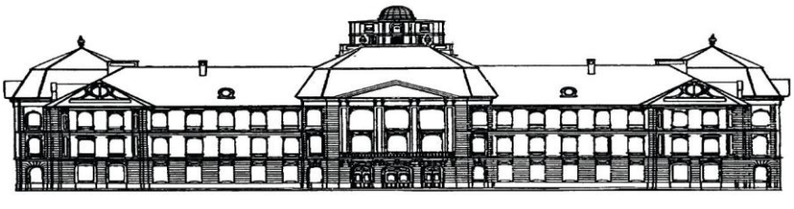
Funkcionális nyelvek

Funkcionális nyelvek

Király Roland



EKF • Eger, 2011

Kézirat lezárva: 2011. január 31.

Nemzeti Fejlesztési Ügynökség
www.ujszechenyiterv.gov.hu
06 40 638 638

MAGYARORSZÁG MEGÚJUL

A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

Tartalomjegyzék

A funkcionális programozási nyelvek világa

A funkcionális programozási nyelvek világa még a programozók között sem igazán közismert. Legtöbbjük az objektum orientált, valamint az imperativ 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 [12] ,a Clean [10], és az Erlang [6]. 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.

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

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

1.1 program Faktoriális függvény - Erlang

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:

fakt n = n * f akt n - 1

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

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

1.2 program Faktoriális függvény - Clean

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.

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

1.3 program Farok-rekurzív hívások

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 (1.3 programlista).

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

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

1.4 program Farok-rekurzív faktoriális függvény - Erlang

```
fact n = factr n 1

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

1.5 program Farok-rekurzív faktoriális függvény - Clean

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

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

1.6 program Farok-rekurzív faktoriális függvény - F#

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.

```
loop ( Data ) ->
        receive
```

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

1.7 program Rekurzív szerver - Erlang

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 1.7 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 tail-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 [6], a másik a Clean [10], a harmadik az F# [11]. 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 [12], 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.

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.

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 fej- lesztőeszköz, 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).

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, valamint 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.

Funkcionális programok általános jellemzői

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

Tiszta és nem tiszta nyelvek. A funkcionális nyelvek között léteznek tiszta, és nem tisztán funkcionális nyelvek A LISP [13] , az Erlang, az F#, és még néhány ismert funkcionális nyelv nem tisztán funkcionálisak, mivel a függvényeik tartalmaznak mellékhatásokat (valamint néhány nyelv esetében destruktív értékadást)

1.1 megjegyzés 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 -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.

1.2 megjegyzés A matematikai logikában, és a számítástudományban a Lambda, vagy -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 -kalkulussal definiálható függvény. Turing-kiszámítható [7], minden „kiszámítható” függvény leírható -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…

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

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 a logikai és imperatív nyelvek olyan nem funkcionális tulajdonságai is, mint a mellékhatás, könnyen leírhatók.

Funkcionális tulajdonságok

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.

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.

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.

Nem frissíthető változók. A funkcionális nyelvekre jellemző, de az OO programozók körében legkevésbé kedvelt technika a nem frissíthető változók használata. A destruktív értékadás nem engedi meg a változók többszöri kötését, vagyis az I = I + 1 típusú értékadásokat.

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.

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.

Magasabb rendű függvények. Funkcionális nyelvekben - 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 függvényeket, majd az így megkonstruált változókat később függvényként használjuk. A 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.

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.

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 [7] 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.

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 [3] 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.

Alapvető Input-Output

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.

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 [14] az egyik legelterjedtebb szerkesztő program az Emacs [16] Ezt a programot nem kezelhetjük szövegszerkesztőként, mivel alkalmas szinte minden ismert nyelv, még a Tex [9] alapú nyelvek kezelésére is. Egy másik érdekes rendszer az Eclipse [18], ami inkább Java [15] 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.

Clean. A Clean [10] 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.

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

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.

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 2.2 megjegyzés. 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!

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

2.1 program Változók összeadása parancssori programmal

A 2.1 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.

2.3 megjegyzés Az f() (2.2 példa) paraméterek nélküli meghívása az összes változót felszabadítja, és minden, a változókban tárolt adat elveszik…

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

2.2 program Változók felszabadítása

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 lefuttatuk 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.

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

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

2.3 program Összadó függvény modulja

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.

2.4 megjegyzés 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.

2.5 megjegyzés 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.

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

2.4 program Parancssori fordítás

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 (2.4) 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.

> f(A).
> f(B).
> f( ).
2.5 program Változók felszabadítása

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 pro ject 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.
-

module sajatmodul
import StdEnv

// i d e k e r ü l a s a j á t kó d

…

..

.

2.6 program Clean példaprogram

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 2.7 listában található sorokat az interaktív ablakba.

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

2.7 program F# példaprogram

Az interaktív ablakban a 2.8 listában látható sor jelenik meg

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

2.8 program Interaktív ablak

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.

```
module sajatModul

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

2.9 program F# module

A parancssori fordító használata mellett a programot egy fájlban kell elhelyeznünk, és azt a 2.9 listában található minta alapján kell elkészítenünk (több modul használata esetén a module kulcsszó kötelező)

Mellékhatások kezelése

Vizsgáljuk meg újra az Erlang nyelven készült sum/2 nevű függvényt (2.3 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. A Clean, az Erlang és az F# nyelvek nem tisztán funkcionálisak, mint a Haskell [12], 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.

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

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

2.10 program Mellékhatásos függvény - Erlang

A 2.10 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 mellékhatásos, 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.

Az adatok kezelése

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 formáknál 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.

Adatok tárolása. Funkcionális nyelvek használata mellett az adatainkat tárolhatjuk változókban, a változókat listákban, vagy rendezett n-esekben (tuple), és természetesen fájlokban, valamint adatbázis kezelők tábláiban.

3.1 megjegyzés Számos funkcionális nyelvi környezetben találunk az adott nyelvhez optimalizált adatbázis kezelő modulokat, vagy önálló rendszereket. Az Erlang ismert adatbázis kezelője a MNESIA [17], és a hozzá tartozó lekérdező nyelv a QLC [17]…

A fájl és adatbázis alapú tárolással nem foglalkozunk részletesen, de az említett három adatszerkezetet mindenképpen meg kell vizsgálnunk ahoz, 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 (3.2 programlista).

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

3.2 megjegyzés 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…

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

3.1 program Erlang 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 elemmel
```

3.2 program F# változók

Kifejezések

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 4.1 lista bemutatja a teljesség igénye nélkül az operátorok használatát mind a numerikus, mind logikai érékekre alkalmazva.

```
> 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}
```

4.1 program Operátorok használata - Erlang

A 4.2 lista azokat az operátorokat tartalmazza, melyeket a különböző kifejezések írásakor használhatunk.

4.1 megjegyzés 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”...

(unáris) +, (unaris) -, bnot, not

/, *, div, rem, band, and (bal asszociatívak)

+, -, bor, bxor, bsl, bsr, or, xor (bal asszociatívak)

++, -- (jobb asszociatívak)

==, /=, =<, <, >=, >, =:=, =/=

andalso

orelse

4.2 program Operátor precedencia - Erlang

!

o !! %

^

* / mod rem

+ - bitor bitand bitxo r

++ +++

== <> < <= > >=

&&

||

:=

‘bind‘

4.3 program Operátor precedencia - Clean

-, +, %, %%, &, &&, !, ~ (prefix operátorok, balasszociatívak)

*, /, % (bal asszociatívak)

-, + (bináris , balasszociatívak)

<, >, =, |, & (bal asszociatívak)

&, && (balasszociatívak)

o r, || (bal asszociatívak)

4.4 program Operátor precedencia - F#

4.2 megjegyzés 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...

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.

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

4.5 program Overloading OO nyelvekben

A 4.5 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 (4.6)

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

4.6 program Túlterhelt metódus alkalmazása

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

4.3 megjegyzés 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.

```
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.
```

4.7 program Mintaillesztés case-ben - Erlang

```
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#-b an
                // ez a nyelv nem ennyire megengedő
        | _ -> ( "error", A) // Itt szintén baj van
        …
```

4.8 program Mintaillesztés case-ben - F#

A mintaillesztést használja ki a case elágazás (4.7 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.

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

4.9 program Üzenetek fogadása - Erlang

A mintaillesztést tehát nem csak függvényekben, elágazásokban, vagy üzenetek fogadására, hanem változók, tuple, lista, és rekord adatszerkezetek kezelésére is felhasználhatjuk A 4.10 programlista a minták további alkalmazását bemutató példákat, és a példákhoz tartozó magyarázatokat tartalmazza.

```
{ A, abc } = { 123, abc }
                %A = 123,
                %abc illeszkedik az abc atomra
{ A, B, C } = { 123 , abc, bca }
                %A = 123, B = abc, C = bca
                %a mintaillesztéssikeres
{ A, B } = { 123, abc, bca }
                %Hibás illeszkedés,
                %mert nincs elegendő elem
                %a jobb 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 ]
```

4.10 program Mintaillesztések - Erlang

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.

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

4.11 program Őr feltétel függvény ágaihoz (clause) - Erlang

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

4.12 program Őr feltétel függvény ágaihoz (clause) - Clean

```
let max x y =
        match x, y w ith
        | a, b when a >= b -> a
        | a ,b when a < b -> b
```

4.13 program Őr feltétel függvény ágaihoz (clause) - F#

A 4.11 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.

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

4.14 program Összetett őr feltételek - Erlang

```
f x
```

```
| x == 0 || 1/x > 2 =
…
```

4.15 program Összetett őr feltételek - Clean

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 4.14.

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 verge. 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 (4.16 lista)

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

4.16 program Az if kifejezés általános formája - Erlang

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

4.17 program Az if kifejezés általános formája Clean nyelven

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

4.18 program F# példa if kifejezésre (legnagyobb közös osztó)

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

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 (4.19 programlista), és az egyes ágakat vezeti be a case.

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

4.19 program A case kifejezés a C alapú nyelvekben

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 4.20 listában) 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 4.4 megjegyzés. 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...

```
case Expr of
        Pattern1 -> Expr_seq1;
        Pattern2 -> Expr_seq2;
        …
end
```

4.20 program A case kifejezés általános formája Erlang nyelven

```
patterns Expr
```

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

4.21 program A case kifejezés általános formája Clean nyelven

```
match Expr w ith
| Pattern1 -> Expr_seq1
| Pattern2 -> Expr_seq2
```

4.22 program A case kifejezés általános formája F# nyelven

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 (4.23 példaprogram).

```
...
case Expr1 of
        Pattern1_1 -> Expr_seq1_1;
        Pattern1_2 -> case Expr2 of
                        Pattern2_1 -> Expr_seq2_1;
                        Pattern2_2 -> Expr_seq2_2;
                        end
end,
…
```

4.23 program Egymásba ágyazott case kifejezések Erlang 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
```

4.24 program Egymásba ágyazott case kifejezések Clean nyelven

```
match Expr w ith
| Pattern1_1 -> Expr_seq1_1
| Pattern1_2 -> match Expr2 w ith
        | Pattern2_1 -> Expr_seq2_1
        | Pattern2_2 -> Expr_seq2_2
```

4.25 program Egymásba ágyazott case kifejezések F# nyelven

4.5 megjegyzés A 4.25 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 melyik minta melyik match -hez tartozik...

A 4.26 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.

4.6 megjegyzés Az angol terminológia a függvény ágakra a clause, míg 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

```
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
```

4.26 program Elágazások használata - Erlang

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 .).

4.7 megjegyzés 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

```
try függvény / kifejezés of
        Minta [ when Guard1 ] -> blokk;
        Minta [ when Guard2 ] -> blokk;
        …
catch
        Exceptiontípus:M inta [ when ExGuard1 ] -> utasítások;
        Exceptiontípus:M inta [ when ExGuard2 ] -> utasítások;
        …
after
        utasítások
end
```

4.27 program Try-catch blokk - Erlang

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

4.28 program Kivételkezelés, try-finally blokk F# nyelven

A hibák kezelésének egyik ismert módja a több ággal (clause) rendelkező függvények használata, a másik az adott nyelvben rendelkezésre álló kivétel kezelők alkalmazása 4.27.

A több ággal rendelkező függvények a hibák egy kis részét megoldják, ahogy ezt a 4.29

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)

4.8 megjegyzés 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.

4.9 megjegyzés 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…

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

4.29 program Több féle paraméter kezelése - Erlang

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 (4.30).

```
kiir( Data ) ->
        try
                io:f ormat( "~ s~n", [ Data ] )
        catch
                H1:H2 ->
                io:fo rmat ( "H1:~w~n H2:~w~n Data:~w~n",
                                                [H1 , H2 , Data ] )
        end
```

4.30 program A kivétel okának felderítése a catch blokkban - Erlang

A 4.30 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.

4.10 megjegyzés 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…

A 4.30 programban látható kivételkezelő ekvivalens a 4.31 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.

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

4.31 program Kivételek kezelése mintákkal – Erlang

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

4.32 program Példa kivételkezelésre F# nyelven

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.

```
try
        raise Invalid Process ( “Raising_Exn" )
wit h
        | InvalidProcess( str ) -> printf "%s \n" str
```

4.33 program Kivételek dobása F# nyelven

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

Összetett adatok

Rendezett n-esek

Tuple. 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.

5.1 megjegyzés 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 (5.1 programlista).

– {A, B, A + B} egy rendezett n-es, ahol a két első elem tartalmazza egy összedó 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 tar- talmazza.

5.2 megjegyzés Arra hogy az átadott függvény látszólag csak egy változó, a -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 5.1 programot, amely egy rendezett n-est használ az adatok megjelenítésére.

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

teszt( ) ->
```

```
    io:format ( "~d~n", [ osszeg ( 10, 20 ) ] ).
```

5.1 program Tuple használata Erlang nyelven

```
module osszeg
import StdEnv

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

teszt = osszeg 10 20

Start = teszt
```

5.2 program Tuple használata Clean nyelven

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

5.3 program Tuple készítése F# nyelven

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-eseknél. 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.

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

osszeg( { A, B } ) -> A + B;
osszeg( [ A, B ] ) -> A + B;
osszeg(_) ->
    0.
teszt( ) -> io:fo rmat( "~d~n~d~n", [ osszeg ( { 10, 20 } ), osszeg ( [ 10, 20 ] ) ] ).
```

5.4 program Tuple több függvény ágban - Erlang

A 5.4 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 rendkívűl 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.

Rekord

A rekord a tuple adatszerkezet névvel ellátható változata olyan könyvtá- ri függvényekkel „felvértezve”, melyek a mezők, vagyis a rekord adatainak elérését segítik.

{szemely, Nev, Eletkor, Cim}

5.5 program Nevesített tuple – Erlang

Az 5.6 programlistában egy tipikus rekord szerkezetet láthatunk, melynek a neve mellett van két 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 (5.6 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 (5.14).

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

5.6 program Rekord – Erlang

```
:: Ingatlan = { ar :: Int,
                megye :: String,
                  hely :: String
```

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

5.7 program Rekord Clean nyelven

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

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

5.8 program F# rekord definíció

A 5.6 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 az 5.9 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.

5.3 megjegyzés 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.

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

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

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

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

5.9 program Tuple és rekord - Erlang

```
type ingatlan = { ar : int; megye : string; hely : s t r i ng }

letrec1 =
        let R1 = { ar = 12000; megye = "Heves"; hely = "Eger" } R1

letrec2 ar megye hely =
        letR2 = { ar = ar; megye = megye; hely = hely } R2
```

5.10 program Tuple és rekord - F#

5.4 megjegyzés 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…

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

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

5.11 program Rekord update - Erlang

```
module rectest
import StdEnv

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

tulaj1 :: Tula j
tulaj1 = { tne v = "Egy_Tula j", eletkor = 10 }
```

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

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

5.12 program Rekord update - Clean

```
type kedvenc = {  nev : s t r in g; tulaj : string }
let writerec ( r : kedvenc ) = printfn "%s" r.ne v
```

5.13 program Rekord update - F#

Természetesen a rekord mezőire egyesével is lehet hivatkozni, ahogy azt az 5.11 programlistában láthatjuk. A writerec/1 függvény egy rekordot kap paraméterként, majd az első mezőjét kiírja a képernyőre.

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

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

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

5.14 program Rekord update - Erlang

Az 5.14 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.

Függvények és rekurzió

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.

```
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
```

6.1 program Erlang függvények általános definíciója

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

6.2 program Clean függvények általános definíciója

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

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

//Rekurzív
```

```
let rec funName Param2_1 …
        Param2_N [ : return-ty pe ] = FunBody2
```

6.3 program F# függvények általános definíciója

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# nyelvben nincs ilyen megkötés. A függvény törzse 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.

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

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

h () ->
        ok.
```

6.4 program Erlang függvények

```
f p1 p2 = p1 + p2.
g x y = f x y
h = "ok"
```

6.5 program Clean függvények

```
let f p1 p2 = p1 + p2
let g x y = f x y
let h = "ok"
```

6.6 program F# függvények

Rekurzív függvények

Rekurzió. 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.

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

6.7 program Imperatív do-while ciklus

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

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

6.8 program F# do-while ciklus

Rekurzív ismétlések. A 6.9 programban láthatjuk, hogy miként lehet a 6.7 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.

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

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

6.9 program Ismétlés rekurzióval – Erlang

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

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

6.10 program Ismétlés rekurzióval - F#

Ez egy igen sa játos megközelítése a rekurziónak, de nem példa nélküli, ezért szerepeljen itt is egy megvalósítása (6.11 programlista).

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

6.11 program Összegzés C nyelven

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

6.12 program Összegzés Erlang nyelven

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

6.13 program Összegzés F#-ban

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

6.14 program Összegzés F#-ban rekurzívan

Magasabb rendű függvények. A magasabb rendű függvények talán a legérdekesebb konstrukciók a funkcionális nyelvek eszköztárában. Segítségükkel függvény prototípusokat - igazság szerint 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.

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

6.2 megjegyzés 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.

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

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

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

6.15 program Függvény kifejezések - Erlang

```
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 ) "i nc"
```

6.16 program Függvény kifejezések - F#

A 6.15 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.

6.3 megjegyzés 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 6.15 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
6.4 megjegyzés 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 6.17 programban.

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

funct( Fun, Data ) ->
```

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

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

6.17 program Magasabb rendű függvények modulja - Erlang

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

let funct1 fn data = fn data

let funct2 fn data List =
    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 )

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

6.18 program Magasabb rendű függvények modulja - F#

A magasabb rendű függvényeket - az „egyszerű” függvények mintájára – egymásba ágyazhatjuk (6.20 példa), valamint ezek is rendelkezhetnek több ággal, ahogy ezt az egyszerű függvényeknél láthattuk.

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

6.19 program Egymásba ágyazott függvények - Erlang

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

6.20 program Egymásba ágyazott függvények - F#

Függvény kifejezések A függvényekre hivatkozhatunk a nevükből, és az aritásukból képzett párossal is Az ilyen függvény kifejezésekre (6.23 programlista) tekinthetünk úgy, mint egy függvényre mutató referenciára.

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

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

6.21 program Függvény referencia - Erlang

```
summa a b = a+b

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

6.22 program Függvény referencia - Clean

A 6.23 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.

```
let sum a b = a + b

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

6.23 program Függvény referencia - F#

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 (6.23), függvény kifejezést használó változatot úgy, hogy a caller/0 függvényben szereplő F = sum/2 kifejezést kibontjuk.

```
-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 ).
```

6.24 program Kibontott függvény kifejezés – Erlang

A 6.24 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 üzene tek 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 (6.25 programlista).

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

6.25 program Függvény hívás üzenetküldéssel - Erlang

Máskor a szerver tartalmazza a futtatni kívánt függvényeket, de a kliens választja ki, hogy aktuálisan melyiket kell futtatni Ekkor a szervert névaritás 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 törénő küldésére, és futtatására.

Listák és halmazkifejezések

A lista adatszerkezet lényegesen bonyolultabb mint a tuple, de cserébe a bo- nyolultságáért 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 [3]. 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.

V = {x |x X, x > 0} 2

7.1 megjegyzés 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 7.1 programlistában látható általános szintaxissal írhatók le.

```
[ E || Qualif ier_1, Qualifie r_2, … ]
```

7.1 program Lista szintaxisa - Erlang

A 7.2 programszöveg bemutatja a statikus listák definiálásának, valamint kezelésének a módját. A programrészlet listákat köt változókba, majd azokat más listákban újra felhasználja.

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

7.2 program Listák kötése változóba – Erlang

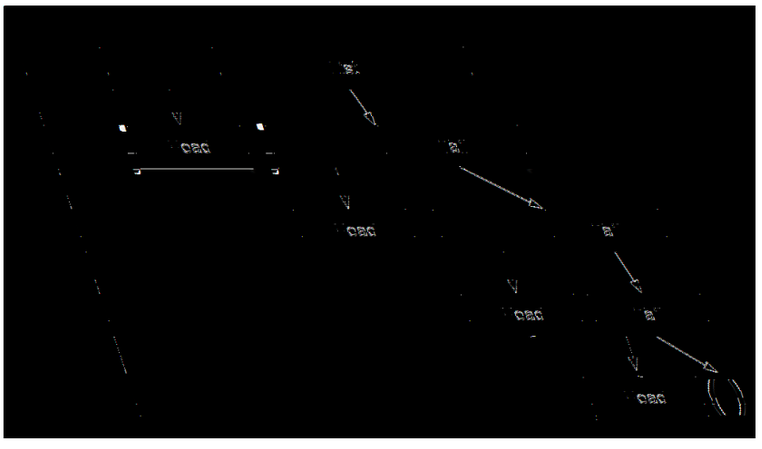
```
let Adatok = [ ( 10, 20 ); ( 6, 4 ); ( 5, 2 ) ]
let Lista = [ A, B, C, D ]
let L = [ Adatok, Lista ]
```

7.3 program Listák kötése változóba F#

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.

Statikus listák kezelése

Listák feldolgozása 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|T ail] = T ail) lehetővé teszi a lista rekurzív bejárását. A 7.4 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.



7.1 ábra Listák bejárása

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

7.4 program Mintaillesztés listákra – Erlang

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

7.5 program Mintaillesztés listákra - F#

7.2 megjegyzés Vegyük észre, hogy a 7.4 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.

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

```
listabejaras( Acc, [] ) ->
        Acc.
```

7.6 program Lista bejárása rekurzív függvénnyel - Erlang

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

7.7 program Lista bejárása rekurzív függvénnyel - Clean

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

7.8 program Lista bejárása rekurzív függvénnyel - F#

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.

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

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

7.9 program Lista elemeinek összegzése – Erlang

```
module list1
import StdEnv
sum [ head : t a i l ] = head + sum tail
```

```
sum [] = 0
```

7.10 program Listafeldolgozás - Clean

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

7.11 program Listafeldolgozás - F#

A 7.9 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.

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

7.4 megjegyzés Amennyiben a paraméterként megadott lista nem számokat tartalmaz, a függvény akkor is működőképes, de mindenképpen olyan típusú elemeket kell tartalmaznia, amelyekre értelmezhető a + operator…

Készítsük el a sum/2 függvényt implementáló modult, fordítsuk le, majd hívjuk meg parancssorból (7.12 programlista).

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

7.12 program Összegzés programjának futtatása - Er- lang

```
modul lista1
import StdEnv
```

```
osszeg [ he ad : tail ] = head + osszegtail
osszeg [] = 0

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

7.13 program Összegzés futtatása – Clean

```
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
```

7.14 program Összegzés futtatása az F# interaktív ablakban

Az összegzés programja egyszerű műveletet implemental. 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.

```
-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( RList ).

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 ).
```

7.15 program Listák kezelése - Erlang

```
module functions
import StdEnv

osszeg [ he ad : 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
```

7.16 program Listák kezelése – Clean

```
let rec osszeg acc list =
        match list w ith
        | 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
```

7.17 program Listák kezelése - F#

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 (7.18 programlista).

```
-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, Ta il );
comp2( List, [] ) ->
        List.

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

7.18 program Listák előállítása és összefűzése - Erlang

```
module functions
import StdEnv

comp1abc = [ a, b, c ]

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

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

7.19 program Listák előállítása és összefűzése - Clean

A 7.18 példamodul három függvényt tartalmaz A comp1/2 egyetlen sora a paraméterként érkező hármas elemeit egy listába helyezi.

```
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 wit h
    | head :: t a i l -> comp3 fn ( list1 @ [ fn head ] ) tail
    | [] -> list1
```

7.20 program Listák előállítása és összefűzése - F#

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 7.21 programlista tartalmazza. A példa azért kapott helyet a fejezetben, hogy megmutassuk a magasabb rendű függvények egyik gyakori felhasználását.

```
-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 ( f un( X ) -> X + 1 end, [], List ).
```

7.21 program Függvény általánosítása - Erlang

```
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 ]
```

7.22 program Lista magasabb rendű függvénnyel – Clean

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

```
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
```

7.23 program Lista magasabb rendű függvénnyel - F#

Amennyiben futtatni szeretnénk ezt a programot, fordítsuk le, majd hívjuk meg a use/0 függvényt a modul-minősítővel (7.24 programlista).

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

7.24 program A use/0 meghívása - Erlang

```
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

7.25 program Use meghívása - Clean

Ha jobban megnézzük a 7.21 és a 7.18 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.
7.5 megjegyzés A függvények általánosítása hatékonyabbá, és széles körben alkalmazhatóvá teszi a programjainkat. Amikor csak lehetőségünk van rá, készítsünk minél általánosabban felhasználható függvényeket…

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 (7.26 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.

```
FList = [ 1, 2, 3, 4, 5, 6 ],
Lista = [ SelectedItems | A <- FList, A /=  0 ] …
```

7.26 program Listafeldolgozás lista-kifejezéssel - Erlang

A 7.26 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.

```
let fLista = [ 1; 2; 0; 3; 4; 5; 6 ]
let lista = List.filter( f un a -> a <> 0 ) fLista
```

7.27 program Listafeldolgozás lista-kifejezéssel - F#

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 (7.28 programlista).

```
Lista = [ begin f1( Elem ), f2( Elem ) end
                || Elem <- KLista, Elem >= 0, Ele m < 10 ]
```

7.28 program Begin-end blokk használata lista kifejezésben - Erlang

```
let lista =
    let kLista = List.filter
        ( fun elem -> elem >= 0 && ele m < 10 ) kLista
    let kLista = List.map f1 kLista
    let kLista = List.map f2 kLista
    kLista
```

7.29 program Begin-end blokk - F# „legközelebbi megoldás”

A 7.28 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 <- 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”.

```
List2 = [ begin filter ( Elem ), add( Elem ) end ||
                Elem <- List1, Elem > 0, Elem < 100 ]
```

7.30 program Összetett feltételek lista kifejezésben – Erlang

A 7.30 példaprogramban elhelyeztünk két feltételt is, ami 1 és 100 között szűri a bemeneti lista elemeit A felsorolt feltételek sorban érvényesülnek, közöttük ÉS kapcsolat van.

```
let list2 = List.filter
    ( fun elem -> ele m > 0 && elem < 100)
```

7.31 program Összetett feltételek lista kifejezésben - F#

7.6 megjegyzés 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.

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

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

7.32 program Lista előállítása függvénnyel - Erlang

```
let add elem value = elem + value

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

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

7.33 program Listák feldolgozása függvénnyel - F# tömör változat

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 7.32 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.

7.7 megjegyzés 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, ma jd 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 7.34 szövegben látható listát 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.

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

7.34 program A 7.32 program kimenete

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.

Összetett és beágyazott listák

Lista-kifejezések beágyazása. 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.

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

7.35 program Beágyazott lista - Erlang

A 7.35 programban az egymásba ágyazás során elsőként a legbelső listakifejezés érvényesül. Ezután a külső kifejezésre kerül a sor, ami a kapott elemeket úgy kezeli, mintha azok egy egyszerű lista elemei lennének.

7.8 megjegyzés 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.

Funkcionális nyelvek ipari felhasználása

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.

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 8.1 listában implementáltunk egy üzenetek küldésére és fogadására alkalmas szervert.

```
-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.

8.1 program Kliens-szerver alkalmazás kódja - Erlang

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 8.2 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.

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

8.2 program Távoli eljárás hívás parancssorból - Er- lang

A 8.1 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 (8.3 programlista).

```
> c( test_server ).
> { ok, test_ server }
> Pid = test_server : start().
> <0.58.0 >
```

8.3 program Szerver indítása - Erlang

A loop/0 négy dologra alkalmas, megszólítani pedig a szintén a modulban szereplő call/2 függvény meghívásával lehet (8.4 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.

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

8.4 program Kérés a szerverhez - Erlang

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 (8.5 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 (8.6 programlista).

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

8.5 program Szerver leállítása – Erlang

```
> test_server : call( Pid, abc 123 ).
> { error, abc 12 3 }
```

8.6 program Hibás kérés kezelése - Erlang

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

8.1 megjegyzés 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.

Irodalomjegyzék

[1] Programozási nyelvek, Volume.: Programozási nyelvek, Nyékyné Ga- izler Judit(ed.) Funkcionális programozási nyelvek elemeiZoltán HorváthProgramozási nyelvek, pages 589-636 ISBN 9-639-30147-7

[2] Horváth Zoltán, Csörnyei Zoltán, Lövei László, Zsók Viktória Funkcio-nális programozás témakörei a programtervező képzésben, Volume.: Informatika a Felsőoktatásban 2005., Pethő Attila, Herdon M.(ed.) Debrecen, Hungary , ISBN 963-472-9096

[3] Horváth Zoltán Funkcionális programozás nyelvi elemei, in.:Technical University of Kosice, Kassa, Slovakia

[4] Lövei László, Horváth Zoltán, Kozsik Tamás, Király Roland, Víg Anikó, Nagy Tamás Refactoring in Erlang, a Dynamic Functional Language, Dig Danny, Cebulla Michael(ed.) in.: Proc of 1st Workshop on Refactoring Tools (WRT07), ECOOP 2007 , ISSN 1436-9915

[5] Kozsik, T., Csörnyei, Z., Horváth, Z., Király, R., Kitlei, R., Lövei, L., Nagy, T., Tóth, M., and Víg, A.: Use cases for refactoring in Erlang In Central European Functional Programming School, volume 5161/2008, Lecture Notes in Computer Science, pages 250-285, 2008

[6] Joe Amstrong Programming Erlang Software for a Concurrent World armstrongonsoftware United States of America 2007 ISBN-10: 1- 9343560-0-X, ISBN-13: 978-1-934356-00-5

[7] Csörnyei Zoltán Lambda-kalkulus Typotex, Budapest, 2007

[8] Csörnyei Zoltán Típuselmélet (kézirat), 2008

[9] Wettl Ferenc, Mayer Gyula, Szabó Péter: L TEX kézikönyv, Panem A Könyvkiadó, 2004

[10] Functional Programming in Clean http://wiki.clean.cs.ru nl/Functional_Programming_in_Clean

[11] F# at Microsoft Research http://research.microsoft.com/en- - us/um/cambridge/projects/fsharp/

[12] The Haskell language http://www.haskell.org/onlinereport/

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

[14] Sikos László Bevezetés a Linux használatába 2005 ISBN: 9789639425002

[15] Imre Gábor Szoftverfejlesztés Java EE platformon Szak Kiadó 2007 ISBN: 9789639131972

[16] GNU Emacs http://www.gnu.org/manual/manual.html

[17] Mnesia Database Management System http://www.erlang.org

[18] Eclipse - The Eclipse Foundation open source community http://www.eclipse.org/