Juhász István

PROGRAMOZÁS 2

mobiDIÁK könyvtár

Juhász István

Programozás 2

mobiDIÁK könyvtár

SOROZATSZERKESZTŐ
Fazekas István

Juhász István

Programozás 2

Egyetemi jegyzet
Első kiadás

mobiDIÁK könyvtár

Debreceni Egyetem
Informatikai Intézet

Lektor

Espák Miklós
Debreceni Egyetem

mobiDIÁK könyvtár
Debreceni Egyetem
Informatikai Intézet
4010 Debrecen, Pf. 12
http://mobidiak.inf.unideb.hu

A mű **A mobiDIÁK önszervező mobil portál** (IKTA, OMFB-00373/2003) és a **GNU Iterátor, a legújabb generációs portál szoftver** (ITEM, 50/2003) projektek keretében készült.

2. OBJEKTUMORIENTÁLTSÁG

Meg fogunk állni a programozási nyelvek szintjén.
Az objektum-orientált programozási módszertan filozófiáját – amelynek alapgondolata, hogy az adat és a funkcionális modell egymástól elválaszthatatlan, lényegében egyetlen modell – követik az objektum-orientált programozási nyelvek. Az OO programozási nyelvek imperatív jellegűek: algoritmikus szemléletet tükröznek. Az OO paradigma bevonul minden más nyelvi osztályba is.

- A ’60-as évek második felében jelenik meg a SIMULA67-ben az OO programozási eszközrendszer.
  A SIMULA67: Algol verzió, szimulációs, absztrakciós nyelv, minden objektum-orientált eszköz megvan benne. Észak-Európában születik meg.
- Alan Kay amerikai egyetemista 1969-ben szakdolgozatában egy új világot vázol föl: az OO világot. A Xerox-nál próbálja megvalósítani. Egy projektet szervez, melynek célkitűzése egy személyi számítógép megtervezése (hardver, architektúra, szoftver). Ő használja az objektum-orientált elnevezést, fogalmakat. (A grafikus interfész, ablak és egér fogalma is ekkor jelent meg először.) Felvázol egy projektet:
  - csinálni kell személyi számítógépet
  - Windows típusú operációs rendszert, grafikus felhasználói felületet (ekkor még csak az elektromos írógép az input periféria)
  - mindezt objektum-orientált programozási környezetben
- 1970-es évek elején megszületik a Xerox-nál a Smalltalk programozási nyelv, melyet objektum-orientáltnak terveznek, tiszta objektum-orientált nyelv, (SIMULA elemekkel). A világot csak objektum-orientáltnak hagyja láttatni, másnak nem. Ekkor még a strukturált elv a döntő.A Smalltatkkal együtt megjelenik egy olyan paradigma, amelyet a szakma még nem tud befogadni, másrészt olyan hardver kell alá, ami még nem létezik. A ‘80-as évek második felétől jelenik meg ilyen hardver. Ezután őrültmód elkezd terjedni az OO. Divattá válik. A Smalltalk napjainkban is él.
- Megszületik a ‘80-as években a C++, jelenleg divatnyelv.
- 1985-ben megjelenik a Meyer által kifejlesztett Eiffel nyelv, ami az OO területén azt a szerepet játsza, amit az eljárásorientált nyelvek területén a Algol. Nincs gyakorlati jelentősége.
- 1989: Turbo Pascal 5.5
- 1990: Turbo Pascal 6.0: OO eszközrendszerrel rendelkezik. A ‘80-as évek második

felében ‘90-es évek első felében minden magára valamit is adó programozási nyelvnek van olyan változata, amely már OO eszközrendszerrel rendelkezik valamilyen szinten.

– Java: az OO vallásának istene. A C++ óta egyetlen, aminek gyakorlati jelentősége is van.

Objektum-orientáltság jellemzői:

– az adatmodell és az eljárásmodell elválaszthatatlan (így szemléli a világot)
– absztrakt eszköz és fogalomrendszer: Az újrafelhasználhatóságot olyan magas szintre elviszi, ameddig lehetséges, a valós világot nagyon megközelíti.
– szemlélete: imperatív (algoritmus – kódolni kell) eszközrendszer

Jelen pillanatban az OO területén többféle iskola létezik, amelyek bizonyos pontokon élesen vitatkoznak egymással, nem csak nüansznyi különbségek vannak köztük. Jelen pillanatban folyik az OO matematikai hátterének elkészítése, kifejlesztése.

Az objektum-orientált programnyelvek fogalomrendszere

Objektum (object):

Az eljárásorientált nyelvek változó fogalmának kiterjesztése (általánosítása), olyan konkrét programozási eszköz melynek vannak:

– Attribútumai (attribute): ez az adatrész, a struktúra, tetszőleges bonyolultságú adatszerkezet. Szokás ezt az objektum statikus részének is nevezni.

  Minden objektum mögött van egy jól definiált tárterület, ezen vannak az attribútumok értékeit reprezentáló bitsorozatok.

  Terminológia: az obektumok állapotairól (state) beszélünk, ahol minden egyes állapotot egy-egy bitkombináció ír le, ami egy jóldefiniált címen van.
– Módszerei (method): a viselkedés leírására szolgál (eljárásmodell leírására) az eljárásorientált nyelvek eljárásai és függvényei. A módszerek adják meg nyelvi szinten az objektum viselkedésmódját (behavior).
– Azonossággal rendelkezik (van azonosság tudata): bármely objektum csak és kizárólag önmagával azonos, minden mástól megkülönböztetett. Minden objektumnak van azonosítója (OID: object identifier). Nyelvi

szinten ezzel nem foglakozunk.

Analógia:

változó – név

objektum – OID (nem egy név!)

A változó neve igazából soha nem azonosító csak hatáskörön belül egyértelmű a névhivatkozás. Az OID viszont tényleg egyedi, még programok között is!

Objektum viselkedése:

Az objektum állapota időben módosul(hat).

Módszerek csoportjai:

- le tudja kérdezni az objektum állapotát
- meg tudja változtatni az objektum állapotát

Objektumok élettartama:

Az objektumot létre kell hozni, és addig él, amíg meg nem szűnik. A megszüntetés lehet a nyelvi rendszer feladata, vagy a programozóé.

Az objektumazonosító minden szinten él, mindig léteznie kell.

Osztály (class):

Absztrakt eszköz, az eljárásorientált nyelvek típusfogalmának általánosítása (gyakran itt is típusként említjük - szinonimák). Az osztály absztrakt adattípus abban az értelemben, aholy az Adában a korlátozott privát típust használjuk. Az osztály azonos attribútumú és módszerű objektumok együttese. Az osztályhoz köthetőek az objektumok; az osztályból származtathatóak az objektumok.

Példány (instance):

Az osztályon belül létrehozok egy objektumot: példányosítás (instantiation).

- Az adott objektum adott osztály példánya. Minden objektum tudja, hogy melyik osztálynak példánya.
- Adott osztályhoz tartozó minden példány ugyanolyan attribútumokkal és módszerekkel rendelkezik. Minden példány tudja, hogy milyen módszerekkel rendelkezik.
- A módszereket mindig konkrét példányon futtathatom le, ezen értelmezhetők: az aktuális példányon.

- Példány létrehozása: ugyanaz az adatszerkezet újra és újra megjelenik a tárban. A módszereket nem többszörözi !
- Létezhetnek olyan attribútumok és olyan módszerek, amelyek nem arra szolgálnak, hogy az egyes példányok állapotait és viselkedését vizsgáljuk velük, hanem magához az osztályhoz tartoznak. (Példány-attribútum, példánymódszer; osztályattribútum, osztálymódszer)
  - Osztályattribútum: hány darab példánya van (az osztály kiterjedése).
  - Az osztályattribútumok nem többszöröződnek.

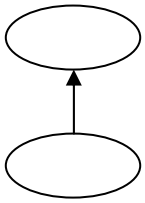
Az OO szemlélet szerint először létre kell hozni egy osztályt, leírni, hogy a hozzá tartozó objektumoknak milyen attribútumai és módszerei legyenek. És ezek után az osztályhoz kapcsolódóan és osztályon belül létre lehet hozni objektumokat. Példányosítás után az osztály példányairól beszélünk.

Öröklődés (inheritance):

Az újrafelhasználhatóság eddig legteljesebb válasza: objektum-orientált programozási elv: az osztályok nem függetlenek egymástól, speciális viszony értelmezhető közöttük, ez az öröklődés. Ez a viszony aszimmetrikus.

(Az absztrakciót a lehető legmesszebb elviszi, viszont a párhuzamosságra nem ad választ, bár az objektumok párhuzamosan léteznek. Nyelvi szinten nem mindenhol jelenik meg ez explicit módon. Az adatfolyamnyelvek adják a párhuzamosságra a legpozitívabb választ.)

Az öröklődés osztályokhoz kötött fogalom: két vagy több osztály között értelmezhető.A szuperosztályhoz kapcsolódóan tudunk létrehozni alosztályokat.



szuperosztály (superclass) / szülőosztály / ősosztály / alaposztály

alosztály (subclass) / gyerekosztály / származtatott osztály

Az alosztály átveszi, örökli a szuperosztály attribútumait és módszereit (azokat, amelyeket a láthatóság módszerével nem tiltottunk le).

Öröklésnél azonnal megvan az újrafelhasználhatóság, rendelkezésre áll az összes eszköz.

Az alosztály ezen túlmenően:

- új attribútumokat vezethet be
- új módszereket vezethet be
- újraimplementálhatja a módszereket
- törölhet attribútumokat
- törölhet módszereket
- a láthatósági szabályokat újraértelmezheti, hatásukat felfüggesztheti
- átnevezhet attribútumokat
- duplikálhat attribútumokat
- duplikálhat módszereket

Öröklés: valamit egy az egyben átvehetek, ha akarom, módosíthatom.

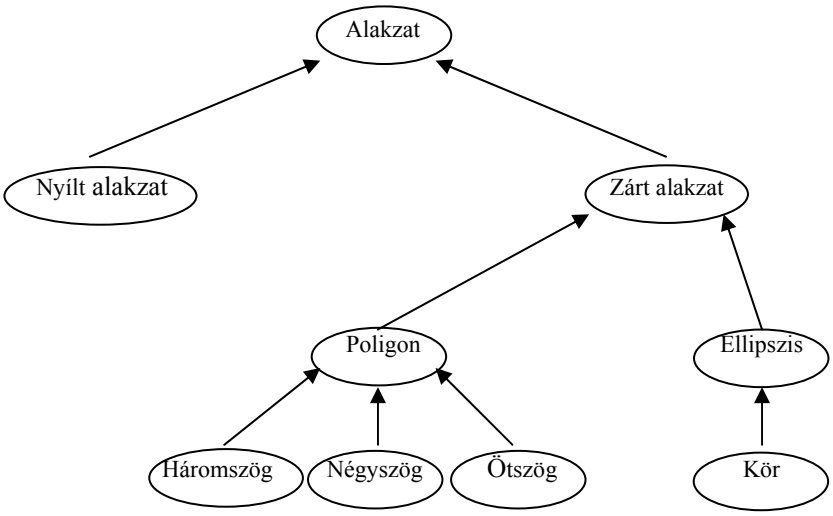
Aszimmetria: a szuperosztály nem látja, nem manipulálhatja alosztályait, de fordítva igen. A szuperosztályt teljes mértékben látja az alosztály. Az alosztály minden objektuma objektuma a szuperosztálynak is! Viszont fordítva ez nem áll fenn. Így minden rendszerben: mindenütt, ahol egy szuperosztály egy példánya szerepel, szerepelhet az alosztály egy példánya is és fordítva nem igaz. Egy osztályból tetszőleges számú alosztály származtatható minden nyelvben.

Az egyes rendszerekben kérdés, hogy: az alosztálynak hány szuperosztálya lehet?

- egy: egyszeres öröklődés (single)
- akárhány: többszörös öröklődés (multiple). Problémák: azonos nevű attribútumok, módszerek esetén: névütközés; ezt a rendszernek kezelnie kell. Rendszerfüggő, hogy hogyan teszi.

Alosztályból másik alosztály származtatható: öröklési hierarchia. Ez egyszeres öröklődés esetén fa., többszörös öröklődés esetén aciklikus gráf.

Öröklési fa:



Például az Alakzat osztály

- attributumai lehetnek: vonalvastagság, nagyság, szín, háttér, kitöltöttség ...
- módszerei lehetnek: kirajzol( ), elforgat( ) …

Az Alakzatot elkezdem specializálni, ekkor a Zárt alakzatnál jöhetnek mégújabb

- jellemzők: terület, kerület …
- módszerek: területszámítás( ), kerületszámítás( ) …

A Zárt alakzat Alakzat is egyben, így a Zárt alakzat minden példánya az Alakzatnak is példánya (is_a).

A Háromszögnek is lehet egy terület( ) módszere: átveszem a Zárt alakzattól, de ezt újraimplementálom, hiszen a Zárt alakzat területét csupán közelítőleg tudom megadni, míg a Háromszögét pontosan.

Terminológia:

- Fa gyökéreleme: ősosztály, amiből az összes többi származik.
- Előd: pl. a Kör elődjei: Ellipszis, Zárt alakzat, Alakzat.
- Leszármazott: pl. a Zárt alakzat leszármazottjai: Ellipszis, Kör.
- Kliens osztályok: azok az osztályok, amelyek között nincsen öröklődési kapcsolat. Pl. Kör – Ötszög.

Bezárás (encapsulation):

Az OO nyelvek legkényesebb fogalma: általában e fogalom mentén válnak el az iskolák, attól függően, hogy melyik mit vall róla. Az eljárásorientált nyelvek hatáskör fogalmának, a láthatóságnak a kiterjesztése. A legtöbbet félreértelmezett fogalom.

Nem objektumhoz kapcsolódik.

- Bezárás_1: Nem objektumhoz kötődik. Az osztály egy absztrakt adattípus. Az osztály rendelkezik egy interfész és implementációs résszel. Az osztály objektumaihoz csak az interfész részen keresztül férhetünk hozzá, az implementációhoz egyáltalán nem, korlátozott hozáférést jelent. Ez az információrejtés elve (Information hiding). Egy osztály objektumai egy az osztály által definiált interfészen keresztül érhetők el, és csak így! A nyelv a benne definiált attributumokat és metódusokat két részre osztja:
  - Nyilvános rész: amelybe tartozó eszközöket minden kliens osztály lát.
  - Privát rész: kívülről nem látható.
- Bezárás_2: A bezárás eljárásorientált nyelvek hatáskör fogalmának általánosítása OO körökben, ahol garantáltan létezik egy olyan eszközrendszer, mellyel a programozó tudja szabályozni, hogy az osztályból mi látható és ki számára.

Polimorfizmus (polimorphism): vagyis többalakúság. Kétfajta polimorfizmus van:

- Objektum polimorfizmus: Minden objektum tudja saját magáról, hogy melyik osztály példányaként jött létre. Egy objektum objektuma saját osztályának, de az öröklődési hierarchiában objektuma valamennyi elődosztálynak is egyben. Így minden egyes objektum szerepelhet minden olyan szituációban, ahol az ősosztály objektuma szerepelhet, nem csak a saját osztálya példányaként használható.
- Módszerpolimorfizmus (overriding):
  Egy leszármazott osztály egy örökölt módszert újraimplementálhat: a módszer specifikációja változatlan marad, de az implementáció más lehet az öröklődéi vonalon. Ld.:
  - Zárt alakzat: terület( ), kerület( ) módszer
  - Háromszög: terület( ), kerület( ) módszer *más*! (új implementáció)

Kötés (binding):

A módszerpolimorfizmushoz kapcsolódik. Ha van egy függvény és több implementáció hozzá, kérdés, hogy mikor melyik kód kapcsolódik a specifikációhoz. Eszerint beszélünk:

- Statikus (static) más néven korai (early) kötésről: a névhez a kód hozzárendelődik fordítási időben. Az OO rendszerek többsége fordítóprogram orientált.
- Dinamikus (dynamic) vagy késői (late) kötésről: kötés futási időben történik, így ugyanahhoz a névhez más-más kód tartozhat, attól függően, hogy melyik osztálykörnyezetben dolgozunk: az aktuális példány osztályában definiált kód, vagy (ha nincs) a hierarchián felfele a legközelebbi kód kötődik.

A nyelvek többsége mindkét kötést ismeri, kérdés, hogy melyik az alapértelmezett.

Üzenet (message):

Tipikusan Smalltalk fogalom. A Smalltalk filozófia szerint az objektum üzenetek segítségével kezelhető. Ha az objektumtól kérni akarok valamit, akkor üzenetet küldök. Az objektum veszi az üzenetet, én nem tudom, mi történik közben, nem tudom, hogy az objektum hogyan találja ki a választ, és az objektum válaszol.

Absztrakt osztályok

Absztrakt osztályoknak hívjuk azokat az osztályokat, amelyeknek nincsenek példányaik, amelyek nem példányosíthatók. Csak öröklтetésre való. Beszélnek nyelvek absztrakt módszerekről. Ezek azok a módszerek, amelyeknek csak a specifikációjuk van megadva implementáció nélkül. Az abszrakt osztályokból konkrét, példányosítható osztályok származtathatók. Az egész eszközrendszer az absztrakciót szolgálja. A rendszerfejlesztési ciklusban és a programfejlesztésnél lesz érdekes.

Konténer osztályok (Container)

Olyan adatszerkezet, amely objektumokat tartalmaz.

Alapvető a tömb, láncolt lista, verem, sor, stb. Nem minden nyelvben vannak realizálva a konténer osztályok, a programozónak kell megvalósítania. Alapvető szerepük az adatbáziskezelőknél van.

Kollekciók (Kollection)

Objektum-orientált adatbázisok esetén a konténerosztályok helyett a terminológia: kollekció (Collection). Ezen kollekcióval kapcsolatos az iterátor fogalma.

Iterátor

Általában ez is egy osztály, típus, ennek példányaihoz tartozó objektumokat be tudjuk járni. A bejárás az adatszerkezeteknek megfelelően történik.

Paraméterezett osztályok:

Egyes objektum-orientált nyelvekben vannak ún. paraméterezett osztályok, a C++ terminológia szerint template-k. Lényegében megfelelnek osztályszinten az Ada generikusnak.

Objektumok élettartama:

A példányosítás mindig egy explicit tevékenység eredménye, minden objektumot minden nyelvben a programozó hoz létre. Meddig él?

- A nyelvek egy részénél az objektumot megszüntetni is explicit módon kell, az objektumok törlése is a programozó feladata. A nem tisztán objektum-orientált nyelvek egy része vallja ezt az elvet. Ld. C++.
- A nyelvek másik része (nagyobb része) alkalmaz egy automatikus objektum törlési mechanizmust (garbage collection), amelynek a feladata az objektumok megszüntetése aszinkron módon úgy, hogy azzal a programozónak ne kelljen foglalkoznia, és úgy, hogy a törölt objektumok tárhelye ismét felhasználható legyen. Ez az automatikus tárfelszabadítás nem csak az objektum-orientált nyelvek sajátja, hanem egy tárkezelési technika. Többféle algoritmus van arra, hogy a rendszer hogyan dönti el, hogy mely objektum törölhető. Nyilvánvaló, hogy garbage collection algoritmus sokkal kényelmesebbé teszi a programozást.

Egységesség:

A nyelvben létezik-e más eszközrendszer, mint az objektum? Minden objektum, vagy van olyan eszköz, ami nem az?

Ezek alapján az OO nyelveknek két nagy csoportja van:

– A tisztán OO nyelvek azt vallják, hogy minden objektum (osztály, attribútum, módszer, objektum). Csak olyan eszközöket tartalmaznak, amelyek obektumorientáltak, és nincs más eszköz. Pl.: Smalltalk, Eiffel csak OO elvek alapján működik. A tisztán OO nyelvek esetén e nyelvi rendszer egyetlen osztályhierarchiából áll. Például a Smalltalk egy osztályhierarchia. A programozás pedig nem más, mint definiáljuk a saját osztályainkat, és azokat elhelyezem az osztályhierarchiában: az adott osztályhierarchiát bővítük, és ezekből

származtatunk objektumokat.

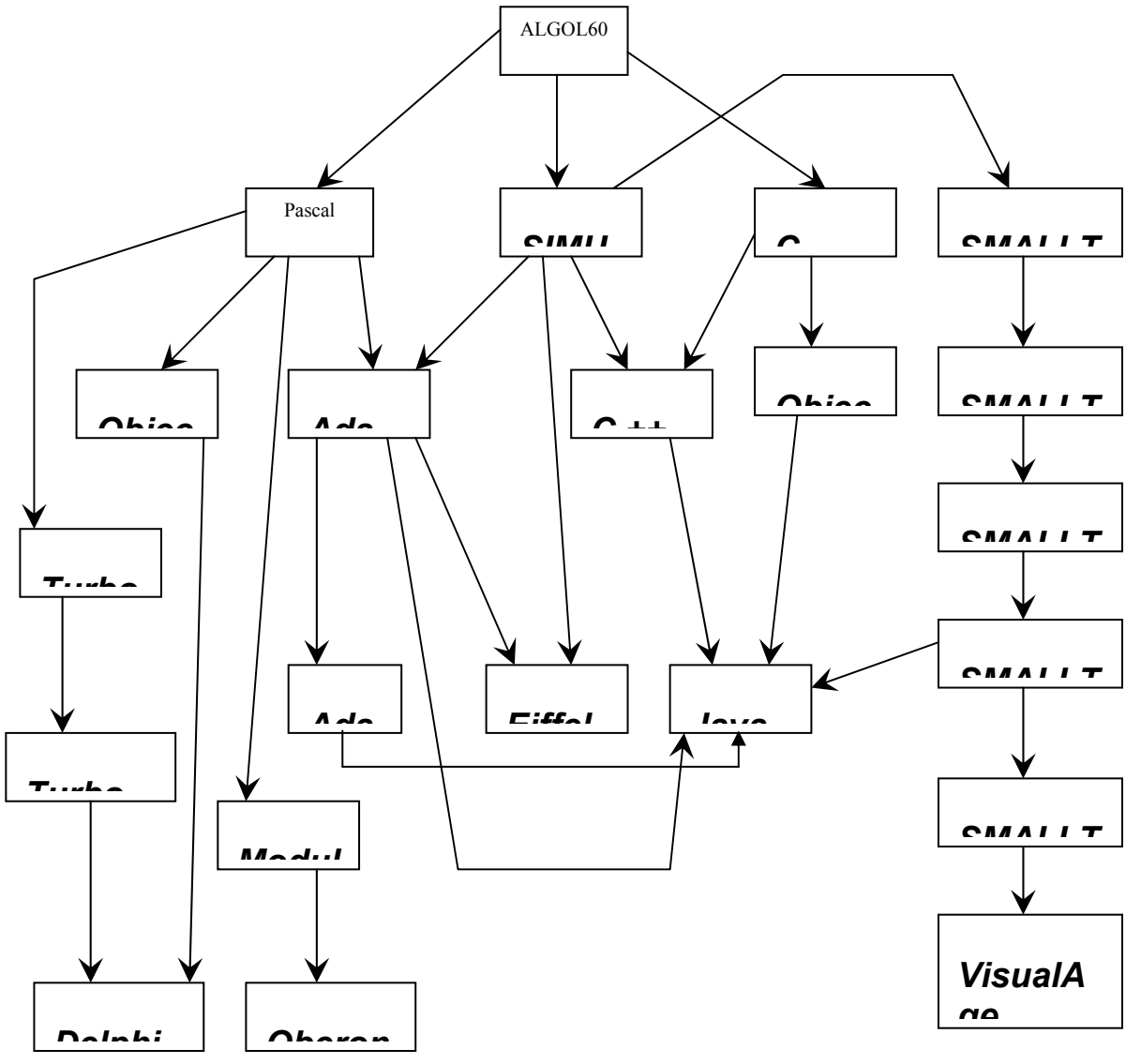
- A hibrid nyelvek alapvetően eljárásorientált, logikai, funkcionális, stb. nyelvi eszközöket tartalmaznak, és ez a nyelvi szközrendszer bővül OO eszközrendszerrel.Van tehát objektum is, és van nem objektum is. Lehetnek bennük eljárásorientált, deklaratív, funkcionális, objektum-orientált eszközök. Programozhatunk benne objektumorientáltan is. A nem tisztán OO nyelvek nem tartalmaznak osztályokat, incs osztályhierarchia. Definiálhatunk önálló osztályokat, és egymástól fügetlen osztályhierarchiákat. Itt is vannak szabvány osztálykönyvtárak, csak ezek nem a nyelv részei, és ezektől fuggetlenül is lehet programozni. Majdnem minden nyelvnek van olyan kiterjesztése, amelyben szerepelnek OO eszközök. Ilyenek például az OO COBOL, Object Pascal, C++.

Terminológia:

- Objektum alapú nyelvek: (object-based) ha a nyelvben van objektum fogalom és bezárás, de nincs osztály és öröklés. (Pl. Ada)
- Osztály alapú nyelvek: (class-based) van osztály, bezárás, objektum fogalom, de nincs öröklődés. (Pl.: CLU)
- Objektum-orientált nyelvek: (object-orinted) minden létezik: bezárás, osztály, öröklődés fogalom. Ezek a nyelvek (imperatív nyelvként) fordítóprogramosak.
- És végül létezik az OO-nak egy olyan speciális nyelve, amelyben nincs osztály fogalom, de minden más OO eszköz megvan benne.

Programozási nyelvek hierarchiája:

(nyíl: származik, hatás; aláhúzás: objektum-orientált)



3. JAVA

A Javát a SUN fejlesztette 1994-től.

A Java, mint nyelv nyílt, heterogén, tetszőleges platformokat egyesítő elosztott rendszerek nyelveként születik meg. A C++ tökéletesítéseként jön létre. A Java megszünteti a C++ néhány eljárásorientált nyűgjét és a mutató-orientáltságot. Meghagyja a kifejezésorientáltságot. Tisztább objektum-orientált nyelv, mint a C++, majdnem tiszta OO nyelv, kevésbé hibrid nyelv. A Java tervezési célkitűzése a hordozhatóság. Fordítóprogramos, de a Javaban megírt forrásprogramot a Java-fordító ún. bájtkód formátumra fordítja le (ez a bájtkód hordozható), és ezt a lefordított programot pedig az ún. Java Virtuális Gép (Java Virtual Machine - JVM) futtatja le. A JVM tehát egy interpreter, de készíthető hozzá célhardver, amelynek gépi kódja a bájtkód, illetve megadható olyan fordító, amely a bájtkódot valamely konkrét gépi kódra fordítja le.

Egyrészt a JVM tekinthető egy:

- Absztrakt számítógépnek, amelynek a gépi kódja ez a bájtkód, és interpreteres, amelyet (általában szoftveresen) szimulálunk egy konkrét platformon. Ennek a koncepciónak a nagy előnye, hogy a Java programok hordozhatóak. Ugyanaz a bájtkód jön létre, ugyanaz a JVM van megírva minden platformon. Egységes. A JVM feloldja az egyes platformok közötti különbségeket.
- Elképzelhető, hogy a JVM egy tényleges hardver, a bájtkód mögött egy célhardver van. És ezt a bájtkódot továbbfordítom egy adott processzor gépi kódjára.

A Java nyelv:

- A Java meglehetősen magas szintre elviszi az absztrakciót.
- Hálózati környezetben biztosítja az újrafelhasználhatóságot.
- Párhuzamos programozást is lehetővé tesz.

Egyesíti az eddig tárgyalt fogalmakat.

Karakterkészlet

Az UNICODE 16 bites kódtáblán alapul, ahol betű minden nyelv betűje lehet. Tartalmazza az összes nemzetközi abc-t.

Egy baj van, hogy a különböző platformok, az operációs rendszerek egy része ezzel az UNICODE-dal nem tudnak mit kezdeni, nem tudja kezelni ezt, tehát konkrét platformon ez az előny nem érvényesül. Ezért maradunk az eddig megszokott ASCII valamely változatánál. A kis- és nagybetűt megkülönbözteti.

Azonosító

Az azonosító fogalom a szokásos, azzal a megjegyzéssel, hogy betű nem csak az angol abc betűi, hanem a nemzetközi betűk, és az _ és $ jel is ide tartozik.

Kulcsszavak

Eléggé C++ szerűek. A `CONST` és a `GOTO` alapszó, de nincs implementálva, nincs mögötte semmi.

Standard azonosító

Nincs.

Megjegyzés

Háromféle megjegyzés formátummal dolgozik:

- // -től sorvégéig megjegyzés.

- /* */ zárójelek között tetszőleges hosszan, ahol szóköz az elhatároló.
- /** */ dokumentációs megjegyzés. (Ezzel nem foglalkozunk.)

Címke

A címke azonosító, utasítás előtt áll, kettősponttal elválasztva. Minden utasítás címkézhető.

Típus: Nem objektumok

Mivel léteznek a Javában olyan eszközök, amelyek nem objektumok, ezért majdnem tiszta OO nyelv.
Ilyenek például:

- Beépített típusok:
  - `boolean`
  - `char`
  - `byte`
  - `short`
  - `int`
  - `long`
  - `float`
  - `double`

    Ezek az eljárásorientáltság értelmében típusok. Nincs mutató típus, de van logikai. Ezek tartományainak elemei jelenhetnek meg literálként.

    Szerepük: változókat lehet velük deklarálni, amelyek szintén nem objektumok. Létezik tehát a Javában a hagyományos változó fogalom.

- Beépített osztályok: az egyes típusok osztályváltozata, amelyek példányosíthatók. Nevük ugyanolyan, mint a beépített típusoké, annyi különbséggel, hogy ezek mindig nagybetűvel kezdődnek. Pl. `Boolean`, `Char`, `Byte`, stb. (Minden típusnak létezik az osztályváltozata.)

  A strukurált típusok közül létezik az egydimenziós tömb**.** Többdimenziós tömb nincs. Az index 0-tól indul. A név mögött [ ] zárójel jelzi, hogy tömbről van szó.

Literál

Léteznek C-szerű literálok, amelyek a beépített típusok értékkészletéből valók.

- `'a'` : karakteres literál
- `"alma"` : sztring literál

Beépített nevesített konstansok

Ilyen például:

- `true, false`
- `null` objektum

  Nevük foglalt szó.

Változó

Változó deklaráció C-szerű:

```
típus névlista;
```

A deklarációban kifejezéssel kezdőérték adható a változóknak futási időben kiértékelődő kifejezéssel:

```
int x=0;
```

Automatikus kezdőértékadás nincs. Az inicializálást a Java megköveteli.

Kifejezések

A C kifejezés fogalmát veszi át a C++-on keresztül. C, C++ szerű a precedencia táblázat és kiértékelés, de a kiértékelést a hivatkozási nyelv szabályozza. Jóval kevesebb implementációfüggő rész van, mint a C-ben. Ha a C-ben meghívok két függvényt, és összeadom őket, akkor implementációfüggő a kiértékelés, de a Javában nem. Szigorúan típusos nyelv, azzal a megkötéssel, hogy a konverziót bizonyos esetekben megengedi (nem úgy, mint az Ada).

Utasítások

1. A *deklarációs rész* csak változó deklarációból áll.
2. A *végrehajtható utasítások*:
   - Kifejezés utasítás. Ld. értékadó kifejezés.
   - Új objektum létrehozása. Példányosítás, ami a Java szempontjából egy művelet. Ld. precedencia táblázat.
   - Módszerhívás.
   - Vezérlő utasítások: ld. később. C, C++ szerű.

Blokk:

- A blokk { } zárójelek között szerepel.
- Cimkézhető.
- Tetszőleges mélységben egymásba skatulyázható.
- Változó blokk lokális változójaként deklarálható.
- A blokkon belül tetszőleges a deklarációs- és végrehajtható utasítások sorrendje.
- A lokális változók hatásköre statikus. Létezik lokális változó, de nincs lyuk a hatáskörben szituáció, mert a Java nem engedi meg az újradeklarálást.
- Ezen változók élettartama dinamikus.

A nemobjektumok szerepe a módszerek törzsén belül van.

Vezérlő utasítások

Feltételes utasítás

```
IF(feltétel) utasítás ELSE utasítás;
```

Teljesen C-szerű, annyi eltéréssel, hogy a C-vel ellentétben a feltétel logikai típusú.

Többszörös elágaztatás

```
SWITCH(egész_kifejezés) {
     CASE egész_literál: utasítások
     [CASE egész_literál: utasítások] ...
     [DEFAULT: utasítások]
}
```

Előfeltételes ciklus

```
WHILE(feltétel) utasítás;
```

Végfeltételes ciklus

```
DO utasítások WHILE(feltétel);
```

Akkor ismétel, ha a feltétel igaz.

Előírt lépésszámú ciklus

```
FOR(p1; p2; p3) utasítás;
```

Break utasítás

```
BREAK[cimke];
```

A fenti konstrukciókban is ott vannak a blokkok.

A `break` utasítás befejezteti azt a legbelső blokkot, amelyben ezt az utasítást kiadtam. Ha meg van adva az opcionális `címke`, akkor bármelyik szintről befejeztethetek egy egész blokksorozatot. Viszont módszerekből és inicializáló blokkokból nem lehet kilépni.

Continue utasítás

```
CONTINUE[címke];
```

Ciklusok esetén adható ki: a törzs hátralevő részét nem hajtja végre, hanem visszatér a vezérlő részhez, a fejhez. Ha szerepel a `címke`, akkor az adott címkéjű ciklus fejére tér rá a vezérlés.

Return utasítás

```
RTURN[kifejezés];
```

Függvényben kötelező a `return kifejezés;`

Osztályok

Az osztály egy absztrakt adattípus implementációja! A hangsúly a típuson van.

A Java program legkisebb egysége az osztály, így a programozás az osztályok, megírásából áll. Az osztály egyben a legkisebb fordítási egység. Természetesen egy fordítási egység akárhány osztályt tartalmazhat, de egyet mindenképpen kell.
Az osztályok csomagokba szervezhetők. Ha egy osztályt lefordítunk, meg kell mondani, hogy melyik csomag osztálya. (Még visszatérünk rá.)
Egy osztály attribútumait változó deklarációk (adattagok – data member), módszereit függvény definíciók alkotják (amely függvényeket a Java tagfüggvényeknek – member function-nek hív). Közös néven tehát tagokról beszélünk. Egy Java osztály tagok segítségével építhető fel. Egy osztály minden példánya saját készlettel rendelkezik az adattagokból (példányváltozók).
A példányosítás során az adatoknak megfelelő tárrész lefoglalódik. Ahány példány van, annyiszor foglalunk helyet a tárban.
Egy osztály valamely módszerének meghívásánál meg kell adni, hogy melyik példányra hívtuk meg. Ez az aktuális példány.
A Java ismeri a példányváltozó, példánymódszer illetve osztályváltozók és osztálymódszerek fogalmát. A példányváltozók a példányok állapotait, a módszerek a példányok viselkedését írják le. Az osztályváltozók és osztálymódszereket a Java statikus tagoknak hívja. Ezek magához az osztályhoz kapcsolódnak, az osztályváltozókból egy-egy van és az osztálymódszerek, ezeken dolgoznak. Ezek akkor is működnek, ha egyetlen példányuk sincs.

Az osztály szerkezete:

- fej
- törzs

Fej:

```
[módosító] CLASS név [EXTENDS szuperosztály_név]
```

Absztrakt adattípus létrehozása a Javaban.

A Java az egyszeres öröklődés elvét vallja. Ha nem adok meg szuperosztályt, akkor automatikusan az Object osztályhoz fog kapcsolódni az adott osztály. Az osztályhierarchia egy fa, amelynek gyökere az Object osztály. Terminológia: a Java időnként nem öröklődésről, hanem kiterjesztésről beszél.

Módosítók a következők lehetnek:

- `abstract`: absztrakt osztály definíció, nem példányosítható.
- `final`: ennek segítségével olyan osztályt definiálunk, amely osztály nem kiterjeszthető. Az osztályhierarchiában levélelemet definiálunk ezzel, nem lehet belőle örökölni.
- `public`: ezzel olyan osztályt definiálunk, amely bármelyik csomagból látható. (Egyébként csak abban, amelyikben van.)

Több is szerepelhet belőlük értelemszerűen egyszerre: pl. `abstract` és `final`-nak nem lenne értelme.

Törzs:

`{ }` zárójelben tetszőleges sorrendben tetszőleges számú tagdefiníció áll.

Attribútumok

Egy példányváltozó definíciója:

```
[módosító] típus név [=kezdőérték] [,név [=kezdőérték]]...
;
```

– A kezdőérték kifejezés. Az objektum származtatáskor sem lehet hatrozatlan állapotban, alapállapotba kell hozni (ld. konstruktorok). Ha nincs explicit kezdőértékadás, akkor a fordító implicit kezdőértéket ad a példányváltozóknak:
  – logikai típus esetén: `false`
  – a többi beépített típus esetén: `0`

  - objektum típus esetén: `null` értékre állít.
- A módosító:
  - Itt is létezik `final`, amely egy konstans adattagot ír elő (nem változtatható adattag), vagyis ha egy explicit kezdőértékkel ellátott adattagot definiálunk, az nevesített konstans lesz. Ha final esetén nem adunk meg kezdőértéket, akkor ez egy üres nevesített konstans lesz, ennek kezdőértékét konstruktorral kell rögzíteni.
  - A láthatóságot (a bezárást) a következő módon szabályozza:
    - Alapértelmezés szerint (amikor nincs alapszó) "friend" azaz félnyilvános. Ekkor ez az adattag az adott csomagból látszik, ahol az osztályt elhelyeztem.
    - A `private` (privát) alapszóval ellátott adattagot csak az adott osztály látja. Ezeket az osztály bezárja.
    - Ha a `protected` (védett) az alapszó, akkor ezt csak a leszármazottak láthatják.
    - A `public` (nyilvános) alapszóval deklarált adattag mindenhonnan látszik.
    - Vannak még további módosítók, amelyekkel később foglalkozunk.

Módszerek

A módszerek definíciója a következőképpen néz ki:

```
[módosító] fej törzs
```

- A fejet szokás objektum-orientált körökben *signature*-nek (lenyomat, szignatúra) hívni (ez a korábbi specifikációnak felel meg), és a törzs a szokásos. Ld.: C, C++. (Kiegészül majd a kivételkezelésnél.)
- A módosítók a példányváltozóknál megbeszéltek, plusz bejönnek újabb módosítók:
  - `final`: nem implementálható újra, nem polimorf.
  - `abstract`, mellyel absztrakt módszer definíciót írunk elő. Ilyenkor nincs törzs (implementáció), csak absztrakt osztályokban szerepelhet értelemszerűen. Valamely leszármazott adja meg az implementációt. Egy osztály mindaddig absztrakt, míg legalább egy módszere absztrakt. Absztrakt osztályokban szerepelhet nem absztrakt módszerdefiníció is.
  - A `static` módosítóval statikus tagokat definiálhatunk.
  - Itt is vannak további módosítók, amelyekkel később foglalkozunk.

Az osztályon belül definiált módszerek látják az osztályon belül definiált minden adattagot. Amikor egy módszert meghívok, az nem más, mint egy függvényhvás.

Paraméterkiértékelés módszereknél:

- sorrendi kötés

- számbeli egyeztetés
- típusegyeztetés

Látni fogjuk, hogy e két utóbbi nagyon-nagyon lényeges.

Paraméterátadás:

A paraméterátadás értékszerinti.

Hivatkozás

Minősítéssel hivatkozunk egy tagra.

```
csomagnév.osztálynév.objektumnév.tagnév
```

Példa osztálydefincióra a Java útikalauzból:

```
public class Alkalmazott{
     String nev;
     int fizetes;
     void fizetesEmel(int novekmeny){
       fizetes = fizetes+novekmeny;
     }
     boolean tobbetKeresMint(Alkalmazott masik){
          return fizetes > masik.fizetes;
     }
}
```

- Van két példányváltozó:
  - `név`, ami a `String` osztály egy példánya
  - `fizetes`, ami egy egyszerű típusú adattag.
- Van két példánymódszer:
  - `fizetesEmel` : egy eljárás
  - `tobbetKeresMint` : egy függvény, benne `return kifejezés;`

Konvenció:

- Az osztálynév nagybetűvel,
- A tagnév kisbetűvel kezdődik.

Módszer hívása:

- Vagy kifejezés
- vagy utasítás: valamit csinál, azaz mellékhatása van.

```
System.out.println(”szövegkiírás”);
```

Példányosítás

A `new` operátor segítségével történik, amely referencia típusú operátor. Amikor egy objektumot hozok létre, egy speciális változót definiálok. A változó neve segítségével nevezzük meg az objektumot. .

```
Alkalmazott a;
```

Definiálunk egy osztályt, az osztály ezáltal része lett a hierarchiának. Definiálhatok ilyen típusú változókat. Az `a` `Alkalmazott` típusú lesz. Ez nem példányosítás, nem rendelkezik értékkel.

Amennyiben egy `Alkalmazott` osztálybeli példányt akarok definiálni, a következő módon tehetem meg:

```
Alkalmazott a=new Alkalmazott();
```

Ekkor létrejön egy `a` nevű változó: lefoglalódik egy tárterület számára. A `new` hatására valahol hely foglalódik a példányváltozók számára, azok kezdőértéke beállításra kerül (a megírt vagy default `Alkalmazott()` konstruktor hatására), és az `a` változó értékül kapja a lefoglalt terület kezdőcímét.

Ez azonban nem egy mutató típus. Ezt a Java referenciának hívja. Különbség a referencia és a mutató között: az `a` változó mindig a mögötte lévő objektumot fogja hivatkozni, nem címként kezelendő, értéke nem machinálható: nem lehet például hozzáadni egy értéket, csak az `a` nevű objektummal dolgozhatok.

Lehet:

```
Alkalmazott a, b;  -- Változók, de nincs értékük,
                   -- vagy null értékűek (ha adattagok).

    a = new Alkalmazott();
    b = a;         -- a cím átmásolását jeleni,
                   -- nincs új terület foglalás
                   -- az a és b ugyanazt az obektumot hivatkozza
    a.fizetes=50000;
```

Ha valamely osztály definícióban ilyen adattagot definiálok, megtehetem:

```
final Alkalmazott a=new Alkalmazott();
```

ami annyit jelent, hogy definiáltam egy `a` nevű referencia típusú változót, amely mindig ugyanarra az alkalmazottra mutat (amely tetszés szerint változtathatja az állapotát), és nem címezhet más objektumot.

A JVM minden példány vonatkozásában tartalmaz egy referenciaszámlálót, amelyet a JVM az objektum létrejöttekor rendeli hozzá az objektumhoz. Ez a referenciaszámláló (mely megmutatja, hogy hány változó címzi az objektumot) nő, ha hivatkozok egy példányra, és ha megszüntetek egy hivatkozást, csökken. Erre épít egy

garbage collection-t (szemétgyűjtögetés). Ha a referenciaszámláló értéke 0, akkor az adott példány törölhető, a rendszer fölszabadítja a helyét. Automatikus.

```
a = null;
```

bármikor lehet. Ez megszünteti a referenciát. Nem kell explicit módon felszabadítani egy fölöslegessé vált objektum területét.

Statikus tagok

Statikus adattagok

Alakja:

```
static int nyugdijKorhatar = 65;
```

Osztálynévvel minősíthető:

```
Alkalmazott.nyugdijKorhatar;
```

Statikus módszerek

Például:

```
static void nyugdijKorhatarEmel(){

    nyugdijKorhatar++;

}
```

Osztálymódszereknél nincs példány.

Főprogram

A program indításakor egy osztálynevet kell megadni, amelyben van egy `main` nevű eljárás a következő specifikációval:

```
public static void main (string args[])
```

- A virtuális gép ennek adja át először a vezérlést.
- A specifikáció kötött
- Paramétere egy tömb, amely az indításkor megadott argumentumokat tartalmazza. Ezek megadása, szerepe rendszerfüggő.

- A program befejeződik, ha:
  - Ez befejeződik vagy
  - Ha a programon belül meghívjuk a System osztály `exit` módszerét.

Ha egy osztály definíciójánál nem adunk meg szuperosztályt, akkor automatikusan az Object osztály alosztálya lesz. Ez az osztály a hierarchia gyökere a Javaban.

Példa:

```
class Factorial{
     public static void main(String[] args){
          int factorial = 1;
          int i = 1;
          while(i < 10){
               factorial = factorial * i;
               System.out.println(i + "!=" + factorial + "  " );
               i++;
          }
          System.out.println();
     }
}
```

A `this` és a `super` pszeudóváltozó

Egy módszer törzsében használható a `this` és a `super` pszeudóváltozó. Ezzel hivatkozhatunk egy módszer törzsében a megfelelő aktuális példányra, illetve a megfelelő szuperpéldányra.

```
boolean kevesebbetKeresMint (Alkalmazott masik ){
     return masik.tobbetKeresMint(this);
}
```

A `this` és a `super` szavak túlterheltek a Javaban.

Módszerek túlterhelése

A módszernevek túlterhelhetők a Javaban, azaz több módszert ugyanazzal a névvel nevezhetünk meg, ha a formális paraméterek száma vagy a sorrendi kötés értelmében a típusa eltérő. Ekkor a módszer meghívásakor a megfelelő kódot a fordító az aktuális paraméterek száma és típusa alapján választja ki. A paraméterkiértékeléskor derül ki, hogy melyik a meghívandó kód. Ld. `Factorial` példánál a `println()` módszer esetén.

Polimorfizmus

Egy örökölt példánymódszer implementációja tetszés szerint megváltoztatható. Az újraimplementált módszernek felülről kompatibilisnek kell lennie az örökölt módszerhez, azaz:

- Specifikációjuk megegyezik.
- Az újraimplementált módszerek csak azokat a kivételeket válthatják ki, mint az örökölt módszer.
- A bezárást csak enyhíteni lehet, szűkíteni nem: pl. `protected` → `public` lehetséges, de fordítva nem.

A Java a dinamikus (késői) kötés elvét vallja, nem kell külön előírni, mint a C++-ban. (Nem kell `virtual`!) Egy módszer meghívásakor egy módszer nevéhez mindig az aktuális példány osztályában definiált, vagy a legközelebbi örökölt kód fog meghívódni. Ezzel szemben az osztálymódszerek statikusan kötnek, osztálymódszereket nem lehet átdefiniálni. (Egy osztályhierarchia van, és az újradefinálással ezt a hierarchiát rúgnánk szét.)

Adattagok elrejtése

Az összes tag öröklődik. A Java tiltja, hogy bizonyos tagokat elhagyjunk az öröklődés során. Viszont bevezeti az elrejtés (elfedés) fogalmát. Jelentése: az öröklődés során a leszármazott újraimplementálhatja a módszereket, és átdefiniálhatja az adattagokat, új tagokat definiálhat, de megszüntetni nem szüntetheti meg az örököltetett. Ezzel elrejti az eredetieket a hozzáféréstől az adott és az innen leszármazott osztályokban.

Konstruktorok

Amikor példányosítunk, a példány alapállapotát be kell állítani, ezt megtehetjük paraméterek segítségével. Ezt a célt szolgálják a konstruktorok (a konstruktorok tehát az alapállapot definícióját segítik elő), amelyek típus nélküli, az osztály nevével azonos nevű módszerek, amelyek láthatóságát szabályozhatjuk csak. A konstruktorok a példányosításnál automatikusan meghívódnak, és inicializálják azt. Az adattagoknak lehet kezdőértéket adni explicit módon, ha nem, akkor a rendszer inicializál.

A programozó egy osztályhoz tetszőleges számú konstruktort írhat, és ezek neve túlterhelhető (a pataméterek száma és/vagy típusa különböző kell legyen). Kostruktor kizárólag a new operátor mellett hívható meg. A konstruktorok nem örökölhetők.

A konstruktor törzsének *első* utasítása lehet egy adott osztálybeli, vagy egy szülőosztálybeli másik konstruktor meghívása a következő szintakszissal:

`this();` illetve `super();`

A programozó nem köteles megadni konstruktort. Ha a programozó nem ad meg konstruktort, akkor a rendszer automatikusan felépít egyet, méghozzá olyat, amely paraméter nélküli és a törzse üres.

Példa:

```
public class Alkalmazott{
        ...
        public Alkalmazott(String egyNev, int egyFizetes){
                nev=egyNev;
                fizetes=egyFizetes
                evesFizetes=12*fizetes;
        }
        public Alkalmazott(String egyNev){
                nev=egyNev;
                fizetes=30000;
                evesFizetes=12*fizetes;
        }
        ...
}
```

Két konstruktort definiáltam. Egyiknek egy paramétere van, a másiknak kettő. Ez alapján tudja a rendszer eldönteni, hogy aktuálisan melyiket hívjuk meg.

Említettük, hogy minden konstruktor törzsének első utasítása lehet egy másik osztályhoz tartozó konstruktor meghívása. Ebben az esetben átírható a következő módon:

```
public class Alkalmazott{
   ...
   public Alkalmazott(String egyNev, int egyFizetes)  {
       nev = egyNev;
       fizetes = egyFizetes;
       evesFizetes = 12 * fizetes;
    }
   public Alkalmazott(string egyNev){
       this(egyNev,30000);
   }
   ...
}
```

A szuperosztály konstruktorai nem öröklődnek, azokat mindig minden osztályhoz meg kell adni (vagy implicit). Viszont minden alosztály bármely konstruktora törzsének első utasításaként a `super` kulcsszóval meghívható a szuperosztály valamelyik konstruktora. (Például új adattagokat definiáltunk, de az átvetteket ugyanígy kell inicializálni.)
A konstruktorok aktuális paramétereit példányosításnál a `new` operátor paramétereiként *kell* megadni a következőképpen:

```
new konstruktor_név(aktuális_paraméter_lista);

Alkalmazott a = new Alkalmazott("Ó Pál ");
Alkalmazott b = new Alkalmazott("Jó Jenő", 55555);
```

A Java lehetővé teszi, hogy osztály-konstruktorokat definiáljunk, az osztály definíción belül akárhol, tetszőleges számút. Ezek lényegében blokkok, amelyek előtt ott áll a `static` kulcsszó. Ezek az ún. statikus-konstruktorok (statikus inicializátorok).
Definiálom az osztályt, a konstruktorokat. Amikor az osztályt először használom fel típusként, a rendszer végrehajt egy osztály inicializálást, melynek során automatikusan lefutnak az osztály-konstruktorok, abban a sorrendben, ahogy felsoroltam őket.

A `finalize()` módszer

Az osztályokkal kapcsolatos fogalomrendszert azzal zárjuk, hogy az Object osztályban létezik olyan módszer, amelynek `finalize()` a neve (`protected` és `void`). A `finalize()` olyan módszer, amelyet minden osztály örököl és átdefiniálhat. Minden osztály implementálhatja.
Amikor egy objektum felszabadulásra kerül (azaz az objektum megsemmisítésekor), a tényleges felszabadítás előtt lefut ez a módszer: "Az adott példány egy utolsó kívánsága. " Meghívása automatikus.

Létezik ennek egy osztály szintű változata is (osztálymódszer):

```
classFinalize()
```

Abszolút platform függő. Ez az osztálymódszer az osztály megszünésekor automatikusan lefut.

Mit jelent az, hogy megszünik egy osztály?

Interfészek

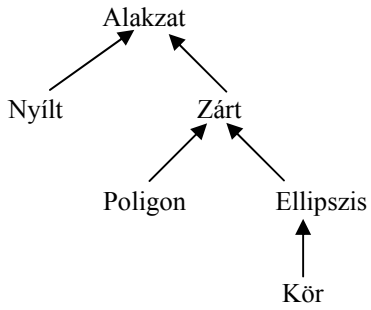
Speciálisan JAVA eszköz.

Az interfész egy speciális referenciatípus, amely konstans adatttagokat és módszer specifikációkat tartalmaz. Az interfész nem objektum.

A célja az absztrakciós szint növelése. Úgy tudunk problémát megoldani, hogy az implementációt nem adjuk meg. Programfejlesztés közben behozok egy absztrakciós szintet, amikor a specifikációval foglalkozom, és az implementációval nem. Az interfészek között is értelmezhető az öröklődés, méghozzá többszörös. Közös ősinterfész nincs, tehát a programfejlesztés folyamán interfész hierarchia-gráfok építhetők fel. Az interfészek implementációját mindig egy osztály végzi teljes mértékben. Az interfészt teljes mértékben implementálnia kell az osztálynak. Tehát az interfész hierarchia alján mindig osztályok állnak. Egy osztály tetszőleges számú interfészt implementálhat. Az interfészek által jelenik meg a többszörös implicit öröklődés a Javaban. Az interfész, mint referencia típus mindenütt szerepelhet, ahol az osztály, mint típus szerepelhet. Lehet vele változókat, tagokat definiálni, és formális paramétereket leírni. Az így definiált eszköz egy olyan referenciát kaphat értékül, amelynek a típusa egy olyan osztály, amely az adott interfészt közvetve vagy közvetlenül implementálja.

Múltkori példánkban eljárhattunk volna úgy, hogy az egyes osztályokat, mint interfészeket definiáljuk, és az Ellipszis interfészt implementálja közvetlenül a Kör osztály, és a Zárt alakzat és az Alakzat interfészeket közvetve.

Példa:



Az interfész definíciója:

```
módosító INTERFACE név [ EXTENDS szuperinterfész_nevek ]
törzs
```

Az interfész `módosító` – ja csak `public` lehet.

A `törzs`ben:

– az adattagok módosítói alapértelmezés szerint:
  `public static final`
  Nem írandók ki, és nem változtathatóak. Konstans adattagokról van szó, tehát a kezdőérték adás kötelező.
– a módszereknek csak a specifikációjuk szerepel
  A módosítók:
  `public abstract`.
  Nem kiírandók, és nem kiírhatók.
  A többszörös öröklődésből származó névütközéseket a Java nem kezeli.

Interfészek implementálása :

módosító CLASS név IMPLEMENTS interfész_nevek

törzs

Az összes absztrakt módszert implementálni kell.
Nagyon kemény absztrakciós eszköz és nagyon jól használható.

Csomag

Abban az értelemben, ahogy az Adában. A csomagok tartalmazzák a Java fejlesztői környezetet és az alkalmazásokat is csomagokban írjuk meg. A csomag egy hatásköri egység.
Fordítási egység a Javaban:
– osztály deklaráció vagy
– interfész deklaráció
– vagy ezek tetszőleges együttese.
Az osztályokat és az interfészeket együttesen hívja a Java típusnak. Tehát fordítási egységek a típusdeklarációk és ezek tetszőleges együttese.
A fordítási egység mögötti kód mindig kötelező módon csomagokban jelenik meg. A csomagok között hierarchia építhető föl (könyvtárszerkezet). A Java rendszer tehát csomagfák (csomagok alkotta fák) együttese, melyek tartalmazzák a fejlesztői környezetet és az alkalmazásokat is. A csomag tartalmazhat alcsomagokat és típusdeklarációkat tetszőleges mélységben.
Megnevezés: névvel. Hivatkozás: minősítéssel. Egyrészt a csomagfán belüli csomagra, másrészt csomagon belül: típus – objektum – tag.

Egy fordítási egység teljes szerkezete:

[CSOMAGLEÍRÁS] [IMPORTLEÍRÁS] [TÍPUSDEKLARÁCIÓK]

- `csomagleírás`:

  `PACKAGE csomagnév ;`

  A megadott nevű csomaghoz fog tartozni a lefordított kód. Ha nem szerepel, akkor a Java rendszer egy név nélküli csomagba tartozónak tekinti (ebbe most nem megyünk bele). A név nélküli csomagok kezelése rendszerfüggő. Nem javallott, hogy név nélküli csomagokat használjunk, a hordozhatóság sérülhet. Általában igaz, hogy egy név nélküli csomag van.

- `importleírás`:

  `import minősített_név ;`

  Gyakorlati eszköz. Az Ada környezet leírásának felel meg. Arra szolgál, hogy más csomagokban deklarált nyilvános típusok itteni használatát segítse elő úgy, hogy egyszerű és ne minősített névvel kelljen rájuk hivatkozni.

  Példa:

  ```
  import Alakzat.Zart.Ellipszis.Kor ;
  ```

  És ezután elég a `Kor` hivatkozás, de akkor ennek egyedinek kell lennie.

  Megengedett az:

  ```
  import Alakzat.Zart.Ellipszis.*;
  ```

  ekkor az összes publikus típust eléri a nem minősítet neve alapján.

- `típusdeklarációk`: itt osztály definíciók állnak.

A Java alapcsomagjai a java csomagban vannak. Ennek alcsomagjai:

- `java.lang`
- `java.io`
- `java.util`
- `java.sql`
- `java.net`
- `java.applet`
- `java.awt`
- `java.awt.event`

A `java.lang` tartalmazza a legalapvetőbb eszközöket, ennek minden nyilvános típusa *automatikusan* importálódik, nem kell külön megadni. Ebben van például az Object osztály, a primitív típusok, stb. Az összes többi tagot importálnom kell.

Kivételkezelés a Javaban

Adaszerű elveket vall. Ez is alapvető eszköze a Javanak. Ipari szabványszerű. A Java program működése közben módszerek hívódnak meg. Ha bekövetkezik egy speciális esemény egy módszer futása közben, akkor egy kivétel-objektum jön létre: vannak kivétel-osztályok és annak kivétel-példányai. Ekkor a módszer "eldobja" a kivételt, és a kivétel a Java virtuális gép hatáskörébe kerül át. Az adott módszer befejezi a futását annál az utasításnál, ahol a kivétel bekövetkezett, és jön a kivételkezelés. A JVM feladata, hogy megkeressen egy adott objektumnak megfelelő típusú, az adott pontba látható kivételkezelőt, amely kivételkezelő az adott kivételt elkapja.

Egy kivételkezelő megfelelő típusú, ha:

- a kivételkezelő típusa megegyezik a kivétel típusával
- a kivételkezelő típusa őse a kivétel típusának.

A láthatóságot maga a kivételkezelő definiálja. Maga a kivételkezelő egy blokk. Az Adában kivételkezelő fordítási egység végén helyezhető el, ezzel szemben a Javában tetszőleges kódrészlethez köthető. Ezek a kivételkezelők tetszőleges mélységben egymásba ágyazhatók. Ha van egy kivételkezelő, amely nem kezel minden kivételt, kérdés: hogyan tovább? A kivételkezelőblokkon lépked kifelé a JVM az Ada illetve a PL/1 filozófiája szerint dinamikusan, amíg nem talál megfelelő kivételkezelőt. Egy blokk végső soron egy módszer törzse. Ha nincs ott kezelve a kivétel, akkor a JVM továbbadja a kivételt a hívási láncon a hívónak. Ha talált megfelelő típusú kivételkezelőt a JVM, átadja annak a vezérlést, a kivételkezelő lefut, és a program folytatódik a kivételkezelő kódját követő utasításon.

A kivételkezelőben bekövetkezett kivételeket ugyanígy kezeli a JVM, mint bárhol máshol.

A kivételek két csoportjáról beszél a Java:

- ellenőrzött
- nem ellenőrzött kivételek

Elengedi az ellenőrzést olyan eseményeknél, amelyek bárhol bekövetkezhetnek, ellenőrzése vagy nagyon kényelmetlen vagy lehetetlen, irreálisan nagy kódtöbbletet eredményezne, vagy a programozó ezekkel a kivételekkel nem tud mit kezdeni. Ez utóbbi kivételek tartoznak a nem ellenőrzött kivételek közé. Javallott, hogy használjunk ellenőrzött kivételeket.

Az ellenőrzött kivételeket, amely egy metódus láthatósági körében felléphet, a programozónak mindig specifikálnia kell, vagy el kell kapnia őket. Ezt a fordító vizsgálja, és hibát jelez, ha ez nem teljesül. Biztonságos kódot kell írni!

Egy módszer fejében tehát meg *kell* adni azokat az ellenőrzött kivételeket kivételeket, melyeket a módszer nem kezel, de futás közben bekövetkezhetnek.

Ennek specifikálása a módszer fejének végén:

```
THROWS kivételnév_lista
```

utasításrész segítségével történik. Itt soroljuk fel, tehát a módszer láthatósági körében keletkezett, ellenőrzött, de nem kezelt kivételeket. A kivételek kezeléséhez a `java.lang` csomagban definiált ősosztály a `Throwable` objektumai "dobhatók el" (miután a kivétel is objektum).

Az eldobás a `THROW` utasítás segítségével történik.

Két standard alosztálya van:

- `Error` : ide tartoznak a rendszerhibák, ezek nem ellenőrzöttek.
- `Exception` : ellenőrzött kivételek osztálya. Ebből az osztályból származtathat a programozó saját ellenőrzött kivételeket. Természetesen a Java csomagjaiban számos leszármazottja van (standard kivételek).

A throw utasítás alakja (a kivétel-osztály példányosítása):

```
THROW NEW_OPERÁTOR;
```

Ez kivált egy megfelelő típusú kivétel objektumot, és eldobja a kivételt, átadja a JVM-nek.

Példa: Saját kivétel definiálása és kivételkezelés:

```
class VeremMegteltException extends Exception{
     Object utolso;
     public VeremMegteltException( Object o){
          utolsó = o;
     }
     public Object nemFertBele(){           -- ezzel érjük el a tetejét
          return utolso;
     }
}

class Verem{
     final static public int Meret = 100;
     Object tarolo[];
     int mutato = 0;
     :
     public void push (Object o){
          ...
          if ( mutato != Meret ){
              tarolo[mutato++] = o;
          }
          else{
              throw new VeremMegteltException(o); --példányosítás
          } ...
     } ...
}
```

A kivételkezelő szerkezete a Javaban:

```
TRY
     { Utasítások }
            -- ellenőrzött kivételek, amelyek látják a kivételkezelőt
     CATCH (típus változónév ) {Utasítások}
     [CATCH ( típus változónév){ Utasítások}]...
     [FINALLY { Utasítások}]
```

A TRY blokkban elhelyezett utasításokban keletkezett kivételek esetén a JVM a CATCH utasításoknak adja át a vezérlést.

Ha ott talál megfelelő típusú ágat, lefutnak a blokkbeli utasítások, és végrehajtódnak a FINALLY utáni utasítások, és a program folytatódik a FINALLY után.

Ha nincs egyetlen megfelelő kivételkezelő sem, és van FINALLY ág, lefutnak az utasításai, és a kivétel az adott kódrészt (TRY) meghívó módszerhez kerül, vagy beágyazott blokk esetén a tartalmazó blokkhoz.

A FINALLY ág akkor is lefut, ha nem volt kivétel!

A CATCH ág teljesen hiányozhat. A típusegyeztetés miatt a felírás sorrendje nagyon lényeges, ugyanis a CATCH ág az ellenőrzött kivételt elkapja. A hierarchiában lentről fölfele kell elkapni. Módosítsuk ezek alapján a kódot:

```
class Verem{
     final static public int Meret = 100;
     Object tarolo[];
     int mutato = 0;
     :
     public void push (Object o){
          ...
          try{
               if ( mutato != Meret ){
               tarolo[mutato++] = o;
               }
               else {
                    throw new VeremMegteltException(o);
               } ...
          } /*try*/
          catch(VeremMegteltException e){
               System.out.println("A(z) "+e.nemFertBele()+
                      " obektum nem fért el a veremben!");
          }
          catch(Exception e){
               System.out.println("Hiba! Hívja a rendszergazdát!");
          }
          finally {System.out.println("A push lefutott.");
     }...
}
```

SZÁLAK

A párhuzamos programozás eszközei a Javában a szálak. A szálak objektumok, a `Thread` osztályból származnak. Ezek `run()` módszere adja a futtatandó kódot. A `Runnable` interfészt implementáló osztállyal is megadhatunk szálakat. Ebben a `run()` módszert is implementálni kell. Itt is ez adja a kódot. A `Thread` osztály a `Runnable` egy implementációja.

A szálak a Javaban a következő állapotban lehetnek:

- új
- futtatásra kész
- fut
- várakozik
- halott

Új szálat (mint objektumot) a

```
new
```

operátorral hozunk létre, ekkor csak létrejön. Él a szál, de semmi több, semmi aktivitást nem mutat.

Futásra kész állapotba hozni a `start()` módszerrel lehet. A futásra kész állapot annyit jelent, hogy beáll a sorba. A Java rendszerek általában egyprocesszorosak. A futásra kész szálak közül az ütemező választja ki a futtatandó szálat. Az egyprocesszoros rendszerek indeterminisztikusak! A kiválasztott szál működni kezd, fut (a `run()` módszer indul el).

Egy szál halott állapotú lesz, ha meghívjuk a `stop()` módszerét, vagy a `run()` módszer kódja elfogy. Nem lehet egy halott szálat újra elindítani.

A várakozás a szinkronizáció eszköze a Javában. A szálak szinkroniációja meglehetősen sokszinű. A szinkronizációra a Hoare-féle monitort alkalmazza a Java. Ezzel kapcsolatban a következő eszközök állnak rendelkezésre:

- Egy szál a következő esetekben kerülhet várakozó állapotba:
  - a `sleep(x)` módszer meghívásával, a paraméter ezredmásodpercben értendő. Hatásra az adott szál adott ideig várakozik.
  - a `wait()` módszer meghívásával. Szintén várakozást tudunk előidézni.
  - a `suspend()` módszer felfüggeszti a szál futását
  - I/O művelet befejezésére vár
- Várakozó szál a következőképpen kerülhet futásra kész állapotba:
  - `sleep()` esetén továbbmegy a szál az idő letelte után.
  - `wait()` esetén `notify()` vagy `notifyAll()` módszer meghívásával megszűnik a várakozás
  - `suspend()` módszer esetén a `resume()` módszerrel újraindul a futás
  - ha befejeződött az I/O művelet

Kölcsönös kizárás

Módszer esetén ha a fejben szerepel a `synchronized` módosító, akkor a rendszer az adott módszer futtatását úgy végzi, hogy érvényesüljön a kölcsönös kizárás. Ha blokk előtt szerepel a `synchronized(objektum)` előírás, a megadott objektum zárát helyezi el a blokkra.

A szálak csoportokba szervezhetők, közösen, együtt kezelhetők a `ThreadGroup` osztály segítségével. Az egy csoportba tartozó szálakat egyszerre vezérelhetjük a `suspend()`, `resume()`, `stop()` módszerekkel.

A Javaban vannak démonszálak. Ezek végszinkronizációs eszközök, akkor fejezik be működésüket, ha az összes nem-démonszál befejeződött.

A `join()` módszer is a szinkronizációt szolgálja. Hatására egy másik szál hívható meg úgy, hogy a hívó szál megvárja, míg lefut a hívott szál. Ez időzíthető is.

Kétfajta programról szokás beszélni:

- Alkalmazás: A Java rendszerben saját osztályokat definiálunk, és mószereket hívogatunk.
- Appletek: programkák.

Appletek (programkák)

HTML oldalba ágyazható Java programok. Végrehajtásukat egy böngésző program végzi (Netscape vagy Explorer) esetleg egy segédprogram. Az appletek letölthetők és futtathatók.
A `java.applet` csomag tartalmazza a szükséges interfészeket és osztályokat.

További eszközök

A `java.net` osztály eszközei arra szolgálnak, hogy hálózatos kommunikációt megvalósító programokat tudjunk írni. Egy alkalmazást olyan komponensekre tudunk bontani, amely komponensek a hálózat különböző pontjain futnak. Ehhez a Java a távoli objektumok kezelését nyújtja (remote object). A távoli objektumok módszereit egy másik gép el tudja érni. A Java protokolja (eszközrendszere) az RMI (Remote Method Invocation). A távoli objektumok módszereit interfészekben kell rögzíteni, specifikálni, és a JVM felépít egy csonkobjektumot, amely csonkobjektum képes felépíteni a kapcsolatot a távoli objektummal: meghívja a távoli objektum módszereit, azaz megszóllítja a másik JVM-et. Ott lefut a meghívott módszer, és a csonk visszaközvetíti a visszatérési értéket.
A `java.sql` csomag adatbázis programozást tesz lehetővé. Relációs adatbázis kezelést valósít meg. Az adatbázisok elérését a Java a JDBC protokolon keresztül teszi lehetővé. Ez a JDBC egy adatbázis kezelő programozói interfész. A következő szolgáltatásokat nyújtja:

- összekapcsolódás egy relációs adatábzis-kezelővel
- SQL utasításokat tudunk felolgozni.

Kliens – szerver architektúrát használunk, ahol a szerver rész a relációs adatbázis kezelő, amit megszólítunk, és a kliens az általában megírt program. Ennek van egy csomó nyűgje, ezt hívják kétrétegű architektúrának. Azonban jelen pillanatban divat a többrétegű architektúra. A program és az adatbáziskezelő közé jön egy középső réteg:

middleware. A JDBC-n keresztül a midleware-t érem el. Ez a középső rétag a kommunikációt szolgálja. Mindkét végpont ezt a középső réteget szólítja meg, és a középső réteg szolgáltat információt a két végnek.

4. EIFFEL

Az Eiffel egy olyan nyelv, amelyet teljesen az objektumorientált paradigma alapján hoztak létre Bertrand Meyer vezetésével az 1980-as évek második felében. Az Eiffel tehát tiszta OO nyelv, az egységesség elvét azonban nem vallja. Az Eiffelnél is igaz, hogy a nyelv elválaszthatatlan a fejlesztői környezettől, azzal egységes egészet alkot.

4.1. Lexikális elemek

Az Eiffel **karakterkészlete** az US-ASCII szabványon alapszik, tehát betű alatt az angol ABC betűit kell érteni. Az Eiffel a kis és nagybetűket nem különbözteti meg.

Az Eiffelben a **megjegyzés** a jelkombinációtól a sor végéig tart. Az Eiffel beszél **szabad** és **elvárt** megjegyzésről. A szabad megjegyzés a program szövegében bárhol elhelyezhető, szintaktikai jelentése nincs. Az elvárt megjegyzésnek szintaktikai jelentése van, bizonyos konstrukciók (l. …) elemeként jelenhet meg.

Az Eiffelben általános elhatároló jelek a szóköz, tabulátor és sorvége jelek. Értelmez speciális és többkarakteres szimbólumokat (pl.), kötött szintaktikai jelentéssel, ezek egy részét a későbbiekben tárgyaljuk.

Az Eiffel a **foglalt szavak** két csoportját különbözteti meg, ezek a **kulcssorok** és az **előredefiniált nevek**. A kulcsszavak a nyelvi konstrukciókhoz tartoznak, az előredefiniált nevek a szövegben ott fordulhatnak elő, ahol egy változó neve (pl. *Result*), vagy egy típusnév (pl. *INTEGER*).

Az Eiffel kulcsszavai a következők:

Az Eiffel kódolási ajánlás szerint a kulcsszavakat kisbetűs félkövér alakban, a típusok nevét nagybetűs dőlt alakban, az előredefiniált egyedek nevét nagy kezdőbetűs dőlt alakban írjuk.

Az Eiffelben az **azonosító** betűvel kezdődik és betűvel, számjeggyel vagy aláhúzás (_) jellel folytatódhat. Hosszkorlátozás nincs.

Az Eiffel konstansai (**literáljai**) a következők:

Egész konstans: [előjel]számjegy[számjegy]…

Például: -3, 0, +222.

Valós konstans: [előjel] {[számjegy{számjegy]…].

számjegy[számjegy]…|

számjegy[számjegy]…[számjegy

[számjegy]…]} [E/e] egész literál]

Például: -1, 0., .0, -12E_12, 36.28E3.

Bit konstans: bit[bit]…[B/b]

Például: 011001B.

Karakter konstans: `karakter`

ahol karakter vagy egy látható karakter (pl. `a`, vagy a %karakter alakú, ahol a karakter1 jelentése speciális (pl. %N-ú; sor, %-aposztróf, %B-backspace), vagy %/kód/, ahol kód egy előjel nélküli **egész**, az ASCII decimális belső kódot jelenti (pl. %/91/-[).

4.2. Típusok

Az Eiffel egy szigorúan típusos nyelv.

A nyelvben minden programozói eszközt valamilyen típussal deklarálni kell. A típus általában egy osztály (pontosan l. 4.11), amelynek módszerei meghatározzák a típus példányszám végezhető műveleteket.

A típus lehet **referencia** vagy **kiterjesztett** típus. A referencia típusú eszközök értékül egy objektumhivatkozást, a kiterjesztett típusúak magát az objektumot vehetik föl. Az objektum viszont mindig a típus példányaiként jön létre.

A kiterjesztett típusok egy igen fontos csoportját képezik az ún. **alaptípusok**, ezek az *INTEGER, REAL, DOUBLE, CHARACTER, BOOLEAN, BIT* és *POINTER*. Ezen osztályok példányai atomiak.

A POINTER típusnak külső (nem Eiffelben megírt) rutinok számára adható át a programbeli eszközök címe.

Az *INTEGER, REAL, DOUBLE* a *COMPARABLE* és a *NUMERIC* absztrakt osztályok alosztályai. A *CHARACTER* a *COMPARABLE* alosztálya.

A *COMPARABLE* példányai összehasonlíthatóak. Módszerei a szokásos hasonlítási műveletek.

A *NUMERIC* műveletei az összeadás, kivonás, szorzás, osztás infix, és a pozitív és negatív előjel, mint prefix műveletek. A módszerek formális paraméterei és visszatérési értékük *NUMERIC* típusú. Az öröklődés során a módszereket az alaposztályok implementálják, a formális paramétereket és a visszatérési típust újradeklarálják, a szükséges specifikus módszereket megadják.

Ennek köszönhetően az eljárásorientált nyelveknél megszokott módon, az aritmetikai kifejezések vegyes típusúak lehetnek az Eiffelben és az operátorok megszokott infix alakját használhatjuk.

Itt jegyezzük meg, hogy az egész és valós konstansok az Eiffelben valójában kifejezések, az előjel operátort alkalmazzák az előjel nélküli egészekre és valósokra.

A *CHARACTER* osztály példányainak attribútuma a … (amely egy pozitív egész értékű) és a reprezentáló bitsorozat.

A *BOOLEAN* osztály módszerei a logikai és (rövidzár és teljes), vagy (rövidzár és teljes), tagadás, kizáró vagy, implikáció műveleteket realizálják. A két logikai értéket a beépített nevesített konstansként kezelhető **true** és **false** attribútumok képviselik. Ezek neve kulcsszó.

A *BOOLEAN, CHARACTER, INTEGER, REAL, DOUBLE* kiterjesztett osztályok rendre a *BOOLEAN_REF, CHARACTER_REF, INTEGER_REF, REAL_REF, DOUBLE_REF* referencia osztályok alosztályai. A *_REF* osztályok teszik lehetővé, hogy az atomi értékek, mint objektumok, referenciával elérhetők legyenek.

Az alaptípusokhoz tartozik a fix hosszúságú bitsorozatok kezelését lehetővé tevő *BIT* típus, amely kiterjesztett típus deklarációnál a kezelendő bitek számát meg kell adni. BIT *N* alakban, ahol *N* egy előjel nélküli egész vagy egy ilyen értékű nevesített konstans. Műveletei a bitenkénti logikai műveletek, az eltolás **és** a rotáció.

Az Eiffel öröklődési hierarchiája gráf, kitüntetett szerepű az *ANY* osztály. Az Eiffelben minden osztály ennek leszármazottja.

4.3. Változó

Az Eiffelben létezik változó, az eljárásorientált értelemben. A változó egy osztály attribútuma vagy pedig egy rutin lokális változója lehet. Értéke a típustól függően egy referencia, vagy egy objektum.

4.4. Kifejezések

Az Eiffel kifejezésfogalma hasonlít az eljárásorientált nyelvek kifejezésfogalmára. A kerek zárójelek használata ugyanaz. Operandus lehet konstans, attribútum, függvényhívás, tömb. Az Eiffel … operátorai prefixek, bináris operátorai infixek. A preferenciatáblázat a következőképpen néz ki:

.	←
old strip not + –	←
^	←
*** / // **	→
= /= < > <= >=	→
and and then	→
or or else xor	→
implies	→
<< >>	→
;	→

Az egyes operátorok jelentése:

.	a minősítés operátora (l.)
Old, strip	(l.)
not	logikai tagadás
+, –	előjelek
^	hatványozás
*****	szorzás
/	osztás
//	egészosztás
****	maradékképzés
+	összeadás
–	kivonás
=, \=	az *any* osztály egyenlőségvizsgáló műveletei. *e* = *f* azonos referenciatípusok esetén azonosságbeli (ugyanazt az objektumot hivatkozzák), azonos kiterjesztett típusok esetén értékbeli egyenlőséget (a két objektum állapota azonos) vizsgál. különböző típusok esetén először konverzió megy végbe (l.).
<, >, <=, >=	A *COMPARABLE* osztály hasonlító műveletei
and	teljes kiértékelésű logikai és
and then	rövidzár kiértékelésű logikai vagy
or	teljes kiértékelésű logikai vagy
or else	rövidzár kiértékelésű logikai vagy
xor	teljes kiértékelésű logikai kizáró vagy
implies	rövidzár kiértékelésű logikai implikáció
<< >>	tömboperátorok
;	l.

A kifejezések kiértékelése balról jobbra a precedenciatáblázat figyelembevételével történik. A kiértékelésnél végbemenő típuskonverziókat l.

4.5. Végrehajtható utasítások

Az Eiffelben az algoritmusok kódolására végrehajtható utasításokat használunk. Az értékadó utasítás általános alakját l. … A következőkben a vezérlési szerkezetet realizáló utasításokat tárgyaljuk.

4.5.1. Összetett utasítás

Alakja: [utasítás][;]utasítás]…]

Az összetett utasításnál a pontosvessző opcionális elhatárolójelként szerepel, kiírni csak akkor kell, ha a követő utasítás zárójellel kezdődik. Az Eiffel konverzió a szerepeltetését javasolja. Az utasítások szekvenciálisan, a fölírás sorrendjében kerülnek végrehajtásra.

4.5.2. Üres utasítás

Az Eiffelben nincs külön alapszava, tisztán szintaktikai, programozástechnikai jelentősége van.

4.5.3. Feltételes utasítás

Alakja:

IF *feltétel* THEN *összetett-utasítás*
[ELSIF *feltétel* THEN *összetett-utasítás*]…
[ELSE *összetett-utasítás*]
END

Szemantikája a szokásos eljárásorientált szemantika.

4.5.4. Többszörös elágaztató utasítás

Alakja:

INSPECT *kifejezés*
WHEN {*konstans* | *nevesített_konstans* | *intervallum*}
[,{*konstans* | *nevesített_konstans* | *intervallum*}]…
THEN *összetett_utasítás*
[-„-]…
[ELSE *összetett_utasítás*]
END
Intervallum: {*karakter_konstans* .. *karakter_konstans* |
egész_konstans .. *egész_konstans*}

A *kifejezés*, *konstans* és *nevesített_konstans* típusa egész vagy karakteres lehet és a típusuknak (beleértve az *intervallum* típusát is) meg kell egyezniük. A WHEN-ágakban szereplő értékeknek különbözniük kell. A *kifejezés* minden lehetséges értékére elő kell írni valamilyen tevékenységet.

Szemantikája a következő: kiértékelődik a kifejezés, az értéke a felírás sorrendjében összehasonlításra kerül a WHEN-ágak értékeivel. Ha van egyezés, akkor végrehajtódik a megfelelő THEN utáni összetett utasítás, és a vezérlés átadódik a következő utasításra. Ha egyetlen WHEN-ágban sincs megfelelő érték és van ELSE-ág, akkor az abban megadott összetett utasítás hajtódik

végre és a vezérlés átadódik a következő utasításra, ha nincs ELSE-ág, akkor pedig egy kivétel váltódik ki.

4.5.5. Ciklus utasítás

Alakja:

```
FROM    [összetett_utasítás]
        [ciklus_invariáns]
        [ciklus_variáns]
UNTIL   feltétel
LOOP    összetett_utasítás
END
```

A FROM utáni összetett utasítás a ciklus inicializáló része. A *feltétel* végfeltételként működik. A LOOP utáni összetett utasítás a ciklus magja. A ciklus invariáns és variáns rész magyarázatát l. …

4.5.6. Feltételes futtatás

A DEBUG-utasítás lehetőséget biztosít arra, hogy az Eiffel-környezet *debug* opciójának állapotától függően egy kódrészletet lefuttassunk vagy ne futtassunk le.

Alakja:

DEBUG *összetett_utasítás* END

Ha a *debug* opció be van kapcsolva, akkor az összetett utasítás lefut, ha ki van kapcsolva (ez az alapértelmezés), akkor nem.

4.6. Egy Eiffel program felépítése

Az Eiffel program legkisebb önálló része az osztály. A klaszter az összetartozó osztályok együttese. Az *univerzum* klaszterek olyan együttese, amelyekből egy Eiffel alkalmazás elkészíthető és egy hatásköri egység. Végül a *rendszer* osztályoknak egy futtatható, végrehajtható egysége, amelynek van egy kitüntetett (*gyökér*) osztálya. Az összes többi osztály leszármazottja, vagy kliense a gyökérnek. A rendszer futtatása a gyökér osztály példányosításával történik.

A fentiek közül csak az osztály az, amely közvetlen nyelvi elemekkel kezelhető. A klaszter, univerzum, rendszer kezelésének feladata a környezet dolga, így ezekkel itt a továbbiakban nem foglalkozunk.

4.7. Osztályok létrehozása

Egy saját osztály definiálásának általános alakja a következő:

[INDEXING *index_lista*]
[DEFERRED | EXPANDED]
CLASS *név*
[*formális_generikus_lista*]
[OBSOLETE *sztring*]
[*öröklődés*]
[*konstruktorok*]
[*eszközdeklaráció*]
[INVARIANT *invariáns*]
END

A fenti sorrend kötött.

Az INDEXING-résznek nincs közvetlen szemantikai hatása az osztályra. Az osztály kísérő, dokumentációs információit (szerző, dátum, javallott felhasználás, stb.) lehet itt megadni ahhoz, hogy egy archiváló eszközt használva az osztály a tulajdonságai alapján is tárolható és visszakereshető legyen.

Az *index_lista* szerkezete:

index_bejegyzés [;*index_bejegyzés*]...

ahol az *index_bejegyzés* alakja:

[*azonosító*:]{*azonosító* | *konstans*}

[,{*azonosító* | *konstans*}]…

Például:

Indexing
absztrakt_adatszerkezet, keresofa, piros_fekete_fa;
szerzo:“Kiss Antal“;
irodalom:”Rivert, Leiserson:Algoritmusok”
“Meyer: Eiffel, The Language”;
keszult: 2003, augusztus, 31;
utolso_modositas: 2003, december, 10

A DEFERRED kulcsszó megadásával absztrakt osztályt tudunk létrehozni. A nem absztrakt osztályt az Eiffel *effektív* osztálynak hívja.

Egy osztály absztrakt, ha legalább egy eszköze absztrakt- Ha szerepel a DEFERRED, akkor ennek kötelezően fenn kell állnia.

Az EXPANDED megadása esetén egy kiterjesztett osztály jön létre, ennek hiányában egy referencia típus keletkezik.

Egy osztályt mindig meg kell nevezni, a névnek egy univerzumon belül egyedinek kell lenni.

Ha szerepel a *formális_generikus_lista*, akkor egy *generikus* (*paremetrizált*) osztály jön létre, ha nem szerepel, akkor egy *nem_generikus*.

A *formális_generikus_lista* alakja:

[*azonosító*[,*azonosító*]…→*típus*
[,*azonosító*[,*azonosító*]…→*típus*]…]

A generikus osztály felhasználása esetén a deklarációban aktuális generikus listát kell megadni. Az aktuális generikus lista elemei számban és sorrendben megegyező olyan osztálynevek lehetnek, amelynek a formális generikus listán szereplő típusok leszármazottai.

A OBSOLETE-rész szerepeltetése arra szolgál, hogy jelezzük az osztály egy korábbi Eiffel verzióban készült. Ekkor az osztályra való hivatkozás esetén egy olyan figyelmeztető üzenetet kapunk, amely tartalmazza a *sztringet*.

Egy osztály *eszközei attribútumok* és *rutinok* lehetnek. Ez utóbbiak a módszerek. Az attribútum vagy *változó*, vagy *nevesített konstans*, a rutin *eljárás* vagy *függvény*.

Az eszközök az Eiffelben *példányszintűek*. Az egyes példányokban az attribútumok *mezőkben* jelennek meg.

Az *eszközdeklaráció* alakja:

FEATURE *eszközök* [FEATURE *eszközök*]…

ahol az *eszközök*:

[*klienslista*]
[*megjegyzés*]
eszközdeklaráció[[;]*eszközdeklaráció*]…

A *klienslista* szerkezete:

{*osztálynév*[,*osztálynév*]…}

Itt az ún. *export státust* adjuk meg, vagyis felsoroljuk azon osztályok nevét, amelyek az *eszközdeklarációban* megadott eszközöket látják. Ha nem adunk meg osztályneveket, akkor publikusak az eszközök. Ha itt a saját osztálynév áll, akkor pedig privátak. Tehát a láthatóságot az Eiffel *explicit* módin szabályozza. Az öröklődésnél az eszközök automatikusan átkerülnek az alosztály hatáskörébe.

A *megjegyzés* egy elvárt megjegyzés.

A nevesített konstans attribútum deklarációja a következő módon történik.

[FROZEN] *név*[,[FROZEN] *név*]…:*típus*
IS *konstans*

A FROZEN megadásával olyan eszközt hozunk létre az osztályban, amely nem felüldefiniálható az alosztályban.

A nevesített konstans attribútum esetén a példányok megfelelő mezői mindig a *konstans* értékét tartalmazzák.

A változó attribútum deklarációja:

[FROZEN]*név*[,[FROZEN]*név*]…:*típus*

Tehát nem adható kezdőérték.

Egy rutin deklarációja a következőképpen néz ki:

[FROZEN]*név*[,[FROZEN]*név*]…
[(*formális_paraméter_lista*)][:*típus*]
IS *rutin_le*írás

Ha szerepel a *típus*, akkor függvényről, egyébként eljárásról van szó. Egy implementációhoz több név is megadható, ezek szinonimák.

A *formális_paraméter_lista* alakja:

név[,*név*]…:*típus*[;*név*[,*név*]…:*típus*]…

A *rutin_leírás* felépítése a következő:

[OBSOLETE sztring]
[megjegyzés]
[előfeltétel]
[lokális deklarációk]
törzs
[utófeltétel]
[kivételkezelő]
END

Az OBSOLETE szerepe ugyanaz, mint az osztálynál volt. A rutint hívó kap egy *sztringet* tartalmazó figyelmeztető üzenetet.

A *megjegyzés* egy elvárt megjegyzés.

A *lokális_deklarációk* alakja:

LOCAL *név*[,*név*]…:*típus*
[[;]*név*[,*név*]…:*típus*]…

Itt lényegében a rutin lokális változóit deklaráljuk. Ezeknek a láthatósága statikus, élettartamuk dinamikus.

A *törzs* szerkezete az alábbi:

{DEFERRED | EXTERNAL"*nyelv*" |
{DO | ONCE} *összetett_utasítás*}

A DEFERRED kulcsszó azt jelzi, hogy a rutin absztrakt, nincs implementálva. Ekkor a tartalmazó osztály is szükségszerűen absztrakt.

Az EXTERNAL után a *nyelv* azt a programnyelvet adja meg, amelyen a rutint implementáltuk, ezáltal egy *külső* rutint meghatározva. Külső rutinoknál nem szerepelhetnek lokális változók és kivételkezelő.

Ha a törzs a PO kulcsszóval indul, akkor az összetett utasítás minden hívásnál lefut, ONCE esetén viszont egy adott objektumra csak a munkamenet első hívásakor fut le. A korábbi hívások hatástalanok. Ha függvényről van szó, a visszatérési érték mindig az első hívás visszatérési értéke lesz.

Függvény esetén létezik egy speciális, előredefiniált egyed (l. …), a *Result*, ez hordozza a visszatérési értéket. A törzsben, vagy az utófeltételben kell neki értéket adni.

Egy rutin hívásánál az Eiffel a paraméterkiértékelésnél sorrendi kötést, számbeli egyeztetést (ez alól kivétel, ha tömböt alkalmazunk – l. …), típusegyeztetést (l. …) alkalmaz. A paraméterátadás érték szerinti. Ezalól kivétel a POINTER típus alkalmazása, amivel egy eszköz címét tudjuk átadni egy külső rutinnak.

Az *előfeltétel* és *utófeltétel* szerepét l. …, a *kivételkezelő* pedig …-ban szerepel.

Egy osztálybeli eszköz hivatkozható a rutinok törzséből, elő- és utófeltételéből és és kivételkezelőjéből.

Az *öröklődés* formája:

INHERIT *szuperosztály_név*[*eszközadaptáció*]
[[;]*szuperosztály_név*[*eszközadaptáció*]]…

Az Eiffelben többszörös öröklődés van. Az Eiffel eszközt ad a névütközések kezelésére.

Ha az *öröklődés* hiányzik egy implicit

inherit *ANY*

rész épül be az osztálydefinícióba.

Az *eszközadaptáció* alakja:

[*átnevezés]*
[*export*]
[*érvénytelenítés*]
[*újradefiniálás*]
[*szelekció*]
END

A sorrend kötött.

Az *átnevezés* formája:

RENAME *örökölt_eszköznév* AS *új_eszkönév*
[,*örökölt_eszköznév* AS *új_eszköznév*]…

Bármely örökölt eszközt az alosztály átnevezhet. Ez szolgál a többszörös öröklődésből származó névütközések feloldására, illetve arra, hogy az osztály az átvett eszközöket a saját környezetének megfelelő néven adhassa tovább az öröklődésnek.

Az *export* az átvett eszközök láthatóságának újraszabályozására való, alakja:

EXPORT {*kliens*[, *kliens*]…} ALL | *eszközök*}
[,{*kliens*[,*kliens*]…}{ALL | *eszközök*}

A *kliens* az univerzum egy osztályának a neve, az eszközlista az örökölt eszközök neveit tartalmazza, vesszővel elválasztva. A felsorolt eszközök vagy minden eszköz (ALL) új export-státusát adja meg.

Az *érvénytelenítés* örökölt effektív rutinok absztrakttá (**deferred**) minősítését, tehát az implementáció érvénytelenítését teszi lehetővé. Alakja:

UNDEFINE *rutinnév_lista*

Az újradefiniálás egy változó vagy rutin nevének újradefiniálását jelenti. Rutin esetén megváltozhat a specifikáció és az implementáció is. Itt csak jelezzük az újradefiniálás tényét, az eszközdeklarációban kell megadni a tényleges új definíciót. Egy absztrakt módszer implementálásánál nem kell a nevét újradefiniálni, hiszen ilyenkor a ,,definiálás” ebben az osztályban történik meg.

Az *újradefiniálás* alakja:

REDEFINE *eszköznév_lista*

szelekció alakja:

SELECT *eszköznév_lista*

A *konstruktorok* az osztály konstruktorait határozza meg. Csak effektív osztályban lehet konstruktorokat létrehozni. Alakja:

CREATION *konstruktor* [CREATION *konstruktor*]...

ahol a *konstruktor* formája:

[kliensek][megjegyzés]*eljárásnév_lista*

A *kliensek* azon osztályok nevét tartalmazza, amelyek számára a konstruktorok exportálódnak.

A *megjegyzés* egy elvárt megjegyzés.

A konstruktorok eljárások, ezeket az eszközdeklarációban kell megadni.

Az INVARIANT-rész magyarázatát l. …

4.8. Objektum, érték, egyed

Egy Eiffel program futás közben *objektumokat* tud létrehozni és kezelni. Az Eiffel beszél *standard* és *speciális* objektumokról. A speciális objektum nem más mint egy adott típussal kompatibilis értékek sorozata. Két fajtája van, a *sztring* és a *tömb* (l. …). A sztring értékei karakterek, a tömbben pedig vagy referenciák vagy egy egyszerű típus példányai helyezkednek el.

Egy standard objektum *példányosítással* vagy *klónozással* jön létre. Két fajtája az *alap* és a *komplex* objektum. Az alap objektum az alaptípusok példánya. A komplex objektum egy nem alaptípusos osztályának példánya, az adott osztály attribútumai által meghatározott, kötött számú (ez lehet nulla is!) mezőből áll.

Egy *érték* lehet objektum vagy referencia.

A referencia vagy *void* (érvénytelen) vagy *csatoló* (érvényes) referencia. Az érvénytelen referencia segítségével nem érhető el semmiféle további információ. Azt, hogy egy referencia érvényes-e az *ANY* osztály *Void* attribútumához való hasonlítással dönthetjük el.

Az érvényes referencia mindig egy konkrét objektumot, a *csatolt* objektumot hivatkozza.

Egy objektumot vagy annak mezőit kifejezések segítségével tudjuk kezelni az Eiffelben. A kifejezés legegyszerűbb formáját jelentik az egyedek. Egy *egyed* egy olyan *név*, amellyel egy adott osztály példányainak az értékeit tudjuk elérni.

Az egyed lehet:

- egy osztály attribútuma,
- egy rutin lokális változója beleértve a *Result* előredefiniált egyedet a függvények esetén,
- rutin formális paramétere,
- a *Current* előredefiniált egyed, amely az aktuális példányt hivatkozza.

A lokális egyedek és az attribútumok *írhatóak* (értékük megváltoztatható), a formális paraméterek és a *Current* csak olvasható (értékük nem változtatható meg).

Az írható egyedek értéke megadható, illetve megváltoztatható *értékadással*, illetve *példányosítással*.

Az *értékadó utasítás* alakja:

egyed := *kifejezés*

4.9. Példányosítás

A példányosítással létrejön egy új objektum, alapállapotba kerül és egy írható egyed referencia értéke beállítódik. A példányosító utasítás alakja:

! [*típus*] ! *írható_egyed* [.*konstruktorhívás*]

A konstruktorhívás csak akkor maradhat el, ha nincs konstruktora az osztálynak. Ekkor a mezők alapértelmezett értéket kapnak (… nullázódnak).

A *típus* akkor adandó meg, ha az az *írható_egyed* deklaráció típusának egy leszármazottja, és ennek konstruktorával akarunk példányosítani.

Az objektumok megszüntetésére az Eiffel egy automatikus szemétgyűjtögetőt alkalmaz. Ennek megvalósítása implementáció függő.

4.10. Objektumok duplikálása és összehasonlítása

Futás közben egy új objektum létrehozásának alapvető eszköze a példányosítás. Néha viszont szükség lehet arra, hogy egy már létező objektum tartalmát másoljuk át egy másik, már létező objektumba. A **másolás** lehet *sekély*, amikor csak egyetlen objektumot másolunk és *mély*, amikor a hivatkozott objektumokat is másoljuk.

A másolás speciális esete a **klónozás**, amikor egy adott objektumot duplikálunk, és így keletkezik egy új objektum. Ez is lehet sekély és mély.

Kapcsolódó probléma az objektumok összehasonlításának kérdése. Itt is beszélhetünk sekély és mély egyenlőségről.

Azok az eszközök, amelyek megvalósítják a fentieket, az *ANY* osztályban találhatók.

Az *y* egyed által hivatkozott objektum másolása *x*-be a következő módon történhet:

```
X.COPY(Y)
```

Ekkor az *y* által hivatkozott objektum minden mezője átmásolódik az *x* által hivatkozott objektum mezőibe. Ha a mezőben referencia van, akkor csak az másolódik át, tehát a mező által hivatkozott objektum nem. A másolás előtt mind *y*-nak, mind *x*-nek rendelkeznie kell csatolt objektummal.

A klónozás esetén egy új objektum keletkezik, az érvénytelen referencia is klónozható. Általában értékadásnál használjuk

```
X:=CLONE(Y)
```

alakban. *Y* típusának *x* valamely leszármazott típusának kell lennie.

A mély másolás és klónozás a `DEEP_COPY` és `DEEP_CLONE` rutinokkal történhet.

Annak meghatározására, hogy az *X* és *Y* egyedek által hivatkozott objektumok mezőről mezőre megegyeznek-e, az

```
EQUAL(X,Y)
```

logikai visszatérési értékkel rendelkező rutin használható. A mély egyenlőségvizsgálat rutinjának neve: `DEEP_EQUAL`

4.11. Típusok kezelése

Az Eiffelben egy **típus** a következő konstrukciókban fordulhat elő:

– függvény visszatérési típusa
– rutin paramétereinek típusa
– rutin lokális egyedének típusa
– szuperosztály
– formális generikus paraméter típusa
– aktuális generikus paraméter
– példányosítás.

Egy típus általánosságban a következő lehet:

– osztály
– kiterjesztett osztály
– átvett típus
– bit típus formális generikus paraméter

Nem minden típus szerepelhet minden konstrukcióban, a részleteket az egyes konstrukciók tárgyalásánál láthatjuk.

Az osztályról (beleértve a generikus osztályt is) és a bit típusról már volt szó. Egy kiterjesztett osztály az

EXPANDED *osztálynév*

segítségével keletkezik. Például

X : **expanded** *y*

Itt *y* egy kiterjesztett vagy referencia osztály neve.

Az **átvett típus** arra szolgál, hogy egy már ismert egyed típusát használhassuk fel a megadott konstrukcióban. Formája:

LIKE *egyed*

Az Eiffel öröklődési gráfjának van ,,kezdete” és ,,vége”. Azt már láttuk, hogy az Eiffelben az ősosztály az *ANY*, azonban a teljes hierarchiában fölötte még van két olyan osztály, amely platformfüggő eszközöket tartalmazza. Az *ANY* szuperosztálya a *PLATFORM* és az ő szuperosztálya a *GENERAL*, amelynek már nincs szuperosztálya.

A *GENERAL* a platformfüggő általános eszközök (pl. a *clone*) osztálya. A *PLATFORM* nevesített konstans attribútumokat vezet be a platformfüggő ábrázolásokhoz. Az ANY már csak platformfüggetlen eszközöket tartalmaz.

A hierarchia alján helyezkedik el a *NONE* osztály, amely **minden** osztálynak leszármazottja. Egy virtuális osztálynak tekinthető, amelynek nincs konkrét forrásszövege (hiszen azt minden új osztály létrehozása esetén újra kellene írni), amely az osztályhierarchia teljessé tételéhez szükséges. Természetesen egyetlen példánya sem létezik és nem lehet alosztálya. Egyetlen eszköze sem hivatkozható.

Az *ANY* osztály *Void* eszköze *NINE* típusú. Minden T típus közvetve vagy közvetlenül egy osztályból származik, ezt a típus **alaposztályának** hívjuk.

Ha T egy osztály, akkor a származtatás közvetlen, T vagy egy nemgenerikus osztály neve, vagy egy generikus osztálynév, aktuális generikus paraméterekkel. tehát itt az alaposztály T.

A többi esetben a származtatás indirekt. Ekkor a felhasznált (kiterjesztett, átvett, bit) típust **bázistípusnak** nevezzük és a bázistípus alaposztálya lesz T alaposztálya.

Az Eiffeilben is a típusegyenértékűség az osztályhierarchián alapul. Általánosságban azt mondhatjuk, hogy egy V típus egyenértékű a T típussal, ha

1. V alaposztálya leszármazottja T alaposztálynak.
2. Ha V generikusa származtatott, akkor aktuális generikus paraméterei típusegyenértékűek T-évél.
3. Ha T kiterjesztett, akkor V maga T, vagy T alaptípusa.

A pontos esetek tárgyalása ennél sokkal finomabban történhet, de ez meghaladja jelen jegyzet kereteit.

A típusegyenértékűség a kifejezésekben és az értékadásnál játszik alapvető szerepet az Eiffelben.

4.12. Sztringek és tömbök

A tömb értékek egy homogén sorozata, amely elemeit egész indexértékeken keresztül érhetjük el. A sztring egy specifikációs tömb, melynek értékei karakterek. Az *ARRAY* és a *STRING* osztályok eszközeivel kezelhetjük őket. Ezek egyike sem kiterjesztett, tehát a tömb és sztring objektumok mindig referenciával hivatkozhatók.

Az *ARRAY* egy generikus típus, paramétere mindig az értékek közös típusát adja. Lényegében egydimenziós dinamikus tömböt kezel. Többdimenziós tömböt úgy tudunk létrehozni, hogy az aktuális generikus típus tömb.

Az indexhatárokat a *make* rutin paramétereiként adhatjuk meg, amely konstruktorként van megírva.

Átméretezhetünk egy tömböt a resize(alsó*_határ, felső_határ*) rutin segítségével.

Tömbelemet az *item*(*index*)-el érhetünk el, felülírásra a *put*(*érték,index*) szolgál. A *force*(*érték,index*) rutin esetén, ha *index* nem esik az indexhatárok közé, akkor a tömb kiterjesztődik az adott indexig.

Tömbkonstanst tudunk létrehozni explicit módon a << , >> operátorok segítségével úgy, hogy felsoroljuk a tömb értékeit:

$<< k_1, \ldots k_n >>$

ahol k_i-k kifejezések, amelyek típusegyenértékűek.

A sztring lényegében egy *ARRAY*[*CHARACTER*] típusú tömb, ahol az alsó határ értéke 1. A *make* itt csak a felső határt rögzíti.

Egy sztring konstans alakja:

"[*karakter*]…"

Például: `"Ez egy sztring".`

4.13. Programhelyesség

Az Eiffelben az osztályok és a rutinok szövegében elhelyezhetünk *programhelyességi előírásokat*. Ezek formális specifikációk, amelyek

- automatikus dokumentációs eszközök,
- segítségével a programfejlesztő leírhatja az egyes programelemek tulajdonságait és helyes működését,
- teljesülése futás közben ellenőrizhető és a kivételkezelésen keresztül lehet reagálni a problémákra.

Egy programhelyességi előírás szerepelhet:

- egy rutin elő- és utófeltételében,
- egy osztály invariánsában,
- egy ciklus invariánsában,
- a CHECK-utasításban.

A programhelyességi előírás alakja:

[*címke*:]{*feltétel* | *megjegyzés*}
[;[*címke*:]{*feltétel* | *megjegyzés*}]…

A *megjegyzés* szerepeltetése csak dokumentációs célokat szolgál. A ; az **and then** logikai műveletnek felel meg. A *címke* egy azonosító (szerepét l. …).

Egy rutin elő- illetve utófeltételeinek alakja a következő:

REQUIRE *programhelyességi_előírás*
ENSURE *programhelyességi_előírás*

Az elő- és utófeltételekben az adott osztály eszközeire és a lokális egyedekre lehet hivatkozni. Az előfeltételnek a rutin működésének kezdetekor, a utófeltételnek a működés befejeződésekor teljesülnie kell. Az utófeltételekben szerepelhet az **old** lineáris operátor, amely operandusának a rutinba való belépésénél meglevő értékét adja meg.

Az utófeltételben általában azt határoztuk meg formálisan, hogy milyen változásokat okoz a rutin lefutása. Hasznos lehet viszont az is, hogy beírjuk, mi **nem** változik meg. Erre szolgálhat speciális esetben a **strip** operátor.

Ha *x.y,z*, egy *A* osztály attribútumai és *A* egy rutinjában használjuk a **strip**(*a,b*) kifejezést, akkor tulajdonképpen egy olyan tömböt kapunk, ahol a **strip** után felsorolt mezőkből áll. **strip**( ) az összes mezőt tartalmazó tömböt eredményezi. Ekkor az

equal(**strip**(a,b), **old strip**(a,b))

azt jelenti, hogy a rutin nem változtathatja meg az *a* és *b* attribútum értékét. A *equal* itt két tömb elemről elemre történő egyezőségének vizsgálatára szolgál.

Példa: Egy tetszőleges objektumokat tartalmazó sor absztrakt adatszerkezetet realizáló osztály egy rutinjának elő- és utófeltétele lehet például a következő:

```
put(c:ANY) is
require
        nincs_tele: not tele
do
–   – írás a sorba
ensure
    szamlalo = old szamlalo + 1;
    old ures implies elem = e;
    not ures
end
```

Egy osztály és egy ciklus invariánsának alakja:

INVARIANT *programhelyességi_előírás*

Osztály esetén az invariánsnak az osztály minden példányára teljesülnie kell.

Példa: A lista absztrakt adatszerkezetet realizáló osztály invariánsa.

```
deferred class LISTA
M
invariant
        ures = (szamlalo = 0);
        elso = (pozicio = 1);
        utolso = (not ures and (pozicio = szamlalo));
        kivul = (pozicio = 0) or (pozicio = szamlalo + 1));
        pozicio >= 0;
        pozicio <= szamlalo + 1;
        ures = (pozicio = 0);
        (elso or utolso) implies not ures
end
```

Egy ciklus invariánsainak a ciklus működésének befejezése után kell teljesülnie. A ciklus **variánsa** viszont arra szolgál, hogy a ciklus futása garantáltan befejeződjön. A variáns alakja:

VARIANT [*címke*:] *egész_kifejezés*

Az *egész_kifejezés* értékét a ciklus inicializáló része nemnegatívra kell, hogy állítsa. Ezután a ciklusmag minden lefutásával értéke 1-el csökken. Ha a variáns negatívvá válik, a ciklus befejezi a működését, függetlenül a feltétel értékétől.

Egy rutin törzsében bárhol elhelyezhető a CHECK-utasítás, amellyel egy adott feltétel teljesülését ellenőrizhetjük a program adott pontján. Alakja:

CHECK *programhelyességi_előírás* END

Az Eiffelben a kivételek a ,,szokásos" események, de a programhelyességi előírások megsértése is kivételt vált ki.

Az Eiffelben a kivételek objektumok. Az ős kivételosztály, az *EXCEPTIONS*. A kivételkezelés csak rutinhoz köthető, kisebb egységhez (pl. utasítás, kifejezés) nem.

Egy kivételnek az Eiffelben *neve* és *kódja* van, ezek az *EXCEPTIONS* attribútumai. A beépített kivételek kódja pozitív, a sajátoké negatív. A név a hibaüzenet szerepét játssza, ez egy sztring.

Az Eiffelben egy rutin törzse után helyezhető el a kivételkezelő, amely a következőképpen néz ki:

RESCUE *összetett_utasítás*

A rutinban bekövetkező bármely kivétel hatására a vezérlés erre adódik át.

Az *ANY* osztályban nem egy

default_rescue is
do
end

alapértelmezett kivételkezelő módszer, amelyet minden osztály örököl és átdefiniálhat. Így tehát az Eiffelben minden osztályban van alapértelmezett kivételkezelő. Amennyiben egy rutinban nem adunk meg explicit kivételkezelőt, akkor implicit módon kiegészül egy

rescue
default_rescue

résszel. Tehát az Eiffelben minden rutinban van kivételkezelő.

Csak a kivételkezelőben használható a RETRY-utasítás. Ennek hatására a rutin újraindul, a paraméterátadás és a lokális egyedek inicializálása nélkül. Alakja:

RETRY

Egy felhasználói kivétel kiváltható a következő eljáráshívással:

RAISE (*kód,név*)

A felhasználói kivétel nevét a *developer_exception_name* attribútum, kódját az *exception* attribútum tartalmazza.

Egy rendszerkivétel figyelése letiltható az

IGNORE (*kód*)

eljáráshívással.

Szintén az operációs rendszer által kiváltott kivételekhez kapcsolódóan az Eiffel lehetőséget ad *általános*, rutintól független kivételkezelésre. A

CONTINUE (*kód*)

eljáráshívás után a *kód* kódú kivétel bekövetkeztekor az EXCEPTIONS osztályban megadott (és természetesen bárhol újraimplementálható) üres törzsű CONTINUE_ACTION eljárás hívódik meg. Ennek egyetlen paramétere a *kód*. Az eljárás lefutása után a program a kivétel bekövetkezésének helyén folytatódik.

Az *ignore*, illetve a *continue* meghívása után az alapértelmezett viselkedés visszaállítása a

CATCH (*kód*)

eljáráshívás hatására következik be.

A viselkedésmódot a

`STATUS`(*kód*)

függvény adja meg, melynek visszatérési értéke *comput, continued, ignored* lehet.

A kivételkezelőben a bekövetkezett kivétel kategóriáját az alábbi logikai függvények segítségével kérdezhetjük le:

`IS_ASSERTION_VIOLATION`(*kód*)
`IS_DEVELOPER_EXCEPTION`(*kód*)
`IS_SIGNAL`(*kód*)

Ezek rendre akkor térnek vissza igaz értékkel, ha programhelyességi előírás megsértése, felhasználói kivétel vagy operációs rendszer által kiváltott kivétel következik be.

A kivétel típusát *EXCEPTIONS* különböző, egész típusú, a bekövetkezett kivétel kódját tartalmazó attribútumai segítségével dönthetjük el. Például *Precondition* (előfeltétel megsértése), *No_mor_memory* (elfogyott a memória), *Void_call_target* (érvénytelen referenciára való hivatkozás).

Programhelyességi előírás megsértése esetén az előírásban szereplő, a kivételt okozó feltétel címkéjét tartalmazza a *tag_name* attribútum.

A kivételkezelés alapértelmezett szemantikája az Eiffelben a következő:

Ha egy rutin futása közben valahol bekövetkezik egy kivétel, akkor

- a hátralevő utasítások nem hajtódnak végre,
- elindul a kivételkezelő,

– ha van benne RETRY-utasítás, akkor a rutin újra lefut (természetesen újra bekövetkezhet valamilyen kivétel és akkor ez ismétlődik rekurzívan),

– ha nincs RETRY-utasítás, akkor a kivételkezelő befejezi a működését és a rutin sikertelenül véget ér. Ez a hívó rutinban egy kivételt vált ki és ennek a kivételnek a kezelése történik a beírt módon. Ha nincs hívó rutin, a vezérlés (sikertelen programfutással) visszatér az operációs rendszerhez.

Mi történik ha kivételkezelőben következik be kivétel??

Az Eiffel tehát a sikeres rutinvégrehajtást kényszeríti a programozóra.

Példa: Egy olyan rutin, amely mindig sikeresen ér véget.

– – A *lehetetlen* egy *BOOLEAN* típusú

– – attribútum, alapértéke *false*. Akkor lesz *true*

– – ha egyik meghívott rutin sem fut le sikeresen.

```
mindig sikeres is
    local
        nem_elso:BOOLEAN - - értéke induláskor false
    do
        if not nem_elso then rutin_1
        elso if not lehetetlen then rutin_2
        end
    rescue
        if nem_elso then lehetetlen := true
        end;
        nem?elso := true;
        retry
    end
```

A rutinunk először meghívja a **rutin_1**-et, ha az sikeresen fut le, ő is visszatér sikeresen, ha kivételt okoz (sikertelenül tér vissza), akkor meghívja a **rutin_2**-t. Ha **rutin_2** sikeresen lefut, akkor ő is visszatér sikeresen, egyébként a *lehetetlent* igazra állítja és sikeresen visszatér.

4.15. I/O

Az input-output a *STANDARD_FILES* és a *FILES* osztályok valósítják meg. Az *ANY* osztálynak vannak olyan rutinjai, amelyek bármely objektum standard outputon való megjelenítését lehetővé teszik. Az Eiffel I/O eszközrendszere közepesnek mondható.

5. SMALLTALK

Nem típusos nyelv.

Karakterkészlete

- a szokásos.
- Kis és nagybetű megkülönböztetendő.

Megjegyzés

- ″ ″ között, tetszőleges karaktersorozat.

Elhatároló jelek

- szóköz
- `(`
- `)`
- `[`
- `]`
- `.`
- `|`

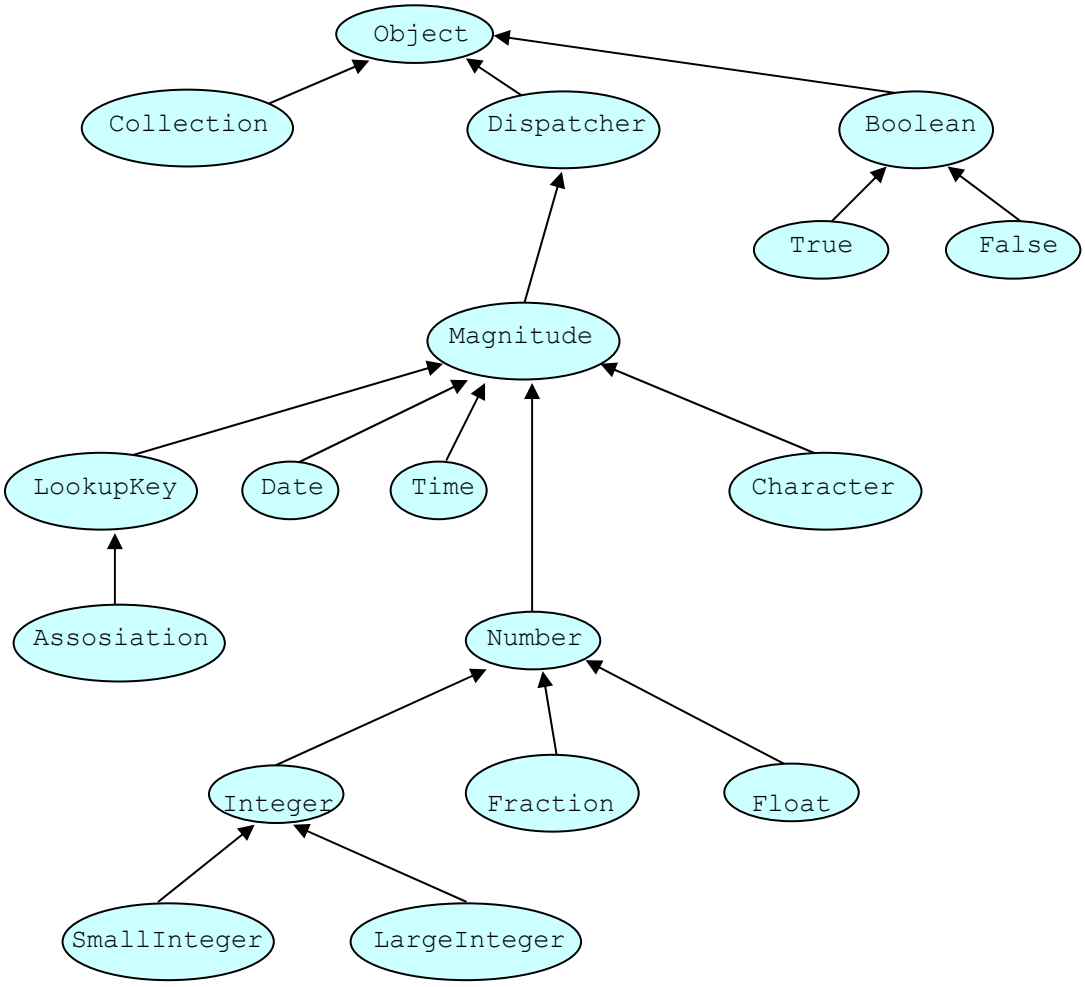
A Smalltalkban minden objektum.

Mint OO nyelv elvei:

- egyszeres öröklés
- késői kötés (más lehetőség nincs)
- üzenetalapú nyelv, alapeszköze az üzenet
- automatikus objektum megsemmisítés: hatékony garbage collection (referencia alapú)
- bonyolult bezárási mechanizmus, láthatóság szabályozása bonyolult
- léteznek példány és osztályszintű elemek
- vannak absztrakt osztályok
- kollekciók léteznek
- nem léteznek `template`-k

– bonyolult standard osztályhierachiája van (fa)

A Smalltalk osztályhierarchia részlete



A `Magnitude`, `Number`, `LookupKey` absztrakt osztályok.

Literálok

A megfelelő osztály példányai.

Írásmód: a belsejében szóköz és egyéb elválasztó karakter nem lehet.

– Számok: egész, tört, lebegőpontos

  Például: `28`, `3/4`, `3.28`

– Létezik a karakter, mint literál.

  Alakja: `$karakter`

  Például: `$c;`

– A sztring, mint literál:

  Alakja: `'tetszőleges karakter sorozat'`

  Például: `'almafa'`

  Létezik egy `String` osztály, ami karakterek egy egydimenziós tömbje.

– Szimbólum: (symbol)

  Alakja: #tetszőleges_karakter_sorozat

  Például: #output

  Speciális jelentése van. Alapvető szerep két szempontból:

  – Mindig egységes és oszthatatlan, bizonyos objektumok neveit kezelhetem bizonyos helyzetekben szimbólumként.
  – A szimbolikus műveletek.

  Atom: Speciális jelentése van.

  Alakja: ##tetszőleges_karakter_sorozat

– Tömb: tetszőleges literálok egydimenziós tömbje, literálként:

  Alakja: `#(tetszőleges literál sorozat)` egymástól szóközzel elválasztva

  Példa:
  ```
  #('egy' 'kettő' 'három' 'négy')
  #(1 2 3 4 5)
  #(1 'kettő' #A)
  #(#1 28.3 $x)
  ```

Változók

Létezik a változó fogalom. Minden változó egy objektumot címez. Van neve, értéke és címe. Nincs típuskomponens (nem is kell).

A változó is objektum, értéke is csak objektum lehet. Speciálisan oldja meg az objektumazonosítást: OID-vel. Mivel minden objektum, a változók nem magát az objektumot tartalmazzák, hanem egy OID-t. Ez egy referencia. Így nem kell a típus, hiszen az OID egyetlen módon van megvalósítva.

Indirekt címzést valósít meg. A változó értékként egy objektum azonosítóját veheti fel. Mutat egy objektumra. Amíg egy változónak nem adunk értéket, addig `NIL`, ahol a `NIL` az UndefinedObject osztály példánya. Az objektumazonosító egységes. Bármilyen osztályt meg tud címezni. (Speciális referencia)

A bezárás szintjei:

- Példányváltozók (ld. Java: objektumok állapotának a leírására szolgálnak).
  - Csak az adott osztály példánymódszerei látják. Az osztálymódszerek nem látják a példányokat.
  - Kisbetűvel kezdődik a nevük.
  - Privát változó.
  - Példányoknál újra elhelyeződik.

- Ideiglenes változók: (temporary) módszerek lokális változói. Kisbetűvel kezdődik a nevük.

- Globális változók: program globális változók.
  - Az adott program minden módszere látja őket.
  - Nagybetűvel kezdődik a nevük.
  - Nyilvános változók.

- Szótárváltozók: a szótár speciális kollekció: táblázat.
  - A Smalltalk lehetővé teszi, hogy bizonyos változók ilyen kollekciókba legyenek szervezve, és őket több osztály lássa. A szótárváltozók bizonyos osztályok módszerei által elérhető változók.
  - Félnyilvános jellegű. Külön eszköz.
  - A korábbi verziók megosztott változóknak hívja.
  - Nagybetűvel kezdődik a nevük.

- Osztályváltozó (ld. Java osztályonként egy példányban létezik). Egy osztályhoz tartozó változó, minden példány ugyanazt a változót látja. Látják a leszármazott osztályok is.
  - Az osztálymódszerek, példánymódszerek látják őket, beleértve a leszármazottakat is. Lentről felfelé látszik.
  - Nagybetűvel kezdődik a neve.

- Osztálypéldány-változó:
  - Egy adott osztályban az osztálymódszerek látják őket, a leszármazottak nem. Nem öröklődik.
  - Kisbetűvel kezdődik a neve.

Terminológia:

- Kisbetűvel kezdődő nevű változók: privát változók, láthatóságuk korlátozott.
- Nagybetűvel kezdődő nevű változók: megosztott változók, egyszerre több osztály látja.

Osztály

Fordítási egység és egyben programegység. (ld. még később)

Blokk

Létezik a módszereken belül programegységként a blokk. (ld. még később)

Módszerek

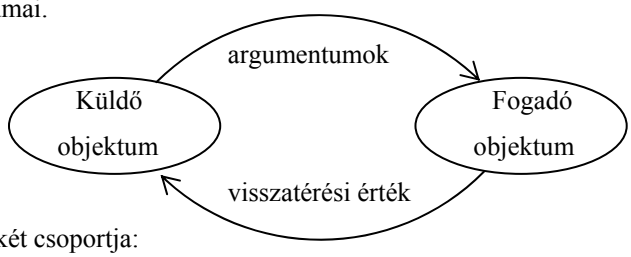
Módszerek nem ágyazhatóak egymásba, blokkok igen. Blokkok módszerekben fordulhatnak elő. (ld. még később)

Utasítások

A program szövege utasításokból áll. Az utasításokat ponttal zárjuk le. A kód utolsó utasítása után nem kötelező a pont.

Üzenet

A Smalltalk egy üzenet alapú rendszer. Az objektumok üzenetek segítségével működnek együtt. (Ez megfelel egy alprogram hívásnak.) Az egyik objektum küldi (küldő); a fogadó megkapja, és válaszol mindig az üzenetre. Az üzenet formáját a módszer interfész része írja le. Az üzenetnek lehetnek argumentumai (~módszer paramétere). Minden üzenet meg van nevezve. A küldő elküldi az üzenet nevét és argumentumait, a fogadó értelmezi az elküldött üzenetet, majd megválaszolja azt. Mindig van visszatérési érték. Tehát az üzenetek módszerek. Mindkét objektumnak tudnia kell, hogy az üzenet mire való.
Elnevezés: az üzenet neve szelektor. Ez a név kisbetűvel kezdődik. Egy üzenetnek van egy szelektora és lehetnek argumentumai.



Üzenetek két csoportja:

- Lekérdező módszerek (üzenetek): az objektum állapotát kérdezi le. Nincs argumentumuk.
- A beállító módszerek (üzenetek): megváltoztatják az objektum állapotát (beállítja a fogadó objektum valamely változójának értékét). Általában van argumentumuk. Nevüket kettősponttal kell lezárni.

A változók és módszerek neve lehet ugyanaz: névtúlterheltség. A kódban levő pozíció dönti el, hogy változóról vagy módszerről van-e szó.

Az üzeneteknek három csoportja van:

- unáris: nincs argumentumuk, lekérdező üzenetek. Csak szelektor van, nincs paraméter.
- kulcsszó (keyword): van argumentumuk a szelektor után. (beállító üzenetek)
- bináris: realizálják az eljárásorientált nyelvek operátorait. Az aritmetikai, logikai, hasonlító operátorokat realizáló üzenetek. Módszerekként vannak realizálva ezen operátorok.

Kifejezés

Egy Smalltalk utasítás egy vagy több kifejezést tartalmaz.

Kifejezés típusai

1. elsődleges kifejezés
2. üzenet kifejezés
3. kaszkád kifejezés

1. Elsődleges kifejezés lehet:

- változónév
- literál
- (kifejezés)

2. Üzenet kifejezés: ez a leggyakoribb.

Alakja:

```
fogadóobjektum üzenet_neve argumentumok
```

Paraméterkiértékelés:

- sorrendi kötés
- számbeli egyeztetés
- (típus nincs)

Paraméterátadás: üzenettől függ, hogy mi kerül átadásra. Az objektum vagy az objektum állapota kerül átadásra.

- Értékadás: A következő üzenetkifejezéssel írható le a Smalltalkban:

```
x := 5.
```

`x` : fogadó objektum (változó)

`:=` : szelektor (üzenet)

`5` : argumentum

Ez egy üzenet kifejezés. Minden osztályban létezik ez a módszer (`:=`). A változó egy objektumazonosítót tartalmaz. Ezen objektum "értéke" állítódik be – egy megadott állapotba kerül. Hatására az `x` változó értékét állítja be arra a referenciára, amely az `5`-t, mint a `SmallInteger` osztály példányát címzi.

Általánosan:

```
változónév := utasítás.
```

– Return kifejezés:

`^utasítás` : Ez egy olyan üzenet, aminek nincs fogadója. Az objektum értékére, állapotára való hivatkozást jelenti. Az utasítás értékével tér vissza.

Példa: egy return kifejezésre:

```
x := x + 1.        (:=  kulcsszavas üzenet)
^x.
```

`x`: fogadó

`:=` : szelektor

`x + 1`: paraméter

Ki kell értékelni a paramétert, ami egy üzenetkifejezés, ahol:

`x`: fogadó

`+`: szelektor

`1`: argumentum

De ugyanazt adja a következő is:

```
^x := x + 1.
```

A

```
^x + 1.
```

is ugyanazzal az értékkel tér vissza (`x`-nél egyel nagyobbat), de `x` értéke nem változik.

– Tömbliterálok osztályában létezik egy módszer az index_módszer:

```
#(1 2 3 4 5) at: 2.
```

A `2` index argumentum.

Az `at`: kulcsszavas üzenet. Fogadó objektum: bármely objektum lehet, amely ismeri ezt a módszert.

A módszer visszaadja a második értéket, mint literálobjekumot.

3. Kaszkád kifejezés: ugyanahhoz a fogadó objektumhoz több üzenetet akarok küldeni, anélkül hogy mindig felírnám a fogadó nevét.

   Alakja:

   `fogadó üzenetek (üzenetkifejezés_lista` pontosvesszővel elválasztva)

Kifejezések kiértékelése

Prioritási sorrend van az üzenetek három csoportja között:

1. unáris
2. bináris
3. kulcsszavas

Egy kifejezés tetszőleges bonyolultságú lehet. Tetszőleges sok üzenet lehet egy kifejezésben.

A balról-jobbra szabály érvényes. Egy kifejezésen belül kiértékeljük az összes unáris üzenetet balról-jobbra, aztán az összes bináris, majd a kulcsszavasok, a felsorolás sorrendjében.

A kiértékelést `()`-kel szabályozhatom. Egyértelmű a kiértékelés.

1. Unáris üzenetek:

   A `not` unáris üzenet:

   ```
   (3 > 2) not  "De a 3>2 not ROSSZ!"
   ```

   `(3 > 2)`: Boolean példány

   `not` : üzenet (argumentum nélküli)

2. Bináris üzenetek (kifejezések):

– Aritmetikai üzenetek: a `Number` osztály módszerei.

Operátorok:

- `+` : összeadás
- `-` : kivonás
- `*` : szorzás
- `/` : osztás
- `//` : egészosztás
- `\\` : maradékképzés

Példa:

`4/3` Literál, a `Fraction` osztály egy példánya.

`4 / 3` Üzenet kifejezés. Keletkezik egy olyan objektum, amelynek értéke: `4/3`.

– Hasonlító operátorok: a `Magnitude` osztály módszerei:

  `<` : nagyobb

  `>` : kisebb

  `=` : egyenlőség, két objektum értékének azonossága

  `<=` : kisebb vagy egyenlő

  `>=` : nagyobb vagy egyenlő

  `==` : két objektum azonossága, referenciában való egyenlőség (ugyanaz az objektum)

  Eredményük mindig egy olyan objektum, amely a `True` vagy a `False` osztály egy példánya.

```
x := $A < $a.
   ⇧      ⇧
kulcsszavas  bináris
```

– Logikai bináris üzenetek: A `Boolean` osztály példányain értelmezett.

  `&` : és

  `|` : vagy

  Teljes kiértékelés van, nincs rövidzár kiértékelés.

Blokk

Olyan objektum, amely végrehajtható kódot tartalmaz.

– Utasításokat tartalmaz.

– Elhelyezhető módszerek törzsében.

– Egymásba skatulyázható.

– Formálisan `[ ]` zárójelek között áll.

– Tartalmazhat lokális változókat. Ezek a változók a blokk argumentumai. Nevük előtt kettőspont áll. A lokális változókat `[ |` között soroljuk fel: közvetlenül a `[` jel után vannak felsorolva, ezután egy `|` jön, majd az utasítások.

  `| ]` között utasítássorozat áll.

Vezérlési szerkezetek

Feltételes végrehajtás:

```
Boolean_példány {ifTrue: / ifFalse:} blokk
```

A `Boolean_példány` a fogadóosztály.

A `blokk` argumentum nélküli.

Az `ifTrue` és az `ifFalse` a `Boolean` osztály egy-egy módszere: üzenet.

Példa:

```
a < b ifTrue: [max := b]; ifFalse: [max := a]. ”kaszkád üzenet”
```

Ezek kulcsszavas üzenetek. Két egymástól független módszer.

Ha a módszereket egymás után alkalmazni akarom, vannak olyan módszerek, amelyek a kettőt összevonják. Ez kiváltja a kaszkád üzenetet. Egyetlen objektumra több üzenetet alkalmazok.

Rövidzár logikai műveleteket realizáló üzenetek:

```
Boolean_példány {and: / or:} blokk        "Logikai objektumot eredményez."
Boolean_példány not
```

Ciklusok:

```
Integer_példány timesRepeat: blokk
```

`Integer_példány` -szor hajtja végre a blokkban lévő kódot. A `blokk` argumentum nélküli.

Kezdőfeltételes ciklusnak megfelelő konstrukció:

```
Boolean_példány {whileTrue: / whileFalse:} blokk
```

A `blokk` argumentum nélküli.

Példa:

```
x := 5.
y := 0.
[y <= x] whileTrue: [y := y + 1]
```

Előírt lépésszámú ciklusnak megfelelő konstrukció: elöltesztelő

```
Number_példány to: Number_példány by: Number_példány do
     [:v | utasítások]
```

Például:

```
k to: v by: l do: [:cv|kód]
```

`k`, `v`, `l` (kezdet, vég, lépésköz) :a Number osztály egy-egy példánya. Ha `l`-et nem adjuk meg, annak értéke alapértelmezés szerint 1.

`cv` : ciklusváltozó

`kód` :ciklusmag

Példa:

```
s := 0.
1/2 to: 1 by: 1/8 do: [:i | s := s + i].
^s.
```

Példa: A magánhangzókat kicsire, a mássalhangzókat pedig nagyra változtatja a sztringben.

```
string := 'Ez a sztring.'.

index := 1.
string size timesRepeat:                "a size egy üzenet"
  [c := string at: index.

   string at: index put:"a    megfelelő    indexű    elem
helyettesítése"

        (c isVowel  ifTrue:[c asUpperCase];
                    ifFalse:[c asLowerCase]).

   index := index + 1].

^string.
```

A `string as: index` a `string` tömb megfelelő értékéhez való hozzáférést teszi lehetővé. A tömbindex mindig 1-től indul.

Az `asUppercase` és az `asLowerCase` unáris üzenet.

Példa: Hány magánhangzó van egy adott sztringben?

```
m := 0.
s := 'No ebben mennyi van?'.
v := s size.
1 to: v do: [:i|c := s at: i.
                c isVowel ifTrue: [m := m + 1]].
^m.
```

A Smalltalk is algoritmikus nyelv. Rengeteg művelet implementálva van üzenet szinten.

Osztályok

Az osztályok definiálása (az osztályhierarchiába való elhelyezése) interaktív módon, az eddigi osztályokhoz kapcsolódóan történik. Önálló osztályok nem léteznek, csak osztályhierarchia. A Smalltalk egy integrált fejlesztői környezetet ad, és ebben több lehetőség van új osztály létrehozására. Az új osztály szuperosztályához új példány-, osztály- és szótárváltozókat, új módszereket definiálhat, és módszereket újraimplementálhat.
Itt kódírás következik, majd a módszerek lefordítása, majd tesztelése.
Minden egyes osztály a `MetaClass` osztály példánya. Egy osztály így jön létre. A `MetaClass` osztály alosztálya az `Object` osztálynak.
Új osztály definíciója:

```
szuperosztály_név SUBCLASS: #név                "A #név egy szimbólum."
```

Példányosítani minden osztályban a megfelelő osztályhoz küldött `new` üzenettel lehet. Az adott változó értékét az új példány OID-jére állítom.

```
változónév := osztálynév NEW.
```

A változók *automatikus kezdőértéke* `nil`, amely az `UndefinedObject` osztály példánya.

Módszerek

A módszerek definíciója három részből áll (a második elmaradhat) a Smalltalk terminológiája szerint:

- interfész
- lokális változók

– kód

1. Az interfész rész:

   Megfelel a korábbi alprogram specifikációnak.

   Szerkezete:

   ```
   név argumentumok [név argumentumok]... .
   ```

   – Ha a név kettősponttal zárul, akkor ezek a módszerek beállító üzenetek. Ekkor argumentumok szükségeltetnek, minimum 1.
   – Ha nincs kettőspont, akkor lekérdező módszer. Itt általában tilos argumentumot megadni.
   – A formális paraméterek, az argumentumok lokális változói szerepkörben vannak.
     Ha több nevet adok meg, nem kell kaszkádolni, felsorolhatóak.

2. Adatrész:

   Az interfész után | | között szerepelnek a lokális változók, amennyiben vannak. Ez az adatrész, ahol a lokális változókat sorolja fel.

3. Kód:

   Utasítások sorozata.

A módszer visszatérési értékét egy `^üzenet` realizálja. Ez megfelel egy `return kifejezés` C utasításnak.
A módszerek neve kisbetűvel kezdődik, ha beállító módszer, akkor a végén ott a kettőspont.
Definiáljuk például az `Object` osztályhoz kapcsolódóan a `Szemely` osztályt.

```
Object subclass: #Szemely
```

Példányváltozói:

– `nev`
– `cim`
– `telefonszam`

Módszerei lehetnek például a következők:

```
nev          "lekérdező módszer, unáris"
 ^nev

nev: egyNev  "beállító, kulcsszavas módszer"
 nev := egyNev

cim
 ^cim

cim: egyCim
 cím := egyCim
```

Változók és módszerek nevei lehetnek azonosak, mert a pozíciójuk egyértelműen eldönti, hogy melyikről van szó. A Smalltalk kifejezetten javasolja is.

OO körökben kódolási szabálynak tekinthető, hogy ha osztályról és példányáról vagy változóról és értékéről van szó, akkor a következő kódolási konvenció érvényes:

`egySzemély` (angolul: `aPerson`)

```
nev: egyNev  cim: egyCim
    self nev: egyNev.
    self cim: egyCim.
```

A `self` az aktuális példányt, mint objektumot jelenti a Smalltalkban.

Az intrefész részt nem zárjuk ponttal, csak az utasítást.

Ezek után a következő utasítások alkalmazhatóak (példányosítás):

```
Valaki := Szemely new.
Valaki nev: 'Kovács Jenő'.
Valaki nev.
```

Az `Integer` osztálynak van egy `factorial` nevű módszere:

```
factorial
    self > 1 ifTrue: [^(self - 1) factorial * self].
    self < 0 ifTrue: [^(self error: 'negative factorial'] .
^1.
```

Példa: Egy sztring számmá konvertálásának a kódja a következő (egy egész vagy valós számot tartalmazó sztringet számmá konvertál)

```
ConvertToNumber: aString
|subStrings whole decimal exponent|
subStrings := aString subStrings: $. .
whole := (subStrings at: 1) asNumber.
subStrings size = 1
     ifTrue: [^whole]
     ifFalse: [decimal := subStrings at: 2.
          (decimal includes: $e)
          ifTrue: [subStrings := decimal subStrings: $e.
               exponent := (subStrings at: 2) asNumber.
               decimal := subStrings at: 1]
          ifFalse: [exponent := 0].
               ^(whole + (decimal asNumber / (10 raisedTo:
                    (decimal(size))))*(10  raisedTo:  exponent)
asFloat].
```

Beszélő üzenetnevek használata javasolt. A sztringet egydimenziós tömbként tekinti, megnézi, hogy van-e benne pont, vagy exponens rész.

A kollekciókról nem szólunk, de a Smalltalk ismeri a következőket: halmaz, multihalmaz, tömb, lista, rendezett lista, táblázat.

6. A funkcionális paradigma

A funkcionális paradigma középpontjában a *függvények* állnak. Egy funkcionális (vagy *applikatív*) nyelvben egy program típus-, osztály-, és függvénydeklarációk, illetve függvénydeklarációk, illetve függvénydefiníciók sorozatából, valamint egy *kezdeti kifejezésből* áll. A kezdeti kifejezésben tetszőleges hosszúságú (esetleg egymásba ágyazott) függvényhívás sorozat jelenhet meg. A program végrehajtását a kezdeti kifejezés kiértékelése jelenti. Ezt úgy képzelhetjük el, hogy a kezdeti kifejezésben szereplő függvények meghívása úgy zajlik le, hogy a hívást szövegszerűen (a paraméterek figyelembevételével) helyettesítjük a definíció törzsével. A helyettesítés pontos szemantikáját az egyes nyelvek kiértékelési (átírási) modellje határozza meg.

A funkcionális nyelvek esetén nem választható szét a nyelvi rendszer a környezettől. Ezek a nyelvi rendszerek interpreter alapúak, interaktívak, de tartalmaznak fordítóprogramokat is. Középpontjukban mindig egy *redukciós* (átíró) rendszer áll. Ha a redukciós rendszer olyan, hogy az egyes részkifejezések átírásának sorrendje nincs hatással a végeredményre, akkor azt *konfliktusnak* nevezzük.

Egy funkcionális nyelvű program legfontosabb építőkövei a saját függvények. Ezek fogalmilag semmiben sem különböznek az eljárásorientált nyelvek függvényeitől. A függvény törzse meghatározza adott aktuális paraméterek mellett a visszatérési érték kiszámításának módját. A függvény törzse a funkcionális nyelvekben kifejezés.

Egy funkcionális nyelvi rendszer beépített függvények sokaságából áll. Saját függvényt beépített, vagy általunk már korábban definiált függvények segítségével definiálni (függvényösszetétel).

Egy funkcionális nyelvben a függvények alapértelmezett módon rekurzívak lehetnek, sőt létrehozhatók kölcsönösen rekurzív függvények.

A kezdeti kifejezés redukálása (a nyelv által megvalósított *kiértékelési stratégia* alapján) mindig egy redukálható részkifejezés (egy *redex*) átírásával kezdődik. Ha a kifejezés már nem redukálható tovább, akkor *normál formájú* kifejezésről beszélhetünk.

A kiértékelés lehet *lusta kiértékelés*, ekkor a kifejezésben a legbaloldalibb, legkülső redex kerül átírásra. Ez azt jelenti, hogy ha a kifejezés egy függvényhívás, akkor az aktuális paraméterek kiértékelését csak akkor végzi el a rendszer, ha szükség van rájuk. A lusta kiértékelés mindig eljut a normál formáig, ha az létezik.

A mohó kiértékelés a legbaloldalibb, legbelső redexet írja át először. Ekkor tehát az aktuális paraméterek kiértékelése történik meg először.

A mohó kiértékelés gyakran hatékonyabb, de nem biztos, hogy véget ér, még akkor sem, ha létezik a normál forma.

Egy funkcionális nyelvet *tisztán funkcionálisnak* (tisztán *applikatívnak*) nevezünk, ha nyelvi elemeinek nincs mellékhatása és nincs lehetőség értékadásra vagy más eljárásorientált nyelvi elem használatára.

A nem tisztán funkcionális nyelvekben viszont van mellékhatás, vannak eljárásorientált (néha objektumorientált) vagy azokhoz hasonló eszközök.

A tisztán funkcionális nyelvekben teljesül a *hivatkozási átláthatóság*. Ez azt jelenti, hogy egy kifejezés értéke nem függ attól, hogy a program mely részén fordul elő. Tehát ugyanazon kifejezés értéke a szöveg bármely pontján ugyanaz. A függvények nem változtatják meg a környezetüket, azaz a tartalmazó kifejezés értékét nem befolyásolják. Az ilyen nyelvnek nincsenek változói, csak konstansai és nevesített konstansai.

A tisztán funkcionális nyelvek általában szigorúan típusosak, a fordítóprogram ellenőrzi a típuskompatibilitást.

Ezek a nyelvek eszközként tartalmaznak olyan függvényeket, melyek paramétere, vagy visszatérési értéke függvény (*funkcionálok*, vagy magasabb rendű függvények). Ez a funkcionális absztrakciót szolgálja.

A funkcionális nyelvek egy részének kivételkezelése gyenge vagy nem létezik, másoknál hatékony eszközrendszer áll rendelkezésre.

A függvényösszetétel asszociatív, így a funkcionális nyelven megírt programok kiértékelése jól párhuzamosítható. Az elterjedt funkcionális nyelveknek általában van párhuzamos változata.

A következő két fejezetben a funkcionális nyelvek két jellegzetes képviselőjét tárgyaljuk, azokon bemutatva a paradigma szokásos eszközeit, … és filozófiáját.

A **Haskell** egy erősen típusos, tisztán funkcionális, lusta kiértékelést megvalósító, a **LISP** egy imperatív eszközöket is tartalmazó, objektumorientált változattal (**LLOS**) is rendelkező, mohó kiértékelést valló funkcionális nyelv.

7. LISP

A LISP a funkcionális paradigma első nyelveként az 1950-es évek második felében jött létre, mint mesterséges intelligencia kutatásokat támogató nyelv. Sok verziója létezik. Mi a Common LISP változatot, illetve ennek objektumorientált eszközökkel kibővített verzióját tárgyaljuk, ennek neve CLOS (Common LISP Object System).

A LISP egy interpreteres, interaktív nyelvi rendszer, amelyet beépített függvények alkotnak. Ezek azonnal, párbeszédes üzemmódba hívhatók.

A LISP programozás saját függvények definiálását és azok alkalmazását jelenti.

A nyelv nem típusos.

7.1. A CLOS alapelemei

Karakterkészlete az ASCII-n alapul, a kis és nagy betűket nem különbözteti meg.

A nyelvnek nincsenek alapszavai. A beépített függvények nevei standard azonosítók.

Az azonosítók betűkből, számjegyekből és a következő karakterekből alkotott, tetszőleges hosszúságú karaktersorozatok: +, –, *, /, @, $, %, ^, &, _, =, <, >, ~, .

Az azonosítók a nyelvben változók és függvények nevei lehetnek. Ezeknél a LISP a kisbetűket automatikusan nagybetűsre alakítja át, az outputban már csak ez az adat jelenik meg.

Megjegyzést bármely sor végén helyezhetünk el a ; karakter után, a sor végéig.

A nyelv alapépítő elemei az *atomok*. Egy atom lehet *numerikus atom* vagy *szám*. Ez megfelel az eljárásorientált nyelvek numerikus literáljának. A LISP decimális számrendszert hanszál. A számok fajtái a következők:

Szám
- Valós
- Racionális
  - Tört
  - Egész

A valós számnak megvan a tizedestört és az exponenciális alakja, az egész a szokásos. A tört esetén a számlálót és a nevezőt egy / választja el.

A LISP a törtet mindig *harmonikus* alakra alakítja (tehát a számláló és nevező relatív prímek és a nevező pozitív).

Példák számokra:

-1, 28, .07, -11.3, 0.888e-3, 1416, -518.

A *szimbolikus atom* vagy *szimbólum* egy azonosító, amely szövegkörnyezettől függően csak önmagát, mint karaktersorozatot jelenti, vagy pedig egy programozói eszköz neve.

A nyelvnek vannak beépített szimbólumai. Például a T egy nevesített konstansnak tekinthető, amelynek értéke a logikai igaz.

Az atomokon kívül a LISP másik alapeszköze a *lista*, amelyről a nevét is kapta (List Processing). A lista kerek zárójelbe zárt atomok és listák sorozata. A hierarchikus lista (1. **Adatszerkezetek és algoritmusok** című tárgy) absztrakt adatszerkezet nyelvi realizációja.

Példák listára:

```
( )        ; üres lista
(Ez egy 5 elemu lista)
((a b) (c d))
```

Az atomokat és a listákat a LISP közös néven *S-kifejezésnek* (*szimbolikus kifejezésnek*) hívja.

A LISP-ben mind a program, mind az adat S-kifejezés segítségével kezelhető. Tehát egy LISP program feldolgozhat egy másik LISP programot adatként és a futás eredménye egy újabb LISP program lehet.

A LISP-ben van változó. A változót definiálni kell, lehet neki explicit kezdőértéket adni és értéke tetszőlegesen megváltoztatható.

A LISP-ben van kifejezés. Ugyanis az S-kifejezés vagy adatot határoz meg, vagy csak önmagát, mint karaktersorozatot jelenti, vagy kifejezés és mint ilyen kiértékelendő. A numerikus atom értéke önmaga, változó értéke az aktuális érték. Lista esetén viszont ekkor a lista első elemének egy függvénynévnek kell lennie és ekkor ez egy függvényhívás. A lista további elemei a függvény aktuális paraméterei. A kifejezés eredményét ekkor a függvény visszatérési értéke adja. Az aktuális paraméterek S-kifejezések lehetnek.

A LISP kifejezése *prefix* kifejezés, az operátoroknak a függvénynevek, az operandusoknak S-kifejezések felelnek meg.

A kifejezés értéke maga is S-kifejezés.

Egy változó értékéül S-kifejezést vehet fel.

A LISP függvényei lehetnek fix és változó paraméterszámúak, így az operátorok egy része tetszőleges számú operandusra értelmezett.

A LISP interpreter ezek után alaphelyzetben a következőképpen működik:

1. Megadunk neki egy S-kifejezést, ez a program. Ezt *beolvassa* (*read*).
2. Értelmezi az S-kifejezést, meghatározza az értékét (*evaluate*).
3. Kiírja az értékét a képernyőre (*print*).

Ezt hívja a LISP read-evaluate-print ciklusnak.

A LISP függvények esetén a paraméterkiértékelésnél mindig sorrendi kötés, a fix paraméterűeknél számbeli egyeztetés érvényesül. A paraméterátadás lehet érték szerinti, ekkor az aktuális paraméterként megadott S-kifejezés kiértékelődik. Az ilyen függvények nem változtatják meg paramétereiket, tehát ebből a szempontból mellékhatás mentesek (tisztán applikatívak).

A paraméterátadás azonban lehet *szimbolikus*, ekkor az aktuális paraméter nem értékelődik ki, hanem mint szimbólum kerül átadásra. Ezek a függvények meg tudják változtatni a paramétereiket, tehát **lehet** mellékhatásuk. Az ilyen függvényeket egyes LISP verziók *álfüggvényeknek* hívják. A CLOS ekkor *makróról* beszél, megkülönböztetve őket a *függvényektől*. A makrókról részletesebben l….

7.2. CLOS beépített függvények

Aritmetikus függvények

+	összeadás, tetszőleges számú paraméter
–	kivonás, legalább 1 paraméter
*	szorzás, tetszőleges számú paraméter
/	egészosztás, 2 paraméter
rem	maradékképzés, 2 paraméter
1+	növelés 1-el, 1 paraméter
1–	csökkentés 1-el, 1 paraméter
Sqrt	négyzetgyök, 1 paraméter
exp	hatványozás, 2 paraméter
gcd	legnagyobb közös osztó, tetszőleges számú paraméter
lcm	legkisebb közös többszörös, tetszőleges számú (legalább 1) paraméter
abs	abszolút érték, 1 paraméter
min	legkisebb érték, legalább 1 paraméter
max	legnagyobb érték, legalább 1 paraméter

Példák:

A példákban a > a promptjel, a LIST válasza az alatta levő sor(ok)ban látható.

Predikátumok

Olyan függvények, amelyek logikai visszatérési értékkel rendelkeznek. Nevük p-re végződik. Itt jegyezzük meg, hogy a logikai hamis értéket a LISP a NIL beépített szimbólumokkal (mint nevesített konstanssal) kezeli. **Az üres lista értéke is NIL!**

> ( )

NIL

Sok LISP verzió az ún. általánosított logikai értékeket kezeli. Ezek azt mondják meg, hogy ha valaminek az értéke **nem NIL**, akkor az **igaz**. A CLOS is ezt az elvet valósítja meg.

Néhány predikátum számok fajtáját dönti el:

numberp	a paramétere szám-e
realp	a paramétere valós-e
rationalp	a paramétere racionális-e
ratiop	a paramétere tört-e
integerp	a paramétere egész-e

Példák:

M

A következő predikátumok szintén számokat vizsgálnak:

zerop	a paramétere nulla-e
plusp	a paramétere pozitív-e
minusp	a paramétere negatív-e
oddp	a paramétere páratlan-e
evenp	a paramétere páros-e

Az alábbi függvények legalább egy paraméterrel rendelkeznek, ezek is predikátumok, a paramétereiket hasonlítják össze:

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

Konverziós függvények

A paraméterük egy szám, amelyet másik fajtájú számmá alakítanak át.

float	valóssá alakít
rational	törtté alakít át
truncate	csonkít
round	kerekít

Példák:

M

Logikai függvények

A not, and, or rendre a logikai tagadás, és illetve vagy műveletet realizálja.

A not egy paraméterű, az and és or tetszőleges számú paraméterrel rendelkezik.

Az and függvény visszatérési értéke NIL, ha valamelyik paramétere NIL, különben a legutolsó paraméterének értéke. Ha paraméter nélkül hívtuk meg, T-vel tér vissza.

Az or visszatérési értéke NIL, ha valamennyi paramétere NIL értékű, egyébként az első nem NIL értékű paraméterének értéke. Ha paraméter nélkül hívtuk meg, NIL-el tér vissza.

Tehát az and és or rövidzár kiértékelésű.

Példák:

M

Feltételes függvények

Segítségükkel feltételes kifejezéseket tudunk összeállítani, szerepük a saját függvény létrehozásánál van.

Az if függvénynek két vagy három paramétere van. Ha az első paraméter értéke nem NIL, akkor a második paraméter kiértékelődik,és ez adja a visszatérési értéket. Ha NIL, akkor, ha meg van adva harmadik paraméter, akkor annak értéke lesz a visszatérési érték, egyébként pedig NIL.

Példák:

A cond függvény nem fix paraméterszámú függvény, paraméterei listák. Formája:

(COND (feltétel [S-kifejezés]…)
[(feltétel [S-kifejezés]…)]…)

Szemantikája a következő: A megadás sorrendjében kiértékelésre kerülnek a feltételek. Ha valamelyik értéke nem NIL, akkor a mellette megadott S-kifejezések közül az utolsó értéke adja a visszatérési értéket. Ha nincs S-kifejezés, akkor a feltétel értéke (ami nem NIL) határozza meg a visszatérési értéket. Ha minden feltétel értéke NIL, akkor a cond is NIL-el tér vissza.

A quote függvény

Egyparaméteres függvény, amelynek visszatérési értéke a aktuális paraméterként megadott S-kifejezés. Arra szolgál, hogy érték szerinti paraméterátadással rendelkező paramétereknél az aktuális paraméter kiértékelését megakadályozzuk

Példa:

> (quote (+ 5 6))
(+ 5 6)

Szerepe annyira fontos, hogy rövidíteni lehet a ’ karakterrel, a fenti függvényhívás ekvivalens az alábbival:

> ’ (+ 5 6)

7.3. Nevesített konstans, változó, saját függvény

A programozó a CLOS-ban saját nevesített konstanst a `defconstant` makróval hozhat létre, melynek első paramétere a nevesített konstans neve, a második az értéke.

Példa:

M

A CLOS-ban egy változót a defrar makróval lehet definiálni. Első paramétere a változó neve, második opcionális paramétere a kezdőértéke.

Példák:

> (defrar a 8)

A

> (defrar a)

Az első esetben a kezdőérték 8, a másodikban nincs kezdőértékadás.

A makró visszatérési értéke a változó neve, mint szimbólum. A defrar a második paraméterét kiértékeli, az elsőt nem. Mellékhatásként viszont az első paraméterének értéket adhat.

Egy változó értékét a setf makró segítségével tudjuk megváltoztatni (értékadás). Legalább két paramétere van, az első a változó neve, a második az új érték. Visszatérési érték a második paraméter értéke. Egyszerre több változónak is tudunk értéket adni segítségével.

Példák:

```
> (setf a 5)
5
> (setf a 5 b 6)
6
```

Saját függvényt a defun makró segítségével hozhatunk létre. A függvénydefiníció általános formája:

(defun *név formális_paraméter_lista törzs*)

A *név* egy szimbólum. A *formális_paraméter_lista* egy szimbólumokat tartalmazó lista. A formális paraméterek a függvény lokális változói. A *törzs* egy S-kifejezés sorozat. A defun függvény visszatérési értéke *határozatlan*, CLOS nem határozza meg azt. Az implementációk viszont generálhatnak valamilyen visszatérési értéket (pl. a függvény nevét).

A visszatérési értéket a törzs határozza meg. Az így definiált függvény paramétereinek paraméterátadási módja érték szerinti lesz. Meghívni annyi aktuális paraméterrel lehet, ahány formális paraméter megadtunk, tehát az új függvény fix paraméterszámú lesz.

A defun segítségével definiálhatunk nem fix paraméterszámú saját függvényt is, úgy, hogy a formális paraméter listán egyetlen szimbólumot adunk meg, és előtte szerepeltetjük az &rest kulcsszót (l. …). Ha a paraméterek számának alsó korlátot akarunk megszabni, akkor a formális paraméter listán megadunk adott számú szimbólumot és a listát zárja a fenti konstrukció.

Példák:

1. A következő saját függvény az abszolútérték függvényt implementálja logikai függvények segítségével)az általánosított logikai értékek miatt):

```
(defun absz (n)
        (or (and (minusp n) (* –1 n)) n))
```

2. A faktoriális függvény rekurzív változata

```
(defun fakt (n)
        (cond ((zerop n) 1)
                (T )(  n (fakt (–n 1))))
```

A továbbiakban a saját függvénydefiníciók lezárásaként a > jelet fogjuk alkalmazni (mint ahogy több implementáció is), az egy kényelmi jelölés, az összes olyan balzárójelhez, amelynek még nincs jobbzárójele, hozzápárosít egy jobbzárójelet.

7.4. Listák

Egy lista fejét a car, a farkát a cdr függvény szolgáltatja. Paraméterük természetesen egy lista. Az üres listára értékük NIL.

A cons függvénynek két paramétere van, ezekből állít elő egy listát úgy, hogy kiértékeli őket és az első paraméter értéke lesz a lista feje, a második a farka.

Példák:

```
> (defrar a '(+ 2  3  4))
A
> 'a
A
> a
(+ 2  3  4)
> (car a)
+
> (cdr a)
(2  3  4)
> (car (cdr ' (a  b  c))
B
> (cons 'a '(b  c))
(A  B  C)
> (cons 'a '( ))
(A)
> (cons 'a 'b)
(A . B)
```

Az utolsó példa mutatja a *lista* és a *valódi lista* közötti fogalmi különbséget. A valódi lista mindig az üres listával végződik. Tehát rendre alkalmazza … a cdr függvényt, az utolsó mindig NIL-el tér vissza. A *nemvalódi lista* esetén viszont az utolsó cdr visszatérési értéke nem NIL, ilyen akkor keletkezik, ha a cons második paramétere nem lista. Ilyenkor a kiírt lista egy *pontozott párt* tartalmaz, ahol a pont utáni rész az utolsó cdr értékét mutatja.

Az endp predikátum a lista *végét* teszteli. Értéke igaz, ha paramétere az üres lista és NIL, ha nem.

A listák memóriában történő ábrázolása a következőképpen történik:

A valódi lista mindig egy pointerpárból áll. Ezek közül az első címzi a lista fejét, a második a farkát. A `cons` függvény ezt a két pointert hozza létre. Tehát a

```
> (cons 'a (cons 'b nil))
(A  B)
```

hatására az alábbi ábrázolási lista jön létre:

A

```
> (cons (cons ‘a nil) nil)
((A)
```

lista viszont a következőképpen néz ki:

A nemvalódi

```
> (cons ‘a ‘b)
(A . B)
```

pontozott pár szerkezete viszont:

A listákra vonatkozó további függvények az alábbiak:

A `list` egy nem fix paraméterszámú függvény, amely paramétereinek értékéből listát képez.

Példák:

```
> (list)
NIL
> (list 1  2  3)
(1  2  3)
> (list ’(ab) ’(cd))
((AB)(CD))
> list ’a (list ’b ’c))
(A (B  C))
```

Az `append` nem fix paraméterszámú függvény, amely listát alkotó paramétereiből egyetlen listát képez (összefűzés).

Példák:

```
> (append ’(ab) ’(cd))
> (A  B  C  D)
```

A `length` a lista elemeinek számát adja, a `reverse` pedig a lista legkülső szintjén levő elemek sorrendjét megfordítja (tehát a beágyazott listák változatlanok maradnak).

Példák:

```
> (length ‘( ))
0
```

```
> (length '((ab) 1 (x))
3
> reverse '((abc) (de))
((D  E) (A  B  C))
```

A `subst` függvénynek három paramétere van. A harmadik paraméter egy lista. Ezen listában a második paraméter összes (tehát nemcsak a legkülső szintű) előfordulását helyettesíti az első paraméterrel.

Példa:

```
> (substr 5  0 '(0  1  2  0))
(5  1  2  5)
```

A függvények a nyelvnek ugyanazon eszközei, mint a szimbólumok vagy a listák. A function függvények egyetlen argumentuma egy függvény neve, és visszatérési értéke maga a függvény. A függvény nevét viszont S-kifejezés értékeként állíthatjuk elő. A `funcall` függvénnyel pedig explicit módon meg tudunk hívni egy függvényt. A `function` függvény rövidíthető a #' szimbólumpárral (a `quote` mintájára).

Példa:

```
> (funcall #' + 1  2  3)
6
```

A LISP listák kezelésének egy igen hatékony eszköze, a procedurális absztrakciót nagyban támogató `mapcar` függvény. Első paramétere egy függvény, továbbiak pedig listák. A visszatérési értéke pedig olyan lista, amely úgy keletkezik, hogy a függvény meghívódik minden paraméterként, megadott lista első, második, stb. elemeire. Az eredmény lista hossza a legrövidebb lista hossza lesz.

A `mapcar` függvény tehát egy olyan függvény, amelynek paramétere függvény. Ezeket hívja a LISP *általánosított függvényeknek* vagy *funkcionálisoknak*.

Példa:

```
> (mapcar #'+ '(1  2  3) '(4  5))
(5  7)
```

A listák kezelésére szolgálnak a következő predikátumok:

symbolp	igaz, ha paramétere szimbólum
consp	igaz, ha paramétere valódi lista
atom	igaz, ha paramétere nem valódi lista
listp	igaz, ha paramétere lista
null	igaz, ha paramétere az üres lista

Itt jegyezzük meg, hogy az üres lista, nem valódi lista és a symbolp szimbólumnak (NIL) tekinti.

Eszközök összehasonlítására szolgál az `eq` és `equalp` függvény. Az `eq` függvény akkor ad igen értéket, ha pontosan azonos két eszköz (vagyis pontosan azonos memóriaterületen vannak!). Az `equalp` viszont akkor igaz, ha paraméterei mind számok, szimbólumok vagy listák azonosak (függetlenül a memóriabeli elhelyezkedésüktől).

Példák:

```
> (eq ‘a ‘a)                          ; azonos szimbólumok címe azonos
T
> (eq ’(a) ’(a))                     ; két lista címe különbözik
NIL
> (equalp ‘(a) ‘(a))                 ; lista és lista
T
> (equalp (+ 2 2) 4)                 ; 4 és 4
T
> (equalp ‘(+ 2 2) 4)                ; lista és szám
NIL
```

A LISP definiálja a `car` és `cdr` függvények kombinációit:

```
(car (cor x)) = (car x)
(car (cdr x)) = (codr x)
(cdr (cor x)) = (cdar x)
(cdr (cdr x)) = (cddr x)
```

A beágyazási szint maximum négy lehet (tehát még létezik a `caaddr`).

A `nth` függvény egy lista i elemét adja vissza (a sorszámozás 0-val indul). Értéke NIL, ha nincs ilyen elem.

Példák:

```
> (nth 0 ’(1  2  3  4)
1
> (nth 4 ’(1  2  3  4))
NIL
```

A `last` egy lista utolsó elemét, a `butlast` egy lista utolsó eleme nélküli listát adja meg.

A `member` és a `remove` függvények első paramétere egy elem, második egy lista. A `member` visszaadja a lista azon részlistáját, amely az elemmel kezdődik, vagy NIL-t. A `remove` a lista legkülső szintjéről eltávolítja az elem összes előfordulását.

Definiáljunk néhány saját függvényt, amelyek listákat kezelnek.

1. Adjuk meg a reverse definícióját.

```
(defun reverse (lista)
        cond ((null) lista) NIL)
                (T (append (reverse (cdr lista))
                (list (car lista>
```

2. Írjunk egy függvényt, amely egy lista legkülső szintjéről eltávolítja a 0-kat.

```
(defun nulla_eltavolitas (lista)
        (remove 0 lista>
```

3. Adjuk meg azt a függvényt, amely egy lista elejéhez hozzáfűzi az első paraméterének értékét, ha az szám

```
(defun szammal_borit (n lista)
        (cond ((numberp n) (cons n lista))
        (T lista>
```

4. Adjunk meg egy predikátumot, amely eldönti, hogy paramétere valódi lista-e.

```
(defun valodi_lista (lista)
        (if (not listp lista)) NIL
                (not (cdr (last lista>
```

5. Egy lista legkülső szintjén szüntessük meg a szimbólumok többszörös előfordulását.

```
(defun tobbsz_elt (lista)
        (cond ((endp lista) NIL)
                ((member (car lista) (cdr lista))
                        (tobbsz_elt (cdr lista)))
                (T cons (car lista)
                        (tobbsz_elt (cdr lista>
```

6. Határozzuk meg, hogy egy atom hányszor fordul elő egy listában (bármelyik szinten).

```
(defun gyakorisag (atom lista)
        (cond ((endp lista) 0)
                ((eq atom lista) 1)
                ((atom lista) 0)
                (T ( + (gyakorisag atom (car lista))
                        (gyakorisag atom (cdr lista>
```

7. Rendszerezzünk egy számokból álló listát nagyság szerint növekvőleg.

```
(defun rendez (szamok)
        (cond ((null szamok) NIL)
                ((null (cdr szamok)) szamok)
                (T beszur (car szamok)
                        (rendez (cdr szamok>
```

8. Vizsgáljuk meg, hogy egy csak szimbólumokat tartalmazó listában előfordul-e egy adott szimbólumsorozat (mintaillesztés). A szimbólumsorozatban szerepelhet a *** elem, amely bármely más szimbólumsorozatra illeszkedik.

```
(defun mink_ill (minta alap)
        (cond ((and (endp minta) (endp alap)) T)
                ((equalp (car minta) (car alap))
                        (minta_ill (cdr minta) (cdr alap)))
                ((eq (car minta) '***)
                        (cond ((endp alap) (endp (cdrp (cdr minta)))
                                (T (or (mink_ill (cdr minta) alap)
```

(minta_ill minta ‘cdr alap))))))

(T NIL>

7.5. Lokális és globális eszközök

A CLOS-ban a nevesített konstansok, a függvények és a `defvar` makróval létrehozott változók nevei *globálisak*, mindenhonnan láthatók. A saját függvények formális paraméterei viszont *lokálisak*, csak az adott függvényben hivatkozhatók. Ha egy lokális név megegyezik egy globálissal, akkor az adott függvény vonatkozásában elfedi azt.

Globális nevet egy függvényben is definiálhatunk és egy globális változó értékét megváltoztathatjuk. Ez viszont **mellékhatás**, a LISP szerint kerülendő. A CLOS ilyen esetben a `defparaméter` makró használatát javasolja, amely segítségével egy függvényben megváltoztatható globális változót tudunk definiálni. A CLOS konvenció szerint az ilyen változók neve előtt és után *, a nevesített konstansoknál pedig + szerepel.

Egy függvénydefiníció részeként definiálhatunk lokális változókat a `let` makróval, ennek alakja:

(LET ((*változónév* [*érték*])
[(*változónév*) [*érték*])]…
törzs)

A *törzs* S-kifejezések sorozata, az így definiált változók csak itt hivatkozhatók. Ha nem adunk kezdőértéket, akkor automatikusan NIL-t kapunk.

Példa: Egy csak számokat tartalmazó tetszőleges (akárhányszorosan egymásbaágyazott) lista elemeinek átlagát határozzuk meg.

```
(defun atlag (lista)
        (let ((a (atl lista 0 0)))
                (/ (car a) (cdr a>
(defun atl (l db ossz)
        (cond ((endp l) ‘(0.. 0))
                ((atom l) (cons (+ ossz l)
                                (+ db 1)))
                (T (val (atl (car l) db ossz)
                        (atl (cdr l) db ossz>
(defun val (x y)
        (cons (+ (car x) (car y))
                (+ (cdr x) (cdr y>
```

A LISP lehetőséget ad arra, hogy meg nem nevezett függvényeket tudjunk használni. Erre szolgál a lambda makró, amely után egy függvény formális paraméter listája és a törzse állhat. Ezzel tulajdonképpen egy lokális, a definíciójánál azonnal meghívásra is kerülő, a globális függvények közé fel nem veendő függvény használatára nyílik lehetőség.

Példák:

```
> ((lambda (n) (+  n  2))  5)
```

7
> (mapcar #'(lambda (x) (+ x 2)) '(1 2 3))
(3 4 5)

A `flot` függvény lehetővé teszi saját függvényen belül *lokális nemrekurzív*, a `labels` pedig *lokális rekurzív* függvény definiálását.

Példák:

1. A `length` függvény definíciója lehet az alábbi:

```
(defun length (lista)
    (labels ((hossz (lista n)
        (cond ((null lista) n)
            (T hossz (cdr lista)
                (1+ n))))))
        (hossz lista 0>
```

2. A `reverse` függvény lokális függvénnyel.

```
(defun reverse (lista)
    (labels ((fordit (lista uj)
        (cond ((null lista) uj)
            (T (fordit (cdr lista)
                (cons (car lista) uj))))))
        (fordit lista NIL>
```

7.6. Karakterek és sztringek

A karakterek és sztringek (akárcsak a számok) a LISP összdefiniáló eszközei.

A látható karakterek alakja:

`# \` *karakter*

Például: `#\a, #\L.`

A nem látható és vezérlő karakterekre pedig a #\ után írt névvel lehet hivatkozni.

Például: `#\tab` (tabulátor), `#\newline` (új sor).

A sztring karakterek listájaként értelmezendő. Sztring viszont sztringbe nem ágyazható. A sztring alakja:

"[*karakter*]…"

A karakterek és sztringek ugyanúgy lehetnek elemei egy listának mint az atomok és a listák.

Példák: " " (üres sztring), "almafa".

```
>"sztring"
"sztring"
>#\a
```

a

A karakterek és sztringek kezelését és predikátumok és egyéb függvények segítik. Izelítőül néhány ezek közül:

stringp	igaz, ha paramétere sztring
characterp	igaz, ha paramétere karakter
alphanumericp	igaz, ha paramétere alfanumerikus karakter
upper_case_p	igaz, ha paramétere nagybetű
char_code	visszaadja a karakter ACSII kódját
code_char	visszaadja az adott kódú karaktert

length	megadja egy sztring hosszát
concatenate	összefűzi a sztringeket.

7.7. I/O

A LISP kezeli az implicit állományokat. I/O-ja az adatfolyam elven alapszik.

Alaphelyzetben a LISP mindig képernyőre írja a legutoljára kiértékelt S-kifejezés értékét.

Az alapértelmezett adatfolyam nevek:

standard-input	alapértelmezett bemenet (billentzűzet)
standard-output	alapértelmezett kimenet (képernyő)
terminal-io	a felhasználói terminál
query-io	a felhasználói interakciók javallott adatfolyama
debug-io	az interaktív belövés adatfolyama
trace-output	a trace makró kimenete
error-output	hibaüzenetek

Az adatfolyamok használata természetesen makrók segítségével történik. Az írásnál a formátumos technika alkalmazható.

7.8. A kiértékelés vezérlése

A LISP alaphelyzetben azt mondja, hogy egy függvény visszatérési értéke a törzset alkotó S-kifejezések közül a legutolsó értéke lesz. Most megismerünk néhány olyan eszközt, amely a szekvencionális kiértékelés megváltoztatását célozza.

A prog1 függvény visszatérési értéke az első paraméterének értéke, a prog2 függvényé pedig a második paraméterének értéke. A progn függvény az alapértelmezett viselkedést mutatja.

Példa:

```
>(prog1 (setf n 3) (setf n (1+ n)))
3
> n
> n
```

A LISP tartalmaz *iteratív* eszközöket, ezek határozottan imperatív jellegűek.

A do makró egy kezdőfeltételes ciklust realizál, alakja a következő:

(do ([(*változó* [*kezdőérték* [*új_érték*]])]…)
(*feltétel* [S-kifejezés]...)
[*S-kifejezés*]...)

A do makró paraméterei három csoportba sorolhatók. Az első paramétere egy maximum háromelemű listákból álló lista. Ez egy inicializációs rész. A *változó* a makró lokális változója, kezdőértéke *kezdőérték*, vagy ennek hiányában NIL. Ezek a változók a ciklusváltozók. A harmadik paramétercsoportban szereplő S-kifejezések alkotják a ciklus magját.

A második paraméter egy legalább egy elemű lista. A *feltétel* nem NIL volta mellett fut le a mag (kezdőfeltétel). A do makró visszatérési értékét a második paraméter határozza meg. Ha a listában csak a *feltétel* szerepel, akkor a visszatérési érték NIL.

Ha az első paraméternél a listák háromeleműek, akkor minden ciklusléptés után a ciklusváltozók értéke felülíródik az *új_érték* értékével.

Példák:

1. A faktoriálist kiszámoló függvény:

```
(defun fakt(n)
        (do     (eredmeny 1 (* szamlalo eredmeny)
                (szamlalo n) (1-szamlalo)))
                ((zerop szamlalo) eredmeny>
```

2. Határozzuk meg egy számokat tartalmazó nemüres lista elemeinek átlagát.

```
(defun atlag(l)
        (do     ((v l (cdr v))
                (szamlalo 1 (1+ szamlalo))
                (osszeg (car v) (+ (car v) osszeg)))
                ((null (cdr v) (/ osszeg szamlalo>
```

3. Adjuk meg egy lista hosszát meghatározó függvény rekurzív és iteratív változatát.

   a.
```
(defun r-length (l)
        (cond (l null l) 0)
                (T (1+ (r-length (cdr l>
```

   b.
```
(defun i-length (l)
        (do ((ll l (cdr ll))
                (eredmeny 0 (1+ eredmeny)))
                ((null ll) eredmeny>
```

A végtelen ciklust valósítja meg a loop makró, melynek paraméterei között szerepelnie kell return függvény meghívásának. A return egyetlen paraméterének értékével tér vissza, vagy

paraméter nélkül a NIL-t adja. Ha nem adtuk meg a return függvényt, akkor a paraméterek (mint ciklusmag) újra és újra kiértékelődnek.

Példa:

Írjuk át az átlagszámító függvényt loop segítségével

```
(defun loop_atlag(l)
        (let    ((szamlalo 0)
                (osszeg 0))
        (loop (cond ((null l (return (/osszeg szamlalo)))
                (T (setf osszeg (+osszeg (car l)))
                        (setf l (cdr l))
                        (1+ szamlalo>
```

A CLOS ismeri a *blokk* fogalmát is, ezt a block makró segítségével kezelhetjük, ennek alakja:

(block *név* [*S-kifejezés*]...)

Egyrészt tekinthető egy egyszerű, megnevezett kifejezéssorozatnak, amikor visszatérési értéke az utolsó S-kifejezés értéke. Másrészt a paraméterei között szerepelhet (return-from *név érték*) függvényhívás, ami megadja a visszatérési értéket.

A prog makró alakja:

(prog ([{*változó* | (*változó* [*kezdőérték*])}]…)
[{*címke* | *S-kifejezés*}]…)

A prog makró imperatív jellegű. A *változó* a makró lokális változója, explicit kezdőérték adható neki, automatikus kezdőértéke NIL. A *címke* egy szimbolikus atom, az S-kifejezések között szerepelhet egy (go *címke*) alakú makróhívás, ami egy GOTO-utasításnak felel meg. Használhatjuk a return függvényt is.

Példa:

```
(defun fakt(n)
        (prog ((k n) (l 1))
        kovetkezo
                (cond ((zerop k) (return l)))
                (setf l (* k l)
                (setf k (1-k))
                (go kovetkezo>
```

7.9. Makrók

A LISP nyelv kiterjeszthető. A programozó újradefiniálhatja a nyelv eszközeit és új eszközöket adhat hozzá a nyelvhez.

Bármelyik függvény nevét tetszőlegesen megváltoztathatjuk a számunkra használhatóbb, ismertebb elnevezést vezetve be.

Példák:

```
(defun fej (x) (car x))
(defun farok (x) (cdr x))
```

A saját függvények definiálása az eddigi eszközrendszert új, a többiektől megkülönbözhetetlen eszközökkel bővíti.

Ennek következménye az, hogy bármely LISP verzió testreszabható, illetve bármely más verzióba átírható.

Az igazi nyelvkiterjesztő eszközök azonban a saját makrók.

Egy saját makrót a defmacro makró segítségével hozhatunk létre. Használatának formája egyébként teljesen azonos a defun használatával.

Egy függvény és egy makró között az alapvető különbség a paraméterek kiértékelésében van.

A függvénynél először kiértékelődik az összes paraméter, majd kiértékelődik a törzs. A makrónál először kiértékelődik a törzs, majd kiértékelődik az eredményül kapott új törzs.

Egy makródefinícióban elkerülhetetlen a ‘ használata. Segítségével egy lista *részleges kiértékelését* tudjuk megvalósítani, ugyanis csak a lista azon eleme kerül kiértékelésre, amely előtt vessző szerepel, a többi nem. A vessző használata nélkül hatása azonos a quote hatásával.

Példák:

```
>’atom
ATOM
>’(a b c)
(A B C)
> ’(a b c)
(A B C)
> ’(setf x (* 3 7))
(SETF X (* 3 7))
> ’(setf x ,(* 3 7))
(SETF X 21)
> ’(setf x ,(car ’(, (+ 3 4) (– 7 3) (* 5 7))))
(SETF X 7)
```

Ugyancsak a ‘ esetén alkalmazható a @, amely egy lista külső zárójeleit elhagyva, a lista elemeinek sorozatát adja vissza.

Példa:

```
> ‘(setf x (*, @(cdr ‘((+ 3 4) (– 7 3) (* 5 7)))))
(SETF X (* (– 7 3) (* 5 7)))
```

Nagyon sok probléma csak makrók segítségével oldható meg, a CLOS sok beépített makrót tartalmaz (jó néhányat már láttunk közülük).

Vizsgáljuk meg a függvény és a makró alkalmazása közötti különbséget.

Definiáljuk a veremből való olvasást, mint függvényt. A vermet most képzeljük el úgy, mint egy olyan listát, ahol a verem tetején az 1. elem van.

```
(defun pop (verem)
        (prog1 (car verem)
                (setf verem (cdr verem>
```

Hívjuk meg.

```
> (setf x '(1 2 3 4))
(1 2 3 4)
> (pop x)
1
> x
(1 2 3 4)
```

Valami probléma van. Igen, mert a függvény csak a lokális verem értékét módosította, amelynek kezdőértéke a meghívásnál x értéke volt, de a globális x nem változott meg.

Nézzük most ugyanazt makróval.

```
(defmacro pop(verem)
        '(prog1 (car , verem)
        (setf , verem (cdr , verem>
```

A makró paramétere nem értékelődik ki, a paraméterátadás név szerinti. Meghívásánál a globális változó csak a törzsben értékelődik ki, egyszer.

A függvény a LISP-ben adatobjektum, átadható paraméterként, a makró azonban nem.

Amikor a LISP kiértékel egy olyan listát, amelynek feje egy szimbólum, akkor a következőképpen jár el:

1. Ha a szimbólum egy foglalt szó, akkor a hozzátartozó kód alapján történik a lista kiértékelése. A CLOS foglalt szavai a következők:
   ⋮
2. Különben, ha a szimbólum egy makró neve, akkor végrehajtja a makrót és kiértékeli az eredményt.
3. Különben a szimbólumot függvénynévnek tekinti.

Példák:

1. Írjunk makrót, amely megcseréli két paraméterének értékét.

```
(def macro csere (x y)
        '(let | (z , x))
        (setf ,x ,y)
        (setf ,y z>
```

2. Írjuk meg azt a makrót, amely megadott számszor kiértékel egy S-kifejezés sorozatot.

```
(def macro ismetel (n rest mag)
```

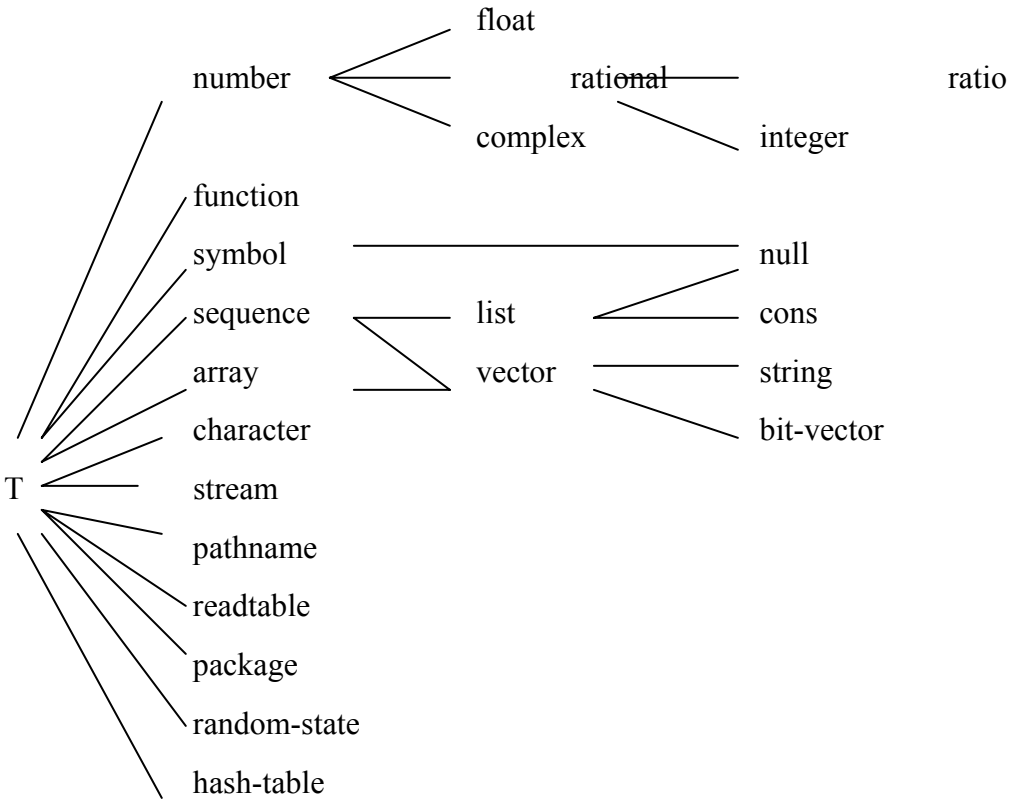
```
,(do ((i ,n (– i 1)))
        ((<= i 0) nil)
        ,@mag>
```

7.10. Objektumorientált eszközök

A legtöbb LISP változat a funkcionális paradigma mentén épül föl, azonban a CLOS egy hibrid nyelv, amely tartalmaz objektumorientált eszközrendszert.

A CLOS-ban nincs láthatóságszabályozás. Az öröklődés többszörös. A nyelv tartalmaz egy beépített osztályhierarchiát, a programozó viszont önálló osztályhierarchiákat hozhat létre, a beépített osztályok nem örököltethetők.

A beépített osztályhierarchia a következő:



A korábbi LISP verziók nem típusosak és az eddigi tárgyalásunkban ez tükröződött. Azonban a CLOS, a beépített osztályhierarchia elemeit típusoknak tekinti és a programozó által definiált osztályok is egy-egy típust képviselnek.

Saját osztály létrehozására a defclass makró szolgál, használatának alakja:

(DEFCLASS *név szuperosztály_nevek*
    attribútum_specifikációk
    osztály_opciók)

A *név* egy szimbólum, az osztály (típus) neve. A *szuperosztály_nevek* egy létező osztálynevekből álló (esetleg üres) lista.

Az *attribútum_specifikációk* alakja:

([*név* [*kulcsszó* [*érték*]]…]…)

A *név* az attribútum neve, a *kulcsszó* az alábbiak valamelyike:

:reader	az attribútum lekérdező módszerének a neve
:writer	az attribútum beállító módszerének a neve
:accessor	egy olyan módszer neve, amellyel az attribútumot egyaránt le lehet kérdezni és le lehet állítani
:allocation	értéke :instance (ez az alapértelmezett), vagy :class lehet. :instance esetén az adott attribútum értéke példányonként különbözhet, :class esetén viszont azonos (vagyis akkor ez egy nevesített konstans attribútum).
:initarg	az attribútum kezdőértéke, ami a példányosításkor beállításra kerül.
:initform	az attribútum alapértelmezett értékr, az :initarg felülírja, ha meg van adva.
:type	az attribútum típusa.
:documentation	értéke egy sztring, ez egy dokumentációs megjegyzés.

Látjuk tehát, hogy a CLOS-ban az attribútumokhoz kapcsolódóan adhatók meg a beállító és lekérdező módszerek nevei, ezek automatikusan létrejönnek. Az osztályhoz kapcsolódó további függvényeket, makrókat az osztálydefiníciótól függetlenül kell létrehozni.

Az *osztály_opciók* a következők lehetnek:

:default_initargs értéklista	az attribútumok kezdőértékét állítja be, az egyes attribútumokból megadott :initarg felülírja ezt.
:metaclass osztálynév	a CLOS-ban az osztályok alapértelmezés szerint a standard_class metaosztály példányai. Ezzel az opcióval egy ettől különböző metaosztályt adhatunk meg.

Egy osztály példányosítása a make_ms_tance függvény segítségével történik, ennek alakja:

(MAKE-INSTANCE *osztálynév* [*kezdőérték*]...)

A CLOS lehetővé teszi *generikus függvények* használatát. Ezeknél külön definiálhatjuk a különböző típusú argumentumok esetén a működést. Ezeket a különböző definíciókat a CLOS *metódusoknak* nevezi (nem tévesztendő össze az OO metódussal!). Egy generikus függvényt a defgeneric makró segítségével tudunk létrehozni. Alakja:

(DEFGENERIC *név formális_paraméter_lista*
(:METHOD *specializált_formális_paraméter_lista törzs*)
[(:METHOD *specializált_formális_paraméter_lista törzs*)]…)

A *specializált_formális_paraméter_lista*

(*formális_paraméter* [*típus*])

alakú listákból álló lista. A *törzs* az adott típusú formális paraméterekre történő működést írja le. Ha nem szerepel a *típus*, akkor a működés tetszőleges típusra vonatkozik.

Egy generikus függvényt (függetlenül az esetleg nem is ismert definíciótól) mindig kiegészíthetünk új metódussal a defmethod makró segítségével. Ennek alakja:

(DEFMETHOD *generikus_név specializált_formális_paraméter_lista törzs*)

A generikus függvények használata alapvető az osztályok esetén.

Példák:

```
(defclass szemely( )
        ((nev :initarg :nev :reader szemely_neve)
        (eletkor :initform 0 :accessor szemely_kora))
        (:documentum “Önálló osztály”))
```

A szemely osztálynak nincs szuperosztálya. Két attribútuma van, ezek közül a nev csak olvasható, a kor írható-olvasható.

```
(defun szemely_kons (nev)
        (make-instance ’szemely :nev nev))
```

Ez a függvény az osztály konstruktorának tekinthető. A kor alapértelmezett értéke 0.

```
> (defvar x (szemely_kons ‘KISS))
#<SZEMELY @#x11bca2e> ;implemetációfüggő’
> (szemely_kora x)
0
> (szemely_neve x)
KISS
> (setf (szemely_kore x) 22)
> 22
```

Adjunk meg egy predikátumot, amely eldönti, hogy paramétere a szemely osztály példánya-e.

```
(defgeneric szemelyp (obj)
        (:method ((obj szemely)) T)
        (:method (obj) NIL))
> (szemelyp x)
T
> (szemelyp ’x)
NIL
```

A CLOS tartalmaz egy print-object nevű generikus függvényt, amely az objektumok megjelenítését szolgálja. Ezt bármikor kiegészíthetjük saját objektumaink megjelenítésének metódusaival.

```
(def method print-object ((obj szemey) stream)
        (format stream "#<SZEMELY:~A (Kora: ~A)>"
                (szemely-neve obj) (szemely-kora obj>
> x
#<SZEMELY: KISS (Kora> 22)>
```

Hozzuk létre a szemely egy alosztályát.

```
(defclass programozo (szemely)
        ((nyelvek :initform NIL initarg :nyelvek
                :accessor ismert))
        (documentation "Ez egy alosztály"))

(defnn programozo-konstr (nev nyelvek)
        (make-instance 'programozo :nev nev
                :nyelvek nyelvek))
```

A CLOS-ban az objektumok futás közben, **dinamikusan** meg tudják változtatni a struktúrájukat és a viselkedésmódjukat!

Tekintsük a korábbi `szemely` osztály definícióját és definiáljuk azt át a következőképpen:

```
(defclass szemely( )
        ((nev :initorg :nev :reader személy_neve)
        (eletkor :initform 0 :accessor személy_kora)
        (munkakor:initform 'Programozo :accessor munkakor :allocation :class)))
```

Ekkor

```
> x
#< SZEMELY: KISS (Kora: 22)>
> (munkakor x)
#< SZEMELY: NAGY (Kora: 0)>
> (munkakor y)
PROGRAMOZO
> (setf (munkakor x) 'Kereskedo)
KERESKEDO
> (munkakor y)
KERESKEDO
```

A CLOS *automatikusan* a régi osztály példányait az új osztály példányaivá, alakítva a struktúrájukat és az új viselkedésmóddal **látva el** őket. KISS-nél megjelenik a `munkakor` attribútum, amely lekérdezhető és beállítható. Ez az attribútum egy megosztott attribútum, amelyet az egyes példányok közösen birtokolnak.

A CLOS-ban az osztályok a metaosztályok példányai. Három beépített metaosztály van:

- `built-in-class` : a beépített osztályok metaosztálya
- `stadard-class` : a `defclass` makróval definiált saját osztályok metaosztálya
- `structure-class` : a `destruct` makróval definiált *rekordok* metaosztálya (nem foglalkozunk vele).

Egy osztály metaosztályát a `class-of` segítségével kérdezhetjük le.

A CLOS metaobjektumainak osztályai a következők:

- `standard-method` : a `defmethod` és `defgeneric` segítségével definiált metódusok osztálya
- `standard-generic-function`: a generikuis függvények metaosztálya
- `standard-object`: `standard-class`, `standard-method`, standard-generic-function szuperosztálya

A CLOS-ban lehetőség van a metaosztályok kiterjesztésére a *metaobject protocol* (MOP) segítségével. Ez a témakör meghaladja jelen jegyzet kereteit, részletesen l. … .

7.11. Általánosított függvények

Az apply függvény a LISP egy általánosított függvénye, alakja:

(APPLY [*függvény lista*]…)

A *függvény* rendre meghívódik a további paraméterekként megadott listák mindegyikére.

Példa: Adjuk meg az `append` rekurzív definícióját.

```
(defun append2 (birtok1 lista2)
        (cond ((endp lista1) lista2)
        (T (cons (car lista1) (append2
                        (cdr lista1) lista2>

(defun append (&rest listak)
        (cond   ((endp listak) 'L))
                ((endp (cdr listak) (car listak))
                ((endp (cddr listak) (append2
                                (car listak) (cadr listak)))
                (T (append (car listak) (apply
                                #'append (cdr listak>
```

Az `apply` a procedurális absztrakció magas absztrakciós szintjét biztosítja. Például segítségével megírhatunk egy általános leválogató függvény, amely egy lista elemei közül azokat írja át egy másik listába, amelyek egy predikátumot igazzá tesznek. A függvény:

```
(defun levelogat (lista predikatum)
        (cond ((null lista) NIL)
        ((apply predikatum (Lista (car lista)))
```

```
(cons (car lista) (levalogat
                  (cdr lista) predikatum)))
(T (levalogat (cdr lista) predikatum>
```

Ennek segítségével egy csak számokat tartalmazó listából a negatívokat a következő függvénnyel tudjuk leválogatni:

```
(defun negativ (lista) (levalogat lista 'minusp))
```

8. Logikai nyelvek: Prolog

A logikai nyelvek legjelentősebb képviselője. '72-ben születik meg. A szoftverkrízisre adott egyfajta válasz, irányzat. Franciaországban születik. A ’80-as években több logikai nyelv születik. Mindegyiknek az alapja a matematikai logika valamely irányzata. Az elsőrendű predikátumkalkulusra épül fel a Prolog. A Prolog vissza fog köszönni a mesterséges intelligenciakutatásban, alapvető szerepet játszik. Az adatbázis-kezelő rendszerekben is felbukkan, pl. deduktív adatbázis-kezelő rendszerekben is fontos. Magyarországon a '80-as években kifejlesztenek egy Prolog rendszert, az MProlog-ot. Ez azon kevés termékek közé tartozik, amely szerepet játszik a világon (a másik az Ada). A Japánok az ötödik generációs gép nyelveként ezt választották.

A logikai programozás szemlélete:

- A Prolog nem típusos nyelv.
- Karakterkészlete a szokásos.
- A Prolog elemi objektumai (alap építőelemei) a következők:
  - Azonosítók: nem azonos az imperatív nyelvek azonosító fogalmával, hanem tetszőleges karaktersorozat lehet. Nincs olyan gond, hogy magyarul írom vagy nem.
  - Numerikus konstansok: A szokásosak.
  - Karakterlánc (sztring), idézőjelben áll.
  - Elhatároló jelek: pl. a kerek zárójelek, idéző jelek ...
- Éles különbséget tesz kis és nagybetűk között.
- A Prolognak is van változó fogalma. A változónak van neve, amely egy speciális azonosító, amely nagybetűvel vagy aláhúzásjellel kezdődik, és betűvel vagy számjeggyel folytatódhat. Nincs attribútuma. Van értéke, de: egy változó két állapotban lehet:
  - nincs értékkomponense, ekkor lehet értéket adni neki
  - van érték komponense, nem lehet ekkor értéket adni neki, a változó értéke nem felülírható. Nem létezik az `s:=s+1` utasítás. Ez az egyszeres értékadás szabálya. Át kell vinni abba az állapotba, hogy ne legyen neki értéke, és csak ekkor adhatok értéket.

Deklaratív jelleg:

A változó értékének a beállítása elsősorban a rendszer feladata. Bizonyos esetekben a programozó is beállíthatja a változók értékét, de ez az eszközrendszer minimális. A változónak címe van, de ehhez a programozó nem férhet hozzá. Van még egy érdekessége a változónak: szemben az imperatív szemlélettel a teljes szövegben ugyanazt az értéket jelenti, ha egyszer értéket kapott.

A deklaratív nyelvek általában szimbolikus nyelvek, ami a következőt jelenti: ha egy imperatív nyelvben leírtam egy változó nevét, akkor az általában a címet vagy értéket

jelentette, elvétve a típust vagy nevet. A deklaratív nyelvekben `Xyz` vagy az objektum neve, vagy attól függően, hogy milyen szövegkörnyezetben van azt a szimbólumsorozatot jelenti, amit leírtam: `Xyz`.

A Prologban: a változó csak a nevet, mint karaktersorozatot jelenti, ha nincs értéke a változónak. Ha van értéke, akkor vagy szimbólumsorozat, vagy a változó értékét jelenti. Ha Prologban programot írok, akkor a matematikai logikában kell gondolkodnunk: meghatározom azt a környezetet, ahol le akarom futtatni.

Alapvető eszközök:

- a tények
- a szabályok
- a feladat.

A tények igaz értékű állítások. A tények alakja formálisan:

```
név(argumentumok).
```

- `név`: azonosító, mint elemi objektum
- `argumentumok`: `()`-ben, egymástól vesszővel elválasztva. Egy argumentumnak illik lennie, több lehet. Szemléletében tények megadásakor az argumentumokra vonatkozó igaz állításokat sorolom fel. Az argumentumok összetett objektumok lehetnek. (összetett objektumok fogalma ld. rövidesen)
- ponttal zárja le

A Szabályok: az elsőrendű predikátumkalkulus következtetési szabályai, logikai formulák. Van a szabályoknak feje és törzse. Alakja:

```
fej:-törzs.
```

- A `fej` szerkezete megfelel a tények szerkezetének.
- A `törzs` pedig egy feltételsorozat, vesszővel elválasztva. Ezek a feltételek éssel vannak összefűzve. A feltételek összetett objektumok (majd beszélünk róla).
  Ez attól egy következtetési szabály, hogy ha igaz a törzs, akkor igaz a fej.
  A `tények` törzs nélküli szabályoknak tekinthetők.

A Feladat vagy kérdés: A rendszer a megadott környezetben keresi azokat az objektumokat, amelyek igazzá teszik a kérdést. Formálisan a kérdés úgy néz ki, mint egy fej nélküli szabály:

```
:-feltételsorozat.
```

A szabályok és tények az imperatív szemlélet szerint alprogramoknak tekinthetők: elsősorban eljárásnak.

A Prolog programozás nem más, mint megadjuk először a tényeket, majd utána a szabályokat, ezzel definiálom a megoldandó feladat környezetét. Végül feltesszük a kérdést:

```
:-feltételsorozat.
```

Keresse meg a rendszer azokat a megoldásokat, amelyek ezt a feltételsorozatot igazzá teszik.

A Prolog a feladatot mintaillesztéssel és backtrack-kel oldja meg. Alapvető a felsorolás sorrendje: tények, szabályok, feladat.

Nézzünk egy feladatot:

```
apja(János, ferenc).
apja(ferenc, péter).
```

Itt van két tény, mindkettőnek két argumentuma van, két azonosító. Ilyen értelemben szimbólumsorozat. Ami mögötte van, igaz állítások, a Prolog szempontjából tények.

Nézzünk egy következtetési szabályt:

```
nagyapja(X,Z) :- apja(X,Y), apja(Y,Z).
```

`X`, `Y`, `Z` változók. Logikailag annyit jelent, hogy `X` nagyapja `Z`-nek akkor, ha `X` apja `Y`-nak, és `Y` apja `Z`-nek.

Felteszünk egy kérdést:

```
:-nagyapja(Valaki, péter).
```

Keressük azon objektumokat, amelyekre igaz, hogy mindegyikük `péter` nagyapja.

Hogyan válaszolja meg a Prolog ezt a kérdést?

Mint már említettük: mintaillesztéssel: a karaktersorozatokra vonatkozóan. Abszolút a szimbolikus szinten vagyunk.

A következőkképpen jár el a Prolog rendszer:

- Nekiesik a feladatoknak, és veszi a kérdés első feltételét, mint karaktersorozatot
- Ezek után veszi a tényeket a felírás sorrendjében és a feladat első feltételét próbálja illeszteni a tényekhez. Először az első tényhez. (Jelen esetben nincs illeszkedés.)
- Ha nem sikerül karakterről karakterre illeszkedést találni, akkor megnézi a rendszer, hogy van-e változó. Ha van változó, akkor a rendszer értéket ad a változónak. Most már nem a nevével, hanem az értékével hasonlít a rendszer. Két válasz lehetséges:
  - Ha talál illeszkedést, a feltételt a változó helyettesítésével olyan formára hozta, hogy megegyezik a ténnyel. A rendszer a feladat törzséből törli az adott feltételt. Megyünk tovább a második feltétellel.
  - Ha semmilyen változó helyettesítéssel nem sikerült ténnyel egyezést találni, veszi a szabályokat a felírás sorrendjében, és az illesztést a szabály fejével játssza el. Két eset lehetséges:
    - Nem talál egyezést. A rendszer megnézi, hogy van-e változó. Ha van változó, akkor a rendszer értéket ad a változónak, és a kapott értékkel hasonlít.
    - Talál egyezést, azaz a szabály feje megegyezik a feltétellel. Ez a feltétel igaz, ha a megfelelő szabály törzse igaz. Az adott feltételt helyettesíti a feladatban az illeszkedő szabály törzsével.

  Ha a ténnyel illeszthető, akkor csökken a törzs. Ha szabállyal illeszthető, akkor nőhet a törzs.
- Ha elfogynak feltételeim, kiürül a feladat törzse, megvan az első megoldás. Minden eredeti feltételt sikerült

igazzá tennem. A kérdésre a válasz igaz, és a megoldást a feladatban szereplő változók értéke adja.

Feladatunk esetén:

```
X ← Valaki
Z ← péter
```

Találtam egy szabályfejet, ami közvetlenül nem egyezik meg, de a változó egy helyettesítésével igen. Be kell másolnom a szabály törzsét, úgy, hogy a szabály törzsben szerepelnek azok a változók, amelyeknek értéket adtam. Makrózás.

```
:-apja(Valaki,Y), apja(Y,péter)
```

-vé alakul a feladat törzse. Vesszük az első feltételt: közvetlenül nem illeszkedik az első tényhez, de változó helyettesítéssel igen.

```
Valaki ← János
Y ← ferenc
```

Így az illeszkedés fennáll. Találtam egy illeszkedő tényt, törlöm az első feltételt. A feladat törzsében marad:

```
:-apja(ferenc, péter)
```

Az elsővel nem egyezik, a másodikkal igen. `:-` . Kiürült a feladat törzse. A megoldás `János`. Tehát `János péter` nagyapja.

Ez az első megoldás. Mi van, ha több megoldás érdekel?

Mi van, ha felcserélem a két tény sorrendjét? Ekkor nem tudjuk kiüríteni a feladat törzsét, zsákutcába jutunk.

Mi van akkor, ha a feladat törzse nem ürül ki ? Ekkor jön a backtrack.

Backtrack(visszalépés):

Törli az utolsó illesztés hatását, ha az utolsó illesztésnél volt változó helyettesítés, akkor a változó értékét megszünteti, visszalép, és próbál új illesztést keresni. Ehhez, mint bármely illesztéses algoritmushoz hozzá lehet rendelni egy illesztési fát. Tegyük fel a

```
:-nagyapja(A,B).
```

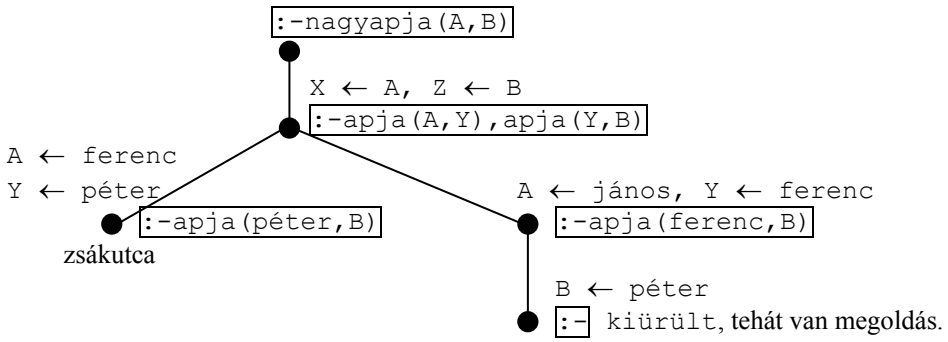
kérdést az előző környezettel. Tehát:

```
apja (János, ferenc).
apja (ferenc, péter).
nagyapja(X,Z) :- apja(X,Y), apja(Y,Z).
:-nagyapja(A,B)
```

Mikor mondja a Prolog, hogy megvan az összes megoldás?

Ekkor az illesztési fa a következőképpen néz ki:

- gyökérelem az induló kérdés
- éleit címkézzük a változó helyettesítésekkel
- a további csomópontokban ott szerepel a kérdés aktuális állapota egy illesztés után



További megoldások keresése visszalépéssel (backtrack).

Azon levélelemeknél van megoldás, ahol a feladat törzse kiürült, egyébként zsákutcába futottunk. Az adott úton szereplő változók értékei képezik a feladat megoldást: `A ← János`, `B ← péter`. Ez a megoldásfa nem létezik eleve, fel kell építeni, majd utána bejárni. A Prolog nem egyenlő a fabejárással. Az összes megoldást akkor találjuk meg, ha a backtrack véget ér.

Alapvetően a Prologban a szabályok rekurzívak.

Összetett objektumok (kifejezések)

- A kifejezés operandusokból, operátorokból áll. Viszont a Prolog operandusainak a köre jóval szélesebb, mint az imperatív nyelveké. Van itt nulla-, egy-, kétoperandosú és többoperandusú operátor.
- Az operandus lehet elemi objektum vagy változó.
- Az operandusok és bizonyos operátorok együtt összetett objektumokat (kifejezéseket) alkotnak. Az összetett objektumok a kifejezések.
- A Prolog tudja kezelni a kifejezés mindhárom alakját: a pre-, in- és postfix alakot.
- A feltétel nem más, mint egy kifejezés.
- Kiértékelésnél a balról-jobbra szabály érvényes.

- A Prolog rendszer ismerete a beépített operátorok ismeretét jelenti.
- A programozó is definiálhat saját operátort.
- A Prolog beépített operátorainak a egy része az imperatív nyelvekből ismert operátorokkal egyezik meg. Vannak:
  - aritmetikai operátorok: `+`, `-` , `*`, `/`, `div`, `mod`. Ezek bizonyos szituációkban a megszokott módon viselkednek (aritmetikai operátorok), néha szimbolumként.

    Például: létezik egy `IS` kétoperandusú operátor, melynek baloldalán állhat egy változó neve, jobboldalán pedig egy a fenti operátorok segítségével felépített kifejezés áll. Ekkor ez egy aritmetikai operátor:

    - Ha a változónak nincs értéke, akkor a változó felveszi az `IS` jobboldalán álló aritmetikai kifejezés értékét. Ez azon ritka esetek közé tartozik, amikor a programozó állítja be egy változó értékét (egyszer!).
    - Ha a változónak van értéke, akkor illesztés történik az `IS` operátor két oldala közt, amire a válasz logikai értékű lesz: igaz vagy hamis.
  - szövegkezelő operátorok
  - I/O operátorok
  - tudásbázis kezelő része is van a Prolognak (deduktív adatbázis-kezelő), van kivételkezelője is, és grafikai része is van
- Az adatszerkezetek közül a Prolog programok kezelik a:
  - a listát és
  - az egydimenziós tömböt.

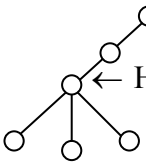
Például:

```
páros_szám(N) :- 0 IS N mod 2.
```

Ha nem változónév áll a baloldalon, akkor egyértelműen illesztés történik.

A backtrack technikát (kiértékelést) tudjuk befolyásolni a Prologban a vezérlésátadó operátorokkal. Ilyen például a vágó, vagy vágási operátor. Jele: `!` jel.

Például, ha csak arra vagyok kíváncsi, hogy egy feladatnak van-e megoldása vagy sem, akkor célszerű ezt az operátort használni. Ha valahol találkozik a `!` operátorral kiértékelés közben, akkor elvágom a fát, és ezen felül a backtracket leállítom, az alsó részfára korlátozom.


← Ha itt elvágom, a backtrack nem lép vissza az első és második szintre.

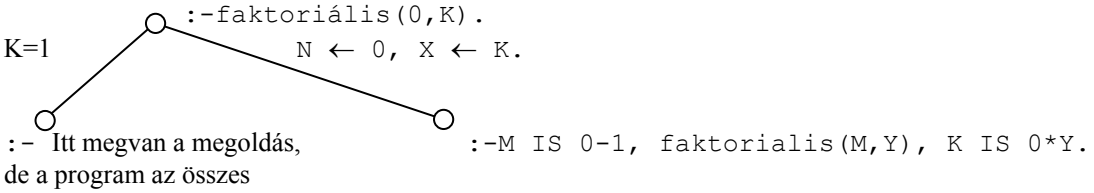
Példa:

```
faktoriális (0,1).

faktoriális (N,X) :-M is N-1, faktoriális (M,Y), X is N*Y.
```

Amit eddig feltételnek hívtunk, a kifejezés. Itt egy rekurzív szabályt láthatunk. Nézzük meg, hogy ilyen környezetben milyen problémák adódhatnak.

– Mennyi a `0!`?



`:-M is 0-1, faktoriális(M,Y), K IS 0*Y.` esetén a nullát kezdi el csökkentgetni, így végtelen rekurzió alakul ki, mert megengedtem, hogy a program belemenjen olyan ágba, amibe nem szabadott volna. Itt a `!` operátor szükségessége.

Javítva tehát:

```
faktoriális (0,1):-!.

faktoriális (N,X) :-M is N-1, faktoriális(M,Y), X is N*Y.
```

– Még mindig probléma van a:

```
: -faktoriális(0,2).
```

esetben. Megint végtelen a rekurzió.

A jó megoldás:

```
faktoriális(0,1):-!, X is 1.

faktoriális(N,X):-M is N-1, faktoriális(M,Y), X is N*Y.
```

9. ADATFOLYAM NYELVEK

Az OO paradigma pillanatnyi állapotában a legmesszebbre elviszi az újrafelhasználás és az absztrakció eszközrendszerét. A párhuzamosság kérdésére viszont nem ad választ. Az OO nyelvek egyrésze ismeri a párhuzamosságot, más része nem.

De van olyan nyelvcsalád, amely specialitásánál fogva „kilóg a sorból", amely választ ad a párhuzamosság kérdésére. Kb. kéttucatnyi nyelv tartozik ide. Ezek a Neumann architektúrát tagadják. Azt mondják, hogy a Neumann architektúra szűk keresztmetszete a tárkezelés illetve a szekvenciális működésű processzor. Egy `z=x+y` jellegű utasítás mögött sok gépi szintű utasítás áll. A tárból elő kell venni az `x`-szel illetve az `y`-nal jelölt adatot stb., mindez szekvenciálisan működik. Ezt totálisan tagadja. Más hardverplattformot igényel.

Azt mondja, hogy minden algoritmust párhuzamos algoritmusként kell tekinteni, és az algoritmust realizáló program pedig minden párhuzamosan végrehajtható kódot hajtson végre párhuzamosan. Totális párhuzamosságra kell törekedni. Ez a paradigma felerősíti a párhuzamos algoritmusok kutatásának elméletét. Gondoljunk a gyorsrendezésre: megírható szekvenciálisan, de párhuzamosan is, és ekkor az összes létező csoportra egyszerre hajtja végre a gyorsrendezést.

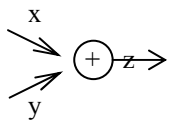
Minden szekvenciális algoritmus átírható párhuzamos algoritmussá. (Eddig csak szekvenciális algoritmusokról beszéltünk.)

Teljesen más világszemlélet. A kultúránk olyan, hogy szekvenciálisan látjuk a világot, így tanítják. Ld. az írás olvasás tipikus szekvenciális tevékenység. Holott az agyunk párhuzamos működésű.

Mint már említettük, minden szekvenciális algoritmus átírható párhuzamos algoritmussá. A szekvenciális algoritmusleírás egyik eszköze a folyamatábra. Ehhez hasonlóan az adatvezérelt paradigmának is van egy leíró eszköze: egy összefüggő, irányított gráf, amelynek van egy kitüntetett kezdőpontja, vannak csomópontok, amelyek tevékenységeket írnak le és lehet több végpontja (kimeneti pontok). Egy csomópontnak egy tevékenység felel meg (operátorok), a műveletek realizálására szolgálnak. Az egyes programnyelvekben kérdés, hogy mi az operátorkészlete.

Irányított a gráfról van szó, tehát a csomópontokhoz vezető- és a csomópontokból kivezető élek nyilak. Az élek mentén a nyilak irányában ún. tokenek, adatcsoportok mozognak (míg a folyamatábra statikus).

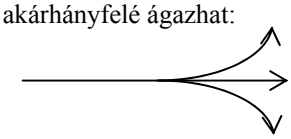
Algoritmus: van egy input token, amit a kezdőponton keresztül halad az operátor felé. Az operátor (csomópont) csak akkor kezd el dolgozni, amikor a bemenő élei mindegyikén megjelenik egy token. És ha rendelkezésre áll minden adat, azonnal dolgozik vele, és a kimenő nyilak mentén kiadja az eredményt. Ez az adatvezéreltség elve.



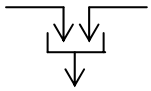
Az adatcsomagok tetszőleges adatszerkezetet reprezentálhatnak, és a műveletek tetszőleges bonyolultságúak lehetnek.

Vannak konvencionális jelölések a modellben:

– Egy él akárhányfelé ágazhat:



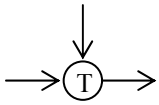
– Kapcsolat: a gráf különböző pontjaiból kapcsolódhatnak a tokenek. Élek nem végződhetnek élen, csomópontban kell végződjenek. Szükséges egy gyűjtő.



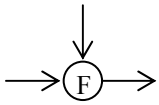
Speciális csomópontok:

– Feladata: egy logikai értékű tokent produkál egy vizsgálat és feltételkiértékelés után.

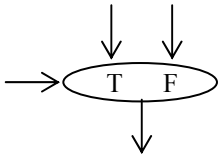
– Igazkapu: két bemenő- és egy kimenő éle van. A bemenő élek egyike egy logikai értékű token, a másik értéke tetszőleges. Ha a logikai token értéke igaz, akkor a másik tokent átengedi, egyébként nem lesz kimenet, a kapu lezár.



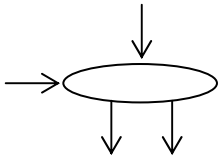
– Hamiskapu: hasonló az igazkapuhoz, annyi kulönbséggel, hogy hamis esetben engedi át a másik input tokent.



– Kiválasztó csomópont: az igaz- és hamis kapu kombinációja. Lényeges a szerkezet is: 2+1 bemenő- és egy kimenő éle van. A legelső bemenő él a vezérlő él, értéke egy logikai token. A két másik bejövő token közül kiválasztja, hogy melyiket engedi át. Ha a vezérlő token értéke igaz, akkor az első tokent teszi a kimenetre, míg ha hamis, akkor a másodikat.



– Elosztó csomópont: két bemenő tokent és két kimenő tokent tartalmaz. A bemenő tokenek közül az első, a vezérlő token csak logikai értéket hordozhat. Működése: ha a vezérlő token értéke igaz, akkor a bemenő adat token az első outputon jelenik meg, ellenkező esetben a másodikon. Egyetlen csomópont, aminek két kimenete van.



T F

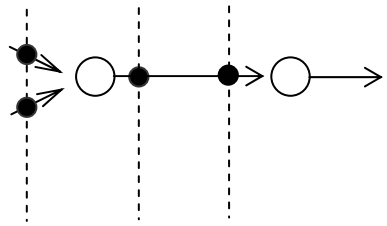
Ilyen elemekből építünk fel egy összefüggő irányított gráfot. Ezzel tudjuk leírni a párhuzamos algoritmusokat.

Az adatvezérelt paradigmán belül többféle modell létezik, attól függően, hogy milyen megszorításokat teszünk a gráfra.

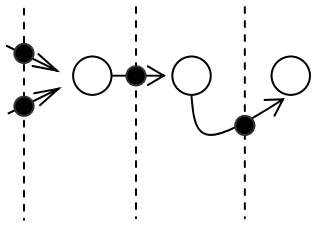
Modellek:

- Alapmodell:
  Determinisztikus, csak korlátozott párhuzamosságot enged meg.
  A következőket mondja:
  - Az egy input tokennel indított tokensorozat hullámfrontszerűen terjedjen végig a gráfon.
  - Később indított tokensorozat nem előzhet meg korábban indított tokensorozatot.
  - A hullámfront mintegy összeköti a tokeneket, nem hajolhat el, nem értelmezhető ciklus és rekurzió.

  Például:



- Denis-féle (MIT) modell:
  Megszorítás: követeljük meg, hogy gráfban egy csomópont csak akkor működhessen, ha nincs az output élen token. Két hullámfront között mindig legyen operátor. A hullámfrontot eltoljuk legalább egy csomóponttal. Így válik lehetővé ciklus létrehozása. Egy tokent visszaküldhetek, beküldhetem egy körútba, és addig nem engedem ki, amíg valamilyen feltételnek nem tesz eleget. Itt is igaz, hogy hullámfront nem előzhet meg hullámfrontot. De még mindig determinisztikus a modellem.
  Például:



– Színezett modell:

Jelenleg a legfejlettebb adatvezérelt programozási modell. Párhuzamos, így nem determinisztikus. Az egy inputtal indított tokensorozatot színezzük egy szinnel. A csomópont akkor működik, ha az azonos színű tokenek közül az összes input, azaz az összes bemenő élen azonos színű tokenek jelennek meg (addig pufferel). Adott szintű tokent ad ki. Értelmezhető az iteráció és a rekurzió.

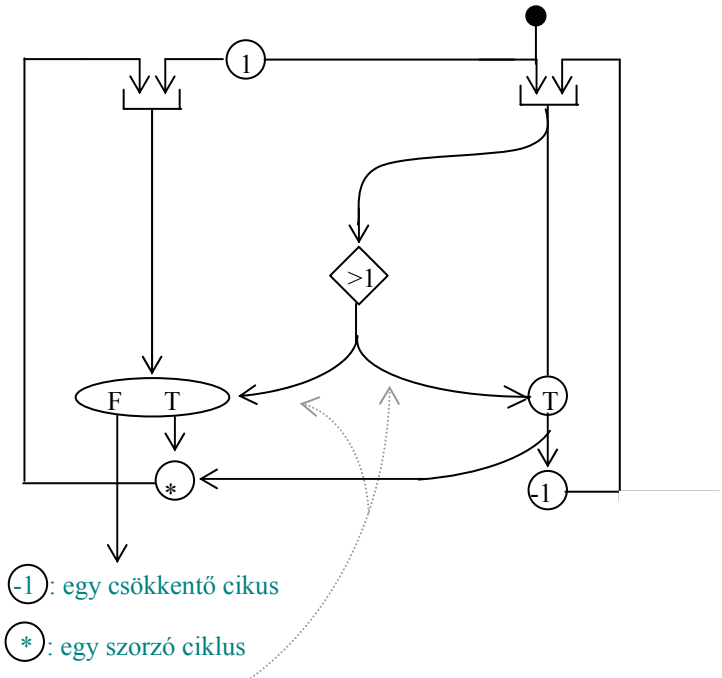
Nincs hullámfront. Ugyanazon feladatot megoldó gráfban az összes azonos jellegű feladat egyszerre megoldásra kerül.

Kérdés, hogy hol van az az architektúra, ahol ez a feladat leprogramozható és futtatható? Léteznek ilyen architektúrák prototípus szinten minden modell mögött.

Jellemzőjük:

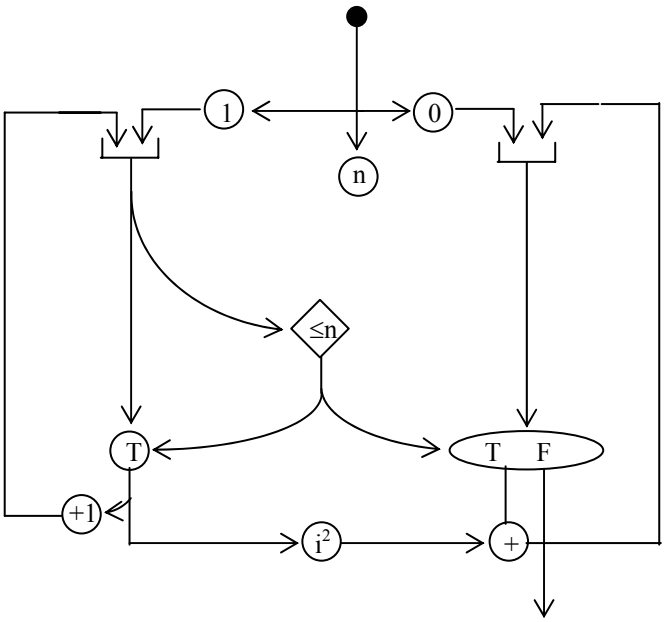
- asszociatív tár
- A program végrehajtásánál biztosítja a párhuzamosságot.
- A tokent kezelni tudja.
- Csomópontok leprogramozását lehetővé teszi.

Példa: Az n! szinezett gráfja hibavizsgálat nélkül.



Példa:

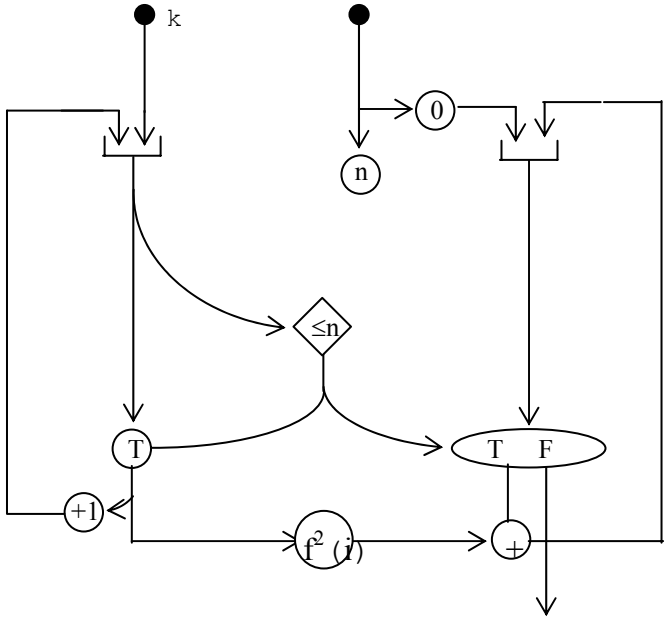
$$\sum_{i=1}^{n} i^2$$



Példa:

Az előző feladat általánosítása tetszőleges f függvény négyzetösszegére.

$$\sum_{i=k}^{n} [f(i)]^2$$



Az adatfolyam nyelvek, mint programozási nyelvek jellemzői

- Az általános csomópontokat általános függvénnyel reprezentálják. Ezeknek a függvényeknek nincs mellékhatásuk. A paraméterátadás mindig értékszerinti. Általában nem használnak globális változókat, de ha mégis, akkor az egyszeres értékadás érvényes. A változó értékének módosítása nem megengedett. Csak lokális adattér van.
- Általában a rekurzió alapeszköz.
- A programfejlesztés is nagyon egyszerű. Megírok egy primitív programot, és azt transzformálom. Az így fejlesztünk bonyolult programok biztosan jók. Egy token akármilyen bonyolult adatot ír le, akkor is egy tokennek számít.
- Gond: az architektúra hiánya. Ezen nyelveknek létezik a Neumann-architektúrán futó implementációja.
- A programhelyességbizonyítás automatikus, és nagyon egyszerű.
- Általában fordítóprogram-orientáltak, és a gépi kódjuk ez a gráf (grafikus gépi kód). Hordozható, hiszen ugyanazt a gépi kódot generálják. Ez a gráf automatikusan könnyen kezelhető.

Igényvezérelt programozás:

Kb. lassan 15 éve a programfejlesztés iránya: az adatvezérelt programozás kiterjesztése az igényvezérelt programozásra. Itt eltűnik az irányított gráf, a tokenek mindkét irányban mozoghatnak. Prioritások vezethetők be. Ez alá architektúra még prototípus szinten sem létezik. Ez még mindig kutatási szinten áll.

Adatvezérelt programnyelvek:

Néhány adatvezérelt programozási nyelv: VAL, LUCID, ID, LAU, SISAL, HDFL.

VAL: Denis féle modellt megvalósító nyelv.

```
    for Y:integer:=1; P:integer:=N
        do                  if P ≠ 1 then iter Y:=Y*P; P:=P-1
enditer
           else Y endif
    endfor
```

LUCID: Deklaratív plattformú.

Példa: `n!` kiszámítása:

```
    FIRST(i,j)=(n,1)
    NEXT(i,j)=(i-1,j*i)
    OUTPUT = j AS SOON AZ i=1
```