

Programozási nyelvek előadás

*Dr. Kertész Attila
Szoftverfejlesztés Tanszék
Szegedi Tudományegyetem*

Elérhetőségek

- Email: *keratt@inf.u-szeged.hu*
- Web: <http://www.inf.u-szeged.hu/~keratt/>

Követelmények

- CooSpace

Az előadások látogatása nem kötelező.

Előfeltétel: Programozás I.

Az értékelés a vizsgaidőszakra meghirdetett időpontban megírt vizsgadolgozat alapján történik.

A dolgozatra maximum 100 pont kapható, elérendő minimum 40 pont.

Az érdemjegy az alábbiak alapján kerül megállapításra:

[0-39]: elégtelen

[40-54] : elégséges

[55-69] : közepes

[70-84] : jó

[85-100] : jeles

Tematika

Programozási nyelvek fejlődése, csoportosítása, általános tulajdonságai

Információ elrejtés, modul, absztrakt adattípus

Objektumorientált programozás, öröklődés

Smalltalk programozási nyelv

Érték és típus, típusképzés, kifejezés, változó, adattárolás, utasítás

Funkcionális programozás, Paraméteres típus, típuskövetkeztetés

Haskell programozási nyelv

Deklaráció, hatáskör, statikus és dinamikus hozzárendelés

Logikai programozás, Prolog programozási nyelv

Vezérlés, kivételkezelés, absztrakció, paraméterátadás, kiértékelés

Párhuzamosság, folyamat, interakció, holtpont

Occam programozási nyelv

Segédanyagok, irodalom

- Nyékyéné G. Judit (szerk): Programozási nyelvek, Kiskapu, 2003
- D. A. Watt: Programming Language Concepts and Paradigms, Prentice Hall, 1990
- T.W. Pratt, M. V. Zelkovitz: Programming Languages, Prentice Hall, 2001
- R. W. Sebesta: Concepts of Programming Languages, Addison-Wesley, 2002

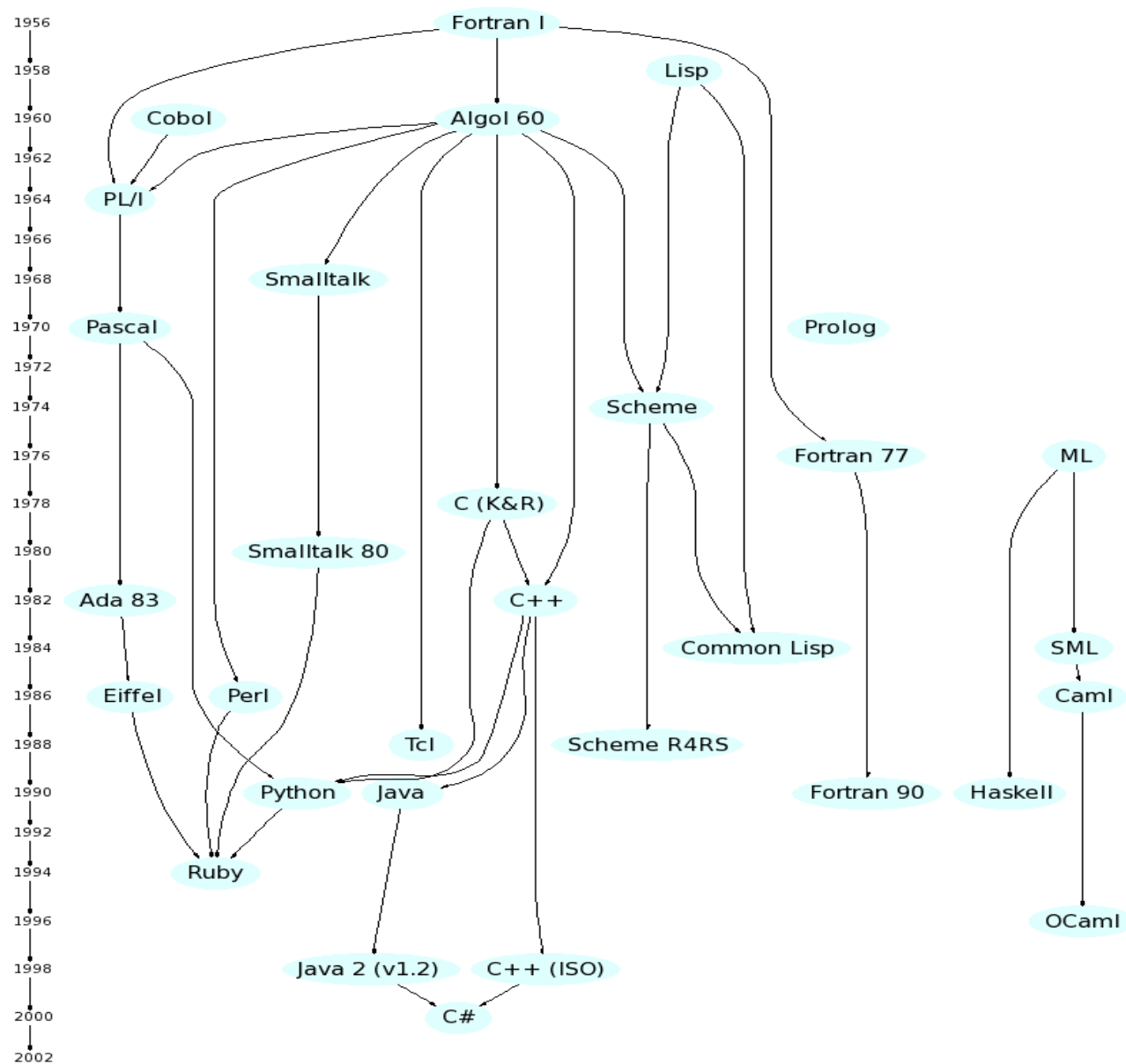
Programozási nyelv

Jelölésrendszer számítógépen
megvalósítható algoritmusok és
adatszerkezetek leírására

Miért (tanuljuk ezt)?

- Ismert nyelvek alaposabb megismerése
- Kifejezőerő növelése, módszerek bővítése
- Megfelelő nyelv kiválasztása
- Új nyelv könnyebb elsajátítása
- Új nyelv tervezése

Történelem



Érdekesességek

- Sun Microsystems: '90 - Green projekt
- James Gosling tölgyfája → Oak → Java
- '93: projekt leállítási terv
- WWW fejlődése → dinamikus oldalak
- Böngésző-integráció → siker

Elvárások

- Világos, egyszerű, egységes
- Ortogonalitás
- Természetes kifejezés mód
- Absztrakció támogatása
- Megbízhatóság
- Programozási környezet
- Hordozhatóság
- Alacsony költségek

Nyelvcsoportok (paradigmák)

- Imperatív, procedurális → C, C++, Pascal
- Objektum orientált → C++, Java, Smalltalk
- Applikatív, funkcionális → Haskell, ML
- Szabály alapú, logikai → Prolog, HASL
- Párhuzamos → Occam, PVM, MPI

Alkalmazási területek

- Tudományos (Fortran)
- Üzleti (Cobol)
- Mesterséges intelligencia (Prolog)
- Rendszerprogramozás (C)
- Adminisztráció (Perl)
- Speciális

Implementációs stratégiák

- Értelmező (interpreter)
- Fordító (translator, compiler)
- Hibrid

Egy programozási nyelv virtuális számítógépnek is tekinthető

könyv

- 1. fejezet (1-26)

Szintaxis (könyv 2. fejezet)

- Formai szabályok
- Formális nyelvek elmélete

Szemantika

- Programok jelentése
- Nehéz formalizálni

Imperatív programozás fejlődése

- Gépi kód, assembly → :(
- Procedurális programozás → Adatok szabadon
- Moduláris programozás → Elrejtés
→ Viselkedésminta
- Absztrakt adattípusok
- Objektum orientáltság → Példányok, öröklődés
- ???

Moduláris tervezés

- Dekompozíció:
A feladatot kisebb részfeladatokra bontjuk
- Kompozíció:
Meglévő egységek újrafelhasználása
- Érthetőség: Önálló egységek
- Folytonosság: Kis változtatások
- Védelem: Hibák hatásának korlátozása

Modularitás alapelvei

- Nyelvi támogatás
- Kevés kapcsolat a modulok között
- Gyenge kapcsolatok a modulok között
- Explicit interfészek
- Információ elrejtés, láthatóság
- Nyitott és zárt modulok
- Egy típus – egy modul
- Egy modul – több típus

Információ elrejtés, bezárás *(information hiding,* *encapsulation)*

- Alapegység a modul
- Csatlakozási felület
(interface)
felhasználó csak ezt látja, exportált komponensek → Mit csinál
- Megvalósítás
(implementation) → Hogyan
felhasználó előtt rejtve,
szabadon megváltoztatható

Stack modul

```
// stack.h

void push( char);
char pop();

const int stack_size= 100;
```

```
// stack.c

#include "stack.h"

static char v[ stack_size];
static char *p= v;

void push( char c)
{
    // ...
}

char pop()
{
    // ...
}
```

```
// main.c

#include "stack.h"

void main()
{
    push( 'a');
    char c= pop();

    // ...
}
```

könyv

- 9. fejezet (314-330)

Absztrakt adattípus

- Indirekt módon, műveletei által definiált típus
- Modul + példányok (változók)
- Műveletek: konstansok, függvények, eljárások
- Konstansokból generálható értékhalmaz
- Konstruktor és szelektor műveletek

Stack (absztrakt adattípus)

```
module Stack (emptyStack, push, pop) where
```

```
type Stack t = [ t]
```

```
emptyStack :: Stack t  
emptyStack = []
```

```
push :: Stack t -> t -> Stack t  
push s t = ( t : s)
```

```
pop :: Stack t -> t  
pop ( t : s) = s
```

```
>:load stack.hs
```

```
>push ( push emptyStack 1) 2  
[2,1]
```

```
>pop push ( push emptyStack 1) 2  
2
```

könyv

- 10.1-10.5.1 (332-336)
- 10.6, 10.7, 10.8 (340-344)

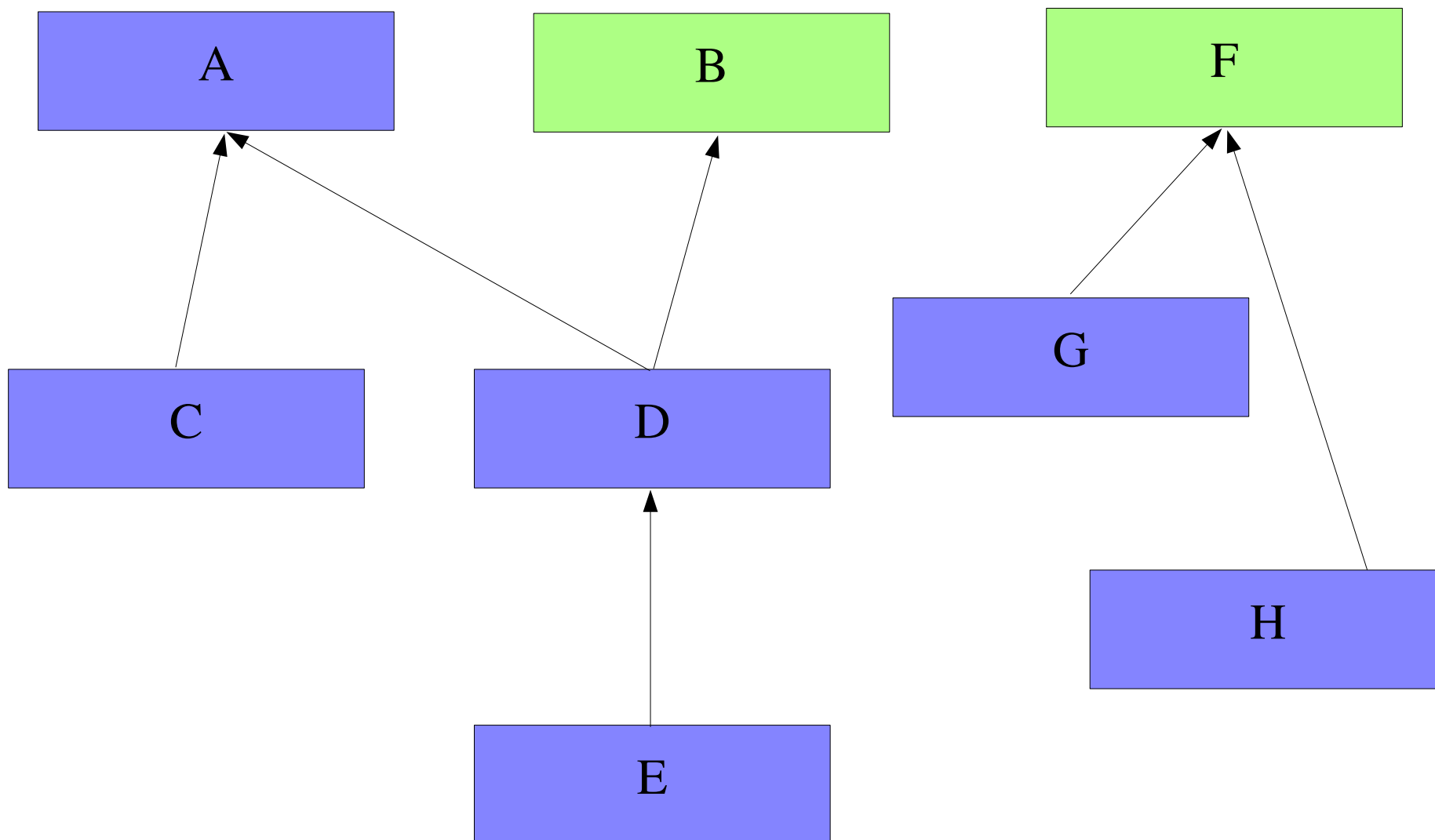
Objektum orientált programozás

- Osztályok, objektumok, metódusok (kizárólagos) használata -> szimuláció, modellezés
- Szemléletmód (analízis, tervezés, megvalósítás), ábrázolásmód
- Adatelrejtés, láthatóság
- Az objektumok elsőrendű értékek
- Öröklődés (osztályok közötti hasonlóság kifejezésére)
- Dinamikus típushozzárendelés

Öröklődés

- Osztályok közötti alárendeltségi viszony
- Szülő-gyerek, ős-leszármazott, ...
- Egyszeres, többszörös öröklődés
- Összefüggő, vagy particionált osztályhierarchia
- Absztrakt osztályok
- Polimorfizmus

Öröklődés



könyv

- 12. fejezet (390-456)
- csak a C, C++, Java, Smalltalk példák

Smalltalk

- www.smalltalk.org
- GNU Smalltalk tutorial
- /pub/ProgramNyelvek/smalltalk
- Goldberg, Robson: Smalltalk 80, The Language

Smalltalk - kifejezések

- számok, szövegek, blokkok, változók
 - Integer: 5, -2, 0
 - Karakter: \$a, \$5, \$\$, \$', Character space
 - String: 'abcd'
 - Blokk: [1. 2. 3]
- Üzenet:
'Hello' println, 3 + 5, 1 max: 5
- Kifejezés:
kiértékelés után értéket (objektum) ad vissza

Smalltalk – változók, értékadás

- Privát változók kisbetűvel kezdődnek
- Közös változók nagybetűvel kezdődnek
- Értékadás:
 $v := e$
 $v _ e$
- Többszörös értékadás
 $v := w := e$
- Értékadás a kifejezés értékét adja
 vissza

Smalltalk - speciális hivatkozások

- `nil`
definiálatlan objektum
- `true`, `false`
logikai igaz és hamis, Boolean
típusúak
- `self`, `super`
az aktuális objektumra hivatkoznak
`super` segítségével az őszosztály
metódusait lehet hívni

Smalltalk - üzenetek

- Unáris üzenetek
5 printNl, 3.14 sin, Character space
- Bináris üzenetek
3 + 5, x - 4, v > 7
- Kulcsszavas üzenetek
 - at:put:
Smalltalk at: #v put: 7
 - max:
1 max: 10
 - subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:

Smalltalk - üzenetek

- Smalltalk kulcsszavas üzenet
 - `at:put:`
`Smalltalk at: #v put: 7`
- „hagyományos” hívás
 - `Smalltalk.atput(#v, 7)`
- Különbségek
 - A sorrend egyértelmű
 - A paraméter szerepe könnyebben azonosítható

Üzenetek sorrendje

- Unáris, bináris balról jobbra
3 odd printNl -> (3 odd) printNl
3 + 5 * 2 -> (3 + 5) * 2
- Unáris után bináris
3 + 5 negated -> 3 + (5 negated)
- Bináris után kulcsszavas
3 max: 1 + 4 -> 3 max: (1 + 4)
- Kulcsszavas üzeneteket zárójelezni kell
(1 min: 2) max: 3

Smalltalk - tömbök

- Tömb literál

```
#( 1 2 3)
```

```
#( 1 'xyz' ( 2 3) $f)
```

- Legfontosabb tömb üzenetek

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

```
#( 1 2 3) copy
```

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

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

Smalltalk - üzenetek

- Forrásszöveg formátum

```
#( 1 2 3)
```

```
    at: 1
```

```
#( 1 2 3) copy
```

```
    at: 1
```

```
    put: 4
```

- Kaszkád: egy objektumnak több üzenet

```
5 + 3; + 1; + 7
```

Smalltalk - Blokk

- Blokk: később kiértékelendő kifejezések sorozatát tartalmazó objektum
x := [1. 2. 3]
y := []
- Blokk kiértékelés: value üzenet
x value
- Blokk kiértékelés után az utolsó kifejezés értékét adja vissza (üres blokk esetén nil-t)

Smalltalk – feltételes vezérlés

- Boolean objektumnak küldött blokk argumentumú üzenettel valósítható meg

```
x > 3 ifTrue: [ 'OK' printNl ]
```

- Feltételes üzenetek:
ifTrue:
ifFalse:
ifTrue:ifFalse:
ifFalse:ifTrue:

Smalltalk - ismétlés

- Rögzített számú ismétlés (for ciklus)
4 timesRepeat: ['Hello' printNl]
- Feltételes ismétlés (while ciklus)
v := #(1 2 3).
i := 1.
[i <= v size]
whileTrue:
 [(v at: i) printNl. i := i+1]

Smalltalk - blokk paraméterek

- Tömb elemeinek feldolgozása

```
sum := 0.
```

```
#(1 2 3) do: [:x|sum := sum + x].
```

```
sum printNl
```

```
#(1 2 3) collect: [:x| x + 1]
```

- Több paraméter

```
[:x :y| x + y] value: 5 value: 7
```

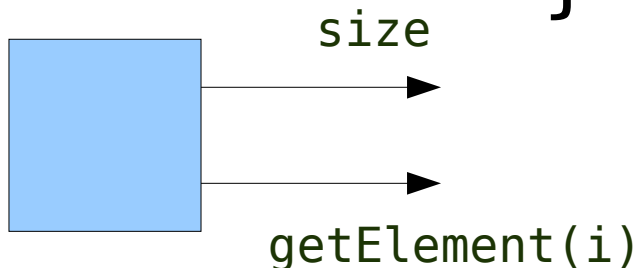
Smalltalk blokk: anonim metódus

- Metódus attribútumai
 - ✓ Bemenő paraméterek
 - ✓ Törzs (kifejezések)
 - × Név
- A blokk név nélküli (anonim) metódus
 - Aktivizálható hivatkozáson keresztül
 - Argumentumok adhatók át neki
 - Visszatérési értéke van
- A metódusok névvel ellátott blokkok

Smalltalk – kollekciók feldolgozása

- Hagyományos (procedurális) módszer
 - Vezérlés és feldolgozás egy helyen, a kollekción kívül

```
for (i=1; i<c.size; ++i) {  
    x = c.getElement(i);  
    // x feldolgozása  
}
```



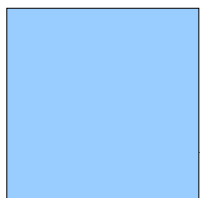
Smalltalk – kollekciók feldolgozása

- Objektumorientált módszer
 - Vezérlés a kollekcióban
 - Feldolgozás a kollekción kívül

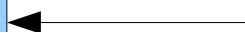
`c do: [:x| x feldolgozása]`

`...`

`c do: [:x| x feldolgozása]`



`do: [:x | ...]`



Smalltalk - Osztályhierarchia

Object

Behaviour

MetaClass

BlockClosure

Boolean

False

True

Magnitude

Character

Date

Number

Float

Fraction

Integer

Object

Collection

Bag

Dictionary

Set

Array

String

Stream

PositionableStream

ReadStream

WriteStream

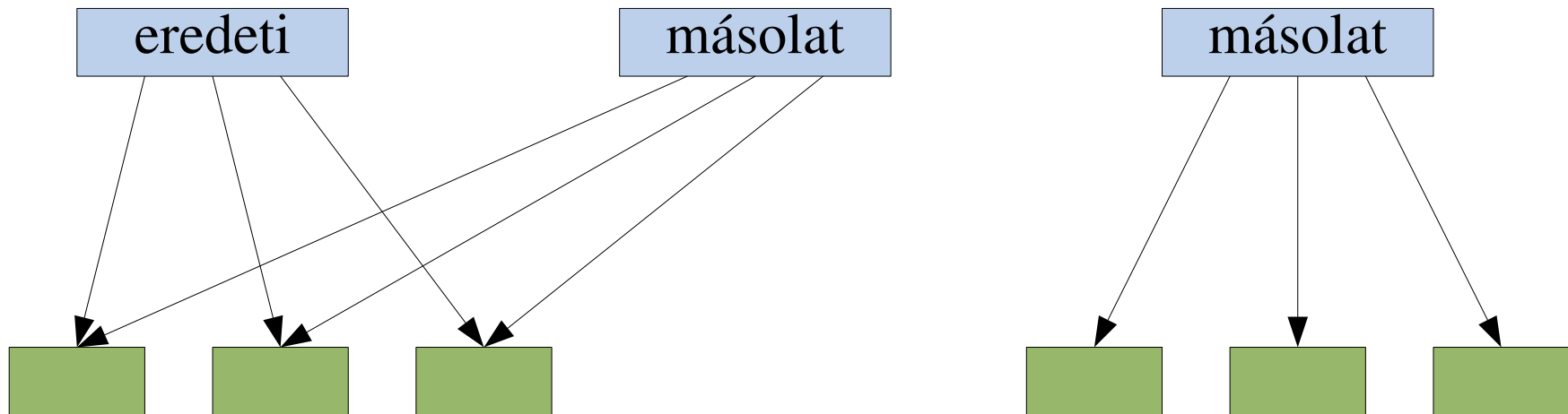
ReadWriteStream

FileStream

Random

Smalltalk - Object osztály

- Szülő nélküli ősosztály: Object
- Metódusok:
==, =, ~~ , ~=, isNil, notNil
copy, shallowCopy, deepCopy



Smalltalk - Boolean

- not, &, |
- and:, or:, xor:, eqv:
- ifTrue:, ifFalse:,
ifTrue:ifFalse:, ifFalse:ifTrue:
- False osztály egyetlen példánya:
false
- True osztály egyetlen példánya:
true

Smalltalk - Collection

- Collection absztrakt osztály
- Hozzáadás: add: addAll:
- Felsorolás: do: collect:
select:
- Teszt: includes: isEmpty
notEmpty
- Törlés: remove: removeAll:
- Tulajdonságok: size

Smalltalk - kollekciók

- Bag: Rendezetlen, ismétlődés előfordulhat
- Set: Rendezetlen, ismétlődés nem lehet
- Array: Rendezett, indexelés egész számmal
- String: Karakterre optimalizált Array
- Dictionary:
Rendezetlen, indexelés tetszőleges objektummal

Smalltalk - kollekciók

- Bag new
add: 'aa' withOccurrences: 3
- Set new add: 5; add: 7
- 'Hello Guys' asUppercase
- d := Dictionary new.
d at: #z put: 'zeta'.
d at: #a put: 'alfa'.
d at: #g put: 'gamma'

Smalltalk - Magnitude

- <, <=, >, >=,
between:and:, min:, max:, ...
- Character
value:, asciiValue, digitValue
isDigit, isLetter, ...
- Date
dayOfWeek:, isLeapYear, ...

Smalltalk - Number

- Műveletek
 - negated, +, -, *, /
 - even, odd, negative, positive
- Konkrét reprezentációk
 - Float, Fraction
 - Integer
 - factorial, gcd:, lcm:

Smalltalk - Stream

- Kollektciók hátrányai:
 - Felsorolás és elemhozzáadás keverve nem lehetséges
 - Aktuális pozíciót a kollekción kívül kell tárolni
- Megoldás a Stream:
 - Alapja egy létrehozáskor megadott kollektció
 - Tárolja az aktuális pozíciót, ahonnan olvashatunk és ahova írhatunk

Smalltalk – Stream kezelés

- atEnd, do:
- soron következő objektum(ok) olvasása
next, next:
- objektumok kiírása
nextPut:, nextPutAll:, next:put:
- pozicionálás
position, position:
- létrehozás
on:, on:from:to:

Smalltalk - osztály információ

- Minden osztályt egy objektum reprezentál
- A példányosítás az osztály (mint obj.) művelete
- A reprezentáns objektum neve megegyezik az osztály nevével (pl. Integer)
- A reprezentáns objektumnak is van típusa: egy osztály, az ilyen metaosztálynak nevezik
- A metaosztályt is reprezentálja egy objektum, sőt annak is van típusa...
- Osztály reprezentáns objektumára hivatkozás:
class üzenet

Smalltalk - osztály létrehozása - régi szintaxis

- 1. Osztály interface létrehozása:
subclass: classNameString
instanceVariableNames: InstVarNames
classVariableNames: ClassVarNames
poolDictionaries: PoolNames
category: categoryNameString
- 2. Leírás hozzárendelése: comment:
- 3. Metódusok definiálása: methodsFor:

Smalltalk - osztály létrehozás régi szintaxis (könyv)

Object

```
subclass: #MyClass
instanceVariableNames: 'x'
classVariableNames: 'a'
poolDictionaries: ''
category: nil !
```

MyClass comment: 'this is my class' !

MyClass comment !

```
!MyClass methodsFor:
'inicializalas'!
init
    | u v |
    u := 3. v := 11.
    x := u + v
!
seta: v
    a := v
!!
```

Smalltalk - osztály létrehozása

- Osztály interface létrehozása (subclass: üzenet)

```
superclass-name subclass: new-class-name [  
  | instance variables |  
  pragmas  
  message-pattern [ statements ]  
  ...  
  class-variable := expression.  
  ...  
]
```

- Pragmák

```
<comment: 'Class comment'>
```

```
<category: 'Examples'>
```

Smalltalk - metódusok definiálása

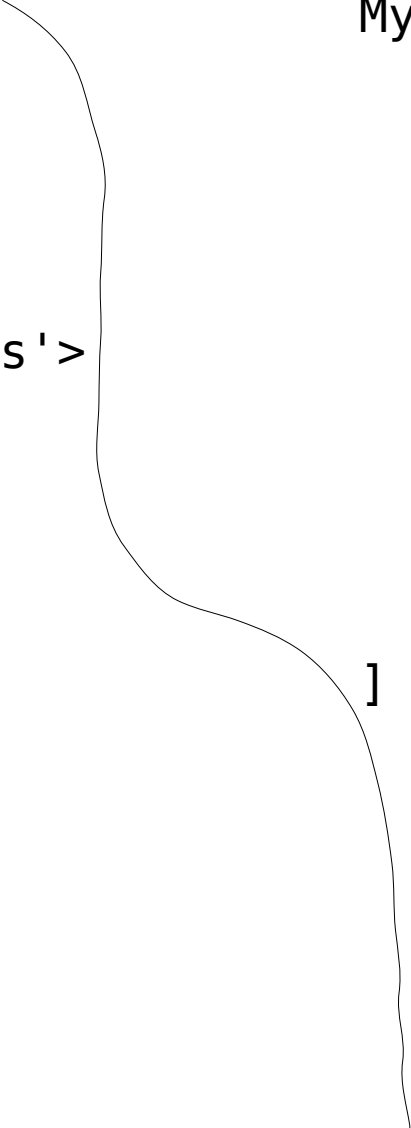
- Metódusdefiníció (extend üzenet)

```
class-expression extend [  
    message-pattern [ statements ]  
    ...  
]
```

Smalltalk – osztály létrehozás

```
Object subclass: MyClass [  
  | x |  
  <comment: 'this is my class'>  
  <category: 'mine'>  
  
  init [  
    x := 3  
  ]  
  
  a := 1.  
]
```

```
MyClass extend [  
  seta: v [  
    a := v  
  ]  
  
  setx: v [  
    x := v  
  ]  
  
  getval [  
    ^ a+x  
  ]  
]
```



Smalltalk - metódus létrehozás

```
MyClass extend [  
  getx [  
    ^x  
  ]  
  
  geta [  
    ^a  
  ]  
]
```

```
MyClass class extend [  
  new [  
    | x |  
    x := super new.  
    x init.  
    ^x  
  ]  
]
```

```
MyClass extend [  
  printOn: stream [  
    super printOn: stream.  
    ', x =' displayOn: stream.  
    x printOn: stream.  
  ]  
]
```

Smalltalk - C++

- Egyszeres öröklődés, példány (tag) és osztály (static) változók, metódusok (üzenetek)
- Minden adattag “private”, objektumszintű védelemmel; minden metódus “public”
- Konstruktor – new osztály metódus
- this – self, Parent:: – super
- Minden metódus “virtuális”
- Absztrakt osztály (is) dinamikusan: subclassResponsibility üzenet

Programozási nyelvek

Jelölésrendszer Kifejezőerő

Történelem Elvárások

Fejlődés

Procedurális

Moduláris

Objektum or. pr.

Értékek

- Érték (adat): amivel műveleteket végzünk
 - Előfordul a programba ágyazva (literál)
 - Változókbán tároljuk
- Literál: érték szöveges formája a programban
 - Formátumból kiderül a típusa (5, 3.0, 'x')
- Adattípus:
 - Értékek halmaza: kódolás, méret, szerkezet
 - Szemantika: műveletek

Típusok

- Típus általában: tulajdonságok a compiler vagy futtatórendszer számára (kifejezések ellenőrzése)
- Lehetséges értékek és megengedett műveletek együttese
- Adattípusok osztályozása
 - Elemi típus: nem felbontható értékek
 - Összetett típus: részekre bontható értékek
 - Rekurzív típus: érték saját típusával megegyező típusú értéket tartalmazhat

Típusok ...

- Típus megadása
 - Implicit (literál) (pl. 3.0)
 - Deklaráció (pl. int i;)
 - Típuskövetkeztetés (kifejezésből)
- Típusosság fajtái
 - Statikus (fordítási időben meghatározható típusok)
 - Erős (fordítási időben ellenőrizhető típusok)
 - Gyenge (futási idejű ellenőrzés szükséges)

Típuskonverzió

- Érték típusának megváltoztatása
 - Automatikus
 - Explicit
- Értékkészlet szerint
 - Bővítő (általában automatikus)
 - pl. $2.3 + 4$ ($4 \rightarrow 4.0$)
 - Szűkítő (általában explicit)
 - pl. $3.2 \rightarrow 3$ (adatvesztés)

Karakterlánc-konverzió

- Nem típuskonverzió!
 - az objektum megváltozik
 - pl. 42 → "42" (`int` → `string`)
- Automatikus
 - Java: `toString()`
- Explicit
 - Ada

Elemi típusok

- A nyelv alkalmazási területére jellemző
 - Pointer, referencia, skalár, diszkrét, valós, ...
- Implementációfüggő értékkészlet lehet
- Tipikus beépített:
Boolean, Integer, Real, Character
- Definiálható:
 - Felsorolás típus (Enumeration)
- Sokszor kitüntetett szerep jut az elemi típusoknak

Pointer

- Imperatív nyelvekben jellemző
 - Hatékonyság (másolás elkerülésére)
 - Rekurzív típusok kifejezéséhez
- Memóriacímek absztrakciója
- Szorosan kapcsolódik a memóriakezeléshez (dinamikus változók, dinamikus tárterület)
- Műveletek: létrehozás, megszüntetés, címképzés, indirekció (közvetett címzés)
- Referencia: biztonságos(abb) mutató

Kifejezések

- Értékekből értékek: függvény
- Elemei:
 - Konstans/literál
 - Változó
 - Függvény/operátor
- Kifejezés típusa: kiszámított érték típusa
- Fontos jellemző: aritás (argumentumok száma - f^k)
- Operátorok: rögzített aritás, asszociativitás, precedencia

Operátorok

- Nevek helyett szimbólumok (+,-,new)
- Függvény jelölések
 - Prefix: $f(1), +(1, 2)$
 - Infix (operátor): $f\ 1, 1 + 2, a ? b :$
c
- Infix természetesebb, de nem egyértelmű
 - Ugyanaz az operátor fordul elő többször:
asszociáció (pl. balról jobbra-balasszociatív)

$$-(5, -(3, 2)) \qquad 5 - 3 - 2$$
 - Operátorok keverednek: precedencia

$$+(2, *(1, 7)) \qquad 2 + 1 * 7$$

Típuskonstrukció

- Három alapvető forma:
- Direktszorzat típus
- Unió típus
- Rekurzív/sorozat/halmaz (iterált) típus

Összetett típusok képzése

- Descartes szorzat (n-es, rekord)
- Diszjunkt unió (variant rekord, union)
 $\{0,1\} + \{0,1\} \Rightarrow \{0_x, 1_x, 0_y, 1_y\}$
- Leképezések (tömb, szótár, függvény)
- Hatványhalmaz (halmaz)
- Rekurzió (dinamikus adatszerkezetek)

Descartes szorzat

- struct s {
 int i;
 char c;
 float f;
} x;

típus

szelektor

értékhalmoz

- $s = \text{int} \times \text{char} \times \text{float};$
- $x \in s$

Diszjunkt unió

- ```
union u {
 int i;
 char c;
 float f;
} x;
```

  - $u = \text{int} + \text{char} + \text{float};$
  - $x \in u$
  - „ $x = i$  vagy  $c$  vagy  $f$ ”
  - kötött/szabad unió
  - OO  $\rightarrow$  öröklődés

# *Leképezéstípus*

$$A = \{0, 1\}$$

$$B = \{a, b, c\}$$

$$A \rightarrow B = \left\{ \begin{array}{ll} (0 \mapsto a, & 1 \mapsto a), \\ (0 \mapsto a, & 1 \mapsto b), \\ (0 \mapsto a, & 1 \mapsto c), \\ (0 \mapsto b, & 1 \mapsto a), \\ (0 \mapsto b, & 1 \mapsto b), \\ (0 \mapsto b, & 1 \mapsto c), \\ (0 \mapsto c, & 1 \mapsto a), \\ (0 \mapsto c, & 1 \mapsto b), \\ (0 \mapsto c, & 1 \mapsto c) \end{array} \right\}$$


# *Leképezéstípusok*

- Tömb: indexek  $\rightarrow$  értékek  
`bool x[3];`  
 $\{0,1,2\} \rightarrow \{\text{true}, \text{false}\}$
- Speciális: vektor, halmaz
- Szótár, hasítótábla: kulcsok  $\rightarrow$  értékek
- Függvények: input  $\rightarrow$  output  
`int f( bool b, char c );`

`bool  $\times$  char  $\rightarrow$  int`

# *Imperatív rekurzív típus*

```
struct node {
 int data;
 node* left;
 node* right;
};
```



type

```
NodePtr= ^Node;
Node= record
 data: Integer;
 left, right: NodePtr;
end;
```

# *Típus ekvivalencia*

- Mikor tekintünk két értéket azonos típusúnak?
  - Szerkezeti ekvivalencia: Ugyanaz az értékhalmoz
  - Név ekvivalencia: Ugyanazon a helyen (névvel) lettek definiálva
- Név ekvivalencia eldöntése könnyebb



# *Típus teljesség*

- Elsőrendű, másodrendű típusok: minden helyzetben használhatók-e?
- Az összetett és/vagy felhasználó által definiált típusok általában hátrányban vannak
- Elv: Ne legyen felesleges megszorítás

# *könyv*

- 5. fejezet (118-165), kivéve 5.1.2
- 6. fejezet (166-201)
- ! Könyvben iterált, de nekünk
  - Függvény
  - Halmaz
  - Rekurzív

# Programozási nyelvek

Jelölésrendszer   Kifejezőerő  
Történelem   Elvárások

## Fejlődés

Procedurális

Moduláris

Objektum or. pr.

## Típusok

Elemi   Összetett   Rekurzív

Típusosság   Típuskonstrukció

# *Programvezérlés*

- Imperatív
- Deklaratív és funkcionális
- Párhuzamos
- Eseményvezérelt
- (Adatfolyam)

# *Utasítások*

- Imperatív programozás fő jellemzője
- Utasítások végrehajtása állapotátmenetet okoz
- Egyszerű utasítások
  - Üres utasítás, értékadás, alprogramhívás
- Összetett utasítások (vezérlési szerkezetek)
  - Szekvencia, (párhuzamosság)
  - Feltételes utasítás, ismétlés

# *Mellékhatás*

- Kifejezés kiértékelése közben külső változó értéke megváltozhat
- A mellékhatások nehezen követhetővé teszik a programot
- A mellékhatásokat lehetőleg kerülni, előfordulásukat megfelelően dokumentálni kell

# *könyv*

- 3. fejezet (vonatkozó részei)

# *Imperatív VS deklaratív*

- Imperatív:
  - Hogyan kell egy feladatot megoldani
- Deklaratív:
  - Mi a megoldandó feladat
  - A specifikáción van a hangsúly, funkcionális esetben a program egy függvény kiszámítása, logikai esetben a megoldás megkeresését a futtató környezetre bízunk



# *Funkcionális programozás*

- Értékek, (matematikai) kifejezések és függvények
- A program egy függvény
- Változók értéke nem változik
- Ciklus helyett rekurzió
- Interakció (input/output) nehezen fejezhető ki
- Hivatkozási átlátszóság (referential transparency)
- Helyesség ellenőrzés könnyebb

# *Haskell – értékek, típusok*

- Int: 5, -2, 0
- Float: 2.5, -7.7
  - +, -, \*, ^, div, mod, abs, negate
- Bool: True, False
  - &&, ||, not
- Char: 'a', '\$', '\n', '\'', '\"'
- ==, /=, <=, >=, <, >

# *Haskell - kiértékelés*

- Függvényhívás:  
*fv arg1 arg2 ...*
- $(7 - 3) * 2$  » 8
- `not True` » False
- `fv 3 'a'` » True
- `fv ( g 7) 'z'` » False

# *Haskell - Függvények*

- Függvény típus, függvény törzs
- Kiértékelés behelyettesítéssel

```
size :: Int
size = 13+72
```

```
square :: Int -> Int
square n = n*n
```

```
length :: Int -> Int -> Int
length x y = y * square x + size
```

```
>length 3 5
130
```

# *Behelyettesítés*

```
size :: Int
size = 13+72
```

```
square :: Int -> Int
square n = n*n
```

```
length :: Int -> Int -> Int
length x y = y * square x + size
```

```
length 3 5 =>
5 * square 3 + size =>
5 * (3 * 3) + size =>
5 * 9 + size =>
45 + size =>
45 + (13+72) =>
45 + 85 =>
130
```

# *Lusta VS mohó kiértékelés*

Mohó:

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

Lusta:

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

- Lusta: normalizáló kiértékelés, mindig megtalálja a normálformát, ha létezik

# *Haskell - függvények*

- Többsoros definíció, mintaillesztés, sorrend

```
exOR :: Bool -> Bool -> Bool
exOR True x = not x
exOR False x = x
```

- Örök

```
max :: Int -> Int -> Int
max x y
 | x >= y = x
 | otherwise = y
```

- Feltételes kifejezés

```
max :: Int -> Int -> Int
max x y = if x > y then x else y
```

# *Haskell - szintaxis*

- Layout (elrendezés) szabály blokkokra
- Azonosítók betűvel kezdődnek
  - Függvény, változó : *kisbetű*; Típusnév: *nagybetű*
- Foglalt szavak (if, else, let, type, ...)
- Operátorok, függvények
  - $2+3 \gg (+) \ 2 \ 3$      $\text{max } 2 \ 3 \gg 2 \ \text{'max'} \ 3$
- Precedencia                     $f \ n+1 \ll\gg (f \ n)+1$



# *Haskell - rekurzió*

```
factorial :: Int -> Int
```

```
factorial x
```

```
 | 0==x = 1
```

```
 | 0 < x = x * factorial (x-1)
```

```
fibonacci :: Int -> Int
```

```
fibonacci x
```

```
 | 0==x = 0
```

```
 | 1==x = 1
```

```
 | 1 < x = fibonacci (x-2) + fibonacci (x-1)
```

# *Rekurzív függvények*

- Rekurzív hívás mindig feltételvizsgálat mögött
- Rekurzív függvényt két esetre kell felkészíteni
  - Bázis eset: nem kell újra meghívnia magát
  - Rekurzív eset: Meghívja magát újra
- Biztosítani kell, hogy mindig elérjük a bázis esetet
- Rekurzió speciális esete: iteráció

# *Haskell - tuple*

- Tuple (*n*-es) adattípus:  
`(1, 'a', "ab") :: (Int, Char, String)`

```
minmax :: Int -> Int -> (Int, Int)
```

```
minmax x y
```

```
 | x < y = (x, y)
```

```
 | otherwise = (y, x)
```

```
first :: (Int, Int) -> Int
```

```
first (n, m) = n
```

# *Haskell – típus szinoníma*

```
type Pair = (Int,Int)
```

```
minmax :: Int -> Int -> Pair
minmax x y = (min x y, max x y)
```

```
type String = [Char]
type Item = (String, Float)
```

```
which :: Item -> Item -> String
which (s1,f1) (s2,f2)
 | f1 < f2 = s1
 | otherwise = s2
```

# *Haskell - lista*

- Lista: azonos típusú elemek sorozata

`[1,2,3,4] :: [Int]`

`[True] :: [Bool]`

`['a','b','c'] :: [Char]`

`[min,max] :: [Int->Int->Int]`

`[[1,2],[1,2,3]] :: [[Int]]`

# *Haskell – lista*

- Üres lista: `[]`
- Megszámlálható típusok listái:

`[1..7] = [1,2,3,4,5,6,7]`

`['a'..'g'] = ['a','b','c','d','e','f','g']`

`[1,3..7] = [1,3,5,7]`

`[0.0,0.3 .. 1.0] = [0.0,0.3,0.6,0.9]`

`['a','c'..'g'] = ['a','c','e','g']`

# *Overloading, polimorfizmus*


- Overloading: Egy név több függvényt is jelöl
- Polimorfizmus: Ugyanaz a függvény többféle típusú paraméterrel is hívható
- Monomorf: Nem polimorf

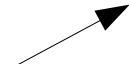
```
second :: (a,b) -> b
second (_,y) = y
```

```
second (5, 'c') » 'c'
second (3.2, [1,4]) » [1,4]
```

- Típus következtetés (type inference)

# *Függvény típusok*

`plusz :: Int -> Int -> Int`  Típusállandó  
`plusz x y = x + y`

`plusz :: Num a => a -> a -> a`  Típusváltozó  
`plusz x y = x + y` } Típusfüggetlen szignatúra

- Az utóbbi a Num típusosztályba tartozó típusokra értelmezett
- Numerikus típusosztály:

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



# *Többváltozós függvények értelmezése*

- A típusdefiníció jobbasszociatív

$\text{fv} :: a \rightarrow b \rightarrow c \quad \gg \quad \text{fv} :: a \rightarrow (b \rightarrow c)$

- A függvényalkalmazás balasszociatív

$\text{fv } x \ y \gg (\text{fv } x) \ y$

# *Haskell - listakezelés*

- Lista részei: fej (head), farok (tail)
- Mintaillesztés listára: `head:tail`

```
head :: [a] -> a
head (h:_) = h
```

```
tail :: [a] -> [a]
tail (_:t) = t
```

```
empty :: [a] -> Bool
empty [] = True
empty _ = False
```

# *Haskell* - case

- Mintaillesztés a függvényen belül

```
case e of
 p1 -> e1
 ...
 pn -> en
```

```
firstDigit :: String -> Char
firstDigit s =
 case (digits s) of
 [] -> '\0'
 (x:_) -> x
```

# *Haskell – listafeldolgozás*

- Lista a legegyszerűbb rekurzív adatszerkezet
- Feldolgozása rekurzív függvényekkel történik

```
sum :: [Int] -> Int
sum [] = 0 {- bázis eset -}
sum (h:t) = h + sum t {- rekurzív eset -}
```

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

# *Haskell – függvény paraméter*

- Függvény paramétere lehet másik függvény is (= magasabb rendű függvény)

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

```
map double [1,2,3] » [2,4,6]
map uppercase "abc" » "ABC"
```

# *Haskell – anonim függvény*

- Függvény literálok is használhatók, nem kell elnevezni őket

```
map (\x -> 3*x) [1,2,3] » [3,6,9]
```

- Anonim függvény:  $\backslash v \rightarrow e$
- Függvény definíció jelentése:

```
f x = x + 1
```

```
f = \x -> x+1
```

# *Haskell - függvények*

- Minden függvény egyargumentumú
- Egy többargumentumú függvény olyan egyargumentumú függvény, amely függvényt ad vissza

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

```
mul2 :: Int -> Int
mul2 = mul 2
```

```
>mul2 7
14
```

# *Részleges függvényalkalmazás*

- Egy többargumentumú függvény néhány paraméter értékét rögzítjük (Curry-módszer)

```
novel :: Int -> Int
novel x = (+) 1 x
```

*vagy*

```
novel = (+) 1
```



# *Haskell – lokális definíciók*

```
sumSquares :: Int -> Int -> Int
sumSquares x y =
 sqx + sqy
 where
 sqx = x * x
 sqy = y * y
```

```
sumSq :: Int -> Int -> Int
sumSq x y =
 let sq a = a*a
 in
 sq x + sq y
```

# *Haskell - általános függvények*

- Elemek feldolgozása: `map`
- Részhalmaz kiválasztása: `filter`  
`isEven [1,2,3,4] » [2,4]`
- Elemek kombinálása: `fold`  
`sum [1,2,3] » 1+2+3 = (1+2)+3`  
`max [1,2,3] » 1 `max` 2 `max` 3`
- Átszervezés: `zip`, `unzip`  
`zip [1,2] [1,2] » [(1,1), (2,2)]`

# *Haskell – általános függvények*

```
filter even [1,2,3,4] » [2,4]
```

```
foldr (+) 0 [1,2,3] » 6
foldr1 max [1,2,3] » 3
```

- Függvény kompozíció: .

```
(f . g) x = f (g x)
```

```
filter ((==0).(`mod` 2)) [1,2,3,4]
```

- Haskell standard library: `prelude.hs`

# *Haskell – adattípusok*

- Haskell típusok:  
alap, összetett (tuple, lista, függvény)

- Felsorolás típus

Konstans adatkonstruktor függvény

```
data Temperature = Hot | Cold
data Season = Spring | Summer | Autumn | Winter
```

- Szorzat típus

Típuskonstruktor

Adatkonstruktor függvény

```
data People = Person String Int
```

```
getAge :: People -> Int
getAge (Person name age) = age
```

# *Haskell - adattípusok*

- Alternatívák: diszjunkt egyesítés (unió)

```
data Shape = Circle Float | Rect Float Float
```

```
isRound :: Shape -> Bool
isRound (Circle _) = True
isRound (Rect _ _) = False
```

```
>:type Circle {- konstruktor függvény -}
Circle :: Float -> Shape
```

```
>:type Rect
Rect :: Float -> Float -> Shape
```

# *Haskell – polimorf típusok*

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

```
>:type Node
```

```
Node :: a -> Tree a -> Tree a -> Tree a
```

```
depth :: Tree a -> Int
```

```
depth Nil = 0
```

```
depth (Node n t1 t2) =
 1 + max (depth t1) (depth t2)
```

```
{- Lista definiálható lenne
```

```
data List a = [] | (:) a (List a)
```

```
-}
```

# *Haskell - modul*

```
module MMM (exportlista) where
```

```
import importlista
```

```
data d = ...
```

```
f x = ...
```

- Main modul
- compiler esetén main függvény a kezdeti kifejezés

# *Haskell – absztrakt adattípus*

```
module Stack (Stack, emptyStack, push, pop,
toList) where
data Stack t = Stack [t]
```

```
emptyStack :: Stack t
emptyStack = Stack []
```

```
push :: Stack t -> t -> Stack t
push (Stack s) t = (Stack (t : s))
```

```
pop :: Stack t -> t
pop (Stack (t : s)) = t
```

```
toList :: Stack t -> [t]
toList (Stack s) = s
```



# *Haskell – lusta kiértékelés*

- Függvényargumentumok átadása kétféleképpen történhet:
  - Normál: argumentumok kiértékelése, majd átadása
  - Lusta: argumentumok átadása, kiértékelés szükség esetén

```
f :: Float -> Float -> Float
f x y = if x>0 then y else 1
```

```
f 1 (1/0) » inf
f 0 (1/0) » 1.0
```

# *Haskell – végtelen listák*

- Lusta kiértékelés miatt használhatunk nem korlátos adatszerkezetet paraméterként

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

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

```
>prefix nat 5
[1,2,3,4,5]
```

# *Haskell – típus osztályok*

```
element :: a -> [a] -> Bool
element x [] = False
element x (h:t) = x == h || element x t
```

- `==` operátornak működnie kell az `a` típusra
- Megoldás: overloading
- Egy műveletcsoportot megvalósító típusokat osztálynak nevezzük
- Beépített osztályok: `Eq`, `Ord`, `Num`, ...

# *Haskell - osztályok*

```
class Visible a where
 toString :: a -> String
 size :: a -> Int
```

```
...
instance Visible Bool where
 toString True = "True"
 toString False = "False"
 size _ = 1
```

```
...
```

```
f x = ">>" ++ toString x ++ "<<"
>:type f
f :: Visible a => a -> [Char]
```

# *Haskell - Monádok*

- A Haskell program dolgozhat imperatív programot reprezentáló adatszerkezettel (függvény is adat)
- Mellékhatások kezelése – csak monádokban!
- Mondaikus műveletek egymás után hajthatók végre
- Az I/O adatszerkezet típusa:  
    `I0 a (monad)`
- `main :: I0 ()` (végrehajtja a programot!)

# *Haskell - Input/output*

- I/O függvények
  - `print :: Text a => a -> IO ()`
  - `interact :: (String -> String) -> IO()`
  - `writeFile :: String -> String -> IO()`
  - `readFile :: String -> IO String`
  - `getChar :: IO Char`
  - `putChar :: Char -> IO ()`

# *Haskell - Input/output*

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

```
main = helloworld
```

```
main = print [(n,n*n)|n <- [0..10]]
```

```
main = interact (filter isUpper)
```

```
main = writeFile "ascii.txt" (show chars)
 where chars = [(x, chr (x)) | x <- [65..90]]
```

# *Haskell - Input/output*

- A beolvasott adatokat biztonságos módon átadhatjuk függvényeknek (a visszatérési érték nem függhet tőlük)
- bind operátor: monadikus műveletek összekapcsolása

$(>>=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$

```
readFile "x.txt" >>= \input ->
 let output = process input
 in
 writeFile "y.txt" output
```



# *Haskell – do jelölés*

- Az imperatív részek írását megkönnyítő szintaktikai segédeszköz a do jelölés

```
process :: String -> String
process s = ...
```

```
main = do input <- readFile "x.txt"
 writeFile "y.txt" (process input)
```

```
main = do putChar 'h'
 putChar 'e'
 putChar 'l'
 putChar 'l'
 putChar 'o'
```

# *könyv*

- 16. fejezet (589-634)

# Programozási nyelvek

Jelölésrendszer   Kifejezőerő  
Történelem   Elvárások

## Fejlődés

Procedurális

Moduláris

Objektum or. pr.

## Típusok

Elemi   Összetett   Rekurzív

Típusosság   Típuskonstrukció

## Programvezérlés

Imperatív

Deklaratív

Utasítások

Függvények

Párhuzamos

Eseményvezérelt

# *Hozzárendelés*

- A programozás során azonosítókat rendelünk hozzá mindenféle dolgokhoz (változók, típusok,...)
- A hozzárendelés különböző időpontokban történhet
  - Nyelv tervezésekor
  - Nyelv implementálásakor
  - Fordítási időben
  - Futási időben

# *Deklaráció*

- Deklaráció: a programozó által előírt hozzárendelés
- A deklaráció hatását fordítási és/vagy futási időben fejti ki
- A deklaráció tekinthető a fordítóprogramnak vagy futtató rendszernek adott utasításnak
- Pszeudo utasítások / valódi utasítások

# *Deklaráció*

- Mit
  - Konstans (érték), típus, változó
  - Függvény, eljárás: csak ahol nem elsőrendű érték, vagy egyszerűsített szintaxis miatt
- Hogyan
  - Független
  - Szekvenciális
  - Rekurzív

# *Környezet*

- Azonosítókra való hivatkozások nem értelmezhetők önmagukban  
 $n := m+1$
- A környezet határozza meg az azonosítók jelentését
- A környezet hozzárendeléseket tartalmaz
- Minden utasításhoz és kifejezéshez tartozik egy környezet, amelyben végrehajtjuk/kiértékeljük

# *Környezet - Pascal példa*

```
program p;
 const z= 0;
 var c: char;

 procedure q;
 const c= 3000;
 var b: boolean;
 begin
 ...
 end;

begin
 ...
end.
```



# *Hatáskör*

- Nem tévesztendő össze az élettartammal
- Hatásköre azonosítóknak van és a (statikus)programszöveg egy részére vonatkozik
- Élettartama változóknak (általában azonosítóval hivatkozunk rá) van és a program futási idejére vonatkozik
- Hatásköri szabályok mondják meg, hogy egy azonosítóra a programszöveg mely részén hivatkozhatunk

# *Blokk struktúra*

- Blokk struktúra: hatásköri egységek egymásba ágyazása
- Fejlődés:
  - Monolitikus, egy blokk az egész program
  - Lapos (kétszintű), külső blokkban belső blokkok
  - Egymásba ágyazott (tetszőleges mélységig)
- A modern nyelvek az utóbbit támogatják

# *Hatáskör*

- Az azonosítóknak kétféle előfordulása van:
  - Definiáló (hozzárendelő) előfordulás  
`const n=7;`
  - Alkalmazott (felhasználó) előfordulás  
`z=n+1;`
- Mindkét előfordulásból lehet több, a hatásköri szabályok egyértelműsítik, hogy egy alkalmazott előforduláshoz melyik definiáló előfordulás tartozik

# *Statikus és dinamikus hozzárendelés*

```
const s= 2;
```

```
function scale(d: integer);
```

```
begin
```

```
scaled := d * s;
```

```
end;
```

```
procedure proc;
```

```
const s= 3;
```

```
begin
```

```
... scale(5) ...
```

```
end;
```

```
begin
```

```
... scale(5) ...
```

```
end
```

# *Blokk utasítás, kifejezés*

- Blokk: Lokális környezet + utasítás/kifejezés

```
if m > n
then begin
 t := m;
 m := n;
 n := t
end
```

```
if (m > n) {
 int t;

 t= m;
 m= n;
 n= t;
}
```

- Haskell: `let x=3*y+2 in x+x`

# *Változók*

- Változó értéket tárol (vagy nem definiált)
- A program állapotának egy részét tartalmazza
- Változón végezhető műveletek: kiolvasás, beírás
- Imperatív/deklaratív változók
- Tárkezelés: memóriaterületek, szabad/foglalt rekeszek

# *Összetett változók*

- Részváltozókból áll
- Teljes/szelektív kezelés
- Tömbváltozók (statikus, dinamikus, flexibilis)

```
int x[10];
```

```
int* px= new int[z];
```

```
vector<int> v;
v.push_back(7);
```

# *Élettartam*

- Változó létrejötte és megszűnése közti idő
- Automatikus, manuális változókezelés
  - Automatikus – automatikus (blokk változók)
  - Manuális – manuális (halom változók)
  - Manuális - automatikus (szemétgyűjtés)
- Érvénytelen hivatkozás
- Perzisztens (tartós) változó



# *könyv*

- 4. fejezet (102-116)

# *Logikai programozás*

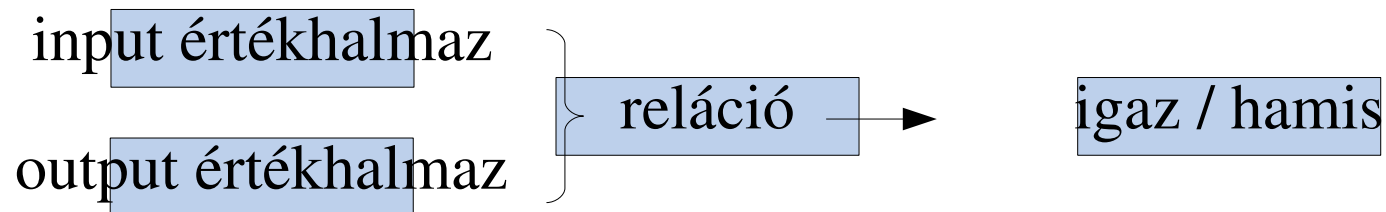
- Imperatív



- Funkcionális



- Logikai



# *Prolog - PROgramming in LOGic*

- Egy Prolog program csak az adatokat és az összefüggéseket tartalmazza. Kérdések hatására a “programvégrehajtást” beépített következtető-rendszer végzi
- Programozás Prologban:
  - Objektumok és azokon értelmezett relációk megadása
  - Kérdések megfogalmazása a relációkkal kapcsolatban

# *Prolog*

- Prolog érték: term
  - Egyszerű term: `alma`, `1000`, `x10`, `mary`
- Relációk megadása:
  - tények
  - következtetési szabályok
- Kérdésfeltevés interaktív módon
  - Eldöntendő kérdés
  - Általános kérdés

# *Prolog - tények*

- Tények azt fejezik ki, hogy a megadott objektumok között fennáll bizonyos reláció

`likes( john, mary).`

`true.`

`female( mary).`

`child_of( john, mary, peter).`

- A tények egy adatbázist definiálnak

# *Prolog - kérdések*

- Eldöntendő kérdések ugyanúgy néznek ki, mint a tények, csak más a szöveggörnyezet
  - ?- likes( john, mary).
  - ?- true.
  - ?- female( mary).
- A rendszer az adatbázis végignézése után válaszol a kérdésre (igen/nem)

# *Prolog – általános kérdés*

- Általános kérdést változók segítségével tehetünk fel
- Változók aláhúzásjellel, vagy nagybetűvel kezdődnek

`X, _x, Long_Var_Name, _111`

`?- likes( john, X).`

# *Prolog - következtetőrendszer*

- Kérdésekben konjunkció is előfordulhat  
?- likes( john, mary), likes( john, bela).  
  
?- likes( john, X), likes( mary, X).  
  
?- female( mary).
- A Prolog következtetőrendszer visszalépéses (backtracking) keresést alkalmaz a válaszok megtalálásához



# *Prolog - következtetőrendszer*

- Egy kérdés (goal) több részcélből (subgoal) áll
- A részcélokra sorban egymás után keres válaszokat
- A célokat és a tényeket illesztéssel kapcsolja össze
- Illeszkedés esetén megjegyzi az illeszkedő adatbázisbeli elemet a keresés esetleges folytatásához (és lép a következő rész célra)

# *Prolog - következtetőrendszer*

- Ha minden rész célra talált választ (illeszkedő adatbáziselemet), akkor a kérdésre talált egy választ
- Ha egy rész célra nem talál választ, akkor visszalép az előző rész célra és ahhoz próbál új illeszkedő elemet találni (backtracking)
- Sikeres keresés után is ki lehet kényszeríteni a visszalépést az összes válasz megtalálásához

# *Prolog – változók, illesztés*

- A változóknak három állapota van:
  - Szabad vagy kötetlen
  - Szabad megosztott
  - Kötött
- A változók illesztéskor válhatnak kötétté (adat-változó), illetve szabad megosztottá (változó-változó)
- A változók létrehozáskor és visszalépéskor válnak szabaddá

# *Prolog - következtetőrendszer*

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

```
?- likes(john, X), likes(mary, X).
X=book ?;
no
```

# *Prolog - szabályok*

- Az adatbázisba nem csak tényeket, hanem szabályokat is felvehetünk

```
likes(john, mary).
likes(john, joe).
likes(john, david).
...
```

---

```
likes(john, X) :- person(X).
...
person(mary).
person(joe).
person(david).
...
```

# *Prolog - szabályok*

```
bird(X) :- animal(X), has_feathers(X).
sibling(A, B) :-
 parent(A, P), parent(B, P), A \= B.
likes(john, X) :-
 female(X), likes(X, wine).
```

- A változók hatásköre egy szabály.
- Az adatbázis egy elemét (tény vagy szabály) klóznak nevezzük
- Következtetésnél ha a szabály baloldala illeszkedik, akkor a jobboldal bekerül a részcélok közé (rekurzív definíció lehetséges)

# *Prolog - term*

- Egyszerű term:
  - Szám: egész vagy valós
  - Atom: kisbetűvel kezdődő karaktersorozat
  - Változó
- Összetett term:  
 $\text{atom}(\text{term}_1, \text{term}_2, \dots)$
- Operátorok:  $a+b \ll \text{+(a,b)}$   
`person( name( tom, smith), age( 17))`  
`f( 113, g( X), a + Y))`

## *Prolog - illesztés ( $t_1 = t_2$ )*

- Kötött változó az értékével vesz részt
- Konstans önmagával illeszkedik (szám, atom)
- 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 termék illeszkednek, ha a funktor és a résztermék száma megegyezik, ill. a résztermék illeszkednek



# *Érték vizsgálat*

egyesítés

azonosság

aritmetikai

| $U$ | $V$    | $U = V$ | $U \neq V$ | $U == V$ | $U \neq V$ | $U \text{ is } V$ | $U ::= V$ | $U \neq V$ |
|-----|--------|---------|------------|----------|------------|-------------------|-----------|------------|
| 1   | 2      | nem     | igen       | nem      | igen       | nem               | nem       | igen       |
| a   | b      | nem     | igen       | nem      | igen       | hiba              | hiba      | hiba       |
| 1+2 | +(1,2) | igen    | nem        | igen     | nem        | nem               | igen      | nem        |
| 1+2 | 2+1    | nem     | igen       | nem      | igen       | nem               | igen      | nem        |
| 1+2 | 3      | nem     | igen       | nem      | igen       | nem               | igen      | nem        |
| 3   | 1+2    | nem     | igen       | nem      | igen       | igen              | igen      | nem        |
| X   | 1+2    | X=1+2   | nem        | nem      | igen       | X=3               | hiba      | hiba       |
| X   | Y      | X=Y     | nem        | nem      | igen       | hiba              | hiba      | hiba       |
| X   | X      | igen    | nem        | igen     | nem        | hiba              | hiba      | hiba       |

# *Prolog - illesztés*

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

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

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

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

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

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

# *Prolog – hívási minta*

- A hívási minta határozza meg, hogy egy adott reláció esetén kérdés megfogalmazásakor az egyes paraméterek helyén használhatunk-e változót
- Lehetséges esetek:
  - Nem lehet változó (+)
  - Csak (szabad) változó lehet (-)
  - Lehet változó is (?)

# *Prolog – beépített predikátumok*

- $=/2, =(?,?)$
- $is(+, +)$  aritmetika
- $:=(+, +), \backslash=(+, +), >(+, +), \dots$
- $==(?, ?)$  literális egyenlőség
- Listák kezelésére speciális jelölések
  - $[], [john, 1, a(Y)], [term|lista]$
  - “abcd”  $\ll \gg [97, 98, 99, 100]$

# *Prolog – listák*

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

```
?- member(d, [a,b,c,d,e]).
?- member(X, [a,b,c,d,e]).
```

```
% trafo(+,?)
trafo([], []).
trafo([H|T], [XH|XT]) :-
 XH is H+1, trafo(T, XT).
```

# *Prolog – megoldások száma*

- Determinisztikus predikátum esetén minden kérdésre legfeljebb egy megoldás van.
- Cut (!) beépített predikátum a keresés befolyásolására, visszalépési pontok megszüntetésére szolgál.
- Előnyei: sebesség, memóriafelhasználás
- Hátránya: nem “logikai programozás”

# *Prolog – dinamikus adatbázis*

- Klóz hozzáadása az adatbázishoz:
  - `asserta(+)`, `assertz(+)`
- Klóz eltávolítása:
  - `retract(?)`

```
asserta(likes(john, whiskey)).
```

```
retract(likes(john, _)).
```

# *Prolog – magasszintű predikátumok*

- Összes megoldás összegyűjtése egy listába:  
`findall( -X, ?G, ?L)`
- A G célnak tartalmaznia kell az X változót
- A G-re talált válaszok során az X által felvett értékeket összegyűjti az L listába

`findall( X, parents(X,eve,adam), L)`



## *Prolog - input/output*

- az I/O imperatív volta miatt nem illeszkedik jól a PROLOG-ba
- speciális, mellékhatásokkal járó predikátumokkal végezzük a file-ok kezelését (olvasás, írás stb.)
- DE: visszalépésnél nem „csinálja vissza” az I/O műveleteket
- `read(X)`, `get(X)`, `write(X)`, `put(X)`, `nl`, ...

# *könyv*

- 17. fejezet (637-683)

# Programozási nyelvek

Jelölésrendszer   Kifejezőerő  
Történelem   Elvárások

## Fejlődés

Procedurális

Moduláris

Objektum or. pr.

## Típusok

Elemi   Összetett   Rekurzív

Típusosság   Típuskonstrukció

## Programvezérlés

Imperatív

Deklaratív

Utasítások

Függvények

Párhuzamos

Eseményvezérelt

## Hozzárendelés

Változó   Hatáskör   Élettartam   Blokk

# *Absztrakció*

- Általánosítás és specializáció

```
...
t := a;
a := b;
b := t;
...
t := x;
x := y;
y := t;
...
```

```
procedure swap(var i:real;var j:real);
var t: real;
begin
 t := i;
 i := j;
 j := t;
end;
...
swap(a, b);
...
swap(x, y);
```

# *Absztrakció*

- Leggyakoribb fajtái:
  - Eljárás, az utasítás általánosítása
  - Függvény, a kifejezés általánosítása
- Más szintaktikus egységek általánosítása is lehetséges
  - programegység - modul
  - típusmegadás – polimorf típusok
- Az absztrakció elősegíti a mit/hogyan szétválasztását

# *Függvények*

- A kifejezés általánosítása, bemenő értékekből eredményt (kimenő értéket) állít elő
- Mellékhatás, külső értékek felhasználása
- A függvény fejléce tartalmazza a felhasználó számára fontos információt
- A függvény törzse csak az implementáló számára lényeges

# *Függvények*

- Az absztrakció és a hozzárendelés nem feltétlenül jár együtt

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

```
inc = \x -> x+1
map inc [1,2,3]
```

```
inc x = x+1
map inc [1,2,3]
```

# *Eljárások*

- Imperatív konstrukció
- Célja változók értékének beállítása, a program állapotának megváltoztatása
- Mellékhatások itt is ronthatják az átláthatóságot
- Javasolt az információáramlást a paramétereken keresztül megvalósítani



# *Paraméterek*

- A paraméterek teszik lehetővé az általánosítást és a specializációt
- Az absztrakció (pl. függvény definíció) során formális paramétereket használunk
- A specializáció (pl. függvény hívás) során aktuális paramétereket adunk meg
- Az aktuális paraméter értéke az argumentum
- Paraméterezés egyedekkel:
  - érték, típus, alprogram, (modul)

# *Paraméterátadás*

- Paraméterátadás: a formális és aktuális paraméter közötti megfeleltetés, az argumentum hozzá-rendelésének módja a formális paraméterhez
- Az információ áramlásának két iránya lehet: be-ki
- Paraméterátadás alapesetei:
  - Bemenő
  - Kimenő
  - Bemenő-kimenő
- Paraméterátadás másolással vagy hozzárendeléssel

# *Paraméterátadási módok*

- Érték szerinti átadás
- Cím szerinti átadás (referencián keresztül)
- Eredmény szerinti átadás
- Érték/eredmény szerinti átadás (másolaton keresztül, adatmozgató)
- (Név szerinti átadás ~ makró helyettesítés)

# *Paraméterátadás*

```
procedure p(var x:real; y:boolean);
...
p(z, true);
```

```
void f(int x, int& y, const int& z)
...
f(5, a, b);
```

```
f x y = if x = 0 then y else 1
...
f 1 (1/0)
```

# *Aliasing (többnevűség)*

- Egy változót többféleképpen is el lehet érni
- Rontja az áttekinthetőséget

```
procedure cfuse(var m, n: integer);
begin
 n := 1;
 n := m+n;
end;
...
i:=4;
cfuse(i, i);
cfuse(v[i],v[j]);
```

# *Megfelelési elv*

- Minden fajta deklarációhoz legyen egy megfelelő paraméterátadási mód

```
int i;
const int j= 5;
int& k= x;
const int& l= x;
const int* p;
int* const q= &v;
```

```
void f(int i, const int j);
void g(int& k, const int& l);
void h(const int* p, int* const q);
```

# *Kiértékelési sorrend*

- Aktuális paraméter kiértékelhető
  - Hívás előtt (mohó kiértékelés)
  - Hívás közben szükség szerint (lusta kiértékelés)
- Tisztán funkcionális nyelv esetén:
  - Ha mindkét módszer ad eredményt, akkor az megegyezik
  - A lusta kiértékelés adhat eredményt olyankor is, amikor a mohó nem

# *könyv*

- 7. fejezet (203-274)



# *Vezérlésátadás*

- Alap konstrukciók: egy bemenet – egy kimenet
  - Szekvencia
  - Feltételes
  - Iteratív
- Vezérlésátadás: eltérés a normál vezérléstől
  - Ugrás (goto)
  - Menekülés (escape)
  - Kivétel (exception)

*Goto*

NE

# *Menekülés*

- Egy bemenet – több kimenet
- Goto-val is megoldható
- C:
  - return – a legbelső függvényből ugrik ki
  - break – a legbelső blokkból ugrik ki
- Egyes nyelvekben valamely blokkhoz rendelt névre hivatkozhat a “break”

# *Kivételkezelés*

- Kivételes események kezelésére kezelő (handler) adható meg
- A kivétel bekövetkezésekor a hívási láncban felfelé kiválasztódik az első megfelelő kezelő
- A kezelő lefutása után az azt tartalmazó alprogram folytatódik
- Ha nincs kezelő, a program leáll
- Kérdés: a kezelő által kiváltott kivétellel mi legyen?

# *könyv*

- 3. fejezet, 3.8 főleg (75-78)
- 8. fejezet (277-313)

# Programozási nyelvek

Jelölésrendszer   Kifejezőerő  
Történelem   Elvárások

Fejlődés  
Procedurális  
Moduláris  
Objektum or. pr.

Típusok  
Elemi   Összetett   Rekurzív  
Típusosság   Típuskonstrukció

Programvezérlés  
Imperatív   Deklaratív  
Utasítások   Függvények  
Párhuzamos   Eseményvezérelt

Hozzárendelés  
Változó   Hatáskör   Élettartam   Blokk

Absztrakció  
Alprogramok   Paraméterátadás  
Vezérlésátadás

# *Párhuzamos programozás*

- Egyszerre több szálon történik a végrehajtás
- Végrehajtási szál: folyamat (process)
- Előnyei:
  - Természetes kifejezés mód
  - Sebességnövekedés megfelelő hardver esetén
- Hátrányai:
  - Bonyolultabb a szekvenciálisnál

# *Párhuzamos programozás*

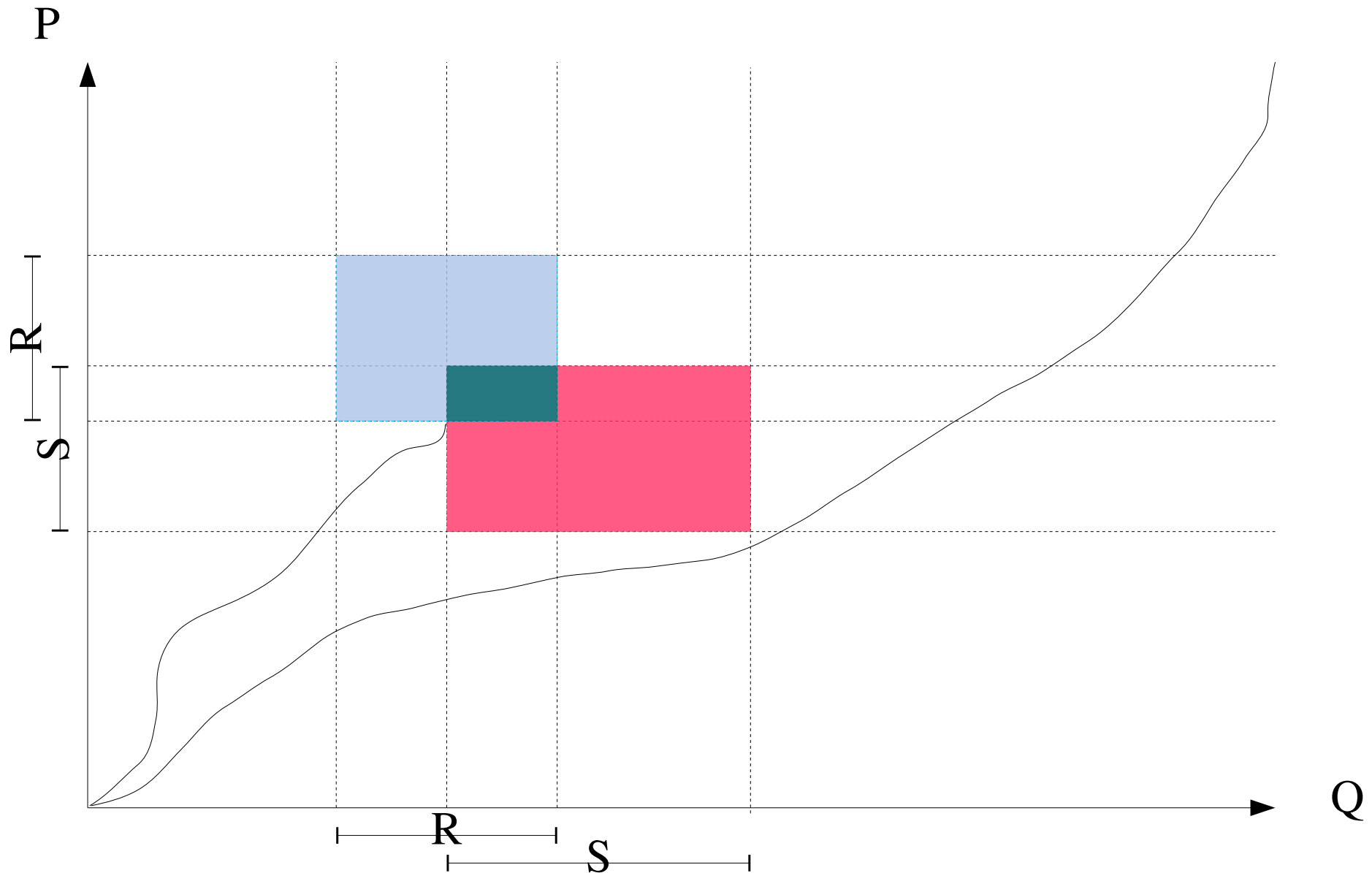
- Sokféle párhuzamos programozási modell van
- Közös problémák:
  - Adathozzáférés folyamatokból
    - Közös memória (shared memory)
    - Osztott memória (distributed memory) + kommunikáció
  - Folyamatok létrehozása, megszüntetése, kezelése
  - Folyamatok együttműködése (interakciója)
    - Független
    - Erőforrásokért versengő



# *Párhuzamos program*

- Sebességfüggő: a folyamatok relatív sebessége minden futáskor más lehet
- Nemdeterminisztikus: ugyanarra az inputra különböző output
- Holtpont (deadlock): kölcsönös egymásra várakozás
- Éhezés (starvation): Nincs holtpont, egy folyamat mégsem jut hozzá az erőforrásokhoz

# *Párhuzamos folyamatok*



# *Occam*

- Imperatív, folyamatok saját memóriával rendelkeznek, üzenetküldéssel kommunikálnak
- Occam program részei:
  - Változók
  - Folyamatok
  - Csatornák
- Szigorú formai követelmények

# *Occam - típusok*

- BOOL (TRUE, FALSE)
- BYTE (0..255), karakterek 'A'
- INT, INT16, INT32, INT64, 3(INT32)
- REAL32, REAL64, 1.7(REAL32)
- Egyetlen összetett típus a tömb:  
  [*méret*] *típus*
  - Stringek is tömbök "ABCD" :: [4]BYTE
- Változódeklaráció: típus változónév:
  - INT x:

# *Occam - csatornák*

- A csatorna két folyamat közötti adatátvitelre szolgál
  - Egyirányú
  - Küldő és fogadó is legfeljebb egy lehet
  - Biztonságos
  - Szinkron: A küldő és a fogadó bevárják egymást, megtörténik az adatátvitel, majd a küldő és a fogadó folytatódik
- Csatorna típusa: protokoll
  - `CHAN OF INT c:`

# *Occam - folyamatok*

- Folyamat élelciklusa:
  - Elindul
  - Csinál valamit
  - Befejeződik (terminál)
- Befejeződés helyett holtpontba is kerülhet, erre különös figyelmet kell fordítani
- Elemi és összetett folyamatok

# *Occam – elemi folyamatok*

- Üres utasítás                      SKIP
- Beépített holtpont              STOP
- Értékadás       $v := e$ 
  - Többszörös értékadás lehetséges       $x,$   
                                  $y := y, x$
- Input                       $c ? v$
- Output                       $c ! e$

# *Occam - szekvenciális végrehajtás*

- Szekvenciális végrehajtás SEQ

SEQ  
P  
Q  
...  
R

SEQ  
x := 1  
y := x+1  
z := y-2

SEQ  
x := 1  
SEQ  
y := x+1  
z := x-2  
x := y+z



# *Occam - feltételes vezérlés*

- Feltételes vezérlés IF

IF

c1

P1

c2

P2

...

cN

PN

IF

x < 0

z := -1

x = 0

z := 0

x > 0

z := 1

IF

x = 0

z := 0

IF

x < 0

z := -1

x > 0

z := 1

# *Occam - ciklusszervezés*

- Ciklusutasítás                      WHILE

```
WHILE c
 P
 INT x, y:
 SEQ
 x, y := 0, 0
 WHILE x < 10
 SEQ
 y := y + x
 x := x + 1
```

# *Occam - replikáció*

- Az összetett folyamatok többségét lehet replikált formában használni (eddig SEQ, IF)

– XXX v=e1 FOR e2

```
SEQ i=1 FOR 3
 x[i-1] := i
```

```
SEQ
 x[0] := 1
 x[1] := 2
 x[2] := 3
```

```
IF i=0 FOR 3
 x[i] > 5
 z := i
```

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

# *Occam - párhuzamosság*

- Párhuzamos vezérlés

PAR

```
PAR
 P
 Q
 ...
 R
```

```
CHAN INT c:
PAR
 c ! 5
 INT x:
 c ? x
```

```
VAL INT s IS 5:
[s]INT v:
SEQ
 PAR i=0 FOR s
 v[i] := 0
 SEQ i=0 FOR s
 c ! v[i]
```

# *Occam - folyamat komponensek*

```
CHAN BYTE c:
PAR
```

```
 BYTE b:
```

```
 SEQ -- A
```

```
 c ? b
```

```
 c ? b
```

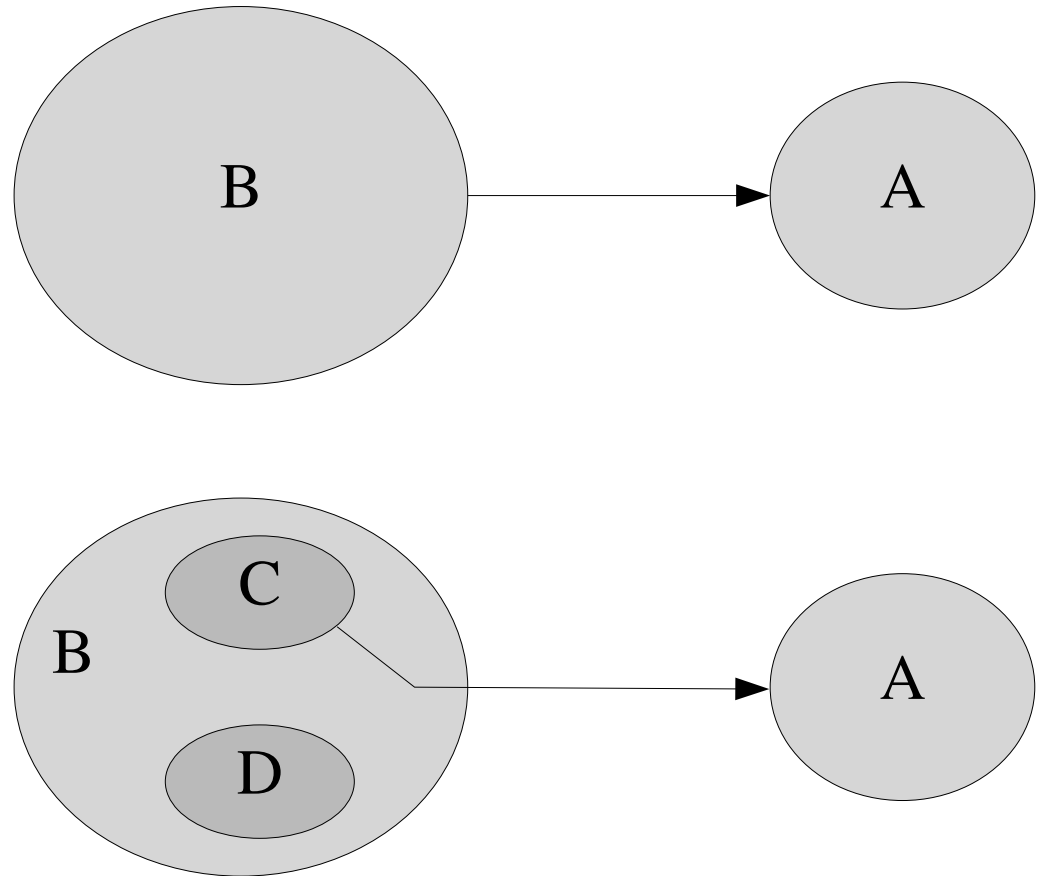
```
 SEQ -- B
```

```
 c ! '1'
```

```
 PAR
```

```
 c ! '2' -- C
```

```
 SKIP -- D
```



# *Occam – névvel ellátott folyamatok*

```
PROC p(VAL INT a, INT b, CHAN INT c)
 process
```

```
:
```

```
PROC q(VAL []BYTE v, []BYTE w, []CHAN BYTE
x)
```

```
 INT s:
```

```
 SEQ
```

```
 . . .
```

```
 s := SIZE x
```

```
 . . .
```

```
:
```

# *Occam - aliasing*

- Az Occam nem engedi meg az aliasingot
- Minden változónak minden pillanatban csak egy neve lehet

```
PROC nocfuse(INT m, INT n)
```

```
 SEQ
```

```
 n := 1;
```

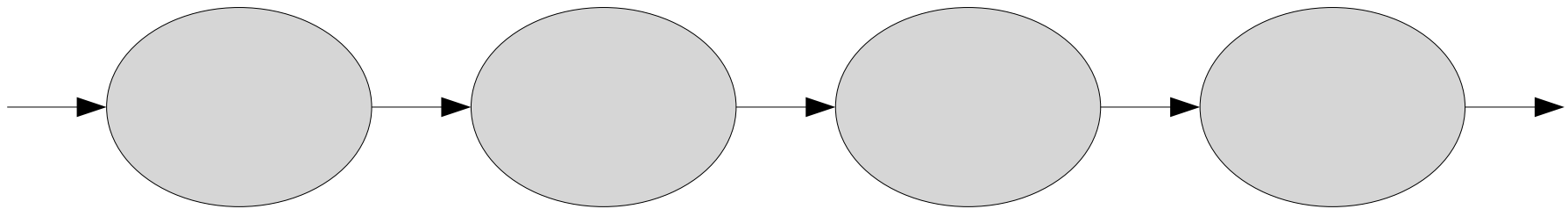
```
 n := m+n;
```

```
 :
```

```
 ...
```

```
nocfuse(i, i) – fordítási hiba
```

# *Párhuzamosság csővezetékekkel*

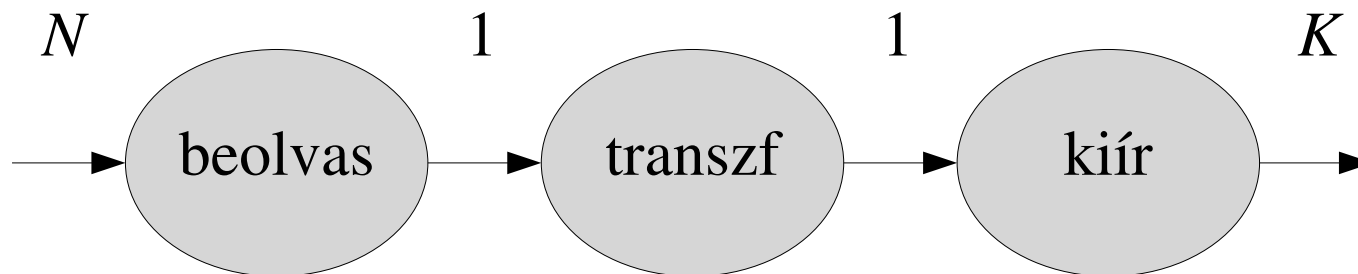


- Minden adatelemet minden folyamat feldolgoz
- Különböző adatelemek a feldolgozás különböző fázisaiban lehetnek
- Sebesség szempontjából a kiegyensúlyozott fázisok a legjobbak



# *Struktúraütközés*

- $N$  méretű rekordok beolvasása, elemek transzformációja,  $K$  méretű rekordok kiírása
- Legtermészetesebb kifejezés mód a párhuzamosság, 3 elemű csővezeték



# *Occam - struktúraütközés*

```
PROC main(CHAN BYTE inp, out, err)
```

```
 PROC be(CHAN BYTE inp, out) ...
```

```
 PROC tform(CHAN BYTE inp, out) ...
```

```
 PROC ki(CHAN BYTE inp, out) ...
```

```
 CHAN BYTE bt, tk:
```

```
 PAR
```

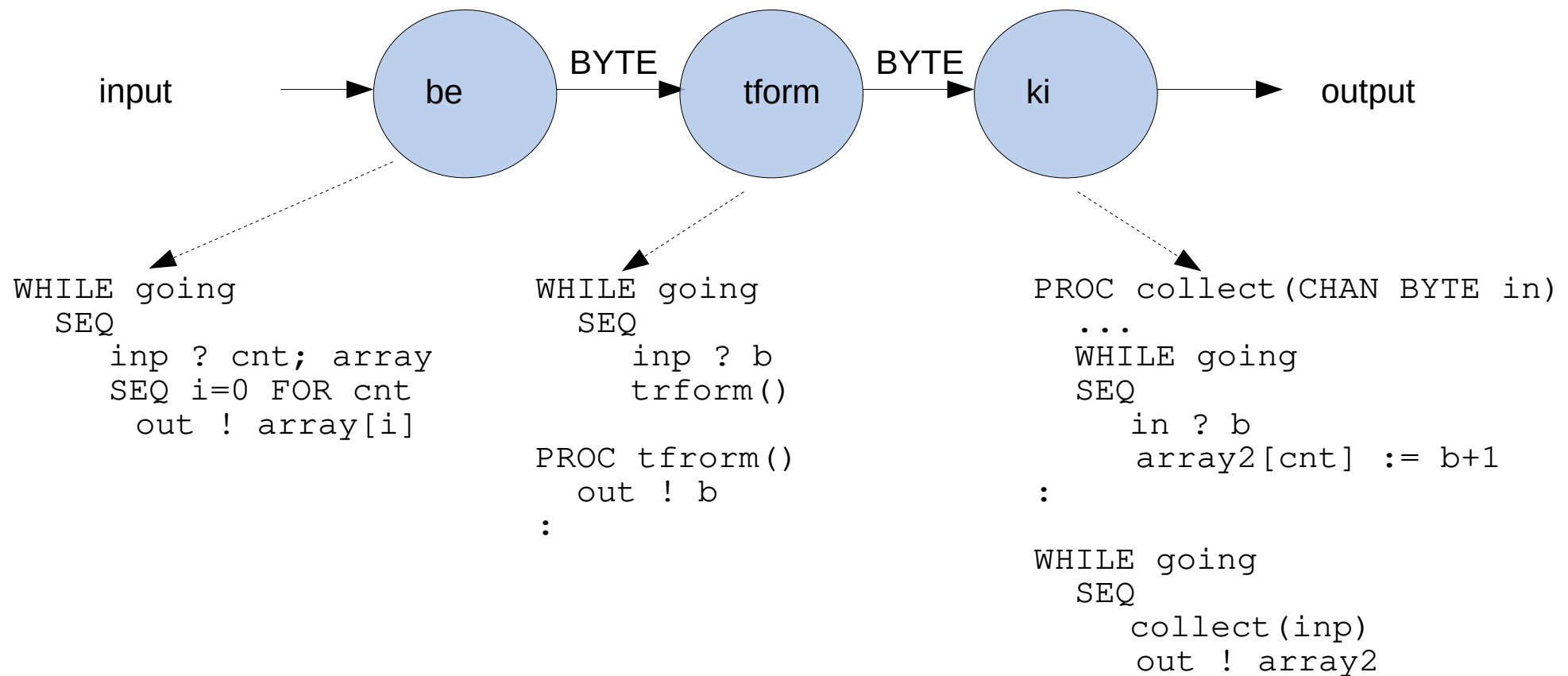
```
 be(inp, bt)
```

```
 tform(bt, tk)
```

```
 ki(tk, out)
```

```
:
```

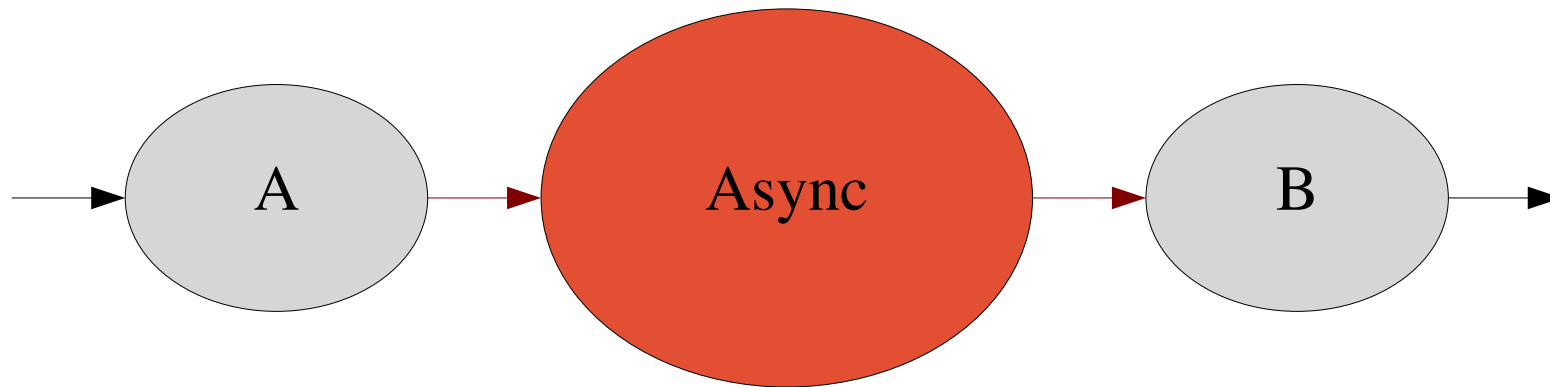
# *Occam - struktúraütközés*



# *Aszinkron kommunikáció*

- Aszinkron kommunikáció esetén a küldő nem vár a fogadóra
- Lehetnek már elküldött, de még nem fogadott üzenetek (pufferelt csatorna)
- A csatornkapacitás határozza meg, mennyi úton lévő üzenet lehet
  - Korlátlan
  - Korlátos, ezt occamban is meg lehet valósítani

# *Occam - aszinkron kommunikáció*



```
PROC async(CHAN BYTE inp, out)
 WHILE TRUE
 BYTE b:
 SEQ
 inp ? b
 out ! b
 :
```

# *Occam – aszinkron kommunikáció*

```
PROC async(CHAN BYTE inp, out, VAL INT N)
 PROC buff(CHAN BYTE inp, out)
 WHILE TRUE
 BYTE b:
 SEQ
 inp ? b
 out ! b
 :
 [N-1]CHAN BYTE c:
 PAR
 buff(inp, c[0])
 PAR i=1 FOR N-2
 buff(c[i-1], c[i]
 buff(c[N-2], out)
 :
```

# *Occam - várakozás*

- Ha több forrásból érkezhetsz üzenet

ALT

ALT

g1

p1

g2

p2

...

gN

pN

ALT

c ? x

y := y+x

d ? x

z := z+x

ALT

c ? x

y := y+x

ALT

d ? x

z := z+x

e ? x

w := w+x

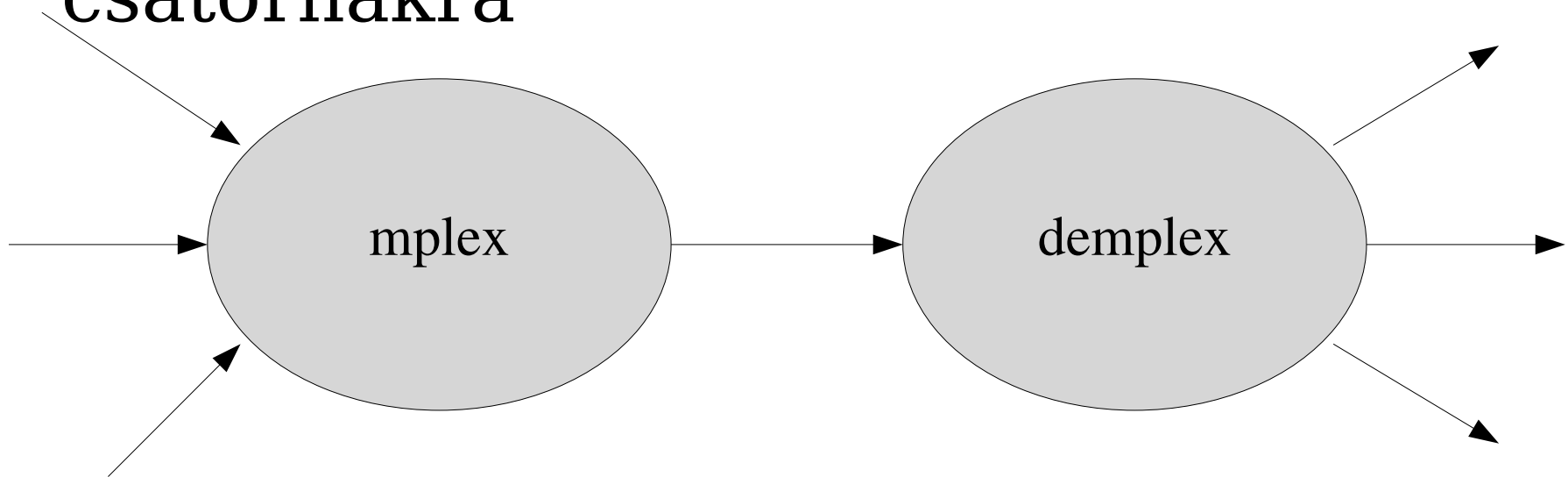
# *Occam - ALT örök*

- Általános alak: *cond* & *c* ? *v*
- Ha *cond* nem szerepel, akkor TRUE
- Csak azokat a csatornákat veszi figyelembe, ahol a feltétel TRUE
- Speciális ör forma: TRUE & SKIP
  - PRI ALT formával használatos



# *Multiplexer*

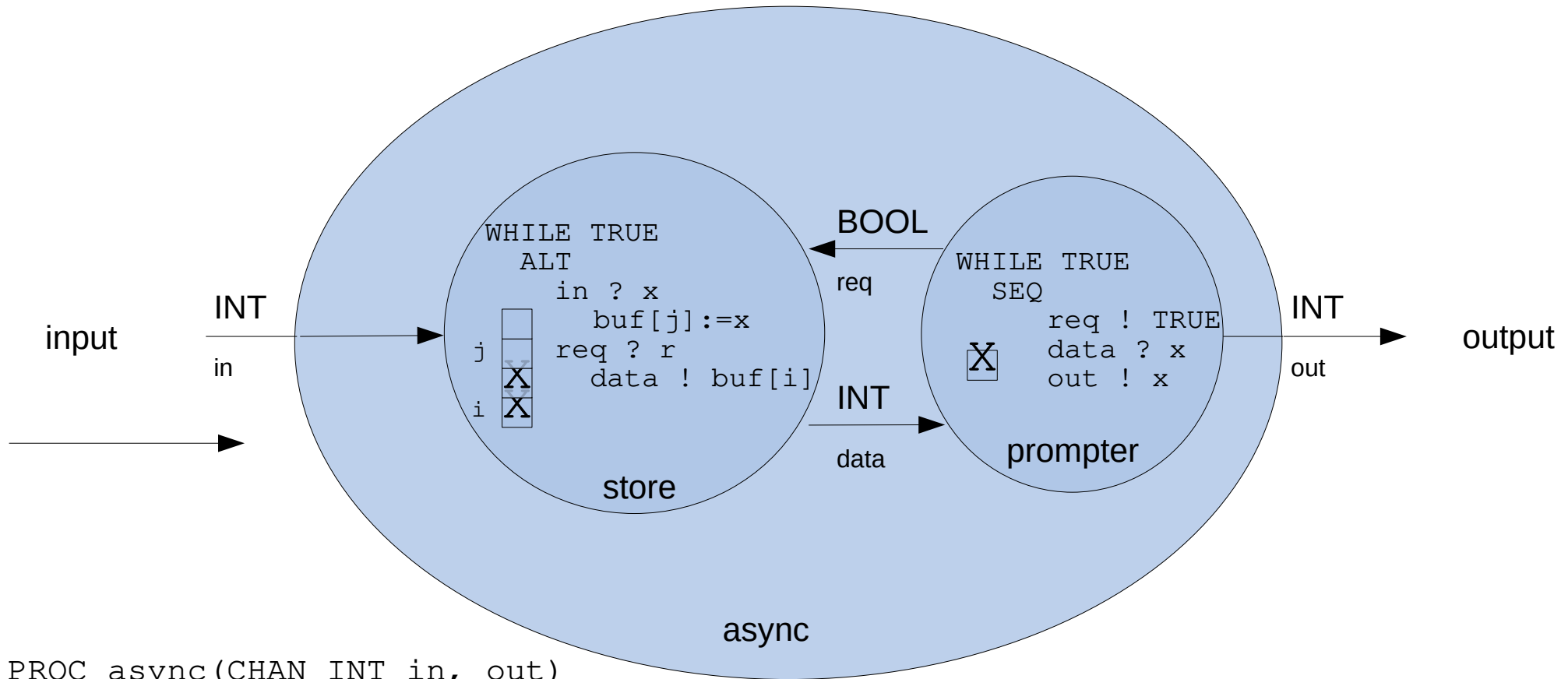
- Multiplexer: több forrásból érkező adatok átirányítása egy kimenő csatornára
- Demultiplexer: Bejövő csatornán érkező adatok szétválogatása kimenő csatornákra



# *Occam - multiplexer*

```
PROC mplex([]CHAN INT inp, CHAN INT out)
 WHILE TRUE
 ALT i=0 FOR SIZE inp
 INT b:
 inp[i] ? b
 SEQ
 out ! i
 out ! b
 :
```

# *Occam - pufferelt csatorna*



```
PROC async(CHAN INT in, out)
 . . .
 PAR
 store(in, data, req)
 prompter(data, out, req)
 :
```

# *Occam – változók használata*

- Változók hatásköre kiterjedhet PAR összetett folyamatra:
  - Ha egy változót csak olvasnak, akkor szerepelhet több folyamatban is
  - Ha egy változóba egy folyamat beír ( $:=$ ,  $?$ ), akkor az a változó nem szerepelhet másik folyamatban
  - Több egy változónak számít, kivéve ha meg lehet állapítani, hogy különböző folyamatok különböző elemeket módosítanak

# *Occam – többszörös értékadás*

|                                      |                      |
|--------------------------------------|----------------------|
| $v_1, \dots, v_N := e_1, \dots, e_N$ | -- $t_1, \dots, t_N$ |
|                                      | SEQ                  |
|                                      | PAR $i=1$ FOR $N$    |
|                                      | $t_i := e_i$         |
|                                      | PAR $i=1$ FOR $N$    |
|                                      | $v_i := t_i$         |

- Illegális:

- $x, x := 3, 4$
- $i, v[i] := 3, 4$

# *Occam - függvények*

- Függvény: érték folyamat

```
BOOL FUNCTION isnull(VAL BYTE x) IS (x='0'):
```

```
INT, INT FUNCTION minmax(VAL INT x, VAL INT y)
 INT a, z:
 VALOF
 IF
 x < y
 a, z := x, y
 TRUE
 a, z := y, x
 RESULT a, z
:
```

a, b := minmax( 7, 1)

# *Occam - szelekció*

- Esetszétválasztás                      CASE

```
CASE e
 l1,...
 p1
 l2,...
 p2
 ...
 lN,...
 pN
```

```
CASE letter
 'a','e','i','o','u'
 vowel := TRUE
ELSE
 vowel := FALSE
```

# *Terheléselosztás*

- Több processzor esetén akkor érhetjük el a legnagyobb sebességnövekedést, ha a processzorok folyamatosan dolgoznak és egyszerre végeznek
- A processzorok teljesítményét figyelembe véve a munkát fel kell osztani ugyanakkora terhelést jelentő részekre
  - Statikus terheléselosztás: futás előtt
  - Dinamikus terheléselosztás: futás közben



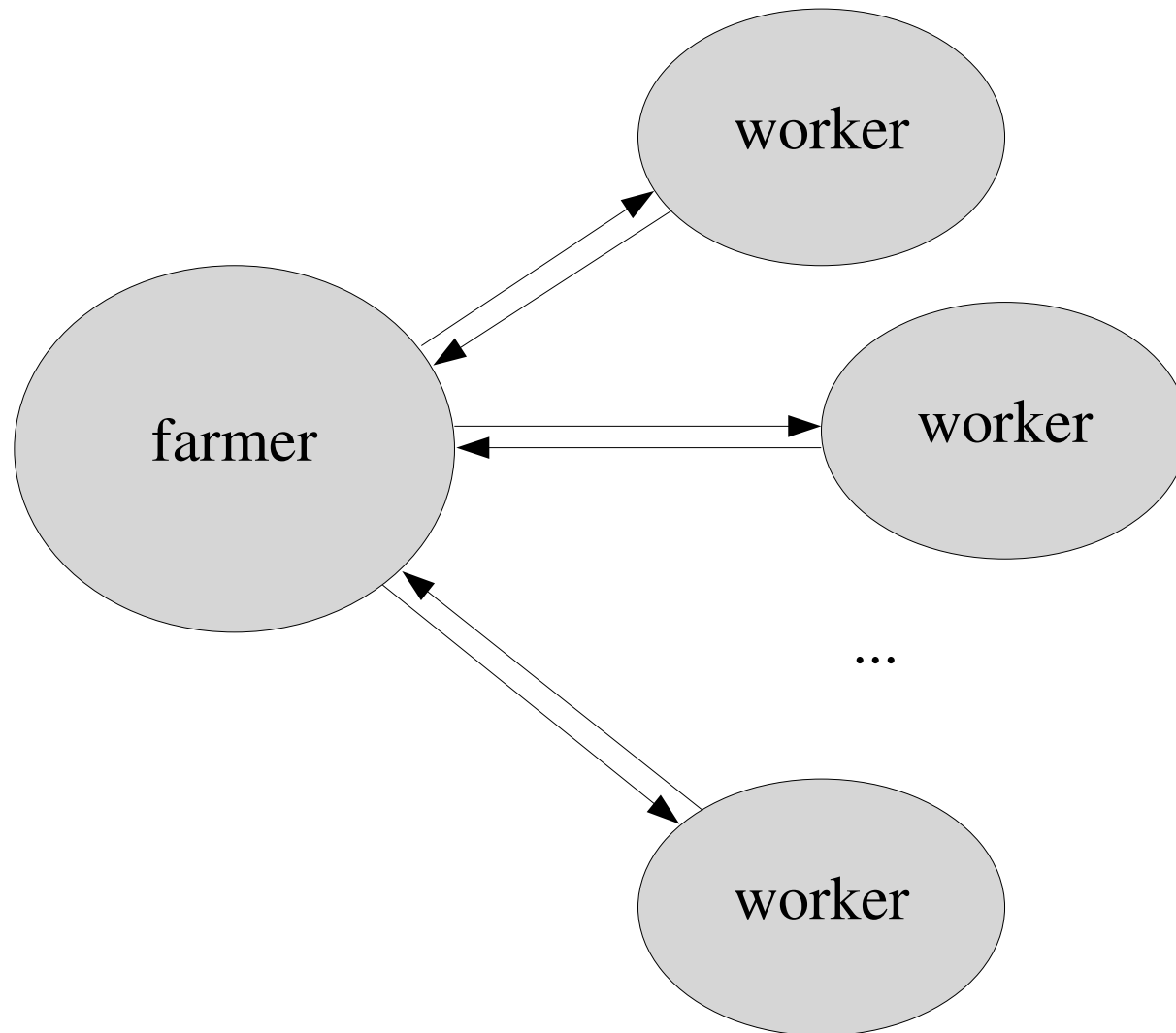
# *Terheléselosztás - munkafelosztás*

- Funkcionális párhuzamosság: különböző folyamatok különböző feladatot végeznek, a szükséges adatokat átadják egymásnak. Lásd csővezeték.
- Adatpárhuzamosság: a folyamatok ugyanazt a feladatot végzik el, az adatokat fel kell osztani közöttük (particionálás)

# *Processzor farm*

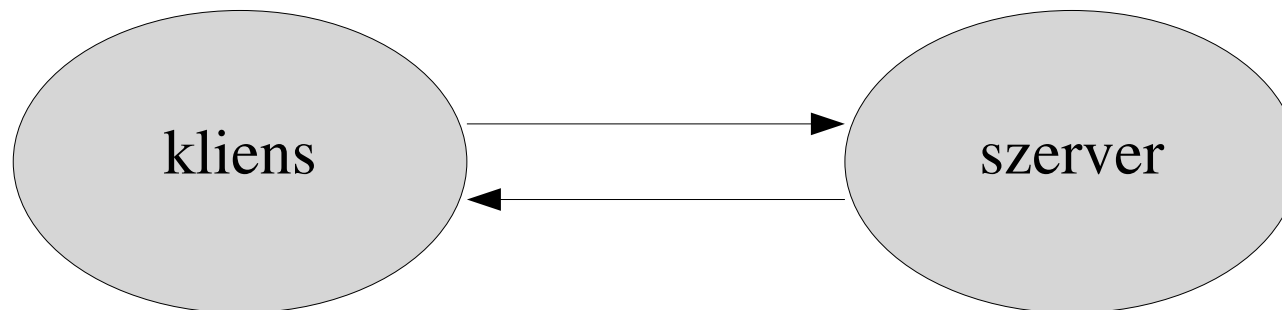
- Az adatpárhuzamosság leggyakoribb megvalósítási formája, 1 farmer és több worker (dolgozó) folyamatból áll.
- A farmer feladata a feldolgozandó adatok particionálása és az eredmények begyűjtése
- A dolgozók feladata az adatok fogadása, feldolgozása és visszaküldése
- Egy nagyságrenddel több részfeladat kell, mint amennyi dolgozó van

# *Occam - processzor farm*



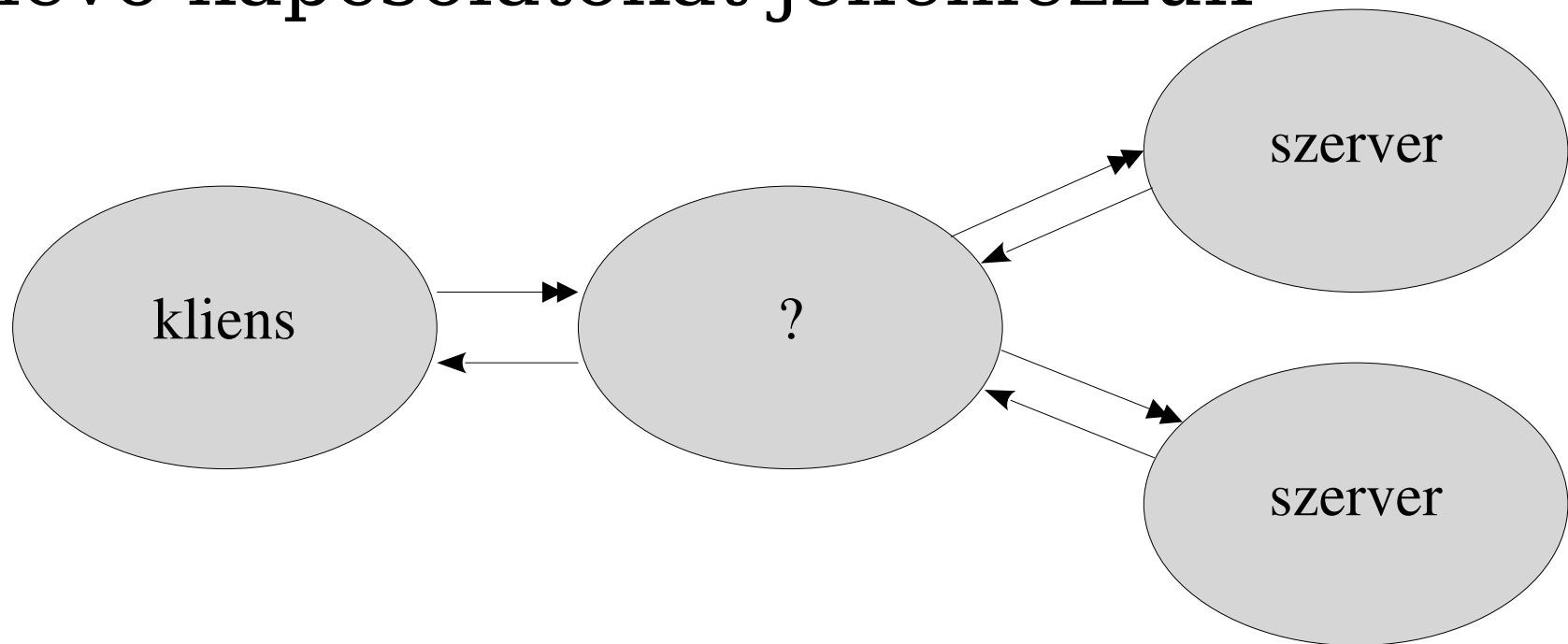
# *Kliens - szerver rendszer*

- Osztott rendszerek gyakori felépítési módja
- A szerver passzív. Kérésre vár, majd feldolgozás után válaszol
- A kliens aktív. Kezdeményezi a kapcsolat-felvételt a szerverrel, majd vár a válaszra



# *Kliens - szerver rendszer*

- Egyes folyamatok kliensként is és szerverként is viselkedhetnek
- Nem a folyamatokat, hanem a közöttük lévő kapcsolatokat jellemezzük



# *Occam – kliens-szerver komponens*

```
PROC cs([]CHAN R sreq, []CHAN A sans,
 []CHAN R creq, []CHAN A cans)
 WHILE going
 ALT i=0 FOR SIZE sreq
 R d:
 sreq[i] ? d -- kérés
 SEQ
 ...
 creq[j] ! x -- kérés szerverhez
 cans[j] ? y -- válasz szervertől
 ...
 sans[i] ! a – válasz
 :
```

# *Folyamatok közös memóriában*

- Osztott modellben a csatornák szinkronizálják is a folyamatokat
- Közös memória esetén is szükség van szinkronizációra az erőforrások kezelésekor

$y := y + 3$

$y := y + 5$

---

$r <- y$

$r += 3$

$y <- r$

$r <- y$

$r += 5$

$y <- r$

# *Kölcsönös kizárás*

- Kritikus szekció: közös erőforrást kezelő kódrészlet egy folyamatban
- Kölcsönös kizárás: egyszerre csak egy folyamat lehet kritikus szekcióban
- Szemafor: kölcsönös kizárás megvalósítására szolgáló eszköz
  - Egész értékű változó, várakozó sor
  - Műveletek: `init`, `wait`, `signal`



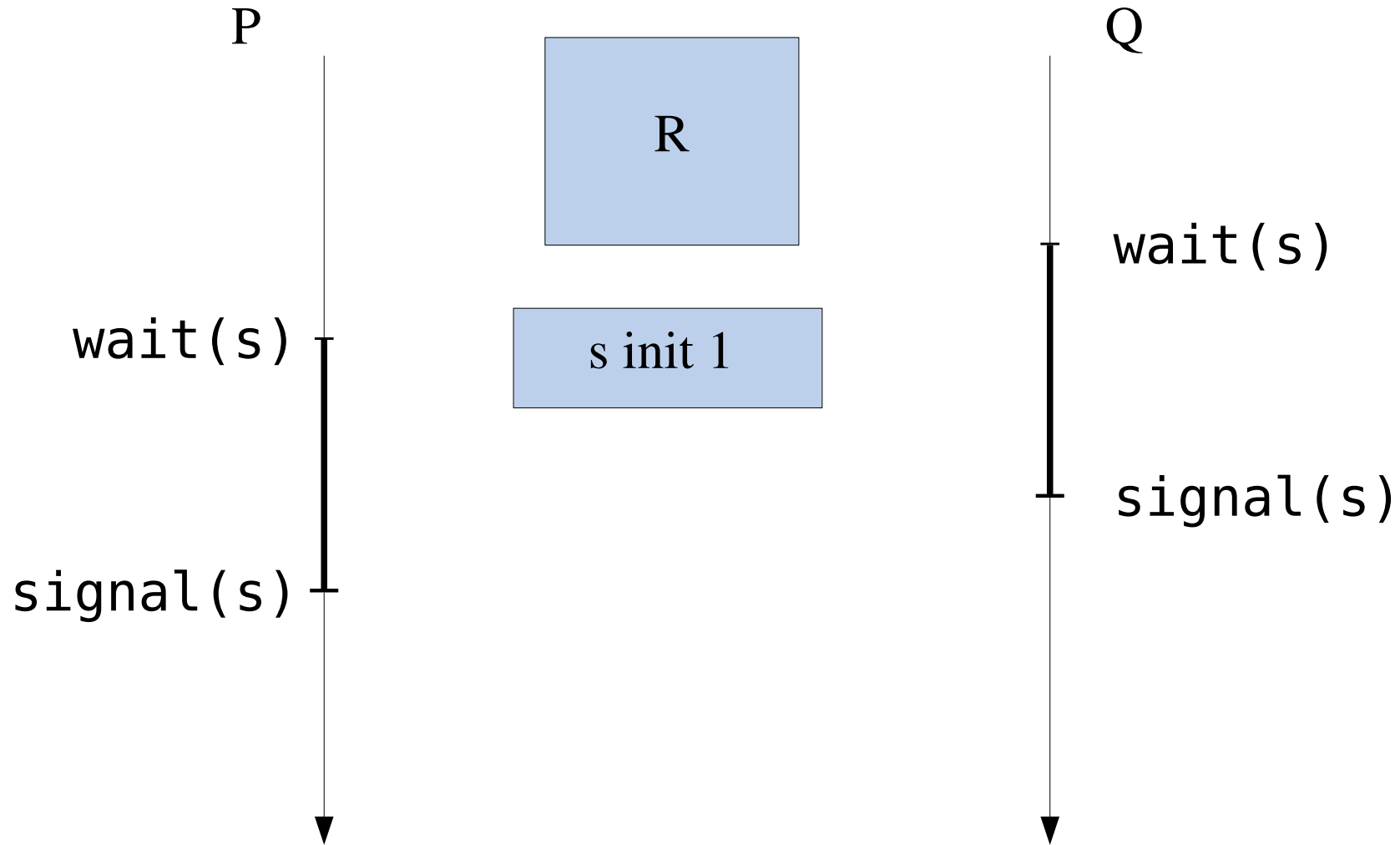
# *Szemafor*

- A szemaforműveletek oszthatatlanok, vagyis egy szemaforon egyszerre legfeljebb egy folyamat hajthatja végre

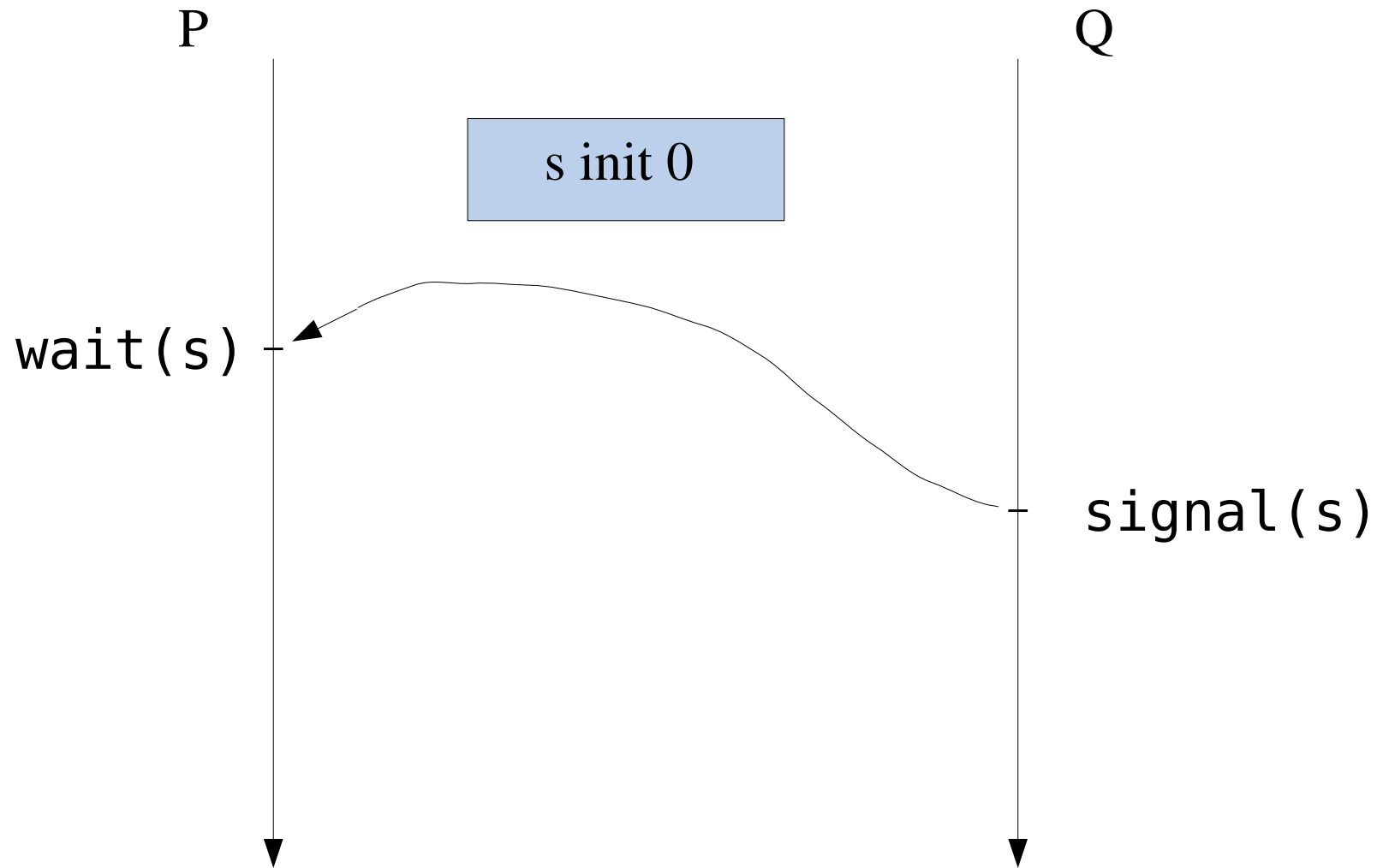
```
wait(s) {
 if (s > 0)
 s= s-1;
 else
 suspend(s);
}
```

```
signal(s) {
 if
 (waiting(s))
 release(s);
 else
 s= s+1;
}
```

# *Kölcsönös kizárás szemaforral*

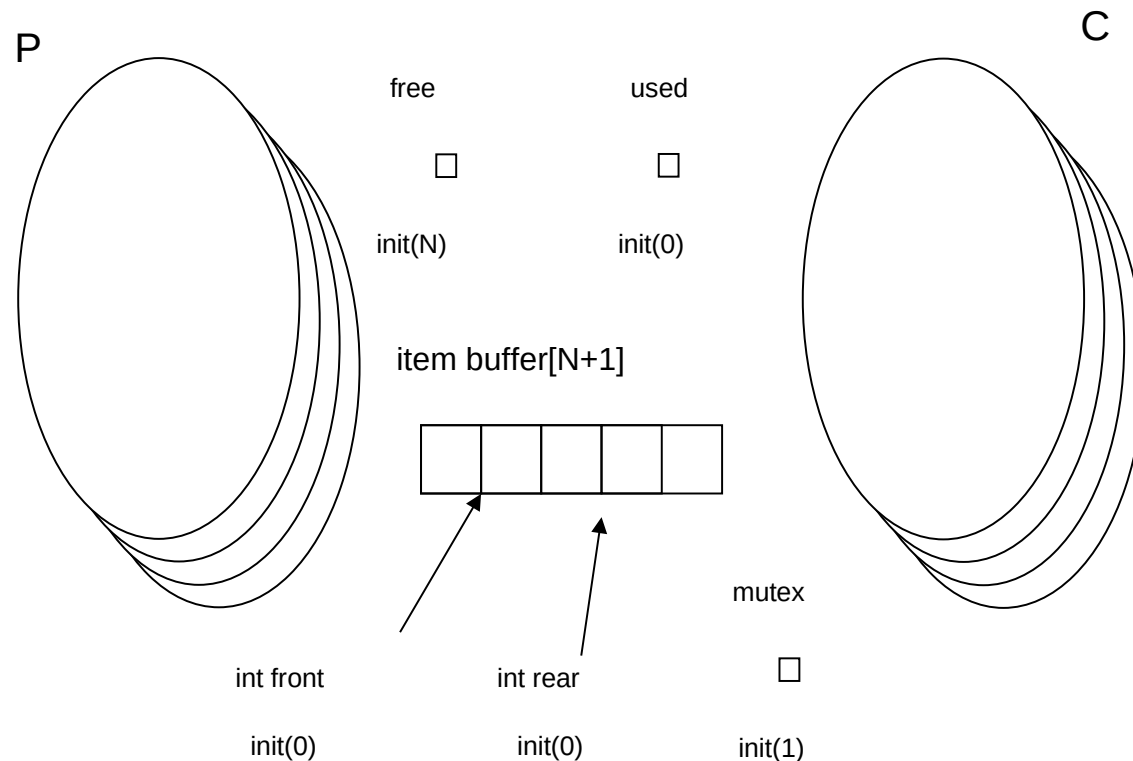


# *Szinkronizáció szemaforral*



# *Termelő-Fogyasztó probléma*

- Bináris szemafor (mutex) a terméktömb hozzáféréshez
- Két további szemafor a termelő (used) és a fogyasztó (free) szinkronizálásához



# *Implementació*

```
class Queue {
 private ITEM data[N]= ...;
 private Semaphore mutex(1);
 private Semaphore used(0);
 private Semaphore free(N);

 public void deposit(ITEM x) {
 free.wait();
 mutex.wait();
 // x -> data
 mutex.signal();
 used.signal();
 }

 public ITEM extract(ITEM x)
 {
 used.wait();
 mutex.wait();
 // x -> data
 mutex.signal();
 free.signal();
 }
}
```

# *Monitor*

- Struktúrált folyamatszinkronizációs eszköz
- Egy objektum, aminek:
  - Privát adata az erőforrás,
  - és egy beépített zárja van az erőforráshoz hozzáférő metódusok atomiságának biztosításához.

# *Kölcsönös kizárás monitorral*

- Atomi műveletek a monitoron belül

```
monitor Queue {

 private ITEM data[N]= ...;

 public void deposit(ITEM x) {
 // x -> data
 }

 public ITEM extract() {
 // x <- data
 return x;
 }
}
```

# *Szinkronizáció monitorral*

- Feltételváltozó:
  - A monitoron belül használható,
  - a kölcsönös kizárás zárjához kötődik.
- Egy olyan objektum, ami:
  - Privát várakozási sort kezel a folyamatokhoz,
  - Műveletei:
    - cwait: adott folyamat várakozási sorba helyezése és zár feloldása,
    - csignal: egy várakozó folyamat felélesztése



## *csignal stratégiák*

- `signal&continue` – `csignal`-t hívó folytatja futását, a felélesztett várakozik
- `signal&wait` – a felélesztett folytatja futását, a hívó várakozik (tradicionális)
- `signal&exit` – a felélesztett folytatja futását, a hívó befejeződik

# *Szinkronizáció monitorral*

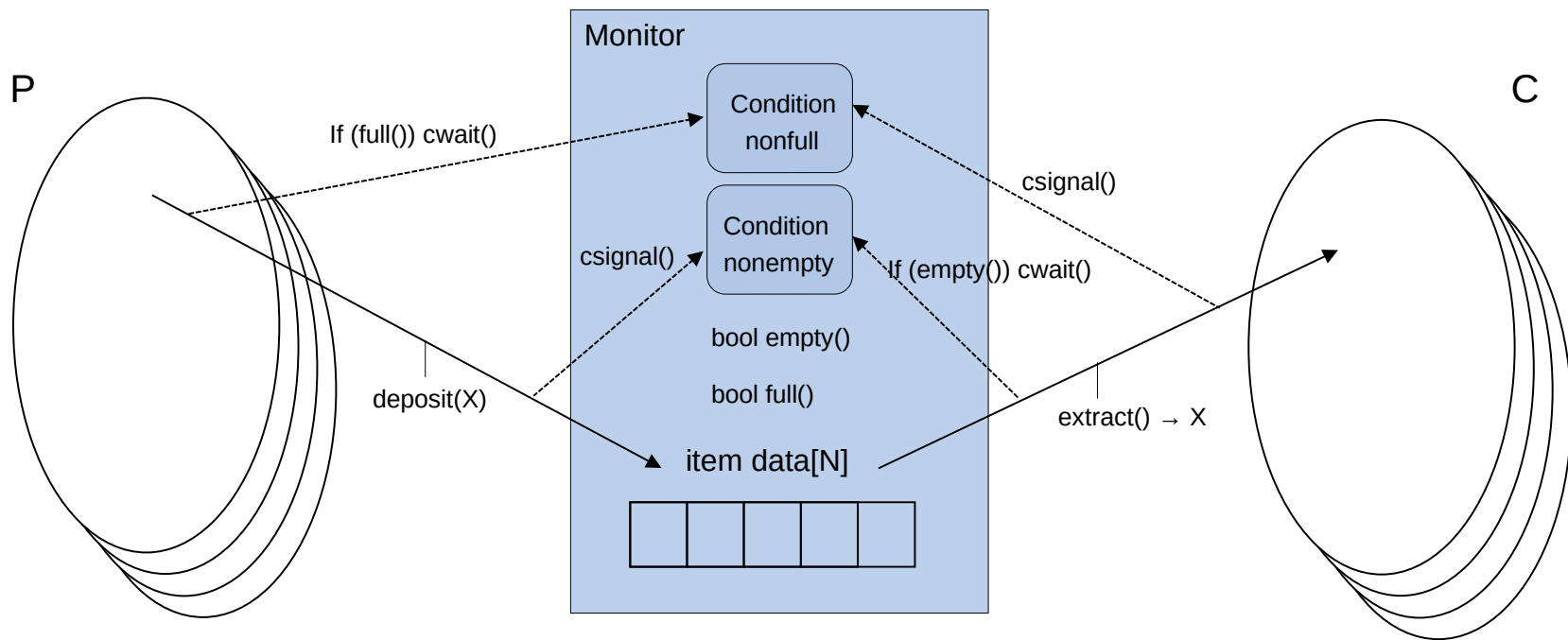
```
monitor Queue {
 private ITEM data[N]= ...;
 private Condition nonempty, nonfull;

 public void deposit(ITEM x) {
 if (full())
 nonfull.cwait();
 // x -> data
 nonempty.csignal();
 }

 public ITEM extract() {
 if (empty())
 nonempty.cwait();
 // x <- data
 nonfull.csignal();
 return x;
 }
}
```

# *Termelő-Fogyasztó probléma*

- Monitor használata két feltételváltozóval



# *Párhuzamosság Java-ban*

- A Thread osztály példányosításával, vagy

```
class ExThread extends Thread {
 ExThread(String name) {
 super(name);
 }

 public void run() {
 ...
 }
}
```

- A Runnable interfész implementálásával

```
class ImThread implements Runnable {
 public void run() {
 ...
 }
}
```

hozhatunk létre szálakat.

# *Szálak kezelése*

```
class Thread implements Runnable {
 public Thread(String name);
 public Thread(Runnable target, String name);
 public String getName();

 public void run(); // empty body

 public void start();
 public void join();
 public void interrupt();

 public static Thread currentThread();
 public static void sleep(long millis);
 public static void yield();
}
```

# *Szálak létrehozása*

```
public class ThreadDemo {
 public static void main(String[] args) {
 ExThread t1= new ExThread("ExThread");

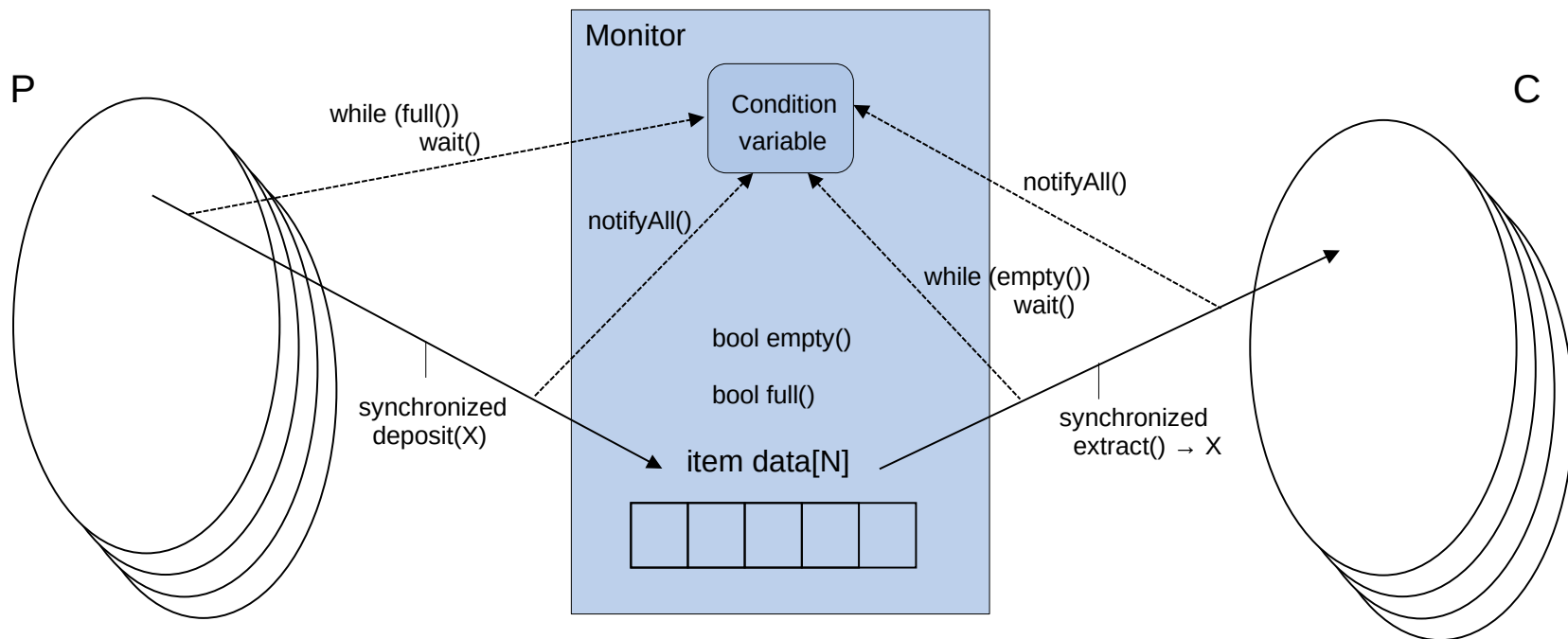
 Thread t2= new Thread(new ImThread(),
"ImThread");

 t1.start();
 t2.start();
 ...
 try {
 t1.join();
 t2.join();
 }
 catch (InterruptedException e) {
 ...
 }
 ...
 }
}
```

# *Monitor Java-ban*

| Monitor                                                                                                                                            | Java                                                                                                                                                                                      |
|----------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| private data<br>initialization<br>monitor procedures<br>lock for monitor procedures<br>condition variables<br>cwait operation<br>csignal operation | private instance variables/constants<br>constructor<br>synchronized blocks & methods<br>lock in class Object<br>1 implicit condition variable<br>wait method<br>notify, notifyAll methods |

# *Termelő-fogyasztó Java-ban*





# *Új interfészek Java 1.5-től*

- Több feltételváltozó kezeléséhez:

```
public interface Lock // implemented by class
ReentrantLock
{
 void lock();
 void unlock();

 Condition newCondition();
 ...
}
```

```
public interface Condition
{
 void await() // monitor: cwait, Object: wait
 throws InterruptedException;
 void signal(); // monitor: csignal, Object: notify
 void signalAll(); // Object: notifyAll
}
```

# *Több feltételváltozó használata*

```
class X {
 private final ReentrantLock _lock= new ReentrantLock();
 private final Condition _cond1= _lock.newCondition();
 private final Condition _cond2= _lock.newCondition();
 // ...
 public void m() { // NOT synchronized
 _lock.lock();
 try { // ... method body
 _cond1.await();
 // condition variable operations only after locking
 _cond2.signal();
 // ...
 }
 finally { // make sure lock is released
 _lock.unlock()
 }
 }
}
```

# *könyv*

- 14. fejezet (505-559)

# Programozási nyelvek

Jelölésrendszer   Kifejezőerő  
Történelem   Elvárások

Fejlődés  
Procedurális  
Moduláris  
Objektum or. pr.

Típusok  
Elemi   Összetett   Rekurzív  
Típusosság   Típuskonstrukció

Programvezérlés  
Imperatív   Deklaratív  
Utasítások   Függvények  
Párhuzamos   Eseményvezérelt

Hozzárendelés  
Változó   Hatáskör   Élettartam   Blokk

Absztrakció  
Alprogramok   Paraméterátadás  
Vezérlésátadás

Programozási paradigmák  
Objektum orientált (Imperatív)  
Funkcionális (Deklaratív)  
Logikai (Deklaratív)  
Párhuzamos