Tartalomjegyzék

I. rész

Tillárom

1. fejezet

A funkcionális programozási nyelvek elemei

A deklaratív programozási nyelvekben a program, a számítási folyamat leírása deklarációk halmaza. A funkcionális program *típus-, osztály- és függvénydeklarációk, ill. definíciók sorozatából, valamint egy* kezdeti kifejezésből *áll. A program végrehajtása nem más, mint a kezdeti kifejezés kiértékelése. A funkcionális nyelvek matematikai számítási modellje a λ-kalkulus. Ebben a fejezetben áttekintjük a funkcionális programozás leglényegesebb fogalmait, történeti előzményeit, matematikai hátterét, kifejezőerejét, hatékonyságát, a funkcionális stílust, az absztrakciót támogató nyelvi elemeket, az ilyen nyelveken írt programok olvashatóságát, módosíthatóságát, megbízhatóságát.*

1.1. Bevezetés

A funkcionális nyelvek elemeinek tárgyalásakor több nyelv[1] elemeit mutatjuk be példákon keresztül, hogy ezzel is kifejezésre jutassuk, hogy elsősorban a funkcionális programozási stílus, ill. a modern funkcionális programozási nyelvekben előforduló közös nyelvi elmek megismerése a cél [Hud89, Hor97, HorFót99], nem pedig egy konkrét nyelv kimerítő tárgyalása [Cla96, Har01, Han00, Tho99, Pla99].

A funkcionális programozás matematikai alapja a Church által 1932-33-ban kidolgozott számítási modell, a *λ-kalkulus*[Bar 84, Csö01]. A funkcionális programozási nyelvek szemantikáját leggyakrabban λ-kalkulus segítségével szokták definiálni. Turing megmutatta, hogy a λ-kalkulusban definiálható, a nem negatív

[1] SML, Miranda, Clean, Haskell.

egészeken értelmezett és hatékonyan kiértékelhető függvények ugyanazok, mint amelyek az imperatív nyelvek számítási modelljeként ismert *Turing géppel* is kiszámíthatók. Funkcionális programozási eszközökkel tehát minden olyan feladat megoldható, amelyik megoldható imperatív nyelven és viszont. A funkcionális programozás modellje egyidős az imperatív programozás modelljével, a Turing géppel, és az első funkcionális nyelv (LISP) is kortársa (1956-1962) az első magasszintű imperatív programozási nyelveknek (FORTRAN, Algol 60). Mindez nem véletlen, a két szemlélet ugyanis nem ellentétes egymással, hanem sokkal inkább egymást kiegészítő. A programozási feladatok megoldása imperatív és funkcionális gondolkodásmódot egyaránt megkövetel.

A számítógépek teljesítménynövekedése és a funkcionális programozási nyelvek fordítási technikájának fejlődése együttesen azt eredményezte, hogy a funkcionális nyelven írt programok hatékonysági szempontból is lényegében egyenértékűek az imperatív nyelven fejlesztett programokkal. Például a Clean nyelv fordítója által előállított kód futási ideje alig haladja meg a C nyelven készített programváltozatét.

A funkcionális nyelvek gyakorlati jelentősége egyre nagyobb, az objektumorientált megközelítést jól kiegészítő jellegük miatt is. Számos szoftverfejlesztő cég használ általános célú vagy saját célra alkotott funkcionális nyelvet egyes különösen összetett részfeladatok megoldására. Ilyen alkalmazási területek például a távközlés (Erlang, Ericsson), vagy a természetes nyelvű szövegek elemzése (Haskell, Lolita), a forgalomirányító rendszerek (Clean), a felhasználói felületek létrehozása.

Az ugyanannak a feladatnak a megoldására funkcionális nyelven írt programkód általában lényegesen rövidebb, kifejezőbb, olvashatóbb és könnyebben módosítható, mint az imperatív nyelven kódolt programszöveg. Ennek az az oka, hogy a tisztán funkcionális nyelvekben az imperatív nyelvekből ismert változófogalom nem létezik, így a kódrészletek mellékhatásokat sem okozhatnak. Egy programrészlet megváltoztatásának hatása sokkal kisebb körben érvénysül, így könnyebben követhető. Funkcionális nyelvek alkalmazásával a nagy programok bonyolultsága lényegesen csökkenthető, a fejlesztéshez szükséges idő lerövidül, a program megbízhatóbb, kevesebb hibát tartalmaz.

A funkcionális programozási nyelvek elterjedését jelenleg leginkább a megfelelően képzett és jó absztrakciós készségekkel rendelkező fejlesztők hiánya akadályozza.

1.1.1. A funkcionális programozási stílus

Funkcionális szempontból nem a program működésének egyes lépései fontosak, hanem a programnak az a hatása, amely a program programfüggvényével (hatás-

relációjával) [Fót83] vagy viselkedési relációjával jellemezhető [Hor94]. Ha programok helyességét vizsgáljuk, akkor a programfüggvényt, a viselkedési relációt vetjük össze a feladattal. A megoldó program egy összetevője mellékhatást okozhat, ha a feladat felbontása, finomítása során nem megfelelő módon választjuk meg az egyes részfeladatokat[2]. A mellékhatások azt eredményezik, hogy a program helyessége csak nagyon körülményesen igazolható, a programkonstrukciókra vonatkozó levezetési szabályok csak nehezen (azaz a mellékhatások figyelembevételével), vagy egyáltalán nem alkalmazhatók. Az utóbbi esetben lehetetlenné válik az elő- és utófeltételekre támaszkodó szintézis, ill. helyességbizonyítási módszer [Tho90, Hor99, MolEek99, HoTePá02]. Tág értelemben *funkcionális programozási stílus*ról beszélünk, ha a megoldó programokat mellékhatásoktól mentes összetevőkből szabályos programkonstrukciókkal építjük fel.

A feladat leírásához használt elő- és utófeltételek a változók[3] értékeire vonatkozó kikötéseket tartalmaznak. Ezen kikötések gyakran függvénykompozíciók értékére vonatkoznak egyenlőségek formájában. A függvénykompozíció alakja sok esetben megfelel a megoldó program felépítésének. A megoldást tehát kereshetjük oly módon is, hogy a feltételeket szabályos függvénykompozíciókkal építjük fel elemi függvényekből. Ezzel egyben közvetett módon azt is meghatározzuk, hogy milyen módon juthatunk el a feladat előfeltételével jellemzett állapotokból az utófeltétellel leírt állapotok egyikébe. Másképpen fogalmazva meghatározzuk az utófeltételben szereplő függvények kiszámítási módját a rendelkezésre álló elemi függvények segítségével. Ha a függvénykompozíciókban *magasabbrendű függvényeket* is használhatunk, azaz olyan függvényeket, amelyeknek az argumentumai és az értéke függvények is lehetnek, akkor egy egységes funkcionális szemlélethez jutunk[4] [Kiss94].

1.1.2. Funkcionális program felépítése és kiértékelése

A *funkcionális program* típus-, osztály- és függvénydeklarációk, ill. definíciók sorozatából, valamint egy *kezdeti kifejezésből* áll. A program végrehajtása nem más, mint a kezdeti kifejezés kiértékelése. A kiértékelést úgy képzelhetjük el, mint a kezdeti kifejezés átírása a benne szereplő függvények szövegszerű behelyettesíté-

[2] Mellékhatás akkor léphet fel, ha nem bővített identitás vagy nem vetítéstartó a feladatot leíró reláció a részfeladat altere és kiegészítő altere felett [Fót83].

[3] A változókat az állapottér projekciós függvényeinek tekintjük.

[4] Hudak [Hud89] nyomán belátható, hogy a funkcionális szemléletben valóban megfogalmazható az imperatív programozásban megszokott *elemi programok* [Fót83] (SKIP, ABORT, értékadás), ill. programkonstrukciók (szekvencia, elágazás, ciklus) mindegyike. Az értékadás például mint a változókon értelmezett *magasabbrendű függvény* definiálható.

vel (1.3. példa). Az átírási lépések pontos jelentését a nyelv modellje határozza meg. A funkcionális nyelven írt programok végrehajtási modellje minden esetben egy konfluens redukciós rendszer (átíró rendszer). Ha egy átíró rendszer *konfluens*, akkor az egyes részkifejezések átírásának sorrendje a végeredményre nincs hatással, a sorrend legfeljebb azt befolyásolja, hogy az átírási lépéssorozattal eljutunk-e a végeredményig. Konfluens rendszer a λ-kalkulus, illetve léteznek konfluens kifejezésátíró, és gráfátíró rendszerek is[5]. Több nyelvben, például az SML-ben, a Haskellben és a Clean-ben megadhatók a λ-kalkulus nyelvtani szabályainak közvetlenül megfelelő kifejezések is [Bar 84].

Egyszerű *függvénydefiníciók* megadásakor a függvény neve és az *argumentumot* azonosító *formális paraméter* áll a definiáló egyenlőség bal oldalán, a függvényértéket meghatározó kifejezés, a *függvény törzse* pedig az egyenlőségjel jobb oldalán. A függvény definícója egy rögzített funkcionális nyelv és kiértékelési módszer mellett egyben meghatározza a függvényérték kiszámításának módját és költségét is adott argumentumértékre. Megjegyezzük, hogy egyes, matematikai értelemben jól definiált függvények funkcionális programozási nyelven hatékonysági vagy kiszámíthatósági szempontok miatt nem adhatóak meg matematikai definíciójukkal azonos alakban.

A továbbiakban megadjuk néhány egyszerű függvény definicióját. Zárójelben mindig feltüntetjük azt is, hogy az adott példa forrásszövegét melyik funkcionális nyelv (vagy nyelvek) fordítója fogadja el [6].

1.1. Példa (Miranda, Clean és Haskell). (Egyszerű függvénydefiníciók).

```
nulla x = 0
id x = x
inc x = x + 1
square x = x * x
squareinc x = square (inc x)
fakt n = product [1..n]
```

A nulla függvény nem függ az `x` paramétertől, konstans függvény, értéke `0`. Az `id` identitásfüggvény a paraméterének értékét adja vissza. Az `inc` függvény (egész típusú) argumentumánál eggyel nagyobb értéket ad eredményül. A

[5] Miranda programok λ-kalkulusra fordítására mutat példát [PlEe93]. A Haskell programok jelentését két lépésben haározzák meg. Megadják a nyelv ún. magjának szemantikáját λ-kalkulusban, majd a további nyelvi elemek jelentését a nyelv magjára támaszkodva [PJH99]. A Clean nyelv szemantikáját egy gráfátíró rendszer határozza meg. Hatékony kiértékelési módszereket: kifejezésátíró rendszereket (TRS), gráfátíró rendszereket (GRS) ismertet [PlEe93].

[6] SML : Moscow ML 2.0; Clean: Concurrent Clean 1.3, 2.1; Haskell: GHCi 5.02.2 for Haskell 98.

`square` függvény az argumentumának négyzetét határozza meg, a `squareinc` pedig az `inc` és `square` függvények kompozíciója. A `fakt` a faktoriális függvény.

1.1. Példa (SML). (Egyszerű függvénydefiníciók).

```
fun square (x) = x * x;
val area = fn  r => pi * r * r
fun plusz a b  =  a + b
```

Az `area` függvény definíciójában az argumentumokat az `fn` kulcsszó[7] és a `=>` jel között helyeztük el. A `fun` kulcsszó alkalmazásával a függvénydefiníció egyszerűbb alakban is felírható, erre példa a `square` és `plusz` függvények definíciója. Az `area` függvény az `r` sugarú kör területét, a `plusz` függvény az argumentumainak összegét határozza meg.

A funkcionális program végrehajtása a kezdeti kifejezésből kiinduló *redukciós* vagy más néven *átírási* lépések sorozata. Egy-egy redukciós lépésben egy – valamely *kiértékelési stratégia* szerint kiválasztott – redukálható részkifejezésben, a *redexben* szereplő függvényhívás helyettesítődik a függvény *törzsében* megadott kifejezéssel a formális és aktuális paraméterek megfeleltetése mellett. *Normál formájú* egy kifejezés, ha további redukcióra nincs lehetőség, ez az átírási lépéssorozat végeredménye.

Megadunk néhány kezdeti kifejezést:

1.2. Példa (Miranda, Clean és Haskell). (Kezdeti kifejezések).

```
Start = sqrt 5.0       // Clean
main = product [1..10] -- Haskell
squareinc 7
```

A normál formák: `2.236068, 3628800, 64`.

1.2. Példa (SML). (Kezdeti kifejezések).

```
Math.sqrt 5.0;
```

A normál forma a `2.2360679775` valós érték[8].

[7] Az `fn` kulcsszó a λ-kalkulus lambda operátorának felel meg.

[8] Az SML értelmezőben csak a `Math` modul betöltése után (`load "Math"`) alkalmazhatjuk a négyzetgyökvonást az `5.0` valós értékre. A sor végére `;`-t kell tenni, ez a *kiértékelő jel*, aminek hatására az értelmező megkezdi a kiértékelést.

Egy-egy kifejezésben gyakran egynél több redex is van, és a kiértékelési módszer határozza meg, milyen sorrendben kerül sor ezek átírására. Bizonyos esetekben egyidőben vagy időben átfedve, egymással párhuzamosan több részkifejezés is redukálható, meggyorsítva ezzel a program végrehajtását.

Konfluens rendszerekben a kifejezések normál formája nem függ a kiértékelési sorrendtől, azaz a normál forma egyértelmű [Bar 84], de nem minden kifejezésnek van normál formája. A kiértékelési módszertől is függ, hogy a normál formához eljutunk-e. A Miranda, a Clean, ill. a Haskell által is alkalmazott *lusta kiértékelés* a kifejezések kiértékelésekor először a kifejezéssel ekvivalens λ-kifejezésben a legbaloldalibb legkülső redexet[9] helyettesíti (azaz a függvénydefiníciót alkalmazza elsőként, ha a kifejezés függvényszimbólummal kezdődik) és az argumentumok kiértékelését csak szükség esetén végzi el. A lusta kiértékelés *normalizáló kiértékelési módszer*[10], azaz mindig megtalálja a normál formát, ha az létezik. Az ML vagy a LISP által alkalmazott *mohó kiértékelés* a legbaloldalibb legbelső redexszel[11], azaz az argumentumok redukálásával kezdi. A mohó kiértékelés gyakran hatékonyabb, de nem mindig ér véget akkor sem, ha létezik normál forma. Ezért a mohó (mohó kiértékelést alkalmazó) nyelvek gyakran tartalmaznak olyan nyelvi elemeket, amelyek egyes kifejezések lusta kiértékelését írják elő[12], ill. a lusta nyelvek gyakran alkalmaznak eljárásokat arra vonatkozóan, hogy szabad-e egy kifejezést hatékonyan, mohó stratégiával kiértékelni[13].

Két példán (1.3.,1.4.) bemutatjuk a két legjellemzőbb kiértékelési módszert, a mohót és a lustát. Az egyes átírási lépésekben a függvényeknek az 1.1. példában megadott definícióit helyettesítjük be.

1.3. Példa. (Redukció - mohó).

```
squareinc 7
-> square (inc 7)
-> square (7 + 1)
-> square 8
-> 8 * 8
-> 64
```

[9] Olyan redex, amelyik nincs egy másik redex belsejében (tehát legkülső), és az ilyen redexek között balról a legelső.

[10] Curry és Feys, 1958.

[11] Olyan redex, amelyik belsejében nincs másik redex (tehát legbelső), és az ilyen redexek között balról a legelső.

[12] Ilyen nyelv például az , ahol létezik a lusta lista.

[13] Ilyen például a Clean nyelv, amelynek fordítója mohóságvizsgálatot végez, ill. a programozó mohóságjelölést alkalmazhat.

1.4. Példa. (Redukció - lusta).

```
squareinc 7
-> square (inc 7)
-> (inc 7) * (inc 7)
-> (7 + 1) * (7 + 1)
-> 8 * 8
-> 64
```

1.1.3. A funkcionális stílus és a modern funkcionális nyelvek főbb jellemzői

Tisztán funkcionális egy programozási nyelv, ha nyelvi elemei felhasználásánál mellékhatások biztosan nem lépnek fel, illetve az előző értéket megsemmisítő[14] értékadás vagy más imperatív programozásra jellemző nyelvi elem nem áll rendelkezésre.

Ellentétben például a LISP-pel és az SML-lel, a Hope, a Miranda, a Haskell, a Concurrent Clean tisztán funkcionális nyelvek.

Az alábbiakban bemutatjuk a modern, tisztán funkcionális programozási nyelvek legfontosabb jellemzőit.

- *Hivatkozási helyfüggetlenség*[15]
  A kifejezések hivatkozási szempontból az előfordulási helyüktől függetlenek, azaz ugyanaz a kifejezés a program szövegében mindenhol ugyanazt az értéket jelöli. A függvények kiértékelésének nincs mellékhatása, azaz egy függvény kiértékelése nem változtathatja meg egy kifejezés értékét. A tisztán funkcionális program *változói*[16] valójában állandók. A változók értéke – akárcsak a matematikában – esetleg még nem ismert, de egyértelmű, semmiképpen sem változhat meg a program végrehajtása során. Ezen a tulajdonságon alapszik a program funkcionális helyességét igazoló ún. *egyenlőségi érvelés*[17] alkalmazhatósága, azaz például az alábbi programrészletben

```
.... x + x ...
       where  x = f a
```

[14] destructive
[15] referential transparency
[16] például függvénydefiníciókban használt formális paraméterek
[17] equational reasoning

az `x` minden *szabad előfordulása* helyébe szövegszerűen behelyettesíthető az `f a` kifejezés a `where` hatáskörén belül, és `f a` értéke minden esetben azonos.

- *Szigorú, statikus típusosság*
  Típusdeklarációk megadása nem kötelező, de megköveteljük, hogy a *Hindley–Milner féle korlátozottan polimorfikus típusrendszer* alapján a kifejezések típusa a *típuslevezetési szabályok* által meghatározott legyen. Ez azt jelenti, hogy egy adott kifejezés *legáltalánosabb típusát* a fordítóprogram általában még akkor is meg tudja határozni a benne szereplő részkifejezések típusa alapján, ha a programszöveg készítője nem deklarálta a kifejezés típusát. Biztosítottak a nyelvi eszközök absztrakt és algebrai adattípusok leírásához is.

- *Magasabbrendű függvények*[18]
  A függvények ugyanolyan értékek, mint az elemi típusértékhalmazok elemei. Magasabbrendű függvénynek nevezzük azokat a függvényeket, amelyeknek valamelyik argumentuma vagy értéke maga is függvény. Magasabbrendű függvények segítségével biztosítható a program *modularitása*, illetve a *funkcionális absztrakció* elmélyítése[19].
  Pld.: `twice f x = f ( f x )`

- *Curry módszer, részleges függvényalkalmazás*
  Ha függvények eredménye függvény is lehet, akkor nincs szükség a *többváltozós függvény* fogalmára. Haskell B. Curry módszere alapján minden függvényt tekinthetünk egyváltozósnak. Ha egy függvénynek több változója van, akkor Curry módszere szerint csak az első változót tekintjük a függvényhez tartozónak. Az első argumentum megadásával már egy ujabb függvényhez jutunk, amelyet a következő argumentumra alkalmazhatunk. Többváltozós függvény alkalmazását tekinthetjük egyváltozós függvények alkalmazása sorozatának. Ebben a szemléletben például az összeadás műveletét is egyváltozósnak tekintjük oly módon, hogy az összeadás első argumentumát tekintjük a `+` művelet egyetlen argumentumának, értéke pedig egy újabb függvény. A `(+) 1` függvény például megegyezik az `inc` függvénnyel. Többváltozós függvény *részleges alkalmazásáról* beszélünk, ha

[18] higher order functions

[19] A magasabbrendű függvények alkalmazása hatással van a számítási folyamatra is. Mohó kiértékelés esetén például egy függvény kiértékelésére előbb kerülhet sor, ha egy magasabbrendű függvény argumentumaként jelenik meg. Magasabbrendű függvények használatával a számítási folyamatot jól meghatározott szakaszokra bonthatjuk.

– balról jobbra haladva – argumentumainak egy részének rögzítésével egy függvényt kapunk eredményül.

- *Függvények alkalmazása önmagukra*
  Lehetőség van *rekurzív és kölcsönösen rekurzív* függvénydefiníciók megadására.

- *Lusta*[20] *kiértékelés – mohó*[21] kiértékelés.
  A modern funkcionális programozási szemléletben a kifejezések jelentését a lusta kiértékelési módszer határozza meg. A függvények argumentumai értékelődnek ki, ha ténylegesen szükség van az értékükre a normál forma eléréséhez. Lusta redukciós módszert alkalmaz a Miranda, a Clean és a Haskell is. Mohó kiértékelésről beszélünk, ha az argumentumot mindig kiértékelik, mielőtt a függvény alkalmazásának megfelelő redukcióra sor kerülne. Ilyen kiértékelési módszert alkalmaz például a LISP, az SML és a Hope.

- *Zermelo-Fraenkel halmazkifejezések*
  Iteratív adatszerkezet elemeinek és azok sorrendjének megadására alkalmas, a matematikában halmazok megadásánál alkalmazott jelölésrendszernek megfelelő nyelvi eszköz a Zermelo-Fraenkel halmazkifejezés. A végtelen adatszerkezetek (listák, halmazok, sorozatok, vektorok) kiértékelése lusta kiértékelési módszerrel történik. A
  `[ x * x | x <- [1..], odd(x) ]`
  kifejezés egy olyan végtelen listát határoz meg, amelyben rendre szerepelnek a páratlan (`odd`) természetes számok (`[1,3,5 ..]`) négyzetei. A teljes program futásának befejeződése az eredményt mint argumentumot felhasználó függvény igényeitől függ (modularitás), a listát pedig lustán, azaz csak addig értékeljük majd ki, ameddig szükség van a benne szereplő elemekre.

- *Argumentumok mintaillesztése*.
  A függvények definiálásakor megadhatunk mintákat[22] (ld. 1.2.3. rész) a formális paraméter helyén. Amennyiben az aktuális paraméter egy megadott mintára illeszkedik, a függvény értékét az ehhez a mintához tartozó függvénytörzsváltozat határozza meg (pontos szabályt ld. a 1.2.3. részben). Például:

[20] lazy
[21] eager,strict
[22] Kevés kivételtől eltekintve mintában csak a formális paraméter típusának algebrai típusdefiníciójában megadott konstruktorokat használhatjuk, ld. 1.4.2. rész.

```
fac 0      = 1
fac n|n>0  = n * fac (n-1)
```

A `fac` függvény definíciójában az első minta a `0` érték. Ha az aktuális paraméter illeszkedik a `0` mintára, azaz nullával egyenlő, akkor a függvény értéke `1`.

- *Margó szabály*[23]
  Összetartozó kifejezések csoportjának azonosítására és deklarációk *hatáskörének* korlátozására alkalmas a bal oldali margó szélességének változtatása. A margó szabály a blokkszerkezet kialakításának egyik nyelvi eszköze. Egy deklaráció hatásköre lokális egy, a forrásszövegben őt megelőzőre nézve, ha a deklaráció úgy követi az előzőt, hogy beljebb kezdődik. Minden nyelv leírása megadja a margó szabály alkalmazásának pontos definícióját.[24]. A hatáskörök egymásba is ágyazhatóak, oly módon, hogy egyre inkább jobbra haladva kezdjük írni a deklarációkat. Az alábbi példában az `add4` függvény egy lokális `succ` függvényt alkalmaz, amely különbözik az `add` függvényben használt `succ` függvénytől:

  ```
  add4 = twice succ
       where
         succ x = x + 2
  add  = ... succ
  ```

- *A modern funkcionális nyelvek rendelkeznek valamilyen I/O modellel is*
  Ilyen modell pl. az *IO monád*, *az egyszeresen hivatkozott, egyedi környezet*[25], *az adatáramlási IO modell*[26], vagy a vele egyenértékű *folytatás*[27] modell (ld. 1.7.).

1.1.4. A funkcionális nyelvek rövid áttekintése

A LISP-et (LISt Processing), az első olyan nyelvet, amelynek λ-kalkulus volt a számítási modellje, John McCarthy alkotta meg 1956-1962 között az MIT-n.

[23] off-side rule (Landin)

[24] Ha az egyes megjelenített karaktarek szélessége nem állandó, nagyon nehéz megállapítani a deklarációk hatáskörét. A forrásszövegen végrehajtott szövegszerkesztési lépések következtében is megváltozhat egyes deklarációk hatásköre például akkor, ha egyes azonosítókat hosszabbakra cserélünk.

[25] unique environment

[26] stream of request, stream of response

[27] continuation

Azóta számos LISP-változat jött létre, például a Common LISP (DARPA, 1981), amely procedurális és objektum-elvű elemeket (CLOS, Common Lisp Object System) is magában foglal, vagy a Scheme nyelv (Steele, Sussman,1975), amelyet inkább az egyetemeken használnak és CAD rendszerekben alkalmaznak. Az első típusos funkcionális nyelv az ML (Meta Language), amely eredetileg az Edinburghban tételbizonyításra tervezett LCF (Logic for Computable Functions) rendszer metanyelve volt. A nyelvet R. Milner tervezte a 70-es évek közepén. A Hope (Burstall, 1980) megalkotása után alkotta Milner, Tofte és Harper az SML-t (Standard ML) 1983 és 1990 között. Az SML legújabb, átdolgozott szabványa 1997-ben készült el [Mil97, Har01, Han00]. ML-változat a Caml (INRIA, 1984-1990, a Coq tételbizonyító alapnyelve) és az Objective Caml (a Caml Light továbbfejlesztése, INRIA, 1990-). Az ML-változatok nem tisztán funkcionális nyelvek, imperatív nyelvi elemeket (például frissíthető változókat) is tartalmaznak. Az ISWIM (If You See What I Mean, Landin, 1966) volt az első lusta kiértékelést alkalmazó tisztán funkcionális nyelv, amely a margó szabályt is bevezette. D. Turner az ISWIM-ből kiindulva több nyelvet is tervezett, az ugyancsak lusta kiértékelésű SASL-t (Single Assignment Language, Turner, 1976), a KRC-t (Kent Recursive Calculator, Turner, 1981), majd 1985-86-ban a Mirandát (Turner, 1986) [Mir90, Tur86, Kiss94, Cla96]. A Miranda üzleti termék, a Research Software Ltd. tulajdona. A Miranda elemeit felhasználó nyelv a Haskell (1990) [PJH99, HudFasPe99, Tho99].

A Haskell megalkotásáról 1987-ben döntöttek egy funkcionális programozási konferencián (FPCA '87). A nyelvet Haskell Brooks Curry matematikusról nevezték el. Legújabb definíciója a Haskell'98 szabvány. A tervező csoport tagjai között a világ számos egyeteméről, illetve kutatóintézetéből találunk kutatókat (J. Hughes, S. Peyton Jones, P. Hudak, K. Hammond, E. Meijer, J. Peterson, P. Wadler és mások). A Haskell esetében alapkövetelményként fogalmazták meg, hogy a nyelv

- egyaránt alkalmas legyen oktatási és kutatási feladatok megoldására, illetve nagy alkalmazói programok fejlesztésére,
- legyen formálisan definiált szintaxisa és a szemantikája,
- szabadon elérhető legyen,
- feleljen meg széles körben elfogadott alapelveknek,
- egységes fejlődési irányt szabjon az egymástól kisebb-nagyobb mértékben különböző modern funkcionális nyelveknek.

A Concurrent Clean (Plasmeijer, Nijmegen, 1987) [PlaEek01, Pla99] egy gráfátíró kisérleti nyelvből (a LEAN-ból) fejlődött ki, tisztán funkcionális, alapvetően lusta kiértékelést alkalmazó nyelv. Jelenlegi változata (a Clean 2.0) közel áll a Haskellhez [HeHoZsó02], de több olyan nyelvi elemet is tartalmaz, amelyet a Haskell'98 nem.

A függvénykompozíció asszociatív művelet, ezért eredményének kiszámítása jól párhuzamosítható. A kifejezések normál formája, ha létezik, független a kiértékelési módszertől (legalábbis a konfluens átíró rendszerekben), így a funkcionális nyelven írt programok jól párhuzamosíthatók. A legtöbb nyelvnek létezik olyan változata (Concurrent Clean [Kes96], GpH - Glasgow Parallel Haskell [Tri98], Distributed Haskell [HuNo01], Eden, Concurrent ML, JoCaml [Fou01] stb.), amely nyelvi elemekkel is támogatja annak meghatározását, hogy melyik részkifejezést célszerű (spekulatív módon) párhuzamosan vagy elosztott módon kiszámítani.

1.2. Egyszerű funkcionális programok

1.2.1. Egyszerű függvények definíciója

A függvények definíciója egy vagy több olyan egyenlőségből áll, amelyek bal oldalán a függvény neve és *formális paraméterei*, jobb oldalán pedig a függvény értékét és kiszámítási módját is meghatározó kifejezés, a *függvénytörzs* áll. A *kifejezés* állhat egyetlen értékből, a formális paraméterből vagy egy *függvény alkalmazásából* annak aktuális paraméterére (amely lehet a definiált függvény formális paramétere is). A kifejezések kiértékelése során a legmagasabb *prioritása* a függvényalkalmazásnak van. Megadhatunk az argumentumokra vonatkozó *feltételeket* és *mintákat* is, amelyek az aktuális és a formális paraméterek *illesztését* határozzák meg. A függvény értelmezési tartománya és értékkészlete is a deklaráció része, ez azonban egyszerűbb esetekben elhagyható.

A függvények definiálásakor a formális paraméterek leírására változóneveket vezetünk be. A változó deklarációjának *hatásköre* ez esetben a változót bevezető egyenlőségre terjed ki. A deklarált függvénynevek hatásköre alapesetben a modulra, egyes esetekben a teljes programra terjed ki. A deklarációk hatásköre szűkíthető a *margó szabály*, illetve lokális deklarációk megadását jelző kulcsszavak (`where`, `local`, `let`, # stb.), valamint a modulok közötti export-import mechanizmus alkalmazásával.

Rekurzív függvénydefiníciót is megadhatunk. Hatékonysági szempontból előnyös lehet, ha a rekurzív hivatkozás egyszeres és az a definició végén helyezkedik el. *Esetszétválasztással* meg kell adnunk az alapesetet, különben a kiértékelés nem ér véget.

1.5. Példa. (Feltételes kifejezéssel megadott rekurzív függvény).

```
fakt n  = if n==0 then 1 else n * fakt (n-1)-- Haskell
fakt n  = cond (n=0) 1 (n*fakt(n-1))       || Miranda
fakt n  = if (n==0) 1  (n * fakt (n-1))   // Clean
fun fakt n = if n=0 then 1 else n * fakt (n-1) (*SML*)
```

1.2.2. Feltételek megadása

Az esetszétválasztást a matematikában megszokott módon is megadhatjuk az argumentumokra vonatkozó *feltételekkel*. A feltételeket a függvény alkalmazásakor az aktuális paraméter értéke alapján, megadásuk sorrendjében vizsgálja meg a kiértékelő rendszer. Az első igaz feltételhez tartozó egyenlőséget alkalmazza, ha pedig nincs ilyen[28], akkor valamilyen hibaüzenettel leáll a kiértékelés vagy egy kivételkezelő kiértékelésére kerül sor (ld. 1.8.). A feltételek megadásának sorrendje szemantikai szempontból lényeges. Ha a sorrendet megváltoztatjuk, általában a definíció jelentése is megváltozik.

1.6. Példa (Clean és Haskell). (Esetszétválasztás, rekurzív függvény).

```
fakt n | n==0 = 1
       | n>0  = n * fakt (n-1)
```

1.7. Példa (Miranda). (Esetszétválasztás, rekurzív függvény).

```
lnko a b = lnko (a-b) b, if a>b
         = lnko a (b-a), if a<b
         = a,           otherwise
```

1.2.3. Mintaillesztés

Mintaillesztés alkalmazásával is szétválaszthatunk eseteket. Mintaillesztés esetén azt vizsgálja a kiértékelő rendszer, hogy az aktuális paraméter értéke vagy alakja megfelel-e valamelyik definiáló egyenlőség bal oldalán megadott mintának. A minták és a hozzájuk tartozó egyenlőségek sorrendje meghatározza a mintaillesztés eredményét. Ha a sorrendet megváltoztatjuk, általában a definíció jelentése is

[28] Ilyen esetben egy parciális függvényt definiáltunk.

megváltozik. A kiértékelő rendszer az első illeszkedő mintához tartozó egyenlőséget alkalmazza, ha pedig nincs ilyen, hibaüzenettel leáll[29]. Minták megadása helyett mindig megadhatunk ekvivalens *feltételsorozatot*, de ez fordítva nem igaz. A mintaillesztés, mint nyelvi elem nem szükséges ahhoz, hogy teljes funkcionális nyelvet alkossunk, de nagymértékben egyszerűsíti a függvénydefiníciók megadását és javítja a kód olvashatóságát. Módosíthatósági szempontból hátrányos, hogy a sorrendben előbb megadott minta módosítása kihat a később megadott minták jelentésére.

1.8. Példa (Miranda, Clean és Haskell). (Mintaillesztés).

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

Mintaillesztés és feltételek egyszerre is alkalmazhatók:

1.9. Példa (Clean és Haskell). (Mintaillesztés és feltételek).

```
fakt 0         = 1
fakt n | n > 0 = n * fakt (n-1)
```

Mintaillesztésre az SML is biztosít nyelvi elemet:

1.3. Példa (SML). (Mintaillesztés).

```
fun fakt 0  = 1
      |  n  = n * fakt (n-1)
```

1.3. Függvények típusa, magasabbrendű függvények

A függvények típusát értelmezési tartományuk és értékkészletük megadásával határozzuk meg. Az értelmezési tartomány és az értékészlet közé `->` jelet teszünk, illetve az SML-ben az értelemzési tartományt és változóját zárójelek közé is tehetjük, az értékkészletet pedig a zárójel után kettősponttal elválasztva adhatjuk meg.

1.10. Példa. (Függvény típusa).

[29] A mintaillesztések kiértékelésének módja lényegesen egyszerűbb és hatékonyabb a logikai programozási nyelvekben alkalmazottnál, mert a mintában – néhány kivételtől eltekintve – csak a formális paraméter típusának algebrai típusdefiníciójában megadott konstruktorokat használhatjuk, ld. 1.4.2.

```
ketszer :: num->num          || Miranda
ketszer :: Num a => a -> a   -- Haskell
fakt :: !Int -> Int          // Clean
val fakt : int -> int        (* SML *)
fun fakt (x:int):int         (* SML *)
```

A Mirandában a numerikus típusok közös azonosítója a `num`. A Haskell nyelv megkülönbözteti az egyes numerikus típusokat egymástól, de lehetőséget ad arra, hogy a numerikus típusok `Num` *osztályának* minden `a` elemtípusára egyszerre definiáljunk egy függvényt. A típusosztályokat a későbbiekben részletesen tárgyaljuk.

A funkcionális nyelvekben és a λ-kalkulusban szigorú matematikai értelemben minden függvénynek legfeljebb egy argumentuma van. *Többváltozós függvények* megadása *magasabbrendű függvények* segítségével lehetséges, a Haskell B. Curryről elnevezett módszer[30] alapján. *Elsőrendű* az a függvény, amelynek sem argumentuma, sem értéke nem függvény.

1.11. Példa. (Többváltozós függvény típusa).

```
plusz :: Num a => a -> a -> a -- Haskell
plusz :: num -> num -> num    || Miranda
plusz :: Int Int -> Int       // Clean
plusz a b = a + b
val plusz : int -> int -> int = (* SML *)
     fn a => fn b =>  a + b     (* SML *)
```

A `plusz` függvény egyetlen argumentuma `a`, eredménye pedig az a függvény, amely az `a`-nak megfeleltetett aktuális paraméterhez adja hozzá azt a számértéket, amelyet az új függvény egyetlen formális paramétere, a `b` azonosít.

`plusz :: Int -> Int -> Int` jelentése tehát azonos `plusz :: Int -> (Int -> Int)`-tel. `plusz 5` egy szabályos függvényalkalmazás, amelynek eredménye egy `Int -> Int` típusú függvény. `plusz 5 6` egyenlő `(plusz 5) 6`-tal. A többváltozós függvények típusdefiníciói tehát jobbról balra *asszociatívak*, míg alkalmazásuk balról jobbra asszociatív: `f :: a -> b -> c` egyenértékű `f :: a -> (b -> c)` -vel, azaz `f` egy olyan függvény, amelynek argumentuma `a` típusú, eredménye pedig egy olyan függvény, amelynek értelmezési tartománya `b`, értékkészlete `c`. Az `f x y` függvényalkalmazás ennek megfelelően egyenértékű `(f x) y`-nal.

A Curry módszer alkalmazásával többváltozós függvényekből könnyen készíthetünk olyan példányokat, amelyekben néhány paraméter értékét rögzítjük (részleges alkalmazás). Ilyen függvény az `inc`.

[30] Schönfinkel, Curry, Feys

1.12. Példa (Miranda, Clean és Haskell). (Részlegesen alkalmazott függvény).

```
inc :: num -> num      || Miranda
inc :: Num a => a -> a -- Haskell
inc :: Int -> Int      // Clean
inc x = (+) 1 x       // Miranda, Clean, Haskell
inc = (+) 1           // Clean, Haskell
fun inc (x:int):int = plusz 1 x (* SML *)
val inc : int->int = secl 1 op+ (* SML *)
```

Az összeadás műveleti jelét (+) általában infix helyzetben alkalmazzuk, azaz a két összeadandó között helyezzük el. A `plusz` függvény definíciója – a függvényekre általánosan érvényes – prefix alkalmazáshoz készült. A legtöbb nyelvben azonban a bináris műveleteknél lehetőség van arra, hogy infix operátort prefix helyzetben használjunk és viszont. Például: `(+) 2 3` (Clean, Haskell, Miranda), `2 ‘plusz‘ 3` (Haskell), `2 $plusz 3` (Miranda), `op+ (2,3)` (SML). Az `inc` függvény `(+) 1`, ill. `secl 1 op+` alakú[31] definíciói éppen ezt a lehetőséget használják fel, hogy az infix bináris + művelet első argumentumát az összeadás művelet prefix változatát alkalmazva rögzítik.

Megjegyezzük, hogy egy függvény alkalmazásának nem csak az értéke lehet függvény, hanem megfelelő módon definiált függvény argumentuma is lehet függvény. Ezt mutatjuk be a következő és a 1.16. példán.

1.13. Példa (Miranda, Clean és Haskell). (Függvény, mint argumentum).

```
nullaban :: Num a => (a -> a) -> a -- Haskell
nullaban :: (num -> num) -> num    || Miranda
nullaban :: (Int -> Int) -> Int    // Clean
nullaban f  = f 0
fun nullaban (f: int -> int) : int
            = f 0                  (* SML *)
nullaban inc
```

A 1.13. példában megadott függvény típusdefiníciójában szereplő zárójelek nem hagyhatók el. A zárójelek elhagyása maga után vonná az asszociativitási szabály alkalmazását és teljesen megváltoztatná a definíció értelmét. A zárójelek megadásával jelölnünk kell, hogy a függvény paramétere nem egy számérték, hanem függvény.

[31] `fun secl x f y = f(x,y)` az `f` bal oldali argumentumának lekötésére használható SML függvény.

1.3.1. Egyszerű típuskonstrukciók

A funkcionális nyelvekben is rendelkezésünkre áll a direkt szorzat, illetve iterált *típuskonstrukció*. Rendezett n-eseket, ill. véges vagy végtelen a *sorozatokat*[32] képezhetünk. A rendezett n-eseket gömbölyű `( )`, a sorozatokat pedig szögletes `[]` zárójelek közé zárjuk.

Rendezett n-esek

A rendezett n-esek összetevőit mintaillesztéssel, vagy előre definiált szelektorfüggvényekkel[33] (`fst, snd, #i`, stb.) érhetjük el.

1.14. Példa (Clean és Haskell). (Rendezett pár legnagyobb közös osztója).

```
lnko :: (a,a) -> a | - , < , == a   // Clean
lnko (a,b) | a>b = lnko (a-b,b)
           | b>a = lnko (a,b-a)
           | a=b = a            -- Haskell
           | a==b = a           // Clean
```

1.4. Példa (SML). (Rendezett pár legnagyobb közös osztója).

```
lnko : int * int -> int
fun lnko (0,b) = b
  | lnko (a,b) = lnko(a mod b, b);
```

A rendezett n-esek között kitüntett szerepe van a rendezett nullásnak, amelynek egyetlen összetevője sincs. A `()` érték az SML `unit` típusának egyetlen típusértéke.

Sorozatok

A `:`, illetve az SML-ben a `::` *konstruktorok* alkalmasak arra, hogy egy elemből és egy sorozatból egy újabb, az eredeti sorozatot az adott elemmel balról kiterjesztő sorozatot készítsenek. A konstruktorokat mintákban is használhatjuk, így mintaillesztéssel könnyen eldönthető, hogy a sorozat üres-e (`[]`), valamint a nem üres (`[a:x]`) sorozat (`a`) első eleme, illetve a fejelem nélküli maradék (`x`) sorozat egyszerűen a minta megadásával meghatározható.

[32] A funkcionális nyelveket ismertető könyvekben a sorozatok helyett *listákról* beszélnek, utalva a megvalósítás módjára. Egyes funkcionális nyelvekben - amelyek nem statikusan típusosak - lehetséges, hogy egy lista elemei különböző típusúak legyenek. Ezt sem a Miranda, sem a Clean, sem a Haskell, sem az SML nem engedi meg.

[33] szelektorfüggvény

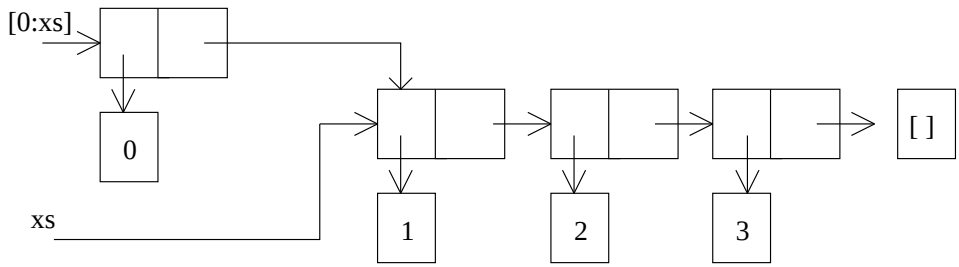
1.15. Példa. (Sorozat elemeinek összege).

```
fun osszeg (a::x) = a + osszeg x
 |  osszeg []     = 0;                (* SML *)

osszeg [] = 0
osszeg (a:x) = a + osszeg x  -- Haskell
osszeg [a:x] = a + osszeg x  // Clean


osszeg [1,2,3,4,5,6,7,8,9,10]
osszeg [1..10]               -- Haskell, Clean
```

Az 1.1. ábra a sorozatok láncolt listaként történő ábrázolását mutatja be. Az ábrán megfigyelhetjük, hogy a lista elemeit külön tároljuk, a lista vázában csak az elemek címe található. Az alábbiakban megadjuk néhány olyan függvény definícióját Clean nyelven, amelyeket listák esetén gyakran használunk. `hd` az első, a `last` a sorozat utolsó elemét adja vissza. A `tl` a sorozat fejelem nélküli részsorozatát adja eredményül. Mivel a tisztán funkcionális nyelvekben a függvényeknek nincs mellékhatásuk, egy lista kezdőrészét (`init`) (az utolsó elem nélküli részét) csak úgy kaphatjuk meg, ha a régi lista vázát megőrizve a kezdőrészhez teljesen új vázat építünk. Az eredeti lista vázát nem módosíthatjuk, az utolsó értékre mutató vázelemet nem szakíthatjuk le[34], mert azzal megsértenénk a hivatkozási helyfüggetlenséget.



1.1. ábra. Sorozatok belső ábrázolása.

```
hd [a:x]  = a                tl [a:x] = x
hd [] = abort "hd of []"     tl []  = abort "tl of []"
last [a]  = a                init [] = []
```

[34] A sorozat belső ábrázolását megvalósító láncolt lista felépítéséhez használt mutatókhoz egyébként sem férhetünk hozzá.

```
last [a:tl] = last tl          init [x]  = []
last [] = abort "last of []"   init [x:xs] = [x: init xs]
```

A `map` függvény segítségével elemenkénti feldolgozást adhatunk meg. A `map` argumentuma az egyes elemekre alkalmazott függvény. A `modseq` függvényt a `map` segítségével definiáljuk: egy számértékeket tartalmazó listából az argumentumként kapott bináris művelet és egy számérték segítségével elemenként határozza meg az új listát. Figyeljük meg, hogy a definícióban mindkét oldalról elhagyhattuk az utolsó paramétert, a listát.

1.16. Példa. (Függvény, mint argumentum).

```
modseq ::(num->num->num)->num->[num]->[num] || Miranda
modseq :: Num a => (a->a->a)->a->[a]->[a]   -- Haskell
modseq :: (Int Int -> Int) Int->([Int]->[Int]) // Clean
modseq f c = map (f c)
modseq :(int->int->int)->int->int list -> int list;(*SML*)
fun modseq f c = map (f c);        (* SML *)

modseq plusz 5 [1,2,3]  // = [6,7,8]
modseq plusz 5 []       // = []
```

Sorozatokat *generálhatunk* is korábban definiált sorozatokból logikai kifejezések, *szűrők* megadásával.

1.17. Példa (Miranda, Clean és Haskell). (Szám osztói).

```
osztok n = [i | i<-[1..n], n ‘mod‘ i == 0 ] -- Haskell
osztok n = [i | i<- [ 1 .. n ]; n mod i = 0 ] || Miranda
osztok n = [i \\ i<-[1..n] | n mod i == 0 ] // Clean
```

Egy `n` szám osztóinak sorozatát úgy kapjuk meg, ha `1`-től `n`-ig előállítjuk az összes `i` értéket az `i<-[1..n]` generátorral, majd az `n mod i == 0` szűrő alkalmazásával kiválogatjuk azokat, amelyekkel `n`-t osztva `0` maradékot kapunk. A `..` szimbólummal (a *pont-pont-jelöléssel*) véges vagy végtelen számtani sorozatokat adhatunk meg, a számtani sorozat differenciáját pedig az első két elem különbségével adhatjuk meg. Ha nem adjuk meg a második elemet, akkor a differencia `1`, ha nem adjuk meg az utolsó elemre vonatkozó felső korlátot, akkor a sorozat végtelen. Egynél több generátort is megadhatunk, amelyek vagy egymásba ágyazva, vagy párhuzamosan futva állítják elő az elemeket:

1.1. Példa (Clean). (Generátorok).

```
[ (x,y) \\ x <- [1..4], y <- [1..x] | x+y > 3 ] // beágyazott
[ (x,y) \\ x <- [1..4] & y <- [6..8] ]          // párhuzamos
```

Rekordok

A rendezett n-esek mellett a Cleanben és az SML-ben névvel ellátott mezőkkel rendelkező rekordokat is használhatunk. Ha egy rendezett n-es nagyon sok összetevőből áll, akkor nagyon körülményes egyes elemeinek mintaillesztéssel történő elérése. Ha a direkt szorzat típuskonstrukció értékeinek elemeit névvel ellátott szelektorfüggvénnyel azonosíthatjuk, akkor a programszöveg áttekinthetőbb és módosíthatóbb lesz. Az SML-ben és a Clean-ben nagyon hasonló a rekord típus, a rekord típusú érték és a rekordminta megadásának alakja, így most csak egy Clean nyelven írt példát mutatunk.

1.2. Példa (Clean). (Rekord).

```
:: Point = {  x       ::  Real
           , y       ::  Real
           , visible ::  Bool
           }
:: Vector = { dx       ::  Real
           , dy       ::  Real
           }
Origo :: Point
Origo = { x = 0.0
        , y = 0.0
        , visible = True
        }
```

Mintaillesztés esetén elegendő, ha csak azokat a rekordmezőket adjuk meg a mintában, amelyek a függvény törzsében is szerepelnek.

1.3. Példa (Clean). (Mintaillesztés, rekordminta).

```
IsVisible :: Point -> Bool
IsVisible {visible = True} = True
IsVisible _                = False
```

A szelektorfüggvények jelölésére a Clean-ben alkalmazhatjuk a más nyelvekben megszokott `.` jelölést, míg az SML-ben a `#<mezőnév>` alakú függvényekkel határozhatjuk meg egy rekord valamely összetevőjének értékét.

```
xcoordinate :: Point -> Real
xcoordinate p = p.x
```

A Clean nyelvben lehetőségünk van arra is, hogy egy rekord típusú érték definíciójában egy másik rekordra hivatkozzunk. A `hide p` egy olyan rekord, amelynek minden mezője megegyezik a `p` rekord megfelelő mezőjének értékével, *kivéve* az `&` jel után felsorolt `visible` mezőt.

1.4. Példa (Clean). (A kivéve művelet).

```
hide :: Point -> Point
hide p = { p & visible = False }

Move :: Point Vector -> Point
Move p v = { p & x = p.x + v.dx, y = p.y + v.dy }
```

A `Move` függvény *nem* tolja el a `p` pontot a síkon a `v` eltolásvektorral, hanem meghatároz egy olyan *új* pontot, amely a `p`-ből úgy származtatható, hogy minden mezőjének értéke ugyanaz, *kivéve* az `x` és `y` koordinátát.

Tömbök

Láncolt listák használatakor a sorozat elemeinek elérése csak úgy lehetséges, ha a listát az adott elemig bejárjuk. Tömböknél a kiértékelő rendszer címaritmetikát alkalmaz, az egyes elemeket azonos idő alatt érjük el. Az SML és a Clean nyelv – továbbá a Haskell egyes kiterjesztései is – tartalmazzák ezt a nyelvi elemet.

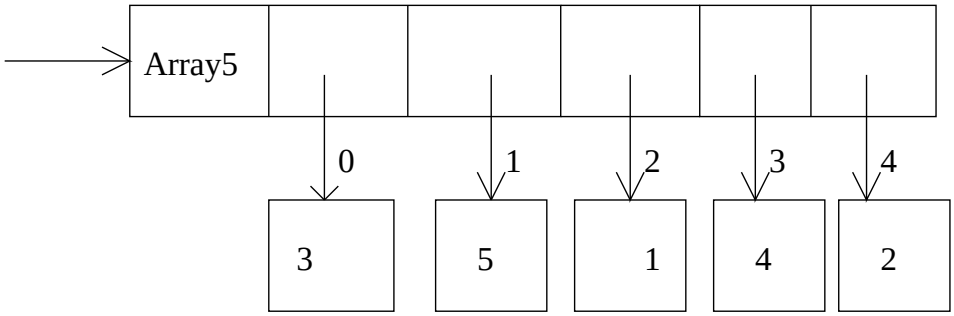
Az SML tömbök a nyelv imperatív részéhez tartoznak, az elemek értéke változhat. Clean-ben csak az *egyszeresen hivatkozott*[35] (ld. 1.6.1.) típusú tömbök elemei kaphatnak új értéket. Ilyenkor a hivatkozási helyfüggetlenség nem sérül. Az egyszeres hivatkozást a tömb típusának megadásánál `*`-gal jelezzük. A frissíthető tömböket a 1.6. részben mutatjuk be.

A Clean megkülönbözteti a *dobozolatlan* és a *dobozolt* tömböket, annak megfelelően, hogy a vázban tároljuk az elemeket vagy a listákhoz hasonlóan a váz csak az elem címét tartalmazza. A dobozolatlan tömbök típusdefiníciójában a nyitó kapcsos zárójel után egy # jel áll.

[35] unique

1.5. Példa (Clean). (Dobozolt tömb).

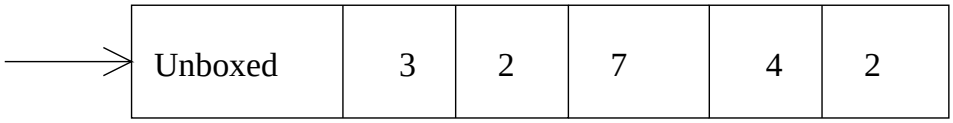
```
Array5 :: *{Int}
Array5 = {3,5,1,4,2}
```



1.2. ábra. Dobozolt tömb.

1.6. Példa (Clean). (Dobozolatlan tömb).

```
Unboxed :: {#Int}
Unboxed = {3,2,7,4,2}
```



1.3. ábra. Dobozolatlan tömb

A tömbök elemeit az imperatív nyelvekben megszokotthoz hasonló módon választhatjuk ki. A tömb első elemét a `0` indexértékhez rendelték hozzá.

```
Array5.[1]+Unboxed.[0]
```

A Clean-ben a tömböket generátorok segítségével is megadhatjuk. Az első példában az `[1,2,3]` lista elemeit a `<-` listagenerátor alkalmazásával sorra kiolvassuk a listából és elhelyezzük a `narray` tömbben. A második példában az `Array5` tömb elemeit olvassuk ki a `<-:` tömbgenerátor alkalmazásával és elhelyezzük az `nlist` listában.

```
narray = { e \\ e <- [1,2,3] }
nlist = [ e \\ e <-: Array5 ]
```

1.3.2. Lokális deklarációk

A Mirandában, Clean-ben és a Haskellben a `where` kulcsszó használatával *lokális* függvényeket vezethetünk be. A `where` *hatáskörét* a *margó szabály* határozza meg. Az SML-ben a `local`, `in`, `end` szerkezetben helyezhetünk el egy deklarációhoz tartozó lokális deklarációkat, a `let`, `in`, `end` szerkezetben pedig egy kifejezésben szereplő lokális deklarációkat adhatunk meg. `let` kifejezéseket Haskellben és Clean-ben is használhatunk. Clean-ben a `let` kifejezéseknek több, formai szempontból is és jelentését tekintve is különböző alakja létezik. A # jel például bevezetheti egy függvénytörzs előtt az abban használt lokális állandó deklarációját. A `where` kifejezésben megadott lokális definíciókban a tartalmazó függvény formális paramétereire közvetlenül hivatkozhatunk.

1.18. Példa (Miranda). (Lokális deklarációk).

```
masodfoku :: num->num->num->[num]
masodfoku a b c = error "nem másodfokú",        if a = 0
                = error "komplex gyökök",       if delta < 0
                = [-b/(2*a)],                   if delta = 0
                = [-b/2*a + radix/(2*a),
                   -b/2*a - radix/(2*a) ],      otherwise
                where
                delta = b* b - 4 * a * c
                radix = sqrt delta
```

1.19. Példa (Haskell). (Lokális deklarációk).

```
masodfoku :: Float->Float->Float->[Float]
masodfoku a b c | a == 0        = error "nem másodfokú"
                | delta <  0    = error "komplex gyökök"
                | delta == 0    = [-b/(2*a) ]
                | delta >  0    = [-b/(2*a) + radix/(2*a),
                                   -b/(2*a) - radix/(2*a) ]
                                where
                                delta  = b* b - 4.0 * a * c
                                radix  = sqrt delta

masodfoku 1 (-3) 2    -- Megj.: a masodfoku 1 -3 2  rossz!
```

1.7. Példa (Clean). (Lokális deklarációk).

```
masodfoku :: Real Real Real -> (String,[Real])
masodfoku a b c
 | a == 0.0     = ("nem másodfokú",[])
 | delta < 0.0  = ("komplex gyökök",[])
 | delta == 0.0 = ("egy gyök",[~b/(2.0*a)])
 | delta > 0.0  = ("ket gyök", [(~b+radix)/(2.0*a),
                                (~b-radix)/(2.0*a)])
 where
     delta = b*b-4.0*a*c
     radix = sqrt delta
```

1.5. Példa (SML). (Lokális deklarációk).

```
fun masodfoku a b c   =
   let
      val    delta  = b* b - 4.0 * a * c;
      val radix  = Math.sqrt delta;
   in
      if  a = 0.0           then ("nem másodfokú",[])
      else if  delta  < 0.0 then ("komplex gyökök",[])
      else if  delta  = 0.0 then ("egy gyök",[~b/(2.0*a)])
      else  ("ket gyök", [(~b+radix)/(2.0*a),
                                (~b-radix)/(2.0*a)])
   end
```

1.3.3. Egy érdekesebb példa: királynők a sakktáblán

Helyezzünk el egy `n*n`-es sakktáblán `n` királynőt úgy, hogy egyik sem üti a másikat! Két királynő akkor üti egymást, ha azonos oszlopban, azonos sorban vagy azonos átlón helyezkednek el. Adjuk meg az összes megoldást!

A megoldásokat `n` hosszúságú sorozatok sorozata formájában adjuk meg. Egy-egy megoldás azt írja le, hogy balról jobbra az egyes oszlopokban hányadik sorban helyezzük el a királynőket. Egy oszlopba tehát semmiképpen sem kerül két királynő.

Az alábbi SML-megoldás a `List.tabulate(n, fn x => x+1)`[36] sorozat, azaz az `[1,2,..,n]` sorozat permutációi között keresi a megoldásokat, így azonos sorban nem lehet két királynő. Egy-egy permutáció vizsgálatakor (`diag`) csak azt kell ellenőrizni, hogy nincs-e két királynő azonos átlón. A jó

[36] A `tabulate` függvény n elemű listát készít, a lista elemeinek értékeit a függvényargumentum [0..n-1] sorozatra történő elemenkénti alkalmazásával kapjuk meg.

permutációkat az `accuqueens` gyűjti össze, a `cycle` pedig a következő permutációt készíti el. `accuqueens` és `cycle` kölcsönösen rekurzív, ennek megfelelően az `and` kulcsszó köti össze a két deklarációt. `@` két sorozatot fűz össze, a `(right as r::rr)` *összefogó minta* pedig lehetővé teszi, hogy ugyanarra a sorozatra a `right` azonosítóval hivatkozzunk, amelynek első elemét `r`-rel, maradékát pedig `rr`-rel jelöljük.

1.6. Példa (SML). (Királynők).

```
local
 fun diag' d1 d2 []       = true
  |  diag' d1 d2 (q1::qr) =
     d1<>q1 andalso d2<>q1 andalso
            diag' (d1+1) (d2-1) qr
 fun diag mid right = diag' (mid+1) (mid-1) right

 fun accuqueens []    tail res = tail :: res
  | accuqueens (x::xr) tail res = cycle [] x xr tail res
 and cycle left mid [] tail res =
          if diag mid tail then
             accuqueens left (mid :: tail) res else res
  | cycle left mid (right as r::rr) tail res =
          cycle (mid::left) r rr tail
               (if diag mid tail then
             accuqueens (left@right) (mid :: tail) res
               else res)
in
    fun queens n =
      accuqueens (List.tabulate(n, fn x => x+1)) [] []
end
```

Az alábbi program fokozatosan próbálja kiterjeszteni a megoldást jobbról balra, oszlopról oszlopra haladva úgy, hogy az új királynő a már elhelyezett királynők egyikét se üsse. Megfigyelhetjük, hogy generátorok alkalmazásával nagyon tömör és jól olvasható megoldáshoz jutunk:

1.20. Példa (Haskell). (Királynők).

```
doqueens  :: Int -> [[Int]]
doqueens n = queens n n  -- n kiralynő, n*n-es táblán
```

```
queens :: Int Int -> [[Int]]
queens n 0     = [ [] ]
queens n (m+1) = [ q:b | b<-queens n m, q<-[0..n-1], safe q b ]
```

A `queens` függvény megkeresi a következő oszlopban a jó helyeket. `queens m` az összes jó elhelyezés az utolsó `m` oszlopban, amelyek közül `b` egy jó elhelyezés. `safe q b` teljesül, ha a `q`-adik sorba tett királynő nem üti azokat a korábbiakat, amelyeket a `b` tartalmaz.

```
safe :: Int -> [Int] -> Bool
safe q b  = and [ not (checks q b i) | i<-[0.. length b - 1 ]]
```

A `safe` függvény ellenőrzi, hogy a `q`-adik sorba tett királynő nem üti a `b` -belieket. `i` befutja a `b`-beli oszlopindexeket. `check q b i` teljesül, ha az aktuális oszlop `q`-adik sorába tett új királynő üti a `b` felállás `i`-edik oszlopában levőt. Az `and` függvény elemenkénti felhasználás, értéke igaz, ha az argumentumában megadott sorozat minden eleme igaz.

```
checks ::  Int -> [Int] -> Int -> Bool
checks q b i = (q == (b !! i)) || (abs(q - (b !! i))==(i+1))
```

`b !! i` jelentése: a `b` sorozat `i`-edik eleme, az `i`-edik királynő helye. || a logikai vagy művelet, `abs` az abszolútérték függvény.

```
doqueens 8
```

1.21. Példa (Miranda). (Királynők).

```
queens n 0 = [ [] ]
queens n (m+1) = [ q:b | b <- queens n m; q <- [0..n-1]; safe q b ]
safe q b = and [ ~checks q b i | i <- [0..#b -1 ] ]
checks q b i = (q = b!i) \/ abs(q - b!i)=(i+1)
queens 8 8
```

1.8. Példa (Clean). (Királynők).

```
queens_ n 0 = [[]]
queens_ n m = [ [q:b] \\ b <- queens_ n (m-1), q <- [0..n-1]  | safe
safe q b = and [not (checks q b i) \\ i <- [0 .. (length b)-1] ]
```

```
checks q b i = (q == b!!i) || (abs(q-b!!i)==(i+1))
queens n = queens_ n n
Start = (length(queens 8),queens 8)
```

A lusta kiértékelés lehetőséget ad *rekurzív hívások helyettesítésére* végtelen sorozatok felhasználásával.

1.22. Példa (Miranda, Clean és Haskell). (Fibonacci számok).

```
fib :: Int -> Int   -- Miranda: num->num
fib 0     = 1       -- elhagyható
fib 1     = 1
fib 2     = 2
fib n = flist!!(n-2) + flist!!(n-1) -- Miranda: !! helyett !
          where
          flist = map fib [ 0 .. ]
```

A `map` függvény elemenként alkalmazza az első argumentumként kapott függvényt a másodikként kapott sorozatra. Az `flist` tehát tartalmazza a Fibonacci számokat rendre a `0`-diktól kezdve az *összes* természetes számra. Így az `n`-edik Fibonacci szám könnyen kiszámítható, mint az `flist` sorozat `n-1`-edik és `n-2`-edik elemének összege. Az `flist` lista összes elemét természetesen nem számítjuk ki, a lusta kiértékelés miatt csak azokat határozzuk meg, amelyekre valóban szükség van.

1.4. Típusok és osztályok

Ebben a részben bemutatjuk, hogy a funkcionális nyelvekben milyen nyelvi elemek segítik az adatabsztakciót, hogyan lehet absztrakt, algebrai típusokat definiálni, típusosztályokat megadni és magasabbrendű típusokat alkalmazni.

1.4.1. Polimorfizmus, típusosztályok

A korábbiakban már több olyan függvénnyel találkoztunk, amely egymástól különböző típusú argumentumokra is alkalmazható. Ezeket a függvényeket *polimorfnak* nevezzük. Ilyen például a sorozatok első elemét visszaadó `hd` függvény, amely tetszőleges (nem üres) listára alkalmazható, függetlenül a lista elemeinek típusától. A függvény megvalósítása sem függ a típustól, ezért definíciója is megadható típusfüggetlen formában. A függvény típusának deklarációjakor egy *típusváltozót*

használunk, ezzel fejezzük ki, hogy a függvény polimorf, bármilyen konkrét típusra alkalmazható.

```
hd  :: [a] -> a   // Miranda, Clean, Haskell
hd (x:xs) = x
```

A + műveletet is számos konkrét típus esetén használhatjuk, például egészekre, valósakra, de akár logikai értékekre is értelmezhetjük. Az összeadás minden esetben két azonos típusú értékből határoz meg egy harmadik, ugyanolyan típusú értéket, és alkalmazása is egységesen infix. A `hd` függvénynél megismert egyszerű polimorfizmussal szemben az összeadás megvalósítása minden konkrét típus esetén más és más. Például egész, illetve valós értékeket bináris alakban általában különböző módon ábrázoljuk, így az összeadás műveletét is másképp kell elvégezni. A polimorfizmus ebben az esetben tehát más jellegű, itt valójában több, egymástól különböző függvényre használunk azonos jelölést, a művelet jelét többszörösen használjuk[37]. Az SML csak egyes előre definiált függvények esetében engedi meg azonosítók többszörös használatát. A Cleanben és a Haskellben a programozó is deklarálhat többszörösen használt azonosítókat, de csak akkor ha a megvalósításukban különböző függvények néhány lényeges dologban azért megegyeznek (paraméterek száma, infix alkalmazás, asszociativitás, stb.). Ebben az esetben ezen műveleteknek létezik egy közös, típusfüggetlen, absztrakt szignatúrája.

A függvény *szignatúrájának* nevezzük paraméterei és eredménye típusának leírását. A + művelet absztrakt leírását típusváltozó segítségével fogalmazhatjuk meg, kicsit leegyszerűsítve például a `(+) :: a a -> a` alakban. Ebből az absztakt szignatúrából az `a` típusváltozó konkrét típussal történő helyettesítésével, *példányosítással* kapjuk az egyes típusokhoz tartozó különböző összeadás műveletek szignatúráit. Ezek a műveletek megvalósításukban is eltérnek egymástól, ezért példányosításkor meg kell adnunk az adott típus értékeire vonatkozó összeadás művelet definícióját is. Az absztrakt szignatúrát Cleanben és Haskellben a `class`, a példányosítást az `instance` kulcsszó vezeti be. Az absztrakt szignatúrák és azok példányosításának megfelelő nyelvi elemek SML-ben nem az osztályok, hanem az SML modulnyelvéhez tartozó szignatúrák, struktúrák és funktorok (ld. 1.5.).

Egy `class` deklaráció több egymáshoz tartozó absztrakt szignatúrát is összefoghat. Egy adott `class` deklaráció absztrakt szignatúráihoz tartozó példányok halmazát *típusosztálynak* nevezzük[38]. Ebben az esetben az absztrakt szignatúra de-

[37] ad hoc polimorfizmus, túlterhelés

[38] A típusosztály elemei függvények, absztrakt függvények különböző konkrét megfelelői. A *függvényosztály* elnevezés alkalmazása ebből a szempontból kifejezőbb lenne. Típusosztályról ab-

finiálja azt is, hogy a művelet infix és balról asszociatív, amit az `infixl` kulcsszó utolsó betűje azonosít (az angol left szó kezdőbetűje). Meghatározzuk a művelet precedenciaszintjét is, ez az összeadás esetében `6`.

1.9. Példa (Clean). (Típusosztály és példányosítás).

```
class (+) infixl 6 a :: !a !a -> a // absztrakt (+)
double :: a  -> a | + a
double n :== n + n
 instance + Bool
 where
 (+) :: !Bool !Bool -> Bool   // peldány
 (+) True b = True
 (+) a    b = b
```

A `double` függvény is polimorf, de nem kell példányosítani. A `double` definíciója a `+` definíciójától függ, ezt a szignatúrájában jelezzük is. Azt mondjuk, hogy a `double` egy *származtatott függvény*, értelmezése attól függ, hogy az adott típushoz definiáltuk-e, és hogy hogyan definiáltuk a `+` *típusosztály* megfelelő példányát.

A több szignatúrát összefogó `Num` típusosztályra Haskell nyelvű példát mutatunk:

1.23. Példa (Haskell). (Numerikus típusosztály).

```
class Num a where
 (+)    :: a -> a -> a
 negate :: a -> a
```

A Haskell és a Clean típusrendszere megkülönbözteti a típusállandókat a típusváltozóktól. A típusváltozók kis betűvel kezdődnek. A fenti példákban `a` típusváltozó. Típusállandók például a `Char, Int, Integer, Float, Double` és `Bool` típusok.

Az osztályok azonosítói általában nagybetűvel kezdődnek. Egy típus vagy függvény deklarációjánál *osztálykörnyezetet* adhatunk meg, azaz a típus vagy függvény deklarációjában, definíciójában szereplő típusváltozók lehetséges értékét osztályok megadásával korlátozhatjuk:

1.24. Példa (Clean és Haskell). (Osztálykörnyezet megadása).

ban az értelemben beszélhetünk mégis, hogy meghatározható azon konkrét típusok halmaza is, amelyekre a `class` deklarációban megadott absztrakt függvények mindegyikének létezik példánya.

```
double :: Num a => a -> a -- Haskell
double :: a -> a  |  +  // Clean
double x = x + x
```

A `double` tehát a `Num` típusosztályhoz tartozó `Int` típusú értékekre alkalmazható, de a típusosztályhoz általában nem tartozó `Char` típus értékeire nem.

Egy `C1` osztálykörnyezet megadásával azt írjuk elő, hogy az `a` típusváltozó értéke csak olyan t típus lehet, amely a `C1` osztályhoz tartozik.

A függvényalkalmazás fogalmának felel meg a *típusalkalmazás* fogalma. Ha a t1 típus például k1→k2 alakú és a t2 típus k1-re illeszkedik, akkor a t1 t2 típuskifejezés értéke a k2 típus (típuslevezetés). A típusváltozók általában implicit univerzális kvantor hatáskörében állnak.

Egy kifejezés típusának mindig levezethetőnek kell lennie a Hindley–Milner féle típuslevezetési eljárás szerint. Az eredmény függ a típuskörnyezettől, amely meghatározza a kifejezésben szereplő szabad változók típusát, illetve az osztálykörnyezettől, amely a típusváltozók értékét korlátozza. A polimorf `double` függvény típusa általánosabb az `Int -> Int` típusnál, a legáltalánosabb típusleírás `a a -> a | + a` alakú. A kifejezés legáltalánosabb típusa a kifejezés alaptípusa, amely az osztálykörnyezet segítségével általában akkor is levezethető, ha a kifejezésben szereplő egyes függvénynevek többszörösen használtak (több típushoz is tartoznak). Néhány esetben meg kell azonban adnunk a szignatúrát, mert a fordító nem tudja meghatározni a legáltalánosabb típust. Az azonosítók többszörös használata viszonylag gyakori oka az automatikus típuslevezetés sikertelenségének, az SML éppen ezért korlátozza ezt a lehetőséget.

1.1. Definíció. (általánosabb típus). *Jelöljön u, v típusváltozó halmazokat, c_1, c_2 ezen típusváltozókra vonatkozó osztálykörnyezeteket, t_1, t_2 pedig ezeket a típusváltozókat is tartalmazó típuskifejezéseket. A $\forall u.c_1 \Rightarrow t_1$ típus(kifejezés) általánosabb a $\forall w.c_2 \Rightarrow t_2$ típusnál, ha van olyan S helyettesítés, amely az u-beli típusváltozókat úgy helyettesíti, hogy t_2 megegyezik $S(t_1)$-gyel és ha c_2 teljesül, akkor $S(c_1)$ is.*

Típusokat megadhatunk algebrai adattípusként, származtatott típusként, és típusszinonímaként. A típuskonstrukció-osztályokat az algebrai adattípus fogalmának bevezetése után ismertetjük.

1.4.2. Algebrai adattípus

Algebrai adattípus deklarációjában egyidejűleg vezetünk be egy új típust (típuskonstrukciót) és annak adatkonstruktorait. Kivétel nélkül minden, az adott típushoz tartozó érték a típus deklarációjakor megadott valamelyik adatkonstruktor

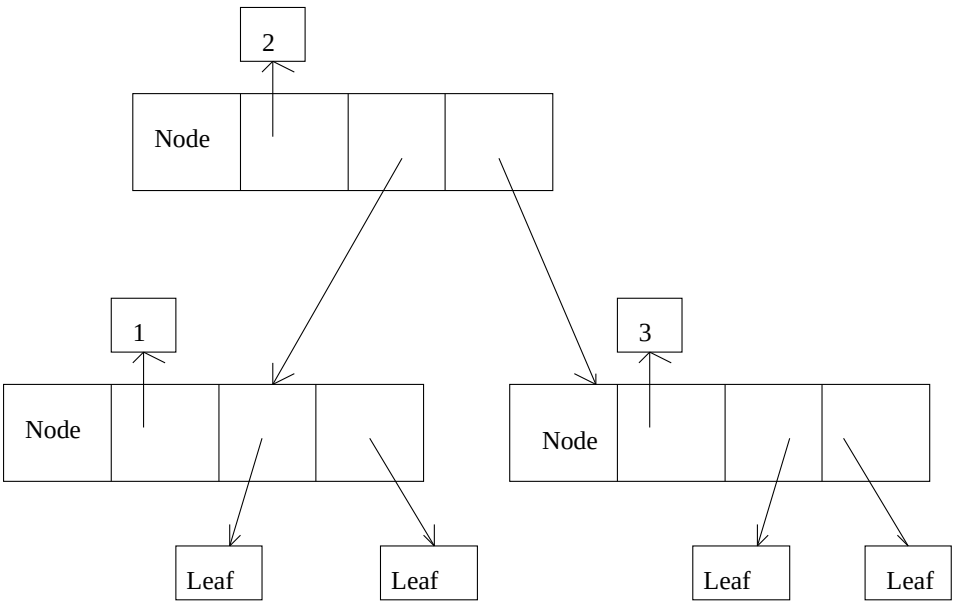
függvény alkalmazásával áll elő. A következő példában egy felsorolási típus algebrai típusdefinícióját adjuk meg. A `Nap` konstans (nulláris) típuskonstruktor, a `Het`, a `Kedd`, stb. pedig konstans adatkonstruktor függvények, melyek értéke `Nap` típusú.

```
:: Nap = Het | Kedd | Sze | Csu | Pen | Szo | Vas
```

A `Tree` típuskonstrukció tetszőleges alaptípusból bináris fa típust hoz létre; a `Tree` típuskonstruktor egyváltozós, argumentuma az `a` típusváltozó. Típuskonstruktorokat tekinthetjük magasabbrendű típusoknak is, melyek egy típusból egy másik típust hoznak létre. A típuskonstrukció adatkonstruktorai közül a `Node` adatkonstruktor-függvény háromváltozós, első paramétere egy `a` alaptípusú érték, amit a fa csúcsában tárol, második és harmadik paramétere pedig egy-egy `Tree` típusú érték. A deklaráció arra is példa, hogy a típuskonstruktort rekurzívan is használhatjuk.

```
:: Tree a = Node a (Tree a) (Tree a)
          | Leaf

atree = Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf Leaf)
```



1.4. ábra. Fa típusú érték: `atree`.

A konstruktorok hatásköre és láthatósága kiterjed arra a teljes modulra, amely a deklarációt tartalmazza.

A Mirandában a `::=` jel vezeti be az algebrai adattípust és a `*` jelöli a típusváltozót.

1.25. Példa (Miranda). (Algebrai adattípus).

```
bool ::= True | False
nat ::= Zero | Suc nat
list * ::= Nil | Cons * (list *)
color ::= Red | Orange | Yellow | Green | Blue | Indigo | Violet
bool_or_num ::= Left bool | Right num
tree * ::= Nilt | Node * (tree *) (tree *)
```

A Haskellben a `data`, az SML-ben a `datatype` kulcsszó vezeti be az algebrai típusdefiníciót. A halmaz típus algebrai definíciójára alább egy Haskell nyelvű példát adunk.

1.26. Példa (Haskell). (Algebrai adattípus).

```
data Eq a => Set a = NilSet | ConsSet a (Set a)
```

Az adatkonstruktorok típusa:

```
\smallskip
NilSet :: Set a
ConsSet :: Eq a => a -> Set a -> Set a
```

A `Set` típuskonstruktor és a `ConsSet` konstruktor egyaránt csak az `Eq` típusosztályhoz tartozó típus esetében alkalmazható. A
`data ( C1 a b , C2 b ) => T a b = ....`
általános alakú típusdefinícióban a típusváltozók lehetséges értékeire osztálykörnyezet megadásával megszorításokat írhatunk elő.

Az egyes konstruktorokhoz tartozó mezőknek nevet is adhatunk:
`data ( C1 a b , C2 b ) => T a b = K1 {f1 :: a, f2 :: b} ....` Ily módon Haskellben is létrehozhatunk rekordra emlékeztető olyan direktszorzat adattípust, ahol mintaillesztést alkalmazva mezőnevekkel dolgozhatunk.

1.27. Példa (Haskell). (Rekord).

```
data Szemely = Ferfi {nev::[Char], kor::Int}
                     | No {nev::[Char]}
```

Ezzel valójában egy variáns rekordot hoztunk létre. Ekkor anélkül, hogy előre tudnánk, hogy az aktuális `x` `Szemely` férfi vagy nő, egyetlen `case` kifejezéssel definiálhatunk egy olyan típusértéket, ahol a név mező értéke `"Gabi"`:

```
case x of Ferfi _ kor -> Ferfi "Gabi" kor
          No _   -> No "Gabi"
```

Az alábbi SML példa a lista típus egy lehetséges algebrai definíciója. Az SML-ben a típusváltozót mindig egy `'` jel előzi meg és a típusváltozó a típuskonstruktor előtt van. Az adatkonstruktort az `of` kulcsszó választja el argumentumától.

1.7. Példa (SML). (Algebrai adattípus).

```
datatype 'a List = Nil | Cons of 'a * 'a List
```

A Mirandában az `abstype` kulcsszó segítségével deklarálhatunk absztrakt adattípust, míg az SML-ben, Clean-ben és Haskellben a modulrendszer az adatabsztrakció megvalósításának eszköze (1.5.).

1.28. Példa (Miranda). (Absztrakt, algebrai adattípus specifikáció).

```
abstype tree *
with mirror :: (tree *) -> (tree *)
     empty  :: (tree *)

numtree ::= tree num
```

A most bevezetett fa típushoz mindössze két műveletet deklaráltunk, az `empty`-t, amelynek értéke egy üres fa, illetve a `mirror`-t, amely egy tetszőleges fát tükröz. A `tree` típuskonstrukció algebrai típusdefiníciója rejtett, az adatkonstruktorok hatásköre pedig korlátozott, a program más részeiben nem használhatók.

```
tree * == Nilt | Node * (tree *) (tree *)
mirror Nilt = Nilt
mirror (Node a x y) = Node a (mirror y) (mirror x )
```

1.4.3. Típusszinonimák

Típusszinonima deklarációja nem vezet be új típust; a szinonima ekvivalens az eredeti típuskifejezéssel, szintaktikai rövidítésnek tekinthető. Típusszinonimára vonatkozó rekurzió, illetve példányosítás nem megengedett.

1.29. Példa (Miranda). (Típusszinonimák).

```
string == [char]
matrix == [[num]]
```

1.30. Példa (Haskell). (Típusszinonimák).

```
type List = []
type Rec a = [Circ a]
data Circ a = Tag [Rec a]
```

1.4.4. Származtatott típusok

A Haskellben az eredeti típussal azonos reprezentációjú, de nem ekvivalens típust kapunk a `newtype` kulcsszó használatával végzett származtatással. A származtatott típusok esetében a példányosítás is megengedett.

1.4.5. Típuskonstrukció osztályok

A Clean-ben készíthetünk olyan osztályokat is, amelyek példányait nem típusokra, hanem típuskonstrukciókra definiáljuk. Ilyen a `length` vagy a `map` függvény, amely egy tetszőleges adatszerkezet elemeinek számát határozza meg, illetve tetszőleges adatszerkezetre elemenként alkalmaz egy függvényt.

Az osztály deklarációjában a `t` típuskonstruktor változó szerepel, amely felveheti a lista vagy a fa[39] típuskonstruktor értékét.

1.10. Példa (Clean). (Típuskonstrukció osztály).

```
class map t :: (a->b) (t a) -> (t b)
instance map []
  where map f l = [ f e \\ e <-l]
:: MTree a = NNode a [MTree a]
instance map MTree
  where map f (NNode el ls) =
          NNode (f el) (map (map f) ls)
```

[39] Ebben a példában egy új - nem bináris - fa típuskonstrukció szerepel.

A Haskellben a `Functor` monadikus osztály (1.6.) általánosítja a `map` függvényt típuskonstruktorokra, `fmap` néven.

1.5. Modulok

Míg a Haskellben csak egyfajta modul létezik, a Cleanben definíciós és implementációs modulokat hozhatunk létre. A definíciós modulhoz általában tartozik egy implementációs modul is, ketten együtt alkotnak egy egészet. A definíciós modul interfészt nyújt az implementáiós modul felhasználói számára; ők csak azt látják a modulból, amit a definíciós modulban megadtunk. A definíciós/implementációs modulpár a Clean nyelvben az adatabsztrakció eszköze. Clean-ben a modulok az `import` utasítás használatával hivatkoznak egymásra, az importálás tranzitív. A Haskellben a definíciós modult az export lista helyettesíti, ennek segítségével adjuk meg, hogy milyen függvények, illetve adattípusok legyenek a modulon kívül láthatók. Az export lista a modul fejlécében helyezkedik el.

1.31. Példa (Haskell). (Modulok).

```
module Uj ( fv1 ) where
fv1 x = x
fv2 x = 3 + x
```

Mindenki, aki az `Uj` modult importálja, csak az `fv1`-et fogja látni, az `fv2`-t nem. Ha nincs export lista, minden, amit a modulban definiáltunk látható kívülről.

Mindkét nyelv lehetőséget ad arra is, hogy egy modulból ne importáljunk mindent, amit az exportál, hanem csak azt, amire valóban szükségünk van.

A Haskell és az SML lehetőséget ad minősített nevek használatára is. Ezek akkor hasznosak, ha más modulokban definiált azonosítókat vegyesen szeretnénk használni. Előfordulhat ugyanis, hogy ugyanaz az azonosító több importált modulban is szerepel, esetleg más funkcióval. A minősített név a következő alakú: *minősítő.azonosító*; a minősítő általában a modul neve, de megadhatunk más nevet is a modul importálásakor.

1.5.1. Absztrakt algebrai adattípus

A 1.11. példa segítségével bemutatjuk, hogy a Clean modulszerkezetének segítségével hogyan valósíthatunk meg absztrakt adattípust. A verem típus reprezentációját az implementációs modulban adjuk meg, típusszinonima formájában. A reprezentációt a definíciós modul nem tartalmazza, tehát a vermet reprezentáló listát az implementációs modulon kívül más modulokból nem érhetjük el, csak a definíciós modul által exportált absztrakt műveleteket használhatjuk.

1.11. Példa (Clean). (Modulok és absztrakt adattípus).

```
definition module stack
:: Stack a
Push :: a (Stack a) -> Stack a
Pop  :: (Stack a) -> Stack a
top  :: (Stack a) ->  a
Empty :: Stack a

implementation module stack
:: Stack a :== [a]
Push :: a (Stack a) -> Stack a
Push e s = [e:s]
Pop  :: (Stack a) -> Stack a
Pop [e:s] = s
top  :: (Stack a) -> a
top [e:s] = e
Empty :: Stack a
Empty = []

module teszt
import StdEnv, stack
Start = top (Push 1 Empty)
```

Az SML modulszerkezete gazdagabb, a modulok kezeléséhez használható nyelvi elemeket az SML modulnyelve írja le [Har01]. A teljes modulnyelv leírása meghaladja a fejezet kereteit, csak néhány fontos nyelvi elemet emelünk ki.

A Clean definíciós moduljainak a *szignatúrák*[40] felelnek meg, a *struktúrák* pedig az implementációs moduloknak. Fontos eltérés azonban, hogy az SML ismeri a szignatúrakifejezés és a struktúrakifejezés fogalmait is, amelyeknek értékük is van és argumentumként is átadhatók.

A szignatúrákat tekinthetjük a struktúrák típusleírásának. Minden struktúrának létezik elsődleges, levezethető szignatúrája, amely a megfelelési reláció[41] [Har01] szerint összehasonlítható egy specifikációt megjelenítő szignatúrával.

A szignatúrákra vonatkozóan kétféle öröklődési relációt is értelmez az SML modulnyelve. Egy szignatúra kiterjeszthet (tartalmazhat) egy másikat, illetve finomítása (specializációja) lehet egy másiknak (1.8. példa). A kiterjesztés alkalmas

[40] Itt a szignatúra szó az SML nyelvi elemét azonosítja. Az SML szignatúrái egyfajta általánosítását jelentik a függvények szignatúráinak, egy egész struktúra interfészét írják le.

[41] pre-order

lehet új műveletek bevezetésére, a finomítás pedig a reprezentáció módját előírva, a szignatúrát a benne deklarált absztrakt típusokra adott definíciókkal egészítheti ki.

1.8. Példa (SML). (Szignatúrák – kiterjesztés, finomítás).

```
signature QUEUE =
  sig
    type 'a queue
    exception Empty
    val empty : 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val remove : 'a queue -> 'a * 'a queue
  end

signature QUEUE_WITH_EMPTY =
  sig
    include QUEUE
    val is_empty : 'a queue -> bool
  end

signature QUEUE_AS_LISTS =
  QUEUE where type 'a queue = 'a list * 'a list
```

Az importálás megfelelője a szignatúrák közötti öröklődési, illetve a nyitott struktúra (`open`), amelynek alkalmazása esetén az importált struktúra teljes belső tartalma láthatóvá válik az azt használó modulban.

Egy struktúra akkor felel meg egy szignatúrának, ha elsődleges szignatúrája tartalmazza mindazt, amit a szignatúra előír .

Az `ORDERED` szignatúra az `ORD` osztály (ld. 1.4. rész) SML-megfelelője, amelynek egy lehetséges megvalósítása a `LessInt` struktúra (1.9. példa). A megfelelési relációt fejezi ki a `:` jelölés, amit úgy is olvashatunk, hogy a `LessInt` struktúra `ORDERED` típusú.

1.9. Példa (SML). (Szignatúra és példány).

```
signature ORDERED =
  sig
    type t
    val lt : t * t -> bool
```

```
    val eq : t * t -> bool
  end
structure LessInt : ORDERED =
  struct
    type t = int
    val eq = (op =)
    val lt = (op <)
  end
```

Ahogy az osztályokhoz több példány is tartozhat, úgy a szignatúráknak is több különböző megvalósítása lehet egyszerre. Ezek közül néhány egy-egy másik struktúrában beágyazva is megjelenhet.

A szignatúrák implementációja esetén a megfelelő struktúra létrehozásakor a `:>` előírás alkalmazásával rendelkezhetünk arról is, hogy a struktúrából – a struktúra használatakor – csak azok az egyedek legyenek majd láthatók, amelyek a szignatúrában is megjelennek. Az ily módon létrehozott struktúra az *adatabsztrakció* eszköze. A 1.10. példa az absztrakt sor megvalósítását mutatja be [Har01]. A sort listák rendezett párja reprezentálja. A `QUEUE_WITH_EMPTY` szignatúra (ld. 1.8. példa) tartalmazta a típusspecifikációt, amelynek úgy felel meg a `Queue` struktúra, hogy a `:>` előírás szerint belső szerkezete *átlátszatlan* marad.

1.10. Példa (SML). (Modulok és absztrakt adattípus).

```
structure Queue :> QUEUE_WITH_EMPTY =
  struct
    type 'a queue = 'a list * 'a list
    val empty = (nil, nil)
    fun insert (x, (bs, fs)) = (x::bs, fs)
    exception Empty
    fun remove (nil, nil) = raise Empty
      | remove (bs, f::fs) = (f, (bs, fs))
      | remove (bs, nil) = remove (nil, rev bs)
  end
```

Az SML-ben arra is lehetőségünk van, hogy modulokat struktúrákkal paraméterezzünk. A paraméterezett modul nyelvi megfelelője a `functor`, amely az Ada sablonhoz hasonló eszköz. A 1.11. példán [Har01] megfigyelhetjük, hogy a `DICT` szignatúrának megfelelő `DictFun` funktort a `K` struktúrával paraméterezzük. A `dict` típus egy olyan típuskonstrukciót specifikál, amelyben az elemeket, amelyek (kulcs, érték) párból állnak, kulcs szerint kereshetjük. A `DictFun` struktúra egy olyan algebrai adattípussal megfogalmazott adatszerkezettel reprezentálja

a szignatúrában megadott típust, amelyben a kulcs típuson értelmezett rendezési reláció alapján a keresés hatékonyan megvalósítható. Ezért a kulcs típust leíró, paraméterként kapott `K` struktúráról kikötjük, hogy megfelel az `ORDERED` szignatúrának (ld. 1.9. példa). A reprezentáció és az implementáció a `:>` előírás szerint átlátszatlan, a megvalósítás részleteit most mi is mellőzzük.

A `K` aktuális értéke lehet például a `LessInt` struktúra is (ld. 1.9. példa), amely az egész számokra a szokásos módon értelmezi a rendezési relációt. Megjegyezzük, hogy a `LessInt` úgy felel meg a az `ORDERED` struktúrának, hogy a struktúra belső szerkezete látható marad (a `:` előírásnak megfelelően), tehát nem rejtjük el, hogy a (`K.t`) kulcs típust az egész számok reprezentálják.

Az `LtIntDict` struktúra a `DictFun` funktor egy olyan példánya lesz, ahol a kulcs típus az egész számok típusa, míg a (kulcs,érték) párban elhelyezett értékek típusa (az `LtIntDict.dict` típuskonstruktor paramétere) továbbra is szabadon megválasztható.

1.11. Példa (SML). (Funktor).

```
signature DICT =
  sig
    structure Key : ORDERED
    type 'a dict
    val empty : 'a dict
    val insert : 'a dict * Key.t * 'a -> 'a dict
    val lookup : 'a dict * Key.t -> 'a option
  end

functor DictFun (structure K : ORDERED) :> DICT
  where type Key.t = K.t =
struct
  structure Key : ORDERED = K
  datatype 'a dict = ...  (* reprezentáció *)
  val empty = Empty
  fun insert  ... =  ...  (* implementáció *)
  fun lookup  ... =  ...
end

structure LtIntDict = DictFun (structure K = LessInt)
```

1.6. Frissíthető változók, monádok, mellékhatás

A bemeneti/kimeneti műveletek szükségszerűen mellékhatással járnak, megváltoztatják a program környezetét, a képernyő tartalmát, háttértárolón elhelyezett fájlokat, stb. A tisztán funkcionális nyelvekben is vannak olyan elemek, amelyeknek van mellékhatásuk, különben nem lehetne ezeken a nyelveken a külső környezetre ható programot írni. A mellékhatásokat lehetőség szerint a programszöveg jól azonosítható részeire kell korlátozni, és a hivatkozási helyfüggetlenség sérülését is el kell kerülni.

1.6.1. Egyszeresen hivatkozott változók

A Clean tervezői bizonyos korlátok között megengedik az előző értéket megsemmisítő értékadást. Egy *egyszeresen hivatkozott*[42] objektumnak újra értéket adhatunk, de csakis akkor, ha az objektumra csak összesen egyetlen helyen hivatkozunk. Ekkor ezen a helyen biztonságosan megváltoztathatjuk az egyed által elfoglalt memóriaelemben tárolt értéket, hiszen senki más nem használja többé a korábbi értékét, a hivatkozási helyfüggetlenség ilyenkor nem sérül. A régi egyed többé nem létezik, az általa korábban elfoglalt memóriaterületet a szemétgyűjtő helyett rögtön egy új egyednek adjuk át. Ez hatékonysági szempontból gyakran előnyös lehet, például nagy adatszerkezetek használata esetén. Ha az adatszerkezet egy olyan változatára van szükségünk, amelyik csak egyetlen helyen különbözik az egyszeresen hivatkozott eredetitől, akkor a megfelelő elem felülírása megengedett és hatékonyan megvalósítható. Fájlműveletek esetén is hasonló a helyzet, nagy fájlok másolatainak elkészítése helyett a fájl megfelelő rekordját írhatjuk felül.

A Cleanben az egyszeresen hivatkozott vázzal rendelkező tömbök tartalmát is újra definiálhatjuk. A 1.12. példában a `mArray5` egy olyan tömb, amely váza a memóriában az egyszeresen hivatkozott vázzal rendelkező `Array5` tömb vázának helyére kerül, és elemei megegyeznek a `Array5` tömb elemeivel, kivéve a `3`-as és `4`-es indexhez tartozó értékeket, amelyek új értékét az `&` jel után adtuk meg. Az `mArray` tömbben minden egyes érték más lesz, mint az `mArray`-ben azonos indexhez tartozó érték[43]

1.12. Példa (Clean). (Tömbök).

```
Array5 :: *{Int}
mArray5 = { Array5 & [3]=3, [4]=4 }
mArray  = { Array5 & [i]=k \\ i <- [0..4] & k <-[80,70..]}
```

[42] unique

[43] A második `&` jel a `//` után megadott két listagenerátor párhuzamos működését írja elő, így a tömb értéke `[80,70,60,50,40]` lesz.

Azt, hogy nincs-e egynél több hivatkozás az egyedre, a fordító ellenőrzi. Minden "értékadáskor" új nevet is kellene adunk a létrejövő új egyednek, de mivel több hivatkozás már nincs az adott nevű régi egyedre, egy lokális definíció alkalmazásával a nevet is gyakran újra felhasználjuk. Ettől kezdve a név már az új egyedet azonosítja. Ezzel a módszerrel az imperatív stílushoz hasonlóan programozhatunk (amikor ez feltétlenül szükséges), anélkül, hogy a tisztán funkcionális programozás kereteit átlépnénk. A megsemmisítő értékadások használatához szükség van soorendjük meghatározására is, különben nem lehetne tudni, hogy az újra felhasznált nevek éppen melyik értékadás eredményét jelölik. Erre mutatunk példát a 1.13. programban.

1.6.2. Monádok

A monád[44] egy kategória-elméleti fogalom, három művelet algebrai tulajdonságainak leírására szolgál [BaWe90]. A Haskellben ennek az algebrai fogalomnak egy programozási nyelvi megfelelőjével találkozunk.

A Haskell három monadikus osztályt definiál: a `Functor` (1.4.5.), `Monad` és `MonadPlus` osztályokat. Ezeket olyan típuskonstrukciókkal példányosíthatjuk, mint például az IO típuskonstrukció (ld. 1.7. rész).

A monadikus osztályok absztrakt műveleteihez olyan aximómák tartoznak, amelyeket az osztály minden példányosításakor figyelembe kell venni. Például a `Functor` monád `fmap` függvényére teljesülnie kell az elemenként feldolgozható függvényekre [Fót83] ismert szabályoknak: `fmap id = id`, `fmap (f . g) = fmap f . fmap g`.

A `Monad` és `MonadPlus` monádosztályok példányaihoz tartozó alapműveletek a monád értékeinek, az ún. *monadikus műveleteknek* az[45] összekapcsolását segítik, ezáltal a monadikus műveletek kiértékelésének sorrendjét határozzák meg. A következő monadikus műveletnek (pl. IO művelet) mindig meg kell várnia a megelőző monadikus művelet eredményét. A monadikus műveletek objektumként is viselkedhetnek, a művelet argumentuma és eredménye egy belső állapotot is tartalmazhat, amelyet a művelet megváltoztat. A monadikus műveletek tehát lépésről lépésre egymásnak adhatják tovább a mindig egyszeresen hivatkozott, belső állapotteret. Az IO műveletek például a külső környezetet viszik magukkal.

A `Monad` osztálynak két fő művelete van. A `return` művelet monadikus értékké alakítva, becsomagolva adja vissza az argumentumként kapott típusértéket, a `>>=` alapművelet pedig két – azonos típuskonstrukcióhoz tartozó – monádérték, azaz két monadikus művelet összekapcsolását biztosítja. A `>>` változat úgy

[44] A monád szó szinonimájaként a szakirodalomban használják a triple-t is.

[45] Különbséget teszünk a monád három alapművelete és a monád értékei, a *monadikus* műveletek között!

kapcsol össze monadikus műveleteket, hogy az első eredményét a második nem használja fel.

A `fail` alapművelet hibakezelésre alkalmas. A `do` kulcsszó segítségével monadikus műveleteket egyszerűbb formában fűzhetünk egymás után, a `do` hatáskörét a margó szabály határozza meg.

```
do  e1 ; e2       =  e1 >>     e2
do  p <- e1; e2   =  e1 >>= (\v -> case v of p -> e2;
                                             _ -> fail "s")
```

A Haskellben kizárólag a monádok okozhatnak mellékhatásokat, így a mellékhatást okozó programrészek jól azonosíthatók és a mellékhatás a monádokon belülre korlátozódik. Az `IO` monádot(1.6.2.,1.7.) használjuk a 1.32.,1.37. példákban. A monádok kifejezőereje nagy, számos nyelvi konstrukció felépítésére alkalmasak [HudFasPe99]. A monádok segítségével például kis, imperatív célnyelvek szemantikáját adhatjuk meg és az ezen nyelveken írt programok kiértékelését végezhetjük el. Ilyen imperatív célnyelvnek tekinthetjük például a Haskellben az előre definiált IO monádot is (ld. 1.7.), amelynek segítségével a programszöveg jól elhatárolható részleteiben imperatív stílusban programozva végezhetünk IO műveleteket.

1.6.3. Frissíthető változók

Az SML-ben a `ref` kulcsszó jelöli a frissíthető változókra (cellákra) hivatkozó mutatókat. A `val r = ref 0`, `val r = !r + 1` SML függvények kiértékelése során az `r : int ref` mutató először a `0 : int` értékre mutat. Az `!r` explicit hivatkozás-feloldással kiolvassuk ezt az értéket és felül is írhatjuk. Az `r` által mutatott memóriacella új értéke `1` lesz.

Frissíthető elemeket tartalmazó tömböket az `Array` modul segítségével használhatunk. Az `array` függvény adott méretű tömböt hoz létre és meghatározza a kezdőértékét, a `sub` függvény alkalmazásával az `i`-dik elemet érhetjük el, az `update` függvénnyel pedig frissíthetjük az értékét:

```
val m : int array = Array.array (size, initv)
   Array.sub (memopad, i)
Array.update (memopad, i, value);
```

1.7. Interaktív funkcionális programok

A Haskell nyelv a program külső állapotterét az `IO` monádok belsejében rejti el. Erre az állapottérre a programozó közvetlenül nem is hivatkozik az IO műveletek

során.

Az `IO` monád a `Monad` osztály `IO a` típuskonstrukcióval való példányosításával jön létre. A `getChar :: IO Char` és a `putChar :: Char -> IO ()` műveletek is az `IO` monádhoz tartoznak, utóbbi alaptípusa az egységelemet tartalmazó rendezett nullás típus. A két IO művelet értékének alaptípusa különböző ugyan, ennek ellenére összekapcsolhatók, mert mindkettő az `IO` monádhoz tartozik. A `getLine`, `putLine` műveletek a `getChar`, `putChar` segítségével építhetők fel. A 1.32. példa azt mutatja be, hogy a `do` kulcsszó segítségével imperatív stílusban írhatunk IO programokat a Haskellben.

1.32. Példa (Haskell). (Egyszerű karakteres io).

```
helloworld :: IO ()
helloworld = putStr "Hello world"

read2lines :: IO ()
read2lines
    = do linea <-  getLine
         lineb <-  getLine
         putStrLn  (reverse lineb)

getInt :: IO Int
getInt = do line <- getLine
            return (read line :: Int)


toScreen :: IO ()
toScreen = print 5
```

A Clean nyelv a program külső állapotterét a `*World` (külvilág) absztrakt típusban rejti el, amelynek típusértékei csak egyszeresen hivatkozhatóak. A Haskell `IO` monáddal szemben a Clean-ben a programozónak közvetlenül kell hivatkoznia a külső állapottérre, vigyázva arra, hogy ezek a hivatkozások mindig egyszeresek legyenek (különben a fordító hibát jelez). Az alábbi program típusleírásából látszik, hogy a `Start` függvény egy új "világot" készít a paraméterként kapott "külvilágból". Az új külső környezet felhasználói események eredményeként jön létre lépésről lépésre. A `stdio` függvény segítségével egy olyan új `w` világot kapunk[46], amelyben a `console` konzol már megnyitott állapotban van. Azt ezt

[46] A `w` változó *nem* kapott új értéket, a baloldalon álló `w` egy új azonosító, amelynek lokális hatáskörén belül a külső `w` nem látszik.

követő lépések során az újabb és újabb konzolra karakteres üzenetek kerülnek, illetve a felhasználó karakteres formában megadott válasza alapján a következő konzol[47] és a `name` lokális konstans értékét egyidejűleg definiáljuk.

1.13. Példa (Clean). (Párbeszéd).

```
module helloconsole
import StdEnv
Start :: *World -> *World
Start w
    # (console,w) = stdio w
      console     = fwrites "Írja be a nevét!:\n" console
      (name,console) = freadline console
      console = fwrites ("Örömmel látom " +++ name +++ "!")
                console
      (_,console) = freadline console
      (ok, nw) = fclose console w
    | not ok = abort "error"
    | otherwise = nw
```

Az SML a `PRIM_IO`, `STREAM_IO`, `IMPERATIVE_IO` modulokkal biztosít adatáramlás alapú IO lehetőséget. A `STREAM_IO` felület puffereli a bemenetet és a kimenetet is. Az `IMPERATIVE_IO` modul a már egyszer megnyitott adatfolyamok dinamikus átirányítását is megengedi. A `TEXT_IO` modul a `STREAM_IO` karakteres IO kezelésére készült példánya.

1.12. Példa (SML). (Adatáramlás alapú IO).

```
let
  ins = TextIO.stdIn
  outs = "output.txt"
in
  TextIO.openIn(ins);
  TextIO.openOut(outs);
  let
    line = TextIO.inputLine()
  in
    TextIO.output(outs,line);
    TextIO.output1 (outs,#"\n");
```

[47] A `console` sem kap új értéket, mindig új konzol jön létre.

```
    TextIO.flushOut(outs);
  end
end
```

Az SML-nek és a Haskellnek nincs szabványos grafikus I/O könyvtára, a Clean Object IO könyvtára [AchWie00] segítségével viszont könnyen kezelhetünk menüket, párbeszédablakokat, ablakokat. A 1.14. példán megfigyelhetjük, hogy a felhasználói felületet algebrai típusokhoz (pl. `Dialog`) tartozó értékek definiálásával írhatjuk le. Az egyes felhasználói akciókhoz, pl. az ablak bezárásához (`WindowClose`) függvényeket (pl. `CloseProcess`) rendelhetünk hozzá, amelyeket az `Object IO` eseménykezelője értékel ki. A `startIO` függvény argumentumában adott `openDialog` függvény kiértékelésének megkezdésével, a párbeszédablak felépítésével indul a felhasználói felülethez rendelt interaktív folyamat.

1.14. Példa (Clean). (Object IO).

```
module hellooio
import StdEnv, StdIO
Start :: *World -> *World
Start world
 = startIO NDI Void (snd o openDialog undef hello) [] world
 where
   hello = Dialog "" (TextControl "Hello world!" [])
                [WindowClose (noLS closeProcess)]
```

1.8. Kivételkezelés

A Miranda és a Clean kivételkezelése lényegében csak arra ad lehetőséget, hogy az `error`, illetve `abort` függvény segítségével a program írója saját maga által megfogalmazott hibaüzenetet küldjön a felhasználónak, mielőtt a program kiértékelése megszakad.

A Haskell `Monad` és `MonadPlus` osztályai (1.6.2.) tartalmaznak egy `fail` alapműveletet, amelynek segítségével lehetőségünk van kivételkezelésre monádokban. Az `IO` monád (1.7.) tartalmaz is kivételkezeléshez szükséges műveleteket. A kivételek ebben az esetben az `IOError` előre definiált absztrakt típushoz tartoznak. A `catch` függvény segítségével az egyes IO hibákhoz kivételkezelőt rendelhetünk, az `ioError` függvény pedig a nem kezelt kivétel továbbadására szolgál [HudFasPe99].

Az SML kivételkezelése általánosabb, az Adáéval rokon. A programozó az `exn` típushoz tartozó olyan kivételeket definiálhat[48], amelyek lehetnek konstans függvények vagy konstruktorfüggvények[49]. A deklarált hibákat ki lehet váltani, és a hívó a kiváltott üzenetekre kivételkezelő függvények kiértékelésével válaszolhat. A 1.13. példában [Har01] a faktoriális (parciális) függvényhez deklarálunk kivételt, amelyet akkor váltunk ki, ha argumentuma negatív érték. A `factorial_driver` függvény több más lehetséges kivétellel együtt kezeli az általunk deklarált kivételt is.

1.13. Példa (SML). (Kivételkezelés).

```
exception Factorial
local
    fun fact 0 = 1
      | fact n = n * fact (n-1)
in
    fun checked_factorial n =
        if n >= 0 then
            fact n
        else
            raise Factorial
end
fun factorial_driver () =
    let
        val input = read_integer ()
        val result = makestring (checked_factorial input)
        val _ = print result
    in
        factorial_driver ()
    end
    handle EndOfFile => print "Done.\n"
         | SyntaxError =>
           let val _ = print "Syntax error.\n" in factorial_driver (
         | Factorial =>
         let val _ = print "Out of range.\n" in factorial_driver ()
```

[48] Az `exn` típus kiterjeszthető típus, ami azt jelenti, hogy adakonstruktorai a típuskonstruktor megadásától elkülönítve, később is deklarálhatóak.

[49] A kivétel kiváltásakor az `exn` értéket előállító adatkonstruktor adott típusú értéket csomagol be.

1.9. Párhuzamos kiértékelés és elosztott programok

Az alábbiakban olyan nyelvi elemeket mutatunk be Clean, Haskell, illetve JoCaml példákon keresztül, amelyek segítségével osztott, párhuzamos programokat készíthetünk funkcionális stílusban. A nyelvek kifejező ereje különböző, az egyes elemek absztrakciós szintje eltérő. Bemutatjuk a legalacsonyabb absztrakciós szintet képviselő annotációkat, az azokra épülő kiértékelési módszereket, az explicit üzenetküldés eszközeit, a csatornákat, a mobil funkcionális kód megfelelőit, a Dynamic-ot és az ágenseket. Ezek hatékonyságban, alkalmazhatóságban, és megvalósításukban is nagyon különböznek egymástól. A megoldandó feladatnak megfelelően választhatunk a bő kínálatból és hozhatunk létre párhuzamos és elosztott alkalmazásokat.

A függvények kompozíciója asszociatív, így a funkcionális nyelvű programok kiértékelése jól párhuzamosítható. A legfontosabb kérdés annak eldöntése, hogy mely részkifejezések kiértékelését célszerű párhuzamos vagy elosztott módon elvégezni.

A párhuzamos és elosztott funkcionális programozás világában többféle irányvonal létezik [ZsóHoTe02]: új absztrakt nyelvi elemek bevezetése, párhuzamos és elosztott függvénykiértékelés létrehozása, a kiértékelési módszerek módosítása, dinamikusan összeszerkesztett mobil kód, valamint TCP/IP kommunikációs protokoll használata. Ezeket a nyelvi lehetőségeket mutatjuk be Concurrent Clean [Kes96, AchPla95, HoZs00, Ser99, AchWie00] és Distributed Haskell with ports [HuNo01], valamint JoCaml[50] [Fou01] példák segítségével.

1.9.1. Párhuzamos és elosztott programozás Concurrent Cleanben

Elsőként a Concurrent Clean nyelvben található, a párhuzamosságot kifejező egyik legrégebbi funkcionális programnyelvi elemeket szeretnénk bemutatni, az annotációkat [Kes96]. Az annotációk segítségével meghatározhatjuk, hogy a fordítóprogram egy függvény vagy egy kifejezés mely részeit értékelje ki párhuzamosan. A lusta nyelvekben általában csak olyan kifejezések kiértékelésére kerül sor, amelyek értékére már szükség van. A párhuzamos kiértékelés kivételt képez ez alól a szabály alól. Az annotációk segítségével, *spekulatív módon* meghatározhatjuk, hogy szerintünk mely részkifejezések kiértékelését érdemes párhuzamosan előre elkezdeni, hogy amikor később szükség lesz a kifejezés értékére, akkor annak normál formája már rendelkezésre álljon. Három típusú annotációt különböztetünk meg:

[50] ML változat, amely imperatív és objektum orientált elemeket is tartalmaz.

- `I`: összefésüléses annotáció, mely lehetővé teszi egy kifejezés páruzamos kiértékelését ugyanazon a processzoron, ahol az `I` annotációt tartalmazó kifejezés kiértékelése folyik. A két kiértékelő folyamat egymással párhuzamosan, felváltva dolgozik[51].

- `P`: párhuzamos annotáció, mely egy kifejezés ugyanazon, vagy lehetőség szerint egy másik processzoron történő párhuzamos kiértékelését jelöli.

- `P at procid`: párhuzamos kiértékelést végez megadott processzoron.

A következő függvény annotációk segítségével számítja ki a Fibonacci szám értékét:

1.15. Példa (Clean). (Annotációk).

```
Nfib :: Int -> Int
Nfib n   | n < 2  =     1
                          =   {| P |}Nfib (n - 1) +  Nfib (n - 2)
```

A fenti példában a `P` annotáció hatására egy új kiértékelő folyamat indul el, azaz `Nfib (n - 1)` kiértékelése az `Nfib (n - 2)` kiértékelésével párhuzamosan történik.

Az annotációk segítségével kiértékelési módszereket[52] fogalmazhatunk meg magasabbrendű függvények segítségével [Tri98, HoZs00]. Két elemi kiértékelési módszer, a `'seq'` (a szekvenciális kiértékelési módszer) és a `'par'` (a párhuzamos kiértékelési módszer) segítségével összetett kiértékelési módszereket építhetünk fel. Például az összetett `parlist` módszer segítségével párhuzamos `map` függvényt írhatunk, amely az eredménylista elemeit egymásssal párhuzamosan határozza meg. Az egyes elemeket pedig a `parlist` paramétereként átvett `s` kiértékelési módszer szerint számoljuk ki.

1.33. Példa (Clean és Haskell). (Összetett kiértékelési módszer).

```
parmap :: (Strategy b) (a -> b) [a] -> [b]
parmap s f x = map f x 'using' parlist s
```

Egy másik párhuzamos programozási lehetőség a Concurrent Clean-ben a csatornák használata. Az üzenetküldés kétféle típusát különböztetjük meg: két különálló

[51] Az összefésüléses szemantika határozza meg a párhuzamos kiértékelés eredményét.
[52] stratégia

program közötti kommunikációt, illetve egy programon belüli párhuzamos szálak üzenetváltását [Ser99].

Az első esetben két absztrakt csatorna adattípust használunk: az `(SChannel Int)` és az `(RChannel Int)` típust, amelyek `Int` alaptípusú értékek egyirányú továbbítására, küldésére, ill. fogadására alkalmas csatornák típusát jelölik. Az üzeneteket a fogadó csatorna listában tárolja, érkezési sorrendben. Csatornát a `createRChannel`-lel hozunk létre[53], a `findSChannel`-lel, illetve `lookupSChannel`-lel pedig megkeressük a már létrehozott csatornát, neve alapján, a másik oldalon. A programok egymással a `send`, a `receive` és az `available` függvények kiértékelésével kommunikálnak. Az alábbi program a termelő–fogyasztó feladat egy megoldása csatornák segítségével. A `produce` függvény rekurzív módon 10 adatot állít elő, a `consume` függvény pedig ugyancsak 10 adatot fogad.

1.16. Példa (Clean). (Csatornák programok között).

```
// 1. a termelő program
Start :: *World -> *World
Start w
   # (sc, w) = findSChannel "Consumer" w
   = produce sc 10 1 w

produce :: (SChannel Int) Int Int *World -> *World
produce sc i n w
   | i == 10   = w
   | otherwise = produce sc (i - 1) (n + 1) (send sc n w)

// 2. a fogyasztó program
Start :: *World -> (Int, *World)
Start w
   # (maybe_rc, w) = createRChannel "Consumer" w
   = consumer maybe_rc w
where
   consumer Nothing w = abort "already exists"
   consumer (Just rc) w = consume rc 10 0 w

consume :: (RChannel Int) Int Int *World -> (Int, *World)
consume rc i r w
   # (n, rc, w) = receive rc w
```

[53] Amennyiben csak helyi használatra szeretnénk egy csatornát létrehozni, akkor azt a `newChannel`-lel hozzuk létre.

```
   | i == 0    = (r + n, w)
   | otherwise = consume rc (i - 1) (r + n) w
```

A második esetben lusta kiértékelésű szálakat hozhatunk létre a `newThread` függvénnyel ugyanazon a processzoron. A szálak közötti üzenetváltást ugyancsak a `send`, a `receive` és az `available` függvényekkel valósíthatjuk meg.

Példaprogramként ismét a termelő–fogyasztó feladat egy megoldását mutatjuk be, ez alkalommal a termelő és a fogyasztó két különálló szálnak felel meg. A két szál törzse ugyanazon programhoz tartozik, de egy szálat elindíthatunk egy másik processzoron is. Ezt a `newThreadAt pid` segítségével tehetjük meg:

1.17. Példa (Clean). (Csatornák szálak között).

```
module ProdCons
import StdEnv, StdParallel, StdThread, StdChannel

Start w = startProcessorsW' myStart w

// létrehoz egy csatornát a pid azonosítóval rendelkező processzoron
// a csatorna a két külön szálon futó produce, illetve a consume
// függvényeket köti össze

myStart :: (Set ProcId) *World -> (Int, *World)
myStart pids w
   # (sc, rc, w) = newChannel w
     w           = newThreadAt pid (produce sc 10 1) w
   = newThread' (consume rc 10 0) w
where
   pid = pickFromSet pids 0

// a termelő, produce függvény

produce :: (SChannel Int) Int Int *env -> *env | ThreadEnv env
produce sc i n env
   | i == 0    = env
   | otherwise = produce sc (i - 1) (n + 2) (send' n sc env)

// a fogyasztó consume függvény
consume :: (RChannel Int) Int Int *env -> (Int, *env) | ThreadEnv en
consume rc i r env
```

```
   # (n, rc, env) = receive' rc env
   | i == 1    = (r + n, env)
   | otherwise = consume rc (i - 1) (r + n) env
```

Csatornákat használ a TCP/IP protokoll alapú párhuzamos funkcionális programozás is [AchWie00, Tan99]. A hálózat bármely gépét IP címe szerint azonosíthatjuk, majd a géphez rendelt csatornákon üzeneteket küldhetünk a vele összekötött gépeknek. Miután létrehoztunk egy kliens-szerver kapcsolatot két gép között, ezek a csatornákon keresztül a következőképpen küldhetnek egymásnak üzeneteket:

1.18. Példa (Clean). (TCP alapú kommunikáció).

```
// a kliens elküldi a "hello server" üzenetet a
// TCP_SChannel-len keresztül
clientSend :: TCP_SChannel *World -> (TCP_SChannel, *World)
clientSend sChannel world=send (toByteSeq "hello server")
                           sChannel world

// a szerver TCP_SChannel-len fogadja az üzenetet
serverReceive ::
   String TCP_RChannel *World -> (TCP_RChannel, *World)
serverReceive expectedMessage rChannel world
   # (message, rChannel, world)       = receive rChannel world
   | toString message <> expectedMessage = abort "wrong message"
   | otherwise                          = (rChannel, world)
```

A Clean nyelv szigorúan statikus típusrendszerét egészíti ki a `Dynamic` típus [Pil, PlaEek01]. A `dynamic` függvény segítségével tetszőleges típusú[54] kifejezést átalakíthatunk `Dynamic` típusúvá, illetve később a `Dynamic` értéket a tartalmazott érték *típusára való mintaillesztéssel* visszakaphatjuk. A `sendDynamic` és `receiveDynamic` függvényekkel lehetőségünk van arra is, hogy Clean programok konstansokat és mobil kódrészleteket küldjenek és fogadjanak [PlaEek01, HorKozs02]. A `writeDynamic` és `readDynamic` függvények alkalmasak arra, hogy fájlba írjunk ki, fájlból olvassunk be `Dynamic` típusú kifejezéseket. Az alábbiakban egy példán azt mutatjuk be, hogy egy Clean program hogyan épít fel egy fát, amelyet fájlba ment; egy másik program elkészít egy olyan függvényt,

[54] Minden olyan típusérték `Dynamic` típusúvá alakítható, amely a `TC` típusosztályba tartozik, azaz létezik típuskódja.

amely megszámolja egy fa leveleit és a függvényt kiírja egy fájlba, végül egy harmadik program beolvassa a fát a fájlból és alkalmazza rá az ugyancsak fájlból beolvasott függvényt, majd kiírja az eredményt.

Az első program létrehozza a `tree2` fát, becsomagolja a `dynamic` függvénnyel egy `Dynamic` típusú értékbe, végül kiírja egy fájlba a `writeDynamic`-kal.

1.19. Példa (Clean). (Dynamic típusú érték készítése).

```
module v
import StdDynamic, StdEnv
:: Tree a = Node a (Tree a) (Tree a) | Leaf
Start world
   #! (ok, world) = writeDynamic (p +++ "value")
                        DynamicDefaultOptions dt world
   | not ok = abort "could not write dynamic"
   = (dt, world)
   where    dt = dynamic (Node 99 tree2 tree2)
         tree2 = (Node 2 (Node 1 Leaf Leaf) Leaf)
             p = "C:\\Clean 2.0\\Examples\\Dynamic\\"


module f
import StdDynamic, StdEnv
:: Tree b = Node b (Tree b) (Tree b) | Leaf
Start world
   #! (ok,world) = writeDynamic (p +++ "function")
                         DynamicDefaultOptions dt world
   | not ok = abort "could not write dynamic"
   = (dt,world)
   where    dt = dynamic count_leafs
             p = "C:\\Clean 2.0\\Examples\\Dynamic\\"
```

Az `apply` alkalmazza a fájlból beolvasott függvényt az ugyancsak fájlból beolvasott fára, majd kiírja az eredményt. A beolvasás során *típusokra vonatkozó mintaillesztéssel* ellenőrzzük, hogy a beolvasott függvény és a beolvasott érték típusa megegyezik-e az elvárt típusokkal.

1.20. Példa (Clean). (Dynamic felhasználása).

```
module apply
```

```
import StdDynamic, StdEnv
:: Tree a = Node a (Tree a) (Tree a) | Leaf
Start world
   # (ok,f,world)  = readDynamic (p +++ "function") world
   | not ok            = abort " could not read"
   # (ok,v,world) = readDynamic (p +++ "value") world
   | not ok            = abort " could not read"
   # result = apply f v;
   = (result, world);
where
     apply (f :: (Tree Int) -> Real) (v :: (Tree Int))= f v
     apply _ _                                         = abort "unmatched"
     p = "C:\\Clean 2.0\\Examples\\Dynamic\\"
```

1.9.2. Distributed Haskell with ports

A Distributed Haskell könyvtár[HuNo01] a Haskell funkcionális programozási nyelv egy kiterjesztése, amelynek használatával osztott Haskell programokat írhatunk. A kiterjesztés lehetőséget ad arra, hogy a programok között egyirányú, pont–pont kapcsolatot létesítsünk kapuk[55] segítségével. Nézzük meg, milyen alapvető nyelvi elemek állnak ehhez rendelkezésünkre. A könnyebb megértés céljából vegyünk egy egyszerű példaalkalmazást, melyben egy szerver és két (azonos kódú) kliens programunk van. A kliensek egy paraméterként megadott számot küldenek el a szervernek, a szerver visszaküldi a klienseknek az elküldött két szám összegét, azok pedig kiírják az összeget a képernyőre.

Ahhoz, hogy a kliensek üzenetet tudjanak küldeni a szervernek, először a szerveroldalon létre kell hoznunk egy kaput. Ezt a `newport` függvénnyel tehetjük meg, amellyel bármilyen olyan típushoz, amely a Haskell `Readable` és `Showable` osztályába is beletartozik létrehozhatunk kaput, amelyen az adott típusú adat küldhető. A kaput csak az a program (és ha esetleg a programon belül több szál van, akkor csak az a szál) olvashatja, amely létrehozta azt. A kliensekkel valamilyen módon tudatnunk kell, hogy melyik az a kapu, amelyre a szervernek üzenetet küldhetnek. Ehhez a szerveroldalon a `registerPort` függvénnyel be kell jegyeznünk a szerverkaput valamilyen (a kliensek által ismert) néven.

1.34. Példa (Haskell). (Port létrehozása és bejegyzése).

```
serverPort <- newPort
registerPort serverPort "ServerPort"
```

[55] portok

Bejegyzés után a kliensoldalon a `lookupPort` függvénnyel lekérdezhetjük az adott névhez tartozó kaput. A néven kívül szükségünk van a szervert futtató gép IP címére is (a példánkban a szerveroldali IP címet a kliensek parancssori argumentumként kapják meg). Ezek után a klienseknek már elég információ áll a rendelkezésükre ahhoz, hogy elküldjék az argumentumként kapott számot a szervernek. Mivel az üzenetküldés csak egyirányú, a klienseknél is létre kell hoznunk egy-egy kaput, amelyre a szerver elküldheti az eredményül kapott értéket. Ugyanakkor tudatnunk kell a szerverrel, hogy melyek ezek a kapuk. Erre a legegyszerűbb módszer, ha a kliensek a paraméterként kapott számokkal együtt az általuk létrehozott kapuk leírását is elküldik a szervernek. A kapukra való adatküldésre a `writePort` függvény szolgál. A függvénynek a küldeni kívánt adatot, illetve azt a kaput kell megadnunk paraméterül, amelyre az adatot küldeni szeretnénk.

1.35. Példa (Haskell). (Adatok elküldése a szerver számára).

```
[host,numstr] <- getArgs
serverPort <- lookupPort host "ServerPort"
inPort <- newPort
writePort serverPort (num,inPort)
```

A szerver az előzőekben leírt kapubejegyzés után a `readPort` függvény meghívásával olvasni próbál egy számból és egy kapuleírásból álló adatpárt a szerver kapuról. A `readPort` függvény akkor ad vissza adatot, ha valaki küld valamit a `writePort` függvénnyel az adott portra, amíg ez meg nem történik, a program kiértékelése felfüggesztődik a függvényhívásnál. Miután a szerver sikeresen olvassa az egyik kliens által küldött értékeket, még egyszer meghívja ugyanazt a függvényt, hogy a másik kliens adatait is megkapja.

1.36. Példa (Haskell). (A kliensoldali adatok fogadása).

```
(num1,outPort1) <- readPort serverPort
(num2,outPort2) <- readPort serverPort
```

Ha ez is sikeresen megtörténik, akkor a már ismert `writePort` függvénnyel elküldi a kapott számok összegét a számokkal egyidejűleg kapott kapukra.

```
-- az összeg visszaküldése
writePort outPort1 (num1 + num2)
writePort outPort2 (num1 + num2)
```

A kliensek a fentiekben leírt adatküldés után megpróbálnak egy `Int` típusú értéket olvasni a `readPort` függvénnyel az általuk létrehozott kapuról, majd miután ez sikerül, kiírják a kapott értéket a képernyőre.

```
-- az eredmény fogadása és kiírása
sum <- readPort inPort
putStrLn (show sum)
```

Ezek után nézzük meg a teljes programokat:

1.37. Példa (Haskell). (Szerver).

```
import IO
import Port
import System

main = do
     serverPort <- newPort
     registerPort serverPort "ServerPort"
     (num1,outPort1) <- readPort serverPort
     (num2,outPort2) <- readPort serverPort
     writePort outPort1 (num1 + num2)
     writePort outPort2 (num1 + num2)
```

1.38. Példa (Haskell). (Kliens).

```
import IO
import Port
import System

main = do
     [host,numstr] <- getArgs
     serverPort <- lookupPort host "ServerPort"
     inPort <- newPort
     client inPort serverPort (read numstr)

client :: Port Int -> Port (Int,(Port Int)) -> Int -> IO ()
client inPort serverPort num = do
     writePort serverPort (num,inPort)
     sum <- readPort inPort
     putStrLn (show sum)
```

1.9.3. A JoCaml párhuzamos és osztott nyelvi elemei

A JoCaml [Fou01] nem tisztán funkcionális nyelv, imperatív és objektum orientált elemeket is tartalmaz. A JoCaml nyelvben a mobil kód megfelelője az ágens. A nyelv tervezésének három fő szempontja volt: eszközök biztosítása ágensek egyszerű létrehozására, összetett elosztott számítások kifejezhetősége, az absztrakt nyelvi eszközök jelentésének pontos leírhatósága. A legfontosabb két absztrakt nyelvi elem a csatorna és az absztrakt hely fogalma. Szinkron és aszinkron csatornát a `let def` utasítás segítségével hozhatunk létre, aszinkron csatorna azonosítóját egy `!` követi. Aszinkron csatornán a csatornához rendelt folyamatok segítségével kezelhetünk adatokat.

```
let def echo! X = print_int x;
val echo : <<int>>
```

A `spawn` az aszinkron csatornához rendelt folyamatpéldányból kifejezést állít elő, melynek kiértékelésekor egy konkurrens folyamat indul el. Az üzenetküldést az alábbi példán szemléltetjük, mely párhuzamos folyamatokhoz csatolt csatornák értékét írja ki, azaz az 1, 2, 3 egész számokat előre nem ismert sorrendben:

```
spawn{
   let x = 1 in
   {let y = x + 1 in echo y | echo (y + 1)} | echo x
```

A konkurens programozás összetett vezérlési és adatkezelési struktúráit szinkronizációs mintákkal írhatjuk le. Csatornahalmazokhoz tartozó kommunikációs események között fennálló konfliktust, illetve szinkronizációt fejezhetünk ki velük. Szinkronizációs minták mindig egyetlen `let def` kifejezésben egyidejűleg definiált csatornákra vonatkoznak. Mintákat a `|` párhuzamos kompozíció művelettel írhatunk le, csatornarészhalmazokra vonatkozó, több alternatív mintát pedig az `or` művelettel kapcsolhatunk össze . Konkurens környezetben használt számlálóra példa a `count` aszinkron csatorna, amelyre küldött üzeneteket vagy az `inc`, vagy a `get` szinkron csatornákra érkező üzenetekkel szinkronizálunk. Egy kezdeti `count 0` üzenet után az így kapott konkurrens rendszer az `inc` üzenet hatására növeli a számlálót, önmagának küldve újabb `count` üzenetet és abban tárolva a számláló aktuális értékét. A `get` üzenet hatására visszaadja a számláló értékét, egyben önmagának egy `count` üzenet küld, ezzel biztosítva a további helyes működést.

```
let def count! n | inc () = count (n + 1) | reply to inc
     or count! n | get () = count n | reply n to get
spawn {count 0}
```

A JoCaml-ban elosztott programokat is írhatunk. A folyamatok vándorolhatnak egyik gépről a másikra. Kezdetben a különböző gépeken futó folyamatok között nincs kapcsolat, a partnerek névszolgáltató segítségével találnak egymásra.

Az absztrakt hely fogalma magában foglalja az ágens kódjának és aktuális fizikai helyének együttesét. Ily módon a kód mozgása során a JoCaml nyelv absztrakt eszközöket biztosít a kompozíció, a kommunikáció és a belső állapot helyfüggetlen kezelésére és távoli szolgáltatások igénybevételére. Az ágensek összetevői együtt mozognak az őket tartalmazó szülő ágenssel, a kommunikációs csatorna pedig fennmarad a helyüket időközben változtató ágensek között is, és az ágensek abból az állapotból folytatják működésüket, ahol a helyváltoztatás előtt voltak.

A következő példában az egyik folyamat bejegyzi az `f` közös erőforrást `square` néven, a másik pedig megkeresi és felhasználja:

```
spawn {let def f x=reply x*x in Ns.register "square" f vartype;}
spawn {let sqr=Ns.lookup "square" vartype in print_int(sqr 2); exit 0;}
```

Absztrakt helyeket a `let loc <azonosító> do {}` konstrukcióval hozhatunk létre. Egy fizikai helyet mint egy absztrakt hely aktuális megfelelőjét, azaz mint az ágens aktuális fizikai helyét azonosíthatjuk. Erre használjuk majd a `here` üres ágenst:

```
let loc here do {
Ns.register "here" here vartype;
Join.server ()  }
```

A `mobile` ágens lekérdezi a névszolgáltatótól a `here` ágens adatait, majd átvándorol a `here` aktuális fizikai helyére és ennek mindenkori helyén, alágenseként végzi el a számításokat.

```
let loc mobile
 do {
    let here = Ns.loopkup "here" vartype in
    go here;
    let sqr = Ns.lookup "square" vartype in
    let def sum (s,n)=reply (if n=0 then s else sum(s+sqr n, n-1)) in
    let result = sum (0, 5) in
    print_string ("q: sum 5= " ^ string_of_int result ^ "\n");
     flush stdout;
  }
```

Letölthető ágenseket is definiálhatunk oly módon, hogy az ágensnek paraméterként adjuk meg, hogy híváskor hova vándoroljon, illetve adatvezérelt vándorlást is megvalósíthatunk.

1.10. Feladatok

1. Adjuk meg egy szám prímosztóit!
2. Döntsük el egy számról, hogy tökéletes szám-e!
3. Rendezzük nagyság szerint növekvő sorrendbe egy sorozat elemeit!
4. Keressük meg egy sorozat legkisebb elemét! A rendezési reláció legyen a függvény argumentuma!
5. Határozzuk meg egy sorozat elemeinek átlagát!
6. Döntsük el, hogy egy karaktersorozat előfordul-e egy másikban!
7. Készítsünk absztrakt, algebrai zsák típust!
8. Írjunk interaktív funkcionális programot! A képernyő tetején jelenjen meg egy menüsor, annak egyik legördülő menüjéből nyissunk meg egy párbeszédablakot. A párbeszédablakban legyen két szerkeszthető beviteli mező, egy válaszsor és egy gomb. A beviteli mezőben olvassunk be két egész számot és a gomb megnyomására összegüket jelenítsük meg a válaszsorban.

Irodalomjegyzék

[BaWe90] Barr,M.-Wells,Ch.: *Category Theory for Computer Science.* Prentice Hall, London, 1990.

[Csö01] Csörnyei Zoltán:Lambda-kalkulus, előadás jegyzet, http://people.inf.elte.hu/csz/lk-jegyzet.html.

[AchPla95] Achten, P.M. - Plasmeijer, M.J.: Concurrent Interactive Processes in a Pure Functional Language. In van Vliet, J.C. ed. Proceedings Computing Science in the Netherlands, CSN'95, Jaarbeurs Utrecht, The Netherlands, November 27-28, Stichting Mathematisch Centrum, Amsterdam, 1995, pp.10-21.

[AchWie00] Achten, P., Wierich, M.:*A Tutorial to the Clean Object I/O Library*, University of Nijmegen, 2000. http://www.cs.kun.nl/~ clean

[PlEe93] Plasmeijer,R.-van Eekelen,M.: Functional Programming and Parallel Graph Rewriting, Addison-Wesley, 1993.

[EeNoPlSm90] van Eekelen,M. et al.: Concurrent Clean, Technical Report no 90-20, November 1990, University of Nijmegen.

[PlaEek01] Plasmeijer, R., van Eekelen, M.: Concurrent Clean Language Report, University of Nijmegen, 2001.

[Bar 84] Barendregt,H.P.: The Lambda-Calculus, its Syntacs and Semantics. Amsterdam: North-Holland, 1984.

[FiJo] Finne,S.-Jones,S.P.: S. Finne and Simon L. Peyton Jones. Programming Reactive Systems in Haskell. In Glasgow Functional Programming Workshop, Ayr, Scotland, 1994. Springer-Verlag.

[Fót83] Fóthi Á.: *Bevezetés a programozáshoz.* Egyetemi jegyzet (ELTE, TTK, Budapest, 1983).

[Fou01] Fournet, C., Le Fessant, F., Maranget, L., Schmitt, A.: *The JoCaml language beta release, Documentation and user's manual*, INRIA, 2001.

[HuNo01] Huch, F., Norbisrath, U.: Distributed Programming in Haskell with Ports. Implementation of Functional Programming Languages, 12th International Workshop, IFL2000, Sep. 4-7, Achen, LNCS 2011, pp. 107-121, Springer 2001, http://www-i2.informatik.rwth-aachen.de/hutch/distributedHaskell.

[Har01] Harper, R.: Programming in Standard ML. Working Draft. Carnegie Mellon University, Spring Semester, 2001. http://www.cs.cmu.edu/~rwh/smlbook/

[Mil97] Milner,R., Tofte, M., Harper, R., MacQueen, D.: *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[Hor94] Horváth Z.: A párhuzamos programozás alapjai, jegyzetek (100 oldal), Bp., 1994/95.

[Tan99] Tandi P., Harmat K., Horváth Z., Wierich M., Plasmeijer R.: Web Computing in Clean. In Proceedings of 4th International Conference on Applied Informatics, Eger, Hungary, 1999., 87-93.

[Hor99] Horváth,Z.-Achten,P.-Kozsik,T.-Plasmeijer,R.: Verification of the Temporal Properties of Dynamic Clean Processes, *Proceedings of the 11th International workshop on the Implementation of Functional Languages, IFL'99, Lochem*, The Netherlands,1999, pp. 203-218.

[MolEek99] Mol, de M.-van Eekelen,M.: A Proof Tool Dedicated to Clean, AGTIVE'99, Kerkade.

Maarten de Mol, Marko van Eekelen, Rinus Plasmeijer. Theorem Proving for Functional Programmers - SPARKLE: A Functional Theorem Prover <ftp://ftp.cs.kun.nl/pub/Clean/papers/2002/molm2002-TheoremProvingFP.ps.gz> . In: Arts, Th., Mohnen M., eds. Proceedings

of the 13th International Workshop on the Implementation of Functional Languages, IFL 2001, Selected Papers, Älvsjö, Sweden, September 24-26, 2001, Springer-Verlag, LNCS 2312, pages 55-71.

[Pil] Pil, M.: Dynamic types and type dependent functions, LNCS 1467, pp. 169-185.

[Han00] Hanák P.: Dekalratív programozás, oktatási segédlet, Bevezetés a funkcionális programozásba (201 oldal), BME, Bp. 2000. (SML)

[Hor97] Horváth Z.: A funkcionális programozási stílus és nyelvi elemei. Jegyzetek (40 oldal), ELTE, Bp. 1997. ftp://valerie.inf.elte.hu/pub/funkc/jegyzet.ps.

[HorFót99] Horváth Z.,Fóthi Á.: A funkcionális programozási stílus nyelvi eszközei. Informatika a Felsőoktatásban, Debrecen, 1999., 570-575.

[HorKozs02] Horváth Z., Kozsik T.: Certified Proven Property Carrying Code (CPPCC) - Safe Functional Mobile Code, ECOOP WS 1 and 18, Malaga, 2002.

[HoTePá02] Horváth Z., Pásztor K., Tejfel M.: Funkcionális programok helyessége, Informatika a Felsőoktatásban, Debrecen, 2002.

[ZsóHoTe02] Zsók V., Horváth Z., Tejfel M.: Párhuzamos funkcionális programozás, Informatika a Felsőoktatásban, Debrecen, 2002.

[HeHoZsó02] Hegedűs H., Horváth Z., Zsók V.: A Haskell és a Clean nyelv nyelv összehasonlító elemzése, Informatika a Felsőoktatásban, Debrecen, 2002.

[HoZs00] Horváth Z., Zsók V., Serrarens P., Plasmeijer R.: Parallel Elementwise Processable Functions in Concurrent Clean, to appear in Proceedings of the 5th International Conference on Applied Informatics, Eger, Hungary, January 2001 and selected for Computers & Mathematics with Applications, Elsevier.

[Hud89] Hudak,P.: Conception, Evolution and Application of Functional Programming Languages, ACM Computing Surveys, Vol. 21. No. 3. Sept. 1989. 359-411.

[HudFasPe99] Hudak,P.-Fasel,J.H.-Peterson J.: A Gentle Introduction to Haskell 98. http://www.haskell.org.

[PJH99] Peyton Jones, J., Hughes J., et al. *Report on the Programming Language Haskell 98, A Non-strict, Purely Functional Language*, February 1999.

[Jon96] Jones,S.P.-Gordon,A.-Finne,S.: Concurrent Haskell. In Proc. of Int. Conf. on Principles in Programming Languages, pp. 295–308. 1996.

[Tho99] Thompson, S. *Haskell, The Craft of Functional Programming*. Addison-Wesley, 1999.

[Pla99] Plasmeijer,R. et al.: *Functional Programming in Clean*, July 1999. Draft., http://www.cs.kun.nl/˜ clean/.

[Kes96] Kesseler, M.H.G.: *The Implementation of Functional Languages on Parallel Machines with Distributed Memory*, PhD Thesis, Catholic University of Nijmegen, 1996.

[Tri98] Trinder, P.W., Hammond K., Loidl, H.W., Peyton Jones, S.J.: Algorithm + Strategy = Parallelism. *Journal of Functional Programming* 8 (1): pp. 23-60, 1998.

[Kiss94] Kiss,A.: A funkcionális programozás specifikációt támogató nyelvi eszközei. Szakdolgozat. Témavezető: Horváth Z., ELTE TTK Informatika Tanszékcsoport. 1993.

[Mir90] Turner, D.A.: Miranda System Manual. Research Software Ltd. Canterbury, England. Research Software Limited, 1990.

[Cla96] Clack,C., Myers,C., Poon,E.: *Programmieren in Miranda.* Prentice Hall, 1995.

[Ser99] Serrarens, P.R.: Explicit Message Passing for Concurrent Clean, In: Hammond, K. et al., eds., *Implementation of Functional Languages, 10th International Workshop, IFL'98*, London, UK, September 1998, LNCS, Vol. 1595, pp. 229-245, Springer-Verlag, 1999.

[Tho90] Thomson, S.: Lawful Functions and Program Verification in Miranda, Science of Computer Programming, Vol. 13, Num. 2-3, May 1990, 181-218.

[Tur86] Turner,D.: An Overview of Miranda, Sigplan Notices, Sigplan Notices, 21(12), 158-166. 1986.

Tárgymutató