

# Programozási nyelvek és módszerek

Leichner Dávid

2016. június 5.

## Tételsor

Minden tétel esetén tisztázni kell az adott nyelvi elem általános meghatározását, valamint az adott nyelvi eszköz célját. Továbbá a lehetséges kérdéseket (amik egy adott nyelv esetén vizsgálandók) is meg kell adni.

1. Szoftverek minőségi szempontjai, moduláris tervezés, a modularitás alapelvei. Programkönyvtárak tervezési elvei
2. Neumann modell, LMC, generatív grammatikák és formális nyelvek alapfogalmai
3. Nyelvi elemek, a programozási nyelvek lexikális elemei Vezérlési szerkezetek, utasítások, értékadás, szekvencia és blokk utasítás, feltétel nélküli vezérlésátadás, elágazási szerkezetek, ciklusszerkezetek (példák: Java, C++, választott nyelv\*)
4. Típusok 1. Típusspecifikáció, típusmegvalósítás, típusosztályok, skalár típusosztály típusai a programozási nyelvekben, (példák: Java, C++, Ada/Pascal)
5. Típusok 2. Pointer és referencia típusok, típusok ekvivalenciája (példák: C++, Ada, Eiffel) Típuskonstrukciók – tömbök, direkt szorzat, unió, halmaz támogatása (példák: Java, C++, Ada/Pascal)
6. Alprogramok Eljárások és függvények, paraméterek fajtái, átadás-átvételi módok, túlterhelés, rekurzió (példák: Java, C++, Ada/SWIFT)
7. A kivételkezelés Alapfogalmai, kivételek kiváltása, terjedése, kezelése, specifikálása, kivételosztályok (példák: Java, C++, Eiffel)
8. Absztrakt adattípusok a programozási nyelvekben Procedurális- és adatabsztrakciós megközelítés, elvárások és eszközök (példák: Java, C++, Ada) Sablonok – Típussal, alprogrammal való paraméterezés, példányosítás, sablon-szerződés modell (példák: Java, C++, Ada, Eiffel)
9. Programok helyessége és a támogató nyelvi eszközök (Hoare módszer, levezetési szabályok is) (példák: Eiffel, Java-eszközök, Code Contracts Library)
10. Objektumorientált programozás Osztályok és objektumok, objektum létrehozása, inicializálása, példányváltozó, példánymetódus, osztályváltozó, osztálymetódus, társítási kapcsolatok, Öröklődés, polimorfizmus, dinamikus kötés, megbízható átdefiniálás leszármazottakban (példák: Java, C++, Smalltalk/Objective-C) A többszörös öröklődés problémái, lehetséges megoldásai, az interfészek fogalma, használata (példák: Java, C++, Objective Pascal, Eiffel)
11. Párhuzamosság Alapfogalmak, Flynn-féle modell, kommunikációs modellek, semafor, monitor, fontos nyelvi elemek (példák: Java, Ada, Go, C++11)
12. A funkcionális és logikai programozás alapjai Példákkal

\*választott nyelv: nem a C nyelvcsaládba tartozó, például Python, Basic, Pascal, Ada, Eiffel, SWIFT, stb. . . , illetve Objective-C

# Tartalomjegyzék

<b>1. Szoftverek minőségi szempontjai, moduláris tervezés, a modularitás alapelvei. Programkönyvtárak tervezési elvei</b>	<b>5</b>
1.1. Szoftverek minőségi szempontjai:	5
1.2. Moduláris tervezés	6
1.2.1. Moduláris dekompozíció	6
1.2.2. Moduláris kompozíció	6
1.3. A modularitás alapelvei	6
1.4. Programkönyvtárak tervezési elvei	7
1.4.1. Újrafelhasználható programkönyvtárak	7
1.4.2. Biztonság	8
1.4.3. Dokumentáció	8
1.4.4. Karbantartás	9
1.4.5. Memóriakezelés	9
<b>2. Neumann modell, LMC, generatív grammatikák és formális nyelvek alapfogalmai</b>	<b>10</b>
2.1. Neumann modell	10
2.2. A "Little Man Computer"	11
2.3. Generatív grammatikák	13
2.4. Formális nyelvek alapfogalmai	14
<b>3. Nyelvi elemek, a programozási nyelvek lexikális elemei Vezérlési szerkezetek, utasítások, értékadás, szekvencia és blokk utasítás, feltétel nélküli vezérlésátadás, elágazási szerkezetek, ciklusszerkezetek (példák: Java, C++, választott nyelv*)</b>	<b>16</b>
3.1. Nyelvi elemek, a programozási nyelvek lexikális elemei	16
3.1.1. Azonosítók	18
3.1.2. Literálok	20
3.2. Vezérlési szerkezetek	20
3.2.1. Utasítások, értékadás	20
3.3. Szekvencia	22
3.4. Blokk utasítások	23
3.5. Feltétel nélküli vezérlésátadás	23
3.6. Elágazás	24
3.7. Ciklusok	28
<b>4. Típusok 1</b>	<b>33</b>
4.1. Típusspecifikáció	33
4.2. Típusmegvalósítás	33
4.3. Típusosztályok	34
4.4. A skalár típusok osztálya	35
<b>5. Típusok 2</b>	<b>42</b>
5.1. Pointer és referencia típusok	42
5.2. Típuskonstrukciók	44
5.3. Tömb típusok	45
5.4. Halmaz	50
5.5. Mikor ekvivalens két típus?	50

<b>6. Alprogramok, Eljárások és függvények, paraméterek fajtái, átadás-átvételi módok, túlterhelés, rekurzió (példák: Java, C++, Ada/SWIFT)</b>	<b>54</b>
6.1. Függvény-absztrakció . . . . .	54
6.2. Eljárás-absztrakció . . . . .	55
6.3. Alprogramok és paraméterek . . . . .	55
6.4. Paraméterek fajtái . . . . .	57
6.4.1. Érték szerinti paraméterátadás . . . . .	57
6.4.2. Cím szerinti paraméterátadás . . . . .	57
6.4.3. Eredmény szerinti paraméterátadás . . . . .	58
6.4.4. Név szerinti paraméterátadás . . . . .	58
6.5. Alprogramok túlterhelése (átlapolása) . . . . .	60
6.6. Rekurzió . . . . .	61
<b>7. A kivételkezelés</b>	<b>64</b>
7.1. Kivételek kiváltása - hibalehetőségek . . . . .	64
<b>8. Absztrakt adattípusok a programozási nyelvekben</b>	<b>70</b>
8.1. Procedurális absztrakció . . . . .	70
8.2. Adatabsztrakció . . . . .	70
8.3. Elvárások és eszközök . . . . .	71
8.4. Sablon . . . . .	74
8.5. Típussal, alprogrammal való paraméterezés . . . . .	74
8.6. Példányosítás . . . . .	75
<b>9. Programok helyessége és a támogató nyelvi eszközök (Hoare módszer, levezetési szabályok is) (példák: Eiffel, Java-eszközök, Code Contracts Library)</b>	<b>80</b>
9.1. Programok helyessége . . . . .	80
9.2. Hoare módszer . . . . .	80
9.3. Levezetési szabályok . . . . .	83
<b>10. Objektorientált programozás</b>	<b>92</b>
10.1. Osztályok és objektumok . . . . .	92
10.2. Objektumok létrehozása, inicializálása . . . . .	94
10.3. Példány- és osztályváltozó, metódus . . . . .	94
10.4. Társítási kapcsolatok . . . . .	95
10.5. Öröklődés . . . . .	96
10.6. Polimorfizmus . . . . .	97
10.7. Dinamikus kötés . . . . .	98
10.8. Megbízható átdefiniálás leszármazottakban . . . . .	98
10.9. Többszörös öröklődés . . . . .	99
<b>11. Párhuzamosság</b>	<b>106</b>
11.1. Alapfogalmak: . . . . .	106
11.1.1. Bevezetés: . . . . .	106
11.1.2. Szálak és folyamatok: . . . . .	107
11.1.3. Processzek közötti kapcsolatok: kommunikáció . . . . .	108
11.2. Flynn féle modell: . . . . .	108
11.3. Kommunikációs modellek: . . . . .	110
11.4. Szemafor . . . . .	112
11.5. Monitor: . . . . .	113
11.6. Fontos nyelvi elemek: . . . . .	113

<b>12. A funkcionális és logikai programozás alapjai, példákkal</b>	<b>122</b>
12.1. A funkcionális programozás alapjai . . . . .	122
12.2. Logikai programozás . . . . .	125

# **1. Szoftverek minőségi szempontjai, moduláris tervezés, a modularitás alapelvei. Programkönyvtárak tervezési elvei**

## **1.1. Szoftverek minőségi szempontjai:**

- helyesség
- megbízhatóság
- karbantarthatóság
- újrafelhasználhatóság
- kompatibilitás
- hordozhatóság
- hatékonyság
- stb.

### **Helyesség**

- A program helyes, ha pontosan megoldja a feladatot, megfelel a kívánt specifikációnak.
- Alapvető: a pontos és minél teljesebb specifikáció.

### **Megbízhatóság**

- Egy programot megbízhatónak nevezünk, ha helyes, és abnormális - a specifikációban nem leírt - helyzetekben sem történik katasztrófa, hanem valamilyen "ésszerű" működést produkál.

### **Karbantarthatóság**

- A karbantarthatóság annak a mérőszáma, hogy milyen könnyű a programterméket a specifikáció változtatásához adaptálni.
- Egyes felmérések szerint a szoftver költségek 70%-át szoftver karbantartásra fordítják!
- A karbantarthatóság növelése szempontjából a két legfontosabb alapelv:
  - a tervezési egyszerűség
  - a decentralizáció (minél önállóbb modulok létrehozása)

### **Újrafelhasználhatóság**

- Az újrafelhasználhatóság a szoftver termékek azon képessége, hogy egészben, vagy részben újrafelhasználhatóak új alkalmazásokban.

### **Kompatibilitás**

- A kompatibilitás azt mutatja meg, hogy milyen könnyű a szoftver termékeket egymással kombinálni. Nem légtüres térben fejlesztünk!

### **Hordozhatóság**

- A program hordozhatósága annak a mérőszáma, hogy mennyire könnyű a programot más géphez, konfigurációhoz, vagy operációs rendszerhez - általában más környezethez - átalakítani.

### **Hatékonyaság**

- A program hatékonysága a futási idővel és a felhasznált memória méretével arányos - minél gyorsabb, ill. minél kevesebb memóriát használ, annál hatékonyabb.

### **Barátságosság**

- A program emberközelisége, barátságossága a felhasználó számára rendkívül fontos: ez megköveteli, hogy az input logikus és egyszerű, az eredmények formája áttekinthető legyen.

### **Tesztelhetőség**

- A tesztelhetőség, áttekinthetőség a program karbantartói, fejlesztői számára fontos.

## **1.2. Moduláris tervezés**

Programjainkat egymástól minél inkább független, de jól definiált kapcsolatban álló programegységek segítségével tervezzük.

### **1.2.1. Moduláris dekompozíció**

- A moduláris dekompozíció a feladat több egyszerűbb részfeladatra bontását jelenti, amely részfeladatok megoldása már egymástól függetlenül elvégezhető. Ennek segítségével csökkenthetjük a feladat bonyolultságát.
- Általában a módszert ismételten alkalmazzuk, azaz az alrendszeret magukat is dekomponáljuk. Ezzel lehetővé tesszük azt is, hogy a feladat megoldásán egyszerre több ember is dolgozzon. A módszer egy fával ábrázolható, ahol a fa csomópontjai az egyes dekompozíciós lépéseknek felelnek meg.

### **1.2.2. Moduláris kompozíció**

- Olyan szoftver elemek létrehozását támogatja, amelyek szabadon kombinálhatók egymással.
- Programjainkat minél inkább már meglévő programegységekből, mint építőkövekből szeretnénk felépíteni.

## **1.3. A modularitás alapelvei**

- A modulokat nyelvi egységek támogatassák
  - A modulok illeszkedjenek a használt programozási nyelv szintaktikai egységeihez.
- Kevés kapcsolat legyen
  - Minden modul minél kevesebb másik modullal kommunikáljon!
- Gyenge legyen a kapcsolat
  - A modulok olyan kevés információt cseréljenek, amennyi csak lehetséges!
- Explicit interface kell
  - Ha két modul kommunikál egymással, akkor annak ki kell derülnie legalább az egyikük szövegéből.
- Információ elrejtés
  - Minden információ egy modullal rejtett kell legyen, kivéve, amit explicit módon nyilvánosnak deklaráltunk.
- Nyitott és zárt modulok

- Egy modult zártnak nevezünk, ha más modulok számára egy jól definiált felületen keresztül elérhető, a többi modul változatlan formában felhasználhatja.
- Egy modult nyitottnak nevezünk, ha még kiterjeszthető, ha az általa nyújtott szolgáltatások bővíthetők vagy ha hozzávehetünk további mezőket a benne levő adatszerkezetekhez, s ennek megfelelően módosíthatjuk eddigi szolgáltatásait.

## 1.4. Programkönyvtárak tervezési elvei

- Egy jelölés...
- Eszköz
  - a számítógép vezérlése
  - programozók közötti kommunikációra
  - algoritmusok leírására
  - magas szintű tervezésre
  - a feladat bonyolultságának kezelésére

### 1.4.1. Újrafelhasználható programkönyvtárak

- Tervezési igények:
  - Az alap-követelmények ugyanazok, mint más szoftver tervezésénél
    - \* helyesség
    - \* hatékonyság
    - \* megbízhatóság
    - \* kiterjeszthetőség
    - \* stb.,
  - De a könyvtár felé az igények erőteljesebbek!
- Az újrafelhasználható könyvtárakban található osztályokkal kapcsolatban az alábbi tulajdonságok várhatóak el:
  - Könnyű és intuitív használat: könnyen megérthető legyen, néhány használat után jól megjegyezhető
  - Azonnali, széleskörű felhasználhatóság szoftverrendszerek széles körében: szolgáltatásokat nyújt
  - A kurrens technológiához mért lehető leghatékonyabb megvalósítás
  - A tesztelés körültekintősége és az osztály megbízhatósága
  - Magas szintű dokumentáció: pontos, jól szervezett, hogy a felhasználó könnyen eligazodjon
  - Platform-függetlenség
  - Meg kell hagyni a későbbi fejlesztés lehetőségét
- Ezen követelmények néhány következménye:
  - konzisztencia - a könyvtár minden komponense egy átfogó, koherens tervezésnek kell megfeleljen, és számos szisztematikus, explicit és egységes konvenciót kell követessen.
  - jó minőségű, homogén komponensek kellenek
  - az objektum-orientált megközelítés az adatabsztrakció támogatása miatt különösen is alkalmas a könyvtárak készítésére

- A fenti kritériumok egy részének egyidejű teljesítése igen nehéz.
  - könnyen lehetséges, hogy a legáltalánosabb algoritmus nem a leghatékonyabb sok speciális, de gyakran előforduló esetben.
- A könnyű felhasználhatóságnak különböző aspektusai vannak.
  - Az egyik ezek közül az osztályok elkülöníthetősége.
    - \* Amennyiben nagyon sok hasonló osztály található egy könyvtárban, akkor a felhasználó számára igen nehéz a pontosan megfelelő kiválasztása.
  - Általában elmondható, hogy előnyösebb kevés számú, majdnem teljesen ortogonális osztály definiálása
    - \* minden esetben egy jól meghatározott szerep betöltésére, s az optimális megvalósítást követve.

#### 1.4.2. Biztonság

- Egy sokak által használt programkönyvtár esetén fontos szempont a megbízhatóság és a biztonság.
- Ennek biztosítása elő- és utófeltételek, valamint invariánsok adásával valósítható meg.
- Ez - a programkönyvtár írói és használói között – egy szerződés létrejöttét jelenti, ahol az előfeltétel a felhasználó kötelezettsége és a programozó biztosítéka, míg az utófeltétel a programozó kötelezettsége és a használó biztosítéka.
- Így a programozónak nem kell "defenzíven" programoznia, (minden lehetséges hibalehetőségre és bemenetre felkészülve) és a kód is hatékonyabb és egyszerűbb lesz. A lehetséges kétfajta megközelítési mód
  - Toleráns megközelítés: a programkönyvtári rutinoknak nincs (vagy csak gyenge) előfeltételük és minden lehetséges bemenetre valamilyen módon reagálnak.
  - Követelődző megközelítés: minden rutin szigorú előfeltételekkel rendelkezik, ezek biztosítása a felhasználó felelőssége.
- Az ajánlott megközelítés:
  - Csak az absztrakt művelet logikai helyes elvégzéséhez szükséges előfeltételeket ellenőrizzük.
  - Csak azt ellenőrizzük, ami, ha nem teljesülne, súlyosan befolyásolná a hatékonyságot.
- Az így létrejött szerződés bármilyen megsértése programhibát jelez. Az előfeltétel megsérülése felhasználó oldali hibát, míg az utófeltétel vagy az invariáns sérülése programkönyvtár hibát jelez. Ezért a programkönyvtár fejlesztése után a terjesztés előtt a hatékonyság érdekében elég csak az előfeltétel teljesülés vizsgálatát bekapcsolni.

#### A lényeg:

Explicit megszorítás kell, hogy a felhasználó tudja, mire számíthat!

#### 1.4.3. Dokumentáció

- Egy programkönyvtár több fejlesztő munkája.
- A felhasználhatóság szempontjából ezért igen fontos a jó dokumentáció.
- Lehetséges megközelítési módok: Forrásszöveg és különálló dokumentáció.



### **Forrásszöveg**

- A forrásszöveg nem elég absztrakt.
- A felhasználó számára fontos információkon kívül az alacsonyszintű implementációt is tartalmazza, ezért használata túl sok információ átfésülését jelentené, vagy arra sarkallná a programozókat, hogy nem publikusnak szánt implementációs lehetőségeket is kihasználjanak.
- Előnye, hogy mindig aktuális.

### **Különálló dokumentáció**

- Különálló dokumentáció esetén nem biztosított a szoftver és a dokumentáció konzisztenciája.
- Előnye, hogy csak a felhasználó számára fontos dolgokat tartalmazza.

### **Belső dokumentáció**

- Belső dokumentáció elve : a szoftver dokumentációja a programszövegbe legyen beágyazva.
- Ennek előnyei:
- Forrásszöveg és dokumentáció mindig konzisztens marad.
- Dokumentáció kinyerése automatizálható: javadoc

#### **1.4.4. Karbantartás**

- Mivel egy programkönyvtár állandó fejlesztés alatt áll, ezért lehetőséget kell biztosítani újabb verziók problémamentes beépítésére.
- A felhasználás szempontjából a következő változtatások nem jelentenek problémát:
  - Osztály bővítése új szolgáltatással.
  - Egy szolgáltatás implementációjának lecserélése.
  - Előfeltétel gyengítése, illetve utófeltétel szigorítása.

#### **1.4.5. Memóriakezelés**

- A programkönyvtár tervezés egyik kulcsfontosságú kérdése. Két alapvető probléma:
  - Memórafoglalás új objektumok létrehozásakor.
  - Memória felszabadítása.
- Míg az első az operációs rendszer feladata, addig a második problémára két megoldási irányzat is született:
- Automatikus szemétygyűjtés (pl. Java). Ez rendszerint referenciák bevezetését és a mutató típus megszüntetését jelenti.
- Objektumok kézzel történő felszabadítása. Hatékonyabb, de sokkal veszélyesebb

## 2. Neumann modell, LMC, generatív grammatikák és formális nyelvek alapfogalmai

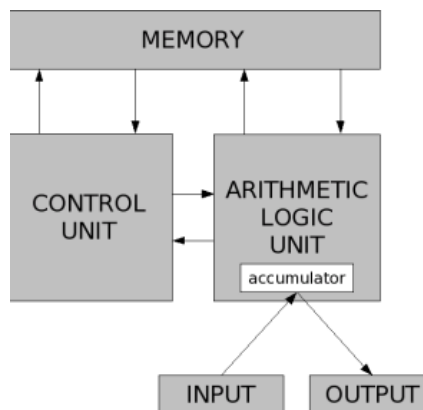
### 2.1. Neumann modell

Neumann-elvek a számítógépről:

- Legyen univerzális, vagyis bármilyen feladat megoldására használható.
- Legyen soros működésű, tehát az utasításokat sorban egymás után hajtja végre.
- Legyen teljesen elektronikus működésű.
- Az egyes utasítások és adatok leírásához és feldolgozásához a kettes számrendszert használja
- Legyen belső memóriája.
- Tárolt program elven működjön. Egy program indításakor a programot alkotó utasítások és a működéshez szükséges adatok is kerüljenek be a belső memóriába, és az utasításokat innen kiolvasva, sorban egymás után hajtja végre a berendezés.

A számítógép a következő egységekből álljon:

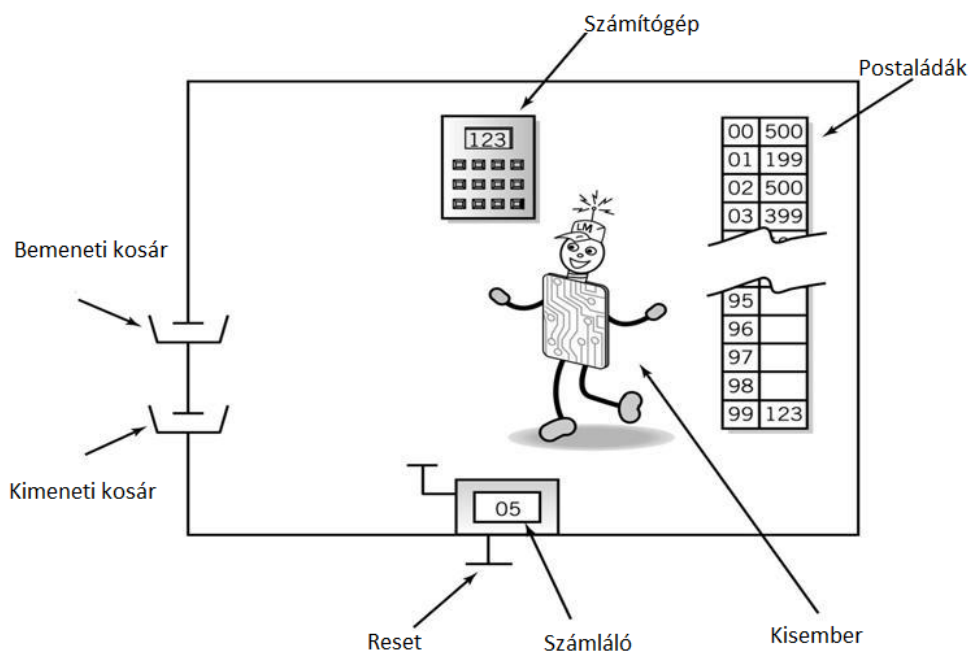
- **Központi egység:** vezérlés, műveletvégzés, adatmozgatás
- **Memória:** az adatok és a programok tárolása a program végrehajtása közben
- **Bemeneti-kimeneti egység:** kommunikáció a külvilággal, az adatbevitel- és kivetel megvalósítása



1. ábra. Neumann-elvű számítógép

## 2.2. A "Little Man Computer"

- Neumann elvek az oktatásban
- Stuart Madnick, MIT
- Szoba, benne:
  - Kisember
  - 100 postaláda (00-99)
  - Helyszámláló (2)
  - Bemenő és kimenő kosár
  - Számítógép



2. ábra. A "Little Man Computer"

### Az LMC elemei

- Bemeneti és kimeneti kosár:
  - Kisember kap/beletesz egy 3 jegyű számot tartalmazó cédulát
- Számláló:
  - 0 és 99 között tud számolni.
  - Pedál megnyomására a számlálóban tárolt szám értéke eggyel megnő.
  - Értékét nullára állíthatjuk egy külső úgynevezett "reset" (beállító) gombbal.
- Számítógép:
  - Háromjegyű decimális számokat tud kezelni.

- Képes:
  - \* Kivonni és
  - \* összeadni, valamint
  - \* a begépett vagy a számítás eredményeként kapott értéket eltárolni.

### Utasítás-végrehajtási ciklus

- Az utasítások végrehajtása:
  - Kikeresés (Fetch): Kisember kikeresi, hogy melyik utasítást kell végrehajtani
  - Végrehajtás: Kisember elvégzi a munkát.
- A számítógép egy-egy utasítás végrehajtásakor elvégzett tevékenység-sorozatot utasítás-végrehajtási ciklusnak nevezzük - ezek ui. ciklikusan ismétlődnek.

### Utasítás-végrehajtási ciklus kikeresési rész

1. Kisember kiolvassa az utasításslámlálóból annak a postaládának a sorszámát (címét), ami az utasítást tartalmazza.
2. Átsétál ahhoz a postaládához, ami megfelel a számláló értékének.
3. Kiveszi belőle cédulát, elolvassa és dekódolja a számot, ami rajta van (a cetlit visszateszi, arra az esetre, ha esetleg később szükség lenne rá, hogy újra elolvassa).

### Végrehajtási rész

1. Kisember odamegy, aminek a sorszámát az éppen kikeresett utasítás tartalmazza.
2. Kiveszi belőle a cédulát és elolvassa a rajta lévő számot (nem felejtí el a cédulát visszatenni, mert később is szükség lehet rá).
3. A kiolvasott utasítás kódja '5' volt, tudja, hogy ez azt jelenti, hogy "betöltés", így odamegy a számológéphez és beüti az ímént kiolvasott számott.
4. Odasétál a számlálóhoz és továbblöki, így folytathatja a következő utasítás kikeresésével.

### LMC utasításkészlete

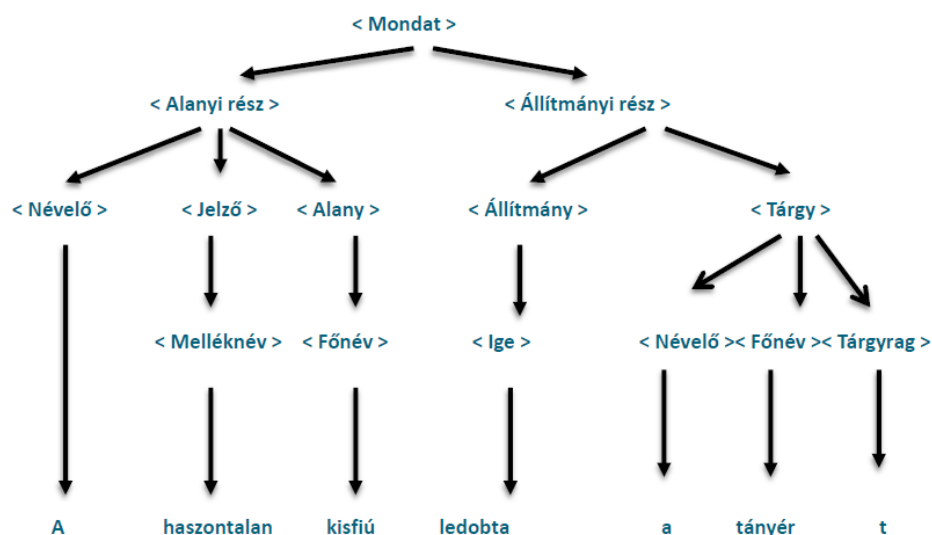
Pl egy ilyen lehet:

Hozzáadás	1xx
Kivonás	2xx
Tárolás	3xx
Beöltés	5xx
Ugrás	6xx
Ugrás o-nál	7xx
Ugrás pozitívnál	8xx
Input	901
Outout	902
Stop	000

## 2.3. Generatív grammatikák

- Természetes nyelvek vizsgálata
- Gyerek - véges számú mondatot hall
- Képes tetszőleges nyelvtanilag helyes mondat megalkotására
- Kell legyen egy szabályrendszer minden nyelvben!
- "A haszontalan kisfiú ledobta a tányért."
- Ez egy mondat → felbontható alanyi és állítmányi részre
- Az alanyi rész névelő, jelző és alany sorozata lehet
- Az állítmányi rész az állítmány és a tárgy egymásutánja stb.
- A struktúra leírásakor úgynevezett "grammatikai jeleket" is használhatunk a mondatrészek, illetve szófajok jelzésére - ezeket < > zárójelek közé tesszük.

### Szintaxisfa



3. ábra. "A haszontalan kisfiú ledobta a tányért" mondat szintaxisfája

Ezzel a probléma az, hogy egy ilyen fával eljuthatunk a továbbiakhoz:

- "a haszontalan tányér ledobta a kisfiút"
- "a haszontalan kisfiú ledoba a kisfiút"
- "a haszontalan tányér ledobta a tányért"

Nehéz jó megoldást találni, ezért a természetes nyelvek helyett vizsgáljuk a mesterséges nyelveket.

## 2.4. Formális nyelvek alapfogalmai

### Általános fogalmak

- A nyelv szintaxisa azoknak a szabályoknak az összessége, amelyek az adott nyelven írható összes lehetséges, formailag helyes programot (jelsorozatot) definiálják.
  - Reguláris kifejezések, BNF forma
- Az adott nyelv programjainak jelentését leíró szabályok összessége a nyelv szemantikája.

### A programok végrehajtása

- Az interpreter egy utasítás lefordítása után azonnal végrehajtja azt
- A fordítóprogram átalakítja a programot egy vele ekvivalens formára, ez lehet számítógép által (majdnem) közvetlenül végrehajtható forma, vagy lehet egy másik programozási nyelv.

### Fordítás

- A lefordítható program forrásprogram.
- A fordítás eredményeként kapott program a tárgyprogram.
- Az az idő, amikor az utasítások fordítása folyik, a fordítási idő.
- Az az idő, amikor az utasítások végrehajtása folyik, a végrehajtási idő.

### Formális nyelvek

**Definíció:** Az **abc** tetszőleges szimbólumok véges, nem üres halmaza, jelöljük  $\Sigma$ -val.

**Definíció:** A **szó** az abc elemeiből képezett  $a_1 a_2 \cdots a_k$  alakú sorozat, ahol  $k \geq 0$ ,  $a_1, a_2, \cdots a_k \in \Sigma$ .

**Definíció:** A  $\Sigma^*$  a  $\Sigma$  ábécé elemeiből képezhető összes szavak halmaza:

$$\Sigma^* = \{a_1 a_2 \cdots a_k \mid k \geq 0, a_1, a_2, \cdots a_k \in \Sigma\} \quad (1)$$

Ha  $k = 0$ , akkor a szó üres szó, jele  $\varepsilon$  vagy  $\lambda$ .

**Definíció:**  $\Sigma^+$  a nem üres szavak halmaza:

$$\Sigma^+ = \Sigma^* / \{\lambda\} \quad (2)$$

### Grammatikák

**Definíció: Generatív nyelvtan (G)**

- $G = (N, \Sigma, S, P)$ , ahol
- $N$  a nemterminálisok (grammatikai jelek) abc-je
- $\Sigma$  a terminálisok abc-je,  $N \cap \Sigma = \emptyset$
- $S \in N$  a kezdőszimbólum
- $P : \alpha \rightarrow \beta$  alakú átírási szabályok véges halmaza  $\alpha$  a szabály baloldala,  $\beta$  a szabály jobboldala  $\alpha, \beta \in (N \cup \Sigma)^*$ ,  $\alpha$ -ban van legalább egy nemterminális szimbólum

## Chomsky nyelvosztályok

### Definíció:

0 típusú nyelvtan (**kifejezés struktúrájú**), ha semmilyen korlátozás nincs.

1 típusú nyelvtan (**környezetfüggő**), ha P-ben minden szabály  $\alpha A \beta \rightarrow \alpha S \beta$  alakú, ahol  $\delta \neq \lambda$ , kivéve esetleg az  $S \rightarrow \lambda$  szabályt, mely esetben  $\alpha = \beta = \delta = \lambda$ , de ekkor S nem szerepelhet egyetlen szabálynak sem jobb oldalán. ( $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$ )

2 típusú nyelvtan (**környezetfüggetlen**), ha P-ben minden szabály  $A \rightarrow \alpha$  alakú. ( $A \in N, \alpha \in (N \cup \Sigma)^*$ )

### Definíció: Nyelv típusa

Egy  $L \subseteq \Sigma^*$  nyelv  $i$  típusú ( $i = 0, 1, 2, 3$ ), ha van olyan  $i$  típusú nyelvtan, ami éppen  $L$ -et generálja ( $\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$ ).

### 3. Nyelvi elemek, a programozási nyelvek lexikális elemei Vezérlési szerkezetek, utasítások, értékadás, szekvencia és blokk utasítás, feltétel nélküli vezérlésátadás, elágazási szerkezetek, ciklusszerkezetek (példák: Java, C++, választott nyelv\*)

#### 3.1. Nyelvi elemek, a programozási nyelvek lexikális elemei

##### A fordítási egység

- A programok fordítási egysége lexikális elemek sorozatai.
- A lexikális elemek karaktersorozatokat, határoló jelekkel elválasztva.
- A nyelvekben általában megkülönböztetjük
  - a grafikus karaktereket
    - \* a Latin-1 vagy az angol abc betűi, számjegyek és speciális karakterek
    - \* például: " #&'(),\* - +/ :: <= \_ [] {} | . -
  - a formátum vezérlő karaktereket
    - \* például az ISO 6429 szabványban:
      - character tabulation (HT)
      - line tabulation (VT)
      - carriage return (CR)
      - line feed (LF)
      - and form feed (FF)
  - az egyéb (implementáció függő) vezérlő karaktereket

##### Lexikális elemek

- azonosító
- numerikus literál
- karakter literál
- string literál
- határoló
- megjegyzés

##### A jelkészlet

- A karakterkészlet különböző karakterek egy halmaza.
  - Nincs feltételezésünk a belső ábrázolásáról, rendezettségéről.
  - Megadjuk a karakterek neveit és a karakter megjelenítését látható formában
    - \* Tartalmazhat olyan karaktereket, amelyek bizonyos megjelenítéseknél ugyanúgy néznek ki, mégis logikailag különbözőnek tekintjük.
    - \* Három különböző karakter, ami nagyon hasonló megjelenésű:
      - a latin nagy "A"
      - a cirill nagy "А"



- a görög nagybetűs Alfa "Α"
- \* Példa
  - EXCLAMATION → !
  - QUESTION\_MARK → ?
  - SEMICOLON → ;

### Karakterkódok - táblázatok

- A karakterkód egy leképezés, amely kölcsönösen egyértelmű megfeleltetést ad a karakterkészlet karakterei és a nemnegatív egészek halmaza között.
- Egy egyedi számkódot, egy kódpozíciót rendel a karakterkészlet minden karakteréhez.
  - Gyakran táblázat
- A karakterkódolás egy algoritmuskarakterek digitális formában történő megadására.
  - Ez a karakterek kódszámainak sorozatait „oktetek” sorozatára képezi le.
  - Például az ISO 10646 karakterkódban az 'a' és az 'ä' karaktereknek megfelelő számkódok a 97, a 228.
- A karakterkódolás adja meg, hogy a karakterkódokat hogyan képezzük le oktetek sorozatára.
  - Az ISO 10646 egy lehetséges kódolásában minden karakter kódolásához két oktetet használva a kódolási algoritmus ezeknek a (0, 97), a (0, 228) és a (32,48) oktetpárokat felelteti meg.

### Milyen karakterek használhatóak egy nyelvben?

- Pascal:
  - csak minimum követelmények pl.: '0', ... '9' rendezett és folytonos kell legyen, 'A' ... 'Z', de nem feltétlenül folytonos, 'a' ... 'z' nem kötelező, de ha van, akkor mint a nagybetűk.
  - Nincs előírva a lehetséges karakterek halmaza, különböző implementációk különböző kódolást használhatnak(!) (pl. EBCDIC vagy ASCII) → különböző implementációk másképpen viselkedhetnek
  - Ez nem tesz jót a programok hordozhatóságának.
- C++ - ISO/IEC 14882:2011 (és 2014 is)
  - Az alap karakterkészlet 96 elemű:
    - \* space
    - \* horizontal tab,
    - \* vertical tab
    - \* new-line
    - \* form feed vezérlő karakterei
    - \* a következő 91 grafikus karakter:
      - 'a' ... 'z' 'A' ... 'Z' '0' ... '9'
      - \_ { } # ( ) < > % : ; . ? + -
  - ISO 10646 szerint kódolva
  - Ez precízebb, mint az 1990-es, ahol nem volt a kód előírva
- CLU, Eiffel:
  - ASCII kód a megengedett.
- ADA (ISO/IEC 8652:2012(E)):

- A teljes ISO/IEC 10646:2011
- Java:
  - „Java programs are written using the Unicode character set”
- C#:
  - „A sourcefile is an ordered sequence of Unicode characters.”
- Python:
  - „Python reads program text as Unicode code points; ...”

### Elhatároló jelek

- A legtöbb programozási nyelvben a helyköz (space), a tab, a sorvége elhatároló jel.
  - Van sok egy karakterből álló elhatároló jel
    - \* Pascal: & ' ( ) + - \* / : ; <=>
    - \* C++: & % ' ( ) + - { } | [ ] " \* / : - ; , <=> ! ?
    - \* Java: ( ) { } [ ] ; , . + - \* / : <=> ! ? : & | %
    - \* ADA: & ' ( ) \* + , - . / ; : <=> |
  - Van számos több karakterből álló elhatároló jel is
    - \* Pascal: :=, >=, <=, <>, <<, \*\*, ..
    - \* C++: - >, ++, --, .\*, - > \*, ==, <=, >=, &&, ||, <<, >>, <=>, >=>, !=, .., + =, - =, & =, | =, :: \* =, / =
    - \* Java: ==, <=, > /, <<, !=, &&, ||, ++, --, >>, >>>, + =, - =, \* =, / = & =, | =, % =, <<=, >>=, >>>=
    - \* ADA: =>, .., \*\*, :=, / =, >=<=, <<, >>, <>, --

### Megkülönbözteti-e a nyelv a kis- és nagybetűket

- Kutya = kutya = KUTYA = KuTya = ..?
- Alapvetően két megközelítés
  - Pascal, Fortran, CLU, ADA, ... esetén ekvivalensek
- De!
  - Eiffel: a kis- és nagybetűk ekvivalensek, de használhatjuk ugyanazt a nevet egy objektumra és típusára
    - \* a: A
    - \* (itt az 'A' a típus és 'a' az objektum)

#### 3.1.1. Azonosítók

- Milyen karakterek használhatóak az azonosítók leírására?
- Mi az azonosító megengedett szintaxisa?
- Van-e hosszúsági megkötés az azonosítókra?
- Vannak-e kötött szavak? Van-e különbség a kulcsszavak és az előre definiált szavak között?

### Milyen karakterek használhatóak azonosítókban

- Pascal (ISO 7185:1990):
  - betűk ('A' ... 'Z', 'a' ... 'z') és számjegyek ('0' ... '9')
- C++ (ISO/IEC 14882:2003), CLU, Eiffel:
  - betűk ('A' ... 'Z', 'a' ... 'z'), számjegyek ('0' ... '9') és '\_'
- Perl:
  - az előzőeken túl \$, @, és % jellel is kezdődhetnek
    - \* \$ a skalárokat, @ számmal indexelt tömböt, % asszociatív tömböt jelöl.
- ADA:
  - Ada2012 szabvány: minden, aminek a nevében szerepel, hogy betű vagy számjegy!
- Java:
  - A Unicode betűi és számjegyei, a '\$' és '\_'
- C#:
  - Az azonosítókat a Unicode szabvány ajánlásának megfelelően kell írni.
  - Érdekesség, hogy a kulcsszavak a @bevezető jellel használhatóak azonosítóként is (például: @bool)
    - \* Ez a jel más azonosítót nem vezethet be.
  - Ezt a lehetőséget a programok hordozhatósága illetve "átjárhatósága" miatt vezették be.
  - A @prefixű szimbólumok neve valójában @nélkül kerül fordításra.
    - \* Így attól függetlenül, hogy egy másik nyelv nem tartalmazza például a 'sealed' kulcsszót, és egy programban azonosítóként használják, C#-ban lehetőség van az azonosító közvetlen használatára.

### Mi az azonosítók megengedett szintaxisa?

- A leggyakrabban:
  - letter
  - néha az '\_' beszúrása is megengedett.
- Pascal: letter{letter|digit}
- ADA: letter{[\_]letter|[\_]digit}
  - itt a betűk halmaza nagyobb, minden, aminek a nevében szerepel az, hogy "letter"
- Java: Java.letter{[\_]Java.letter|[\_]digit}

### Vannak-e "fenntartott" szavak?

- Pascal, C, C++, Java, Perl, CLU, ADA, etc.:
    - a fenntartott szavak (kulcsszavak) nem használhatók azonosítóként
  - ADA:
    - különbséget tesz a kulcsszavak és az előredefiniált szavak között, mint pl. : Integer, True, etc..
    - Az előredefiniált szavak átdefiniálhatóak:
      - \* type Integer is range -999\_999..+999\_999
      - \* így a program implementáció független
      - \* de: True: Integer .. – nincs értelme..
- Fortran, PL/1:
- \* megengedett a kulcsszavakat azonosítóként használni!

### 3.1.2. Literálok

- Milyen numerikus literálok vannak?
- Milyen más alapok megengedettek a 10-esen kívül?
- Mi az egész, ill. valós literálok szintaxisa?
- Többsoros sztring literálok megengedettek-e?
- Vezérlőkarakterek sztring literálokban megengedettek-e, és hogyan írjuk le őket?

### Numerikus literálok

- A gyakorlatban csak '0' .. '9' és az 'A' .. 'F' karakterek használhatók
- További lehetőségek:
  - ' \_ ' : 456\_789 ADA, Perl, Eiffel
  - Exp. alak: 1E6 ADA
  - 'L' 'U' a típus megadására (long, unsigned): 4L C, C++, Java C#

## 3.2. Vezérlési szerkezetek

### 3.2.1. Utasítások, értékadás

#### Egyszerű utasítások

- Értékadás - mit jelent?
  - "változó" értékadásjel kifejezés
  - helyette inkább: "balérték" értékadásjel "jobbérték"
    - \* (vagy fordítva?)
  - Mit jelent ez?
    - \* értékadásjel, általában ':=' , '=' , de van '->' is és '<-' is!
    - \* Szemantikája
    - \* "megfelelő" típusú kifejezés kell!

## Kifejezés

- operandusokból és operátorokból áll
  - minden operandus lehet egy újabb kifejezés
  - kiértékelem és megkapom az értékét

## Kifejezés az értékadás

- Wulf (BLISS): Of course, everything is
- Richie (C): Yes, why not
- Wirth (Pascal): No, only math-like things are expressions
- Két vonulat:
  - Pascal, CLU, ADA, Eiffel, ...
  - C, C++, Java
- Van-e többszörös értékadás?

## Szemlélet - COBOL

- Eredetileg
  - MOVE 23 TO A
  - MOVE B TO C
  - ADD A TO B GIVING C
  - SUBTRACT A FROM B GIVING C
  - MULTIPLY A BY B GIVING C
  - DIVIDE A BY B GIVING C
- Újabban
  - $COMPUTE A = (A + B - C / (A * B) - A * B)$

## Pascal

- "balérték" := kifejezés;
  - A "balérték" és a kifejezés típusa megegyező, de legalább kompatibilis kell legyen.
  - m: integer  $m := m + 1$ , vagy  $p := keres(gyoker, x)$ ;
- Többszörös értékadás nem megengedett.

## CLU

- referenciákat használ
- értékadás hatására két referencia hivatkozhat ugyanarra az objektumra
- Többszörös értékadás megengedett:  $x, y := y, x$ .

## C++

- Számos értékadó operátor jobbról balra feldolgozva
  - $A = B = C$ , jelentése  $A=(B=C)$ ;
- Értékadó operátorok
  - $=, *=, /=, \%=, +=, -=, >>=, <<=, \&=$
  - $y += g(x)$

## Java

- A C++-hoz hasonló, számos értékadó operátor:
  - $=, *=, /=, +=, -=, >>=, <<=, \&=$
  - $E1 \text{ op} = E2$
- Összetett értékadó operátoroknál mindkét operandus primitív típusú kell legyen (kivéve:  $+=$ , ha a bal operandus String típusú))
- Implicit cast előfordulhat! ( $\text{short } x=3; x+=4.6$ ; eredménye:  $x==7$ !)
- final-nak deklarált változónak nem adható érték.
- Alapvetően referenciákat használ

## Eiffel

- Az értékadás és a paraméterátadás szemantikája megegyezik.
- Ha  $x:TX, y:TY$ , akkor az  $x:=y$  eredménye TX és TY-től függ: referencia vagy „kiterjesztett” típusok?

## Üres utasítás

- Pascal: megengedett (case)
- ADA: null; (case)
- C++, Java, stb: ”;” használható (pl  $\text{for}(\;;\;)$ )
- Eiffel: ”;” használható (nincs valódi szerepe)

## 3.3. Szekvencia

- Lehet-e üres?
- Terminátor vagy utasítás elválasztó-e a ’;’?
- Lehet-e blokkutasítást létrehozni?
- Elhelyezhető-e a blokkutasításban deklaráció?
- Pascal: a ”;” elválasztja az utasításokat:
  - $\text{begin } ut1; ut2; ut3 \text{ end}$
  - üres utasítás is lehet:
    - \*  $\text{begin } ut1; ut2; ut3; \text{ end}$

- Az üres utasítás lehetősége miatt a ";" beírása megváltoztathatja a program jelentését!
- ADA: A ";" lezárja az utasítást
- C++, Java: a ";" lezárja az utasítást
  - de nem mindet, például a blokk utasítást nem
- Eiffel: nincs szükség elválasztójelre, de a ";" megengedett.

### 3.4. Blokk utasítások

- Utasítások sorozatából alkothatunk egy összetett utasítást, blokkot.
- Bizonyos nyelvekben lehet deklarációs része is.
- Főleg azokban a nyelvekben fontos, ahol a feltételes és a ciklus utasítások csak egy utasítást tartalmazhatnak (Pascal, C++, Java, ...)
- Pascal: begin utasítássorozat end
- ADA:

```

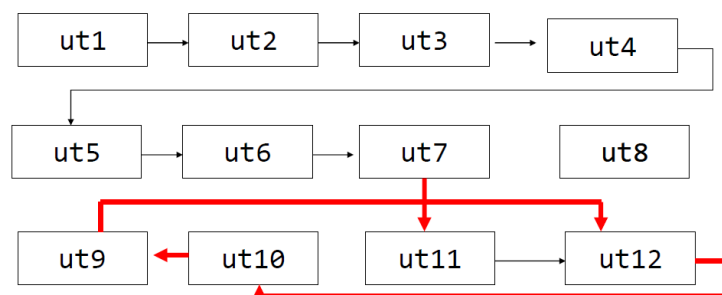
1  Csere:
2  declare
3  Temp: Integer;
4  begin
5      Temp:= I; I:= J; J:= Temp;
6  end Csere;

```

- C++, Java.. { és } között.

### 3.5. Feltétel nélküli vezérlésátadás

- goto



4. ábra. utasítás sorozat

- Ha mégis - Pascal, C stb...
  - címke:
  - ... utasítások ...
  - goto címke;
- Nincs: Modula-3, Java...

ut1
ut2
ut3
ut4

5. ábra. goto nélkül

### 3.6. Elágazás

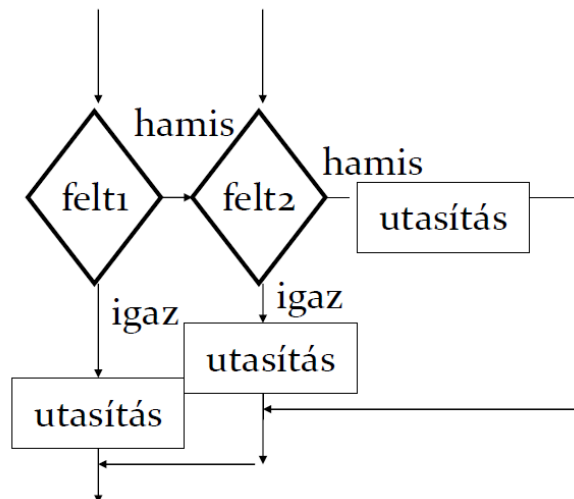
- Feltétel értékétől függően valamilyen utasítás(csoport) végrehajtása
- Feltétel igaz vagy hamis értékétől függően valamilyen utasítás(csoport) végrehajtása

```

1  if feltetel then ut
2  if feltetel then ut1 else ut2
3    "csellengo" else:
4  if felt1 then if felt2 then ut1 else ut2
5  if felt1 then (if felt2 then ut1 else ut2)
6  if felt1 then (if felt2 then ut1)else ut2

```

#### Többirányú elágazások



6. ábra. többirányú elágazás

```

1  if felt1 then s1;
2    else if felt2 then s2;
3      else if felt3 then s3;
4        else s4;
5  end if;

```



## Pascal

```
1  if (a>b)
2      then writeln("a>b")
3  else if (a<b)
4      then writeln("a<b")
5  else
6      writeln("a=b");
```

## ADA

- end if a végén, elsif megengedett:

```
1  if A>B then
2      Put_Line("a>b");
3  elsif A<B then
4      Put_Line("a<b");
5  else
6      Put_Line("a=b");
7  end if;
```

## C++

- aritmetikai kifejezés kell, nemnulla: true.
- ”csellengő” else: legbelső if
  - blokk kell különben

```
1  if (a>b) cout << "a>b";
2  else
3      if (a<b) cout << "a<b";
4      else cout << "a=b";
```

## Java

- Hasonló a C++-hoz
  - kivéve a kifejezés típusa boolean kell legyen

```
1  if (a>b)
2      System.out.println("a>b");
3  else
4      if (a<b)
5          System.out.println("a<b");
6      else
7          System.out.println("a=b");
```

## Eiffel

```
1  if (a>b) then
2      io.putstring("a>b");
3  elseif (a < b) then
4      io.putstring("a < b")
5  else
6      io.putstring("a = b");
7  end
```

## Python

- Csellengő else: a bekezdés számít!!

```
1 a, b = 1, 2
2 if a == 1:
3     if b == 1:
4         print("a és b is 1")
5 else:
6     print("a nem 1")
```

## if és értékadás

- Ruby
  - `num += -1 if num < 0`
- Scala:
  - `val x = if (a > b) a else b`
- Jobban olvasható, mint a `? :`

## Esetkiválasztós elágazás

- valamilyen kifejezés (szelektor) értékétől függően

## Case utasítások

- Sokszor igaz a case konstansokra:
  - Tetszőleges sorrend,
  - Nem feltétlenül egymás után
  - Több is vonatkozhat ugyanarra az alutasításra
  - Mind különböző kell legyen - különben, ha  $const_i$  és  $const_j$  egyenlőek, akkor melyiket választanánk,  $st_i$ -t, vagy  $st_j$ -t?
  - A kifejezés típusa diszkrét kell legyen (egész vagy felsorolási típus)
  - Kell adni egy "others" ágat, hogy minden lehetőséget lefedjünk

## Pascal

- A szelektor típusa lehet: integer, character, boolean, vagy tetszőleges felsorolási, vagy intervallum típus.
- "else" megengedett, de nem kötelező (skip).

```
1 var Age: Byte;
2 case Age of
3     0..13: Write('Child');
4     14..23: Write('Young');
5     24..65: Write('Adult');
6     else Write('Old');
7 end
```

## ADA

- others kötelező, ha nincs minden lefedve!
- ..- intervallum, |-vagy

```
1 case Today is
2   when Monday => Initial_Balance;
3   when Friday => Closing_Balance;
4   when Tuesday..Thursday => Report(Today);
5   when others => null;
6 end case;
```

## C++

- kifejezés "integral" típusú
- default: címke lehtësége (nem kötelező)
- itt break kell, ha befejezzük

```
1 switch (intexpr) {
2   case label1: statm1; break;
3   case label2: statm2; break;
4   ...
5   default: statm;
6 }
```

## Java

Pont mint a C++

```
1 switch (val){
2   case 1 : System.out.println("case 1\n"); break;
3   case 2 : System.out.println("case 2\n"); break;
4   default: System.out.println("default case "); break;
5 }
```

## Eiffel

- Pascal-szerű szemantika
- A változó típusa INTEGER vagy CHARACTER
- Exception lép fel, ha nem gondoskodtunk a megfelelő választék-elemről (null utasítás szükséges lehet).

```
1 inspect var
2   when expr1 then statmblock1
3   when expr2 then statmblock2
4   ...
5   else statmblock
6 end
```

## C#

- szemantika
- ciklusként is működhet!

```
1  switch(kif)
2  {
3      case ertek1 : utasítások... break;
4      case ertek2 : utasítások... break;
5      case ertek3 : utasítások.. break; vagy:
6                      goto case KONST / default;
7      ...
8  }
```

## Swift

- Lehetséges az értékre rész-megszorítást adni

```
1  switch vegetable {
2      case "celery":
3          let vegetableComment= ...
4      case "cucumber", "watercress":
5          let vegetableComment= ..."
6      case let x where x.hasSuffix("pepper"):
7          let vegetableComment= ...
8      default:
9          let vegetableComment= ...
10 }
```

## 3.7. Ciklusok

- utasítások ismételt végrehajtása
- valamilyen feltételtől függően

### Ciklusok általános kérdései

- Vannak-e nem ismert lépésszámú ciklusok?
  - Van-e elől/hátul tesztelős ciklus?
  - A ciklusfeltétel logikai érték kell legyen, vagy más típusú lehet?
  - Kell-e blokkot kijelölni a ciklusmagnak?
  - Van-e előre ismert lépésszámú ciklus?
    - \* A ciklusváltozó mely jellemzője állítható be a következők közül?
      - alsó érték - felső érték - lépésszám
    - \* Mi lehet a ciklusváltozó típusa?
    - \* Biztosított-e a ciklusmagon belül a ciklusváltozó változtathatatlansága?
    - \* Mi a ciklusváltozó hatásköre, definiált-e az értéke kilépéskor?
    - \* Van-e általános (végtelen) ciklus?
    - \* Léteznek-e a következő vezérlésátadó utasítások?
      - break, continue
    - \* Van-e ciklusváltozó iterátor?

### Nem ismert lépésszámú ciklusok

- Elöltesztelő "while" ciklusok:
  - while <kif> do <utasítás>
- Pascal: a ciklumag egyetlen utasítás lehet (de blokk is)
  - while  $a < b$  do  $b := b - a$ ;

### ADA

- end loop zárja, így a ciklusmag utasítássorozat is lehet:

```
1   while kifejezés loop
2       ciklusmag
3   end loop
```

- kifejezés Boolean típusú

### C++

- A kifejezés aritmetikai értéket ad
  - nem 0: true, 0: false
- A ciklusmag egyetlen utasítás lehet (de blokk is)

```
1   while (<expr>) <statm>
2   while (a < b)
3       b = b - a;
```

### Java

- Hasonló a C++-hoz, kivéve: a kifejezés típusa boolean kell legyen.

```
1   i = 0;
2   while (i<10) {
3       System.out.println(i);
4       i++;
5   }
```

### Eiffel

- Amíg a ciklusfeltétel hamis!
- end zárja, így a ciklusmag utasítássorozat is lehet.

```
1   from
2       init
3       [invariant loop - invvariant
4         variant-func]
5   until loop-cond
6   loop
7       loop-body
8   end
```

### Hátultesztelő "repeat-until" ciklusok

- Amíg egy feltétel igaz nem lesz.
- Ciklusmag egyszer biztosan lefut

```
1 repeat
2   utasítássorozat
3 until ciklusfelt
```

- nincs szükség blokk utasításra - a ciklusmag vége meghatározott
- while E do S

– Jelentése

```
1   if E then
2     repeat S until not E
```

### Előre ismert lépésszámú ciklusok "For" ciklusok

- index változókezelése-lépésköz-határ
- Egyszer, a ciklusba való belépés előtt értékeli ki a lépésköztétele határt, vagy minden végrehajtás után újra?
- A határt a ciklusmag végrehajtása előtt vagy után ellenőrzi?
- Mi lehet a ciklusváltozó típusa?
- Biztosított-e a ciklusmagon belül a ciklusváltozó változtathatlansága?
- Mi a ciklusváltozó hatásköre, definiált-e az értéke kilépéskor?

### Pascal

- lépésköz és határ: egyszer
- határ ellenőrzése ciklusmag előtt
- ciklusváltozó nem változtatható (ma már)
- értéke kilépéskor nem definiált
- ciklusváltozó diszkrét típusú lehet

```
1 for i := 1 to n do sum := sum + i;
2 for c := 'z' downto 'a' do write(c);
```

### Ada

- Az index a ciklus lokális konstansa. Diszkrét típusú kell legyen. A ciklus értékintervallumát csak egyszer, a ciklusba való belépés előtt számítja ki.
- A "reverse" hatására: fordított sorrend.

```
1 for <var> in loop-range loop
2   <utasítások>;
3 end loop;
```

## C++

- a for(for-init-stm[expr-1]; [expr-2]) stm jelentése

```
1 { for-init-stm
2   while (expr-1) {
3     stm
4     expr-2;
5   }
6 }
```

- Kivéve, hogy egy continue a stm-ben expr-2-t hajt végre az expr-1 kiértékelése előtt.
- Hiányzó expr-1 ekvivalens: while(1)
- Ha a for-init-stm egy deklaráció, akkor a deklarált nevek hatásköre a for utasítást tartalmazó blokk

```
1 sum = 0;
2 for (int i = 0, i < n; i++)
3   sum += i;
```

## Java

- Eredeti for ciklus: hasonló a C++-hoz:

```
1 public class ForDemo {
2   public static void main(String[] args) {
3     int[] arrayOfInts = {32, 87, 199, 622, 127, 485};
4     for (inti= 0; i< arrayOfInts.length; i++) {
5       System.out.print(arrayOfInts[i] + " "); i++;
6     }
7     System.out.println();
8   }
9 }
```

- foreach

```
1 List<Number> szamok= new ArrayList<Number>();
2 szamok.add(new Integer(42));
3 szamok.add(new Integer(-30));
4 szamok.add(new BigDecimal("654.2"));
5 for ( Number number: szamok){
6   ...
7 }
```

## C#

- "hagyományos" for ciklus hasonlóan a C++-hoz:

```
1 using System;
2 public class ForLoopTest
3 {
4   public static void Main()
5   {
6     for(int i = 1; i <= 5; i++)
7       Console.WriteLine(i);
8   }
9 }
```

## Swift

- For ciklus

```
1  var firstForLoop= 0
2  for i in 0..< 4 {
3      firstForLoop += i
4  }
```

```
1  var secondForLoop = 0
2  for var i = 0; i < 4; ++i {
3      secondForLoop += i
4  }
```

- A ..< használható arra, hogy az i 0 és 3 közötti értékeket veszi fel, ha 0 és 4 közötti értékeket szeretnénk, akkor i in 0... 4 a használandó szintaktika

## Végtelen ciklusok

- ADA

```
1  loop
2      <statm>1; ...
3      exit when <feltétel>;
4      <statm>2; ...
5  end loop;
6
7  loop
8      get(Current_Char);
9      exit when Current_Char = '*';
10 end loop;
```

- break: kiugrás a vezérlési szerkezetből -pl:

```
1  while feltétel do
2      if spec eset then
3          nyomjad
4          break;
5      end if;
6  end while;
```

- continue: ciklusfeltétel újraértékelése
- alprogram hívás
- return alprogramból hívóhoz visszatérés
- goto - !!Veszélyeket rejt



## 4. Típusok 1

**Típusspecifikáció, típusmegvalósítás, típusosztályok, skalár típusosztály típusai a programozási nyelvekben, (példák: Java, C++, Ada/Pascal)**

### 4.1. Típusspecifikáció

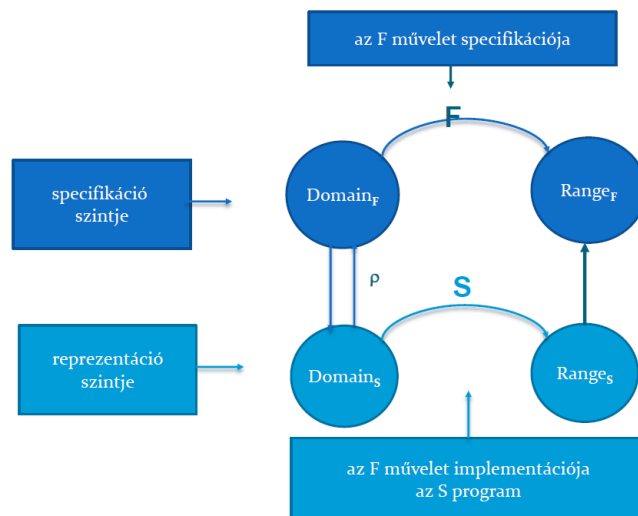
- Egy értékhalmoz és egy művelethalmoz.
- specifikáció - reprezentáció - implementáció
  - pl. specifikáció: egészek  $-\infty$ -tól  $+\infty$ -ig, és a megengedett műveletek  $+$ ,  $-$ ,  $*$ , stb.
  - reprezentáció: 8, 16, 32, 64-bit, előjeles kettes komplementes kóddal (megszorítások)
  - implementáció: a műveletek megvalósítása

**Típus a programozó szemszögéből**

- Típusspecifikáció ("mit")
  - alaphalmaz: a valós világ minket érdeklő objektumainak halmaza
  - specifikációs invariáns ( $I_S$ ) - ezt a halmazt tovább szűkíti
  - típusműveletek specifikációja - csak a "mit"!

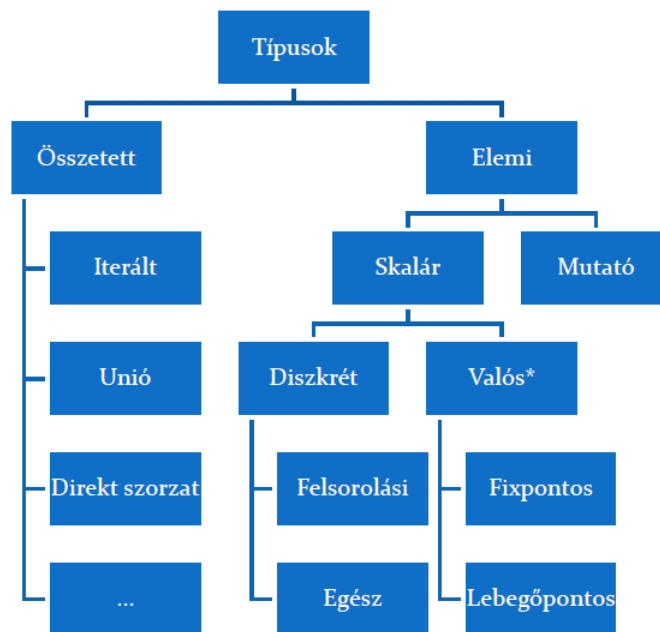
### 4.2. Típusmegvalósítás

- Hogyan?
  - Reprezentációs függvény
  - Típus invariáns
  - Típusműveletek megvalósítása
  - (Pl. komplex számok, verem, stb.)



7. ábra. A típus specifikáció és a típus kapcsolata

### 4.3. Típusosztályok



8. ábra

#### Van-e speciális tulajdonsága a nyelv típusrendszerének?

- ADA:
  - Új típus létrehozása már létező típusból: származtatás
  - Itt kicsit mást jelent, mint az OOP-ben
- Java:
  - kétféle típus létezik: a primitív típusok (numerikus és logikai), és a referencia típusok
  - A felhasználó által definiált típusok az osztályok
  - A primitív típusoknak van "csomagoló" osztálya, boxing, unboxing lehetőségekkel
- Eiffel:
  - alapértelmezés szerint az értékek objektumokra vonatkozó referenciák, de definiálhatunk kiterjesztett (expanded) típusokat is, ezek változói maguk az objektumok.
  - Az alap típusok mindig kiterjesztettek.
  - Új típus = új osztály létrehozása.
- C#: érték és referencia típusok
  - Érték típusok: egyszerű típusok (pl. char, int, float), felsorolási és struct típusok.
  - Referencia típusok - osztály (class), interface típusok, delegate és tömb típusok.
  - Boxing – unboxing lehetőségek vannak itt is.

#### 4.4. A skalár típusok osztálya

- A skalár típusok a diszkrét és a valós típusok.
  - Értékei rendezettek, így a relációs operátorok (<, <=, =, >=, >) előredefiniáltak ezekre a típusokra.
- Eiffel-ben ezek a típusok mindig kiterjesztettek.
- C++ ezek az ún. "integral" típusok
- Perl:
  - Skalár adattípus:
    - \* a neve kötelezően \$ jellel kezdődik
    - \* számok, stringek (!) és referenciák tárolására, a típusok közötti (numerikus, string, ...) konverzió automatikus

#### Felsorolási típusok

- A típusértékhalmoz megadható egy explicit felsorolással
  - type days is (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);
- Gyakran a karakter és a logikai típus is előredefiniált felsorolási típus
  - A logikai értékek kezelésére szolgáló típust sok programozási nyelvben Boolean-nak hívják, két lehetséges értéke van, a True és a False, a szokásos műveletek a not, and, or, és a xor.
  - De vannak kivételek: például Yes, No Objective-C esetén
- Object Pascal:
  - Egy felsorolási típus definíciója:
  - type Flower = (Rose, Tulip, Violet, Geranium);
  - Az azonosítóknak különbözőeknek, és a programban egyedieknek kell lenniük.
  - Standard függvények:
    - \* Pred(X)
    - \* Succ(X)
    - \* Dec(X)
    - \* Inc(X)
    - \* ...
- ADA:
  - Minden felsorolási literál a felsorolási típus egy önálló típusértéke, és van egy pozíció száma.
    - \* Az első felsorolási literál pozíció száma 0.
    - \* A rendezési relációt a felsorolás sorrendje adja. Egy felsorolási típus értékei lehetnek azonosítók vagy karakterek.
    - \* type Color is (Red, Blue, Green);
    - \* type Roman\_Digit is ('I', 'V', 'X', 'L', 'C', 'D', 'M');
    - \* Megengedett a felsorolási nevek átlapolása, és ha szükséges a típusnév segít a megkülönböztetésben: Color'(Red).
  - Három előredefiniált karaktertípus van: Character, Wide\_Character, Wide\_Wide\_Character
  - Logikai:

\* type Boolean is (False, True);

- C++:

- Az "enum" kulcsszót használják a felsorolási típusok, a megfelelő egész érték 0-val kezd, egyesével nő, kivéve, ha "=expr" szerepel valahol(!).
  - \* enum color {red, blue, green=20, yellow};
- Az előredefiniált char, signed char, és unsigned char típusok írják le a lehetséges karakter típusokat, sok probléma származik abból, hogy ezeket nem szabványosították igazán.
- Előredefiniált logikai típus a bool a false és a true értékekkel, kifejezésekben a 0 értéket is hamisként kezeli, és minden nem 0-t igaznak.

- Java:

- Nem volt felsorolási típus - a Java 5.0-ig!
- Az előredefiniált char típus a 16 bites Unicode character-készletet támogatja '/uoooo'-tól '/uffff'-ig, azaz 0-tól 65535-ig.
- Előredefiniált logikai típus, a boolean a false és true értékekkel
- A műveletek:
  - \* Relációs: == !=
  - \* not: !
  - \* and, or: && ||
  - \* and, or, xor (bitwise): & |
  - \* feltételes kifejezés: ? :
- enum Season {WINTER, SPRING, SUMMER, FALL }

- C# - Enum

- Rendelkezik alaptípussal - ez tetszőleges integral típus lehet
  - \* byte, sbyte, short, ushort, int, uint, long, ulong
- Deklaráció példa:

```
1 enum Color: long {  
2     Red  
3     Green  
4     Blue  
5 }
```

- Alapértelmezésben int
- System.Enum a közös őssztály

## Egész típusok

- Az egész típusok nagyon közel vannak a számítógépes reprezentációhoz
- A műveletek általában a szokásosak, néhol gond van az osztással és hatványozással
- Object Pascal lehetőségek:
  - ShortInt - signed 8 bit
  - SmallInt - signed 16 bit
  - Integer - signed system-dep
  - LongInt - signed 32 bit

- Byte - unsigned 8 bit
- Word - unsigned 16 bit
- Cardinal - unsigned system-dep
- ADA:
  - Előjeles egészek:
    - \* type Page\_Num is range 1 .. 2000
    - \* type Line\_Size is range 1 .. Max\_Line\_Size
  - maradékosztályok
    - \* type Byte is mod 256;
    - \* type Hash\_Index is mod 97;
  - Minden egész típust úgy tekintenek, mint ami a névtelen, előredefiniált root\_integer típusból lett származtatva.
  - Egész literálok ennek az universal\_integer osztályába tartoznak.
  - Az előredefiniált egész típus az Integer, van két előredefiniált altípus
    - \* subtype Natural is Integer range 0 .. Integer'Last;
    - \* subtype Positive is Integer range 1 .. Integer'Last;
  - Az értékintervallumnak egy tetszőleges implementáció esetén tartalmaznia kell a  $-2^{15}+1$  ..  $+2^{15}-1$ -t.
  - Megengedett, de nincs előírva:
    - \* Short\_Integer
    - \* Long\_Integer
    - \* Short\_Short\_Integer
    - \* Long\_Long\_Integer
  - A műveletek a szokásosak...
- C++:
  - Az előredefiniált egész típusoknak 4 mérete lehet: short int, int, long int, long long int.
  - Az unsigned int, unsigned long int, unsigned short int típusokat modulo  $2^n$  aritmetikával használja (n a reprezentációban a bitek száma).
  - A szabvány definiálja a minimum értékintervallumokat
  - C++11 fix méretű típusok: int8\_t, int16\_t, int32\_t, int64\_t
  - A műveletek:
    - \* Relációs: ==, !=, <, <=, >, >=
    - \* Unáris: \*, +, -, &
    - \* Multiplikatív: \*, /, %
    - \* Additív: +, -
    - \* incr. prefix postfix: ++
    - \* decr. prefix postfix: --
    - \* shift előjeles: <<, >>
    - \* Komplement bitenként: ~
    - \* Feltételes op.: ? :
    - \* Sizeof
    - \* pointer\_to\_member: ->\*, .\*
    - \* Értékadó op.: =, \*=, /=, %=, +=, -=, >>=, <<= &=, !=, ^=

- Java

- Az előredefiniált egész típusoknak előírt specifikációja van:
  - \* byte: -128.. 127
  - \* short: -32768.. 32767
  - \* int: -2147483648.. 2147483647
  - \* long: -9223372036854775808.. 9223372036854775807
  - \* char: '\u0000'.. '\uffff', vagyis: 0.. 65535
- Operátorok
  - \* Relációs: ==, !=, <, <=, >, >=
  - \* Unáris: +, -
  - \* Multiplikatív: \*, /, %
  - \* Additív: +, -
  - \* incr. prefix postfix: ++
  - \* decr. prefix postfix: --
  - \* shift előjeles: <<, >>
  - \* Komplement bitenként: ~
- További hasznos előredefiniált műveletek a csomagoló osztályokban

- C#

- Előírt specifikáció van itt is
  - \* sbyte: signed 8-bit → -128 ... 127
  - \* byte: unsigned 8-bit → 0 ... 255
  - \* short: signed 16-bit → -32768 ... 32767
  - \* ushort: unsigned 16-bit → 0 ... 65535
  - \* int: signed 32-bit → -2147483648 ... 2147483647
  - \* uint: unsigned 32-bit → 0 ... 4294967295
  - \* long: signed 64-bit → -9223372036854775808 ... 9223372036854775807
  - \* ulong: unsigned 64-bit → 0 ... 18446744073709551615.

- Eiffel:

- Az INTEGER kiterjesztett, a COMPARABLE és a NUMERIC osztályok leszármazottja.
- A reprezentáció legalább Integer\_bits bitet használ, ez egy platform-függő konstans, amelyet a PLATFORM osztály definiál.
- Műveletek:
  - \* <, <=, >, >= a COMPARABLE-ból
  - \* +, -, \*, / a NUMERIC-ből
  - \* és az újak:
    - hatványozás ^
    - egész osztás //
    - maradék \

## Valós típusok

- A valós típusok a valós számok közelítései.
  - a lebegőpontos típusok - relatív pontosság
  - fixpontos típusok - abszolút pontosság
- A legtöbb programozási nyelv támogatja az előjeles lebegőpontos típusokat, ahol 1 bit az előjel, és a szám formátuma: ej, mantissza, kitevő
- A kitevőnek is lehet előjele
- Pascal
  - A nyelv lebegőpontos valósakat használ.
    - \* Single → 32 bit
    - \* Real → 48 bit
    - \* Double → 64 bit
    - \* Extended → 80 bit
    - \* Comp → 64 bit
- Object Pascal:
  - Single, Double, Extended
    - \* IEEE nemzetközi szabvány szerint.
  - Új fixpontos valós típus 4 számjegy pontossággal
  - Műveletek a szokásosak
- ADA:
  - A nyelv lehetőséget ad a lebegőpontos és a fixpontos típusok kezelésére, de csak egy előredefiniált lebegőpontos típus van, a Float.
  - A lebegőpontos típusoknál a relatív pontosság, míg a fixpontos típusoknál az abszolút pontosság megadására van lehetőség. Megadható egy értéktartomány is:
    - \* type Real is digits 8;
    - \* type Coefficient is digits 10 range -1.0 .. 1.0;
    - \* type Mass is digits 7 range 0.0.. 1.0E35;
  - Minden valós típust úgy tekintenek, mint egy előre definiált root\_real típusból származtatott típust. A valós literálok ennek az osztályába tartoznak, így universal\_real típusúak
  - Ha egy implementációban a lebegőpontos típusok pontossága legalább 6 számjegy, akkor a Float típus pontossága is legalább ennyi kell legyen.
  - Megengedett, hogy egy implementáció támogasson további előre definiált lebegőpontos típusokat, pl.: Short\_Float,...
  - A szokásos műveletek
- C++:
  - Az előre definiált valós típusok a float, double és a long double.
  - Műveletek
    - \* Relációs: ==, !=, <, <=, >, >=
    - \* Unáris: \*, +, -, &
    - \* Multiplikatív: \*, /, %

- \* Additív: +, -
- \* incr. prefix postfix: ++
- \* decr. prefix postfix: --
- \* shift előjeles: <<, >>
- \* Komplement bitenként: ~
- \* Feltételes op.: ? :
- \* Sizeof
- \* pointer\_to\_member: ->\*, \*
- \* Értékadó op.: =, \*=, /=, %=, +=, -=, >>=, <<=, &=, !=, ^=

- Java:

- Az előredefiniált lebegőpontos típusoknak előírt pontossága van
  - \* float → 32 bit IEEE 754
  - \* double → 64 bit IEEE 754
- A float típus  $s \cdot m \cdot 2^e$  alakú, ahol
  - \*  $s = +1$ , vagy  $-1$
  - \*  $m$  egy pozitív egész, kisebb, mint 224
  - \*  $e$  egy egész a -149.. 104 intervallumból.
- A double típus  $s \cdot m \cdot 2^e$  alakú, ahol
  - \*  $s = +1$ , vagy  $-1$
  - \*  $m$  egy pozitív egész, kisebb, mint 253
  - \*  $e$  egy egész -1075.. 970 intervallumból
- Érdekességek: van POSITIVE\_INFINITY és NEGATIVE\_INFINITY, illetve speciálba van NaN, Not a Number érték is.
- Operátorok
  - \* Relációs: ==, !=, <, <=, >, >=
  - \* Unáris: +, -
  - \* Multiplikatív: \*, /, %
  - \* Additív: +, -
  - \* incr. prefix postfix: ++
  - \* decr. prefix postfix: --
  - \* shift előjeles: <<, >>
  - \* Komplement bitenként: ~
  - \* Feltételes op.: ? :

- C#:

- float → 32 bit IEEE 754
- double → 64 bit IEEE 754

- Pozitív és negatív 0

- Pozitív és negatív végtelen

- NaN itt is van (pl 0.0/0.0)

- Decimal típus pénzügyi számításokhoz:

- 128 bites adattípus, értékei:  $1.0 \times 10^{-28} \dots 7.9 \times 10^{+28}$



- Lényeg, hogy a tízes számrendszerbeli törtértékeket is pontosan ábrázolja, műveleteknél szokásos módon kerekít
- Eiffel:
  - A REAL osztály kiterjesztett, a COMPARABLE és a NUMERIC osztályok leszármazottja.
  - A reprezentáció legalább Real\_bits bitet használ, ez egy platform-függő konstans, amelyet PLATFORM osztály definiál.
  - A műveletek:
    - \* <, <=, >, >= a COMPARABLE-ból
    - \* +, -, \*, / a NUMERIC-ből
    - \* és az újabbak: hatványozás pl ^

## 5. Típusok 2

**Pointer és referencia típusok, típusok ekvivalenciája (példák: C++, Ada, Eiffel) Típuskonstrukciók – tömbök, direkt szorzat, unió, halmaz támogatása (példák: Java, C++, Ada/Pascal)**

### 5.1. Pointer és referencia típusok

- Egy pointer (és egy referencia) egy olyan objektum, amely megadja egy másik objektum címét a memóriában
- Egy pointer értéke egy memóriacím
  - gépi nyelvekben az indirekt címzés lehetősége motiválta a pointerek létrehozását.
- Vannak típusos és típus nélküli pointerek

#### Referencia szint

- Pointerek segítségével magasabb referencia-szinten hivatkozhatunk objektumainkra.
- Van mondjuk egy szám 42 → referencia szint 0
- egy olyan változó referencia-szintje, amely tartalmazza a 42 értéket → 0
- a pointer referencia-szintje, amely erre a változóra mutat → 2

#### Pointer hatékonyság

- Ahelyett, hogy nagy adatszerkezeteket mozgatnánk a memóriában, sokkal hatékonyabb, ha az erre mutató pointert másoljuk, mozgatjuk.
- Read-only elérési lehetőségre figyelni kell!
- Objektumorientált funkciókhoz
  - a programozási nyelvekben a polimorfizmust akkor tudjuk támogatni, ha a változók objektumokra való referenciákat tartalmaznak.

#### Pointer műveletek

- **értékadás** - a pointerek között, hivatkozott objektumok között (copy, clone, deep-copy, deep-clone!)
- **egyenlőség vizsgálat** - ha két ugyanolyan típusú pointer ugyanarra az adatszerkezetre mutat (ez is több szinten lehet)
- **dereferencing** - a mutatott objektum részére vagy egészére való hivatkozás
- **referencing** - egy objektum címe
- új objektum dinamikus **allokálása**
- egy objektum **deallokálása** - explicit művelettel vagy implicit módon egy garbage collector-al
- néha (pl. C, C++) **összeadás**, **kivonás** is megengedett

#### ”Csellengő” pointerek

A ”csellengő” pointer: kísérlet olyan változó elérésére, ami már nem létezik.

## Pointerek az egyes nyelveken

- Csak konkrét típusra, vagy típus nélküli pointerek is megengedettek?
- Lehetnek-e alprogramra mutató pointerek?
- A pointereknek kell-e önálló név, vagy elég csak a mutatott típus?
- Kapnak a pointer változók kezdeti (üres) értéket a deklarációnál?
- Lehetséges-e ugyanazt az adatot két (vagy több) pointeren keresztül is változtatni/elérni?

Lássuk a pointeraritmetikát az egyes nyelveken:

- CLU:
  - A CLU-ban nincs hagyományos pointer típus
  - A program végrehajt műveleteket objektumokon
  - Az objektumok mint egy univerzum részei léteznek, a program változói hivatkoznak ezekre az objektumokra
    - \* Garbage collection a felszabadításra
  - A programban kétféle objektum lehet:
    - \* mindig ugyanaz az értéke (immutable)
    - \* változhat az értéke (mutable)
- C++
  - A legtöbb T típusra, T\* a megfelelő pointer típus:
    - \* `int *p;`
  - A tömbökre és függvényekre mutató pointereknek kicsit bonyolultabb jelölése van
    - \* `int (*vp)[10];`
    - \* `int (*fp) (char, char*);`
  - A megengedett műveletek
    - \* Dereferencing: prefix\*
    - \* Dinamikus allokálás: new
    - \* Deallokálás: delete
    - \* Értékadás: =
    - \* Egyenlőség: ==
    - \* Additív műveletek: +, -
    - \* Increment, decrement: ++, --
    - \* Member ref.: .\*, → \*
  - van egy speciális operátor az "address.of" '&', ennek segítségével adhatjuk értékül változók címét pointereknek:
    - \* `int i = 10;`
    - \* `int *pi = &i;`
    - \* `int j = *pi;`
  - Az '&' operátorral létrehozhatjuk objektumok referenciáit is
    - \* egy referencia úgy tekinthető, mint egy konstans pointer, ami mindig automatikusan dereferenciát hajt végre:
      - `int &r = i;`

· r = 2;

- Az increment, decrement, .. veszélyesek is lehetnek, vigyázzunk, ne keverjük össze a jelentését!
  - \* pi++ a pointert inkrementálja, és a következő memóriacímre fog mutatni, ennek akkor van értelme, ha pi egy tömbre mutat, míg r++ inkrementálja i értékét.

- Java:

- Nincs hagyományos pointer típus
- A változókban kétféle érték tárolható
  - \* primitív értékek (egy numerikus típusból vagy egy logikai)
  - \* referencia értékek
- Az objektumokat (osztályok példányai vagy tömbök) referenciákkal kezeli.
- Ugyanarra az objektumra számos referencia hivatkozhat.
- Objektumok referenciáinak műveletei:
  - \* mező elérés, metódus hívás, casting, string concatenation, instanceof, '==', '!=' (referencia egyenlőségének vizsgálata) stb.

- Eiffel

- Itt sincsenek hagyományos pointer típusok
- A változókban kétféle érték tárolható - kiterjesztett értékek és referencia értékek
- Ugyanarra az objektumra számos referencia hivatkozhat

- C#

- A referencia típusok objektumait kezelhetjük referenciákkal.
- Egy "unsafe" környezetben egy típus lehet pointer is, erre számos művelet megengedett (pl. a ++, – is).

## 5.2. Típuskonstrukciók

- Példa

- Pascal
  - \* type <typn> = <value desc>;
  - \* type myint = integer;
- C++
  - \* typedef <value desc> <typn>;
  - \* typedef int myint;
- ADA
  - \* subtype <typn> is (new) <typ1>;
  - \* subtype Int is (new) Integer ;
- CLU
  - \* CLU
    - cluster...
  - \* Java, Eiffel, C#
    - class...

### Melyek a megengedett típus konverziók?

- Iterált
  - egy kiinduló típusból
- Direkt szorzat
  - több kiinduló típusból
- Unió
  - több kiinduló típusból

### 5.3. Tömb típusok

- "Egy tömb egy olyan adatszerkezet, amely azonos típusú elemek sorozatait tartalmazza"
- Általában egy tömb egy leképezés egy folytonos diszkrét intervallumról elemek egy halmazára, nem igazi sorozat típus.
  - Tömbnév (indexértékek)  $\rightarrow$  elem
- A diszkrét intervallum elemeit hívjuk index értékeknek.
- Az elemek száma ebben az intervallumban definiálja a tömb méretét.

### Indexelés

- Az alapművelet az indexelés
  - $A[i]$
  - Elvárás, hogy az  $A$  tömb  $i$ . elemét gyorsan el kell tudni érni!
- Vannak programozási nyelvek, ahol az elemek különböző típusúak is lehetnek (SmallTalk) de általában az elemek ugyanahhoz a típushoz, vagy egy adott típus lehetséges leszármazottai is lehetnek.

### Példák tömbökre

- ADA
  - Tömb típusok definiálhatók rögzített és megszorítás nélküli indexhatárokkal:
    - \* `type A is array(Integer range 2 .. 10) of Boolean;`
    - \* `type Vect is array(Integer range <>) of Integer;`
    - \* `type Matr is array(Integer range <>, Integer range <>) of Integer;`
  - Konkrét indexhatárokat az adott objektum deklarációjánál kell meghatározni:
    - \* `V : Vect(1 .. 30);`
    - \* `B : Matr(1 .. 2, 1 .. 4);`
  - Gyakran használják alprogramok paramétereként, sablonoknál.
  - Az index típusa tetszőleges diszkrét típus lehet, az elemek típusa tetszőleges típus
  - A definíciókat futási időben értékeli ki, az aktuális indexhatárok nem kell, hogy statikusak legyenek.
  - Tömbök szeletei is létrehozhatók:
    - \* `V(2 .. 12)`
      - `de: V(1..1)  $\leftrightarrow$  V(1)!`

- Megengedett az értékadás azonos típusú tömbök között:
  - \*  $V(1..5) := V(2..6);$
- Lehetséges értékadás tömb "aggregátorokkal" is:
  - \*  $V := (1..3 \rightarrow 1, \text{others} = );$
- Három előredefiniált string típus:
  - \* type String is array (Positive range <>) of Character;
  - \* type Wide\_String is array (Positive range <>) of Wide\_Character;
  - \* type Wide\_Wide\_String is array (Positive range <>) of Wide\_Wide\_Character;
- Vannak speciális attribútumai:
  - \* A'First/A' First(N)
  - \* A'Last/A' Last(N)
  - \* A'Range/A' Range(N)
  - \* A'Length/A' Length(N)

#### • C++

- Egy T típusra:
  - \*  $T \ x[\text{size}]$  a T típusú elemek size méretű tömbje. Az indexek 0 és size-1 között.
    - `float v[3];`
    - `int a[2][5];`
- Kezdeti érték adható:
  - \* `char v[2][5] = {{ 'a', 'b', 'c', 'd', 'e' }, { '0', '1', '2', '3', '4', '5' } };`
- A pointerek és tömbök szoros kapcsolatban vannak
  - \* egy tömbnév mindig a tömb nulladik elemére hivatkozik, így használható a pointeraritmetika tömbökre.
- Nem tudja a méretét!
- STL vector template osztály!

#### • Java

- `int[] ai;`
- `short [][] as1, as2;`
- ha a `'[]'`-t változó neve után írjuk, akkor csak ez a változó lesz a tömb:
  - \* `long l, a[];`
- Tömb létrehozása: `a = new int[20];`
- A tömb mérete lekérdezhető: `a.length`
- Kezdeti érték adható: `String[] colours = {"red", "white", "green"};`
- Egy többdimenziós tömbben az elemeknek lehet különböző mérete:

```

1      int[][] m = new int [3][];
2      for (int i = 0; i < m.length; i++) {
3          m[i] = new int[i+1];
4          for (int j = 0; j < m[i].length; j++) {
5              m[i][j] = 0;
6          }

```

- A szövegek kezelését a String és StringBuffer (StringBuilder) osztályokkal oldották meg. (karakterek egy tömbje nem string!)

- C#

- A tömb elemeinek számozása 0-val kezdődik
- Kétféleképpen lehet deklarálni
  - \* adott hosszúságú vagy dinamikus
- A nyelvben a tömbök objektumok, a deklaráció után szükség van a tömb példányosítására (new)
- Inicializálásra: {}
- Adott méretű tömb deklarálása: `int[] Tomb; Tomb = new int[3];`
- Ugyanez a tömb inicializálva: `Tomb = new int[3] {1,2,3}`
- Dinamikus tömb létrehozása inicializálással: `Tomb = new int[] {1,2,3}`
- A deklarációval egybekötött inicializáció: `int[] Tomb = new int[3] {1,2,3}`
- Ha egy tömböt nem inicializálunk, akkor a töb elemei automatikusan inicializálódnak az elem típusának alapértelmezett inicializáló értékére.
- A tömbök lehetőségei:
  - \* egydimenziós tömbök
  - \* többdimenziósak vagy négyzsögszerűek
  - \* kesztyűszerűek (tömbök tömbjei)
  - \* kevert típusúak (az előzőekből)
- Példa egy kesztyűszerű dinamikus tömbre:
  - \* `int[][] numArray = new int[][] {new int[] {1,2,3}, new int[] {2,4,6,8}};`
- minden tömb típusa `System.Array` bázistípusból "származik"
  - \* Az `Array` osztály egy absztrakt bázisosztály, de a `CreateInstance` metódusa létre tud hozni tömböket.
  - \* Ez biztosítja a műveleteket a tömbök létrehozásához, módosításához, bennük való kereséshez, illetve rendezésükhöz
- Az `Array` osztály tulajdonságait megadó függvények:
  - \* `IsFixedSize` - rögzített hosszúságú-e
  - \* `IsReadOnly` - írásvédett-e
  - \* `IsSynchronized` - a tömb elérése kizárólagos-e (szálbiztos)
  - \* `Length` - a tömb elemeinek száma
  - \* `SyncRoot` - Visszatér egy objektummal, amit a tömb szinkronizált hozzáféréséhez használhatunk
- Az `Array` osztályban még számos szolgáltatás:
  - \* `BinarySearch` - bináris keresés a tömbön
  - \* `Clear` - minden elemet töröl a tömbből és az elemszámot 0-ra állítja
  - \* `Clone` - másolatot készít
  - \* `Copy` - egy tömb részét átmásolja egy másik tömbbe, végrehajtja az esedékes típuskényszerítést és csomagolást (boxing)
- Az `Array` osztályban még számos szolgáltatás megtalálható, pl.: `CopyTo`, `CreateInstance`, `GetEnumerator`, `GetLength`, `GetLowerBound`, `GetUpperBound`, `GetValue`, `IndexOf`, `Initialize`, `LastIndexOf`, `Reverse`, `SetValue`, `Sort`

- Eiffel

- Az Eiffel tömbök az `ARRAY[G]` sablon osztály példányai.
- A stringeket a `STRING` osztály objektumai valósítják meg.

## Asszociatív tömbök

- Egy asszociatív tömb elemek egy rendezetlen halmaza, amelyet megegyező számú, kulcsnak nevezett értékek indexelnek.
- Ezeket a kulcsokat is tárolni kell
  - az elemek így (kulcs, érték) párok
- Perl
  - A hash skaláris adatok gyűjteménye, az indexek tetszőleges skalárok.
  - Ezek a kulcsok, amiket használunk az elemek elérésére.
  - A hash-eknek nincs sorrendjük.
  - A hash változók % jellel kezdődnek. A hivatkozás {}-l történik.
    - \* %szinek = ('piros' → 0x00f, 'kék' → 0x0f0, 'zöld' → 0xf00);
  - A hash változókra:
    - \* a keys függvény a kulcsok listáját adja vissza,
    - \* a values pedig az értékeket.
    - \* delete-tel lehet kulcs szerint törölni,
    - \* az each függvény végigmegy a hash-en visszaadva a kulcs-érték párokat
    - \* az exist függvény megadja, hogy egy adott kulcs szerepel-e a hash táblában.
- A Java, a C++, az Eiffel szabványos osztálykönyvtárában is megtalálhatók
- A .NET keretrendszer osztálykönyvtárában is
- Egyéb nyelvek:
  - PHP
  - Python
  - Ruby
  - Lua
  - Pike
  - stb.

## Uniók és variáns rekordok

- Az unió típusértékhalmaza az unió komponensei típusértékalmazának az uniója.
  - Pl. bútor
    - \* szék vagy asztal vagy szekrény
    - \* színe, anyaga - van mindegyiknek
    - \* egyéb speciális jellemzők - külön-külön
  - A variáns rekordok olyan direktszorzatok, ahol a direktszorzat egy - az utolsó - komponense unió.

## ”Választó” típusműveletek

- A programozási nyelvek a megbízhatóság különböző szintjén támogatják ezt az adatszerkezetet.
- Az unió típusnak van egy speciális, tag-nek nevezett komponense, és egy kiválasztási mechanizmusa, ami megadja a tag különböző értékeinek megfelelő alstruktúrákat
- A tag-et tárolja a rekord, és az alkomponensek elérhetősége ennek aktuális értékétől függ,
  - akkor ez egy ”megkülönböztetett” (discriminated) unió,
  - különben ez egy szabad (free) unió.



## Unió (Variáns rekord)

- Meg lehet-e állapítani, hogy rekord melyik változat szerint lett kitöltve?
- Ki lehet-e olvasni a kitöltéstől eltérő változat szerint?
- Szerkezete (gyakran):

```
1  case <tag-name> : <tag-type> of
2    <const1> : (<fields1>);
3    <const2> : (<fields2>);
4    ...
5    <constv> : (<fieldsv>);
```

- A szabad uniók esetében a tag mezők használata opcionális, és a fordító nem ellenőrzi a kiválasztott mező és a tárolt érték konzisztenciáját.
  - Ez a nyelv típusrendszerét megbízhatatlanná teszi.
- A megkülönböztetett unió esetében fontos kérdés, hogyan lehet új értéket adni a tag mezőnek.
  - A tag értéket beállítja a rekord létrehozásakor.
  - Míg a program képes kell legyen új értéket adni a rekord "normális" komponenseinek, a tag megváltoztatása a rekord szerkezet megváltozását vonja maga után!

## Példa az ADA nyelvből

```
1  type Állapot is (Egyedülálló, Házass, Özvegy, Elvált);
2  subtype Név is String(1..25);
3  type Nem is (Nő, Férfi);
4  type Ember (Családi_Áll: Állapot := Egyedülálló) is
5    record
6      Neve: Név;
7      Neme: Nem;
8      Születési_Ideje: Dátum;
9      Gyermekek_Száma: Natural;
10     case Családi_Áll is
11       when Házass => Házastárs_Neve: Név;
12       when Özvegy => Házastárs_Halála: Dátum;
13       when Elvált => Válás_Dátuma: Dátum; Gyermekek_Gondozója: Boolean;
14       when Egyedülálló => null ;
15     end case;
16   end record;
```

Ha ezt sokszor példányosítjuk, akkor ott lehet direkt szoroztatni.

- Hugó: Ember(Házass)
- Eleonóra: Ember(Egyedülálló)
- Ödön: Ember(Özvegy)
- Vendel: Ember(Elvált)
- Aladár : Ember;

Megszorítatlan altípus használata

- Aladár : Ember;
- Aladár := (Házass, ...);

- Aladár := Elek;
- Aladár := Húgó

Viszont bizonyos programozási nyelvekben nincs unió típus, a tervezők az osztályok és az öröklődés használatát javasolják helyette:

- SmallTalk
- Eiffel
- Java
- C#

## 5.4. Halmaz

- Speciális iterált típus
  - Mi lehet az eleme?
  - Hány eleme lehet?
  - Megvannak-e a 'hagyományos' halmazműveletek
- Pascal
  - Alaphalmaz: diszkrét típus
  - Elemek száma: max. 256
  - Értékek sorszáma csak 0..255 között
    - \* type Small\_Letters = set of 'a' .. 'z';
    - \* type Digits = set of '0' .. '9';
    - \* type Day = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);
    - \* type Days = set of Day;
    - \* var A, B : Days;
    - \* A:=[Monday, Wednesday]
  - Műveletek:
    - \* értékadás
    - \* halmazműveletek:
      - eleme-teszt (in),
      - az =, <>, részhalmaz reláció ('<=' , '>=')
      - (a '<' és '>' nem megengedett!)
      - unió ('+'), differencia ('-').
      - metszett ('\*')
    - \* Ez a precedencia-sorrend is.

## 5.5. Mikor ekvivalens két típus?

- x, y: array[0..9] of integer;
- z: array[0..9] of integer
- Strukturális ekvivalencia esetén:
  - A rekordok mezőnevei is figyelembe vannak véve, vagy csak a struktúrájuk?
  - Számít-e a rekordmezők sorrendje?
  - Tömböknél elég-e az indexek számosságának egyenlőnek lenni, vagy az indexhatároknak is egyezniük kell?

## Név ekvivalencia esetén

- Deklarálhatók-e egy típushoz, amelyekkel ekvivalens?
- Névtelen tömb - illetve rekortípusok ekvivalensek-e valamivel?

## Típuskonverziók

- Van-e, és hogyan működik?
  - az automatikus konverzió?
  - az identitáskonverzió?
  - a bővítő konverzió?
  - a szűkítő konverzió?
  - a toString konverzió?
- Java
  - identitáskonverzió
    - \* a boolean csak ez szabad
  - bővítő konverzió
    - \* byte to short, int, long, float or double
    - \* short to int, long float or double
    - \* int to long, float or double
    - \* long to float or double
    - \* float to double
  - szűkítő konverzió
    - \* byte to char
    - \* short to byte or char
    - \* char to byte or short (miért is?)
    - \* int to byte, short or char
    - \* long to byte, short, char or int
    - \* float to byte, short, char, int or long
    - \* double to byte, short, char, int, long or float
  - Változó = (név, attribútumhalmaz, hely, érték)
  - Láthatóság, Elérhetőség
  - Hogyan definiálhatunk változókat és konstansokat?

	szintaxis	példa
Pascal	var <identif>: <type>;	var i : integer
C++	<type> <identif> [= <value>];	int i = 0
Java	<type> <identif> [= <value>];	int i = 0
ADA	<identif>: <type>[:= <value>];	I : Integer i = 0
CLU	<identif>: <type>[:= <value>];	i : int i = 0
Eiffel	<identif>: [expanded]<type>;	I : INTEGER;

## Kifejezés

- Prefix jelölés: +ab
- Postfix jelölés: ab+
- Infix jelölés: a+b
- a műveletek precedencia szintjei fontosak

## Precedencia - Pascal

zárójel	()
függvényhívás	fv(..)
unáris operátorok	not, @, +, -, ^
multiplikatív operátorok	*, /, and, div, shl, shr
additív operátorok	+, -, or, xor
relációk	in <, >, <=, >=, <>
Balról jobbra kiértékelés	

## Precedencia - C++

zárójelek	()
scope	::
selection, call, size	.->, [], (), sizeof
postf, pref, compl, not...	++, -, , !, +, -, &, *, new, delete, ()
member	.*, ->*
multipl. op.	*, /, %
binary add.	+, -
shift	<<, >>
relációs	<, <=, >, >=
egyenlőség	==, !=
bitwise AND	&
bitwise excl. OR	^
bitwise OR	
logical AND	&&
logical OR	
cond. expr	?:
értékdás	=, *=, %=, +=, -=, >>=, <<=, &=, ^=, !=
throw	throw
comma (sequence)	,

## Precedencia - Java

zárójelek	()
scope	::
selection, call, size	.->, [], (), sizeof
postf, pref, compl, not...	++, -, ~, !, +, -, &, *, new, delete, ()
postfix	., [], (), ++, -
prefix	++, -, ~, !, +, -
constr, cast	new()
multipl. op.	*, /, %
binary add.	+, -
shift	<<, >>, >>>
reláció	<, <=, >, >=, instanceof
egyenlőség	==, !=
bitwise AND	&
bitwise excl. OR	^
bitwise OR	
logical AND	&&
logical OR	
cond. expr	?:
értékkadás	=, *=, /=, +=, -=, >>=, <<=, &=, ^=, !=

## 6. Alprogramok, Eljárások és függvények, paraméterek fajtái, átadás-átvételi módok, túlterhelés, rekurzió (példák: Java, C++, Ada/SWIFT)

### Alprogramok, mint absztrakció

- Absztrakció a programozásban: különböztessük meg a szinteket:
  - Mit csinál a program?
  - Hogyan van implementálva?
- Minden programozási nyelv a gépi kód absztrakciója
- Magasabb szintű absztrakciók:
  - Eljárás: Mit csinál az eljárás?
    - \* Hogyan csak akkor fontos, amikor implementáljuk.
  - Eljárást hívó eljárások
    - \* Az absztrakció akárhány szükséges szintje bevezethető.
- Absztrakciós mechanizmus
  - a programozási nyelvi konstrukció, ami megengedi, hogy a programozó megragadja az absztrakciót, és a program részeként reprezentálja - egyfajta számítási mintaként
    - \* Egyszerű példa:
    - \* Eljárások és függvények
    - \* Egyéb példák: később..

### A legegyszerűbb absztrakciós mechanizmus

- Eljárások és függvények: egységek, melyek számításokat tartalmaznak
  - Függvény-absztrakció: egy kiszámítandó kifejezést tartalmaz
  - Eljárás-absztrakció: egy végrehajtandó parancsot tartalmaz.
- A tartalmazott számítás mindig végre lesz hajtva, amikor egy absztrakciót hívtunk.
- Az érdekeltségek szétválasztása:
  - A Hívó azzal törődik, mit csinál a számítás.
  - Az implementáló azzal is törődik, hogy hogyan kell a számítást végrehajtani természetesen a "mit" szerint
- A hatékonyságot a paraméterezéssel javítjuk.

### 6.1. Függvény-absztrakció

- Egy kiértékelendő kifejezés - amikor hívjuk, egy értéket ad vissza
- ADA példa

```
1  function Kerulet (R : Float) return Float is
2  begin
3      return 2.0 * R * 3.14;
4  end;
```

## Függvény definíció

```
function I (FP1; ...; FRn) return T is body
```

- Egy I azonosítót hozzákapcsol egy adott függvény-absztrakcióhoz.
- A függvény-absztrakció a megfelelő aktuális paraméterekkel való híváskor eredményt ad vissza.
  - A függvényhívás felhasználói szemlélettel egy leképezés az argumentumok és az eredmény között.
  - Megvalósítói szemlélet: a függvénytörzs kiértékelése. Az algoritmus változása csak a megvalósítóra tartozik.

## 6.2. Eljárás-absztrakció

- Egy végrehajtandó parancsot testesít meg.
  - A felhasználó csak a változók megváltozását érzékeli
- Sok nyelvben nem lehet létrehozni egy eljárás-absztrakciót név nélkül. Az általános formátum:
  - procedure I (FP 1; ...; FP n) body
- Felhasználói szemlélet
  - type Dictionary = array[...] of Word;
  - procedure sort (var words : Dictionary);
  - ...
  - sort(a); (\* érzékeljük 'a' változását \*)
- Megvalósítói szemlélet: a kódolt algoritmus

## Absztrakciós elv

- Függvény-absztrakció: egy kifejezés absztrakciója.
  - Egy függvényhívás egy kifejezés, amikor hívjuk, kiértékeli, és egy értéket ad vissza.
- Eljárás-absztrakció: egy parancs absztrakciója.
  - Egy eljáráshívás egy parancs, ami az eljárástörzs végrehajtása során frissíti/frissítheti a változók értékét.
- Általánosítás:
  - Tetszőleges szintaktikai osztály fölött létrehozhatunk absztrakciókat, feltéve, hogy ez valamifajta számítást specifikál.
    - \* Absztrakt adattípusok
    - \* Generic

## 6.3. Alprogramok és paraméterek

- Alprogramok használata - az egyik legelső programozási eszköz
- Charles Babbage - Analytical Engine - 1840-ben már tervezte, hogy lyukkártyák egy csoportját fogja használni nagyobb számítások gyakrabban ismételt részeinél.
- Alprogramok használatlával nevet adhatunk egy kódrészletnek és paraméterezhetjük a viselkedését.

## Paraméter átadása

- Absztrakció általánosítása

```
1  val pi = 3.1415926535;  
2  val r = 1.0;  
3  fun perimeter() = 2*pi*r;  
4  
5  fun perimeter (r: real) = 2*pi*r;  
6  perimeter(1.0);  
7  perimeter(a+b);
```

## Alprogram

- Progamegység
- Végrehajtás kezdeményezése: meghívással
- A program darabolásának eszköze
- Különböző számítások elkülönítése egymástól
- Újrafelhasználhatóság
- Alprogram hívása: szokásosan, paraméterátadás, és függvény scope...
- alprogram = (név, paraméterek, környezet, törzs).
- az alprogram definíciójakor a formális paraméterekkel írjuk le az adatcsere elmeit, a külvilággal való kommunikáció alterének komponenseit
- Az alprogram használatakor pedig az aktuális paraméterek kerülnek ezek helyére
  - ez a paraméteresátadás

## Függvény

```
1  function Faktoriális (N: natural) return  
2  Positiva is  
3    Fakt : Positive := 1;  
4  begin  
5    for I in 1..N loop  
6      Fakt := Fakt * I;  
7    end loop;  
8    return Fakt;  
9  end Faktoriális;
```

## Eljárás

```
1  procedure Cserél (A, B: in out Integer) is  
2    Temp: Integer := A;  
3  begin  
4    A := B;  
5    B := Temp;  
6  end Cserél;
```



## 6.4. Paraméterek fajtái

- Az információáramlás iránya szerint
  - Input: hívó  $\rightarrow$  alprogram
  - Output: hívó  $\leftarrow$  alprogram
  - Update: hívó  $\leftrightarrow$  alprogram

### Paraméterátadási technikák

- A paraméterátadás módja
  - Különféle nyelvekben különféle módon adják át a paramétereket
  - Legismertebb paraméterátadási módok:
    - \* érték szerint
    - \* cím szerint
    - \* eredmény, érték-eredmény szerint
    - \* név szerint

#### 6.4.1. Érték szerinti paraméterátadás

- A formális paraméter az alprogram egy lokális változója, aminek az aktuális paraméter adja a kezdőértéket
  - in módú átadásra alkalmas
- Az aktuális paraméter tetszőleges kifejezés, kiértékelésére egyszer, az alprogram végrehajtásának megkezdése előtt kerül sor.
- Az alprogram végrehajtása közben sem az aktuális paraméter értéken esetleg megváltozása nincs hatással a formális paraméter értékére, sem a formális paraméter értékének megváltoztatás nincs hatással az aktuális paraméterre.
- pl C:

```
1  int lnko (int a, int b) {  
2      while (a != b )  
3          if (a>b) a-=b;  
4          else  b-=a;  
5      return a;  
6  }
```

- Csak egyszer értékelődik ki

#### 6.4.2. Cím szerinti paraméterátadás

- Az aktuális paraméter vagy egy változó, vagy egy változó komponensét meghatározó kifejezés - balérték - ( $x[i+2*j]$ ) lehet.
- Az aktuális paraméter kiértékelése a hozzárendelt memóriaterület címének meghatározását jelenti.
- Az aktuális paraméter kiértékelésére az alprogram végrehajtásának megkezdése előtt kerül sor, s az aktuális paraméter memóriaterülete rendelődik hozzá.
- Az alprogramban hivatkozhatunk is a formális paraméter értékére, és adhatunk is neki új értéket (update módú átadásra is alkalmas)
- Példa Pascalban:

```

1  procedure Csere(var a, b: Integer);
2  var temp: Integer;
3  begin
4      temp := a;
5      a := b;
6      b := temp;
7  end;

```

- "Alias" - amikor egy pontján két különböző változó ugyanazt az egyedet jelenti

```

1  var globalis : Integer;
2  procedure r (var lokalis : Integer);
3  begin
4      globalis := globalis + 1;
5      lokalis := lokalis + globalis;
6  end r;
7  ... globalis := 1;
8  r(globalis);

```

Mennyi lesz globalis értéke?

#### 6.4.3. Eredmény szerinti paraméterátadás

- A kimenő paraméterek megvalósításához.
  - Az alprogram formális paraméterében kiszámított eredményt helyezi vissza az aktuális paraméterbe.
- A formális paraméter, csakúgy, mint az érték szerinti paraméterátadás esetén, az alprogram lokális változója, melynek értéke az alprogram befejeződésekor kimásolódik az aktuális paraméterbe.
- Az aktuális paraméter balérték kell legyen.
- A formális paraméter nem kapja meg az aktuális paraméter értékét az alprogram hívásakor, ezért az információáramlás egyirányú (out módú átadásra alkalmas).
- ADA példa:
  - procedure Get(Item: out Character);
- Object Pascal példa:
  - procedure GetInfo(out Info: SomeRecordType);

#### Érték/eredmény szerinti paraméterátadás

- Az alprogram befejezésének pillanatáig megegyezik az érték szerinti paraméterátadással.
- Az alprogram végrehajtásának befejezésekor az aktuális paraméter felveszi a formális paraméter pillanatnyi értékét (update módú átadásra alkalmas).
- Az aktuális paraméter csak "balérték" lehet
- Miben különbözik a cím szerinti átadástól?

#### 6.4.4. Név szerinti paraméterátadás

- Az aktuális paraméterként leírt teljes kifejezés adódik át és minden használatkor (dinamikusan!) kiértékelődik.
  - például az a[i] hivatkozás változhat, ha menet közben az i értéke változik(!)
  - (update módú átadásra is alkalmas)
- Lusta kiértékelés...

```

• procedure Paramproba is
  ... A,B:Integer;...
  procedure Parcsere(X,Y: in out Integer) is
    Temp:Integer;
    begin
      Temp:=X; X:=Y; Y:=Temp;
      A:=12; B:=99; -- ...
    end;
  begin
    A:=1;B:=2;
    Parcsere(A,B);
  end;

procedure s (a: in out Integer)...
  - kiírja és lenullázza a paraméterét
procedure p (a, b: in out Integer)
  - kiírja és lenullázza a paramétereit
begin
  s(a);
  s(b);
end;

globalis: Integer;
procedure r (lokalis: in out Integer);
begin
  globalis := globalis + 1;
  lokalis := lokalis + globalis;
end r;
...
globalis := 1;
r(globalis);      itt mennyi lesz a globalis?

```

9. ábra. Érték/eredmény szerinti paraméterátadás

```

real procedure sum (expr, i, low, high);
  value low, high;
  real expr;      -- név szerint az expr és az i
                  -- ez a default Algolban!
  integer i, low, high;
  begin
    real rtn;
    rtn := 0;
    for i := low step 1 until high do
      rtn := rtn + expr;
    sum := rtn;
  end sum

```

10. ábra. Egy híres példa - Jensen's device kifejezések kiértékelésére - algol60

### Alprogramok paramétere

- Az alprogramok paramétereinek száma általában kötött, de lehet változó is.
- Egy szintig szimulálható a változó számú paraméter egy tömb átadásával, de ezzel nem mindig oldható meg eltérő típusú paraméterek átadása.

### Paraméterek száma

- A programnyelvek kezelhetik a túl sok vagy túl kevés aktuális paraméter megadását.

- A túl kevés paraméter megadás esetén több módszer alkalmazható:
  - alapértelmezett értékek adhatók meg, amelyek a hiányzó aktuális paraméterek helyébe lépnek (pozíció vagy név szerint)
  - vesszőket kell tenni a kihagyott paraméterek helyett
  - amíg nincsenek alapértelmezett értékek, addig kötelező az aktuális paramétereket megadni, tehát az alaptértelmezések csak a paraméterlista végén lehetnek

### Formális-aktuális paraméterek megfeleltetése

- procedure Get\_Line (File : in File\_Type; Item : out String; Last : out Natural)
- pozícionálás formában: az aktuális paramétereket abban a sorrendben kell felsorolni, ahogy a formális paraméterek voltak az alprogram specifikációjában: Get\_Line(F,S,N);
- névvel jelölt formában: a paraméterek sorrendje tetszőleges: Get\_Line(File → F, Last → L, Item → S);
- kevert formában: mindig a pozícionálisan megadott paramétereknek kell elől állniuk: Get\_Line(F, Last → L, Item → S);

### Paraméterek feltételezett értéke

- Az in módú paraméterekhez rendelhetünk feltételezett bemenő értéket.
- Az alprogram specifikációs része tartalmazza ezt a feltételezett értéket, amit akkor vesz fel a formális paraméter, ha neki megfelelő aktuális paramétert nem adunk meg.

## 6.5. Alprogramok túlterhelése (átlapolása)

```

1  procedure Cserél (A, B: in out Integer);
2  procedure Cserél (A, B: in out Boolean);

```

- Azonos név - paraméterek száma és/vagy típusa különböző
- A használatból (a hívásból) fog kiderülni, - kell kiderülnön! - hogy melyikre gondolunk

### Kérdések

```

1  z:= sin(x)
2  y:= sin(x)
3  z = y?
4
5  zz:= rnd();
6  xx:= rnd();
7  zz = xx ?

```

- Mellékhatás 1: globális változók használata

```

1  with Text_IO; use Text_IO;
2  procedure Globprobe is
3      I:Integer;
4      function Glob(J:Integer) return Integer is
5          begin
6              I:=I+1; return I+J;
7          end;
8  begin
9      i:=2;

```

```

10 Put_Line(Integer ' Image (Glob(1)));
11 ...
12 Put_Line(Integer' Image (Glob('')));
13 end;

```

- **Mellékhatás2: függvény paraméter is változik**

```

1  int f(int val, int &ref) {
2      val++;
3      ref++;
4      return val+ref;
5  };
6  void main() {
7      int i = 1;
8      int j = 1;
9      int k ;
10     k = f(i, j);
11 }

```

## Operátorok

- infix alakban is írható műveletek A+42 "+"(A, 42)
- Operátorok is átlapolhatók:

```

1  function "+" (A, B: Mátrix) return Mátrix;

```

## 6.6. Rekúzió

- Közvetlenül vagy közvetve önmagát hívó alprogram

```

1  function Faktoriális (N:Natural) return Positive
2  is
3  begin
4      if N > 1 then
5          return N * Faktoriális(N-1);
6      else return 1;
7      end if;
8  end Faktoriális

```

- A rekurzív alprogramok paraméterátadás szempontjából nem különböznek a szokásos alprogramoktól.
- Ezért a paramétereket rekurzív alprogramoknak cím szerint átadni csak elővigyázatosan szabad, mert a sorozatos hívások interferálhatnak.

## Példák

- Algol
  - Algol60: Az alprogramok paramétereit alapértelmezésben név szerint veszi át, de a value kulcsszó használatával érték szerinti lesz az átadás.
  - 1966-ban lett az eredmény szerinti paraméterátadás (result) is.
  - Algol68: Az érték szerinti paraméterátadás lett az alapértelmezés, a cím szerinti pedig a ref kulcsszóval lehet elérni.
    - \* Nincs se név, se eredmény szerinti.
  - Van operátor átdefiniálás

\* (sőt, a precedenciák is változhatnak!)

- Pascal

- A paraméterek alapértelmezésben érték szerint adódnak át, de a VAR kulcsszóval cím szerinti átadás érhető el:

```
1      program A;
2          procedure Megszoroz (Var Mit: Integer; Szorzo:Integer);
3              begin
4                  Mit:= Mit * Szorzo;
5              end;
6      Var N, K :Integer;
7      begin
8          N:= 5; K:= 3;
9          Megszoroz (N,4); -> N = 20
10         Megszoroz (N,K); -> N = 50, K = 3
11         Megszoroz (5,N); -> HIBA!
12     end.
```

- C/C++

- csak függvények van - void visszatérési érték, ha eljárást szeretnénk
- csak érték szerinti paraméter átvétel, cím szerint: mutatók vagy referencia kell. Lehet a paraméterek számára is kezdőértéket adni.

```
1      void f(int val, int& ref){
2          val++;
3          ref++;
4      }
5
6      void g() {
7          int i=1;
8          int j=1;
9          f(i,j);
10     }
11
12     i==1
13     j==2 lesz.
```

- C++

- const& paraméter - nem változtatható a törzsön belül

```
1      float fsqr(const float&);
2      ..
3      float r = fsqrt(x);
```

- Tömb paraméterek

- \* a tömb első elemére hivatkozó mutató adódik át
- \* vagyis nem érték szerint adódnak át
- \* méretük nem adódik át

```
1      int strlen(const char*);
2      void f(){
3          char v[] = 'egy tomb';
4          int i = strlen(v);
5      }
```

- \* túlterhelés lehetséges

- Java

- Csak valamely osztály metódusa lehet
- Az aktuális objektum attribútumaira hivatkozik
- Paraméterátadás
  - \* primitív típusok érték szerint
  - \* összetett típusú paraméterek referencia szerint
    - mutable-immutable lehetőség!
- SmallTalk
  - Üzenetküldésekkel érjük el a metódusokat, így a paraméterátadás érték szerintinek tekinthető - referenciák figyelembevételével
    - \* A következő sor az az objektumnak küld üzenetet, két paraméterrel a két paraméter megnevezésével és aktuális értékével
    - \* a at:3 put:5
- ADA
  - in, out, in out  
Az összetett típusú értékek esetén a fordítóprogram választhat az érték-eredmény, illetve a cím szerinti átadás között.
  - A függvényeknek csak in paramétereik lehetnek
  - Az in paramétereknek lehet alapértelmezett értékük is
  - Az alprogramok határozatlan méretű tömb típust is elfogadnak a formális paramétereik között.
- Swift
  - Konstans paraméter, inout paraméter, kevert formába történő átadás, feltételezett érték létezik, függvényt is át lehet adni paraméterként, változó számú paraméter átadásának lehetősége

```

1 func greet(name: String, day: String) -> String {
2     ...
3 }

```

- Több visszatérési érték is lehet:

```

1 func calculateStatistics(scores: [Int]) ->
2     (min: Int, max: Int, sum: Int) {
3     ...
4     return (min, max, sum);
5 }

```

## 7. A kivételkezelés

**Alapfogalmak, kivételek kiváltása, terjedése, kezelése, specifikálása, kivételosztályok (példák: Java, C++, Eiffel)**

### 7.1. Kivételek kiváltása - hibalehetőségek

- A programok futása közben hibák léphetnek fel.
- Szoftverhibák
  - a futtatott programban
    - \* kísérlet nullával való osztásra
    - \* aritmetikai túlcsordulás, alulcsordulás
    - \* üres pointer/referencia dereferencia kísérlete
    - \* hibás típus konverzió kísérlete
    - \* tömb hibás indexelése
    - \* hibás input
  - az operációs rendszerben meglévő programozói hibák

#### Elvárások:

1. meg tudjuk különböztetni a hibákat,
2. a hibát kezelő kód különüljön el a tényleges kódtól,
3. megfelelően tudjuk kezelni - a hiba könnyen jusson el arra a helyre, ahol azt kezelni kell,
4. kötelező legyen kezelni a hibákat

#### Hiba terjesztése hagyományosan

- Tegyük fel, hogy a jonatan()-ban fellépő hibát gyumolcs() kell lekezelje:

```
1  gyumolcs {
2      ret= alma();
3      if (ret!=ok) {
4          //hibakezelés
5      }
6      else {
7          //folytatás
8      }
9  }
10
11  alma {
12      ret=jonatan();
13      if (ret!=ok) {
14          return hiba;
15      }
16      else {
17          //folytatás2
18      }
19  }
20
21  Jonatan {
22      //műveletek
23      if (művelethiba) {
24          return hiba;
25      }
26  }
```



- Kivételkezeléssel a felfelé terjesztést egyszerűen és kevesebb módosítással lehet megoldani:

```

1      gyumolcs {
2          try {
3              alma();
4              //folytatás
5          }
6          catch(VmiExc) {
7              //hibakezelés
8          }
9      }
10
11     jonatan throws VmiExc {
12         //műveletek
13     }
14
15     alma throws VmiExc {
16         ...
17         jonatan();
18         //folytatás
19     }

```

## Kérdések

- Hibát - vagy kivételkezelést ad-e a nyelv?
- Hogyan kell kivételeket definiálni-kiváltani-kezelni?
- Milyen a kivételkezelés szintaktikai formája
- Mihez kapcsolódik a kezelés: utasítás-, blokk - vagy eljárás/függvényszintű?
- A kivételekből lehet-e kivételcsoportokat, kivételosztályokat képezni, amelyeket a kivételkezelőben egységesen lehet kezelni?

## C++

- Kivétel lehet egy objektum, aminek tetszőleges a típusa.
- Hasznos, ha definiálunk osztályokat a felhasználó kivételeihez
- Példa
  - `class MyException ;`
- néhány predefinit kivétel:
  - `bad_cast`: ha a `dynamic_cast` operátor nem használható egy referenciára,
  - `bad_typeid`: ha a `typeid` operátora egy null pointer dereferenciája. (mindkettő a `typeinfo.h`-ban deklarálva).
- Kivétel kiváltása
  - `throw [expression] ;`  
A kivételkezelés egy időszakos objektumba kerül
  - A normál program lefutás megszakad, a vezérlés a legközelebbi megfelelő kezelőhöz kerül
  - ha nincs kifejezés: csak a kivételkezelőben, illetve innen hívott függvényen lehet, az aktuális kivétel újrakiváltása.
  - A fordító nem biztos, hogy ellenőrzi, ha `throw` utasítás kivétel-objektum nélkül: `terminate` függvény hívása

- Példák

- throw 5;
- throw "An exception has occurred";
- throw MyException();
- throw;

- További információk: a kivételosztályban lehetnek nyilvános attribútumok, konstruktor beállíthatja stb.

```

1      class ExceptionWithParameters {
2      public:
3          ExceptionWithParameters(int a0, int b0) {
4              a = a0; b = b0;
5          }
6
7      };
8      throw ExceptionWithParameters(0,1);

```

- Kivételek kezelése - try blokkal:

- try compound\_statement handler\_list, ahol handler\_list = handler{handler}
- handler = catch "(" exception\_declaration ")" compound\_statement exception\_declaration = type [ident] | "..."

- A dobott kivételeknek megfelel egy catch ág, ha a következő feltételek közül valamelyik teljesül:

- A két típus pont ugyanaz
- A catch ág típusa publikus bázisosztálya az eldobott objektumnak
- a catch ág típusa mutató, és az eldobott objektum olyan mutató, melyet valamely standard mutató konverzióval át lehet konvertálni a catch ág típusára.

- A kezeletlen kivételek továbbgyűrűződnek a hívó try blokkban, majd az azt hívóban. A legkülső try blokk után a terminate függvény kerül meghívásra.

- Lehet a kivételobjektumra is hivatkozni:

```

1      catch (ExceptionWithParameters e) {
2          cout << e.a; // kiírja
3      }

```

- Lehet...

- ez bárminek megfelel - utolsó legyen a try-blokkban

- A terminate hívása általában abort-ot jelent, de a felhasználó a set\_terminate-tel megadhat mást is, de ez is le kell állítsa a programot.

- Miközben a vezérlés átadódik a kivételkezelőnek, destruktort hív minden automatikus objektumra, ami a try-blokkba való belépés óta keletkezett.

- Kivétel specifikációk

- throw "(" [type {"", type}] ")"
- pl: void f(int x) throw(A,B,C);

- Az f függvény csak A, B és C típusú kivételt generál, vagy azok leszármazottait.

- `int f2(int x) throw()`
- `void f3 (int x)`
- Ha specifikáltunk lehetséges kivételtípusokat, akkor minden más esetben a rendszer meghívja az `unexpected()` függvényt, melynek alapértelmezett viselkedése a `terminate()` függvény meghívása. Ez a `set_unexpected` segítségével szintén átállítható.
- Példa:

```

1  class X {};
2  class Y: public X {};
3  class Z {};
4  void A() throw(X,Z)
5  { ... }
6  void B() throw(Y,Z)
7  {
8      A();
9  }
10 void C();

```

## Java

```

1  try {
2      ...
3      throw new EgyException ("parameter");
4  }
5  catch (típus változónév) {
6      ...
7  }
8  finally {
9      ...
10 }

```

- A C++ szintaxistól nem sokban különbözik
  - Eltérések a szemantikában.
- A továbbiakban csak az eltérések:
  - `finally` nyelvi konstrukció, ezzel a Java megbízhatóbb programok írását segíti elő.
- Egy függvény által kiváltható kivételek specifikálása:

```

1  void f(int x) throws EgyikException,
2  MasikException

```

- Egy szálon futó program esetén ha a virtuális gép nem talál a kivétel kezelésre alkalmas kódot, akkor a VM és a program is terminál.
  - Ha többszálú a program, akkor egy `uncaughtException()` metódus fut le.
- Minden kivétel a `java.lang.Throwable` leszármazottja.
- Ha olyan kivételt szeretnénk dobni, amely nem a `Throwable` leszármazottja, akkor a fordítási hibát okoz.
- A kivételek két nagy csoportba sorolhatóak
  - ellenőrzöttek: `Exception` leszármazottjai
  - nem-ellenőrzöttek: `Error` leszármazottjai

- Azért volt arra szükség, hogy a kivételeket a fenti két csoportba sorolják, mert számos olyan kivétel van, amely előre nem látható és fölösleges lenne mindenhol lekezelni őket.
  - Ezeket a kivételeket nevezzük nem-ellenőrzött kivételeknek
  - Például nem ellenőrizzük le minden utasítás végrehajtása előtt, hogy van-e elég memóriánk stb.
- Sajnos, a Java a fenti két csoportosítást nem konzisztens módon végzi, ugyanis az Exception egyik gyermek osztálya, a RuntimeException és leszármazottjai sem ellenőrzöttek.
- Az ellenőrzött kivételek esetén fordítási hiba lép fel, ha nincsenek specifikálva vagy elkapva; illetve ha olyan ellenőrzött kivételt kívánunk elkapni, amely hatókörön kívül van.
- Néhány predefinit nem ellenőrzött kivétel
- A RuntimeException leszármazottjai
  - ArithmeticException
  - ClassCastException
  - IndexOutOfBoundsException
    - \* ArrayIndexOutOfBoundsException
    - \* StringIndexOutOfBoundsException
  - NullPointerException
- Az Error leszármazottjai
  - OutOfMemoryError
  - StackOverflowError
  - stb.

- Predefinit kivételek előfordulása:

```

1  class A { ...
2  }
3  class B extends A { //...
4  }
5  class C {
6      void X() {
7          A a = new A;
8          B b = (B) a; //ClassCastException
9      }
10     void Y() {
11         int ia[] = new int [10];
12         for (int i=1; i<=10; i++) ia[i]=0;
13         //10nél ArrayIndexOutOfBoundsException
14     }
15     void Z() {
16         C c = null;
17         c.X(); //null pointer exception
18     }
19 }

```

- Kivételek kezelése

```

1  try {
2      //utasítások
3  }
4  catch (MyException e) {
5      //utasítások MyException kezelésére
6  }

```

```

7      catch (AnotherException e) {
8          //utasítások AnotherException kezelésére
9      }
10     finally {
11         //Mindig végrehajtódó utasítások
12     }

```

- Gyártás

```

1      class ExcA extends Exception {}
2      void A() throws ExcA {
3          //..
4          throw new ExcA();
5          //..
6      }

```

## Eiffel

- Újrakezdés

- A komponens írásakor egy kivétel lehetőségét előre lehet látni és egy alternatív megoldást találni a szerződés betartatására.
- Ekkor a végrehajtás megpróbálja ezt az alternatívát.

- Szervezett pánik

- Ha nincs rá mód, hogy teljesítsük a szerződést, akkor az objektumokat egy elfogadható állapotba kell hozni (típusinvariáns helyreállítása), és a felhasználónak jelezni kell a kudarcot.

- Eiffel - Újrakezdés

```

1      try_once_or_twice is
2          local
3              already_tried : BOOLEAN
4          do
5              if not already_tried then
6                  method_1
7              else
8                  method_2
9              end
10         rescue
11             if not already_tried then
12                 already_tried := true;
13                 retry
14             end
15         end -- try_once_or_twice

```

- Eiffel szervezett pánik

```

1      attempt_transaction(arg : CONTEXT) is
2          --megpróbáljuk a transaction-t arg argumentummal;
3          --ha nem sikerül, az akt. obj. invariánság visszaállítjuk
4          require
5              ..
6          do
7              ..
8          ensure
9              ..
10         rescue
11             reset(arg)
12         end -- attempt_transaction

```

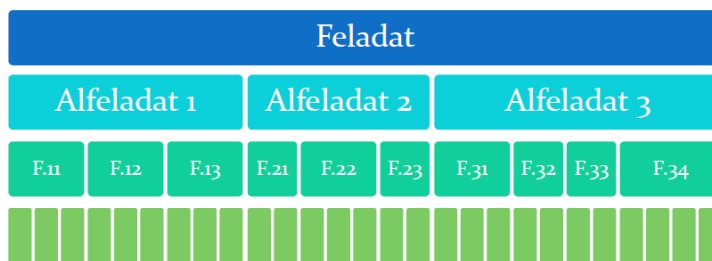
## 8. Absztrakt adattípusok a programozási nyelvekben

**Procedurális- és adatabsztrakciós megközelítés, elvárások és eszközök (példák: Java, C++, Ada)**

**Sablonok – Típussal, alprogrammal való paraméterezés, példányosítás, sablon-szerződés modell (példák: Java, C++, Ada, Eiffel)**

### 8.1. Procedurális absztrakció

- Top-down megoldás
  - A megközelítés hátránya: ha változik a specifikáció
- Mivel egy alprogram specifikáció a magasabb szintű specifikációtól függ, így a változás tovább gyűrűzik lefelé, és végül egy egész kis változásnak nagyon nagy hatása (és költsége) lehet
  - Specifikáció változásának hatására



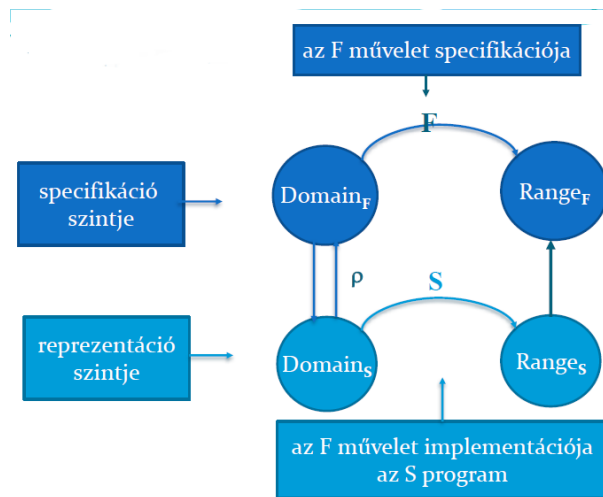
11. ábra. Procedurális absztrakció

### 8.2. Adatabsztrakció

- Ha elég magas szintről analizáljuk a problémát, úgy tűnik, a rendszerek jobban jellemezhetőek hosszú távon objektumaikkal, mint a rájuk alkalmazott tevékenységekkel.
- Ezt a megközelítést választva, először megpróbáljuk jellemezni az objektumokat, és az objektumok osztályait (~ az adattípusokat), amelyek szerepet játszanak a rendszerben. Az új adattípusokat a már meglévők felhasználásával tervezzük - ez a bottom up tervezés- Ez azt jelenti, hogy az elérhető komponensek szintjéről indulunk, abból építkezünk.
- Ezután oldjuk meg a problémát

#### Típus a programozó szemszögéből

- Típus-specifikáció ("mit")
  - alaphalmaz: a valós világ minket érdeklő objektumainak halmaza
  - specifikációs invariáns ( $I_S$ ) - ezt a halmazt szűkíti
  - típusművelet specifikációja
- Típus megvalósítás ("hogyan")
  - Reprezentációs függvény
  - Típus invariáns
  - Típusműveletek megvalósítása



12. ábra. A típus specifikáció és a típus kapcsolata

### 8.3. Elvárások és eszközök

- Elvárások a programozási nyelvekkel szemben
- Absztrakt típusok megvalósításának eszközei

#### Elvárások a programozási nyelvekkel szemben

- Modularitás
  - Az egyes típusokat önálló fordítási egységekben lehessen megvalósítani!
  - Ez biztosítja a típusok újrafelhasználhatóságát, valamint a hatékony programfejlesztést, hiszen az egyes modulok könnyedén átvihetők más programokba, és a különböző egységeket más-más programozó is fejlesztheti, nem zavarva egymás munkáját.

#### Absztrakt típusok megvalósításának eszközei

- Modulokra bontás
  - A professzionális használatra szánt programozási nyelvek mindegyikében megjelenik a modularizáció támogatása.
  - A modernebb nyelvekben a modularizáció alapján egyre inkább a típusokra bontást jelenti, azaz egy modul egy típust implementál.
  - Teljesen ezen az alapon csak kevés nyelv működik
    - \* sokszor van szükség "alprogram könyvtárak"-ra
    - \* szükség lehet csak implementációs célokat szolgáló típusok megvalósítására is, amelyeket ilyenkor lehetőség szerint az azt használó adattípus moduljában rejtünk el.

#### Elvárások a programozási nyelvekkel szemben

- Adatrejtés
  - A nyelv támogassa a reprezentáció elrejtését
  - Ezen eszköz segítségével a nyelv maga biztosítja, hogy az adott típus használója csak a specifikációban megadott tulajdonságokat használhassa ki.

- Ez a megszorítás lehetővé teszi a reprezentáció, illetve az implementáció megváltoztatását anélkül, hogy a változások a programban felfelé gyűrűznének.

### **Absztrakt típusok megvalósításának eszközei**

- A reprezentáció elrejtése
  - Az adatrejtés támogatására a nyelvek gyakran az összetett típusok komponenseinek láthatóságát szintekre bontják, ezek meghatározzák, hogy pontosan kik férhetnek hozzá az adott komponenshez.
  - Leggyakrabban három szintű:
    - \* nyilvános (public) - az adott komponens mindenki számára látható
    - \* védett (protected) - az adott komponens csak a leszármazottak számára látható
    - \* privát (private) - a reprezentáció teljesen rejtett része, csak a műveletek implementációjában használható komponensek.
- Adatrejtéshez új láthatósági szintek bevezetése
  - Java - package-private
  - Delphi - published (futás idejű típusinformáció), strict private, strict protected
  - C++ - friend
  - stb..
- Típus vagy objektum szinten szabályoz-e? (C++, SmallTalk)
- Vannak olyan nyelvek amelyekben egyáltalán nincs ilyen jellegű szabályzás (Python, bár itt azért vannak konvencionális dolgok...)
- Egyes nyelvekben (Eiffel) a láthatóság még ennél is finomabban szabályozható, a típus megvalósításakor rendelkezhetünk arról, hogy mely osztály (és leszármazottai) férhessenek hozzá az adott komponenshez.

### **Elvárások a programozási nyelvekkel szemben**

- Konzisztens használhatóság
  - A felhasználói és a beépített típusok ne különbözzenek a használat szempontjából!
  - A típusokat minél inkább a valós világban megszokott, "természetes" módon lehessen kezelni!
  - Ugyanúgy lehessen összetett típusokat (tömböket, rekordokat stb.) definiálni a segítségükkel, ugyanúgy lehessen változókat definiálni a saját típusokkal stb.

### **Ezzel kapcsolatban visszautalás - Absztrakt típusok megvalósításának eszközei**

- Konzisztens használat
  - Az új típust be lehessen illeszteni a nyelv logikájába!
  - Ha például az adott nyelvben az a konvenció, hogy egy típust a Read művelet olvas be, akkor fontos, hogy a saját típusokhoz is definiálhasson a programozó egy Read nevű műveletet, azaz legyen lehetőség a Read azonosító túlterhelésére, vagy átlapolására
  - Egy azonosító túlterhelése vagy átlapolása (overloading) azt jelenti, hogy a programszöveg egy adott pontján az azonosítónak több definíciója is érvényben van.
  - Speciálisan az operátor-túlterhelésnek nagy jelentősége van abban, hogy a felhasználói típusokat természetes használatuknak megfelelően használhassuk.



## Elvárások a programozási nyelvekkel szemben

- Generikus programsémák támogatása
  - A programozó lehetőleg minél általánosabban írhasa meg a programjait!
  - A nyelv adjon lehetőséget az ismétlések minimalizálására, nem csak közvetlen kódismétlések elkerülésére, hanem a magasabb szintű megoldási struktúrák többszörös megírásának elkerülésére.
  - Ez nagyban javítja a kód olvashatóságát és karbantarthatóságát.
- Java - generikusok
- C++ - template struktúra

## További elvárások a programozási nyelvekkel szemben

- Specifikáció és implementáció szétválasztása külön fordítási egységbe
  - Ekkor az adott típust használó más modulok a típus specifikáció birtokában elkészíthetők, függetlenül a tényleges implementációtól.
- Modulfüggőség kezelése
  - A fordító program kezelje maga a modulok közötti relációkat (egyik használja a másikat, stb.).

## Erre is egy megvalósítási eszköz

- Specifikáció és implementáció szétválasztása
  - Azon nyelvekben, melyek támogatják az "egy modul egy típus" elvet, néha lehetőség van a típusspecifikációnak az implementációtól külön, önálló fordítási egységben történő leírására.
  - Ez segíthet abban, hogy az egyes modulok egymástól függetlenül elkészíthetők legyenek.

## Példák

### C/C++

- A C/C++ nyelvekben a specifikációt fizikailag be kell másolni minden olyan fordítási egységbe, amely az adott típust használni akarja, így persze itt nem beszélhetünk igazi szétválasztásról
  - A bemásolás megkönnyítésére a specifikáció egy külön, speciális forrásfájlban (header fájl) leírható és az előfordító segítségével azt beemelhetjük a megfelelő fordítási egységekbe.
  - Ha egy típus specifikációját többször is beírjuk egy fordítási egységbe, az hibát jelent, így ennek kivédéséről a header fájl megírásakor a programozónak gondoskodnia kell.

```
1 typedef double Angle;
2 class Complex {
3     public:
4         Complex(double r = 0, double i=0) {R = r; I = i;}
5         Complex operator= (Complex z) {
6             R = z.R; I = z.I; return *this;
7         }
8
9         Complex operator+ (Complex z) {
10             return Complex(R+z.R, I+z.I;
11         }
12
13         Complex operator+ (double x) {
14             return Complex(R+x, I);
15         }
16 }
```

```

16
17     Complex operator*...
18     Complex operator*...
19
20     double Re();
21     double Im();
22     double Abs();
23     double Phi();
24 private:
25     double R;
26     double I;
27 }

```

- Itt ugye frienddel is lehet.

### Modulfüggőség kezelése

- Egy program legmagasabb szintű építőkövei a modulok.
- A program működése ezen modulok interakciója.
- Minden modul igényel szolgáltatásokat és segítségükkel más szolgáltatásokat valósít meg, így a modulok között egyfajta függőségi reláció alakul ki, s egy modul megváltozása esetén szükségessé váhat a tőle függő modulok újraindítása.
- Némelyik programozási nyelv (pl. C, vagy C++) teljes egészében a programozóra bízta ezen függőségek kezelését, minden fordítási egységet önállóan kezel.
- Más nyelvekben a függőségek kezelése a program feladata (pl. ADA with utasítás)

## 8.4. Sablon

- Újrafelhasználható, könnyen karbantartható programokra van szükség.
  - Ennek a jegyében készít a programozó minél általánosabb típusokat és alprogramokat, hogy azokat mind az éppen aktuális, mind egyéb programjaiban a lehető legszélesebb körben használhassa.
  - Maga a programozási módszertan is olyan programozási megoldásokat tanít, amelyek általánosan használhatóak.
  - Ezt a törekvést a programozási nyelvek is támogatják kisebb-nagyobb mértékben. Most ezeket az eszközöket gyűjtjük össze. Az eszközök némelyike olyan megszokott és hétköznapi, hogy talán nem is vesszük észre, hogy ebbe a kategóriába tartoznak...

## 8.5. Típussal, alprogrammal való paraméterezés

- Alprogramok paraméterezése
- Típusok paraméterezése
  - Például az Ada-ban a diszkriminációs rekordok, vagy a határozatlan indexhatárú tömb típusok
- Alprogrammal történő paraméterezés

## Típusokkal történő paraméterezés

- Programozási tételeink, alapvető adatszerkezetek és adattípusaink is "absztrakt" fogalmakat használnak, azaz csupán a számunkra feltétlenül szükséges megszorításokat teszik ezekre a fogalmakra.
- Ennek következménye, hogy általában nem egyetlen típus elégíti ki ezen megszorításokat, hanem sok.
- Szeretnénk a fenti tételek, adatszerkezetek, típusok implementációját csak egyszer megadni és azt a leírást általánosan az összes, a feltételeket kielégítő típussal használni.

## 8.6. Példányosítás

Itt csak annyi volt megjegyezve, hogy sablon törzs  $\leftrightarrow$  sablon specifikáció  $\leftrightarrow$  példányosítás és itt a lényeg, hogy a példányosításnak és a sablonnak nem szabadna találkoznia.

- Ezen igény kielégítésére többféle eszköz kínálkozik.
  - A C nyelvben például nem áll rendelkezésre ilyen eszköz nyelvi szinten, ám hasonló hatások - korlátozott mértékben - elérhetők az előfeldolgozó rendszer használatával.
- Az előfeldolgozó rendszer szerves része a C nyelv specifikációjának, mégsem tekinthető igazán nyelvi eszköznek, hiszen a forrásnyelvű programot manipulálja ha úgy tetszik, közelebb áll a megíráshoz használt szövegszerkesztőhöz.

## Példák

- CLU
  - `generic_proc_name = proc [ t : type ] (...) returns (...) signals (...)`
  - `sort proc[ t : type ] ( A : array[ t ] ) returns (array[ t ] )`
  - `where t has lt : proctype (t, t) returns (bool)`
  - Objektumorientált programozási nyelvekben a hatás elvileg elérhető lenne kizárólag az öröklődés és a polimorfizmus használatával, hiszen a szükséges közös, tulajdonságokat össze lehetne fogni egyetlen absztrakt osztályba - vagy interfészbe -, amely aztán közös őse lehetne az összes, a feltételeknek eleget tevő osztálynak.
  - Az egyetlen hibája a megoldásnak, hogy előre - a nyelv alapvető hierarchiájának kialakításakor - ismerni kellene az összes számításba jövő feltételkombinációt.
  - A típussal paraméterezést támogató nyelvek általában valamilyen önálló konstrukciót vezetnek be.
  - Ezekben a struktúrákban megadhatunk formális paraméterként típusokat, amelyeket aztán más típusokhoz hasonlóan használhatunk a struktúra belsejében.
  - Amikor a formális paramétereknek aktuális értéket adva létrehozuk a struktúra megadott típusokhoz tartozó változatát, akkor a struktúra példányosításáról beszélünk.
- C++

```
1  #define MAX(a,b) a > b ? a : b
2  MAX(x,y)*2 --> x > y ? x : y*2
3  #define MAX(a,b) ((a) > (b) ? (a) : (b))
4  MAX(++x, y) --> ++x > y ? ++x : y
5
6  void swap (double& x, double& y) {
7      double temp = x;
8      x = y;
9      y = temp;
10 }
```

- A C++ nyelvben a típussal paraméterezés lehetőségét a template-ek biztosítják.
- Egy template kicsit több, mint az előző makróhelyettesítés, ahol a fordító a megadott aktuális típussal helyettesíti a hozzá tartozó formális paraméter minden előfordulását. Bizonyos minimális szintaxisellenőrzést ugyan végez a fordító, de a formális paraméternél csak annyit tudok meghatározni, hogy az adott paraméter egy típust jelöl.
- Nem tudom előírni a megvalósítandó alprogram, vagy típus számára fontos tulajdonságok (például bizonyos műveletek) meglétét.
- Az ebből származó hibák így példányosításkor jelentkeznek
- Előnye a nyelvnek, hogy a template példányosítása teljesen automatikusan történik.

```

1  template <typename T>
2  void swap(T& x, T& y) {
3      T temp = x;
4      x = y;
5      y = temp;
6  }

```

```

1  template <class T>
2  T max (T x, T y) {
3      return (x>y) ? x : y;
4  }

```

- C++ új szabvány
  - \* tervezték a concept-ek lehetőségét
    - megfogalmazhatjuk elvárásainkat egy típussal szemben, amit még példányosítás előtt tud ellenőrizni a fordítóprogram, és meg tudja nevezni a hiba okát olvasható formátumban

```

1  template<class T> const T& min(const T &x, const T &y) {
2      return y < x ? y : x;
3  }
4
5  template<LessThanComparable T>
6  const T& min(const T &x, const T&y) {
7      return y < x ? y : x;
8  }
9
10 auto concept LessThanComparable<class T>
11 {
12     bool operator<(T,T);
13 }

```

- Eiffel:generic

- Erősen objektum orientált megközeleítés
- A formális paraméterei osztályok lehetnek, ahol a szükséges speciális tulajdonságokat azáltal lehet meghatározni, hogy megadható, mely osztályból kell származnia az aktuális paraméternek.
  - \* Előnye: A generic belsejében használt műveletek a megadott őosztály műveletei lehetnek, így a használat helyessége a generic megírásakor ellenőrizhető.
  - \* Hátránya: a szükséges közös tulajdonságokat jóval előre tudni kell, hogy a megfelelő közös ő definiálható legyen. Ekkor viszont a problémák jórészt megoldhatóak az öröklődés és a polimorfizmus eszközeivel.
- Megszorítás nélküli generic
 

```
dereffered class TREE[G]...
class LINKED_LIST[G]...
class ARRAY[G]...
```

- Az osztályoknak akárhány formális generic paramétere lehet.
- Típus létrehozása egy generic osztályból: minden formális generic paraméterhez kell egy típus, az aktuális generic paraméter.
- Ez egy generikusan származtatott (példányosított) típust eredményez.
- Generic származtatások - példányosítás  
`TREE [INTEGER]`  
`TREE [PARAGRAPH]`  
`LINKED_LIST[PARAGRAPH]`  
`TREE [TREE [TREE [PARAGRAPH]]]`  
`ARRAY [LINKED_LIST [TREE [LINKED_LIST [PARAGRAPH]]]`
- Korlátozott generic
  - \* `class HASH TABLE [G, KEY → HASHABLE]...`
    - KEY-t megszorítjuk a HASABLE osztállyal.
    - Minden T aktuális generic paraméter bázisosztálya, amit a KEY-hez használunk a HASABLE megszorító osztály leszármazottja kell legyen.
  - \* `HASH TABLE [PARAGRAPH, STRING]`
  - \* A korlátozott formális generic paraméterek általános szintaxisa:  $T \rightarrow \text{Class-type}$

#### • ADA

- A sablonok paramétereizhetősége sokkal szélesebb körű és rugalmasabb a korábbiaknál.
  - \* Lehet:
    - megadott őstípusból származó típussal paraméterezni,
    - kiköthető, hogy az aktuális paraméter valamilyen típusosztályba (diszkrét típus, felsorolási típus, tetszőleges típus, stb.) tartozzon.
    - megadható a használni kívánt további műveletek, amelyek segítségével kikerülhető a kötelező ős megléte.
    - az explicit megadott műveletek segítségével a generic még rugalmasabb használatára nyílik lehetőség, megtekinthető például, hogy egy rendezési relációt használó sablont az egész számokkal példányosítva nem a szokásos rendezési relációt adjuk meg, hanem például a oszthatóság parciális rendezési relációját.

```

1  generic
2      type Item is private;
3      type Index is (<>);
4      type Vector is array (Index range <>) of Item;
5      with function "<"(X, Y : Item) return Boolean is <>;
6
7  procedure Log_Search(V: in Vector, X: in Item; Found: out Boolean; Ind: out Index
    );

```

#### ▲ Java

- Generic bevezetés

\* Korábban:

```

1  List myIntList = new LinkedList();
2  myIntList.add(new Integer(0));
3  Integer x = (Integer) myIntList.iterator().next();

```

\* Most már lehet:

```

1      List<Integer> myIntList = new LinkedList<Integer>();
2      myIntList.add(new Integer(0));
3      Integer x = myIntList.iterator().next();

```

– Ehhez a java.util-ban példa

```

1      public interface List<E> extends Collection<E>
2      {
3          void add(E,x);
4          Iterator<E> iterator(); ...
5      }
6
7      public interface Iterator<E> {
8      {
9          E.nex();
10         boolean hasNext();...
11     }

```

– java.util-ban lehet még Serializable-t használni...

– java.lang

```

1      public interface Comparable<T>...

```

– generic metódusok lehetősége is megvan

```

1      pubic <U> void isnpsect (U u)...

```

– Az eredmény:

- \* T: java.lang.Integer
- \* U: java.lang.String

– Típustörítés - nyers(raw) típusok

- \* Box rawBox = new Box(); - ez is lehetséges - ezzel a generic előtti viselkedést kapjuk...
- \* Box a nyers típusa Box<T>-nek

– A korábbi programok így gond nélkül működhetnek

– Futási időben minden generic Object-tel paraméterzve lesz, ellenőrzés fordításkor.

– Megszorított típusparaméter lehetősége:

- \* < U extends Valami >  
és akkor az aktuális típusparaméter U-ra csak a Valami leszármazottja lehet
- \* vagy, ha interfész megvalósítását is kérjük:  
<U extends Valami & MyInterFace >
- \* A generic törzsében számíthatunk a Valami és a MyInterFace műveleteire!
- \* Mindig először kell az osztály, azután az interfészek.

– Öröklődéssel való kapcsolata később, OOP-nél!

– Típushelyettesítők

- \* Tételezzük fel, hogy

```

1      public class Gen<A> {
2          public Gen(A aa) { a = aa;};
3          public A getElem(){return a;};
4          public void setElem(A aa) a=aa;};
5          protected A a;
6      }

```

\* És készítünk egy f fv-t:

```
1      static<A,B> Gen f(Gen<A> ga, Gen<B> gb) {  
2          Object o1 = ga.getElem();  
3          Object o2 = gb.getElem();  
4          if (o1.toString().length() < o2.toString().length())  
5              {  
6                  return ga;  
7              else  
8                  {  
9                      return gb;  
10                 }
```

## 9. Programok helyessége és a támogató nyelvi eszközök (Hoare módszer, levezetési szabályok is) (példák: Eiffel, Java-eszközök, Code Contracts Library)

### 9.1. Programok helyessége

- De szép ez program → ez a program helyes.
- Feladat
  - Megoldja?
    - \* Teszteljük
      - Fekete doboz
      - Fehér doboz
    - \* Nem biztos a megoldás
- Formális matematikai megközelítés szükséges!

### 9.2. Hoare módszer

#### Hoare-féle specifikáció

- A típusműveletekhez elő- és utófeltételeket rendelünk
- A típusértékalkalmazok leírása invariánsokat is tartalmazhat

#### Elő- és utófeltételes specifikációt alkalmaz számos nyelv

- közvetlenül:
  - Alghard
  - Eiffel
  - D
  - Oxygene
  - Cobra
  - ...
- közvetve:
  - C#
  - Java
  - ...

#### Maga a módszer

- A feladat:
  - adjuk össze egy valós elemeket tartalmazó vektor elemeit!
- Kiinduló adatok:
  - egy n hosszú, valós elemeket tartalmazó v vektor
- Eredmény:



- s valós érték, ami tartalmazza a vektor elmeinek összegét!
- Írjunk rá megoldó algoritmust!

```

1  float sum(float *v, int n) {
2      float s = 0.0;
3      int i = 0;
4      while (i < n) {
5          s = s + v[i];
6          i = i + 1;
7      }
8      return s;
9  }

```

- Hogyan indokoljuk, hogy ez a program helyes (megoldja a feladatot)?

### Feladat specifikációja

- Szerződés a felhasználó és az implementáló között
  - előfeltétel: egy állítás, ami leírja azt a feltételt, ami szükséges a feladat helyes működéséhez
  - utófeltétel: egy állítás, ami leírja azt a feltételt, amit a függvény teljesít a helyes végrehajtás után
- Helyesség a specifikáció figyelembevételével:
  - ha a függvény felhasználója teljesíti az előfeltételt, a függvény elkezd futni, s amikor befejezi, akkor az utófeltétel igaz lesz.
- (Mit kell az implementációnak teljesíteni, ha a felhasználó megsérti az előfeltételt?)
- Állítás: egy logikai függvény a program állapotával kapcsolatosan
- Például:
  - $x = 3$
  - $y > x$
  - $(x \neq 0) \rightarrow (y + z = w)$
  - $s = \sum_{i \in 1..n} v[i]$
  - $\forall i \in 1..n : v[i] > v[i - 1]$
  - true
- Álapottér
  - Jelölje  $\mathcal{V}$  az n hosszú, valós elemekből álló vektorok típusát! (0-tól n-1-ig indexelve)
  - $\mathcal{V} \times \mathcal{R}$
  - Változók: v, s
  - Előfeltétel:  $v = v'$  és  $n > 0$
  - Utófeltétel:  $s = (\sum i | 0 \leq i \leq n : v[i])$

### Hoare-hármas

- A programhelyesség formális indokláshoz, elő- és utófeltételek használatával
- Szintaxis:  $\{Q\} S \{R\}$ 
  - Q és R állítások
  - S program
- Jelentése: Ha kiinduláskor igaz a Q és végrehajtjuk S-t, akkor S egy olyan állapotban terminál, ahol R igaz.

### Példák Hoare-hármasokra

- $\{\text{true}\} x := 5 \{x=5\}$
- $\{x = y\} x := x+3 \{x=y+3\}$
- $\{x > 1\} x := x*2 \{x > 2\}$
- $\{x=a\} \text{if } (x<0) \text{ then } x:=-x \{x=|a|\}$
- $\{\text{false}\} x:=3 \{x=8\} \rightarrow \text{ez nem jó!}$

### Legerősebb utófeltétel

- Néhány érvényes Hoare-hármas:
  - $\{x=5\} x:=x*2 \{\text{true}\}$
  - $\{x=5\} x:=x*2 \{x>0\}$
  - $\{x=5\} x:=x*2 \{x=10 \mid x=5\}$
  - $\{x=5\} x:=x*2 \{x=10\}$
- Mind igaz, de az utolsó a leghasznosabb:  $x = 10$  a legerősebb utófeltétel
- Definíció: Ha  $\{Q\} S \{R\}$  és minden olyan  $R'$ -re, amire  $\{Q\} S \{R\}$ ,  $R \rightarrow R'$ , akkor  $R$  az  $S$   $Q$ -ra vonatkozó legerősebb utófeltétele
  - ellenőrizzük:  $x = 10 \rightarrow \text{true}$
  - ellenőrizzük:  $x = 10 \rightarrow x > 0$
  - ellenőrizzük:  $x = 10 \rightarrow x = 10 \mid x = 5$
  - ellenőrizzük:  $x = 10 \rightarrow x = 10$

### Leggyengébb előfeltétel

- Néhány érvényes Hoare-hármas:
  - $\{x = 5 \ \&\& \ y = 10\} z := x/y \{z < 1\}$
  - $\{x < y \ \&\& \ y > 0\} z := x/y \{z < 1\}$
  - $\{y \neq 0 \ \&\& \ x/y < 1\} z := x/y \{z < 1\}$
- Mind igaz, de az utolsó a leghasznosabb, mert ez az, ami a legáltalánosabb feltételekkel engedi hívni a programot:
  - $y \neq 0 \ \&\& \ x/y < 1$  a leggyengébb előfeltétel
- Definíció: Ha  $\{Q\} S \{R\}$  és  $\forall Q'$ -re, ahol  $Q' \rightarrow Q$  igaz, hogy  $\{Q'\} S \{R\}$ , akkor  $Q$  az  $S$   $R$ -re vonatkozó leggyengébb előfeltétele (weakest precondition).
  - Jelölje ezt  $\text{wp}(S,R)$ .

### A wp általános tulajdonságai

1. A csoda kizárásának törvénye:  $\text{wp}(S, \text{false}) = \text{false}$
2. Monotonitás tulajdonság: ha  $P \rightarrow R$  akkor  $\text{wp}(S, P) \rightarrow \text{wp}(S, R)$
3.  $\text{wp}(S, P \text{ and } R) = \text{wp}(S, P) \text{ and } \text{wp}(S, R)$
4.  $\text{wp}(S, P) \text{ or } \text{wp}(S, R) \rightarrow \text{wp}(S, P \text{ or } R)$

### 9.3. Levezetési szabályok

- Értékadás

- $\{Q\}x := 3\{x + y > 0\}$
- Mi a Q leggyengébb előfeltétel?
- Hogy kapjuk meg?
- Mi az a legáltalánosabb értéke az y-nak, amire  $3 + y > 0$ ?
- $y > -3$
- Értékadási szabály
  - \*  $wp(x := E, V) = V^{X \leftarrow E}$
  - \*  $\{V^{X \leftarrow E}\}x := R\{V\}$
  - \*  $(x + y > 0)x \leftarrow 3$
  - \*  $= (3) + y > 0$
  - \*  $= y > -3$

- Szekvencia

- $\{Q\}x := x + 1; y := x + y\{y > 5\}$
- Mi a Q leggyengébb előfeltétel?
- Szekvencia szabályai:
  - \*  $wp(S; T, R) = wp(S, wp(T, R))$
  - \*  $wp(x := x + 1; y := x + y, y > 5)$
  - \*  $= wp(x := x + 1, wp(y := x + y, y > 5))$
  - \*  $= wp(x := x + 1, x + y > 5)$
  - \*  $= x + 1 + y > 5$
  - \*  $= x + y > 4$

- Elágazás

- $\{Q\} \text{ if } x > 0 \text{ then } y := x \text{ else } y := -x\{y > 5\}$
- Mi a Q leggyengébb előfeltétel?
- Intuitív megközelítés
  - \* ha a feltétel igaz:  $\{Q1\}y := x\{y > 5\}$
  - \*  $Q1 = x > 5$
  - \* else ág:  $\{Q2\}y := -x\{y > 5\}$
  - \*  $Q2 = -x > 5$
  - \*  $Q2 = x < -5$
  - \*  $Q = x > 5 \vee x < -5$
- Elágazás szabálya:
  - \*  $wp(\text{if } B \text{ then } S \text{ else } T, R) = B \rightarrow wp(S, R) \ \&\& \ \neg B \rightarrow wp(T, R)$
  - \*  $wp(\text{if } x > 0 \text{ then } y := x \text{ else } y := -x, y > 5)$ 
    - $= x > 0 \rightarrow wp(y := x, y > 5) \ \&\& \ x \leq 0 \rightarrow wp(y := -x, y > 5)$
    - $= x > 0 \rightarrow x > 5 \ \&\& \ x \leq 0 \rightarrow -x > 5$
    - $= x > 0 \rightarrow x > 5 \ \&\& \ x \leq 0 \rightarrow x < 5$
    - $= x > 5 \vee x < -5$

- Ciklusok

- $\{Q\}$  while (felt) törzs $\{R\}$
- Mi a  $Q$  leggyengébb előfeltétel
- Mit kell belátni?
  1.  $Q \rightarrow Inv$ 
    - \* az invariáns kezdetben igaz
  2.  $\{Inv \ \&\& \ B\} S \{Inv\}$ 
    - \* a ciklusmag minden végrehajtása megőrzi az invariánst
  3.  $(Inv \ \&\& \ \neg B) \rightarrow R$ 
    - \* az invariánsból és a ciklus kilépési feltételéből következik az utófeltétel

### Ciklus invariáns keresése

- Általában az utófeltétel valamilyen gyengítése
  - $s = (\sum i | 0 \leq i < n : v[i])$
- Függ valamilyen módon a ciklusváltozótól
  - Tudjuk, hogy  $j$  kezdetben 0, és inkrementáljuk, amíg el nem éri  $n$ -t
    - \* Így  $0 \leq j \leq n$  valószínűleg az invariáns része
  - Ciklus kilépési feltétel  $\&\&$  invariáns  $\rightarrow$  utófeltétel
    - \* Ciklus kilépési feltétel:  $j=n$
  - Jó ötlet: helyettesítsük az utófeltételben  $n$ -t  $j$ -vel:
    - \*  $(\sum i | 0 \leq i < j : v[i])$
  - Invariáns
    - \*  $0 \leq j \leq n \ \&\& \ s = (\sum i | 0 \leq i < j : v[i])$

### Eiffel

- Az Eiffel beépített nyelvi eszközökkel rendelkezik formális specifikációk megadására és az Eiffel futtató rendszer ellenőrizni is tudja, hogy az egyes programegységek nem sértik-e meg a specifikációt.
- A típusok műveleteinek elő- utófeltételes specifikációja fontos szerepet játszik az Eiffel módszertanában:
  - felhasználják a program tervezése során a követelmények pontosabb megfogalmazására, de teszteléskor és dokumentációs célokra is.
- A nyelv tervezésekor ugyanakkor nem volt cél, hogy a fordítprogram bizonyítsa a program helyességét, mert ez a gyakorlatban nem megoldható.

### Elő- és utófeltételek

```

1 forth is
2 --Move forward one position.
3 require
4 not_after: no fater
5 do
6 ...Some appropriate implementation
7 ensure
8 position = old position + 1
9 end - forth
10 put_i_th(v:like first; i: Integer) is
11 --Put item v at i-th position.
```

```

12  require
13  index_large_enough: i >= 0;
14  index_small_enough: i <= count;
15  deferred
16  ensure
17  not empty
18  end - put_i_th

```

## Ciklushelyesség

- A C osztály r metódusa akkor és csak akkor ciklushelyes, ha minden ciklusra:
  1.  $\{\text{True}\} \text{INIT} \{\text{INV}\}$
  2.  $\{\text{True}\} \text{INIT} \{\text{VAR} \geq 0\}$
  3.  $\{\text{INV and then not EXIT}\} \text{BODY} \{\text{INV}\}$
  4.  $\{\text{INV and then not EXIT and then } (\text{VAR} = v)\} \text{BODY} \{0 \leq \text{VAR} < v\}$

## Exception helyesség

- Egy C osztály egy r rutinja exception-helyes akkor és csak akkor, ha a rescue blokkjának minden 'b' ágára a következő igaz:
  - Ha b egy retry-jal végződik:
    - \*  $\{\text{true}\} b \{\text{INV}_c \text{ and } pre_e\}$
  - Ha b nem retry-jal végződik:
    - \*  $\{\text{true}\} b \{\text{INV}_c\}$

## Öröklődés és helyesség

- Az öröklődés során az elő-utófeltételek újradeklarálhatóak, amelyre az Eiffel a require else ... alternatív előfeltételt és az ensure then ...extra utófeltételt adja
- A DbC ezt úgy fogalmazza meg, hogy egy rutint újradefiniálni azt jelenti, hogy alszerződést kötök a leszármazottal, amire a kliensek az őseivel már kötöttek. Egy becsületes alvállalkozó legalább olyan jól teljesíti a követelményeket, mint az eredeti. Ez azt jelenti:
  - megtartja, vagy gyengíti az előfeltételt, vagyis nincs új követelménye a kliens felé,
  - megtartja, vagy szűkíti az utófeltételt, vagyis az eredmény legalább olyan jó, mint az eredeti szerződésben volt.

## Java

### Java-assert

- Az utasítással különböző elő- és utófeltételeket fogalmazhatunk meg.
- Előnye a feltételes utasításokkal szemben: használata ki- és bekapcsolható.
- az utasításnak két alakja van:
  1. `assert logikai_kifejezés;`
  2. `assert logikai_kifejezés : üzenet;`
- pl: `assert  $x < y$  - ha nem teljesül AssertionError.`

- Lehet írni rá külön tesztelő Classst:

```
1 public class AssertTest { ... }
```

### Java - jContractor

- Design by Contract
- Függvények formájában adhatjuk hozzá
- Nagy előnye, hogy használatához magán a könyvtáron kívül nincs szükség további eszközökre, módosított compilerre, előfordításra.
- Pl: precondition:

```
1 protected boolean push_Precondition(Object o) {  
2     return o != null;  
3 }
```

- Szabályok az előfeltételre:
  - A contractor metódusoknak nem lehet előfeltétele
  - Natív metódusoknak nem lehet előfeltétele
  - a `main(String [] args)` metódusnak nem lehet előfeltétele
  - Egy static metódus előfeltétele static kell, hogy legyen
  - Egy nem static metódus előfeltétele nem lehet static
  - Egy nem-private metódus előfeltétele `protected` kell legyen
  - Egy private metódus előfeltétele `private` kell, hogy legyen
- Utófeltétel `_Postcondition`
  - Boolean visszatérési érték és ki lehet egészíteni `RESULT` elnevezésű argumentummal is.

```
1 protected boolean push_Postcondition (Object o, void Result) {  
2     return implementation.contains(o) && (size() == OLD.size()+1);  
3 }
```

- Szabályok az utófeltételre
  - A Contract metódusoknak nem lehet utófeltétele
  - Natív metódusoknak nem lehet utófeltétele
  - Egy static metódus utófeltétele static kell, hogy legyen

- Egy nem static metódus utófeltétele nem lehet static
- Egy nem-private metódus utófeltétele protected kell legyen
- Egy private metódus utófeltétele private kell legyen
- A konstruktorok utófeltételei nem hivatkozhatnak az OLD-ra

### jContractor - invariáns

- Az invariáns ellenőrzését egy boolean visszatérési értékű, `_Invariant` nevű, argumentum nélküli protected függvényben valósítjuk meg.
- Szabályok
  - A contract, a static és a natív metódusokra az invariánst nem ellenőrzik
  - Az invariánst konstruktor esetén csak kilépéskor ellenőrzi
  - Az `_Invariant()` metódus protected és nem-static kell legyen

### JContractor - Kvantorok

- A jContractor szerződésekben a JaQuaL könyvtár (Java Quantification Library) használhatjuk az egzisztenciális és univerzális kvantorokat használó feltételek megfogalmazásához.
- Négyféle JaQuaL kifejezést használhatunk:
  - a `ForAll.in(collection).ensure(assertion)` kifejezéssel teljesül az assertionben megadott feltétel

```

1      Assertion connected = new Assertion() {
2          boolean eval (Object o) {
3              return ((Node) o).connections >= 1;
4          }
5      };
6      return ForAll.in(nodes).ensure(connected);

```

- Az `Exsist.in(collection).suchThat(assertion)` kifejezéssel az egzisztenciális kvantort tudjuk kiváltani.
- Az `Elements.in(collection).suchThat(assertion)` függvény egy Vectort ad vissza azokból az elemekből, melyek kielégítik az assertionben megadott feltételt.
- A `Logical.implies(a,b)` függvény a logikai következtetést valósítja meg: igazat ad vissza, ha az a hamis vagy ha a és b mindegyike igaz volt.

### Java - JML

- Java Modelling Language (JML)
- A JML egy formális viselkedési interfész-specifikációs nyelv. A viselkedési specifikáció alatt azt értjük, hogy a Java osztályok interfészének szokásos szintaktikus leírásán (függvénynevek, láthatóság, visszatérési értékek stb.) túl az egyes függvények viselkedését is megpróbáljuk specifikálni a rendszerben.
- A JML nyelvben szerződéseket megfogalmazó feltételeket annotációs megjegyzések formájában helyezhetjük el a programkódban.
- Ez lehet `/*`-gal kezdődő sor vagy egy `/*@ ... @*/` blokk
- A kifejezésben a Java jelölésrendszerét használhatjuk, kibővítvé néhány speciális elemmel
- Az elkészült kódot a JML saját compilerével, a `jmlc`-vel kell lefordítanunk, de mivel a kontraktusokat megjegyzések formájában írtuk fel, a kód fordítható marad az eredeti Java fordítóval is.

- A kontraktusok ellenőrzése kikapcsolható.

```

1 public class IMath {
2     /*@ requires (*x is positive *);
3     @ ensures \result >= 0 &&
4     @ (*result is an int approximation to square root of x *)
5     @ */
6     public static int isqrt(int x) {...}

```

- Minősítők

- Univerzális és egzisztenciális ( $\forall, \exists$ )
- Általános ( $\sum, \Pi, \min \max$ )
- Numerikus

Szintaxis	jelentés
\result	Eredmény
$A \implies B$	A-ból következik B
$A \Leftarrow B$	B-ből következik A
$A \iff B$	A iff B
$A \Leftarrow! \Rightarrow B$	$\neg(A \iff B)$
\old(E)	E eredeti értéke

- JML példák

```

1 public class IntegerSetf
2 ...
3     byte[] a; /* The array a is sorted*/
4     /*@invariant
5     (\forall int i; 0 <= i && i < a.length-1;
6     a[i] < a[i+1]);
7     @*/

```

```

1     /*@ requires amount >= 0
2     ensures
3         balance == \old(balance) - amount &&
4         \result == balance;
5     /*@
6
7     public int debit (int amount) {
8         ...
9     }

```

## Contracts for Java (cofoja)

- A Google-nél fejlesztették 2011-ben
- Annotációkat lehet használni
  - Típus invariáns - Invariant kulcsszó - minden publikus és csomagszintű láthatóságú metódus be- és kilépésénél, és a konstruktorok végén ellenőrzi, öröklődésnél and kapcsolat
  - Előfeltételek - Requires kulcsszó - metódus belépésnél ellenőrzi, öröklődésnél or kapcsolat
  - Utófeltételek - Ensures kulcsszó - metódus normál kilépésnél ellenőrzi, öröklődésnél and kapcsolat
  - Kivételes utófeltételek - ThrowEnsuresChecked - ha exception lépett fel, ezt ellenőrzi
  - old és result lehetősége



## Előfeltételek

- `Contract.Requires(...)`
- általában paraméterek ellenőrzésére
- az előfeltételben szerepelő összes tagnak elérhetőnek kell lennie az adott helyen (különben az előfeltételt nem tudja értelmezni a hívó metódus)
- a megadott feltételeknek nem lehet mellékhatásuk.

```
1 Contract.Requires (x != null); // az x nem lehet null
```

```
1 Contract.Requires<ArgumentNullException>(x != null);  
2 //throw exception
```

- Örökölt követelmények

## Utófeltételek

- Ellenőrzése az adott metódus végrehajtása után
- Közöséges utófeltétel
  - `Contract.Ensures()`
  - `Contract.Ensures(this.F > 0);`
- Kivételes utófeltételek: Ha a metódus végrehajtása során kivétel váltódik ki, akkor is lehetőség van az utófeltétel ellenőrzésére. Ekkor a kivétel típusától(T) függően is lehet megadni feltételt:
  - `Contract.EnsuresOnThrow< T >(this.F > 0);`
- Speciális metódusok az utófeltételben:
  - csak utófeltételek belsejében lehet használni
  - `Contract.Result< T >()`
    - \* hivatkozik a T típusú visszatérési értékre (void típusú függvényeknél nem használható)
  - `Contract.Ensures(0 < Contract.Result<int>());`
- `Contract.OldValue< T >(e)` az e kifejezés metódus hívása előtti értéket adja vissza
- Megszorítások
  - az e kifejezés nem tartalmazhat másik régi értéket lekérdező fv-t.
  - csak olyan kifejezésre hivatkozhat, aminek létezett értéke a metódus meghívása előtt.

## Invariánsok

- Az összes invariáns void típusú függvény kell legyen, ha az adott osztályból lehet származtatni, akkor a láthatóságnak `protected`-et kell megadni
- A `[ContractInvariantMethod]` attribútummal jelöljük, hogy az adott metódus invariáns
- Az invariáns ellenőrzése az összes publikus metódus végrehajtás után megtörténik
- Ha az invariánson belül hivatkozunk az osztály egy másik publikus metódusára, akkor csak a legkülső függvényénél történik ellenőrzés

- ```

1  [ContractInvariantMethod]
2  protected void ObjectInvariant()
3  {
4      Contract.Invariant(this.y >= 0);
5      Contract.Invariant(this.x > this.y);
6      ...
7  }
```

## Egyebek

- **Contract.Assert**

- a program egy bizonyos pontján tudunk ellenőrizni:

```

1  Contract.Assert(this.privateField > 0);
2  Contract.Assert(this.x == 3, "Why isn't the value of x 3?");
```

- **Contract.Assume**

- Egy feltevést ír le, működése megegyezik az Assert-tel futási időben, de itt fordítási idejű ellenőrzés van!

```

1  Contract.Assume(this.privateField > 0);
2  Contract.Assume(this.x == 3, "Static checker assumed this");
```

- **Contract.EndContractBlock**

- ha az előfeltételek if-then-throw formában vannak leírva, akkor ez jelzi, hogy ezek előfeltételek, itt van az ellenőrző blokk vége:

```

1  if (x == null) throw
2      new ArgumentNullException("x");
3  if (y < 0) throw
4      new ArgumentOutOfRangeException(...);
5  Contract.EndContractBlock();
```

- **Contract.ForAll**

- Ellenőrző ciklus, contract-on belül használható
- Két paraméteres változat:

```

1  public int Foo<T>(IEnumerable<T> xs){
2      Contract.Requires(Contract.ForAll(xs, (T x) => x != null) );
```

- Első paraméter egy kollekció
- Második paraméter egy prédikátum
- létezik 3 paraméteres változat
  - \* Első paraméter az alsó határ
  - \* Második paraméter a felső határ
  - \* A harmadik a predikátum, aminek van egy index argumentuma

- **Contract.Exist**

- Ugyanolyan paraméterei vannak mint a ForAll-nak
- Akkor tér vissza igaz értékkel, ha a kollekció legalább egy elemére teljesül az adott predikátum és hamis értékkel tér vissza, ha egyre sem.

- Interfész contract-ok

- Itt nem írhatunk függvény törzset, ezért egy külön contract osztály kell, amit az interfésszel attribútumok kapcsolnak össze.

```

1      [ContractClass(typeof(IFooContract))]
2      interface IFoo {
3          int Count {get;}
4          void Put(int value);
5      }

```

- Absztrakt metódus szerződések

- itt se lehet függvény törzset írni, ezért itt is egy külön contract osztály kell, amit az absztrakt osztállyal attribútumok kapcsolnak össze:

```

1      [ContractClass(typeof(FooContract))]
2      abstract class Foo {
3          public abstract int Count {get;}
4          public abstract void Put(int value);
5      }

```

- Contract metódusok túlterhelése

- Mindegyik metódusnak lehet egy string típusú paramétere is.
- Ez kiíródik, ha a feltétel nem teljesül. A string értékének fordítási időben ismertnek kell lennie.

```

1      Contract.Requires(x != null, "If x is null, then the missiles are fired!");

```

- Contract öröklődés

- Egy szerződés a típus altípusára is öröklődik.
- Az altípus előfeltételének gyengébbnek kell lennie ezért, ha az őstípusnál nincs megadva semmilyen előfeltétel, akkor az az jelenti, hogy azonosan igaz előfeltétele ezért az altípusokhoz nem lehet előfeltételt írni.
- Az utófeltételnél erősebbet kell / lehet megadni.

## 10. Objektumorientált programozás

**Osztályok és objektumok, objektum létrehozása, inicializálása, példányváltozó, példánymetódus, osztályváltozó, osztálymetódus, társítási kapcsolatok, Öröklődés, polimorfizmus, dinamikus kötés, megbízható átdefiniálás leszármazottakban (példák: Java, C++, Smalltalk/Objective-C) A többszörös öröklődés problémái, lehetséges megoldásai, az interfészek fogalma, használata (példák: Java, C++, Objective Pascal, Eiffel)**

### 10.1. Osztályok és objektumok

#### Objektum

- Belső állapota van, ebben információt tárol, (adattagokkal valósítjuk meg)
- Kérésre feladatokat hajt végre - metódusok - melyek hatására állapota megváltozhat
- Üzeneteken keresztül lehet megszólítani - ezzel kommunikál más objektumokkal
- Minden objektum egyértelműen azonosítható

#### Osztály, példány

- Osztály (class)
  - Olyan objektumminta vagy típus, mely alapján példányokat (objektumokat) hozhatunk létre
- Példány (instance)
  - Egy osztály (minta) alapján létrejött konkrét példány
  - Minden objektum születésétől kezdve egy osztályhoz tartozik

#### Példa C++-ban

```
1 class Employee {
2     string first_name, family_name;
3     short department;
4 public:
5     void print() const {...}
6
7     Employee empl;
8     Employee *empl;
```

#### Példa Objective-C-ben

- Az osztály interfész része

```
1 @interface ClassName : ItsSuperclass
2 {
3     float width;
4     float height;
5     BOOL filled;
6     NSColor *fillcolor;
7     //...
8 } //Itt az adattagok helye
9 + alloc; //osztálymetódus előtt +
10 - (void)display; //példánymetódus előtt -
11 - (void)setWidth; (float)width height:(float)height;
12 - makeGroup:group, ...;
13 @end
```

- Az osztály implementációs része

```

1  #import "ClassName.h"
2  @implementation ClassName
3      + alloc
4      {
5      }
6
7      (void)display
8      {
9      }
10
11     -makeGroup:group, ...
12     {
13         va_list ap;
14         va_start(ap, group);
15         /...
16     }
17 @end

```

### Példa Java-ban

```

1  public class Alkalmazott{
2      String nev;
3      int fizetes;
4      ...
5
6      public void fizetesEmel(int novekmény){
7          fizetes += novekmény;
8      }
9  }
10
11  Alkalmazott a;
12  a = new Alkalmazott();
13  ...
14  a.fizetesEmel(60000);

```

- final, mint módosító
- Osztálynév előtt: tiltja a további származtatást
- Metódus előtt: tiltja a felüldefiniálást
- Változó előtt: tiltja a kezdeti értékadás után az érték megváltoztatását
  - Objektum esetén referencia
- nincs operátor túlterhelés (de túlterhelés van)
- Ha egy osztálynak nincs konstruktora, a fordító a következőt generálja neki:

```

1  class MyClass extends OtherClass {
2      MyClass() {super();}
3  }

```

- Destruktorok: a nyelvben nem szerepelnek
  - helyette a finalize() (a GC hívja felszámolás előtt)
- Interface: metódusok egy csoportját specifikálja, törzsük implementálása nélkül
  - konstans változókat deklarálhatunk benne
  - többszörös interface öröklés engedélyezett

## 10.2. Objektumok létrehozása, inicializálása

- Objektumok életciklusa: "megszületik", "él", "meghal"
- Az objektum inicializálása
  - Konstruktor végzi
  - Adatok kezdőértékkadása
  - Objektum működéséhez szükséges tevékenységek végrehajtása
  - Típusinvariáns beállítása

### Példa C++-ban

```
1 class Employee {  
2     string first_name, family_name;  
3     short department;  
4     public:  
5     Employee(const string& f, const string& n, short d):  
6         first_name(f), family_name(n), department(d) {}  
7 }
```

- Konstruktorok: "nincs objektum konstruktor nélkül", ha kell, implicit hívódnak
  - Neve megegyezik az osztály nevével
  - Ha már megadtunk konstruktort, akkor default konstruktor nem definiálódik
  - A default konstruktor meghívja az attribútumok konstruktorát, de a beépített típusokat nem inicializálja (konzisztensen a C-vel)
  - A konstruktornak nem lehet visszatérési értéke
- A destruktort is expliciten lehet hívni a delete operátorral, vagy implicit hívódik a blokkból való kilépéskor - fordított sorrendben.

## 10.3. Példány- és osztályváltozó, metódus

- Példányváltozó: Példányonként helyet foglaló változó
- Példánymetódus: Példányokon dolgozó metódus
- Osztályváltozó: Osztályonként helyet foglaló változó
- Osztálymetódus: Osztályokon dolgozó metódus

### Példa C++-ban

- Osztályváltozó, osztálymetódus

```
1 class Employee {  
2     string first_name, family_name;  
3     short department;  
4     static int num_emp; //-> 0 lenne az  
5  
6     public:  
7     Employee(const string& f, const string& n, short d):  
8         first_name(f), family_name(n), department(d) {  
9         num_emp++; //-> osztályon belül is elérhető  
10    }  
11  
12    static int get_num_emp() {return num_emp;}  
13    static void print_num_emp() {...} //-> kívülről elérhető public metódusokkal  
14 };  
15 int Employee::num_emp(0); //-> osztály kívül kell definiálni!
```

## A "this"

- Ha egy osztályból több objektumot példányosítunk, honnan tudjuk, hogy éppen melyik objektum hívta meg a megfelelő metódust, és a metódus melyik objektum adataival fog dolgozni?
- Szükségünk van egy olyan mutatóra, amely mindig a metódust meghívó objektumpéldányra mutat. Ezt szolgálja a "this" "paraméter". Ez a metódushívásakor egyértelműen rámutat azokra az adatokra, amelyekkel a metódusnak dolgoznia kell.
- Ez azt is jelenti, hogy ha az objektum saját magának akar üzenet küldeni, akkor a this.Üzenet(Paraméterek) formát kell, hogy használja, vagyis a metódustörzsekben az adott példányra mindig a this segítségével hivatkozhatunk. (Ez számos nyelvben alapértelmezett.)

## OOP elvárások

- Bezárás (encapsulation)
  - Adatok és metódusok összezárása
  - Egybezárás, egységbezárás - osztály (class)
- Információ elrejtése (information hiding)
  - Az objektum "belügyeit" csak az interfészen keresztül lehet megközelíteni (láthatóságok!)
- Kód újrafelhasználhatóság (code reuse)
  - Megírt kód felhasználása példány létrehozásával vagy osztály továbbfejlesztésével

### Példa C++-ban

- Az adattagok és metódusok elrejtése megoldott
- A láthatóság minősítője lehet
  - public: külső felhasználók elérik
  - protected: csak a leszármazottak érhetik el
  - private: csak az adott osztály és "barátai" számára elérhető (ez az alapértelmezés osztályoknál)

### Példa Objective-C-ben

- Láthatóság szabályozása a következő direktívákkal lehetséges
  - @public
  - @private
  - @protected (alapértelmezett)
  - @package

## 10.4. Társítási kapcsolatok

- Ismeretségi (használati)
  - Két objektum ismeretségi (használati) kapcsolatban van egymással, ha azok léte egymástól független, és legalább az egyik ismeri, illetve használja a másikat.
- Tartalmazási kapcsolat
  - Két objektum tartalmazási (egész-rész) kapcsolatban van egymással, ha az egyik objektum fizikailag tartalmazza, vagy birtokolja a másikat. A tartalmazási kapcsolat erősebb, mint az ismeretségi. Jele: üres rombusz.

## 10.5. Öröklődés

- Az alapgondolat: a gyerekek öröklik az ős metódusait és változóit
- A öröklő terminus azt jelenti, hogy az ősosztály minden metódusa és adattagja a gyerekosztálynak is metódusa és adattagja lesz
- A gyerek minden új művelete vagy adattagja egyszerűen hozzáadódik az örökölt metódusokhoz és adattagokhoz
- Minden metódus, amit átdefiniálunk a gyerekbe, a hierarchiában felülbírálja az örökölt metódust

### Példa C++-ban

```
1  class Employee {
2      string first_name, family_name;
3      short department;
4      static int num_emp; //-> 0 lenne az
5
6      public:
7      Employee(const string& f, const string& n, short d):
8          first_name(f), family_name(n), department(d) {
9          num_emp++; //-> osztályon belül is elérhető
10     }
11
12     static int get_num_emp() {return num_emp;}
13     static void print_num_emp() {...} //-> kívülről elérhető public metódusokkal
14 };
15
16 class Manager: public Employee {
17     Employee* group;
18     short level;
19     public:
20     Manager (const string& f, const string& n, short d, short lvl):
21         Employee(f,n,d), level(lvl){};
22
23     void print() const {
24         ...
25     }
```

### Példa Objective-C-ben

- Egyszeres öröklődés
- NSObject a legfelső szinten
- Polimorfizmus, dinamikus kötés - automatikusan
- Interface (@protocol)

```
1  @protocol Archiving
2  - read: (FILE *) f;
3  - write: (FILE *) f;
4  @end
5
6  @protocol ReferenceCounting
7  - incrementCount;
8  - decrementCount;
9  - (unsigned) refCount;
10 @end
11 //.....
12 @interface Shape : NSObject
13 <Archiving, ReferenceCounting>
14 ...
```



- Kategóriák

- A kategóriák segítségével már meglévő osztályokhoz adhatunk hozzá metódusokat
  - \* akkor is, ha nincs meg az osztály forrása
- Ez egy rendkívül erős eszköz, amellyel öröklés nélkül terjeszthetjük ki az osztályok funkcionalitását
  - \* akár beépített osztályokét is

```

1  #import "ClassName.h"
2  @interface ClassName (CategoryName)
3  //method declarations
4  @end
5  //.....
6  #import "ClassName+CategoryName.h" //ilyen neve legyen!
7  @implementation ClassName (CategoryName)
8  // method definitions
9  @end

```

- Példa - NSString minden objektumának legyen isURL metódusa

```

1  @interface NSString (Utilities)
2  - (BOOL) isURL;
3  @end

```

- Egy implementációja lehet:

```

1  #import "NSString+Utilities.h"
2  @implementation NSString (Utilities)
3  - (BOOL) isURL
4  {
5      if ([self hasPrefix:@"http://"])
6          return YES;
7      else
8          return NO;
9  }
10 @end

```

## 10.6. Polimorfizmus

- Polimorfizmus (többalakúság): az a jelenség, hogy egy változó nem csak egyfajta típusú objektumra hivatkozhat.
  - Statikus típus: a deklaráció során kapja meg
  - Dinamikus típus: run-time éppen milyen típusú objektumra hivatkozik = statikus típus, vagy annak leszármazottja.
  - Aki Manager az egy (is-a) Employee is.
  - A Háromszög az egy Alakzat.

### Példa C++-ban

- Tekintsük az alábbi kódot

```

1  Employee emp ("Kati", "Fekete", 3);
2  Manager m("József", "Kovacs", 3,2);
3  emp = m;

```

- Megengedett

```

1  emp.print();
2  m.print();

```

## Altípusos polimorfizmus

```
1 Shape *s;
2 ...
3
4 s = new Triangle (...);
5 ...
6 s= new Rectangle (...);
```

- Ha B (pl. Trangle, Rectangle) altípusa az A (pl. Shape) atípusnak, akkor B objektumainak referenciái értékül adhatók az A típus referenciáinak.

```
1 Shape *s;
2 ...
3
4 s = new Triangle (...);
5 ...
6 s= new Rectangle (...);
7
8 Shape *a;
9 a = h; a->draw();
10 a = t; a->draw();s
```

## 10.7. Dinamikus kötés

- Run-time fogalom. Az a jelenség, hogy a változó éppen aktuális dinamikus típusának megfelelő metódus implementáció hajtódik végre.
- A háromszög kirajzolása...
- A manager kinyomtatása...

```
1 class Employee {
2     ...
3     public:
4     ...
5
6     virtual void print() const {
7         cout << ... << endl;
8     }
9 }
```

## Altípus

- Szabályok:
  - Reflexív
  - Tranzitív

## 10.8. Megbízható átdefiniálás leszármazottakban

- Az altípus metódusai megőrzik az őstípus viselkedését:
  - Szignatúra:
    - \* Az argumentumok kontravarianciája, az eredmény kovarianciája
    - \* Az  $m_s$  kivételei benne vannak az  $m_T$  kivételei között

- Metódusok szabálya
  - \* Előfeltétel: "kontravariancia - altípus gyengébb"
  - \* Utófeltétel: "kovariancia - altípus erősebb"
- Az altípusok megőrzik a szupertípusok tulajdonságait - típusinvariánsukra:
  - "kovariancia - altípus erősebb"

### Példa C++-ban

- lehet bevezetni kovariáns metódusokat az altípusokban, de ezek túlterhelik az eredeti metódust, nem átdefiniálják!
- Példa:

```

1      class sielo {
2      public:
3          virtual void szobatars(sielo *s){
4              cout << ... << endl;
5          };
6      };
7
8      class sielo_lany : public sielo {
9      public:
10         virtual void szobatars(sielo_lany g){
11             cout << .. << endl //túlterheli!
12         };
13
14         virtual void szobatars(sielo *g){
15             cout << ... << endl; //átdefiniálja!

```

## 10.9. Többszörös öröklődés

- Egy osztálynak egynél több közvetlen őse lehet
- Problémák
  - Adattagok hányszor?
  - Melyik metódus

### Megoldási lehetőségek

- A legtöbb esetben az ilyen kódot nem lehet lefordítani, a fordító, vagy a futtató környezet kétértelműsége (ambiguous) hivatkozva hibajelzéssel leáll.
- Az ősosztály mondja meg, hogy mit szeretne tenni ilyen esetben.
- A származtatott osztály mondja meg, hogy melyiket szeretné használni.

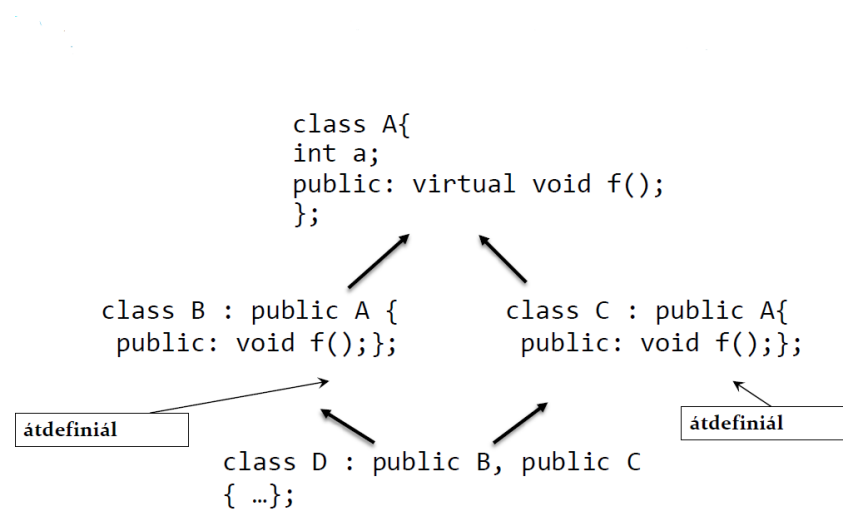
### A C++ megoldása

```

1      class D : public B, public C
2      {
3      public:
4          using C::f;
5      };

```

Tekintsük egy példát:



13. ábra. A többszörös örklődés problémái

```

1 class Animal {
2   public:
3     virtual void eat();
4 };
5
6 class Mammal: public Animal {
7   public:
8     virtual void flap();
9 };
10
11 class WingedAnimal : public Animal {
12   public:
13     virtual void flap();
14 }
15
16 class Bat : public Mammal, public WingedAnimal {
17   ...};

```

No most mi lesz?

```

1 class Mammal: public virtual Animal {
2   public:
3     virtual void flap();
4 };
5
6 class WingedAnimal : public virtual Animal {
7   public:
8     virtual void flap();
9 }
10
11 class Bat : public Mammal, public WingedAnimal {
12   ...};

```

### Absztrakt osztály

- Tervezés eszköze
- Egy felső szinten összefogja a közös tulajdonságokat
- A metódusok között van olyan, aminek csak specifikációja van, törzse nincs

- Nem hozható létre példánya
- A leszármazott teszi konkréttá

### Példa C++-ban

```

1  class Vehicle {
2      int capacity;
3  public:
4      void get_capacity() {
5          return capacity;
6      }
7  }
8
9  class Train : public Vehicle {
10 public:
11     void set_capacity {
12         this.capacity = capacity;
13     }

```

### További megoldás az öröklődési fajták

- public, protected, private öröklődés

### Példa Java-ban

- Abstract, mint módosító
- Metódus előtt: a metódus deklarálható, de nem definiálható az adott osztályban, csak a származtatottban
- Osztály előtt: az osztálynak van abstract metódusa (nem kötelező kiírni, de az osztály akkor is abstract lesz implicite)
- Egy osztály nem lehet egyszerre final és abstract

### Eiffel

- többszörös öröklődés
- ismételt öröklődés
- a metódusok default virtuálisak
- van absztrakt osztály és metódus
- Lehet bevezetni kovariáns metódusokat az altípusban
- Példa:

```

1  class Sielo
2      feature szobatars (Sielo s)
3  end
4  class Sielo_Lany inherit Sielo
5      redefine szobatars
6      feature szobatars (Sielo_Lany g)
7  end

```

- Részleges megoldás

```

1  class Sielo
2      feature szobatars (like Current)...
3  end

```

### Most nézzük az Eiffel többszörös öröklődés formáit

- A leszármazott mondja meg, hogy hogyan szeretné az örökölt attribútumokat és a metódusokat felhasználni (animália megoldása)
- Ez lehet:
  - átnevezés (anomália)(rename kulcsszó)
  - export státusz megváltoztatása a származtatottban
  - metódus felüldefiniálása (pl. csak az előfeltételt, vagy az implementációt)
  - műveletek absztrakttá tétele (deferred)(késleltetett)
  - ismételt öröklődés (ismételt öröklődésnél csak egyet örököl, ill. megmondhatja, hogy legyen mindkettő, ha akarja)

```

class A ...
  feature
    f ...
  end;

class B inherit A
  redefine
    f ...
  end;
  feature
    f ...
  ...
end;

class C inherit A
  redefine
    f ...
  end;
  feature
    f ...
  ...
end;

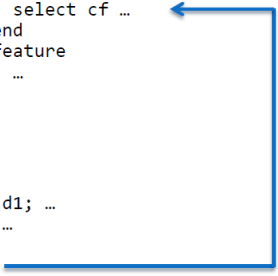
```

14. ábra. Eiffel öröklés kiválasztás

```

class D inherit
  B
    rename f as bf ...
  end;
  C
    rename f as cf
    select cf ...
  end
  feature
    ...
  end;
  ...
  a1:A;
  d1:D;
  ...
  create d1; ...
  d1.bf; ...
  a1:=d1;
  a1.f;

```



15. ábra. Eiffel (select)

```

class A
  feature
    f is ...
  end -- class A

class B
  inherit A
    redefine f
    select f
  end;
  inherit A
    rename f as old_f
  end
  feature
    f is ...
  do
    old_f ...
  end - f
end -- class B

```



16. ábra. Eiffel ismételt öröklés

#### Mit örököl a leszármazott?

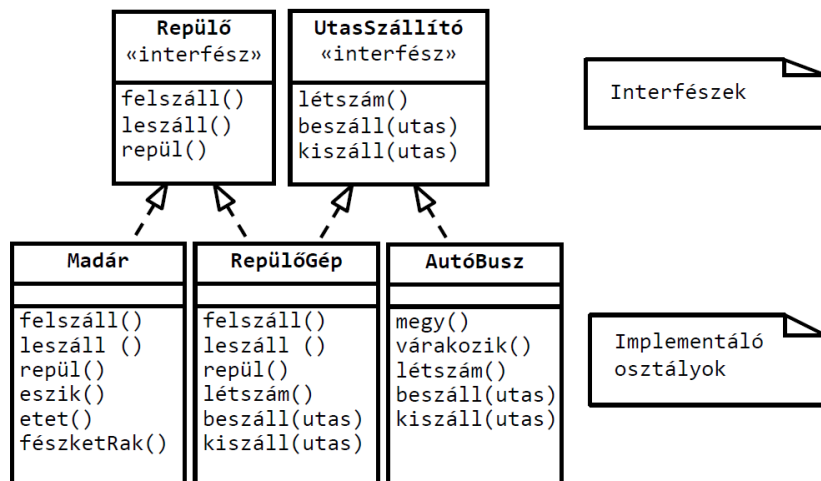
- Adattagokat
- Metódusokat
- Mit nem?
  - űosztály konstruktorait, destruktorát
  - űosztály értékadás operátorát
  - űosztály barátait

#### Mit vezethet be a leszármazott osztály?

- Új adattagokat
- Új metódusokat
- Felüldefiniálhat már meglévőket
- Új konstruktorokat és destruktorokat
- Új barátokat

#### Interfészek

- Típuspecifikáció támogatása
- Szolgáltatások
- Meg kell valósítani a szolgáltatásait
- Többször öröklődéssel nincs (annyi) probléma
  - összefogja a közös jellemzőket



17. ábra. Intefész használat

- Intefész  $\neq$  Absztrakt osztály!

#### Példa Java-ban

- Többszörös öröklődés helyett van interface
  - extends - implements
- Szabványos osztálykönyvtárak
- Beépített típusoknak (boolean, char, byte, short, int, long, float, double) csomagoló osztálya ("wrappere")
- van GC.
- Anomáliák: egyszerűbb a helyzet az interfészek miatt
  - "melyik f() implementáció fusson le?"
- Intefészek

```

1  interface Savanyu { }
2  public interface Gyumolcs{
3      int IZ = 1;
4      void egyedMeg();
5  }
6  interface Alma extends Gyumolcs{
7      int SZIN = 2;
8      int milyenSzinu();
9  }
  
```

- Interfacek öröklődése:

```

1  interface Narancs extends Gyumolcs, Savanyu{
2      int MERET = 1;
3  }
  
```

- Implementálása

```

1  class Golden implements Alma{
2      public void egyedMeg() {...}
3      public int milyenSzinu() {...}
4  }
  
```



- Java 8 óta default interface bevezetése

```

1  public interface Addressable
2  {
3      String getStreet();
4      String getCity();
5
6      default String getFullAddress()
7      {
8          return getStreet()+" "+getCity();
9      }
10 }
11
12 public class Letter implements Addressable {
13     private String street;
14     private String city;
15     public Letter(String street, String city)
16     {
17         this.street = street;
18         this.city = city;
19     }
20
21     @Override
22     public String getCity()
23     {
24         return city;
25     }
26
27     @Override
28     public String getStreet()
29     {
30         return street;
31     }
32     public static void main(String[] args)
33     {
34         Letter l = new Letter("123 AnyStreet", "AnyCity");
35         System.out.println(l.getFullAddress());
36     }
37 }

```

## Most tekintsük példaként az Objective Pascalt

- Az interface-ben:
  - csak a metódusok fejrésze van leírva
  - tartalmazhat property-ket is, de ezek szintén nincsenek kidolgozva, csak a property neve van megadva, típusa és az, hogy írható, vagy olvasható
  - mezőket, konstruktort, destruktort, nem tartalmazhat
  - minden rész az interface-ben automatikusan publikus
  - a metódusokat nem lehet megjelölni virtual, dynamic, abstract, override kulcsszavakkal
  - az interface-eknek lehetnek ősei melyek szintén interface-ek.

```

1  Type TKor = class (IMozgathato)
2      public
3          procedure Mozgat(x,y: integer);
4      end;
5  TLabda = class(IMozgathato)
6      public
7          procedure Mozat(x,y: integer);
8      end;
9  ...

```

## 11. Párhuzamosság

**Alapfogalmak, Flynn-féle modell, kommunikációs modellek, szemafor, monitor, fontos nyelvi elemek (példák: Java, Ada, Go, C++11)**

### 11.1. Alapfogalmak:

#### 11.1.1. Bevezetés:

- A hagyományos számítógépeken:
  - Egy adott időben egy program futhat, a számítógép architektúrája pedig egy egyprocesszoros gép.
    - \* Ez azt jelenti, hogy csak egy program lehet aktív, egy adott pillanatban egy utasítás kerül végrehajtásra.
    - \* A különböző programok végrehajtása egymás után történhet csak.
- Felmerült az igény a párhuzamosításra
  - Hardver szintjén
  - Szoftver szintjén
- Párhuzamosság:
  - Van egy rendszer és egy időben több komponense működik.
  - Előnyei a szekvenciális programozással szemben:
    - \* gyorsabb, hatékonyabb programvégrehajtás megfelelő hardver esetén (hardver szempont)
  - Természetesebb kifejezőmód lehetősége a nagyobb számú elemi művelet segítségével (szoftver szempont)
- Osztott (distributed) jelző - többféle jelentés –
  - Elosztott - a „párhuzamos” speciális esete, amikor a rendszer komponensei térben elkülönülten helyezkednek el
  - Megosztott memória használata
- A konkurrens (concurrent) szó gyakran azt hangsúlyozza, hogy a rendszer komponensei vetélkednek egymással az erőforrásokért.

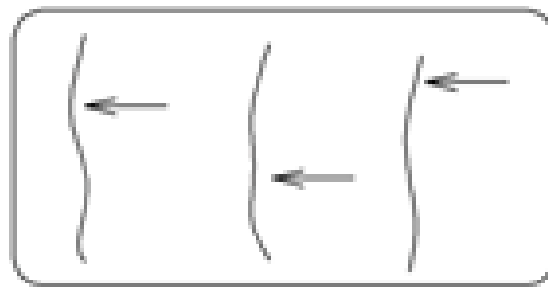
#### Folyamatok(processzek):

A processz egy olyan műveletsor, amelyet egy szekvenciálisan végrehajtott program valósít meg.

- egy folyamat a programból, az ahhoz kapcsolódó bemenő-, kimenő adatokból és egy állapotból áll.
- egy folyamatnak öt állapota lehet:
  - Futó: a CPU éppen ezt a folyamatot futtatja
  - Kész (ready): futtatható, éppen áll, hogy egy másik folyamat futhasson
  - Blokkolt: vár egy külső esemény bekövetkezésére (pl. I/O)
  - Megszakított: a folyamatoknak jeleket (signal) küldhetünk, melyekre megállnak
  - Halott: befejeződött a folyamat



18. ábra. vezérlő szál



19. ábra. Egy folyamat több szállal

### 11.1.2. Szálak és folyamatok:

Egy hagyományos folyamatban egy vezérlő szál és egy utasításmutató található:

A mai operációs rendszerek képesek már több vezérlő szál kezelésére egy folyamatban is.

Egy szál koncepcionálisan hasonlít egy folyamathoz.

A különbség:

- a folyamat (processz) kernel szintű fogalom, ezért minden egyede speciális adatokkal rendelkezik, mint pl. virtuális memória, fájlleíró, UID, GID, PID (User IDentification number, Group ID, Process ID), stb.
- Más folyamatok csak rendszerhívásokon keresztül férnek hozzá a folyamat adataihoz, állapotához.
- A folyamatleíró struktúra minden része a kernelben van, így a felhasználói program nem tudja elérni azt.
- A program eljárásai, függvényei viszont a felhasználói területen vannak, így közvetlenül elérhetőek.

A szál felhasználói szintű fogalom:

- A szálleíróstruktúra is a felhasználói területen van, és ezért közvetlenül elérhető.
- A szálak, mint programokat futtató taszkok, a folyamatokhoz hasonlóan rendelkeznek regiszter adatokkal (utasításszámláló - program counter, veremmutató - stack pointer, stb.) valamint állapotleíró adatokkal. Vagyis minden szálnak például saját veremterülete van.
- A szálaknak ugyanúgy lehet futó, kész és blokkolt állapota.
- Egy folyamat minden szála osztozik az erőforrásokon, azaz ugyanazon memória területet látja, ugyanazokat a műveleteket és adatokat használja mindegyik szál. Ha egy szál megváltoztat egy folyamat szintű adatot, akkor a folyamat összes szála érzékeli fogja ezt az adat legközelebbi hozzáférésekor.

– (Ha egy szál megnyit egy fájlt olvasásra, akkor a többi szál is olvashatja ugyanazt a fájlt.)

### 11.1.3. Processzek közötti kapcsolatok: kommunikáció

Ez a processzek közötti adatcserét jelenti. A kommunikáció során egy folyamat hozzáfér egy másik, vele párhuzamosan futó folyamat által szolgáltatott információhoz.

Történhet:

- osztott változók használatával - a program változóinak egy része (esetleg minden változó) hozzáférhető egyszerre több folyamat számára.
  - Nagyon fontos a folyamatok összehangolása
- explicit üzenetküldéssel - az egyik folyamat üzenetet küld a másik folyamatnak.
  - aszinkron kommunikáció: ha egy folyamat üzenetet küld, akkor nem várja meg, hogy a másik folyamat fogadja az adott üzenetet, a küldő folyamat végrehajtása csak az üzenet elküldésének idejére függesztődik fel.
  - szinkron kommunikáció (randevú): ha egy folyamat kommunikálni akar egy másik folyamattal, akkor végrehajtása felfüggesztődik addig, amíg a megszólított folyamat alkalmas nem lesz az üzenetvételre. A két folyamat csak a kommunikáció befejeztével futhat tovább.

Holtpont (deadlock) problémája:

A holtpont –leegyszerűsítve - olyan állapot, melynél a folyamatok körkörösén egymásra várakoznak, és egyik sem tud tovább haladni.

## 11.2. Flynn féle modell:

A párhuzamos hardver rendszerek sokféleképpen osztályozhatók

Egy ilyen a máig is használt Flynn-féle osztályozás:

- komponensei:
  - a processzor,
  - a memória és
  - a vezérlő (controller).

A processzor(ok) a vezérlőtől kapják a végrehajtandó utasítássorozatot (instruction stream), a memóriá(k)ból pedig a feldolgozandó adatokat (data stream).

### Csoportok:

Aszerint, hogy hány utasításfolyam (Instruction stream), illetve adatfolyam (Data stream) különíthető el a rendszerben. Mindkettő egyszeres (Single), vagy többszörös (Multiple) lehet.

### SISD(Single Instruction Stream, Single Data Stream)

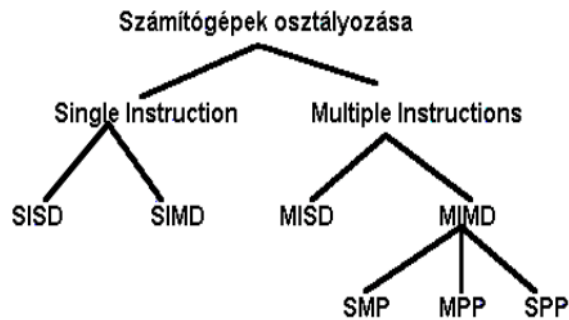
Ezek a „hagyományos” számítógépek:

### SIMD(Single Instruction Stream, Multiple Data Stream)

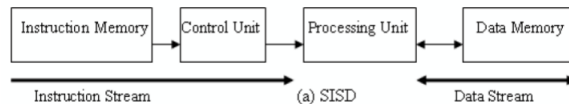
Nagyon sok egyszerű processzorból álló gépek, amelyek mindegyike rendelkezik saját memóriával, de mindegyiken azonos program fut.

### MISD(Multiple Instruction Stream, Single Data Stream)

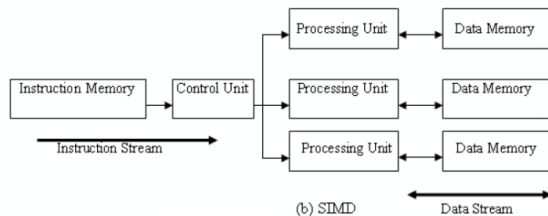
Nem gyakori az ilyen gép, csak az osztályozás teljessége miatt került bele Pipeline-szerűen lehet elképzelni. . . Hibátűrő rendszerekhez pl. 2 a 3-ból logikához



20. ábra. Számítógépek osztályozása



21. ábra. SSID



22. ábra. SIMD

### **MIMD(Multiple Instruction Stream, Multiple Data Stream)**

Ez a legfontosabb csoport.

Ezen belül: SMP(Symmetric multiprocessing) vagy SM-MIMD (shared-memory MIMD) rendszerek jellemzője, hogy a rendszerbuszra egyetlen fizikai memória csatlakozik, amelyet minden processzor lát. (Pl. többprocesszoros PC).

#### **SM-MIMD architektúra:**

Előnye:

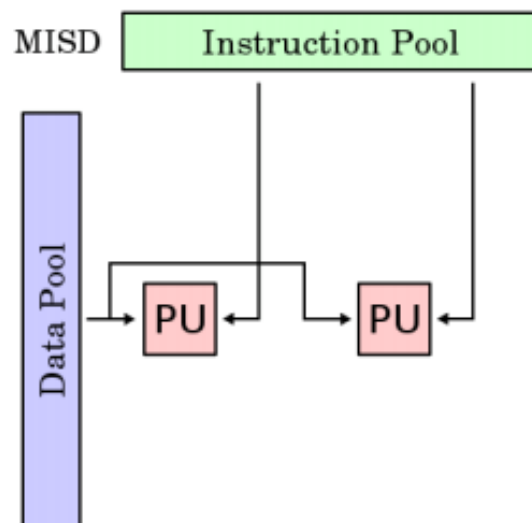
- az osztott memórián keresztül nagyon egyszerű a programok közötti kommunikáció, illetve szinkronizáció megoldása,
- hagyományos programok nagyon könnyen átvihetők ilyen környezetbe.

Hátránya:

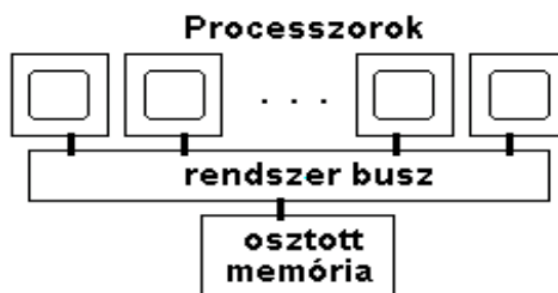
- a memória elérésének fizikai korlátai korlátozzák a lehetséges processzorok számának.

#### **MPP(Massively Parallel Processing) vagy DM-MIMD(Distributed Memory MIMD) architektúrák**

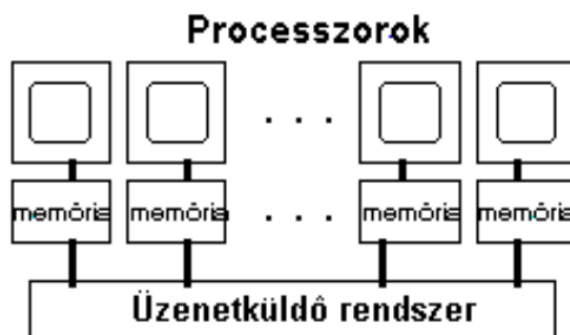
Önálló memóriával rendelkező processzorokból felépített nagy számítógépek – üzenetküldéssel kommunikálnak



23. ábra. MISD



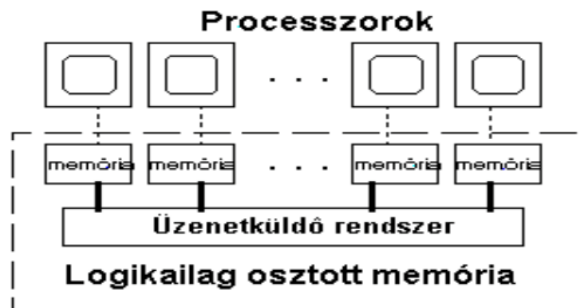
24. ábra. SM-MIMD



25. ábra. MMP

### SPP(Scalable Parallel Processing)

Olyan DM-MIMD rendszer amelyben a processzorok közötti kommunikációs eszköz szimulálni tudja az egyetlen osztott memóriát. A processzorok számára nem saját memóriájuk elérése csak időbeni különbséget jelent. Ötvözi az SM-MIMD és DM-MIMD rendszerek előnyeit - sokkal bonyolultabb hardware megoldásokra van szükség.



26. ábra. SPP

### 11.3. Kommunikációs modellek:

#### Szinkron kommunikáció - randevú

Ha egy folyamat kommunikálni akar egy másikkal, akkor végrehajtása felfüggesztődik addig, amíg a szólított folyamat hajlandó nem lesz a randevúra. A randevút kérő folyamat csak a randevú kódjának lefutása után futhat tovább. A randevú előnye az aszinkron kommunikációval szemben: az információ áramlása kétirányú lehet. Ezt a kommunikációs modellt használja pl. az Ada nyelv.

- Szinkron:
  - A randevú akkor jön létre, amikor mindketten akarják
  - Bevárják egymást a folyamatok az információcseréhez
  - Az aszinkron kommunikációt le kell programozni, ha szükség van rá
  - Pont-pont kapcsolat
  - Egy randevúban mindig két taszk vesz részt
- Az accept törzse:
  - A randevú az accept utasítás végrehajtásából áll
  - Ez alatt a két taszk együtt van
  - A fogadó taszk hajtja végre az accept-et
  - Az accept-nek törzse is lehet, egy utasítássorozat
- Kommunikáció:
  - A randevúban információt cserélhetnek a taszkok
  - Paramétereket használunk erre a célra
  - Az entry specifikációja formális paramétereket is tartalmazhat
  - A kommunikáció kétirányú lehet: Adában a paraméterek módja szerint (in, out, in out)
  - Az alprogramok hívására hasonlít a mechanizmus

#### Aszimmetrikus

- Megkülönböztetjük a hívót és a hívottat
  - diák és HF
- Egész másként működik a két fél

- Szintaxisban is kimutatható a különbség
- A hívó ismeri a hívottat, de fordítva nem
- A "hívó" és a "hívott" csak szerepek, amik egy randevúra vonatkoznak, nem egy egész taszkra
  - ugyanaz a taszk az egyik randevúban lehet hívó és a másikban hívott

#### **Aszinkron kommunikáció üzenetekkel:**

- Ha egy folyamat üzenetet küld, akkor az üzenet a fogadó folyamatnál egy üzenetsorba kerül.
- A küldő folyamat végrehajtása csak az üzenet elküldésének idejére függesztődik fel.
- Az üzenet feldolgozásához a fogadó folyamatnak ki kell venni az üzenetsorából az üzenetet.
- Ilyen kommunikációs modellt valósít meg például a PVM-rendszer.

Közös változók: a folyamatok ugyanazt a memóriarészt látják.

Ebből az a probléma adódhat, hogy két folyamat egy időben írhatja az adott erőforrást (pl. változót), ilyenkor az eredmény határozatlan érték lesz. Hasonló probléma merülhet fel egy írási és egy olvasási művelet összekapcsolásánál.

### **11.4. Szemafor**

- A szemafor, mint típust, Dijkstra vezette be egy 1968-as művében; a szemafor egy egész értékű számláló, és a hozzá tartozó várakozási sor.
- Egy inicializált szemafornak két megengedett művelete van:
  - P = passeren (áthaladni)
  - V = vrijmaken (szabaddá tenni)
- A p - wait, a v - signal.
- A műveletek szemantikája az alábbi módon van definiálva: S: semaphore;
  - wait(S): Ha  $S = 0$ , akkor  $S := S - 1$ , különben a folyamat blokkolódik, és a szemaforhoz tartozó várakozási sorba kerül mindaddig, amíg valaki fel nem ébreszti (azaz rá vagyunk utalva a többi folyamatra).
  - signal(S): Ha van várakozó folyamat az S-hez tartozó várakozási sorban, akkor felébreszti, különben  $S := S + 1$

#### **Probléma a szemaforral:**

- A szemafor nagyon alacsony absztrakciós szintű eszköz
- Könnyű elrontani a használatát item Akár a P, akár a V művelet meghívását felejtjük el, a program megbomol
- Nem lokalizált a kód, sok helyen kell odafigyeléssel használni Kölcsönös kizárás
- A szemafor segítségével kritikus szakaszokat tudunk leprogramozni
- Csak egy folyamat tartózkodhat a kritikus szakaszában
- A kritikus szakaszra (a kritikus erőforráshoz való hozzáférésre) kölcsönös kizárást (mutual exclusion) biztosítottunk



### 11.5. Monitor:

A monitor a folyamatok szinkronizálására használt eszköz és az osztott adatokról ad információt.

- A monitor biztosítja, hogy csak egy folyamat hajthat végre egy
- monitor-eljárást egy adott időben. Mielőtt egy másik folyamat
- belép a monitorba, az előzőnek véget kell érnie.
- Sok nyelvben megtalálható a monitor, mint könyvtári osztály.

### 11.6. Fontos nyelvi elemek:

Milyen nyelvi elemekre van szüksége egy programozási nyelvnek a párhuzamosság támogatására?

- Elsősorban a párhuzamosan elindított kód részeit leíró nyelvi elemekre (task, szál, folyamat, process) és ezeknek a végrehajtását szolgáló leírásra, módszerre.
- Egy folyamat most éppen végrehajtható-e?

Kommunikációhoz: Osztott adatok esetén kölcsönös kizárást garantáló eszközök. Két fő irányzat:

1. nyelvi mechanizmus az adatok védelmére (pl. szemaforok, monitorok).
  2. a folyamatok üzenetek révén kommunikálnak az üzenetek eljárshívásnak felelnek meg, a megosztott adatok pedig az átadott paraméterek
- Konkurens részek szinkronizálására megfelelő eszközök
  - A prioritások meghatározása
  - Késleltetések, időzítések kezelése

Lehetséges nyelvi elemek még:

- A fork egy párhuzamos folyamat elindítását váltja ki.
- A join újra egyesíti a párhuzamos folyamatokat.
- A cobegin-coend a párhuzamos folyamat kezdeti, illetve végpontját jelöli. Ugyanilyen nyelvi elem a parbegin-parend is.
- A select feltételhez kötött párhuzamos indítást valósít meg.
- A wait, delay késleltet egy folyamatot.
- A quit befejez egy processzt.

### ADA

- Egy Ada programban egy folyamatot egy task-objektum reprezentál.
- A taskok rendelkezhetnek belépési (entry) pontokkal. Ezek hívhatóak egy másik taszkból - így kommunikálhatnak a taszkok.
- A taszknak van egy törzse, ez írja le azt a tevékenységet, amelyet a folyamat végez.
- A törzsben minden entry-hez tartozik egy accept utasítás, amikor itt tart a törzs végrehajtása, akkor hajlandó elfogadni egy másik taszk hívását erre a randevúra.

- A kommunikáció szinkron, a hívó folyamat a randevú elfogadásáig és az accept utasítás végéig felfüggesztődik.
- Az entrynek lehetnek paraméterei (in, out vagy in out típusúak is).
- Ha egy task meghívja egy másik task entry-pontját, akkor futása felfüggesztődik a randevú létrejöttéig és lekezeléséig.

Randevú az Adában:

- Taszkok szinkronizációjához és kommunikációjához használható
- Belépési pont (entry) definiálható az egyik taszkban, amihez az accept utasítással kódot rendelünk
- A másik taszkban meghívhatjuk a belépési pontot
- A belépési pontok is "callable unit"-ok, mint az alprogramok

```

1  task HF is
2    entry Bead;
3  end HF;
4    task body HF is
5  begin
6    accept Bead;
7  end HF;
8
9  task Diák;
10 task body Diák is
11 begin
12   HF.Bead;
13 end Diák;
14
15 task HF is
16 entry Bead;
17 end HF;
18 task body HF is
19 begin
20   accept Bead do ... end Bead;
21 end HF;
22
23 task Diák;
24 task body Diák is
25   begin HF.Bead;
26 end Diák;
```

Aszinkron kommunikáció Adában:

```

1 task A;
2 task body A is
3   Ch: Character;
4 Begin
5   ...
6   Get (Ch);
7   Tároló.Betesz (Ch);
8   ...
9 end;
10
11 task B;
12 task body B is
13   Ch: Character;
14 Begin
15   ...
16   Tároló.Kivesz (Ch);
17   Put (Ch);
18   ...
19 end;
```

Az A és a B taszk a Tárolón keresztül aszinkron módon kommunikálnak.  
Szemafor Adában , lehetséges megvalósítás:

```
1 task Szemafor is
2   entry P;
3   entry V;
4 end Szemafor;
5
6 task body Szemafor is
7 begin
8   loop
9     accept P;
10    accept V;
11  end loop;
12 end Szemafor;
13
14 -- kritikus szakasz előtti utasítások
15 Szemafor.P; -- megvárjuk, amíg beenged
16 -- kritikus szakasz utasításai
17 Szemafor.V;
18 -- kritikus szakaszok közötti utasítások
19 Szemafor.P;
20 -- kritikus szakasz utasításai
21 Szemafor.V;
22 -- kritikus szakasz utáni utasítások
```

Általánosított szemafor:

```
1 task type Szemafor ( Max: Positive := 1 ) is
2   entry P;
3   entry V;
4 end Szemafor;
```

Legfeljebb Max számú folyamat tartózkodhat egy kritikus szakaszában.  
Általánosított szemafor megvalósítása:

```
1 task body Szemafor is
2   N: Natural := Max;
3 begin
4   loop
5     select
6       when N > 0 => accept P; N := N-1;
7       or
8         accept V; N := N+1;
9       or
10      terminate;
11    end select;
12  end loop;
13 end Szemafor;
```

Monitor Adában (szerkezete):

```
1 monitor_név
2 begin
3   lokális adatok;
4   eljárások;
5   lokális adatok értékadásai;
6 end név.
```

Csak egy taszk írhat ki egyszerre:

```
1 task Kizáró is
2   entry Kiír( Str: String );
3 end Kizáró;
4
5 task body Kizáró is
```

```

6 begin
7   loop
8     accept Kiír(Str: String) do -- védi
9       Text_IO.Put_Line(Str);
10      end Kiír;
11    end loop;
12  end Kizáró;
13
14  protected Védett is
15    procedure Kiír ( ... );
16  end Védett;
17
18  protected body Védett is
19    procedure Kiír ( ... ) is ... end;
20  end Védett;
21
22  Védett.Kiír(...);

```

## Java

- A Java nyelvben a folyamatok (szálak) leírására a Thread osztályt használják, az osztály egy objektuma reprezentál egy folyamatot.
  - A folyamat törzse a run metódusban leírt kód.
- Egy programban tetszőleges számú szál indíthatunk. Ezeknek megadhatjuk a prioritását, felfüggeszthetjük, újraindíthatjuk, megállíthatjuk.
- A szálakat szinkronizálhatjuk a join segítségével. A kommunikáció osztott változókon keresztül történik, a kölcsönös kizárás biztosítására a synchronized kulcsszó szolgál. Ha egy blokk vagy metódus synchronized akkor a hozzá csatolt monitor biztosítja a kölcsönös kizárást.
- Runnable interfész megvalósítása kell, ha nem tud örökölni a Threadtől!

```

1 class MyThread extends Thread {
2   public void run() {
3     System.out.println("Helló, ez itt a" + getName() + "Thread" );
4   }
5 }
6
7 public class ExtendedThread {
8   static public void main(String args[]) {
9     MyThread a, b;
10    a = new MyThread();
11    b = new MyThread();
12    a.start();
13    b.start();
14  }
15 }
16
17 class MyThread implements Runnable {
18   public void run() {
19     System.out.println("Helló, ez itt a " +
20       Thread.currentThread().getName() + " Thread" );
21   }
22 }
23
24 public class RunnableThread {
25   static public void main(String s[]) {
26     MyThread work2do;
27     Thread a, b;
28     work2do = new MyThread();
29     a = new Thread(work2do);

```

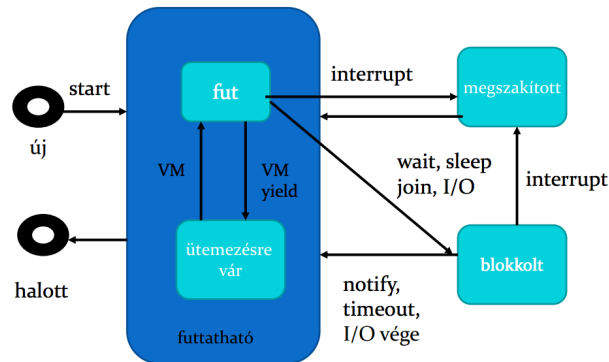
```

30     b = new Thread(work2do);
31     a.start();
32     b.start();
33 }
34 }

```

Szálak állapotai:

- Futó: a VM éppen ezt a szálát futtatja
- Kész (ready): futtatható, éppen áll, hogy egy másik szál futhasson
- Blokkolt: vár egy külső esemény bekövetkezésére (pl. I/O)
- Megszakított: a folyamatoknak jeleket küldhetünk, melyekre megállnak
- Halott: befejeződött a folyamat



27. ábra. Szálak állapotai

Szálcsoportok is létrehozhatók – a ThreadGroup segítségével – csoportszinten is kezelhetők a szálak. Futásvezérlés:

- Felfüggesztés – újraindítás
- suspend() - resume() pár, ezt ma már nem ajánlják – holtpontvesztély
  - pl. egy szál zárol egy erőforrást, és felfüggesztik ...
- Javaslat
  - segédváltozó és a wait() – notify() pár!
- Megszakítás – interrupt()
- Appleteknél a stop() metódusban kell a felfüggesztést is elengedni, ill. a megszakítást meghívni.
- Leállítás
  - volt: stop() – nem biztonságos
  - most: volatile Thread referencia, ami a futó szála mutat, ha a szálát leállítják, ez null lesz, amit a run() vizsgál
- Szálak összekapcsolása – join() – szinkronizálásra – megvárja, amíg a másik befejeződik (vagy egy adott ideig vár)

- Prioritások kezelése – 10 szint, állítható

Versengés:

- több szál szimultán szeretne ugyanahhoz az erőforráshoz hozzájutni:
  - synchronized blokk
    - \* objektum- és osztályszinten
- Problémák
  - író/olvasó esetben egyszerre csak egy írhat és olvashat, bár elvben több is olvashatna

A synchronized kulcsszó

- Metódusok elé írhatjuk (de pl. interfészekben nem!)
  - Kölsönös kizárás arra a metódusra
  - Kulcs (lock) + várakozási sor
    - A kulcs azé az objektumé, amelyiké a metódus
    - Ugyanaz a kulcs az összes szinkronizált metódusához
1. Mielőtt egy szál beléphetne egy szinkronizált metódusba, meg kell szereznie a kulcsot
  2. Vár rá a várakozási sorban
  3. Kilépéskor visszaadja a kulcsot

Szinkronizált blokkok

- A synchronized kulcsszó védhet blokk utasítást is
- Ilyenkor meg kell adni, hogy melyik objektum kulcsán szinkronizáljon
  - synchronized(obj)...
- Sokszor úgy használjuk, hogy a monitor szemléletet megtörjük
  - Nem az adat műveleteire biztosítjuk a kölcsönös kizárást, hanem az adathoz hozzáférni igyekvő kódba tesszük...
  - Létjogosultság: ha nem egy objektumban vannak azok az adatok, amelyekhez szerializált hozzáférést akarunk garantálni

Wait-notify:

- Szignálokra hasonlít
- Egy feltétel teljesüléséig blokkolhatja magát egy szál
- A feltétel (potenciális) bekövetkezését jelezheti egy másik szál
- Alapfeladat: termelő - fogyasztó (korlátos) bufferen keresztül kommunikálnak
  - egy termelő, egy fogyasztó
  - több termelő, több fogyasztó
- Minden objektumhoz tartozik a sima kulcshoz tartozó várakozási soron kívül egy másik, az ún. wait-várakozási sor
  - A wait() hatására a szál bekerül ebbe

- A notify() hatására az egyik várakozó kikerül belőle
- A wait() és notify() hívások csak olyan kódrészben szerepelhetnek, amelyek ugyanazon az objektumon szinkronizáltak
- synchronized(obj){ ... obj.wait(); ... }
- A szál megszerzi az objektum kulcsát, ehhez, ha kell, sorban áll egy ideig (synchronized)
  - A wait() hatására elengedi a kulcsot, és bekerül a waitvárakozási sorba
  - Egy másik szál megkaparinthatja a kulcsot (kezdődik a synchronized)
  - A notify() vagy notifyAll() metódussal felébreszthet egy vagy az összes wait-es alvót, aki bekerül a kulcsos várakozási sorba
  - A synchronized végén elengedi a kulcsot
  - A felébredt alvónak (is) lehetősége van megszerezni a kulcsot és továbbmenni
  - A synchronized blokkja végén ő is elengedi a kulcsot...

## GO

- Google nyelve
- 2007 – ben kezdték el fejleszteni
- Első hivatalos verzió 2012. május 28-án jelent meg
- Jelenlegi verzió: 1.4, 2014. december 10.
- Fordított, konkurens, imperatív programozási nyelv
- C-szerű, statikus típusozású nyelv
- Szemétgyűjtés

Go rutinok:

- Osztott memória helyett csatornákon való kommunikáció
- goroutine:
  - Konkurens folyamat
  - Közös címtérületen, saját veremmel, ami dinamikusan nő
  - Saját szemétgyűjtő
  - A tényleges szálak kezelését elrejt a programozó elől
  - A go kulcsszóval hívott függvény új goroutine-ban indul

```

1 import (
2     "fmt"
3     "time"
4 )
5 func IsReady(what string, minutes int64) {
6     time.Sleep(time.Duration(minutes));
7     fmt.Println(what, "is ready")
8 }
9 func main() {
10    go IsReady("tea", 4);
11    go IsReady("coffee", 2);
12    // nem var a befejezesre, ha o
13    // befejezte, akkor kész, ezért kell ide is egy
14    // Sleep...
15    time.Sleep(time.Duration(6));
16    fmt.Println("I'm ready...");
17 }

```

A Go a párhuzamosságot osztott memória helyett adatcsatornákon történő üzenetküldésekkel támogatja.

- `chan` // kétirányú csatorna
- `<-chan` // fogadó csatorna
- `chan < -` // küldő csatorna
- Az elemek típusa tetszőleges
- Az inicializálatlan csatorna értéke nil
- Új csatornát a `make()` függvénnyel hozhatunk létre
  - `make(típus, kapacitás)`
  - `make(chan int, 100)`
- Csatorna típusnak kell lennie
- Ha a buffer méret 0, akkor blokkol, amíg a fogadó goroutine nem áll készen fogadni, egyébként aszinkron

## C++11

- `std::thread` szabványos könyvtár
- `thread` osztály
- Szálkezelő függvények
- Kölcsönös kizárást biztosító osztályok: `mutex` és variációi: `timed_mutex`, `recursive_mutex`, `recursive_timed_mutex`, `shared_timed_mutex`

C++11 thread példa:

```
1 void egyik()
2 {
3     cout << "Ez az egyik.\n";
4 }
5 void masik(int x) {
6     cout << "Ez a masik."; cout << " A parametere: " << x << "\n";
7 }
```

A használata:

```
1 thread elso(ejik); // uj szalat indit, ami az egyik-et hivja
2 thread masodik(masik, 0); // uj szalat indit, ami a masik(0)-t hivja
3 cout << "main, egyik es masik most parhuzamosan futnak ...\n"; // szalak szinkronizalasa:
4
5 elso.join(); // var, amig elso befejezodik
6 masodik.join(); // var, amig masodik befejezodik
7
8 cout << "elso es masodik lefutott.\n";
9
10 mutex mtx; // mutex a kritikus szakaszra
11 void masik_mutex(int x)
12 {
13     mtx.lock();
14     cout << "Ez a masik.";
15     cout << " A parametere: " << x << "\n";
16     mtx.unlock();
17 }
```

Az előző indítás helyett:



```
1 thread.masodik(masik_mutex, 0);
```

### C++11 mutex példa:

```
1 int main () {  
2     std::thread th1 (print_block, 50, '*');  
3     std::thread th2 (print_block, 50, '$');  
4     th1.join();  
5     th2.join();  
6     return 0;  
7 }  
8  
9 #include <iostream> // std::cout  
10 #include <thread> // std::thread  
11 #include <mutex> // std::mutex  
12  
13 std::mutex mtx; // mutex a kritikus szakaszhoz  
14  
15 void print_block (int n, char c) {  
16     //kritikus szakasz (kizárólagos hozzáférés a std::cout hoz):  
17     mtx.lock();  
18     for (int i=0; i<n; ++i) {  
19         std::cout << c;  
20     }  
21     std::cout << '\n';  
22     mtx.unlock();  
23 }
```

## 12. A funkcionális és logikai programozás alapjai, példákkal

### 12.1. A funkcionális programozás alapjai

- Egy funkcionális program felépítése:
  - függvény definíciók, kezdeti kifejezés, operátorok, adatszerkezetek,...
  - típus definíciók
- Például - néhány egyszerű függvénydefiníció:
  - nulla  $x = 0$
  - id  $x = x$
  - inc  $x = x + 1$
  - square  $x = x \cdot x$
  - squareinc  $x$  square(inc  $x$ ) - kompozíció
- A funkcionális programozás végrehajtása:
  - a kezdeti kifejezés értékének meghatározása, kiértékelése, redukálása
  - a normál forma meghatározása, -egyértelmű, nincs lehetőség további redukciós lépésre
- A program végrehajtása a kezdeti kifejezésből kiinduló redukciós vagy más néven átírási lépések sorozata.
- Néhány kezdeti kifejezés
  - Clean: Start = sqrt 5.0;
  - Haskell: main = product \$ [1..10]
  - A normál formák: 2.23608, 63, 324564
- Egy-egy redukciós lépésbe egy - valamely kiértékelési stratégia szerint kiválasztott - redukálható részkifejezésben, a redex-ben szereplő függvényhívás helyettesítődik a függvény törzsében megadott kifejezéssel a formális és aktuális paraméterek megfeleltetése mellett.
- Normál formájú egy kifejezés, ha tovább redukcióra nincs lehetőség.
  - Ez az átírás lépéssorozat végeredménye

#### Kiértékelési stratégiák

- lusta kiértékelés
  - a kifejezések kiértékelésekor először a legbaloldalibb legkülső redexet helyettesíti - azaz, ha a kifejezés függvény megadással kezdődik, előbb a függvény definíció lesz alkalmazva, és az argumentumok kiértékelését csak szükség esetén végzi el.
  - A lusta kiértékelés normalizáló kiértékelési módszer, azaz mindig megtalálja a normál formát, ha létezik.
    - \* Pl. Clean, Haskell, Miranda
- mohó kiértékelés: a legbaloldalibb redex, az argumentumok helyettesítése történik meg először
  - hatékonyabb, mint a lusta rendszer
  - de: nem mindig ér véget a kiértékelési folyamat (Lisp, SML, Hope)
- a két stratégia kombinálható

#### Példák

- Mohó kiértékelés:

- squareinc 7
- $\rightarrow$  square (inc 7)
- $\rightarrow$  square (7 + 1) - először az argumentum(ok)
- $\rightarrow$  square 8
- $\rightarrow 8 \cdot 8$
- $\rightarrow 64$

- Lusta kiértékelés:

- squareinc 7
- $\rightarrow$  square (inc 7)
- $\rightarrow$  (inc 7)  $\cdot$  (inc 7) - először a függvény
- $\rightarrow (7 + 1) \cdot (7 + 1)$
- $\rightarrow 8 \cdot 8$
- $\rightarrow 64$

### Függvény definíciók:

- Nem mindegy, hogy milyen a kiértékelés, pl:

- $f\ x = f\ x$
- $g\ x\ y = y$  - ezek a függvény definíciók
- $g\ 1\ 2 \rightarrow 2$
- $g\ (f\ 3)\ 5 \rightarrow 5$  csak a lusta!

### A modern funkcionális nyelvek fő jellemzői

- Hivatkozási átláthatóság (referential transparency)

- A kifejezések értéke hivatkozásuk előfordulási helyétől független azaz ugyanaz a kifejezés a program szövegében mindenhol ugyanazt az értéket jelöli. A függvények kiértékelésének nincs mellékhatása, azaz egy függvény kiértékelése nem változtathatja meg egy kifejezés értékét!
- Pl:  $f(x) + 2 \cdot f(x) = 3 \cdot f(x)$  - biztosan
- Ez nem mindig igaz Pl:

```

1  int x = 1;
2  int f(int a){
3      ++x;
4      return a+x();
5  }
6
7  in main() {
8      int y = f(x) + 2*f(x);
9      int z = 3*f(x);
10 }

```

- $x \neq y$

- feltételek megadási lehetősége (definition by cases):

- a függvény értékét az első, igaz egyenlőséghez tartozó kifejezés adja, mely kifejezést az aktuális paraméter értéke alapján kapjuk

- rekurzivitás lehetősége (definition by recursion):

- a függvények hivatkozhatnak önmagukra és kölcsönösen egymásra
- Példa: két egész szám legnagyobb közös osztója:

```

1   lnko :: Int -> Int -> Int
2   lnko a b | (a==b) = a
3   lnko a b | (a>b)  = lnko (a-b) b
4   lnko a b | (a<b)  = lnko a (b-a)
5   main = print $ lnko 18 56

```

- A függvények ugyanolyan értékek, mint az elemi típusérték-halmazok elemei.
- Magasabbrendű függvénynek nevezzük azokat a függvényeket, amelyeknek valamelyik argumentuma vagy értéke maga is függvény.

- Példa: Twice  $d\ x = f(f\ x)$

- Mintaillesztés

- azt vizsgálja a kiértékelő rendszer, hogy az aktuális paraméterek értéke, vagy alakja megfelel-e valamelyik egyenlőség bal oldalán megadott mintának.

- \* a minták sorrendje meghatározza az eredményt
- \* az első illeszkedést keresi

- Példa: Faktoriális

```

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

```

- Szigorú, statikus típusosság

- Típusdeklarációk megadása nem kötelező, de megköveteljük hogy a kifejezések típusa a típuslevezetési szabályok által meghatározott legyen.
- Ez azt jelenti, hogy egy adott kifejezés legáltalánosabb típusát a fordítóprogram általában még akkor is meg tudja határozni a benne szereplő részkifejezések típusa alapján, ha a programszöveg készítője nem deklarálta a kifejezés típusát.

- Iteratív adatszerkezet elemeinek és azok sorrendjének megadására alkalmas, a matematikában halmazok megadásánál alkalmazott jelölésrendszernek megfelelő nyeli eszköz a Zermelo-Fraenkel halmazkifejezés.

- A végtelen adatszerkezetek kiértékelés lusta kiértékelési módszerrel történik.

- PL:

```

1   negyzetek n = [n^2 | x <- [1..n]]

```

- A függvények típusát értelmezési tartományuk és érték-készletük megadásával határozzuk meg. Pl:  $\text{int} \rightarrow \text{int}$
- Többváltozós függvények  $\rightarrow$  minden függvénynek egyetlen argumentuma lehet, de ez esetleges egy függvény!
- Egyszerű típuskonstrukciók

- rendezett nesek
- iterált - véges vagy végtelen sorozatok

- Sorozatok

- konstruktor - egy elemből és egy sorozatból új, az adott sorozatot balról kiterjeszthető sorozatot készít  
- mintában is használhatóak
- első elem (fejelem, head), lista fejelem nélküli maradéka (tail),...
- Pl:

```

1      ossz :: [Int] -> Int
2      ossz [] = 0
3      ossz (x:xs) = x + ossz xs

```

- Sorozatok generálhatóak is
- Párosak valameddig

```

1      paros n= [x | x <- [1..n], even x]

```

- Példák magasabb rendű fv-kre:
  - Filter - adott tulajdonságot teljesítő elemek leválogatása

```

1      filter :: (a -> Bool) -> [a] -> [a]
2      filter p [] = []
3      filter p (x:xs)
4      | p x = x : filter p xs
5      | otherwise = filter p xs
6
7      even x = x `mod` 2 == 0
8      evens = filter even [0..]
9      filter even [3,2,2,1] -> [2,2]

```

## 12.2. Logikai programozás

- Ha adott egy rendszer logikai formulákkal definiált modellje, kiegészítendő vagy eldöntendő kérdéseket fogalmazhatunk meg bizonyítandó állítások formájában.
- Kiegészítendő kérdés esetén a megfelelő tulajdonságú objektumok, tervek stb. meghatározása számítási folyamatnak tekinthető.
- Ez a logikai programozás (LP) alapgondolata

### Modell

- A logikai program egy modellre vonatkozó állítások (axiómák) egy halmaza. Az állítások a modell objektumainak tulajdonságait és kapcsolatait, (relációit) írják le.
- Ha például adottak állítások, amelyek arra vonatkoznak, hogy ki kinek az apja, ezek együtt az apja nevű, kettő aritású relációt írják le.
- Ha megmondjuk, hogy kik a férfiak, vagy kik a nők, akkor az erre vonatkozó állítások együtt a férfi, ill. nő nevű egy aritású relációt írják le.

```

apja('Ábrahám', 'Izsák').      apja('Ábrahám', 'Ismáel').
apja('Ábrahám', 'Ismeretlen').
apja('Izsák', 'Jákób').        apja('Izsák', 'Ézsau').

anyja('Sára', 'Izsák').        anyja('Hágár', 'Ismáel').
anyja('Rebeka', 'Jákób').      anyja('Rebeka', 'Ézsau').

férfi('Ábrahám').             férfi('Izsák').   férfi('Ismáel').
férfi('Jákób').               férfi('Ézsau').

nő('Sára').                   nő('Hágár').      nő('Rebeka').
nő('Ismeretlen').

```

28. ábra. Pl. atomi formulák

- Az állítások egy adott relációt meghatározó részhalmazát predikátumnak nevezzük.
  - A program futtatása minden esetben egy, az állításokból következő tétel konstruktív bizonyítása, azaz - a logikai programozásban szokásos szóhasználat - a programnak feltett kérdés, vagy más néven cél megválaszolása.
  - Ennek során a predikátumok eljárásokként működnek.
- A ma használatos LP nyelvekben minden, a predikátumokat alkotó állítás tény vagy szabály lehet.
- Prolog: Programming in Logic (1972-től)

#### Az előző példára visszatérve

```

|?- apja('Ábrahám', 'Izsák').
yes
|?- anyja('Sára', 'Jákób').
no
- Van-e olyan X, akinek az apja Izsák?
|?- apja('Izsák', X).
X = 'Jákób' ;      X = 'Ézsau' ;

```

29. ábra. Kérdéseket tehetek fel

#### Horn Klózok

- A logikai programok állításait Horn klózoknak is nevezik.
- Egy Horn klóz azt fejezi ki, hogy egy adott állítás igaz, ha nulla vagy több másik állítás igaz. Pl:
  - ha (if) egy személy öreg és bölcs, akkor (then) ez a személy boldog
  - ha (if) X az apja Y-nak és Y az apja Z-nek, akkor (then) X a nagyapja Z-nek
- Nulla vagy több állítás lehet if-részben, és pontosan egy következtetés a then-részben
- Egy feltétel nélkül Horn klóz egy tényt ír le.
- Egy Horn klóz a következő formában írható:
  - $G_0 \leftarrow G_1, G_2, \dots, G_n$ .
- Ennek jelentése:
  - Ha (if)  $G_1 \dots G_n$  mind igazak, akkor (then)  $G_0$  is igaz.
- A szabály baloldala a feje (head); a jobboldala a törzse (body)
  - $G_0 \dots G_n$  a célok

- A törzsben lévő célokat alcéloknak (subgoals) hívjuk
    - A vesszőket logikai AND szimbólumként kell olvasni
  - Megjegyzés:  $a \leftarrow$  csak azt jelenti, hogy "ha" ("if") - nem pedig azt, hogy "akkor és csak akkor" ("if and only if") - lehetnek más szabályok is  $G_0$ -ra! ( $a \leftarrow$  jelölése :-)
  - Speciális eset, ha  $n = 0$ , ekkor  $G_0$  egy egyszerű tény, ami mindig igaz.
  - A logikai programozás lehetővé teszi, hogy a programozó megadja a tények és if-then szabályok egy listáját, és azután a rendszer automatikusan el tudja dönteni, hogy egy állítás igaz-e, vagy meg tudja mondani, mire igaz stb.
  - Szabályok
    - 'Ábrahám lánya'(X) :- apja('Ábrahám',X), nő(X).
    - szülője(X,Y) :- anyja(X,Y).
    - szülője(X,Y) :- apja(X,Y).
  - A szülője reláció az állítások által definiált relációk uniója.
- ```

1   fia(X,Y) :- szülője(Y,X), férfi(X).
2   lánya(X,Y) :- szülője(Y,X), nő(X).
```
- A fia reláció a szülője és a férfi relációk metszete, és hasonlóan adódik a lánya reláció is.
  - nagyszülője(X,Y) :- szülője(X,Z), szülője(Z,Y).
  - Ez eltér az előzőektől abban, hogy a jobb oldalán új változó is előfordul. Így kétféle olvasata is van:
    - Minden X,Y,Z-re nagyszülője(X,Y), ha a szülője(X,Y) és szülője(Z,Y).
    - Minden X,Y-ra nagyszülője(X,Y), ha van olyan Z, hogy szülője(X,Z) és szülője(Z,Y).
  - Ezek a deklaratív olvasatai a szabálynak. Ezzel szemben áll a procedurális olvasat, ami a logikai programok futtatásához, vagyis a konstruktív bizonyítási eljáráshoz kapcsolódik.
  - Minden esetben felteszünk egy kérdést.
    - Ez a bizonyítandó cél (sorozat)
    - Ezután a részcélok bizonyítása, és így kiejtése a feladat. Felülről lefelé érdemes...

## Rekurzív szabályok

- Tegyük fel, hogy azt szeretném leírni, X mikor őse Y-nak.
- Világos, hogy akkor őse, ha szülője, vagy valamelyik ősének a szülője.
- Látható, hogy a relációnak két esete van, és az egyik eset rekurzív. Ezt ennek megfelelően egy nem rekurzív és egy rekurzív szabállyal fejezhetjük ki:
  - őse(X,Y) :- szülője(X,Y).
  - őse(X,Y) :- szülője(X,Z), őse(Z,Y).
  - kérdés: ?-őse('Ábrahám',X)
- a keresőfa előállításához célszerű a legbal részcélkiválasztási módszert alkalmazni - általában is érdemes a nem rekurzívvval kezdeni...

```

?- nagyszülője('Ábrahám',X).
  szülője('Ábrahám',Z),szülője(Z,X).
    anyja('Ábrahám',Z),szülője(Z,X). % meghiúsul
    apja('Ábrahám',Z),szülője(Z,X).
      { Z <- 'Izsák' }
        szülője('Izsák',X).
          anyja('Izsák',X). % meghiúsul
          apja('Izsák',X).
            { X <- 'Jákób' } % 1. megoldás
            { X <- 'Ézsau' } % 2. megoldás
      { Z <- 'Ismáel' }
        szülője('Ismáel',X).
          anyja('Ismáel',X). % meghiúsul
          apja('Ismáel',X). % meghiúsul
      { Z <- 'Ismeretlen' }
        szülője('Ismeretlen',X).
          anyja('Ismeretlen',X). % meghiúsul
          apja('Ismeretlen',X). % meghiúsul

```

30. ábra. Keresési fa a részcélokhoz

#### Felhasználási területek:

- mesterséges intelligencia
- számítógépes nyelvészeti
- adatbáziskezelés