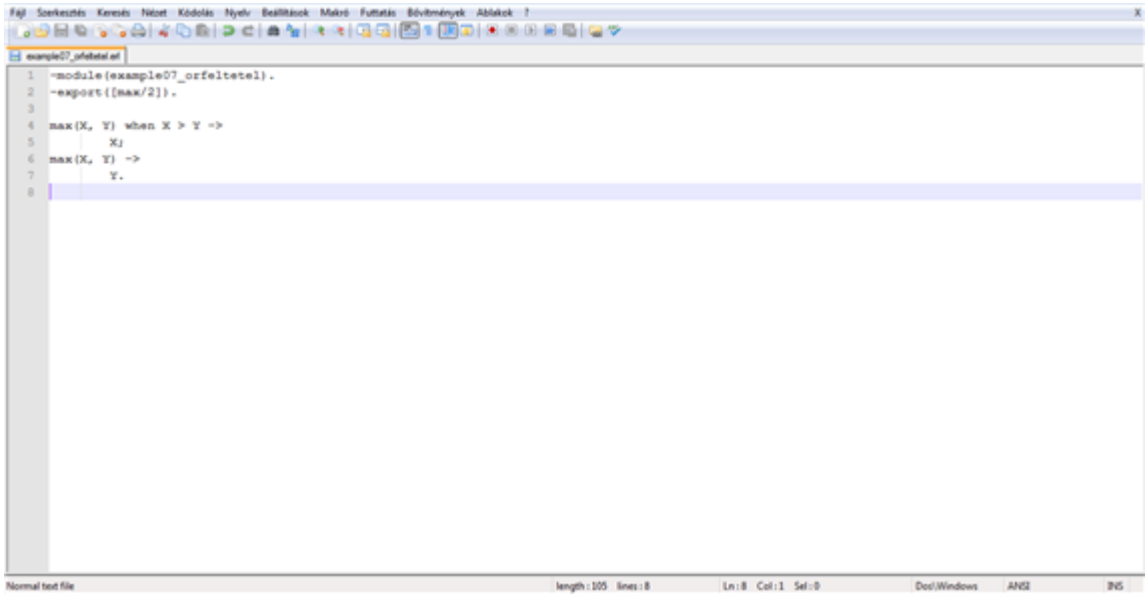


```
-module(example06_pattern).
-export([patterns/1]).

patterns(A)->
        case A of
            {add, B, C} -> B + C;
            {mul, B, C} -> B * C;
            {inc, B} -> B + 1;
            {dec, B} -> B - 1;
            _ -> {error , A}
        end.
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example06_pattern:patterns({add, 2, 91}).
93
2> example06_pattern:patterns({mul, 2, 91}).
182
3> example06_pattern:patterns({inc, 9}).
10
4> example06_pattern:patterns({dec, 9}).
8
5> example06_pattern:patterns({dec, 9, 34}).
{error,{dec,9,34}}
```

Maximum meghatározása

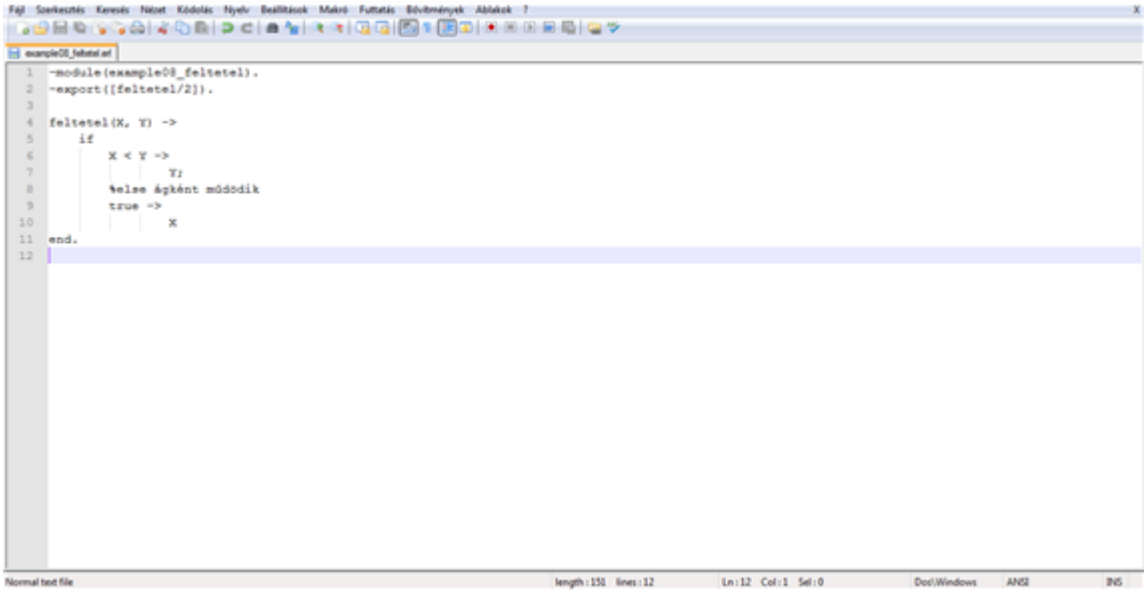


```
-module(example07_orfeltetel).
-export([max/2]).

max(X, Y) when X > Y ->
        X;
max(X, Y) ->
        Y.
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example07_orfeltetel:max(36,27).
36
2> example07_orfeltetel:max(36,64).
64
```

Elágazás

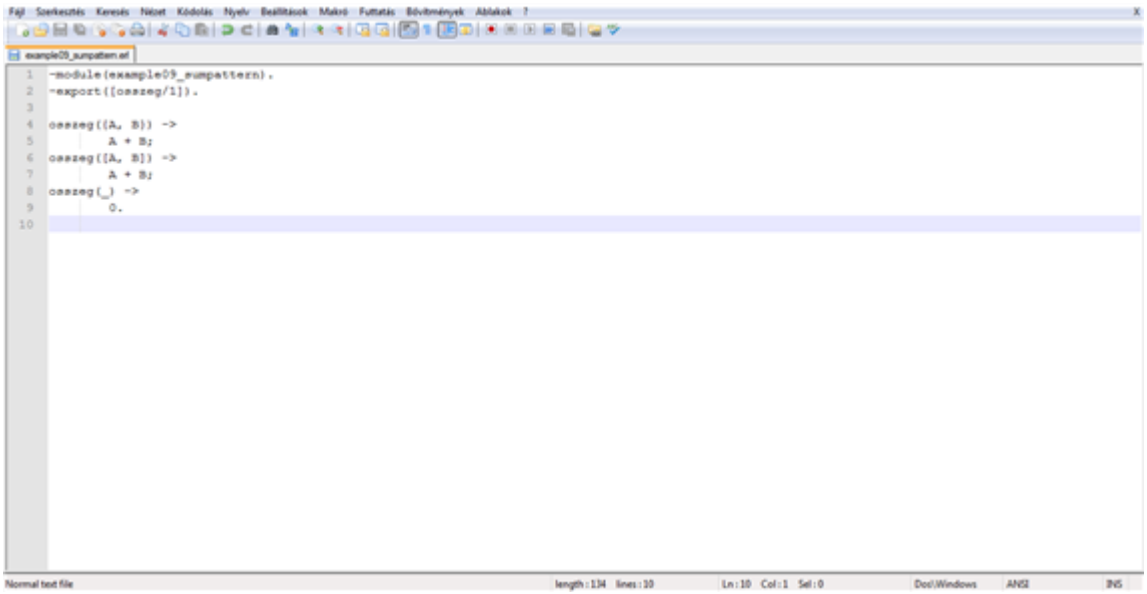


```
-module(example08_feltetel).
-export([feltetel/2]).

feltetel(X, Y) ->
    if
        X < Y ->
            Y;
        %else ágként működik
        true ->
            X
end.
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example08_feltetel:feltetel(67,20).
67
2> example08_feltetel:feltetel(1,20).
20
```

Két elemű lista, vagy tuple összegzése

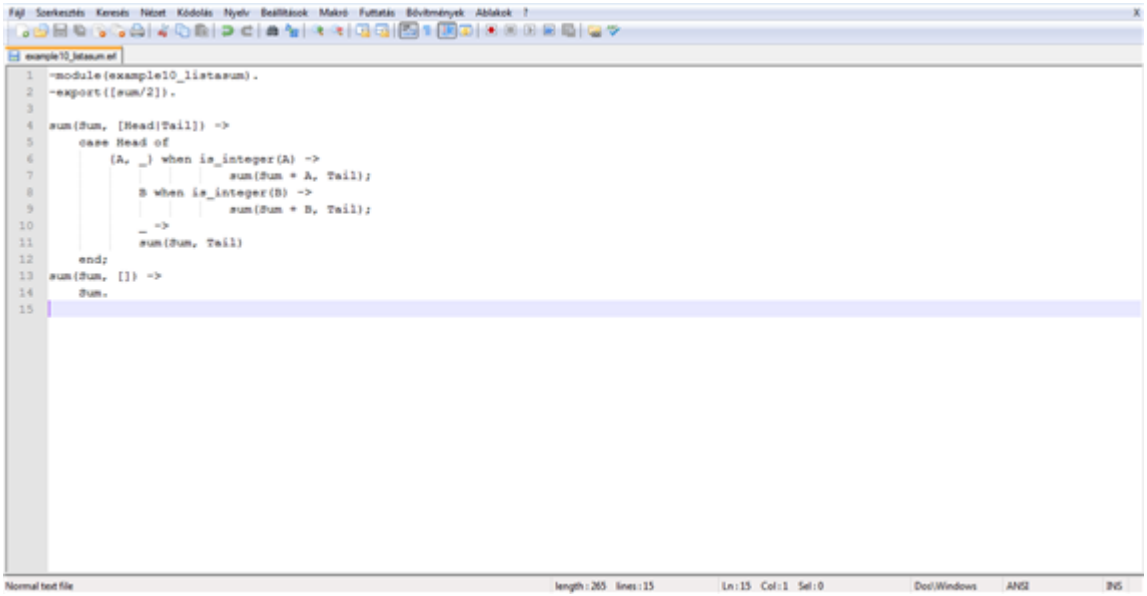


```
-module(example09_sumpattern).
-export([osszeg/1]).

osszeg({A, B}) ->
      A + B;
osszeg([A, B]) ->
      A + B;
osszeg(_) ->
      0.
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example09_sumpattern:osszeg({88,54}).
142
2> example09_sumpattern:osszeg([88,54]).
142
3> example09_sumpattern:osszeg([88,54,ksd]).
0
4> example09_sumpattern:osszeg(11).
0
```

Lista összegzése

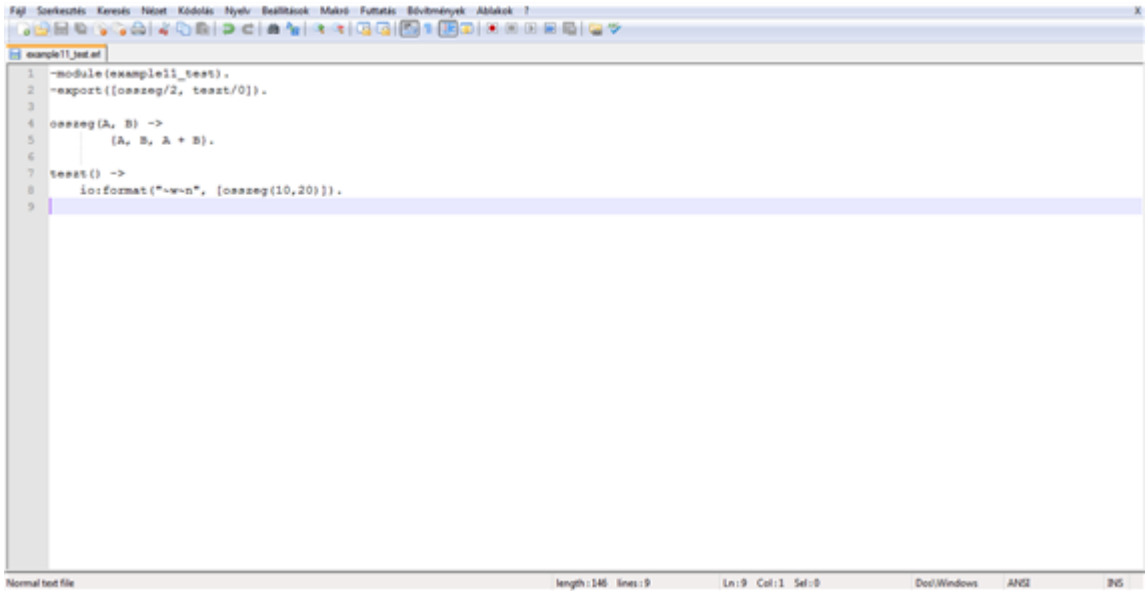


```
-module(example10_listasum).
-export([sum/2]).

sum(Sum, [Head|Tail]) ->
    case Head of
        {A, _} when is_integer(A) ->
                    sum(Sum + A, Tail);
        B when is_integer(B) ->
                    sum(Sum + B, Tail);
        _ ->
        sum(Sum, Tail)
    end;
sum(Sum, []) ->
    Sum.
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example10_listasum:sum(0, [1,2,3,4,5]).
15
2> example10_listasum:sum(22, [23,0,89,40,77]).
251
3> example10_listasum:sum(22, [23,0,-89,40,-77]).
-81
```

Összetett visszatérési érték kezelése

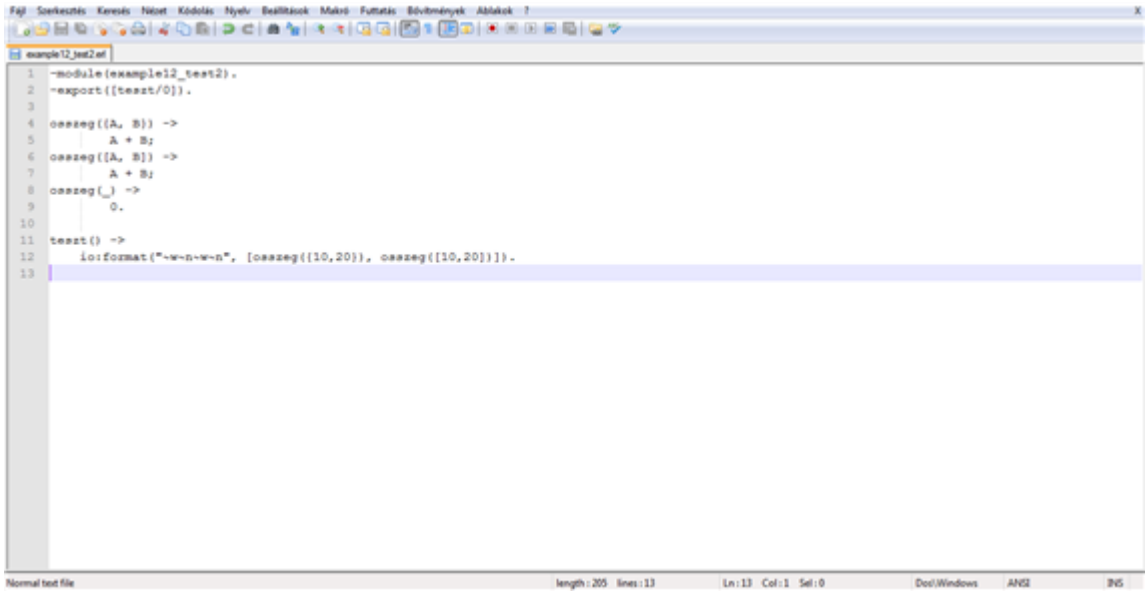


```
-module(example11_test).
-export([osszeg/2, teszt/0]).

osszeg(A, B) ->
        {A, B, A + B}.

teszt() ->
    io:format("~w~n", [osszeg(10,20)]).
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example11_test:teszt().
{10,20,30}
ok
```

Több összetett visszatérési érték kezelése

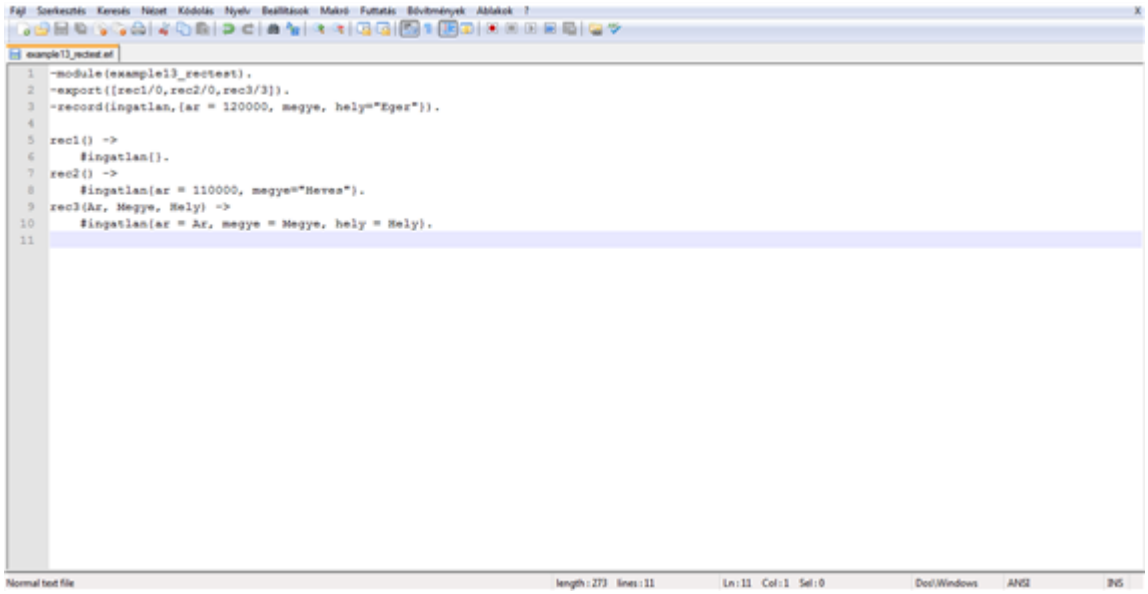


```
-module(example12_test2).
-export([teszt/0]).

osszeg({A, B}) ->
        A + B;
osszeg([A, B]) ->
        A + B;
osszeg(_) ->
        0.

teszt() ->
    io:format("~w~n~w~n", [osszeg({10,20}), osszeg([10,20])]).
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example12_test2:teszt().
30
30
ok
```

Rekord kezelése

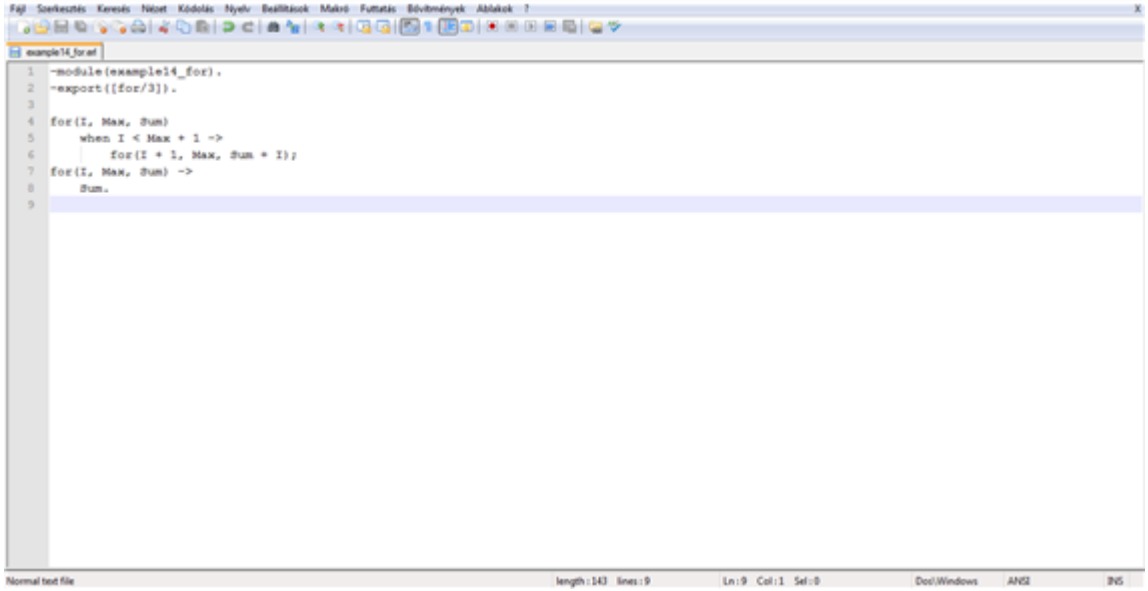


```
-module(example13_rectest).
-export([rec1/0,rec2/0,rec3/3]).
-record(ingatlan,{ar = 120000, megye, hely="Eger"}).

rec1() ->
    #ingatlan{}.
rec2() ->
    #ingatlan{ar = 110000, megye="Heves"}.
rec3(Ar, Megye, Hely) ->
    #ingatlan{ar = Ar, megye = Megye, hely = Hely}.
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example13_rectest:rec1().
{ingatlan,120000,undefined,"Eger"}
2> example13_rectest:rec2().
{ingatlan,110000,"Heves","Eger"}
3> example13_rectest:rec3(250000,"Nograd","Balassagyarmat").
{ingatlan,250000,"Nograd","Balassagyarmat"}
```

Ciklus készítése, összegzés

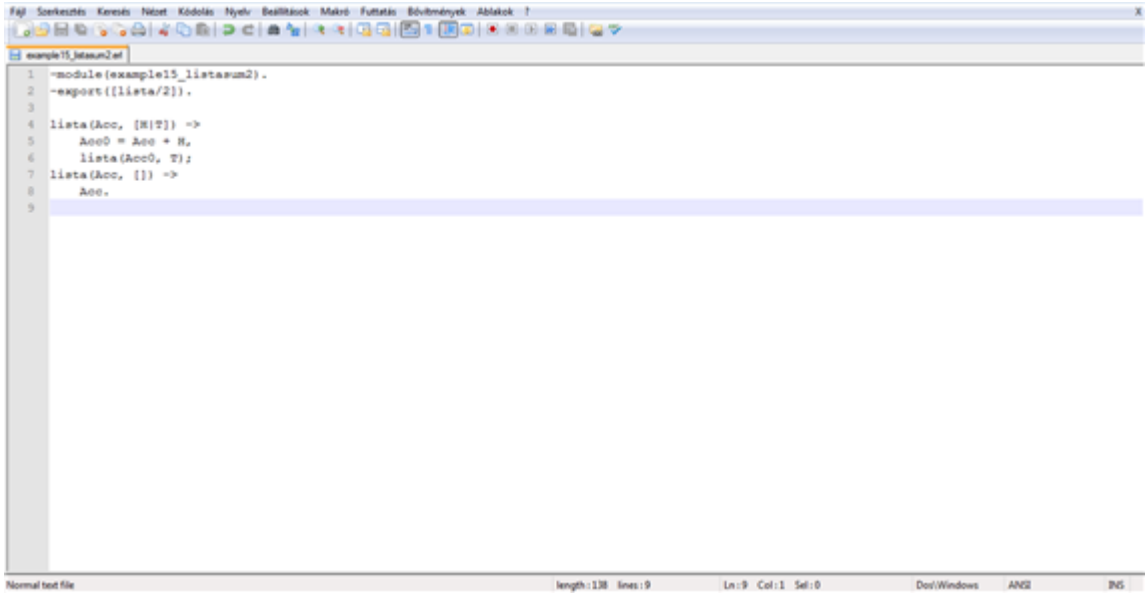


```
-module(example14_for).
-export([for/3]).

for(I, Max, Sum)
    when I < Max + 1 ->
        for(I + 1, Max, Sum + I);
for(I, Max, Sum) ->
    Sum.
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example14_for:for(0,5,0).
15
2> example14_for:for(0,10,0).
55
3> example14_for:for(8,10,0).
27
```

Lista összegzése 2

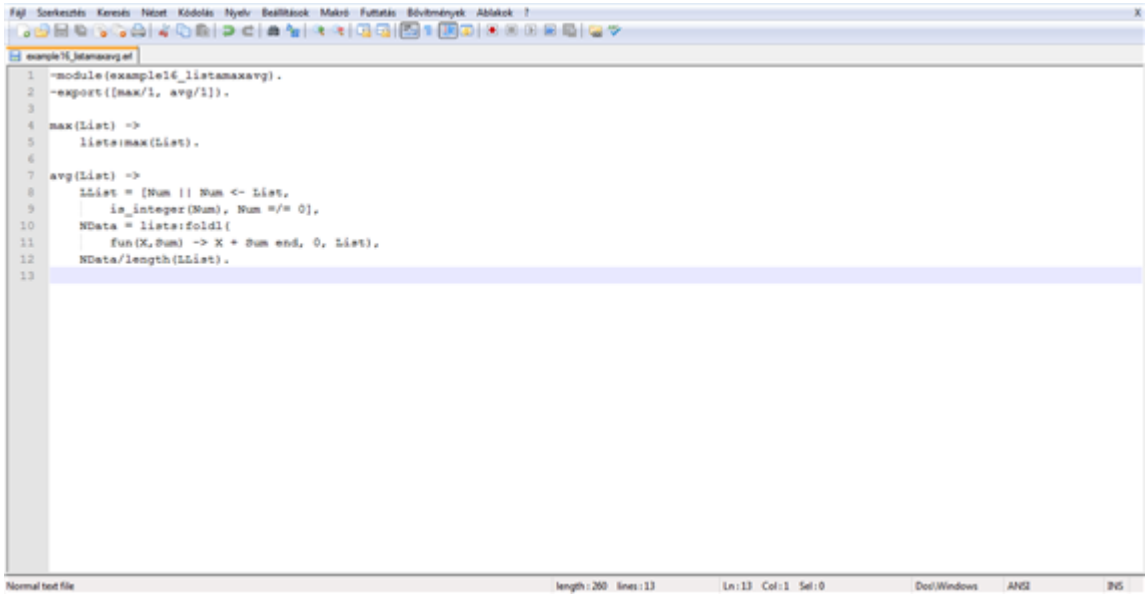


```
-module(example15_listasum2).
-export([lista/2]).

lista(Acc, [H|T]) ->
    Acc0 = Acc + H,
    lista(Acc0, T);
lista(Acc, []) ->
    Acc.
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example15_listasum2:lista(0,[3,27,10,5,97]).
142
2> example15_listasum2:lista(0,[-3,2,-10,1,20]).
10
```

Maximum, és átlag számítása

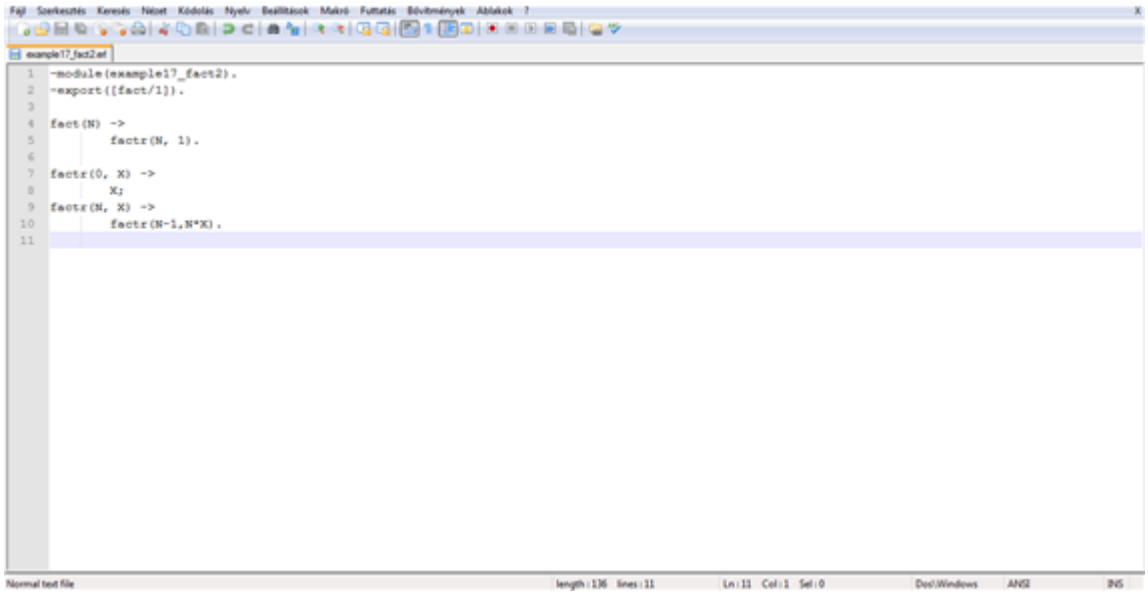


```
-module(example16_listamaxavg).
-export([max/1, avg/1]).

max(List) ->
    lists:max(List).

avg(List) ->
    LList = [Num || Num <- List,
        is_integer(Num), Num =/= 0],
    NData = lists:foldl(
        fun(X,Sum) -> X + Sum end, 0, List),
    NData/length(LList).
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example16_listamaxavg:max([3,50,16,92]).
92
2> example16_listamaxavg:avg([3,50,16,92]).
40.25
```

Faktoriális 2

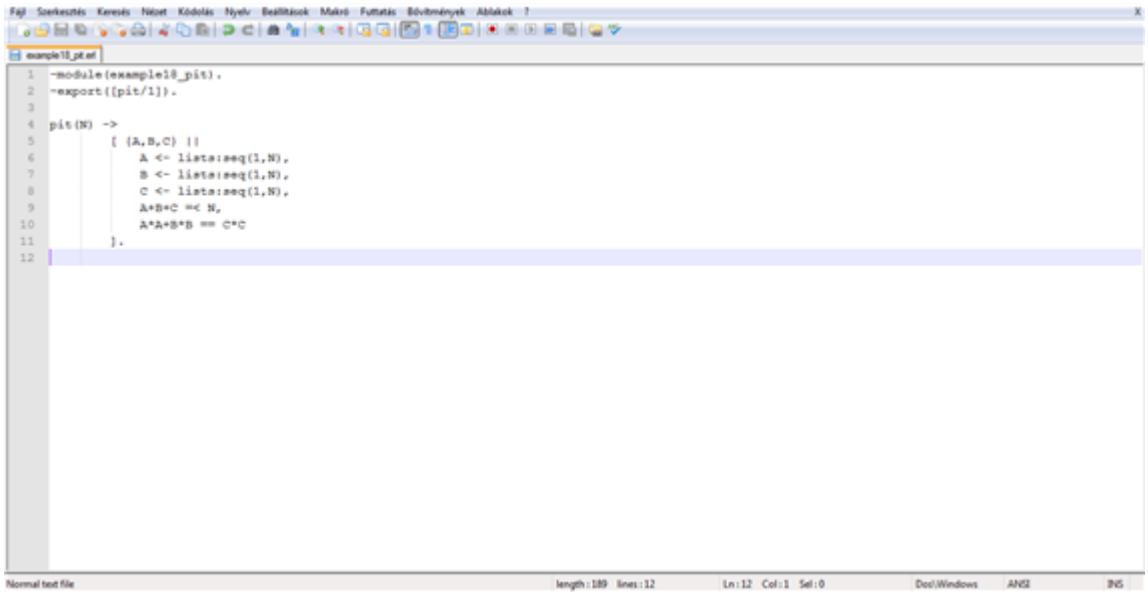


```
-module(example17_fact2).
-export([fact/1]).

fact(N) ->
        factr(N, 1).

factr(0, X) ->
        X;
factr(N, X) ->
        factr(N-1,N*X).
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example17_fact2:fact(6).
720
2> example17_fact2:fact(15).
1307674368000
```

Pythagoras

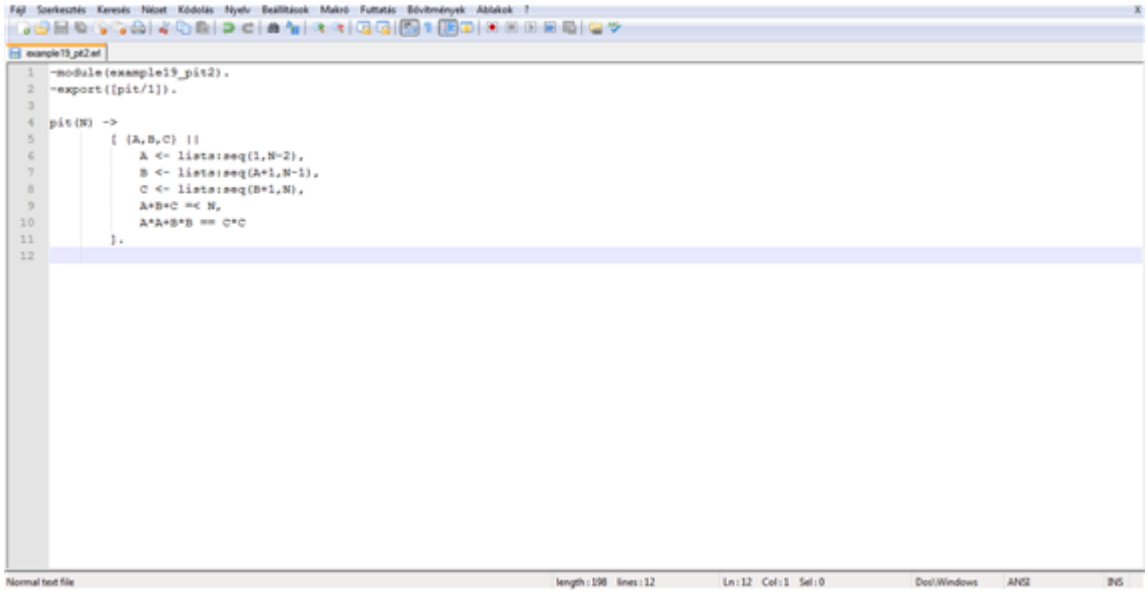


```
-module(example18_pit).
-export([pit/1]).

pit(N) ->
        [ {A,B,C} ||
           A <- lists:seq(1,N),
           B <- lists:seq(1,N),
           C <- lists:seq(1,N),
           A+B+C =< N,
           A*A+B*B == C*C
        ].
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example18_pit:pit(12).
[{3,4,5},{4,3,5}]
2> example18_pit:pit(24).
[{3,4,5},{4,3,5},{6,8,10},{8,6,10}]
3> example18_pit:pit(48).
[{3,4,5},
 {4,3,5},
 {5,12,13},
 {6,8,10},
 {8,6,10},
 {8,15,17},
 {9,12,15},
 {12,5,13},
 {12,9,15},
 {12,16,20},
 {15,8,17},
 {16,12,20}]
```

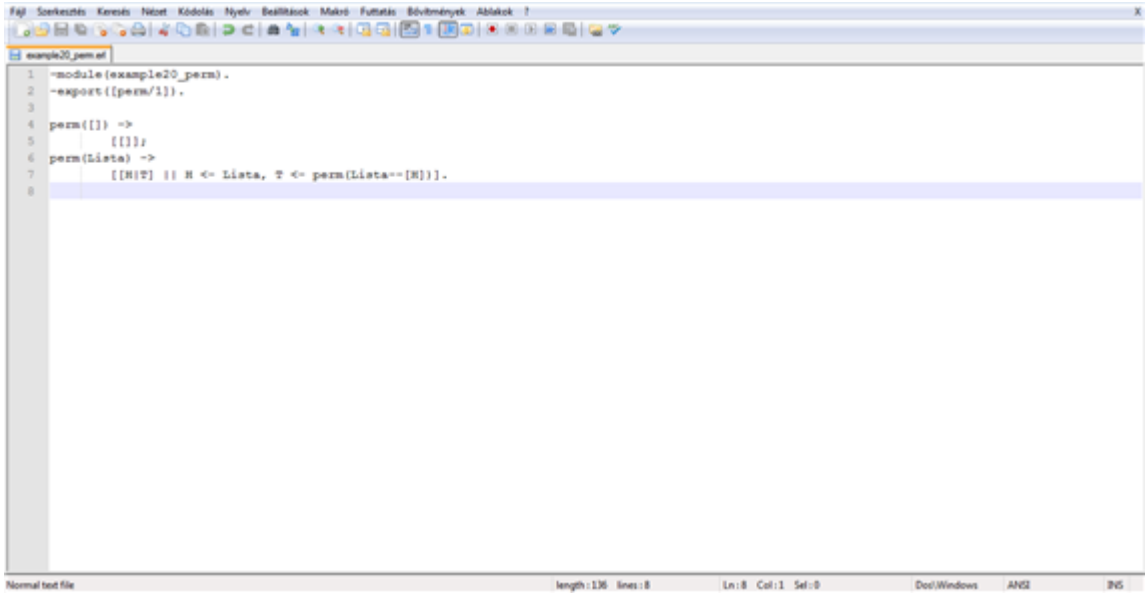
Pythagoras 2



A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example19_pit2:pit(12).
[{3,4,5}]
2> example19_pit2:pit(24).
[{3,4,5},{6,8,10}]
3> example19_pit2:pit(48).
[{3,4,5},{5,12,13},{6,8,10},{8,15,17},{9,12,15},{12,16,20}]
```

Permutáció

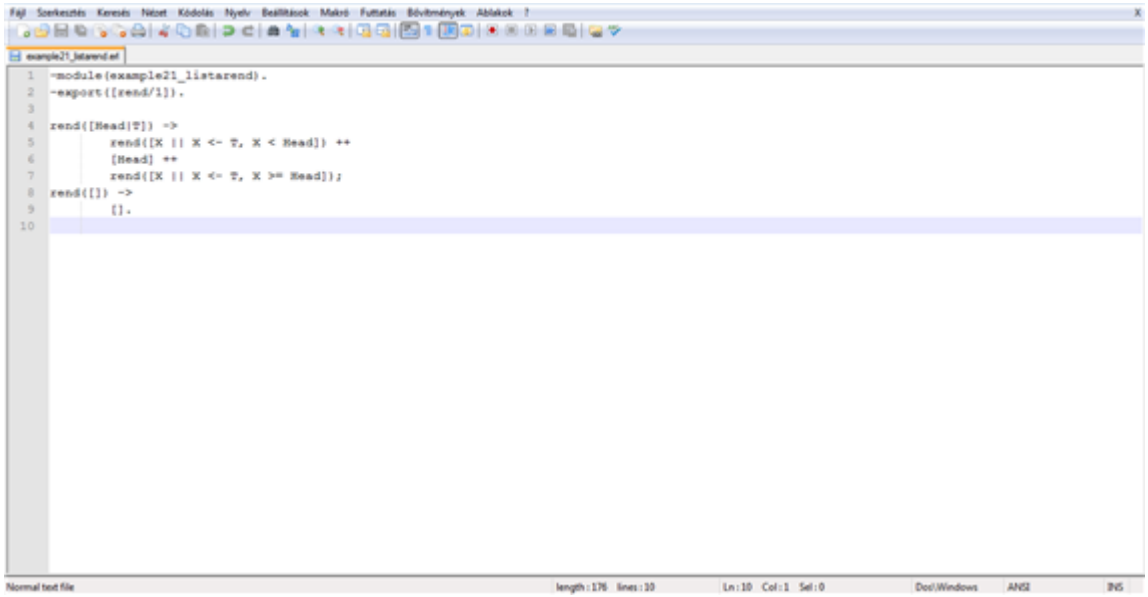


```
-module(example20_perm).
-export([perm/1]).

perm([]) ->
      [[]];
perm(Lista) ->
      [[H|T] || H <- Lista, T <- perm(Lista--[H])].
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example20_perm:perm([a,b,c]).
[[a,b,c],[a,c,b],[b,a,c],[b,c,a],[c,a,b],[c,b,a]]
2> example20_perm:perm([5,34,1]).
[[5,34,1],[5,1,34],[34,5,1],[34,1,5],[1,5,34],[1,34,5]]
```

Lista rendezése

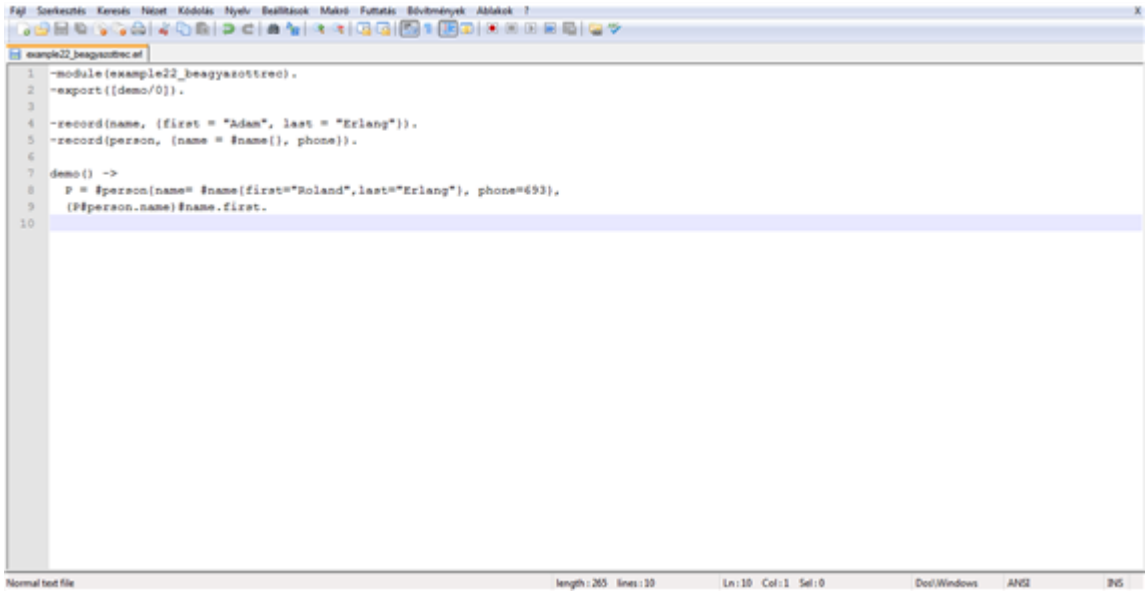


```
-module(example21_listarend).
-export([rend/1]).

rend([Head|T]) ->
      rend([X || X <- T, X < Head]) ++
      [Head] ++
      rend([X || X <- T, X >= Head]);
rend([]) ->
      [].
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example21_listarend:rend([92,5,20,12,0,67,-23]).
[-23,0,5,12,20,67,92]
2> example21_listarend:rend([g,e,a,b,d,z,q]).
[a,b,d,e,g,q,z]
```

Rekordok kezelése 2

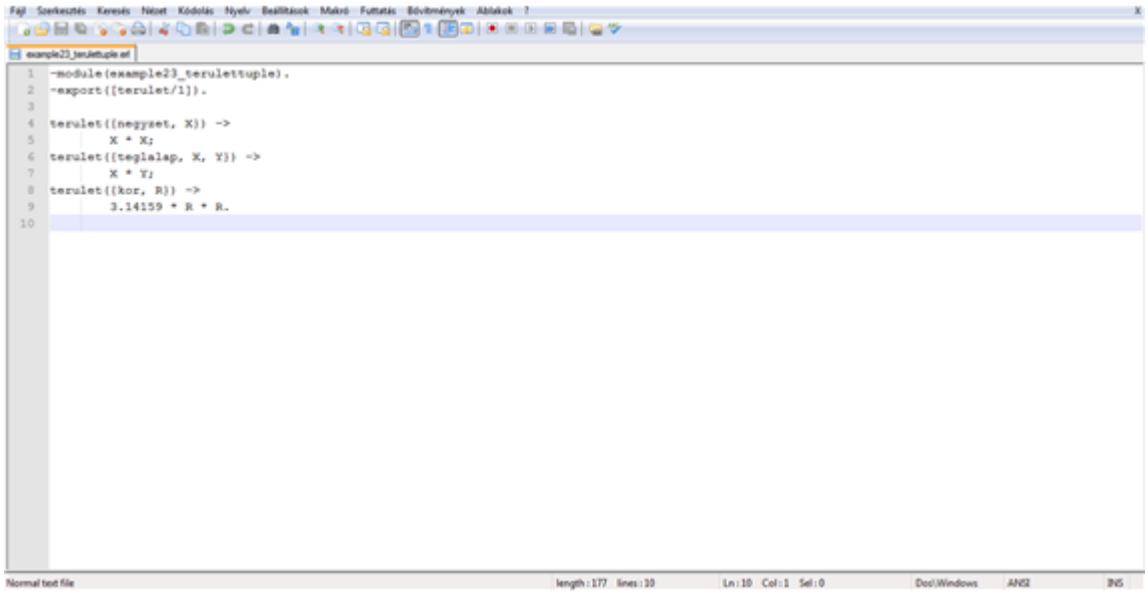


```
-module(example22_beagyazottrec).
-export([demo/0]).

-record(name, {first = "Adam", last = "Erlang"}).
-record(person, {name = #name{}, phone}).

demo() ->
  P = #person{name= #name{first="Roland",last="Erlang"}, phone=693},
  (P#person.name)#name.first.
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example22_beagyazottrec:demo().
"Roland"
```

Terület számítás

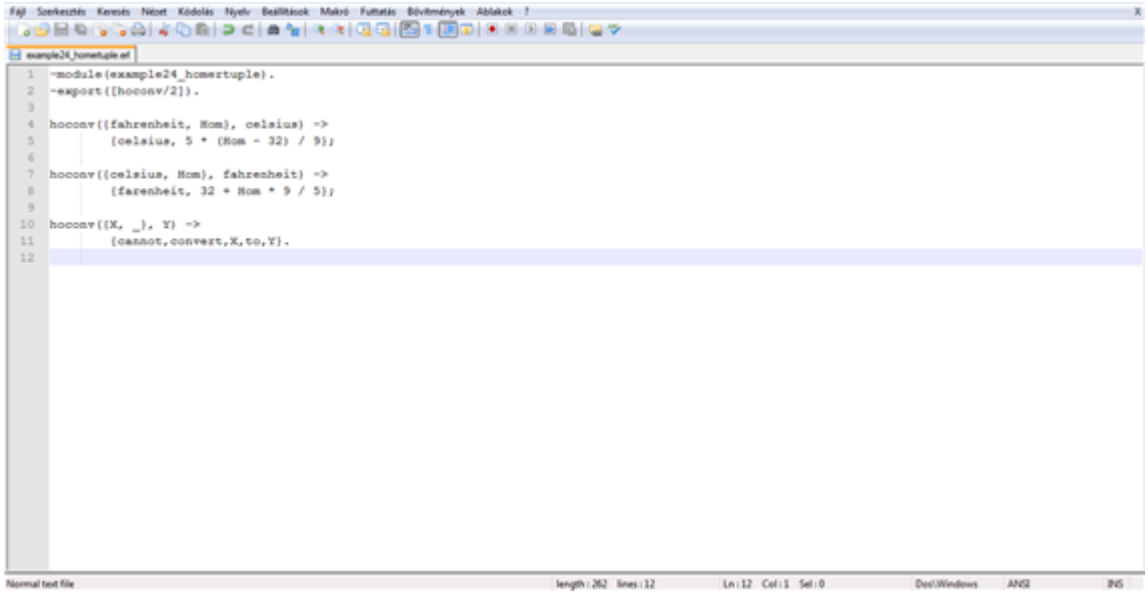


```
-module(example23_terulettuple).
-export([terulet/1]).

terulet({negyzet, X}) ->
        X * X;
terulet({teglalap, X, Y}) ->
        X * Y;
terulet({kor, R}) ->
        3.14159 * R * R.
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example23_terulettuple:terulet({negyzet,20}).
400
2> example23_terulettuple:terulet({teglalap,20,43}).
860
3> example23_terulettuple:terulet({kor,39}).
4778.358389999999
```

Hőmérséklet konvertálása

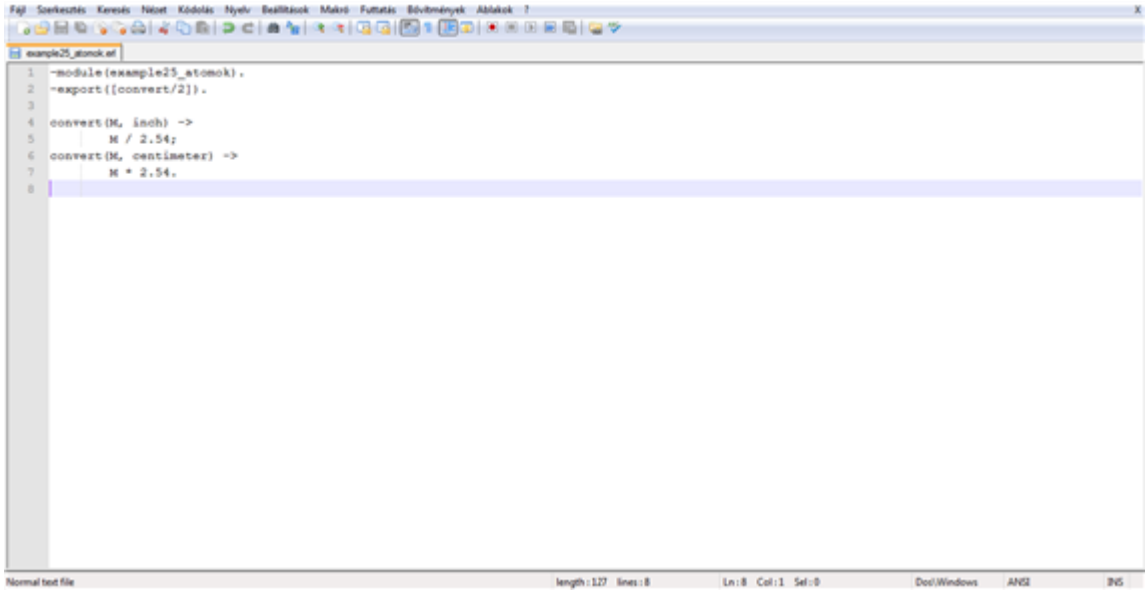


```
-module(example24_homertuple).
-export([hoconv/2]).

hoconv({fahrenheit, Hom}, celsius) ->
        {celsius, 5 * (Hom - 32) / 9};

hoconv({celsius, Hom}, fahrenheit) ->
        {farenheit, 32 + Hom * 9 / 5};

hoconv({X, _}, Y) ->
        {cannot,convert,X,to,Y}.
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example24_homertuple:hoconv({fahrenheit, 46}, celsius).
{celsius,7.777777777777778}
2> example24_homertuple:hoconv({celsius, 25}, fahrenheit).
{farenheit,77.0}
```

Atomok használata

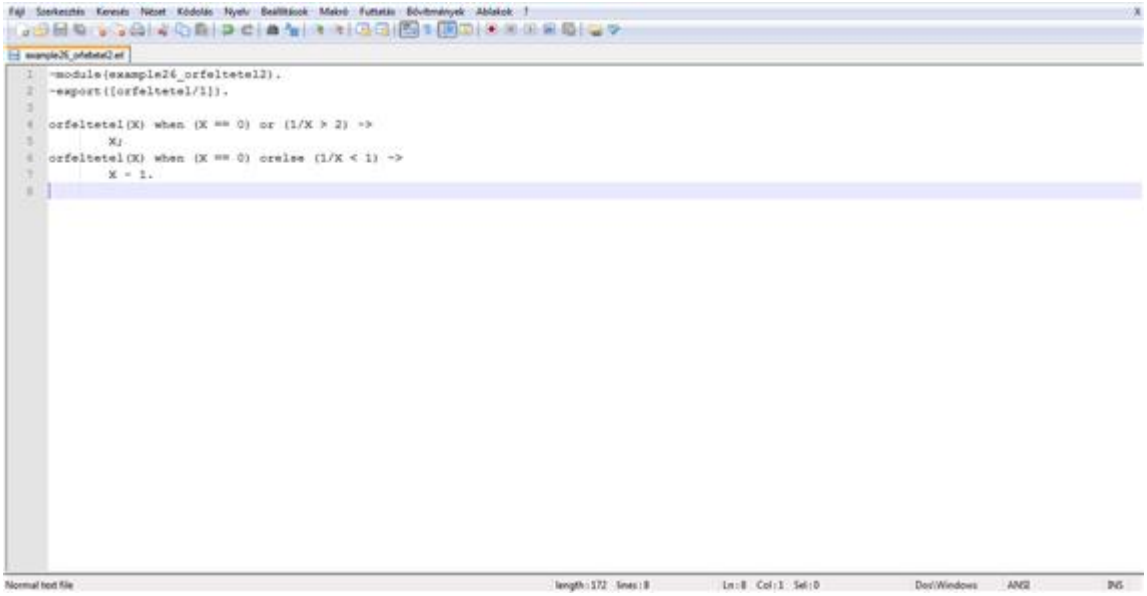


```
-module(example25_atomok).
-export([convert/2]).

convert(M, inch) ->
        M / 2.54;
convert(M, centimeter) ->
        M * 2.54.
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example25_atomok:convert(18, inch).
7.086614173228346
2> example25_atomok:convert(2, centimeter).
5.08
```

Őrfeltétel

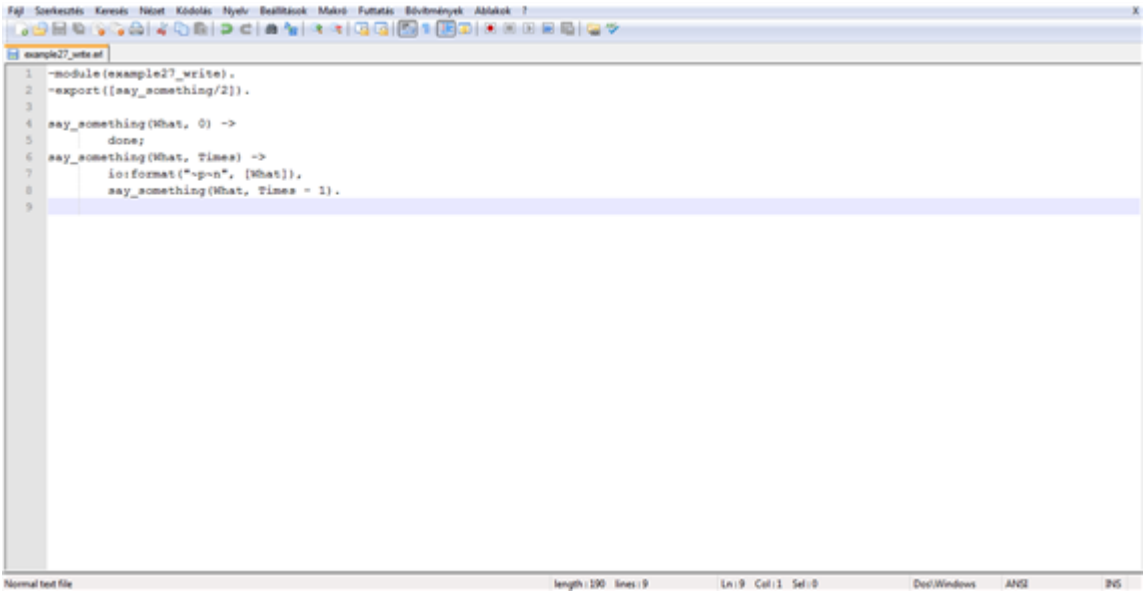


```
-module(example26_orfeltetel2).
-export([orfeltetel/1]).

orfeltetel(X) when (X == 0) or (1/X > 2) ->
        X;
orfeltetel(X) when (X == 0) orelse (1/X < 1) ->
        X + 1.
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example26_orfeltetel2:orfeltetel(0).
-1
2> example26_orfeltetel2:orfeltetel(63).
62
```

IO teszt

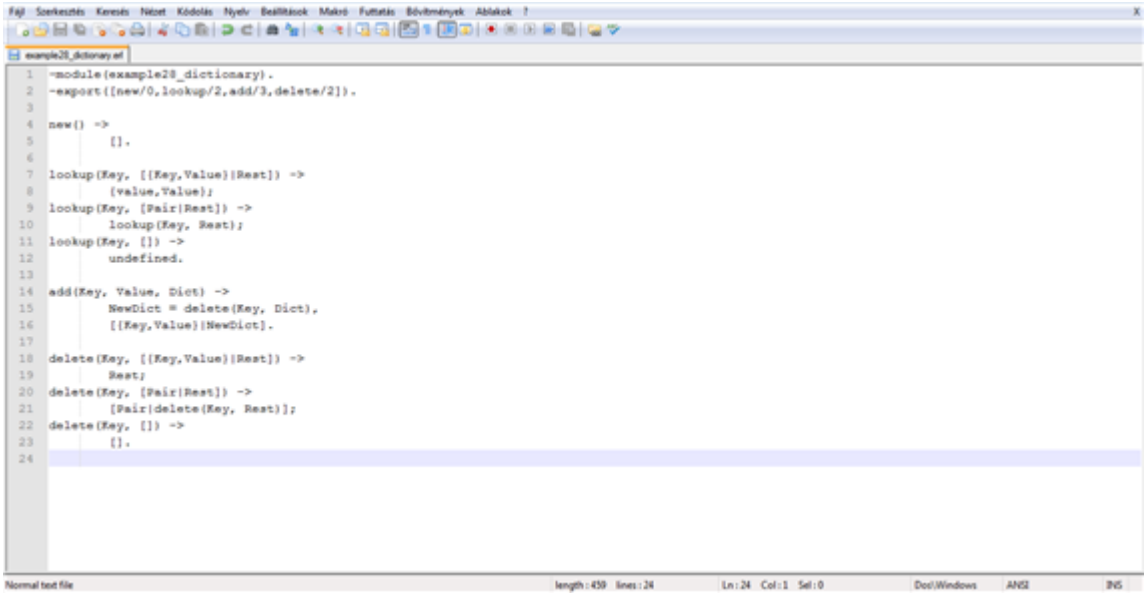


```
-module(example27_write).
-export([say_something/2]).

say_something(What, 0) ->
        done;
say_something(What, Times) ->
        io:format("~p~n", [What]),
        say_something(What, Times - 1).
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> example27_write:say_something(helloworld, 10).
helloworld
helloworld
helloworld
helloworld
helloworld
helloworld
helloworld
helloworld
helloworld
helloworld
done
```

Dictionary használata



```
-module(example28_dictionary).
-export([new/0,lookup/2,add/3,delete/2]).

new() ->
        [].

lookup(Key, [{Key,Value}|Rest]) ->
        {value,Value};
lookup(Key, [Pair|Rest]) ->
        lookup(Key, Rest);
lookup(Key, []) ->
        undefined.

add(Key, Value, Dict) ->
        NewDict = delete(Key, Dict),
        [{Key,Value}|NewDict].

delete(Key, [{Key,Value}|Rest]) ->
        Rest;
delete(Key, [Pair|Rest]) ->
        [Pair|delete(Key, Rest)];
delete(Key, []) ->
        [].
```

A feladat kimeneti képernyője

```
Eshell V5.9  (abort with ^G)
1> D0 = example28_dictionary:new().
[]
2> D1 = example28_dictionary:add(joe,42,D0).
[{joe,42}]
3> D2 = example28_dictionary:add(mike,41,D1).
[{mike,41},{joe,42}]
4> D3 = example28_dictionary:add(robert,43,D2).
[{robert,43},{mike,41},{joe,42}]
5> example28_dictionary:lookup(joe, D3).
{value,42}
6> example28_dictionary:lookup(helen, D3).
undefined
```

Bibliográfia

Programozási nyelvek, Volume.: Programozási nyelvek, Funkcionális programozási nyelvek elemei 2006., Szerk.: Nyékyné GaizlerJudit Horváth, Zoltán pages 589-636. ISBN 9-639-30147-7

Funkcionális programozás témakörei a programtervező képzésben 2005., Szerk.: PethőAttila HerdonMiklós Horváth, Zoltán Csörnyei, Zoltán Lövei, László Zsók, Viktória

Funkcionális programozás nyelvi elemei Technical University of Kosice, Kassa, Slovakia, 2005., Horváth, Zoltán

Refactoring in Erlang, a Dynamic Functional Language Proc. of 1st Workshop on Refactoring Tools (WRT07)}, ECOOP 2007 , ISSN 1436-9915, 2007., Lövei, László Horváth, Zoltán Kozsik, Tamás Király, Roland Víg, Anikó Nagy, Tamás DigDanny CebullaMichael

Refactoring in Erlang, a Dynamic Functional Language Central European Functional Programming School, volume 5161/2008, Lecture Notes in Computer Science, pages 250-285, 2008, 2008., Lövei, László Horváth, Zoltán Kozsik, Tamás Király, Roland Víg, Anikó Nagy, Tamás Tóth, Melinda Kitlei, Róbert

Lambda-kalkulus Typotex, Budapest, 2007., Csörnyei, Zoltán

Típuselmélet (kézirat) 2008., Csörnyei, Zoltán

\LaTeX\ kézikönyv Panem Könyvkiadó, 2004., Wett, Ferenc Mayer, Gyula Szabó, Péter

Functional Programming in Clean 2010., http://wiki.clean.cs.ru.nl/Functional_Programming_in_Clean

F# at Microsoft Research 2010., http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/

The Haskell language 2010., http://www.haskell.org/onlinereport/

Lisp Massachhussetc Institute of Technology - ISBN: 0-201-08329-9 Addison Wesley 8329 USA , 1981., Henry Winston, Patric Klaus, Bertold Horn, Paul

Bevezetés a Linux használatába ISBN: 9789639425002, 2005., Sikos, László

Szoftverfejlesztés Java EE platformon Szak Kiadó ISBN: 9789639131972 , 2007., Imre, Gábor

GNU Emacs 2010., http://www.gnu.org/manual/manual.html

Mnesia Database Management System 2011., http://www.erlang.org

The Eclipse Foundation open source community 2011., http://www.eclipse.org/