Programozási Nyelvek
Gyakorlati jegyzet

Készítette:
Michnay Balázs – V. Informatika
Michnay.Balazs@stud.u-szeged.hu
http://www.stud.u-szeged.hu/Michnay.Balazs
2004

Tartalomjegyzék

Objektumorientált programozás - Smalltalk ... 3
Egyéb segédanyag Smalltalk-hoz ... 3
A Smalltalk használata a kabinetes termekben ... 3
Magáról a nyelvről ... 3
Változók létrehozása ... 4
Blokkok ... 4
A cascade ... 4
Vezérlési szerkezetek megvalósítása Smalltalk nyelven ... 4
Feltételes vezérlés ... 4
Ismétléses vezérlés ... 5
For ciklus megvalósítása ... 5
A gyakorlat példafeladatai, programjai ... 5
Gyakorló feladatok ... 7
Összefoglaló kérdések ... 7
Kollekciók ... 8
A collect ... 8
A select ... 8
Objektumok összehasonlítása ... 9
Másolat készítése objektumról ... 9
Metaosztály fogalma ... 9
Az Object osztály ... 9
A Smalltalk osztályhierarchiája (részlet) ... 10
Osztályok létrehozása ... 10
Metódusok definiálása osztályokhoz ... 11
A gyakorlat példafeladatai, programjai ... 13
Összefoglaló kérdések ... 14
Egy konkrét feladat megoldása: Könyv nyilvántartó ... 15
Funkcionális programozás - Haskell ... 19
Egyéb segédanyag Haskell-hez ... 19
A Haskell használata a kabinetes termekben ... 19
A szemléletmódról és a nyelvről néhány mondatban ... 19
Típusok és operátorok ... 19
Atomi típusok ... 19
Numerikus argumentumú infix relációs operátorok ... 20
Lista típusok ... 20
Megjegyzések Haskell programban ... 20
Függvények definiálása ... 20
Esetvizsgálatok ... 20
Esetvizsgálat az if kifejezéssel ... 21
Esetvizsgálat „őrökkel” (Boolean Guards) ... 21
Esetvizsgálat mintaillesztéssel ... 21
Esetviszgálat a case kifejezéssel ... 21
A gyakorlat függvényeinek definíciója ... 21
A többargumentumú függvényekről ... 24
Függvény paramétere függvény ... 24

Lokális definíciók függvénydefiníciókban ... 25
Listák feldolgozása ... 25
Függvények kompozíciója ... 26
Típusok létrehozása ... 26
A lusta kiértékelésről bővebben ... 28
Logikai Programozás - Prolog ... 30
Egyéb segédanyagok Prolog-hoz ... 30
Telepíthető Prolog rendszerek ... 30
Egy bevezető példa ... 30
A Prolog program felépítése ... 32
Termek ... 32
Kiértékelés ... 32
Aritmetikai Operátorok ... 33
Illesztés ... 34
A gyakorlat feladatai ... 35
A member predikátum értelmezése ... 36
További listával kapcsolatos feladatok ... 37
select – egy elem törlése listából ... 37
append – Listák összefűzése ... 37
rev – Lista megfordítása ... 38
kiir – Listaelemek kiírása ... 38
len – Lista hosszának kiírása ... 39
Nyomkövetés ... 40
Párhuzamos programozás - Occam ... 41
Egyéb segédanyagok Occam-hoz ... 41
Occam fordító ... 41
A KroC használata kabinetes termekben ... 41
Párhuzamos programozás ... 41
Hasznos tudnivalók a nyelvről ... 41
Az Occam elemi folyamatai ... 42
Az Occam adattípusai ... 43
Tömbök ... 43
SEQ ... 43
PAR ... 44
Vezérlési szerkezetek ... 44
Feltételes vezérlés – IF ... 44
Ismétléses vezérlés – WHILE ... 45
Összetett folyamatok replikációja ... 45
PROC-ok ... 46
Az ALT konstrukció ... 47
Az Occam-ban megismert szerkezetek C-beli megfelelői ... 47
Egy összetettebb példa – Multiplexer ... 49

Objektumorientált programozás - Smalltalk

Egyéb segédanyag Smalltalk-hoz

- pub/Programnyelvek/Smalltalk/tutorials (angol nyelven, kivéve: Smalltalk-Leiras.html)

Telepíthető Smalltalk rendszerek:

- Linuxhoz
  - GNU Smalltalk (pub/Programnyelvek/Smalltalk/tcltk)
    Itt van 5 RPM file, ezek kellenek a pub/Programnyelvek/Smalltalk/Smalltalk-2.0.3.tar.gz telepítéséhez.
- Windowshoz
  - Squeak – ingyenes Smalltalk rendszer Windowshoz. Letölthető innen: http://www.squeak.org

A Smalltalk használata a kabinetes termekben

Indítsunk Linux operációs rendszert, majd indítsunk egy terminálablakot (System Tools → Terminál).
Innen indítható a Smalltalk: gst –q (-q = quiet, „silent" mód, kényelmesebb ezzel dolgozni)

Ha „gst -q" -val indítjuk, akkor a programot „be kell gépeljük", majd le kell futtassuk a ! kiadásával. Ez hosszú programok esetén nem túl hatékony, apró hiba esetén is újra kell gépelni az egész programot. Ezért folyamodjunk más megoldáshoz. A programunkat írjuk szövegszerkesztőbe, mentsük el, majd indítsuk a Smalltalkot a forráskódállomány nevét tartalmazó paraméterrel. Ha feltételezzük, hogy van egy 01.st állomány a home-unkban a prognyelvek/Smalltalk könyvtárban, akkor indítsuk így a Smalltalkot:
gst -q ~/prognyelvek/Smalltalk/01.st
A Smalltalk beolvassa a 01.st sorait a !-ig, majd végrehajtja az addig beírt kifejezéseket.

Magáról a nyelvről

A Smalltalk interpreter a következőképpen működik: beolvas minden karaktert az első ! –ig. A „!" jellel jelezzük, hogy végre szeretnénk hajtani az addig beírt kifejezéseket. Több kifejezés futtatása esetén itt is – mint sok más nyelven – jeleznünk kell azt, hogy hol fejeződik be egy kifejezés erre való a „pont" (.). Az utolsó kifejezés után nem pontot teszünk, hanem ! jelet. Például tekintsük a következő programot, mely értéket ad egy változónak, majd kiírja azt:

```
x := 15.
x printNl !
```

Kiírásra a `print` üzenetet is használhatjuk! A `printNl` annyiban különbözik a `print`-től, hogy az kiírás után sort is emel!

A nyelv érzékeny a kis- és nagybetűkre, de a „szóközökre" nem. A műveletek kiértékelése – ha nem zárójelezünk – mindig balról jobbra történik, így a 2+3*10 értéke 50 lesz, használjunk zárójelet: 2+(3*10).
A Smalltalk nyelv egy objektumorientált nyelv → MINDENT objektumnak tekintünk. A programozás során üzeneteket küldünk az egyes objektumoknak.
Egy objektumnak háromféle üzenetet küldhetünk:

- Unáris: *szintaxis*: <objektum> <üzenet>*pl*.: **`'Hello' printNl !`**
- Bináris: *szintaxis*: <objektum> <operátor> <argumentum> *pl*.: **`3+5`**
- Kulcsszavas: *szintaxis*: <objektum> <kulcsszó_1>**:** <arg_1> … <kulcsszó_n>**:** <arg_n>

  pl.: **`tomb at:1 put: 10`**

Egy objektumnak való üzenetküldést tekinthetünk függvényhívásnak is, azaz van visszatérési értéke.

Változók létrehozása

2 féle módszer is adódik:

1. |<változó>| <változó>**:=**<kezdeti_érték> pl:

   ```
   |x| x:=10
   ```

   Az ilyen módon deklarált változók lokálisak. A fenti példában az `x` változó elvész az értékadás után.

2. `Smalltalk at:` #<változó> `put:` <kezdeti_érték> pl:

   ```
   Smalltalk at: #x put: 10
   ```

   Az így deklarált változók nem vésznek el, véglegesek.

Blokkok

Más programozási nyelveken megismert programblokkok szerepével egyezik meg. Vannak paraméteres és nem paraméteres blokkok.
Paraméteres blokkok rendelkeznek lokális változóval (változókkal), melynek (melyeknek) a blokk kiértékelésekor adunk értéket. A változó(k) élettartama és láthatósága korlátozódik az adott blokkra. Általánosan egy paraméteres blokk a következőképpen néz ki:

[:lok.vált_1 … **:**lok.vált_n | kifejezés_1 **.** … **.** kifejezés_m**]** **value:** a_1 … **value:** a_n

A blokk eddig tart

Például:

```
[:i | i printNl ] value: 5.
```

→ kiírja, hogy 5

Egy nem paraméteres blokk kiértékelése (végrehajtása) a `value` üzenettel történik. (argumentum nélkül)
Például:

```
['Hello' print . 'world' printNl] value.
```

A blokk visszatérési értéke megegyezik az utolsó kifejezés értékével.

A cascade

Lehetőségünk van több üzenetet küldeni egy objektumnak, erre való a „;”. Példát a cascade-re a 19. feladatban találunk.

Vezérlési szerkezetek megvalósítása Smalltalk nyelven

Feltételes vezérlés

feltétel **ifTrue: [**igaz ág**]**
ifFalse: [hamis ág**]**

Pl:

```
valtozo > 10 ifTrue: ['x erteke nagyobb 10-nel' printNl]
             ifFalse: ['x erteke nem nagyobb 10-nel' printNl]
```

Ismétléses vezérlés

[feltétel] **whileTrue:** [ciklusmag blokk] Pl:

```
a:=1.
[a<10] whileTrue: [a printNl . a:=a+1]
```

Addig hajtódik végre a ciklusmag blokkja, amíg a feltétel igaz. A ciklus a **whileFalse:** üzenettel is működik. Ez persze addig futtatja a ciklusmagot, amíg a feltétel hamis.

For ciklus megvalósítása

<alsó határ> **to:** <felső határ> **by:** <lépésköz> **do:** [:<ciklusváltozó> | <ciklusutasítások>]

A **by:** <lépésköz> elhagyható, ha nincs, a Smalltalk automatikusan 1-el lépteti a ciklusváltozót. Pl:

```
1 to: 10 do: [:i | i printNl]
```

Csökkenő számlálásos ismétléses vezérlés is megvalósítható. Ekkor az <alsó határ> természetesen nagyobb, mint a <felső határ> és a lépésköz –1.

Mégegy megoldás for ciklusra: <ismétlések száma> **timesRepeat:** [ciklusmag] Pl:

```
10 timesRepeat: ['Hello' printNl]
```

A gyakorlat példafeladatai, programjai

1. Írjuk ki az 5-t a képernyőre!

```
5 printNl !
```

Az 5 – mint objektum – megkapja a `printNl` üzenetet.

2. Írjuk ki a "B" karaktert!

```
$B printNl !
```

A `printNl` (a $ jel miatt) egy karaktert tartalmazó objektumnak kerül elküldésre.

3. Írjuk ki a "Hello" szöveget!

```
'Hello' printNl !
```

Az 'hello' – mint sztringobjektum – megkapja a printNl üzenetet

4. Írjunk ki egy szóközt, majd egy "x" karaktert!

```
(Character space) print . $x printNl !
```

A Character osztály reprezentáns objektuma a Character objektum. Ennek küldünk egy space üzenetet, majd kiírjuk azt. Mindegyik osztálynak egyetlen reprezentáns objektuma van!

5. Számítsuk ki a 3 és 5 összegét, majd az eredményt írjuk ki!

```
(3+5) printNl!
```

Ha nem írunk zárójelet, akkor csak az 5-t írná ki, mivel a printNl magasabb precedenciával rendelkezik, mint a bináris + operátor

6. Írjuk ki egy tömbliterál elemeinek számát!

```
#(1 2 3) size printNl !
```

Tömbliterál létrehozása a #-al történik. Ha ennek elküldjük a size unáris üzenetet, visszakapjuk a tömb elemeinek számát.

7. Olvassuk ki egy tömbliterál első elemét és írjuk ki!

```
(#(1 2 3) at: 1) printNl !
```

Ha a létrehozott tömbliterálnak elküldjük az 1 argumentumú at: kulcsszavas üzenetet, visszakapjuk a tömb első elemét.

8. Hozzunk létre egy váltzót, adjunk neki értéket, majd írjuk azt ki!

```
|x| (x:=5) printNl !
```

Változó létrehozásának egyik módja. Inicializáljuk, majd kiírjuk.

9. A feladat ugyanaz, csak a változót másképp hozzuk létre:

```
Smalltalk at: #v put: 5.
vprintNl !
```

Változó létrehozásának másik módja. A `Smalltalk` (mint objektum, nem mint nyelv) egy előre definiált „szótár" (egy asszociatív tömb). A

10. Hozzunk létre 2 változót és többszörös értékadással adjunk nekik értéket, majd írjuk ki mindkét változó értékét!

```
|x y| x:=y:=$Z.
x printNl.
y printNl !
```

Többszörös értékadás módja:
változó1:=változó2:=…:=változón:=érték
Az egyes kifejezéseket ponttal zárjuk le!

11. A nil…

```
nil printNl !
```

A `nil` egy speciális konstans. Igazából egy változó, de mindig ugyanarra az objektumra hivatkozik.

12. Számítsuk, majd írjuk ki 4 négyzetgyökét!

```
(4 sqrt) printNl !
```

Az eredményben látjuk hogyan ábrázolja a Smalltalk a valós számokat. Ennek eredménye: 2.0

13. Számítsuk ki a 3+5*2 értékét, majd írjuk ki az eredményt! Figyeljük meg milyen eredményt kapunk!

```
(3+5*2) printNl !
```

A zárójel azért kell, hogy a kifejezés védett legyen a `printNl`-től. Ha nem lenne, 2-t írna ki. Ahhoz hogy az eredmény is helyes legyen, az 5*2-t is zárójelezni kéne.

14. Mit ír ki a következő program?

```
3+2 printNl !
```

A program a 2-t írja ki, mert a `printNl` precedenciája nagyobb, mint a bináris + operátoré.

15. Döntsük el egy számról, hogy az páratlan-e!

```
3 odd printNl !
```

A 3 objektum kap egy `odd` üzenetet, melynek visszatérési értéke természetesen a logikai true.

16. Adjuk össze a 3-at és az 5 ellentettjét!

```
(3+5 negated) printNl !
```

A `negated` üzenet az ellentettjére változtatja a címzett objektumot, így az 5-ből –5 lesz. Mivel a `negated` üzenet magasabb precedenciájú, mint a + operátor, ezért nem az összeg, hanem az 5 kerül negálásra..

17. Maximumkiválasztás. Döntsük el egy Smalltalk programmal, hogy a 3 vagy az 1+4 összeg a nagyobb?

```
(3 max: 1+4) printNl !
```

A `max:` egy kulcsszavas üzenet. Kiválasztja a címzett objektum és az argumentum közül a nagyobbikat.

18. Változtassuk meg egy tömbliterál első elemének értékét!

```
#(1 2 3) copy at: 1 put: 4; printNl !
```

Egy tömbliterál természetesen csak olvasható. Ezért abból egy másolatot kell létrehoznunk a `copy` unáris üzenet segítségével. A másolat is egy objektum, amely megkapja az értékváltoztatást végző üzenetet, melyért az `at:-put:` pár a felelős. (nem két, hanem egy üzenetként kezeljük őket). A pontosvesszővel jelezzük, hogy a `printNl` üzenetet a tömbnek küldjük.

19. Mi lesz az eredménye a következő programnak?

```
(5+3; +1; +7) printNl !
```

Ez az ún. cascade. A pontosvesszővel tudunk egy objektumnak több üzenetet küldeni. Ebben a példában az 5 objektum kapja az összes üzenetet: 5+3, 5+1 és 5+7. Kiírásra csak az utolsó számítás kerül, tehát ennek kimenete a 12.

20. Értékeljünk ki egy blokkot!

```
[1. 2. 3] value printNl !
```

Egy blokkot a `value` üzenettel értékelhetünk ki.

21. Értékeljünk ki egy üres blokkot! Mi lesz az eredmény?

```
[] value printNl !
```

Ha egy üres blokkot kiértékelünk, `nil`-t kapunk.

22. Hozzunk létre egy változót es adjunk neki értéket! Vizsgáljuk meg, hogy kisebb-e mint 4. Ha igen, írjuk ki, hogy "OK".

```
|x| (x:=3).
x<4 ifTrue: ['OK' printNl] !
```

Példa feltételes vezérlésre. `ifTrue` esetén egy logikai kifejezésnek – mint objektumnak – küldünk egy kulcsszavas üzenetet, melynek paramétere a feltételhez tartozó igaz ág. Az `ifFalse` argumentumblokkja a hamis ágat tartalmazza.

23. Hozzunk létre egy ciklust, melynek magja 4-szer fut le. A ciklusmagban a "Hello" szöveg kiíratása legyen!

```
4 timesRepeat: ['Hello' printNl] !
```

A `timesRepeat` üzenetet kapó objektum egy szám. Ennyiszer fut le a kulcsszavas üzenet argumentumblokkjában felsorolt kifejezés(ek). Ha több kifejezést szeretnénk a blokkba tenni, az kifejezéseket ponttal kell elválasszuk.

Gyakorló feladatok

1. Egész számok kiíratása 1-től N-ig.
2. Az első N darab szám összegének kiszámolása és kiíratása ciklussal.
3. Egy számokból álló tömb maximális/minimális elemének kiíratása.
4. Egy számokból álló tömb páratlan értékű elemeinek kiíratása.
5. Egy tömb megfordítása, azaz az 1. és n. elem, a 2. és n-1. elem, stb. cseréje (ahol n a tömb elemszáma).
6. Kétdimenziós tömb elemeinek sorfolytonos kiírása. Ez egy olyan tömb, melynek minden eleme tömb, és ezen belső tömbök elemeit kell egymás után kiíratni.
7. Faktoriális kiszámítása ciklussal.
8. N alatt K kiszámítása, lehetőleg nem a faktoriálisok előre kiszámításával.
9. Számokból álló tömb nagybetűs sztringgé alakítása. A tömb karakterek ASCII kódjait tartalmazza, az ezeknek megfelelő karakterekből kell sztringet csinálni úgy, hogy a kisbetűket nagybetűkre cseréljük.
10. Másodfokú egyenlet megoldása. Adottak az Ax^2+Bx+C alakú képlethez tartozó együtthatók, ezekből kell kiszámolni az egyenlet gyökeit.

Összefoglaló kérdések

1. Hányféle üzenetet küldhetünk egy objektumnak? Nevezzük meg őket és mondjunk rájuk példát!
2. Mi a műveletek precedenciája?
3. Hogy hozhatunk létre változókat? Mi a különbség köztük?
4. Hogy definiálhatunk karaktereket?
5. Hogy hozhatunk létre egy tömbliterált?
6. Milyen üzenetek léteznek kiírásra? Mi a különbség közöttük?
7. Milyen üzenetet küldjünk egy tömbliterálnak ha az elemeinek számát szeretnénk megtudni?
8. Mit jelent a „.” egy Smalltalk programban és hogyan használjuk?
9. Mit jelent a cascade és hogyan használjuk?
10. Feltételes vezérlés Smalltalkban.
11. Ismétléses vezérlések Smalltalkban.

Kollekciók

Set

A Set kollekció egy rendezetlen, ismétlődés nélküli halmazként fogható fel. A Set-nek küldhető fontosabb üzenetek:

- *new* (unáris)
  Ezt az üzenetet a Set reprezentáns objektumnak kell elküldenünk, értékül adva egy változónak, amely ezután egy set objektum lesz.
- *add*: (kulcsszavas)
  a paraméterében levő objektumot hozzáadja a halmazhoz. Ha már létezett olyan elem, akkor az nem adódik hozzá, mivel a Set ismétlés nélküli.
- *addAll:* (kulcsszavas)
  paramérete egy kollekció, mely minden olyan eleme ami eddig nem volt benne a Set-ben belekerül.

Bag

A Bag egy olyan Set, amiben megengedjük az ismétlődést. Elem hozzáadásához használjuk ugyanazokat az üzeneteket, mint a Set esetében.
Az **`add:-withOccurrences:`** párral érhetjük el, hogy egy adott elem akár többször is belekerülhessen a Bag-be.

Például az

```
x add: 'Hello' withOccurrences: 3
```

hatására az `x`-ben a 'Hello' sztring háromszor ismétlődik.

Dictionary

A Dictionary egy asszociatív tömb (egy olyan tömb, amit nem csak számokkal, hanem (itt) tetszőleges objektummal is indexelhetünk). A Dictionary is rendezetlen.

A `collect`

Általános formája:

<kollekció> **`collect:`** **[**<:változó> **|** <utasítások>**]**

A címzett kollekció elemein lépked végig, mely minden egyes elemére végrehajtja az üzenet argumentumblokkjában található utasításokat. Az aktuális elemre a blokkban definiált <:változó> segítségével hivatkozhatunk.

Pl. Duplázzuk meg egy tömb elemeit! Létrehozunk egy változót, melynek értékül adjuk a `collect` üzenetküldés eredményét.

```
|tomb|
tomb := #(10 3 43 29) collect: [:tombelem | tombelem*2]
```

A `select`

Általános formája:

<kollekció> **`select:`** **[**:<változó> **|** <feltétel>**]**

A `select` végiglépked a címzett kollekción és visszatér egy olyan elemeket tartalmazó kollekcióval, melyek megfelelnek az argumentumblokkban található feltételnek. A aktuális iterációnak megfelelő elemre a <változó>-val hivatkozhatunk.

Pl. Gyűjtsük ki egy tömb páros elemeit!

```
#(10 3 43 29) select: [:tombelem | tombelem even]
```

Objektumok összehasonlítása

Kétféle összehasonlítást különböztetünk meg: az egyenlőséget és az azonosságot. Két objektum egyenlő, ha ugyanazt az objektumot reprezentálják és azonos, ha értékük megegyezik és egyazon objektumok. Az azonosság fogalma osztályfüggetlen, amíg az egyenlőségé osztályonként változhat.

Példa összehasonlításokra:

```
|x b| x:='Ez egy string'.
b := x copy.
"Egyenlőségvizsgálat. Az x értéke egyenlő-e b értékével?"
(x = b) printNl.     → true
"Ekvivalenciavizsgálat. A x és b ugyanazt az objektumot reprezentálják-e?"
(x == b) printNl !  → false
```

A fenti két alapösszehasonlító üzenet alapján még a következő összehasonlításokat is használhatjuk, melyek az `Object` osztály összehasonlító metódusai:

~~ → az == elentettje

~= → az = ellentettje, továbbá

`isNil` → Az (== nil)-el egyezik meg

`notNil` → az `isNil` ellentettje

Másolat készítése objektumról

deepCopy (unáris üzenet)
Teljes másolat készítése objektumról.

shallowCopy (unáris üzenet)
Felszíni másolat

copy (unáris üzenet)
Osztályonként változó lehet, az `Object` osztályban a `shallowCopy`-t jelenti.

Metaosztály fogalma

Mint korában említettük, a Smalltalkban mindent objektumnak tekintünk. Még az osztályok is objektumok. De ha az osztály objektum, akkor az is - mint minden más objektum - valamilyen osztályhoz kell tartozzon. Másképp fogalmazva minden osztály (pontosan) egy másik osztály példánya. Ezen "másik" osztályt metaosztálynak hívjuk.

Az Object osztály

Az Object osztály minden osztály közös őse, tehát minden objektum az Object osztály egy példánya. Ezért minden, az Object osztálynak rendelkezésre álló művelettel minden más objektum is rendelkezik.

Ezek közül néhány fontosabb:

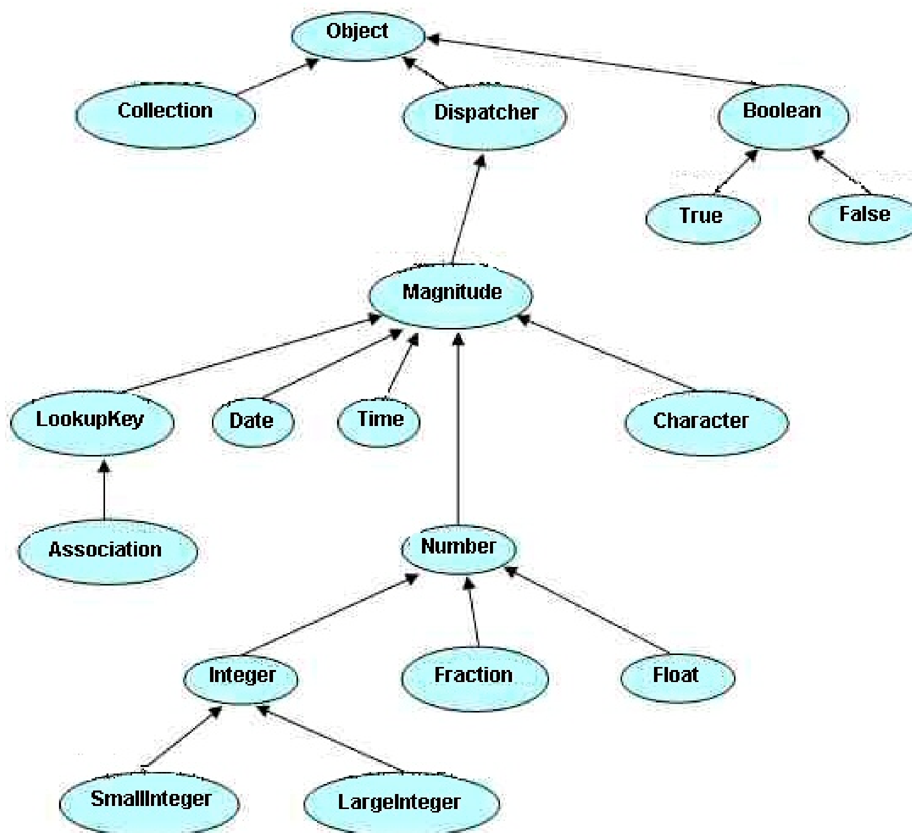
class (unáris üzenet)
Visszatérési értékként megkapjuk, hogy a címzett objetum milyen osztályhoz tartozik. Pl:

`'Hello' class !` → String

isMemberOf (kulcsszavas üzenet)
Visszatérési értéke egy logikai érték. Az üzenet argumentuma egy osztálynév. Ha a címzett objektum példánya ezen osztálynak, akkor "true" a visszatérési érték, egyébként "false". Pl:

`'Hello' isMemberOf: String !` → true

A Smalltalk osztályhierarchiája (részlet)



Osztályok létrehozása

Általános formája:

<ősosztály neve> **subclass:** <származtatott osztály neve>
instanceVariableNames: 'változó_1 változó_2 ... változó_n' → példányváltozók
classVariableNames: 'változó_1 változó_2 ... változó_n' → osztályváltozók
poolDictionaries: 'szótár_1 szótár_2 ... szótár_n'
category: nil.

Mit jelent a példány- és osztályváltozó? Tekintsünk úgy egy osztályra, mintha egy C-beli `struct` lenne. Ha C-ben létrehozunk egy „`struct` típusú" változót, akkor a struktúra

adattagjaival minden ebből a típusból deklarált változó rendelkezik. Ilyen adattagoknak képzeljük el az osztályon belüli példányváltozót. Objektumorientáltan fogalmazva: minden, egy adott osztályból példányosított objektum rendelkezik az osztály összes példányváltozójával. Természetesen ezek példányonként más és más értékeket vehetnek fel. Az osztályváltozó inkább egy C-beli `static` kulcsszóval ellátott globális változóhoz hasonlítható. Egy osztályból példányosított objektumok hozzáférnek az osztály osztályváltozóihoz, de egyik példány sem rendelkezik ilyen „adattaggal" (példányváltozóval).

A következő példában egy kör x és y koordinátája és sugara példányváltozók, mert szeretnénk, ha minden, ebből az osztályból példányosított objektum rendelkezne ezekkel az adattagokkal. A KorokSzama változó egy osztályváltozó, mert nincs szükség arra, hogy minden egyes Kor objektum tárolná az eddig létrehozott körök számát, elég ezt egy globális változóban tárolnunk:

```
Object subclass: #Kor
  instanceVariableNames: 'koord_x koord_y sugar'
  classVariableNames: 'KorokSzama'
  poolDictionaries: ''
  category: nil.
```

Metódusok definiálása osztályokhoz

Nézzünk egy komplexebb példát. Definiáljuk a fenti Kor osztályhoz a következő metódusokat (üzeneteket):

new	új példány létrehozása
init	ezzel inicializáljuk az újonnan létrehozott Kor objektumot
set_x	x koordináta beállítása
set_y:	y koordináta beállítása
sugar	kor sugarának beállítása
kiir	egy objektum példányváltozóinak kiírására fog szolgálni (ezt kétféleképpen is megvalósítjuk)

Általános formája:

!<osztálynév> <class> **methodsFor:** 'Megjegyzés'**!**
<metódus neve>
<utasítás_1 **.** utasítás_2 **.** … **.** utasítás_n>
!!

A `class` üzenet nem minden esetben kell, magyarázatot lásd lentebb.

A program teljes forráskódja:

```
Object subclass: #Kor
  instanceVariableNames: 'koord_x koord_y sugar'
  classVariableNames: 'KorokSzama'
  poolDictionaries: ''
  category: nil.
```

A `Kor` osztály definíciója, ezt nem módosítottuk.

```
  Kor comment: 'Ez egy kor osztaly definicioja'.
```

Egy osztályhoz megjegyzést is fűzhetünk.

```
  !Kor methodsFor: 'Inicializalas'!
  init
    koord_x := 0.
    koord_y := 0.
    sugar := 0
  !!
```

A `Kor` osztályhoz definiálunk egy `init` metódust. Nincs `class` kulcsszó, tehát ez egy olyan metódus lesz, amit egy `Kor` objektumnak küldünk, nem közvetlenül a `Kor` osztálynak.. Ebben a metódusban 0-ra állítjuk a példányváltozókat.

```
!Kor class methodsFor: 'Letrehozas. A konstruktor szerepet tolti be'!
 new
   |a| a := super new.
   a init.
   ^a
 !!
```

A new medtódusdefiníció. A szülőosztálynak elküldjük a new üzenetet, melynek eredményét a lokálisan deklarált „a” változóba kerül. Ezen „a” objektumnak küldjük a már definiált init metódust, végül visszatérünk az „a” változó értékével.

```
!Kor methodsFor: 'Kiiras'!
kiir
'A kor x koordinataja: ' print. koord_x printNl.
'A kor y koordinataja: ' print. koord_y printNl.
'A kor sugara: ' print. sugar printNl.
 !!
```

A kiíró metódus egyik változata. A címzett objektum koord_x, koord_y es sugar példányváltozóit írja ki a képernyőre.

```
!Kor methodsFor: 'x koordinata beallitasa'!
  set_x: ertek
  koord_x := ertek
 !
  set_y: ertek
  koord_y := ertek
 !
  set_r: ertek
  sugar := ertek
!!
```

Eddig csak unáris üzeneteket hoztunk létre. A set_x metódust egy kulcsszavas üzenetként definiáljuk, melynek argumentuma tartalmazza azt az értéket, melyet hozzá szeretnénk rendelni a címzett objektum koord_x változójához.
A set_x, set_y és set_r egyszerre kerülnek definiálásra.

```
|x| x:= Kor new.
x set_x: 50.
x set_y: 45.
x set_r: 10.
x kiir !
```

Eddig tartott az osztály és a metódusok definiálása. Elsőként példányosítjuk a Kor osztályt, értéket adunk a példányváltozóknak, majd azokat kiírjuk.

Vegyük észre, hogy a létrehozó metódus (new) kissé eltér a többi metódus definíciójától, rendelkezik egy class üzenettel. Az ilyen metódusok azokat az üzeneteket reprezentálják, melyek közvetlenül az osztálynak küldünk, jelen esetben a Kor-nek. Nyilvánvalóan egy létrehozó üzenet ebbe a kategóriába tartozik. Ha nem írunk class-t a metódusdefinícióba, akkor egy olyan metódust hozunk létre, amit – jelen esetben – egy Kor objektumnak szeretnénk majd küldeni.

Ha kiíráskor azt vesszük észre, hogy valamelyik példányváltozónk nil értéket ad vissza, akkor az érintett példányváltozó nem került inicializálásra.

A kiíró metódust úgy is megvalósíthatjuk, hogy újradefiniáljuk a Smalltalk printNl üzenetét. Ekkor a kiíró metódus a következőképpen nézne ki:

```
!Kor methodsFor: 'Kiiras'!
    printOn: stream
      super printOn: stream.
      'A kor x koordinataja: ' printOn: stream.
      koord_x printOn: stream.

      'A kor y koordinataja: ' printOn: stream.
      koord_y printOn: stream.

      'A kor sugara: ' printOn: stream.
      sugar printOn: stream
!!
```

Ebből a kódból látszik, hogy a központi kiíró metódus a printOn kulcsszavas üzenet és nem a printNl. Minden nyomtató üzenet ezt hívja meg. A nyomtatás helyét az argumentum határozza meg.

A

```
super printOn: stream
```

sor biztosítja azt, hogy a szülőosztály által nyomtatandók is megjelenítésre kerüljenek. A következő sor

```
'A kor x koordinataja: ' printOn: stream
```

a stringobjektumot nyomtatja ki, majd végül a

```
koord_x printOn: stream
```

sor hatására a koord_x objektum kerül kiírásra. Az y koordináta és a sugar példányváltozók kiírására írt metódusok teljesen hasonló módon működnek.

A super és a self

A `super` kulcsszó a `Parent::`, a `self` pedig a `this`, C++-ból ismert kulcsszavak megfelelői. Tehát egy metódusban levő self mindig arra az objektumra hivatkozik, melyre azt meghívtuk, a super pedig a szülőosztályra.

A gyakorlat példafeladatai, programjai

24. Példa szótárra

```
|x y| x := Dictionary new.
x at: 1 put: 'aaaa'.
y := x shallowCopy.
y at: 1 put: 'bbbb'.
(x at: 1) printNl !
```

x egy szótár, melynek első elemét 1-el indexeljük, értéke az 'aaaa' sztring. Ha ebből egy "shallow-másolatot" készítünk és a másolatot módosítjuk 'bbbb'-re, akkor az eredeti x objektumban is módosul az érték.

25. Példa Set-re

```
|x| x := Set new.
x add: 1.
x addAll: #(8 9 10).
x printNl !
```

Először egy elemet, majd egy tömb elemeit adjuk hozzá a Set-hez. Végül kiírjuk a halmaz elemeit.

26. Példa collect-re

```
|x y| x := #(1 2 3 4).
y := x collect: [:x|x+1].
y printNl !
```

Ez a program végiglépked az x tömb elemein, minden elemhez hozzáad egyet, majd visszatér a növelt elemeket tartalmazó tömbbel.

27. Példa select-re

```
|x y| x := #(1 2 3 4).
y := x select: [:x|x odd].
y printNl !
```

A címzett tömbobjektum elemei közül kigyűjti azokat, melyek az argumentumblokkban található feltételnek megfelelnek. A példában szereplő feltétel akkor igaz, ha az adott tömbelem páratlan (odd). Az `odd` helyett az `even`-t is használhatjuk, mely annak ellentéte (páros).

28. Példa Bag-re

```
|x| x := Bag new.
x add: 'aaaa' withOccurrences: 3.
x printNl !
```

A Bag megengedi az elemek ismétlődését. Ebben a példában az 'aaaa' sztring háromszor fordul elő.

Feladat	Magyarázat
29. Példa asUppercase-re és displayNl-re `'Hello' asUppercase displayNl !`	Az `asUppercase` üzenet a címzett objektumot nagybetűssé alakítja. A `displayNl` a `printNl`-hez hasonsó kiíró üzenet. Segítségével kiküszöbölhetjük a ’’ kiírását sztringek esetén és a $ kiírását karakterek esetén.
30. Melyik osztály objektuma a 4000? `4000 class printNl !`	A `class` üzenet annak az osztálynak reprezentáns objektumával tér vissza, melybe a címzett objektum tartozik. A 4000 a SmallInteger osztály egy objektuma.
31. Mit ír ki a következő program? `SmallInteger class printNl !`	Minden osztály rendelkezik egy reprezentáns objektummal. De ezen objektum is egy osztályhoz kell tartozzon. Itt a `SmallInteger class` egy reprezentáns objektum és a SmallInteger osztályhoz tartozik, a kimenet tehát `SmallInteger class`
32. Mit ír ki a következő program? `SmallInteger class class printNl !`	A „Smallinteger class osztály osztálya” egy Metaclass
33. Mit ír ki a következő program? `Metaclass class printNl !`	A 8. feladathoz hasonló program, kimenete `Metaclass class`
34. Mit ír ki a következő program? `Metaclass class class printNl !`	a 9. feladathoz hasonló program. Mivel itt is egy osztály osztályáról van szó, ezért függetlenül attól, hogy milyen osztályról beszélünk `Metaclass` lesz az eredmény.

A további kiadott megoldandó feladatok osztályok és metódusok létrehozásával, illetve alkalmazásával kapcsolatosak, fentebb már volt róluk szó.

Összefoglaló kérdések

11. Milyen kollekciókat ismersz, mi a különbség közöttük?
12. Mire való a `collect` iterátor?
13. Mire való a `select` iterátor?
14. Milyen lehetőségeink vannak objektumok összehasonlítására?
15. Hányféleképpen készíthetek másolatot egy objektumról?
16. Mik a metaosztályok?
17. Mit jelent a példány- és osztályváltozó?
18. Osztály létrehozásának általános formája.
19. Metódusok létrehozásának általános formája.
20. Mit jelentenek a `super` és `self` kulcsszavak? Mik ezen kulcsszavak C++-beli megfelelői?

Egy konkrét feladat megoldása: Könyv nyilvántartó

A feladat egy könyv osztály létrehozása. Az osztályból származtatott példányok könyvobjektumoknak feleljenek meg. Minden könyvről a következő adatokat szeretnénk tárolni:

- címe
- ára
- rendelkezésre álló mennyiségek száma

Az osztály legyen képes kezelni könyvek eladásának és beszerzések mennyiségét, ennek megfelelően tároljuk le globálisan (azaz egy osztályváltozóban), hogy mennyi volt eddig az eladásokból származott összbevétel.
Ehhez természetesen el kell készítsük a megfelelő metódusokat, melyek legyenek a következők:

Osztálymetódusok:

`new`

Unáris, egy (könyv)objektum létrehozásáért felelős. Hívjon megy egy `init` metódust, ami minden példányváltozót inicializál.

`beveteltnovel`

Ez a kulcsszavas üzenet növelje a bevételt tároló osztályváltozó értékét az argumentumban szereplő összeggel. Ezt a metódust majd a könyv eladásáért felelős metódus fogja meghívni.

`beveteltcsokkent`

Ez a kulcsszavas üzenet csökkenti a bevételt tároló osztályváltozó értékét az argumentumban szereplő összeggel. Ezt a metódust majd a könyv beszerzéséért felelős metódus fogja meghívni.

`bevetelinit`

A főprogram elején fogjuk meghívni. Semmi más ne csináljon, csak a bevételt tároló osztályváltozónk értékét állítsa nullára.

`bevetelerteke`

Térjen vissza az eddigi bevételt tároló változó értékével, amely egy osztályváltozó, amit közvetlenül CSAK osztálymetódusok érhetnek el. Ahhoz, hogy egy példánymetódus is hozzáférjen ehhez az adathoz, készítenünk kell egy olyan (osztály)metódust, ami visszaadja ennek értékét.
Ezt a metódust tehát egy példánymetódus fogja meghívni.

Példánymetódusok:

`init`

Állítsa üres sztringre a könyv címét, illetve 0-ra az árát és mennyiségét tartalmazó példányváltozók értékét!

`elad`

Ez a kulcsszavas üzenet regisztrálja az eladott könyvek számát. Eladáskor vonjuk ki az adott könyv mennyiségéből az eladott mennyiség számát. (azaz az adott könyvobjektum mennyiségét tároló példányváltozójának értékét csökkentsük az eladott mennyiséggel) Amennyiben kevesebb könyv áll rendelkezésre, mint amennyit eladunk, írjunk ki egy figyelmeztetést. (azaz kössük feltételhez az eladást. Ha az érintett objektum mennyiségét tároló példányváltozó értéke kisebb, mint az üzenet argumentumának értéke, akkor ne végezzük el a kivonást.)

`beszerez`

Szintén kulcsszavas üzenet, ami egy adott könyv beszerzését teszi lehetővé. (azaz valamennyivel képes növelni egy könyvobjektum mennyiségét tároló példányváltozó értékét). Mivel beszerzésről van szó, ezért a bolt bevételét tároló változó értéke nagyobb, vagy egyenlő kell legyen mint az adott könyv árának és beszerzések számának szorzata. Ha nem nagyobb, írjunk ki egy figyelmeztető szöveget!

`konyvadat`

Írja ki egy könyv objektum példányváltozóinak értékeit.

`cimmegad`

Egy kulcsszavas üzenet, amely egy könyvobjektum címét tároló példányváltozóját az üzenet argumentumában levő értékre állítja!

Az előzőhöz hasonlóan készítsük el az "`armegad:`" és "`mennyisege:`" kulcsszavas üzeneteket.

A profibbaknak még erre is maradhat idejük:

`dragabbkonyv`

Ez ismét egy kulcsszavas üzenet. Argumentuma egy másik könyvobjektum legyen. A metódus feladata azon könyv árának megjelenítése, amelyik drágább.
(Segítség: mivel az argumentum is egy ugyanolyan objektum mint a címzett objektum, ezért szükségünk lesz egy olyan (példány)metódusra ami visszatérési értékként megadja egy könyvobjektum árát.)

Ajánlott osztálystruktúra:

```
Object subclass: #Konyv
instanceVariableNames: 'cim ar mennyiseg'      → példányváltozók
classVariableNames: 'osszbevetel'              → osztályváltozó(k)
poolDictionaries: ''
category: nil.
```

Példák "főprogramra":

```
Smalltalk at: #konyv1 put: Konyv new!
konyv1 cimmegad: 'Programozzunk Smalltalk-ban';
       armegad: 2500;
       mennyisege: 10.

Konyv bevetelinit.

konyv1 elad: 5.
konyv1 konyvadat.
konyv1 elad: 20
```

```
!
```

Ennek kimenete:

```
-----------
A konyv cime: Programozzunk Smalltalk-ban
A konyv ara: 2500
Ebbol a konyvbol meg ennyi van: 5
-----------
Nincs eleg konyv az eladashoz!
```

```
Smalltalk at: #konyv1 put: Konyv new!
Smalltalk at: #konyv2 put: Konyv new!

konyv2 := Konyv new.
konyv1 cimmegad: 'Programozzunk Smalltalk-ban';
       armegad: 2500;
       mennyisege: 10.

konyv2 cimmegad: 'Smalltalk referenciakonyv';
       armegad: 1950;
       mennyisege: 20.

Konyv bevetelinit.

konyv1 dragabbkonyv: konyv2
!
```

Ennek kimenete:

```
Ezek kozul a dragabb: 2500
```

```
Smalltalk at: #konyv1 put: Konyv new!
konyv1 cimmegad: 'Programozzunk Smalltalk-ban';
       armegad: 2500;
       mennyisege: 10.

Konyv bevetelinit.

konyv1 elad: 5.
konyv1 konyvadat.
Konyv mennyiabevetel.

konyv1 elad: 3.
konyv1 konyvadat.
Konyv mennyiabevetel.

konyv1 beszerez: 6.
konyv1 konyvadat.
Konyv mennyiabevetel
!
```

Ennek kimenete:

```
A konyv cime: Programozzunk Smalltalk-ban
A konyv ara: 2500
Ebbol a konyvbol meg ennyi van: 5
-----------
A bolt eddigi osszbevetele: 12500
-----------
A konyv cime: Programozzunk Smalltalk-ban
A konyv ara: 2500
Ebbol a konyvbol meg ennyi van: 2
-----------
A bolt eddigi osszbevetele: 20000
-----------
A konyv cime: Programozzunk Smalltalk-ban
A konyv ara: 2500
Ebbol a konyvbol meg ennyi van: 8
-----------
A bolt eddigi osszbevetele: 5000
```

A program teljes forráskódja letölthető: konyv.st

Funkcionális programozás - Haskell

Egyéb segédanyag Haskell-hez

- pub/Programnyelvek/Haskell/

Telepíthető Haskell rendszerek:

- Linuxhoz és Windowhoz
  - Hugs98, ami mindkét operációs rendszerhez letölthető a pub-ból.

A Haskell használata a kabinetes termekben

Indítsunk Linux operációs rendszert, majd indítsunk egy terminálablakot (System Tools → Terminál).
Innen indítható a Hugs interpreter a `hugs` paranccsal.
További lehetőségek az indításra: `hugs +t +s` (a függvény típusát és a redukálások számát is kiírja)

A szemléletmódról és a nyelvről néhány mondatban

Egy funkcionális programozási nyelven írt programban nem a kifejezések egymásutánján van a hangsúly. A program egy függvényhívással hajtódik végre. Egy funkcionális program típus-, osztály-, és függvénydeklarációk, illetve definíciók sorozatából és egy kezdeti kifejezés kiértékeléséből áll. A kiértékelést úgy képzeljük el, mint a kezdeti kifejezésben szereplő függvények behelyettesítését. Tehát egy program végrehajtása nem más, mint a kezdeti kifejezésből kiinduló *redukciós* lépések sorozata. Egy kifejezés *normál formájú*, ha már tovább nem redukálható (nem átírható) állapotban van. Egy redukálható kifejezést *redexnek* hívunk.
Egy kifejezés redukciós lépéseinek sorrendje nyelvfüggő. Megkülönbözetünk *lusta kiértékelési* és *mohó kiértékelési* szabályokat. A lusta kiértékelés során mindig a legkülső redex kerül helyettesítésre, az argumentumokat csak szükség esetén értékeli ki. Ez a módszer mindig megtalálja a kezdeti kifejezés normál formáját.
A mohó kiértékelés az argumentumok kiértékelésével kezdődik, csak ezután hajtja végre a függvény alkalmazásának megfelelő redukciós lépést.
A Haskell nyelv a lusta kiértékelési stratégiát használja.

Példa a két kiértékelésre: (tegyük fel, hogy adottak az `inc` és `duplaz` egyargumentumú függvények. Az `inc` eggyel növeli, a `duplaz` megkétszerezi argumentumának értékét, definiálásukat lásd lentebb)

Lusta kiértékelés

```
duplaz (inc 9)
(inc 9) + (inc 9)
(9 + 1) + (9 + 1)
10 + 10
20
```

Mohó kiértékelés

```
duplaz (inc 9)
duplaz (9 + 1)
duplaz 10
10 + 10
20
```

Típusok és operátorok

Atomi típusok

`Int` (egész típusok)

Infix operátorok: `+`, `-`, `*`, `div`, `mod`, `^`

Prefix operátorok: `negate`, `-`

Float (valós típus, lebegőpontos számok)
Infix operátorok: +, -, *, /
Prefix operátorok: negate, -, sqrt, log, sin, …
Bool (Logikai típus, a True, False értékeket veheti fel)
Infix operátorok: && (logikai ÉS), || (logikai VAGY)
Prefix operátorok: not

Numerikus argumentumú infix relációs operátorok

==	(egyenlő)
/=	(nem egyenlő)
>	
>=	
<	
<=	

Lista típusok

A lista azonos típushoz tartozó elemek sorozata, melyeket a [és] jelek között sorolunk fel ,-vel elválasztva.

Pl:

[1,2,3,4]	Egy 4 elemű lista.
[]	Üres lista
[[1,2],[3],[4,5,6,7]]	Listákat tartalmazó lista

Infix operátorok:

++ Lista konkatenálás
Pl: [1,2] ++ [3,4] = [1,2,3,4]

: Egy elemből és egy listából egy újabb listát konstruál.
Pl: 4:[5,6] = [4,5,6]

Megjegyzések Haskell programban

Egysoros megjegyzés: -- ez egy megjegyzés egy sorban

Többsoros megjegyzés: {- Ez egy megjegyzés, ami akár több soros is lehetne -}

Függvények definiálása

Általánosan egy függvénydefiníció a következőképpen néz ki:

```
függvényazonosító :: <arg1 típusa> -> <arg2 típusa> … -> <visszatérési típus>
függvényazonosító arg1 arg2 … = <kifejezések>
```

A visszatérési értéket a <kifejezések> kiértékelése határozza meg, ami lehet egy konstans érték vagy akár egy rekurzív kifejezés is. Így például egy egyargumentumú duplaz nevű függvény definíciója (amely az argumentumát megduplázza) a következőképpen néz ki:

```
duplaz :: Int -> Int
duplaz x = x + x
```

Mind az argumentum, mind pedig a visszatérés típusa Int. Az argumentum értékére (a második sorban) az x-el hivatkozunk. A visszatérési érték az (x+x) kifejezés.

Esetvizsgálatok

A különböző fajta módszerek közti különbségek könnyebb láthatósága miatt mindegyik esetvizsgálatot ugyanazon a példán keresztül nézzük meg (faktoriális számítása).

Esetvizsgálat az `if` kifejezéssel

```
factorial :: Int -> Int
factorial n = if n == 0
              then 1
              else n * (factorial (n-1))
```

A visszatérési értéket az argumentumban érkező szám értékétől tesszük függővé. Ha az 0, térjünk vissza 1-el, egyébként hajtsuk végre az `else` ág rekurziós kifejezését.

Esetvizsgálat „őrökkel" (Boolean Guards)

Logikai kifejezéseket adunk meg. A függvény visszatérési értékét azon feltétel mögötti kifejezés határozza meg, amely teljesül. Az egyes feltétel-kifejezés párokat |-el választjuk el.

```
factorial :: Int -> Int
factorial n | n == 0 = 1
            | n >  0 = n * (factorial (n-1))
```

Esetvizsgálat mintaillesztéssel

Az egyes „mintákat" külön-külön sorokban definiáljuk és a függvény neve után adjuk meg. A visszatérési értéket azon minta mögötti kifejezés határozza meg, amelyikre illik az argumentum.

```
factorial :: Int -> Int
factorial 0     = 1
factorial (n+1) = (n+1) * (factorial n)
```

Esetviszgálat a `case` kifejezéssel

A hagyományos esetkiválasztásos szelekció Haskell-beli megfelelője. A teljesülő „eset" mögött megadott kifejezés határozza meg a visszatérési értéket. Jelen esetben vagy egy konstansérték vagy egy rekurzív kifejezés.

```
factorial :: Int -> Int
factorial n = case n of
              0     -> 1
              (n+1) -> (n+1) * (factorial n)
```

A gyakorlat függvényeinek definíciója

A programírás a következő szerint fog menni. Létrehozunk egy .hs kiterjesztésű állományt, ami a függvények definícióját fogja tartalmazni. Ezután elindítjuk a Haskell interpretert (hugs) és betöltjük az általunk megírt definíciós forrásállományt. Betöltés után rendelkezésre áll az összes általunk megírt függvény, melyek közül bármelyiket meghívhatjuk a függvény nevének beírásával (a megfelelő paraméterezéssel). Amennyiben módosítjuk a definíciós állományt, újra kell tölteni azt.

A hugs elindítása után forrásállományt a következőképpen tudunk betölteni:

> `:load haskellgyak.hs` (a `:load` helyett használhatjuk a `:l`-t is)

Újratöltés (az esetleges módosítások után)

> `:reload haskellgyak.hs` (a `:reload` helyett használhatjuk a `:r`-t is)

Betöltés (újratöltés) után használhatjuk a .hs forrásban definiált függvényeket.

> `add 3 4` (az `add` definíciója a haskellgyak.hs-ben van)

Segítségkérés hugs-ban: >`:?`

Kilépés a hugs-ból: >`:quit` >`:q` vagy a CTRL-D billentyűkombinációval

Ezeknek megfelelően hozzunk létre egy .hs kiterjesztésű állományt, melybe írjuk meg a következő függvényeket.

1.
```
size :: Int
size = 13 + 72
```
Ez a függvény nem vár egyetlen argumentumot sem, mert visszatérési értéke egy konstans.

2.
```
square :: Int -> Int
square n = n * n
```
A függvény négyzetre emeli az argumentumában levő számot. Az egyik `Int` az argumentum típusa, a másik a visszatérési érték típusa.

3.
```
length :: Int -> Int -> Int
length x y = y * square x + size
```
Ezen függvény a két korábbi függvényt hívja meg. Kiértékelése a másik két függvény átírásával (behelyettesítésével) történik. A három `Int` közül az első kettő az argumentumok típusai.

4.
```
exOR :: Bool -> Bool -> Bool
exOR True x = not x
exOR False x = x
```
A két `Bool` típusú argumentumra végrehajtja a logikai kizáró-vagy műveletet. (a visszatérési érték is `Bool`) A megoldáshoz mintaillesztést használunk.

5.
```
xmax :: Int -> Int -> Int
xmax x y | x >= y    = x
         | otherwise = y
```
Eldönti, hogy a két `Int` típusú argumentum közül melyik a nagyobb és anak értékével tér vissza. A megoldáshoz őröket (Boolean Guards) használunk

6.
```
zmax :: Int -> Int -> Int
zmax x y = if x > y then x else y
```
Az előző feladat megvalósítása az `if` kifejezéssel.

7.
```
fibonacci :: Int -> Int
fibonacci x
  | x == 0 = 0
  | x == 1 = 1
  | x >  1 = fibonacci (x-2) + fibonacci (x-1)
```
A Fibonacci számsorozat számítása őrökkel, rekurzióval

8.
```
xminmax :: Int -> Int -> (Int,Int)
xminmax x y
  | x > y =     (x,y)
  | otherwise = (y,x)
```
Két input számot (`Int -> Int`) vár a függvény. Visszatérési értéke a két szám növekvő sorrendben., tehát a függvény visszatérési értékének típusa egy „`Int` pár”.

9. Típusdefiníció

```
type Pair = (Int,Int)
```
Ha később egy „`Int`-pár” adattípust szeretnénk használni, akkor a `Pair`-el egy ilyen típusra hivatkozhatunk. Egy típusra hivatkozhatunk a definícióját megelőző ponton is.

10.
```
minmax :: Int -> Int -> Pair
minmax x y = (min x y,max x y)
```
Itt hivatkozunk a fent definiált `Pair` típusra. Tehát ezen függvény visszatérési értéke `(Int,Int)` lesz. A függvény növekvő sorrendbe szervezi a két argumentumot, melyhez két előre beépített függvényt használunk. (`min`, `max`)

11. Egy újabb típusdefiníció

```
type StrFlt = (String,Float)
```
Mostantól az `StrFlt` (mint típus) egy `String` és egy `Float` típusokból álló pár.

12.
```
which :: StrFlt -> StrFlt -> String
which (s1,f1) (s2,f2)
  | f1 < f2   = s1
  | otherwise = s2
```
Az `StrFlt` típus felhasználása. Azaz két argumentum egy `String` és `Float` típusú értékek által alkotott pár. A függvény azon `String` értékkel tér vissza, amelyikhez kisebb `Float` tartozik.

13.
```
second :: Pair -> Int
second (x,y) = y
```
Két input szám közül a második értékével tér vissza. A két input szám (`Int,Int`) típusú.

14. Összegző

```
sum x y = x + y
```

A két input szám összegével tér vissza.

15. Lista megfordítása

```
rev :: [a] -> [a]
rev [] = []
rev (h:t) = rev t ++ [h]
```

A `rev` függvény az argumentumában levő listával tér vissza fordított sorrendben. A mintaillesztés szerint az üres lista esetén üres listával térünk vissza, egyébként pedig meghívjuk a rekurzív kifejezést. A `(h:t)` a listát fej-, és farokrészre (head:tail) bontja. A fej a lista első eleme, a farokrész a többi elemet tartalmazza. A rekurzió minden iterációjában az ahhoz tartozó (h:t) lista leghátsó elemét a legelső mögé konkatenáliuk.

Ezek önmagukban nem programok, hanem csak függvénydefiníciók sorozata. Ahhoz hogy ezeket a függvényeket kihasználhassuk be kell töltsük hugs-ban. (`:l haskellgyak.hs`)

Sikeres betöltés után bármelyik függvényt meghívhatjuk. Hívjuk meg mindegyik függvényt és nézzük meg a visszatérési értékeket. (A hugs által visszaadott értékeket a „→” után írtam)

`size`	→ 85
`square 4`	→ 16
`length 2 3`	→ 97
`exOR True False`	→ True
`xmax 2 7`	→ 7
`zmax 2 7`	→ 7
`factorial 5`	→ 120
`fibonacci 6`	→ 8
`xminmax 3 4`	→ (4,3)
`xminmax 2 3`	→ (3,2)
`minmax 3 4`	→ (3,4)
`which ("aa",1) ("bb",2)`	→ "aa"
`second (9,3)`	→ 3
`sum 4 6`	→ 10
`rev "abcdefg"`	→ "gfedcba"
`rev [1,2,3,4]`	→ [4,3,2,1]
`rev [1..10]`	→ [10,9,8,7,6,5,4,3,2,1]
`rev ['a','c'..'m']`	→ "mkigeca"
`rev [[1,2],[1,2,3]]`	→ [[1,2,3],[1,2]]
`rev [rev[1,2],rev[1,2,3]]`	→ [[3,2,1],[2,1]]

A többargumentumú függvényekről

A többargumentumú függvények is valójában egyargumentumúak. Egy többargumentumú függvény egy olyan egyargumentumú függvény, ami függvényt ad vissza. Lesznek olyan függvénydefinícióink, amik a múlt gyakorlaton általánosan felírt függvénydefiníciótól eltérő lesz (teljes paraméterezésű függvények voltak). A részleges paraméterezés lehetővé teszi, hogy egy részlegesen paraméterezett függvényt más függvények definíciójaként adjuk meg. Tekintsük a következő példát:

```
mul :: Int -> Int -> Int
mul x y = x*y

duplaz :: Int -> Int
duplaz = mul 2
```

A `mul` függvény definíciójában semmi újdonság nincs, ám ha azt részleges paraméterezéssel használjuk (nem kap meg minden paramétert), akkor azt felhasználhatjuk más függvények (itt a `duplaz`) definíciójában. Így hívástól függően a `mul` vagy egy függvényt, vagy egy egész számot ad vissza. A mul függvény első sora így is kinézhetne, de a zárójelezés elhagyható: `mul :: Int -> (Int -> Int)`

Függvény paramétere függvény

Erre kiváló példa a `map` függvény. A `map` egy listát jár be és minden egyes elemére végrehajt egy függvényt, majd az eredményt egy listába gyűjti össze. Előre definiált, a definíciója:

```
map :: (a->b) -> [a] -> [b]
map f []    = []
map f (h:t) = f h : map f t
```

A mintaillesztésnek megfelelően ha az input illik arra a mintára, hogy egy függvény (f) és egy üres lista ([]), akkor térjen vissza üres listával, illetve ha nem üres lista érkezik a függvény után, akkor hajtsuk végre az „f” függvényt a h (head) argumentummal és konkatenáljuk hozzá a „t” (tail)-re vonatkozó rekurzív hívás eredményéhez.

Ennek megfelelően a

```
map (mul 2) [1,2,3,4]
```

eredménye `[2,4,6,8]` lesz.

Előre definiált függvények helyett használhatunk „konstans módon” definiált függvényeket is. Ugyanezt az eredményt kapjuk a következő függvényhívással is:

```
map (\x -> x*2) [1,2,3,4]
```

Ezeket nevezzük anonim függvényeknek.

Az add függvény (ami a paraméterekben érkező számok összegével tér vissza) most következő két megvalósítása teljesen megegyezik:

```
add :: Int -> Int -> Int
add x y = x + y
```

```
add :: Int -> Int -> Int
add = \x y -> x+y
```

A második definíciót anonim függvényekkel valósítottuk meg.

Lokális definíciók függvénydefiníciókban

Kétféle lehetőség kínálkozik: a `let` és a `where`. Nézzük a következő példát: számítsuk ki egy derékszögű háromszög átfogójának hosszát, ha adott a két befogó. Lokális definíciók nélkül a függvény így nézne ki:

```
atfogo :: Float -> Float -> Float
atfogo a b = sqrt(a*a + b*b)
```

Ha ezt a példát lokális definíciók segítségével szeretnénk megoldani (a `let` segítségével), akkor a forráskód:

```
atfogo_let :: Float -> Float -> Float
atfogo_let a b =
  let negyzetre n = n*n
  in
    sqrt(negyzetre a + negyzetre b)
```

Ha szövegesen kéne elmondjuk, hogy mit csinál a lenti lokális definíció, akkor azt valahogy így lehetne: „legyen (`let`) a negyzetre függvény egy olyan függvény, ami az argumentumát négyzetre emeli abban a kifejezésben (`in`), hogy `sqrt(negyzetre a + negyzetre b)`”. Ez a példa ahhoz túl rövid és egyszerű, hogy a lokális definíció nagyon hasznosnak tűnjön, de egy hosszabb függvénydefiníciót rövidebbé és könnyebben olvashatóbbá teheti. A `where` esetén így nézne ki a függvény:

```
atfogo_where :: Float -> Float -> Float
atfogo_where a b = sqrt(negyzetre a + negyzetre b)
  where
    negyzetre n = n*n
```

Tehát legyen a `atfogo_where` visszatérési értéke az `sqrt(negyzetre a +negyzetre b)`, ahol (`where`) a `negyzetre` egy olyan függvény legyen, ami az argumentumát négyzetre emeli. Látható, hogy a `where` a függvény végén definiálja az „új” függvényt a `let`-el ellentétben.

A `where`-t esetvizsgálatkor is felhasználhatjuk:

```
fgvny x y | y>z          = ...
          | y==z         = ...
          | y<z          = ...
        where z = x*x
```

Ezt a `let` segítségével nem tehetjük meg az általa definiált függvény láthatóságának korlátozottsága miatt, ugyanis a `let` által definiált függvény az `in` utáni kifejezésben lesz csak látható.

Fontos, hogy mindkét esetben lényeges a tabulálás. Sem a `where`, sem a `let` nem kezdődhet sor elején. A `let` esetén további megkötések is vannak: az `in` sem kezdődhet sor elején és az `in` utáni függvényhívás nem kezdődhet kijjebb, mint az `in` maga.

Listák feldolgozása

Listaelemek feldolgozására már láttunk példát (`map`, ami a lista minden egyes elemére végrehajt egy függvényt, majd az eredményt egy listába gyűjti).

`filter` – listaelemek valamilyen feltétel szerinti megszűrése

filter even [1,2,3,4,5,6,7,8] (kiírja, hogy [2,4,6,8,10])
filter odd [1,2,3,4,5,6,7,8] (kiírja, hogy [1,3,5,7,9])

`fold` – elemek kombinálása

foldr1 (+) [1..4] (azaz 1+2+3+4, ami 10, az 1 a függvény nevében azt jelenti, hogy minimum 1 elemű kell legyen az input lista)
foldr1 (*) [1..4] (azaz 1*2*3*4, ami 24)
foldr1 (max) [1..4] (azaz 1 'max' 2 'max' 3 'max' 4, ami 4)
foldr (+) 0 [1..8] (a `foldr` függvény elfogad üres listát is inputként. Ekkor a visszatérési érték az operátor után megadott érték lesz.)

`zip`, `unzip` – listaelemek átszervezése. A `zip` egy listával tér vissza, ami a két input lista minden n-edik elemét rendre párban adja vissza. Az `unzip` a `zip` elentettje, azaz egy olyan listát vár inputként, mint amilyen a `zip` kimenete és visszatérési értéke olyan (a zárójelezéstől és a vesszőtől eltekintve), mint amilyet a `zip` vár. Pl:

zip [1,2,3] "abc" (kimenete: [(1,'a'),(2,'b'),(3,'c')]
unzip [(1,2),(3,4)] (kimenete: ([1,3],[2,4])

Függvények kompozíciója

Legyen adott egy lista: [1..8]. Válogassuk ki belőle azokat az elemeket, melyek kettővel osztva 0-át adnak maradékul. Ahelyett, hogy az egyszerű egysoros

```
filter even [1..8]
```

függvényt használnánk, használjuk két függvény kompozícióját. Az egyik függvény elvégzi a kettes modulus képzést, a másik függvény ellenőrzi, hogy az 0-e. Két függvény kompozíciójának általános formája:

(f . g) x (ami megegyezik azzal, hogy **f(g x)**)

Ennek megfelelően a függvényünk:

```
filter ((==0).(`mod` 2)) [1..4]
```

A mod-ot azért kell `` jelek közé tenni, mert így operátorként fogja kezelni a Haskell.

FIGYELEM! Csak `` (AltGr-7) jelek jók, az '' jelek nem!

A fenti függvényhívás először elvégzi a listaelemekre a (`mod 2`)-t (működik így, mert a itt a `mod`-ot mint operátort kezeljük), majd ellenőrzi, hogy 0-e (`==0`)

Típusok létrehozása

Valódi típuslétrehozásról van szó, nem pedig típus szinonímák használatáról. (emlékeztetőül: a típus szinoníma nem új típust hoz létre, hanem már létező típus(ok)ból egy új névvel ellátott összetett típust konstruál). Új típus létrehozására a `data` deklarációt használjuk. A `Bool` Haskell egy előre definiált típusa, melynek definíciója így néz ki:

```
data Bool = False | True
```

A |-al ezen típus által felvehető értékeket adhatjuk meg. Ehhez hasonlóan definiálhatunk egy színeket tároló típust:

```
data Color = Feher | Kek | Fekete | Zold | Narancssarga
```

Mind a Bool, mind pedig a Color felsorolás (enumerated) típusok, melyek nem rendelkeznek ún. adatkonstruktorral (data constructor).

Hozzunk létre egy bináris-fa típust. Ezt a típust rekurzív, polimorf típusként fogjuk definiálni. Egy olyan bináris fát fogunk létrehozni, melynek csak a leveleiben tárolunk értékeket, a belső pontokban nem.

```
data BinFa a = Level a | Reszfa (BinFa a) (BinFa a)
```

Ez tehát egy polimorf bináris fa típus, melynek elemei a következőkből kerülhet ki: vagy egy levél, amely egy ’a’ típusú (itt a polimorfizmus) értéket tárol, vagy egy belső fa-pont, azaz egy tovább ágazó BinFa (itt a rekurzió) A BinFa megint vagy egy levél lehet vagy egy másik belső pont lehet.

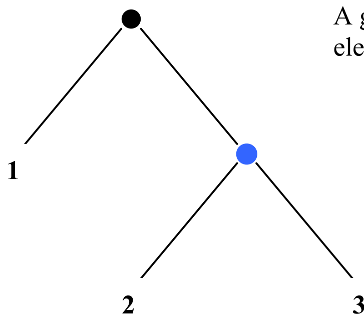
Írjunk meg egy olyan függvényt, amely egy fent definiált típussal rendelkező bináris fát bejárja és kiírja listaként a levélelemeinek értékét.

A függvény így néz ki:

```
bejar :: BinFa a -> [a]
bejar (Level x) = [x]
bejar (Reszfa bal jobb) = bejar bal ++ bejar jobb
```

A fejléc egyértelmű, egy ’a’ típusú BinFa az input és egy szintén ’a’ típusú lista az output. Mintaillesztést használunk. Azaz, ha egy levélelem érkezik, akkor azt írjuk ki, ha nem, azaz ha az input illik arra, hogy két bináris fa típusú részfa, akkor rekurziót használva járjuk be a bal, majd a jobb részfáját (a ++ operátor listákat konkatenál).

A függvény kész, töltsük be hugs-ba, majd hívjuk meg a megfelelő paraméterezéssel. Építsük fel a következő bináris fát, majd hívjuk meg rá a „bejar” függvényt.



A gyökérpont egy Reszfa pont lesz, melynek egyik eleme egy Int típusú, 1 értéket tároló Level érték.

Akárcsak a gyökér, ez is Reszfa pont lesz, de ennek mindkét eleme szintén Int típusú Level pont lesz.

A fenti ábrának megfelelően a felépített fa a következőképpen kéz ki:

```
Reszfa (Level 1) (Reszfa (Level 2) (Level 3))
```

A kék színnel jelzett csúcs egyik levele

A kék színnel jelzett csúcs másik levele

A fekete színnel jelzett csúcs bal részfája egy Int típusú levélcsúcs, értéke: 1

A fekete színnel jelzett csúcs jobb részfája egy szintén részfát tartalmazó csúcspont lesz.

Erre már csak a bejar függvényt kell meghívni és meg is kapjuk a bináris fa bejárásának eredményét listában.

```
bejar (Reszfa (Level 1) (Reszfa (Level 2) (Level 3)))
```

Ennek kimenete az [1,2,3], Int típusú elemeket tartalmazó lista.

A Binfa típus és a bejar függvény polimorf voltának köszönhetően olyan bináris fára is működik a függvény, amelynek levelei szöveget tárolnak, azaz a következő függvényhívás ugyanúgy érvényes:

```
bejar (Reszfa (Level ″egy″) (Reszfa (Level ″ketto″) (Level ″harom″)))
```

Kimenete egy [Char] típusú lista:

```
[″egy″, ″ketto″, ″harom″]
```

A lusta kiértékelésről bővebben

Emlékeztető: A lusta kiértékelés során mindig a legkülső redex (redukálható kifejezés) kerül helyettesítésre, az argumentumokat csak szükség esetén értékeli ki. Ez a módszer mindig megtalálja a kezdeti kifejezés normál formáját.
A lusta kiértékelés ennélfogva gyorsabb lehet, mert nem feltétlenül értékelődik ki egy függvény argumentuma. Legyen egy „lusta” függvényünk, amely két Float típusú argumentumot vár. Ha az egyik nagyobb, mint 0, térjünk vissza a másik input számmal, egyébként 1-el. Ez a függvény:
lusta :: Float -> Float -> Float

```
lusta x y =
            if x>0
              then y
              else 1
```

Nyilván a **lusta 1 (0/1)** kimenete a 0/1 értéke lesz, mert a másik argumentum értéke nagyobb mint 0. Mi lesz az eredmény ha 0/1 helyett 1/0-t írjuk? Nyilvánvalóan hibaüzenetet kapunk, mert 0-val osztani értelmetlen. Hívjuk meg a függvényt a következő argumentumokkal:

```
lusta 0 (1/0)
```

Nem kéne meglepő legyen az, ha ez is hibaüzenetet generálna, hiszen az egyik argumentum szintén értelmetlen. A függvény definíciója szerint ha az egyik argumentum értéke nem nagyobb mint 0, a másik argumentum értékétől függetlenül 1-el térjünk vissza. Így ha az első argumentum nem nagyobb mint 0, akkor a második argumentum nem is kerül kiértékelésre, amit abból látunk, hogy az 1/0 nem generál hibát (mivel tehát ki sem értékelődik).

A lusta kiértékelésnek köszönhetően végtelen adatszerkezetek átadhatók argumentumként. Definiáljuk a következő rekurzív kifejezést:

```
nat :: [Int]
nat = 1 : map (+1) nat
```

A függvény az 1-hez konkatenálja a nála eggyel nagyobb számot. Ez egy olyan rekurzív kifejezés, melynek nincs normál formája, azaz a normál formát végtelen sok redukciós lépéssel sem találja meg. Egy Int típusú listával tér vissza, ami „1-től végtelenig tartalmazza" a számokat (persze csak képletesen értve). Függetlenül attól, hogy ez egy végtelen rekurzió, mégis használható a lusta kiértékelés miatt. Írjunk egy olyan függvényt, ami egy bármilyen típusú lista első n elemét kiírja:

```
prefix :: [a] -> Int -> [a]
prefix (h:_) 1 = [h]
```

```
prefix (h:t) n = h : prefix t (n-1)
```

Tehát inputként egy típustól független listát és egy egész számot vár, visszatérési értéke az input lista első `n` eleme, melynek típusa megegyezik az input lista típusával.

A mintaillesztés első egyenletében található _ tetszőleges értéket helyettesít, azaz ha egy olyan listát kapunk az inputban, aminek van fej-eleme (azaz nem üres) és mindössze 1 elemét szeretnénk kiírni, akkor térjünk vissza a [h] listával. Emennyiben van farok része a listának, hívjuk meg a rekurzív kifejezést, aminek hatására a tail-t n-1 hosszal hozzákonkatenáljuk h-hoz. Hívjuk meg a függvényt a nat (azaz a végtelen lista) argumentummal:

```
prefix nat 5
```

A függvény hibajelzés nélkül a végtelen lista első 5 elemével tér vissza, az [1,2,3,4,5]-el.

Logikai Programozás - Prolog

Egyéb segédanyagok Prolog-hoz

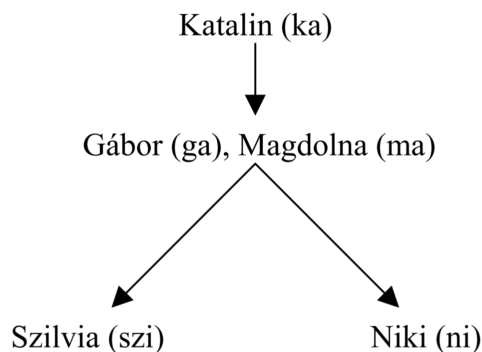
/pub/Programnyelvek/Prolog
Itt van egy magyar nyelvű Prolog segédanyag is

Telepíthető Prolog rendszerek

Windowshoz és Linuxhoz: (SWI-Prolog) – fent van a pub-ban (w32pl*.exe)
http://www.swi-prolog.org
Linuxhoz még (SiCStus Prolog)

Egy bevezető példa

A logikai programok egy modellre vonatkoztatott állítások halmaza, melyek a modell tulajdonságait és azok között fellépő kapcsolatokat (relációit) írják le. Egy adott relációt meghatározó állítások részhalmazát predikátumnak nevezzük. A predikátumokat alkotó állítások *tények* vagy *szabályok* lehetnek. A tényeket és szabályokat (és majd a Prolognak feltett kérdéseket is) ponttal zárjuk le.
Tekintsük a következő példát, mely egy család tagjai között fellépő kapcsolatot írják le.



Ennek megfelelően a családot leíró tények halmaza a következő:

```
szulo(ka,ma).
szulo(ka,ga).
szulo(ga,szi).
szulo(ma,szi).
szulo(ga,ni).
szulo(ma,ni).
```

A `szulo` predikátum argumentumait szándékosan írtuk kis betűkkel. A kis betűkkel írtakat a Prolog konstansként kezeli. (ka, katalin, szilvia, stb…) Minden nyomtatott nagybetűt vagy nagy kezdőbetűvel kezdődőket változónak tekinti. (X, Y, Szilvia, Magdolna, stb…) A fenti 5 sort gépeljük be egy .pl kiterjesztésű szöveges állományba, majd mentsük el. A kiterjesztés szigorúan pl kell legyen!
A Prolog egy terminálablakba beírt „sicstus" paranccsal indítható. Egy Prolog állományt a következőképpen „tölthetjük be": (feltéve, hogy az aktuális könyvtárban létezik egy prolog.pl állomány)

`consult(prolog).` vagy **`[prolog].`**

Ezután kapunk egy „| ?-" promptot, ahova a Prolog várja az általunk feltett kérdéseket. A fenti család struktúra alapján feltehetnénk azt a kérdést, hogy Magdolna szülője-e Szilvinek. A Prolognak feltett kérdés a következőképpen néz ki:

```
szulo(ma,szi).
```

Mivel ez egy igaz állítás, ezért a Prolog a „yes" válasszal tér vissza. Nyilván fordított paraméterezéssel a „no" választ kapnánk.
Mit írjunk a Prolognak, ha a következő kérdésre szeretnénk választ kapni: „*Van-e olyan X, akinek Gábor az apja?*". Ekkor az ismeretlen helyébe egy változót írunk: Változókat nagy nyomtatott betűkkel vagy nagybetűvel kezdődő „szavakkal" jelölünk!

```
szulo(ga,X).
```

Mivel Gábor 2 embernek is szülője, ezért a Prolog visszatérési értéke nem egy érték lesz, egyenként tér vissza az egyes értékekkel, először:

```
X = szi
```

Itt vagy ENTER billentyűt ütjük le, nyugtázva, hogy megkaptuk a választ kérdésünkre, vagy ;-t, amivel azt jelezzük, hogy további alternatívákat kérünk. Ezesetben a következő lehetséges értékkel tér vissza a Prolog:

```
X = ni
```

Itt is vagy ENTER-t ütünk, vagy ;-t. Ha ;-t ütünk olyan esetben, amikor már nincs több alternatíva a Prolog a „no" válasszal tér vissza, egyébként (ENTER esetén) a „yes"-el.
Mit kell tegyünk ha azt a kérdést szeretnénk feltenni, hogy Katalin nagyszülője-e Szilvinek? Ehhez már szabályt kell definiáljunk (az eddigiek tények voltak). Definiáljuk általánosan az „nsz" szabályt:

```
nsz(X,Y) :- szulo(X,Z), szulo(Z,Y).
```

Azaz, X az Y nagyszülője, ha X szülője Z-nek és Z szülője Y-nak. Ebből az is látszik, hogy hogyan tudunk szabályokat definiálni Prologban.
Mivel módosítottuk a tényeket (és mostmár szabályt is) tartalmazó állományunkat, újra kell töltsük, amit a betöltésnél beírt utasítás megismétlésével tehetünk meg. ([prolog].)
Mostantól a nagyszülőre vonatkozó kérdéseket is feltehetjük:

```
nsz(ka,szi).
```

(azaz, Katalin nagyszülője-e Szilvinek?)
vagy általánosan is kérdezhetünk a fenti példához hasonlóan:

```
nsz(ka,X).
```

Ekkor mindkét lehetőséggel visszatér a Prolog. Ha a tények között szerepel mind a `szulo(ka,ma).` és `szulo(ka,ga).` tény, akkor az egyes visszatérési értékekkel kétszer tér vissza a Prolog. (X=szi; X=ni; X=szi; X=ni). Ez azért van, mert Katalintól Szilviához és Nikihez két féleképpen is el lehet jutni (egyszer Gáboron, egyszer pedig Magdolnán keresztül).

Feladat: Mindenki készítsen Prolog tényeket (és szabályt) saját családja alapján!

A Prolog program felépítése

Egy Prolog program tények

```
szulo(katalin, magdolna).
```

szabályok

```
os(X,Y) :- szulo(X,Y).
```

és egy cél

```
os(katalin,X).
```

halmazából áll.

Termek

- Egyszerű termek (implementációfüggő)

Term fajtája	Értékek
boolean	true, fail
integer	egész számok
real	valós számok
variable	változók (nyomtatott nagybetűvel kezdődik) Értékadás az = jellel történik: `X = 5.`
atom	karaktersorozatok (idézőjelek közti szöveg vagy kisbetűvel kezdődő szöveg)

- Összetett termek
  - Lista

    Nagyon hasonlít a Haskell-ben megismert listára. Itt sincsenek indexelve az elemek, rekurzióval fogjuk bejárni a listát. Példa listára:

    ```
    [1,2,3,4,5].
    ```

    A karaktereket a Prolog karakterkódokkal azonosítja (akárcsak a C).

    `X = "abcd".` (ennek kimenete: `X = [97, 98, 99, 100]`)

Kiértékelés

Kifejezések kiértékelésére a beépített, infix `is` operátort használhatjuk. Általános alakja:

<szabad változó> `is` *<kifejezés>*. vagy *<szabad változó>* `is` *<kötött változó>*.

Pl:

`X is (3+5).` Kimenete: `X = 8`

Aritmetikai Operátorok

+	összeadás
-	kivonás
*	szorzás
/	osztás
//	egészosztás
mod	modulusképzés
**	hatványozás

Az infix operátorok az alábbi alakban is írhatók:

3+4	+(3,4)
3**4	**(3,4)
3 mod 4	mod(3,4)

stb…

A következő részt a logikai operátorokról Kertész Attila jegyzetéből szedtem:

		Egyesítés		Azonosság		Aritmetika		
U	V	U = V	U \= V	U == V	U \== V	U =:= V	U =\= V	U is V
1	2	no	yes	no	yes	no	yes	no
a	b	no	yes	no	yes	error	error	error
1+2	+(1,2)	yes	no	yes	no	yes	no	no
1+2	2+1	no	yes	no	yes	yes	no	no
1+2	3	no	yes	no	yes	yes	no	no
3	1+2	no	yes	no	yes	yes	no	yes
X	1+2	X=1+2	no	no	yes	error	error	X=3
X	Y	X=Y	no	no	yes	error	error	error
X	X	yes	no	yes	no	error	error	error

Jelmagyarázat: yes — siker; no — meghiúsulás, error — hiba.

U = V: U egyesítendő V-vel. Soha sem jelez hibát.	?- X = 1+2. ⟹ X = 1+2 ?- 3 = 1+2. ⟹ no
U == V: U azonos V-vel. Soha sem jelez hibát és soha sem helyettesít be.	?- X == 1+2. ⟹ no ?- 3 == 1+2. ⟹ no ?- +(1,2)==1+2 ⟹ yes
U =:= V: Az U és V aritmetikai kifejezések értéke megegyezik. Hibát jelez, ha U vagy V nem (tömör) aritmetikai kifejezés.	?- X =:= 1+2. ⟹ hiba ?- 1+2 =:= X. ⟹ hiba ?- 2+1 =:= 1+2. ⟹ yes ?- 2.0 =:= 1+1. ⟹ yes ?- 2.0 is 1+1. ⟹ no
U is V: U egyesítendő a V aritmetikai kifejezés értékével. Hiba, ha V nem (tömör) aritmetikai kifejezés.	?- X is 1+2. ⟹ X = 3 ?- 1+2 is X. ⟹ hiba ?- 3 is 1+2. ⟹ yes ?- 1+2 is 1+2. ⟹ no
(U =.. V: U „szétszedettje” a V lista)	?- 1+2 =.. X. ⟹ X = [+,1,2] ?- X =.. [f,1]. ⟹ X = f(1)

Illesztés

Az előadásjegyzetből:

- Kötött változó értékével vesz részt
- Konstans önmagával illeszkedik
- Ha csak az egyik szabad változó, akkor illeszkednek és a változó kötötté válik
- Ha két szabad változó, akkor illeszkednek és szabad megosztott változók lesznek
- Összetett termek illeszkednek, ha a funktor és a résztermek száma megegyezik, ill. a résztermek illeszkednek

Ezek alapján vizsgáljuk, hogy az alábbi kifejezések illeszkednek-e:

`point(X, Y) = point(A, B).`

Illeszkednek, mert mindegyik változó (X, Y, A, B) szabad. Illeszkedés után szabad megosztott változók lesznek (X az A-val, Y a B-vel).

`f(X, X) = f(a,b).`

Nem illeszkednek, mert az X változó nem tud egyszerre a-ra és b-re is illeszkedni. Egy apró módosítással illeszkednének: **`f(X, Y) = f(a, b).`**

`f( X, a(b, c)) = f( Z, a(Z, c)).`

Illeszkednek. A Z szabad változó kötött lesz (b) és az X is kötött lesz (X=Z=b)

`operation( X, 5+3) = operation(7, 8).`

Logikus lenne, hogy illeszkednek, de nem illeszkednek, mert az 5+3 nem kerül kiértékelésre. Ha az 5+3 helyére 8-at írnánk, akkor illeszkednének, mert az X változó kötötté válik az 7 értékkel.

`A*7+b = +(b, *(A, 7)).`

A jobb oldali kifejezés `(b+A*7)`, tehát nem illeszkednek, attól függetlenül, hogy kiértékelés esetén ugyanazt az eredményt kapnánk. Az illeszkedés azért nem teljesül, mert a Prolog az + operátor két oldalát próbálja illeszteni, `A*7+b`-t a `b`-hez, ami nem teljesül. Írjuk át a kifejezéseket (az operátorokat a jobb oldalon továbbra sem infix módon használva) úgy, hogy illeszkedjenek. A bal oldali kifejezés legyen `b*(A+7)`. Ehhez illeszkedik a következő kifejezés: `*(b, +(A, 7))`.

`pilots(A, london) = pilots(london, paris).`

Nem illeszkednek, mert a london <> paris. Ha mindkét helyen paris állna, illeszkednének, az A változó kötött lesz, felveszi a london értéket:

```
pilots(A, paris) = pilots(london, paris).
A = london
```

A gyakorlat feladatai

16. Kérdések konkatenációja

```
likes(joe, fish).
likes(john, wine).
likes(mary, john).
likes(mary, book).
likes(joe, mary).
likes(john, book).
```

A definiált tényhalmazból azt szeretnénk megtudni, hogy van-e olyan X, melyet John és Mary is szeret. Összetett kérdéseket a "vessző" segítségével tehetünk fel. A vesszővel elválasztott kérdések mindegyike teljesülni fog. A fenti kérdésre a következő kérdéssel kapunk választ:

```
likes(john, X), likes(mary, X).
```

Ennek kimenete:

X = book

---

17. Szabály definiálása

```
likes(john, X) :- person(X).

person(mary).
person(joe).
person(david).
```

A szabály szerint John csak "olyan dolgokat" szeret, melyekre igaz az, hogy ők személyek.
A szabály és tényhalmaz definiálása után a

```
likes(john, Z).
```

kérdés kimenete a 3 személy lesz:

```
X = mary ;
X = joe ;
X = david
```

---

18. Kiértékelés, értékadás

```
X is 2+3, X =:= 2+3.
```

Az első kifejezés az X változónak a 2+3 értéket adja. A `=:=` logikai operátor kiértékeli a bal és jobb oldalán levő kifejezéseket, majd ellenőrzi azok egyenlőségét. A Prolog az X = 5-el tér vissza.

---

19. Karakterek kóddal való aznosítása

```
X = "abcd".
```

A Prolog – akárcsak a C – kódokkal azonosítja a karaktereket. Az `X = "abcd"` hatására a Prolog a következő listát hozza létre:

```
X = [97, 98, 99, 100]
```

---

20. Tartalmazás ellenőrzése

```
member( X, [X| _]).
member( X, [_| T]) :- member( X, T).
```

A member ellenőrzi, hogy egy elem eleme-e egy listának. Akárcsak Haskell-ben, itt is fej-, és farokrészre bontjuk a listát. Haskellben (h:t)-vel hivatkoztunk, itt [X|T]-vel, ahol X és T változók. Ha a keresendő elem a lista „fej eleme", teljesül az illeszkedés, egyébként a farok részre – mint listára – rekurzívan „meghívjuk" a membert.

A `member` predikátum értelmezése

Kövessük végig hogy kell értelmezni és hogy működik a már megírt `member` predikátum. Egy tényt és egy szabályt definiáltunk:

```
member( X, [X| _]).
member( X, [_| T]) :- member( X, T).
```

Feltéve, hogy a Prolog interpreterbe betöltöttük a `member` definícióját tartalmazó forrást, tegyük fel a következő kérdést a Prolognak:

```
member(c,[a,b,c,d]).
```

A Prolog olyan mintákat keres, melyre illeszkedik az általunk feltett kérdés. Az első alternatíva a tény. Ha egy tényre illeszkedik egy kérdés, akkor a Prolog a „yes" válasszal tér vissza. Erre már láttunk példát: a `likes(john,book).` tény létezése esetén a `likes(john,book).` kérdésre a Prolog a „yes" válasszal tért vissza. A fenti kérdést (`member(c,[a,b,c,d]).`) a Prolog próbálja illeszteni a tényre (`member( X, [X| _]).`), de mivel a lista első (head) eleme nem egyezik meg a predikátum első argumentumában megadott értékkel, nem teljesül az illeszkedés. A következő lehetőség a szabály. A szabály szerint a lista farok részét (annak első elemét) próbáljuk ismét illeszteni ugyanahhoz az X keresendő elemhez. Ezáltal a lista nem illeszkedő első elemét mostantól nem vizsgáljuk, mert a rekurziós hívás második argumentuma az előző lépésben használt lista első elemét már nem tartalmazza. Kövessük végig a fenti példát és azt, hogy a Prolog milyen lépéseket hajt végre:

```
member(c,[ a, b,c,d]).
```

fejrész farokrész

Mivel a lista fej része nem egyezik meg az első argumentum értékével, ezért a tényre nem illeszkedik ez a kérdés.

A rekurzióban a lista farok részét próbáljuk illeszteni. Ebben a példában a T értéke: T_1=[b,c,d]. Ezzel a listával próbálja illeszteni a tényt, ugyanazzal az X-el (X=c). Az eredeti problémát a következőre vezettük vissza:

```
member(c,[ b, c,d]).
```

fejrész farokrész

Ugyanúgy mint a legelső illesztéskor a b sem illeszkedik a c-re, így a szintén 2 részre bontott lista farok részét próbáljuk illeszteni. A második rekurziós lépében a T értéke: T_2=[c,d]. Ezután a következő illesztéssel próbálkozik a Prolog:

```
member(c,[c,d]).
```

Ez viszont már illeszkedik a tényre, melynek hatására a „yes" választ kapjuk. Az illeszkedést magunk is ellenőrizhetjük, kérdezzük meg a Prologtól, hogy a két kifejezés illeszkedik-e:

```
member(c,[c,d]) = member(X,[X|_]).
```

Az illeszkedés az `X=c` esetében létrejön, melyet a Prolog megad visszatérési értékként.

További listával kapcsolatos feladatok

select – egy elem törlése listából

Hozzunk létre egy `select` predikátumot, amely az első paraméterében szereplő elemet kiveszi a második paraméterében levő listából, majd a harmadik paraméterbe megadott változó értékeként visszatér az eggyel csökkentett elemszámú listával.

```
select(mit, honnan, maradék lista).
```

Ismét egy tényünk lesz és egy szabályunk. A tény ellenőrizze azt, hogy a keresendő elem az a lista fej eleme-e. Ha az, a harmadik paraméterben térjünk vissza a lista farok részével. Amennyiben a fej elem nem egyezik meg a keresendő elemmel, akkor a fej elem legyen a kimeneti lista fej eleme (mert csak az egyező elemet szeretnénk kivenni a listából) és rekurzióval a farok részt próbáljuk illeszteni a tényhez (ugyanazt az elemet keresve). az utóbbi egy szabály lesz. Ennek megfelelően a `select` predikátum definíciója a következőképpen néz ki:

```
select(E, [E|L], L).
select(E, [A|L], [A|L1]) :- select(E,L,L1).
```

Példa:

```
select(a,[c,a,b,d],X).
```

A tényre nyilvánvalóan nem illeszkedik, mert a [c,a,b] lista fej eleme (c) nem egyezik meg az első paraméter értékével (a). A szabályt követve ezt kapjuk:

```
select(a, [ c, a,b,d], [ c, L1] ) :- select(a, [a,b,d], L1).
```

Nem ez a keresett elem ezért a kimeneti listában szerepelnie kell, ezért tesszük a harmadik lista első helyére.

A lista farok részét továbbkell vizsgáljuk, ezért szerepel a „rekurziós illesztés” második paramétereként.

A kimeneti lista farok része megegyezik a „rekurziós illesztés” eredményeként kapott listával.

A rekurziós illesztés során – mivel már a tény illeszkedni fog – az L1 listába a (rekurzióban szereplő) lista farokrésze kerül. Csak a farokrész, mert az illeszkedő elemet ki szeretnénk venni a listából. A farokrész ebben a példában egy a [b,d] lista. Így a Prolog a következővel tér vissza:

```
X = [c, b, d]
```

Hasonló logikával hozzuk létre a következő predikátumokat: (1 tény és 1 szabály)

append – Listák összefűzése

```
append(<lista1>,<lista2>,<kimeneti lista>).
```

A lista1-et fűzze a lista2-höz, majd az összefűzött lista kerüljön a harmadik paraméterben érkező szabad változóba. Megoldás:

```
append([] ,L ,L).
append([X|L1], L2, [X|L3]) :- append(L1,L2,L3).
```

A tény egyszerű: ha az első argumentum egy üres lista, akkor az összefűzött lista megegyezik a második argumentum listájával.

A szabály az első listát szedi szét fej- és farokrészre, a fej részét a kimeneti listába teszi, a farokrészt és a második listát pedig rekurzióval fűzi össze.
Példa:

```
append([a,b,c], [d,e,f], X).
```

Ennek eredménye:

```
X = [a, b, c, d, e, f]
```

<u>`rev` – Lista megfordítása</u>

```
rev(lista1,<kimeneti lista>).
```

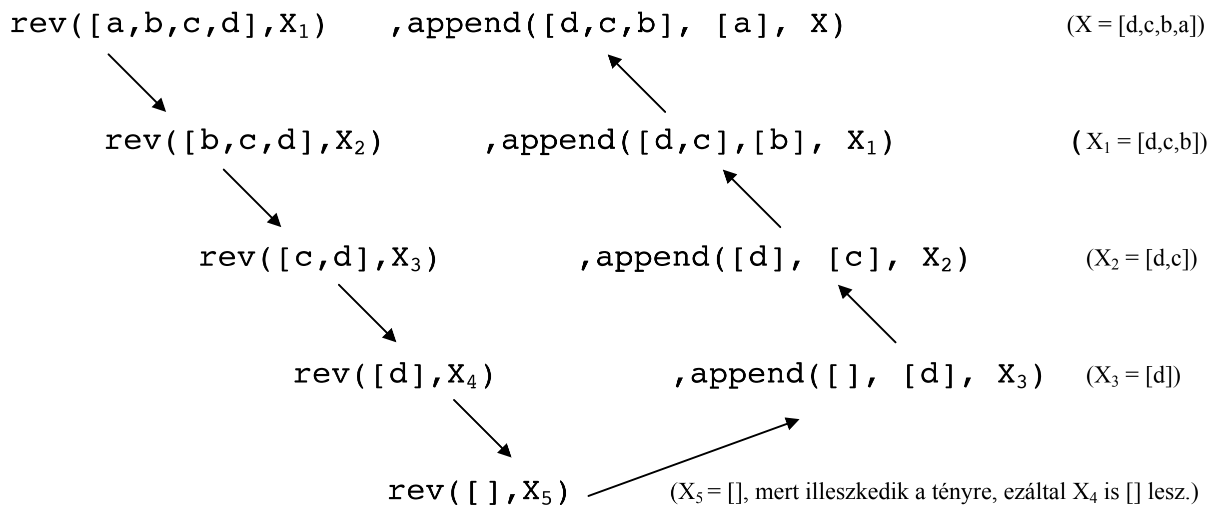
A második paraméter egy szabad változó, mely az első paraméterben érkező lista megfordítottját tartalmazza. Megoldás:

```
rev([],[]).
rev([X|L],R) :- rev(L,R1), append(R1, [X],R).
```

A `rev` felhasználja a már megírt `append` predikátumot. A tény szintén egyértelmű: üres lista megfordítottja egy üres lista. Egy példán kövessük végig a rekurziót:

```
rev([a,b,c,d],X).
```

Az input lista nem üres, így a Prolog a szabály fogja illeszteni, amelyben a rekurzió a következő lépésekkel (a nyilazás mentén) követhető nyomon:



A legutoljára végrehajtott `append` hatására az X-be létrejön a megfordított lista. Példa:

```
rev([aa,bb,cc,dd],X).
```

Ennek eredménye:

```
X = [dd, cc, bb, aa]
```

<u>`kiir` – Listaelemek kiírása</u>

```
kiir(lista).
```

Az input lista elemeit vesszővel elválasztva írjuk ki. Megoldás:

```
kiir([]).
kiir([E]) :- write(E).
kiir([E|L]) :- write(E), write(','), kiir(L).
```

A tény elhagyható, nélküle is kiírásra kerülnének a listaelemek. Ha kihagyjuk, akkor a Prolog a listaelemek kiírása mellett a „no”-t is kiírná, mivel a rekurzió során előbb-utóbb üres listát illeszt a `kiir` predikátum paraméteréhez. Ha nem definiálunk ilyet, akkor nincs mihez illessze, ezért „no”-t ír ki. A MÁSODIK szabály egy rekurzív szabály, mely kiírja a lista fej elemét, egy vesszőt, majd ezt megismétli a farokrészre. Ennek hatására viszont az utolsó listaelem kiírása után is vessző lenne, amit az első szabállyal küszöbölhetünk ki. Erre a szabályra csak akkor teljesül az illeszkedés, ha a `kiir` predikátum paraméterében levő lista csak 1 elemű. A szabály értelmében csak ezt az elemet írjuk ki.

Példa:

```
kiir([1,2,3]).
```

Ennek eredménye:

```
1,2,3
```

<u>`len` – Lista hosszának kiírása</u>

```
len(lista,<szabad változó>).
```

Megszámolja, hogy egy lista hány elemet tartalmaz, majd ezt értékül adja a szabad változónak. Megoldás:

```
len([],0).
len([_|L],N) :- len(L,N1), N is N1+1.
```

Itt feltétlen szükség van a tényre is, ugyanis a rekurzió során eljutunk addig, hogy egy üres lista lesz a `len` paramétere. Az üres lista hossza 0, így a visszalépés során van mit inkrementáljon a szabályban szereplő `N is N1+1` kifejezés. Hasonló módon követhető végig, mint ahogy azt a `rev` esetében tettük.

Példa:

```
len([a,b,c,d],X).
```

Ennek eredménye:

```
X = 4
```

Nyomkövetés

Kérhetjük a Prologtól, hogy minden egyes lépését írja ki. Ezzel könnyen tudunk hibát keresni vagy végigkövetni egy bonyolultabb rekurziót. Ezt a predikátum előtt beírt „`trace,`" utasítással tehetjük meg. Kövessük végig a `len` rekurzióját nyomkövetéssel:

```
trace,len([a,b,c,d],X).
```

Minden egyes lépésben megáll a Prolog, ENTER hatására lép tovább. A nyomkövetés eredménye:

```
Call: (9) len([a, b, c, d], _G319) ? creep
Call: (10) len([b, c, d], _L174) ? creep
Call: (11) len([c, d], _L193) ? creep
Call: (12) len([d], _L212) ? creep
Call: (13) len([], _L231) ? creep
Exit: (13) len([], 0) ? creep
^Call: (13) _L212 is 0+1 ? creep
^Exit: (13) 1 is 0+1 ? creep
Exit: (12) len([d], 1) ? creep
^Call: (12) _L193 is 1+1 ? creep
^Exit: (12) 2 is 1+1 ? creep
Exit: (11) len([c, d], 2) ? creep
^Call: (11) _L174 is 2+1 ? creep
^Exit: (11) 3 is 2+1 ? creep
Exit: (10) len([b, c, d], 3) ? creep
^Call: (10) _G319 is 3+1 ? creep
^Exit: (10) 4 is 3+1 ? creep
Exit: (9) len([a, b, c, d], 4) ? creep
```

Itt jut el a rekurzió az üres listáig, aminek hossza 0. Az innen való visszalépések száma határozza meg hányszor növeljük az üres lista hosszát, azaz a 0-t.

az 1 elemű [d] lista hossza: 1

Az utolsó visszalépés után a szabad változót 4-szer inkrementáltuk, így 4 lesz annak értéke

```
X = 4
```

Feladat: írjunk egy `sum` predikátumot, amely második paraméterében megadott szabad változóba összegzi a lista elemeit.

Párhuzamos programozás - Occam

Egyéb segédanyagok Occam-hoz

/pub/ProgramNyelvek/occam

Occam fordító

KroC, csak Linux-hoz. Letölthető a pub-ból.

A KroC használata kabinetes termekben

Ahhoz, hogy a KroC-ot megtalálja a Linux, módosítani kell a `.bashrc` (ha bash-t használsz) állományt a home könyvtárban. Az alábbi sorral kell kiegészíteni (mindegy hova):

```
export PATH=$PATH:/usr/local/kroc-1.3.2/bin
```

A módosítás csak akkor lép érvénybe, ha újraindítjuk a terminálablakot.

Fordítás:

`kroc –d` <filenév> Pl: `kroc –d pelda.occ`

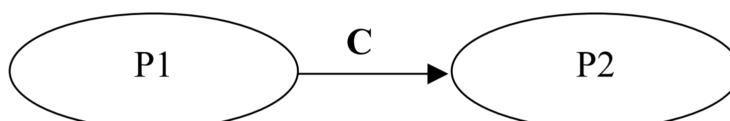
Hibamentes fordítás esetén egy futtatható állomány készül, melyett így futtathatunk:

```
./pelda
```

Párhuzamos programozás

Az Occam egy párhuzamos programozási nyelv. Ezen paradigma szerint az egyes folyamatok párhuzamosan futnak. Ez több processzoros gépek esetén valós párhuzamosságot jelent (egy processzor egy folyamatot dolgoz fel), de egy processzor esetén ez nyilván nem valósulhat meg, az egyes folyamatok „időszeleteket” kapnak, az Occam a párhuzamosságot időosztással szimulálja.

Az egyes folyamatok közötti kommunikáció csatornákon keresztül valósul meg. A P1 és P2 folyamatok a C csatornán keresztül kommunikálnak:



A folyamatok közötti kommunikációt mindig csatornákkal valósítjuk meg. A fenti példában a P1 folyamat a C csatornán keresztül valamilyen adatot küld a P2 folyamatnak. Ez a következőképpen valósul meg: ha egy folyamat elérkezik arra a pontra, ahol értéket küld [fogad], várakozik a másik folyamatra, amíg az is el nem ér a fogad [küld] pontra. Amikor mindketten készen állnak az adatcserére (azaz mindkét folyamatban a küldés [fogadás] pontra került a vezérlés) létrejön az adatcsere, majd mindkettő folytatja a futását.

Hasznos tudnivalók a nyelvről

- Minden, a nyelvben lefoglalt kulcsszót nagy betűvel kell írni (`SEQ`, `PAR`, `PROC`, stb…)
- A blokkstruktúrát indentációval jelöljük (két szóközzel beljebb kezdjük)
- Minden egyes kifejezés új sorban kezdődik (esetlegesen két szóközzel beljebb)
- Egy soros megjegyzés: `-- ez egy megjegyzés`

Egy Occam program a következőképpen épül fel:

<deklarációk>
<folyamat>

Például:

```
INT x:
SEQ
  x := 10
  x := x + 1
```

Az Occam elemi folyamatai

5 elemi folyamatot különböztetünk meg:

Megnevezés	Szintaxis	Példa
Értékadás	<változó> := kifejezés	`k := k + 10`
Küldés	<csatorna> ! <kifejezés>	`C ! k + 5`
Fogadás	<csatorna> ? <változó>	`C ? x`
SKIP	SKIP	SKIP
STOP	STOP	STOP

A fenti példában, küldés esetében egy kifejezést (`k + 5`) küldünk a C csatornára, fogadás esetén pedig a C csatornáról várunk egy értéket, amely az x változóban kerül.

A `SKIP` folyamat a legegyszerűbb elemi folyamat, „semmit nem csinál". Haszontalannak tűnhet, de összetettebb programok esetében (például még nem kifejlesztett programrészek esetében) hasznos lehet. Párhuzamos folyamatok esetében fontos, hogy minden folyamat termináljon, ellenkező esetben az egész, folyamatokból álló „rendszer" leáll. A `STOP` szintén „nem csinál semmit", de ez sosem terminál – ellentétben a `SKIP`-el. Egy folyamatban a `STOP` (feltéve hogy a vezérlés odakerül), annak holtpontba jutását eredményezi. Szintén haszontalannak tűnhet, de ezzel egy folyamatot leállíthatunk más folyamatok működésének befolyásolása nélkül, ami hibakeresésnél hasznos lehet.

Azt mondjuk, hogy egy folyamat holtpont állapotba került, ha az már nem képes további működésre (vezérlése leáll), és ez a leállás nem a folyamat helyes lefutásának eredménye. Párhuzamos folyamatok közül akár egy folyamat holtpont állapotba kerülése az egész program holtpont állapotba kerülését eredményezi, hiszen az összes többi folyamat várja a holtpontban levő folyamat terminálását, ami sosem fog bekövetkezni.

A kifejezésekben, operátorok között precedenciát nem határozunk meg, így MINDIG zárójelezést kell használni a precedencia meghatározásához. A következő kifejezés például helytelen:

```
k := k * 5 + 4
```

Ennek helyes változata:

```
k := (k * 5) + 4
```

Küldés [fogadás] esetén használhatjuk a ;-t több kifejezés küldése [fogadása] esetén. Például:

```
C ! m; n; m + n
```

Ezesetben a C csatorna protokollja egy `(Int; Int; Int)` szekvenciális protokoll lesz.

Az Occam adattípusai

Az Occam – akárcsak a Pascal – egy erősen típusos nyelv, minden változót deklarálni kell, melynek általános alakja:

<típus> <azonosítók vesszővel elválasztva>**:**

Például két egész típusú x és y változó deklarációja így néz ki:

```
INT x, y:
```

A típusokat a következő táblázat foglalja össze:

BOOL	Logikai típus, lehet `TRUE` vagy `FALSE`
BYTE	8-bites, nem előjeles egész érték (mint a `char` a C-ben)
INT	általában 32 vagy 64-bites egész érték (mint az `int` a C-ben)
INT16	16-bites előjeles egész
INT32	32-bites előjeles egész
INT64	64-bites előjeles egész
REAL32	32-bites valós érték
REAL64	64-bites valós érték

Csatorna deklarálásának általános alakja:

CHAN [OF] <típus> <azonosító>**:**

Az `OF` elhagyható. Például egy egész értékeket továbbító c csatorna deklarációja a következő:

CHAN OF INT c: vagy **CHAN INT c:**

Az azonosítók deklarációi bárhol elhelyezhetők a kódban, nem kell a program legelején legyenek.

Mivel az Occam nyelvet biztonságos párhuzamos programozásra tervezték, ezért a pointerek használata nem engedélyezett.

Tömbök

A tömb az egyetlen rendelkezésre álló adatszerkezet, amely – részben az Occam „nem-dinamikus” volta miatt – csak előre rögzített méretű lehet. Az Occam-ban a dimenziót a típus előtt kell megadjuk. Néhány példa:

[5]INT egeszek: (egy 5 elemű, egészeket tároló vektor)
[10][10]REAL64 matrix: (egy 2 dimenziós, 10x10-es mátrix)

A tömbelemek indexelése – akárcsak a C-ben – 0-val kezdődik.

SEQ

A `SEQ` (SEQuential) blokkjában definiált folyamatok szekvenciálisan kerülnek végrehajtásra. Például a következő elemi folyamatok szekvenciálisan, egymás után hajtódnak végre:

```
SEQ
  x := 10
  y := x + 1
  z := x – 1
```

Vegyük észre, hogy mindhárom folyamat 2 szóközzel beljebb kezdődik, amivel a blokkstruktúrát határozzuk meg. Ez nem egy lehetőség a könnyebb olvashatóságra, ezt kötelezően így kell írjuk!

PAR

A PAR (PARallel) blokkjában definiált folyamatok párhuzamosan kerülnek végrehajtásra. Definiáljunk két párhuzamosan működő elemi folyamatot:

```
PAR
  INT m:
  c1 ? m -- adatfogadás a c1 csatornáról
  INT n:
  c2 ? n -- adatfogadás a c2 csatornáról
```

Bármennyi folyamat (független attól, hogy azok elemi vagy nem elemi folyamatok) futtatható párhuzamosan. Az egész PAR blokk akkor terminál, ha a benne „elindított” folyamatok mindegyike terminál.

Vezérlési szerkezetek

Feltételes vezérlés – IF

Az IF szerkezettel feltételhez köthetjük egy folyamat kiválasztását, ami más nyelvek esetkiválasztásos vezérléséhez hasonlítható:

```
IF
  <logikai kifejezés 1>
    <folyamat 1>
  <logikai kifejezés 2>
    <folyamat 2>
  <logikai kifejezés 3>
    <folyamat 3>
```

Az a folyamat kerül kiválasztásra, mely feletti logikai kifejezés értéke igaz. Ha egyik logikai értéke sem igaz, akkor az IF szerkezet STOP-ként funkcionál, ezért ezt mindig kerüljük. A STOP elkerülésére két lehetőség is adódik. Az egyik lehetőség, ha a logikai vizsgálatok minden lehetséges értéket lefednek, így valamelyik feltétel mindig teljesül, pl:

```
IF
  x > y
    a := 1
  x < y
    a := 2
  x = y
    a := 3
```

Nem tudunk úgy értéket adni x-nek és y-nak, hogy a fenti feltételek közül egyik se teljesüljön.

A másik lehetőség, ha a feltételek halmazába felvesszük a `TRUE` értéket, amelyhez a `SKIP` folyamat tartozik. A `SKIP` nem csinál semmit, de hatására a folyamat nem kerül holtpontba, pl:

```
IF
  x < 10
    a := 1
  TRUE
    SKIP
```

A `TRUE-SKIP` nélkül és x>9 teljesülése esetén az `IF` szerkezet holtpontba kerülne.

Ismétléses vezérlés – WHILE

A megszokott elöltesztelő ismétléses vezérlés Occam-beli megfelelője:

```
WHILE <logikai kifejezés>
  <folyamat>
```

Pl:

```
WHILE x<10
  x := x+1
```

Összetett folyamatok replikációja

Az összetett folyamatok (`SEQ`, `PAR`, `IF`, …) felírhatóak egy speciális, ún. replikált formában is. Ehhez egy for-ciklushoz hasonló szerkezetet fogunk használni.

Tekintsük a `SEQ` egy replikált változatát:

```
SEQ i = 1 FOR 4
  c[i-1] ! x[i-1]
```

Ennek a nem replikált megfelelője:

```
SEQ
  c[0] ! x[0]
  c[1] ! x[1]
  c[2] ! x[2]
  c[3] ! x[3]
```

az `IF` összetett folyamat egy lehetséges replikált alakja:

```
IF i=0 FOR 4
  x[i] > 5
    z := i
```

Ennek nem replikált megfelelője:

```
IF
  x[0] > 5
    z := 0
  x[1] > 5
    z := 1
  x[2] > 5
    z := 2
  x[3] > 5
    z := 3
```

PROC-ok

A `PROC` egy előre definiált, névvel ellátott folyamat. Tekinthetünk úgy rá, mintha egy eljárást definiálnánk. Egy `PROC` paraméterezése tetszőleges lehet, kivéve azt a `PROC`-ot, mely a program belépési pontja lesz, annak paraméterezése (és a paraméterek sorrendje) ugyanis kötött:

```
(standard input, standard output, standard error)
```

A standard input egy `BYTE` csatorna. Ennek megfelelően a belépési pontot meghatározó `PROC` paraméterezése így néz ki (ahol a `keyboard`, `screen` és `error` helyett tetszőleges azonosítókat írhatunk):

```
(CHAN BYTE keyboard, screen, error)
```

Egy PROC deklarálásának általános alakja:

```
PROC <azonosító>(paraméterek)
  <folyamatok>
:
```

Ha egy Occam program egyetlen `PROC` deklarációt tartalmaz, akkor azt a KroC a program fő belépési pontjának tekinti. Ha több is van, akkor egymásba ágyazás esetén a legkülső, több egyszintű `PROC` esetén pedig a legutoljára deklarált lesz a program fő belépési pontja.

A következő `PROC` egy „Hello World” Occam program. Mivel más `PROC`-ot nem is deklarálunk, ezért ez lesz a programunk belépési pontja:

```
PROC hello(CHAN BYTE keyboard, screen, error)
  VAL szoveg IS "Hello*c*n":
  SEQ i=0 FOR (SIZE szoveg)
    screen!szoveg[i]
:
```

A második sor egy „`szoveg`” azonosítójú, „Hello” sztringet tartalmazó konstans. A `*c*n` rendre a C-beli `\r\n` megfelelői. Ezt követi egy replikált `SEQ`, mely 0-tól a szöveg konstans hosszáig megy. Ez a blokkjában levő folyamatok szekvenciális végrehajtását írja elő. A `SEQ` blokkja a `screen` csatornára (a képernyőre) a sztringet – mint `BYTE` típusú értékeket tároló tömböt – karakterenként a képernyőre írja. Mint általában minden deklaráció, ez is `:`-al fejeződik be.

Az ALT konstrukció

Az ALT az Occam egy speciális szerkezete, más nyelvekben nem találunk ennek megfelelő szerkezetet. Egy folyamat kiválasztását *őrök* segítségével végzi, melyek lehetnek tiltottak vagy engedélyezettek. Az ALT minden egyes csatornát figyel és a hozzá tartozó őr engedélyezetté válik, ha az általa figyelt csatornára egy folyamat készen áll adat küldésére. Az őr, melynek csatornájára nem érkezik adat, tiltott. Egy őr általánosan a követezőképpen néz ki:

<feltétel> **&** <csatorna> **?** <változó>

A <feltétel> elhagyható, ha nem írjuk ki, akkor a feltétel TRUE-nak felel meg. Az ALT csak azokat az őröket veszi figyelembe, mely előtt a feltétel TRUE.

Az ALT általános alakja:

```
ALT
  <őr 1>
    <folyamat 1>
  <őr 2>
    <folyamat 2>
...
  <őr n>
    <folyamat n>
```

Ha egy őr engedélyezetté válik, akkor a benne megadott változó felveszi a csatornáról érkező adat értékét és „elindítja” a hozzá tartozó folyamatot.

Példa ALT-ra:

```
ALT
  c1 ? x
    y := y+x
  c2 ? x
    z := z+x
```

Az x változó értéke attól függ, hogy c1-re vagy c2-re érkezik előbb adat.

Mivel a program írásakor nem tudhatjuk, hogy melyik csatornáról fog adat érkezni, ezért az ALT-ot tartalmazó programok nemdeterminisztikusak. A holtpont veszélye sincs kizárva, mert azt sem tudhatjuk előre, hogy a felsorolt csatornákra egyáltalán fog-e érkezni adat.

Az Occam-ban megismert szerkezetek C-beli megfelelői

Az alábbi táblázat foglalja össze az Occam szerkezeteit és hasonlításképpen azok C-beli megfelelőit. Az ALT és a PAR szerkezeteknek érthető okok miatt nincsenek C-beli megfelelői.

Szerkezet	Occam példa	C példa

Feltételes vezérlés	`IF`
	`  x = y`
	`    eljaras1(x)`
	`  y = 0`
	`    eljaras2(x)`
	`  TRUE`
	`    SKIP`
		`if (x == y){`
		`    eljaras (x);`
		`} else if (y == 0){`
		`    eljaras2 (x); } else {`
		`    /* SEMMI */`
		`}`

Feltételes vezérlés TRUE-őr nélkül	`IF`
	`  FALSE`
	`    SKIP`
		`if (0){`
		`    /* SEMMI */`
		`} else{`
		`    /* !HIBA generálása! */`
		`    *(byte)5 = 10;`
		`}`

While ciklus	`WHILE (NOT end.of.file)`
	`  folyamat`
		`while (!end_of_file){`
		`  folyamat`
		`}`

Eljárás	`PROC f (VAL INT x, REAL64 r)`
	`  folyamat`
	`:`
		`void f (int x, double *r){`
		`  folyamat`
		`}`

Függvény	`INT FUNCTION f (VAL INT v)`
	`  INT r:`
	`  VALOF`
	`    r := (v * 10)`
	`    RESULT r`
	`:`
		`int f (int v){`
		`  int r;`
		`  r = (v * 10);`
		`  return r;`
		`}`

Esetkiválasztás	`CASE array[i]`
	`  'a','b','c','d','e'`
	`    ch := array[i]`
	`  'f','g'`
	`    ch := 'z'`
	`  ELSE`
	`    ch := #00`
		`switch (array[i]){`
		`case 'a': case 'b':`
		`case 'c': case 'd':`
		`case 'e':`
		`  ch = array[i];`
		`  break;`
		`case 'f': case 'g':`
		`  ch = 'z';`
		`  break;`
		`default:`
		`  ch = 0;`
		`  break;`
		`}`

For-ciklus	`SEQ i = 0 FOR count`
	`  P (i)`
		`  for (int i = 0; i < count; i++){`
		`    P (i);`
		`  }`

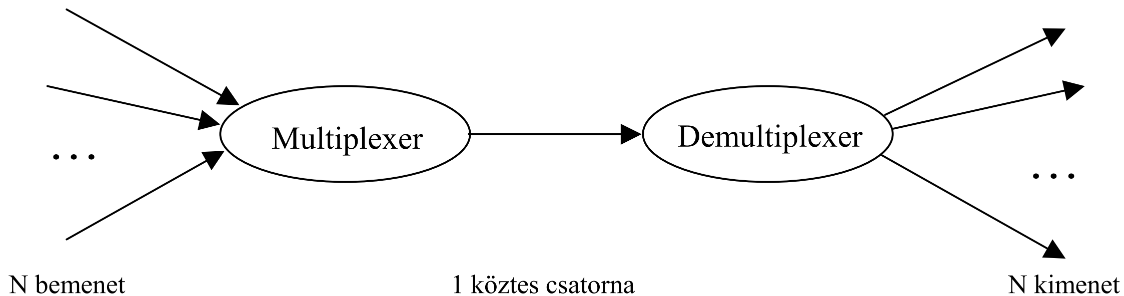
Egyszerű típus létrehozása	`DATA TYPE tipus IS INT:`	`typedef int tipus;`

Összetett típus létrehozása	`DATA TYPE rekordtip`
	`  RECORD`
	`    INT x, y:`
	`    REAL64 i, j:`
	`    [16]BYTE string:`
	`:`
		`typedef struct{`
		`  int x, y;`
		`  double i, j;`
		`  char string[16];`
		`} rekordtip;`

A táblázat alsó két sorában található szerkezetek csak az Occam 2.1-es verziójától használhatók!

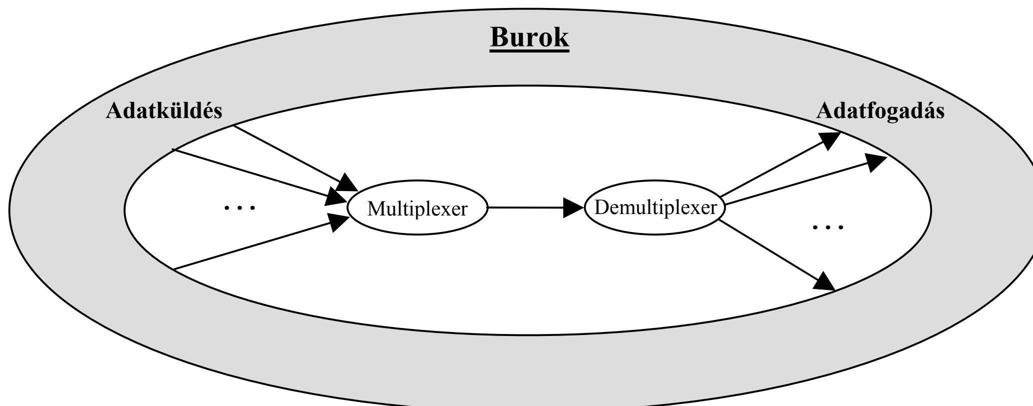
Egy összetettebb példa – Multiplexer

A multiplexer lehetővé teszi több forrásból érkező adatok egyetlen csatornán keresztüli továbbítását. Ahhoz hogy a továbbított adatot az ugyanannyi kimenettel rendelkező demultiplexer a megfelelő csatornáján továbbítani tudja, tudnia kell, hogy a multiplexer hányadik csatornájára érkezett az adat. Ezért a multiplexer nem csak az érkező adatot kell továbbítsa, hanem azt is, hogy az melyik csatornáról érkezett.



Ennek megfelelően ha például a multiplexer második csatornáján érkezik egy adat, akkor a kimeneti csatornán először továbbítjuk a csatorna számát, majd közvetlen utána az adatot is. A demultiplexer ilyen sorrendben fogadja az információkat, majd a megfelelő csatornájára irányítja az adatot.

Az Occam program a következő struktúrájú lesz. Létrehozunk egy „burkot”, mely felelős lesz a multiplexernek való adatküldésért és a demultiplexertől való adatfogadásért és a felhasználó tájékoztatásáért.



A forráskód felépítése:

```
PROC plexer(CHAN BYTE keyboard, screen, error)
  PROC multi([]CHAN INT inp, CHAN INT out)
    ...
  :
  PROC demulti(CHAN INT inp, []CHAN INT out)
    ...
  :
```

```
    PROC burok(VAL INT csatkuld,
               VAL INT ertekkuld,
               []CHAN INT multi,
               []CHAN INT demulti)
    :
    főprogram
  :
```

A tényleges forráskódban ezt a 4 sort 1 sorba kell írni, ide 1 sorba nem fért ki…

A külső PROC (plexer) lesz a program belépési pontja, mivel az a legkijjebb deklarált PROC, így ennek paraméterezése kötött. A multiplexert egy `multi`, a demultiplexert egy `demulti` nevű PROC-ban valósítjuk meg. A `burok` PROC első 3 paraméterét a multiplexernek való adatküldéshez, utolsó paraméterét pedig a demultiplexertől való adatfogadáshoz fogjuk felhasználni. Az első paraméterében adjuk meg, hogy a multiplexer hányadik csatornájára szeretnénk elküldeni a második paraméterben érkező értéket. Harmadik paramétere a multiplexer inputja, csatornák tömbje. A negyedik paraméter pedig a demultiplexer kimeneti csatornája, melyet ez a PROC fog figyelni.

A főprogramban deklaráljuk a megfelelő változókat, majd meghívjuk a `burok` PROC-ot a megfelelő paraméterezéssel. Ezzel párhuzamosan „elindítjuk" a multiplexert és a demultiplexert.

A PROC-ok megvalósításai:

A `multi` PROC megvalósítása

```
PROC multi([]CHAN INT inp, CHAN INT out)
  WHILE TRUE
    ALT csat=0 FOR SIZE inp
      INT ertek:
      inp[csat] ? ertek
        SEQ
          out ! csat
          out ! ertek
:
```

Az ALT egyik tipikus felhasználása a multiplexer megvalósítása, az eljárás magját a benne definiált replikált ALT konstrukció képezi. Ez figyeli a multiplexer összes csatornáját és adat érkezésekor a csatorna számát és az adatot a kimeneti csatornára küldi. A replikációnak köszönhetően a `csat` változóban kapjuk meg annak a csatornánk a számát, melyről az adat érkezett. A multiplexer bemeneti csatornáinak számát konstansként fogjuk a főprogramban megadni, itt ezt a `SIZE` segítségével tudhatjuk meg. Az `out.string` és `out.int` eljárások előre definiáltak, a program legelején töltjük be őket. (az `#INCLUDE "qwer.inc"` által) Mivel egy multiplexer folyamatosan figyeli az input csatornáit, ezért ezt az ALT-ot egy végtelen ciklusba helyezzük. Ezzel elérjük azt, hogy bármikor, bármelyik csatornára érkező adat továbbításra fog kerülni.

A demultiplexer az input csatornájáról (ami megegyezik a multiplexer output csatornájával) csatornaszám-adat sorrendben elolvassa az értékeket, majd ezek alapján a megfelelő csatornára írja az adatot.

A demulti PROC megvalósítása:

```
PROC demulti(CHAN INT inp, []CHAN INT out)
  WHILE TRUE
    INT csat, ertek:
    SEQ
      inp ? csat
      inp ? ertek
      out[csat] ! ertek
:
```

A demultiplexer is folyamatosan figyeli az inputot, ezért itt is végtelen ciklust használunk. Az input csatornáról először a csatorna számát olvassuk el, mert a multiplexer ezt küldi elsőként, majd ezt követi az adat fogadása. Ezután az adatot a megfelelő kimeneti csatornára küldjük. A demultiplexer kimeneti csatornáit a burok figyeli és ha az egyiken adat érkezik, fogadja.

A burok PROC megvalósítása:

```
PROC burok(VAL INT csatkuld,VAL INT ertekkuld,[]CHAN INT multi,[]CHAN INT demulti)
  SEQ
    multi[csatkuld] ! ertekkuld
    out.string(screen,"*c*n----*c*nAdat elkuldve a multiplexernek*c*nCsatorna: ")
    out.int(screen,csatkuld)
 1  out.string(screen,"*c*nErtek: ")
    out.int(screen,ertekkuld)
    out.string(screen,"*c*n")

    INT csatfogad:
    INT ertekfogad:
    ALT csatfogad=0 FOR SIZE multi
      demulti[csatfogad] ? ertekfogad
        SEQ
 2        out.string(screen,"*c*nAdat erkezett a demultiplexertol*c*nCsatorna: ")
          out.int(screen,csatfogad)
          out.string(screen,"*c*nErtek: ")
          out.int(screen,ertekfogad)
          out.string(screen,"*c*n")
:
```

A burok elsődleges célja a multiplexernek való adatküldés, a demultiplexertől való adatfogadás és a csatornaműveletek képernyőre írása. De más okból is szükség volt rá. Egy erőforrást két párhuzamosan futó folyamat egyszerre nem használhat, ezért a képernyőre írást a multiplexer és a demultiplexer egyike sem hajthatja végre.

Az **1**-el jelölt rész végzi el a demultiplexernek való adatküldést és a küldéssel kapcsolatos információk (a burok által küldött csatornaszám és adat) képernyőre írását.

A **2**-vel jelölt rész figyeli a demultiplexer összes kimeneti csatornáját és adat érkezésekor fogadja azt, majd az ezzel kapcsolatos információkat (a multiplexer által küldött csatornaszámot és adatot) a képernyőre írja. A csatornák figyeléséhez ugyanolyan ALT konstrukciót használunk, mint amilyet a multiplexer esetében is használtunk.

A főprogram:

```
VAL N IS 10:
[N]CHAN INT multiinp,demultiout:
CHAN INT multiout:

PAR
  SEQ
    burok(5,10,multiinp,demultiout)
    burok(3,20,multiinp,demultiout)

  multi(multiinp,multiout)
  demulti(multiout,demultiout)
:
```

Az `N` konstansban rögzítjük, hogy a multiplexer hány inputtal (és a demultiplexer hány outputtal) rendelkezzen, majd deklaráljuk a szükséges csatornákat. A `PAR` blokkjában (párhuzamosan) elindítjuk a multiplexert és a demultiplexert. A burkon keresztül szekvenciálisan, de az előzőekkel párhuzamosan 2 adatot küldünk a multiplexernek. Először a multiplexer ötödik csatornájára elküldjük a 10-et, majd a harmadikra a 20-at.

A program lefutásának eredménye:

```
Adat kuldese a multiplexernek
-----------
Csatorna: 5
Ertek: 10
-----------
Adat erkezett a demultiplexertol
Csatorna: 5
Ertek: 10

Adat kuldese a multiplexernek
-----------
Csatorna: 3
Ertek: 20
-----------
Adat erkezett a demultiplexertol
Csatorna: 3
Ertek: 20
```

A program teljes forráskódja letölthető: plexer.occ