

Juhász István

MAGAS SZINTŰ PROGRAMOZÁSI NYELVEK 2

mobiDIÁK könyvtár

Juhász István

MAGAS SZINTŰ PROGRAMOZÁSI NYELVEK 2

mobiDIÁK könyvtár

SOROZATSZERKESZTŐ
Fazekas István

Juhász István

Magas szintű programozási nyelvek 2

Egyetemi jegyzet
Második kiadás

mobiDIÁK könyvtár

Debreceni Egyetem
Informatikai Kar

Lektor

Espák Miklós
Debreceni Egyetem

Copyright © Juhász István 2009

Copyright © elektronikus közlés mobiDIÁK könyvtár, 2009

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

A mű egyéni tanulmányozás céljára szabadon letölthető. Minden egyéb felhasználás csak a szerző előzetes írásbeli engedélyével történhet.

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

TARTALOMJEGYZÉK

ELŐSZÓ	8
1. AZ OBJEKTUMORIENTÁLT PARADIGMA	9
2. JAVA	14
3. SMALLTALK	15
3.1. Alapvető jellemzők	15
3.2. Literálok	16
3.3. Változók	16
3.4. Kifejezések	17
3.5. Blokk	19
3.6. Vezérlési szerkezet	19
3.7. Osztályok	20
3.8. Metódusok	21
3.9. Szálak	24
3.10. Kivételkezelés	25
4. EIFFEL	27
4.1. Lexikális elemek	27
4.2. Típusok	28
4.3. Változó	29
4.4. Kifejezések	29
4.5. Végrehajtható utasítások	31
4.6. Egy Eiffel program felépítése	33
4.7. Osztályok létrehozása	33
4.8. Objektum, érték, egyed	38
4.9. Példányosítás	39
4.10. Objektumok duplikálása és összehasonlítása	40
4.11. Típusok kezelése	40
4.12. Tömbök és sztringek	42
4.13. Programhelyesség	43
4.14. Kivételkezelés	45
4.15. I/O	47

5. A LOGIKAI PARADIGMA ÉS A PROLOG	48
6. A FUNKCIONÁLIS PARADIGMA	55
7. LISP	57
7.1. A CLOS alapelemei	57
7.2. CLOS beépített függvények	59
7.3. Nevesített konstans, változó, saját függvény	62
7.4. Listák	63
7.5. Lokális és globális eszközök	69
7.6. Karakterek és sztringek	70
7.7. I/O	71
7.8. A kiértékelés vezérlése	71
7.9. Makrók	73
7.10. Objektumorientált eszközök	76
7.11. Általánosított függvények	81
8. ADATVEZÉRELT PARADIGMA, ADATFOLYAM NYELVEK	83
IRODALOMJEGYZÉK	89

ELŐSZÓ

Jelen jegyzet a Debreceni Egyetem *Programtervező informatikus és Mérnökinformatikus* szakán alapozó tárgy, a **Magas szintű programozási nyelvek 2** elméleti anyagát tartalmazza. A tantárgy előfeltétele a **Magas szintű programozási nyelvek 1** tárgy, vele együtt alkotnak szerves egészet. Ez a jegyzet teljes egészében felhasználja az ott elsajátított ismereteket, azok nélkül nem érthető és nem tanulható. Továbbá sokban támaszkodik, és gyakran hivatkozik az **Operációs rendszerek 1** és az **Adatszerkezetek és algoritmusok** című tárgy témaköreire is.

A jegyzet megírásának időpontjában a tárgy gyakorlatán a Java vagy a C# a kötelező nyelv. A gyakorlaton Gábor András és Fazekas Imre **Java példatár** című elektronikus jegyzete használható.

1. AZ OBJEKTUMORIENTÁLT PARADIGMA

Az *objektumorientált* (OO) paradigma középpontjában a programozási nyelvek absztrakciós szintjének növelése áll. Ezáltal egyszerűbbé, könnyebbé válik a modellezés, a valós világ jobban leírható, a valós problémák hatékonyabban oldhatók meg.

Az OO szemlélet szerint az adatmodell és a funkcionális modell egymástól elválaszthatatlan, külön nem kezelhető. A valós világot egyetlen modellel kell leírni, és ebben kell kezelni a statikus (adat) és a dinamikus (viselkedési) jellemzőket. Ez az *egységbezárás* elve.

Az OO paradigma az absztrakt adattípus fogalmán épül fel. Az OO nyelvek legfontosabb alapeszköze az absztrakt adattípust megvalósító *osztály*. Az osztály maga egy absztrakt nyelvi eszköz, ezen nyelvek implementációi gyakran egy osztályegüttesként jönnek létre.

Az osztály rendelkezik *attribútumokkal* és *metódusokkal*. Az attribútumok tetszőleges bonyolultságú adatstruktúrát írhatnak le. A metódusok szolgálnak a viselkedés megadására. Ezek fogalmilag (és általában ténylegesen is) megfelelnek az eljárásorientált nyelvek alprogramjainak.

A másik alapeszköz az *objektum*, ami konkrét nyelvi eszköz. Egy objektum mindig egy osztály *példányaként* jön létre a *példányosítás* során. Egy adott osztály minden példánya azonos adatstruktúrával és azonos viselkedésmóddal rendelkezik.

Minden objektumnak van *címe*, az a memóriaterület, ahol az adatstruktúra elemei elhelyezkednek. Az adott címen elhelyezkedő értékegyüttest az objektum *állapotának* hívjuk. A példányosítás folyamán az objektum *kezdőállapotba* kerül.

Az OO szemléletben az objektumok egymással párhuzamosan, egymással kölcsönhatásban léteznek. Az objektumok kommunikációja *üzenetküldés* formájában történik. Minden objektum példányosító osztálya meghatározza azt az *interfészt*, amely definiálja, hogy más objektumok számára az ő példányainak mely attribútumai és metódusspecifikációi látszanak (erre szolgál a *bezárási* eszközrendszer – l. később). Tehát egy objektum küld egy üzenetet egy másik objektumnak (ez általában egy számára látható metódus meghívásával és az üzenetet fogadó objektum megnevezésével történik), ez pedig megválaszolja azt (a metódus visszatérési értéke, vagy output paraméter segítségével). Az üzenet hatására lehet, hogy az objektum megváltoztatja az állapotát.

Az objektumnak van *öntudata*, minden objektum csak önmagával azonos és az összes többi objektumtól különbözik. Minden objektum rendelkezik egyedi *objektumazonosítóval* (Object Identifier – OID), amelyet valamilyen nyelvi mechanizmus valósít meg.

Egy osztály attribútumai és metódusai lehetnek *osztály* és *példány szintűek*. A példány szintű attribútumok minden példányosításnál elhelyezésre kerülnek a memóriában, ezek értékei adják meg a példány állapotát. Az osztály szintű attribútumok az osztályhoz kötődnek, nem „többszöröződnek”. Például ilyen attribútum lehet az osztály *kiterjedése*, ami azt adja meg, hogy az adott osztálynak az adott pillanatban hány példánya van.

A példány szintű metódusok a példányok viselkedését határozzák meg. Ezen metódusok meghívásánál mindig meg kell adni egy konkrét objektumot. Azt az objektumot, amelyen egy metódus éppen operál, *aktuális példánynak* hívjuk. A nyelvek az aktuális példányt *this* vagy *self* megnevezéssel hivatkozzák.

Az osztály szintű metódusoknál nincs aktuális példány, az osztály szintű attribútumok manipulálására használjuk őket.

A példány szintű metódusok lehetnek *beállító* és *lekérdező* metódusok. A beállító metódusok hatására az aktuális példány állapotot vált, valamelyik (esetleg mindegyik) attribútumának értéke megváltozik. Ezek eljárás jellegűek, az új attribútum-értékeket paraméterek segítségével határozhatjuk meg. A lekérdező metódusok függvény jellegűek. Az aktuális példány aktuális állapotával térnek vissza.

Példányosításkor lefoglalódik a memóriaterület az objektum számára, ott elhelyezésre kerülnek a példány szintű attribútumok, és beállítódik a kezdőállapot. Az objektum ettől kezdve *él* és tudja, hogy mely osztály példányaként jött létre. A kezdőállapot meghatározására az OO nyelvek általában egy speciális metódust, a *konstruktort* használják.

Az aktuális példány kezelését az OO nyelvek a példány szintű metódusokba implicit paraméterként beépülő speciális hivatkozással oldják meg.

Az OO nyelvek az osztályok között egy aszimmetrikus kapcsolatot értelmeznek, melynek neve *öröklődés*. Az öröklődés az újrafelhasználhatóság eszköze.

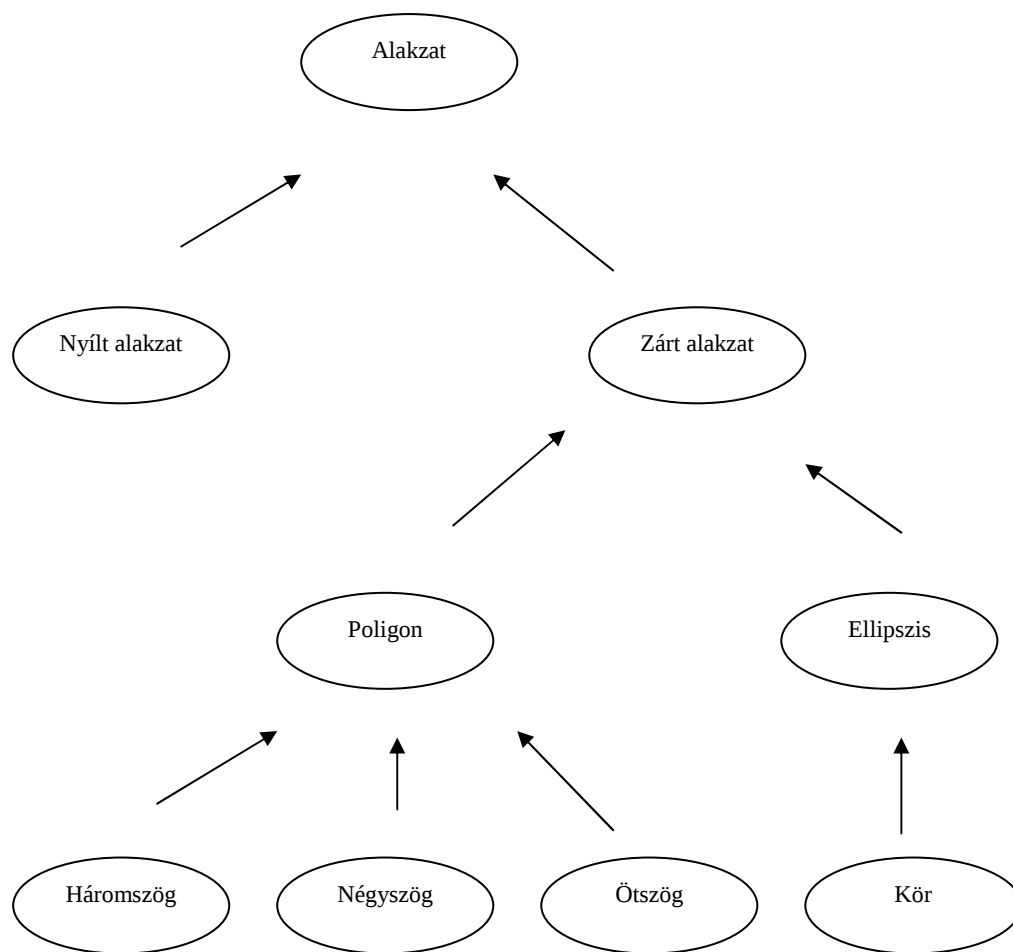
Az öröklődési viszonynál egy már létező osztályhoz kapcsolódóan – melyet *szuperosztálynak* (*szülő osztálynak*, *alaposztálynak*) hívunk – hozunk létre egy új osztályt, melynek elnevezése *alosztály* (*gyermek osztály*, *származtatott osztály*). Az öröklődés lényege, hogy az alosztály átveszi (örökli) szuperosztályának minden (a bezárás által megengedett) attribútumát és metódusát, és ezeket azonnal fel is tudja használni. Ezen túlmenően új attribútumokat és metódusokat definiálhat, az átvett eszközöket átnevezheti, az átvett neveket újradeklarálhatja, megváltoztathatja a láthatósági viszonyokat, a metódusokat újraimplementálhatja.

Az öröklődés lehet *egyszeres* és *többszörös*. Egyszeres öröklődés esetén egy osztálynak pontosan egy, többszörös öröklődés esetén egynél több szuperosztálya lehet. Mindkét esetben igaz, hogy egy osztálynak akárhány alosztálya létrehozható.

Természetesen egy alosztály lehet egy másik osztály szuperosztálya. Így egy *osztályhierarchia* jön létre. Az osztályhierarchia egyszeres öröklődés esetén fa, többszörös öröklődés esetén aciklikus gráf. A többszörös öröklődést valló nyelvek osztályhierarchiájában is van azonban általában egy kitüntetett „gyökér” osztály, amelynek nincs szuperosztálya, és léteznek olyan „levél” osztályok, amelyeknek nincsenek alosztályai.

A többszörös öröklődésnél gondot okozhat a különböző szuperosztályokban használt azonos nevek ütközése.

A következő ábra egy öröklődési fát szemléltet:



Az öröklődési hierarchiában az egy úton elhelyezkedő osztályok közvetlen vagy közvetett öröklődési viszonyban vannak. Az alosztályok irányába haladva az osztályok *leszármazott* osztályai helyezkednek el, a másik irányban viszont az *előd* osztályok találhatók. Példánkban a Zárt alakzat, a Poligon és a Háromszög az Alakzat leszármazottjai; az Ellipszis, a Zárt alakzat és az Alakzat a Kör elődei.

Az egymással előd-leszármazott viszonyban nem levő osztályokat *kliens* osztályoknak hívjuk.

Az osztályok eszközeinek láthatóságát szabályozza a *bezárás*. Az OO nyelvekben általában a következő bezárási szintek léteznek. *Publikus* szint esetén az eszközt látja az összes kliens osztály. *Védett* szintnél az eszközökhöz csak a leszármazott osztályok férhetnek hozzá. A *privát* szintű eszközök viszont csak az adott osztályban használhatók (pontosabban: egy alosztály természetesen öröklí a privát attribútumokat és metódusokat, de ezekre közvetlenül, explicit módon nem hivatkozhat – *láthatatlan öröklés*).

Több OO nyelv értelmez még egy negyedik szintet is, amely általában a nyelv programegység szerkezetén alapul.

Az öröklődésen alapul és az újrafelhasználhatóságnak egy igen jellegzetes megnyilvánulása a *helyettesíthetőség*. Az OO paradigma azt mondja, hogy egy leszármazott osztály példánya a

program szövegében minden olyan helyen megjelenhet, ahol az előd osztály egy példánya (az utóbbi *helyettesíthető* az előbbivel). Egy konkrét *Kör* objektum egyben *Ellipszis* és *Zárt alakzat* is, tehát ezeknek az osztályoknak is objektuma. Viszont csak a *Kör* osztálynak a példánya.

Egy alosztály az örökölt metódusokat újrainplementálhatja. Ez azt jelenti, hogy különböző osztályokban azonos metódusspecifikációkhoz különböző implementáció tartozik. Az ilyen metódusokat *polimorf*nak nevezzük. Ezek után a kérdés az, hogy ha meghívunk egy objektumra egy ilyen metódust, akkor melyik implementáció fog lefutni. A választ egy nyelvi mechanizmus, a *kötés* adja meg.

Az OO nyelvek két fajta kötést ismernek. *Statikus kötés* esetén már fordításkor eldől a kérdés. Ekkor a helyettesíthetőség nem játszik szerepet. A forrásszövegben megadott objektum deklaráló osztályának metódusa fog lefutni minden esetben.

Dinamikus kötés esetén a kérdés csak futási időben dől el, a megoldás a helyettesíthetőségen alapul. Annak az objektumnak a példányosító osztályában megadott (vagy ha nem írta fölül, akkor az öröklött) implementáció fog lefutni, amelyik ténylegesen kezelésre kerül.

Tehát ha a *Poligon* osztályban van egy *Kerület* metódus, amelyet újrainplementálunk a *Négyszög* osztályban, akkor statikus kötés esetén, ha egy poligon objektumra meghívjuk, akkor a *Poligon* implementáció fut le akkor is, ha a futás közben egy négyszög objektum kerületét határozzuk meg. Dinamikus kötés esetén viszont az utóbbi esetben a *Négyszög* implementációja fog futni.

Az OO nyelvek egy része a dinamikus kötést vallja. Másik részükben mindkettő jelen van, az egyik alapértelmezett, a másikat a programozónak explicit módon kell beállítania.

Az OO nyelvek általában megengedik a metódusnevek túlterhelését. Ez annyit jelent, hogy egy osztályon belül azonos nevű és természetesen eltérő implementációjú metódusokat tudunk létrehozni. Ekkor természetesen a hivatkozások feloldásához a specifikációknak különbözniük kell a paraméterek számában, sorrendjében vagy azok típusában (ez nem mindig elég).

Az OO nyelvek általában ismerik az *absztrakt osztály* fogalmát. Az absztrakt osztály egy olyan eszköz, amellyel viselkedésmintákat adhatunk meg, amelyeket aztán valamely leszármazott osztály majd konkretizál.

Egy absztrakt osztályban vannak *absztrakt metódusok*, ezeknek csak a specifikációja létezik, implementációjuk nem. Egy absztrakt osztályból származtatható absztrakt és konkrét osztály. A *konkrét osztály* minden metódusához kötelező az implementáció, egy osztály viszont mindaddig absztrakt marad, amíg legalább egy metódusa absztrakt.

Az absztrakt osztályok nem példányosíthatók, csak örököltethetők.

Egyes OO nyelvekben létrehozhatók olyan osztályok, amelyekből nem lehet alosztályokat származtatni (ezek az öröklődési hierarchia „levelei” lesznek). Ezek természetesen nem lehetnek absztrakt osztályok, hiszen akkor soha nem lehetne őket konkretizálni.

Egyes OO nyelvek ismerik a *paraméterezett osztály (kollekció)* fogalmát. Ezek lényegében az OO világ generikusai.

Az OO nyelvekben az objektumok memóriában kezelt konstrukciók. Egy objektum mindig *transziens*, tehát nem éli túl az őt létrehozó programot. I/O segítségével természetesen bármely objektum *állapota* állományba menthető és azután bármikor létrehozható egy *másik* objektum, amelynek állapota ugyanaz lesz. A nem nyelvi OO rendszerek (pl. adatbázis-kezelők) ismerik a *perzisztens* objektum fogalmát. Ekkor az objektum túléli az őt létrehozó alkalmazást, bármikor újra betölthető a memóriába, és *ugyanaz* az objektum marad.

Természetesen egy program működése közben is fel kell szabadítani a már szükségtelen objektumokhoz rendelt tárterületet. Erre az OO nyelvek kétféle megvalósítást tartalmaznak. Egy részük azt mondja, hogy a programozónak kell explicit módon megszüntetnie az objektumot. Más részük automatikus törlési mechanizmust biztosít (garbage collection). Ezeknél a háttérben, aszinkron módon, automatikusan működik a „szemétgyűjtő” a szokásos algoritmusok valamelyike (pl. hivatkozásfigyelés) alapján.

Az OO nyelveknek két nagy csoportja van. A *tiszta* OO nyelvek teljes mértékben az OO paradigma mentén épülnek fel, ezekben nem lehet más paradigma eszközeinek segítségével programozni. Ezen nyelvekben egyetlen osztályhierarchia létezik. Ez adja a nyelvi rendszert és a fejlesztői környezetet is egyben. Ezen nyelvekben a programozás azt jelenti, hogy definiáljuk a saját osztályainkat, azokat elhelyezzük az osztályhierarchiában, majd példányosítunk.

A *hibrid* OO nyelvek valamilyen más paradigma (eljárásorientált, funkcionális, logikai, stb.) mentén épülnek fel, és az alap eszközrendszerük egészül ki OO eszközökkel. Ezen nyelvekben mindkét paradigma mentén lehet programozni. Általában nincs beépített osztályhierarchia (hanem pl. osztálykönyvtárak vannak), és a programozó saját osztályhierarchiákat hozhat létre.

Egyes *tiszta* OO nyelvek az *egységesség* elvét vallják. Ezen nyelvekben egyetlen programozási eszköz van, az objektum. Ezekben a nyelvekben tehát minden objektum, a metódusok, osztályok is.

Az OO paradigma imperatív paradigmaként jött létre. Tehát ezek a nyelvek algoritmikusak, és így eredendően fordítóprogramosak. A SIMULA 67 az első olyan nyelv, amely tartalmazza az OO eszközrendszert, de a paradigma fogalmait később a Smalltalk fejlesztői csapata tette teljessé. Aztán folyamatosan kialakultak a hibrid OO nyelvek, és megjelent a deklaratív OO paradigma (CLOS, Prolog++) is.

2. JAVA ÉS C#

A Java és a C# „majdnem” tiszta OO nyelvek. A C++ egy továbbfejlesztett változataként tekinthetők. Szintaktikájuk nagyon hasonlít a C (C++) szintaktikájához. a C# közelebb áll a C-hez. Tartalmaznak eljárásorientált elemeket, de programozni bennük csak az OO paradigma mentén lehet.

A Java és C# egyszeres öröklődést, késői kötést, automatikus memóriakezelést valósítanak meg. Vannak bennük primitív típusok, ezek megfelelnek az eljárásorientált egyszerű típusoknak. Van változójuk, ami primitív típusú értéket, vagy objektumreferenciát vehet fel értékül. A referencia típusú változók mindig a hivatkozott objektumot adják. Kifejezésefogalmuk teljesen C-szerű. A metódusokat C értelemben vett függvények segítségével, az attribútumokat változókkal lehet megadni. Bezárási eszközszerük négy szintű. A publikus, privát és védett szint mellett létezik egy fordítási egység szint is. Mindkét nyelvnek hatékony kivételkezelő és párhuzamos programozást lehetővé tevő eszközszerük van.

A Java fordítóprogram egy közbenső formára, az ún. bájt kódra fordít, amelyet aztán a Java Virtual Machine (JVM) interpretál. Ez egyrészt segíti a hordozhatóságot, másrészt viszont lassítja a futást.

A C# aWindows (.NET) környezetben használt CLR (közös nyelvű futási idejű környezet) része. Az ebbe integrált nyelvek közös kódra fordítanak, és használhatják egymás eszközeit.

3. SMALLTALK

3.1. Alapvető jellemzők

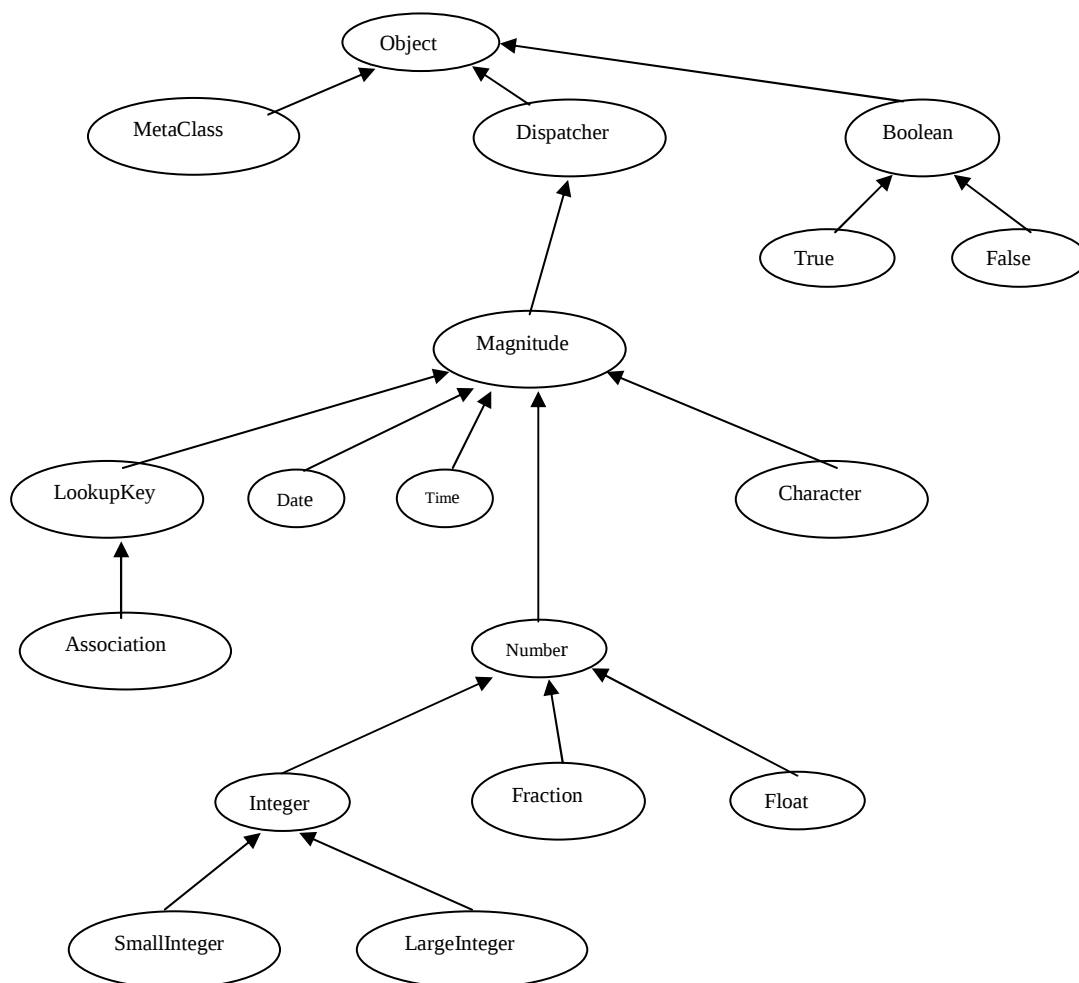
A Smalltalk egyszeres öröklődést, késői kötést megvalósító, az egységesség elvét valló, automatikus objektum megszüntetéssel rendelkező, üzenetalapú OO nyelv. Nem típusos, mivel benne minden objektum.

A Smalltalknál teljesen integrált a nyelv és a grafikus fejlesztői környezet. Az osztályhierarchia gyökerét az **Object** osztály adja.

A Smalltalk korai verzióiba nem építették be az absztrakt osztály és absztrakt metódus fogalmát, a maiakban azonban ez már megjelenik.

A továbbiakban az IBM Smalltalk verzió eszközeinek segítségével illusztráljuk a nyelv legalapvetőbb lehetőségeit.

A következő ábra az osztályhierarchia egy részletét mutatja:



Itt a `Magnitude` és `Number` osztályok például absztrakt osztályok. A `Magnitude` objektumok összehasonlíthatóak.

3.2. Literálok

A `Smalltalk` literáljai formálisan megfelelnek az eljárásorientált nyelvek literáljainak, programszövegbeli megjelenéseik általában a szokásosak, de maguk a megfelelő osztály példányai.

Például a `36` a `SmallInteger`, a `3.28` a `Float`, a `3/4` a `Fraction` példánya.

A karakter literál alakja `$karakter`, például `$a`, `S`;

A sztring literál `'[karakter]...'` alakú.

3.3. Változók

A `Smalltalk`-ban van változó. Minden objektumot tud változó segítségével kezelni úgy, hogy a változó értékül az objektum OID-jét veszi fel (tehát referencia értékű). És minden változóhivatkozás az objektumot jelenti. Egy változó alapértelmezett kezdőértéke `nil`, ami az `UndefinedObject` osztály egyetlen példánya.

Az osztályok attribútumai a `Smalltalk`-ban változók, a metódusok pedig az üzenetek. A bezárás több szintű. A metódusok mindig publikusak, tehát mindenhol láthatók.

Az objektumok állapotát meghatározó *példány változók* bezárási szintje *objektumszintű*. Tehát csak az adott objektum példány metódusai hivatkozhatják őket közvetlenül, még ugyanazon osztály más példányainak metódusai sem. Egy ilyen változót más objektum csak úgy tud elérni, hogy az *objektumnak* küld üzenetet, tehát lennie kell a példány változót író (beállító) és olvasó (lekérdező) metódusnak (üzenetnek).

Az *osztály változókat* látják az adott osztály, annak példányai, a leszármazott osztályok és ezek minden példánya.

Az *osztály példány változót* közvetlenül elérheti az osztály (pontosabban az osztály szintű metódusok), de példányai és leszármazott osztályai nem (ennek azért van értelme, mert az osztály maga is objektum).

A *globális változók* minden osztály minden metódusa számára láthatók.

Az *ideiglenes változók* a metódusok lokális változói, hatáskörük csak az adott metódus, élettartamuk dinamikus.

A *megosztott változók* olyan változók, amelyeket minden egyes osztály explicit módon ilyenek definiál, és ekkor ezen osztályok közösen tudják használni őket.

3.4. Kifejezések

Egy Smalltalk program egymástól *ponttal* elválasztott *utasítások* sorozatából áll. Egy *utasítás kifejezésekből* épül fel. Egy kifejezés lehet *elsődleges kifejezés*, *üzenet kifejezés* vagy *kaszkád kifejezés*. Az elsődleges kifejezés lehet *változónév*, *literál* vagy kerek zárójelek közé zárt kifejezés.

Az *üzenet kifejezés* lehet *unáris*, *bináris* vagy *kulcsszó kifejezés*. Az *üzenet kifejezés* általános alakja:

fogadó_objektum *üzenetnév* [*argumentum*]

Az *unáris üzenetnek* nincsenek *argumentumai*, a *binárisnak* egy *argumentuma* van, a *kulcsszó üzenet* egy vagy több *kulcsszóból* és a *kulcsszavakat* követő egy-egy *argumentumból* áll.

A *fogadó_objektum* bármely olyan objektum, amely megérti az üzenetet.

Minden *üzenet* megfelel egy, a *fogadó objektum példányosító osztályában* definiált (vagy ott ismert) *metódus meghívásának*. Az *üzenetnév* a *metódus neve*, az *argumentumok* az *aktuális paraméterek*.

A *fogadó objektum* mindig *válaszol* az *üzenetre*, tehát az *üzenetnek* van *visszatérési értéke*, ami *alapértelmezés szerint* a *fogadó objektum* (pontosabban annak *OID-je*).

A *kaszkád üzenet* nem más, mint egy *közös fogadó objektumnak* küldött, egymástól *pontosvesszővel* elválasztott *üzenetsorozat*.

A Smalltalk a következő módon értékeli ki a kifejezést:

1. A kiértékelés balról jobbra történik.
2. A bezárójelezett részkifejezések értékelődnek ki először, a legbaloldalibb legbelső ilyennel kezdve.
3. Egy *üzenet* a *kifejezésben* helyettesítődik a *visszatérési értékkel*.
4. Egy *literál értéke* önmaga; *változó értéke* a *hivatkozott objektum*; az *üzenet kifejezések végrehajtási sorrendje*: minden *unáris*, majd minden *bináris*, végül minden *kulcsszó üzenet*.

Egy *változónak* értéket adhatunk az alábbi módon:

változó := *utasítás*

A := egy *kulcsszó üzenet* neve, hatására a *változó* az *utasítás* által meghatározott objektumot fogja címezni.

Példa értékadásokra:

```
x := 5.  
y := 3.  
a := x + y.  
z := 'ez egy sztring'.
```

A Smalltalk bináris üzenetei az eljárásorientált nyelvek bináris operátorainak nyelvi megfelelői. Nézzük ezeket:

Aritmetikai üzenetek

A `Number` osztály metódusai.

<code>+</code>	: összeadás
<code>-</code>	: kivonás
<code>*</code>	: szorzás
<code>/</code>	: osztás
<code>//</code>	: egészszosztás
<code>\\</code>	: maradékképzés

Hasonlító üzenetek

A `Magnitude` osztály metódusai:

<code>></code>	: nagyobb
<code><</code>	: kisebb
<code>=</code>	: értékben egyenlő
<code>~=</code>	: értékben nem egyenlő
<code><=</code>	: kisebb vagy egyenlő
<code>>=</code>	: nagyobb vagy egyenlő
<code>==</code>	: azonosságban egyenlő

Logikai üzenetek

A `Boolean` osztály metódusai:

<code>&</code>	: és
<code> </code>	: vagy

Teljes kiértékeléssel rendelkeznek.

Logikai üzenet a `not` is, de az unáris.

Itt jegyezzük meg, hogy a `True` és a `False` osztálynak nincsenek példány változói és egyetlen példányuk a `true`, illetve a `false`. Tehát minden változó, amely a `true` objektumot hivatkozva, ugyanazt az objektumot hivatkozta. Tehát az

```
a := true.
```

```
b := true.
```

értékkadások után az

```
a == b
```

kifejezés értéke `true`.

3.5. Blokk

A Smalltalk ismeri a blokk fogalmát. Ez egy olyan objektumnak tekinthető (példányosító osztálya a **Block**), amely végrehajtható kódot tartalmaz. Egy blokk egy metódusba ágyazottan jelenhet meg, ezért értelmezhetjük egy metódusba ágyazott metódusként is. A blokk lokális változóit (melyeknek hatásköre a blokk) a Smalltalk argumentumoknak hívja. Ezek neve kötelezően kettősponttal kezdődik. A blokk maga utasításokat és megjegyzéseket tartalmazhat. A megjegyzés a Smalltalkban idézőjelek közé zárt karaktersorozat. A blokkban használhatók azok a változók, amelyeket a tartalmazó metódus lát.

A blokknak is van visszatérési értéke.

A blokk formálisan a következőképpen néz ki:

`[ [változó ... ] kód ]`

3.6. Vezérlési szerkezet

Feltételes végrehajtás

A **Boolean** kulcsszó üzenetei az `ifTrue:` és az `ifFalse:`, amelyek az alábbi módokon használhatók:

Boolean_objektum ifTrue: blokk ifFalse: blokk

Boolean_objektum ifTrue: blokk

Boolean_objektum ifFalse: blokk

A *blokk* argumentum nélküli. A *Boolean_objektum* egy kifejezés, melynek értéke `true` vagy `false`.

Az eljárásorientált nyelvek kétirányú elágaztató utasításának megfelelő konstrukció.

A következő utasítás kiválasztja a és b közül a nagyobbikat

`(a > b) ifTrue: [y := a] ifFalse [y := b].`

Ciklusok

A `timesRepeat:` az **Integer** osztály kulcsszó üzenete, formája:

Integer_objektum timesRepeat blokk

A blokk itt is argumentum nélküli. Hatása az, hogy az *Integer_objektum* értékének megfelelő számszor végrehajtja a blokkot.

Példa: Egy sztringben változtassuk az összes mássalhangzót kisbetűre, a magánhangzókat pedig nagybetűre.

```

string := 'Ez a sztring.'.
index := 1.
string size timesRepeat:
    [c := string at: index.
     string at: index put:
        (c isVowel ifTrue: [c asUpperCase]
         ifFalse: [c asLowerCase]).
     index := index + 1]

```

A Smalltalk a sztringet karakterek egydimenziós tömbjének tekinti, ez index 1-ről indul. Az `at:` a megadott indexű elemet adja, a `put:` felülírja azt.

Az `asUpperCase` és `asLowerCase` az átalakító üzenetek. Az `isVowel` egy olyan üzenet, amely visszatérési értéke `true`, ha a karakter magánhangzó, `false`, ha mássalhangzó.

Az eljárásorientált nyelvek kezdőfeltételes ciklusainak felelnek meg a `whileFalse:` és `whileTrue:` Boolean metódusok, melyek kulcsszó üzenetek. Argumentumuk egy argumentum nélküli blokk, fogadó objektumuk egy `true` vagy `false` értéket adó blokk.

Példa:

```

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

```

Az eljárásorientált előírt lépésszámú ciklusnak felel meg a következő konstrukció:

```

k to: v [ by: l ] do: [cv | kód ]

```

Itt `k`, `v` és `l` `Number` (szokásos módon `Integer`) objektumok (rendre megadják a kezdőértéket, végértéket, lépésközt), a `cv` a ciklusváltozó. Az irányt a lépésköz előjele határozza meg. Ha `by:` hiányzik, a lépésköz 1.

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]]

```

3.7. Osztályok

A Smalltalk legkisebb önálló egysége és fordítási egysége az osztály. Az osztályok mint objektumok a `MetaClass` osztály példányaként jönnek létre. Nevük globális változó.

Egy saját osztály létrehozásánál mindig meg kell adnunk a szuperosztály nevét és az öröklött eszközökhöz új példány, osztály, megosztott változókat adhatunk, az öröklött metódusokat újraimplementálhatjuk és saját példány és osztály metódusokat definiálhatunk.

Az új osztály létrehozásánál használhatjuk az integrált fejlesztői környezetet, vagy írhatunk forrásszöveget.

Az IBM Smalltalk névkonvenciója a következő. Egy programozói név felépítése az eljárásorientált azonosítónak felel meg. A példány, osztály példány és ideiglenes változók és a példány metódusok neve kisbetűvel, a globális, osztály és megosztott változók, valamint az osztály metódusok neve nagybetűvel kezdődik. Ha egy név több szóból áll a második, harmadik, stb. szó kezdőbetűje nagybetű, a többi kicsi. A blokk lokális változói nevének első és beállító metódusok nevének utolsó karaktere kettőspont.

A példány változók objektumszintű bezárása miatt ezekben célszerű egy-egy beállító és lekérdező metódust definiálni, ezek neve gyakran azonos a változó nevével.

Saját osztály létrehozásának alakja:

```
szuperosztály_név subclass: #név  
[classInstanceVariableNames: sztring]  
instanceVariableNames: sztring  
classVariableNames: sztring  
poolDictionaries: sztring  
metódusdefiníciók
```

A *sztring* rendre az osztály példány (ez hiányozhat a definícióból), példány, osztály és megosztott változók nevét tartalmazza szóközzel elválasztva. Az üres *sztring* azt jelenti, hogy olyan kategóriájú attribútuma nincs az osztálynak.

A *név* lesz az új osztály neve.

A *subclassInstanceVariableNames:instanceVariableNames:*

classVariableNames:poolDictionaries: kulcsszó üzenet hatására létrejön egy metaosztály, amely az új osztály jellemzőit leírja, majd ennek példányaként létrejön maga az új osztály.

Egy osztályt példányosítani az osztály nevének elküldött unáris *new* üzenettel lehet. A Smalltalk nem ismeri a konstruktor fogalmát, a példányosításkor létrejövő objektumnak viszont bármilyen üzenet küldhető, amely beállítja az állapotát.

A *new* egy osztályszintű metódus, az állapot beállító metódusoknak viszont példányszintűeknek kell lenniük.

3.8. Metódusok

Egy metódus formája a következő:

```
interfész [lokális_változók] kód
```

Az interfész alakja:

```
név [argumentum] [név argumentum] ...
```

Az argumentumok a metódus formális paraméterei. A paraméterek száma mindig fix.

A lokális változókat `|` jelek közé írva, egymástól szóközzel elválasztva kell felsorolni, ha vannak.

A kód egy tetszőleges utasítás és megjegyzés sorozat.

A kódban az aktuális példányt a `self` hivatkozza. Ezt használhatjuk akkor is, ha az objektum saját magának akar üzenetet küldeni.

A Smalltalk szemlélete szerint minden üzenetnek van visszatérési értéke, ez alapértelmezett módon a fogadó objektum (tehát a `self`).

Minden utasításnak van visszatérési értéke, ez az utolsónak kiértékelt kifejezés értéke.

Minden blokknak van visszatérési értéke, ez az utoljára végrehajtott utasítás értéke.

Végül a metódus visszatérési értéke az utoljára végrehajtott utasítás értéke.

A metódusokban a visszatérési értéket meghatározhatjuk explicit módon is

`^utasítás`

formában. Ez megfelel az eljárásorientált nyelvek RETURN-utasításának. Tehát befejezteti a metódust és annak visszatérési értéke az utasítás értéke lesz.

Ha az interfész egyetlen nevet tartalmaz, akkor egy lekérdező metódust adunk meg, ha több név van, akkor kötelezőek az argumentumok, ekkor ez egy beállító kulcsszó üzenet lesz, ahol a nevek a kulcsszavak és az argumentumok a formális paraméterek.

Az `Integer` osztálynak van egy `factorial` unáris üzenete, melynek mint metódusnak a kódja az alábbi:

`factorial`

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

Az `Object` osztály egyik unáris üzenete pedig a következő:

`yourself`

```
"Answer the receiver."
^self
```

Példa: Hozzuk létre a `Személy` osztályt, melynek példány változói a név, cím, telefonszám, és adjuk meg a lekérdező és beállító példány metódusokat.

```
Object subclass: #Szemely
  instanceVariables: 'nev cim telefonszam'
  classVariableNames: ''
  poolDictionaries: ''
```

```

name
    "Lekérdezi a nevet. "
    ^nev

name: egyNev
    "Beállítja a nevet. "
    nev:= egyNev

cim
    "Lekérdezi a címet. "
    ^cim

cim: egyCim
    "Beállítja a címet. "
    cim:=egyCim

telefonszam
    "Lekérdezi a telefonszámot. "
    ^telefonszam

telefonszam: egyTelefonSzam
    "Beállítja a telefonszámot. "
    telefonszam:= egyTelefonSzam

nev: egyNev cim: egyCim
    "Beállítja a nevet és a címet. "
    self nev: egyNev.
    self cim: egyCim

nev: egyNev cim: egyCim telefonSzam: egyTelefonSzam
    "Beállítja a nevet, címet, telefonszámot. "
    self nev: egyNev.
    self cim: egyCim.
    self telefonSzam: egyTelefonSzam

```

Az utolsó metódus kódját egyszerűsíthetjük, ha kaszkád üzenetet használunk:

```

nev: egyNev cim: egyCim telefonSzam: egyTelefonSzam
    "Beállítja a nevet, címet, telefonszámot. "
    self
        nev: egyNev;
        cim: egyCim;
        telefonSzam: egyTelefonSzam

```

3.9. Szálak

Az IBM Smalltalk a párhuzamos programozás megvalósítására a Dijkstra-féle szemafor technikát alkalmazza (1. **Operációs rendszerek 1**). A szálak kezeléséhez négy osztályt definiál, ezek: `Process`, `ProcessorScheduler`, `Delay` és `Semaphore`.

Egy szál létrehozni egy blokkból lehet. Tehát a szál példányosító üzenetei a `Block` osztály üzenetei. Hatásukra egy `Process` objektum keletkezik.

Egy blokknak küldött `fork` üzenet létrehoz egy szálát és az azonnal ütemezésre is kerül az aktív szállal azonos prioritással. A `forkAt:` üzenettel prioritást rendelhetünk a szálhoz, ezzel kerül ütemezésre.

A `newProcess` üzenet létrehoz egy szálát az aktív szállal azonos prioritással, a `newProcessWith:` pedig a megadott prioritással, de az csak akkor kerül ütemezésre, ha megkapja a `resume` üzenetet.

A szál befejezi a működését, ha a blokknak elfogynak az utasításai, vagy megkapja a `terminate` üzenetet.

Egy szál explicit módon felfüggesztett állapotba kerül, ha elküldjük neki a `suspend` üzenetet. Újraindítani a `resume` üzenettel lehet.

Egy szál működése közben szinkron módon lefuttathatunk egy `queueInterrupt:` üzenet argumentumaként megadott blokkot.

A `ProcessorScheduler` osztály egyetlen példánya a `Processor` globális változó, amely az ütemezést végzi. Ez az osztály egy abszolút prioritásos round-robin ütemezést implementál. Ez alapján egy szál mindaddig fut, azaz birtokolja a CPU-t (tehát ő az *aktív szál*), amíg a következők valamelyike be nem következik:

- olyan műveletet hajt végre, amely hatására a futása felfüggesztődik (pl. `suspend`, `wait`),
- egy nála magasabb prioritású szál ütemezésre, vagy továbbindításra kerül,
- véget ér.

Tehát a CPU-t mindig a legrégebben várakozó, legnagyobb prioritású futásra kész szál kapja meg.

Hét prioritási szint van, a `ProcessorScheduler` osztály metódusai ezeket kezelik.

A `Delay` példányai egy adott szál rendszerórával való szinkronizációját teszik lehetővé. Egy szálát várakoztatni lehet egy megadott időintervallum leteléséig, vagy egy megadott abszolút időpontig. Például a `wait` felfüggeszti a szál futását egy osztály változóban meghatározott, ezredmásodpercben megadott időintervallum leteltéig.

A `Semaphore` többértékű szemaforokat kezel. Egy szemaforhoz egy `signal` üzenet hozzáad egy jelet, a `wait` elvesz egyet.

Ha egy szemafornak nincs egyetlen jele sem és kap egy `wait` üzenetet, akkor a futó szál felfüggesztődik a következő `signal` üzenetig. Ha több szál vár ugyanarra a szemaforra és érkezik egy `signal`, akkor a legrégebben várakozó indul tovább.

3.10. Kivételkezelés

Az IBM Smalltalkban a kivételek kezelésére két osztály szolgál. Az `ExceptionalEvent` példányai a *kivételek*, ezek *kiváltása* esetén hívódik meg egy kivételkezelő. A `Signal` példányai a *kiváltott kivételek*, ezek paraméterként átadódnak a kivételkezelőnek.

Egy kivétel úgy jön létre, hogy a `newChild` üzenetet elküldjük egy már létező („szülő”) kivételnek. A legáltalánosabb kivétel az `ExceptionalEvent` egy példánya, az `ExAll`. Ő minden kivétel szülője, neki viszont nincs szülője. Ezáltal egy kivételhierarchia hozható létre, melynek gyökere az `ExAll`.

Az `ExceptionalEvent` példány változói a következők:

<code>description</code>	A kivételhez rendelt sztring (a „hibaüzenet”).
<code>parent</code>	A szülő kivétel, <code>ExAll</code> esetén <code>nil</code> .
<code>resumable</code>	Boolean objektumot kell tartalmaznia. Ha <code>true</code> , akkor a program futása folytatható azon a helyen, ahol a kivétel kiváltódott. <code>ExAll</code> esetén <code>false</code> .
<code>defaultHandler</code>	Egy argumentumú blokk, amelyik működésbe lép, ha explicit módon nem kezeljük a kivételt.

Példa:

```
| endOfFileException |
(endOfFileException := ExAll newChild)
  description 'állomány vége'
```

Egy kivételt kiváltani a kivételnek küldött következő üzenetekkel lehet:

`signal`, `signalWith:`, `signalWith:with`, `signalWithArguments:`.

Ekkor rendre 0, 1, 2, akárhány argumentum adható át, az utolsó esetben az argumentumok egydimenziós tömböt alkotnak. Ezek hatására egy `Signal` példány keletkezik, amelyik információkat tartalmaz a kiváltódás körülményeiről.

Példa:

```
endOfFileException signal
```

Saját kivételkezelőt úgy adhatunk meg, hogy egy blokknak elküldjük a `when:do:` üzenetet. A kivételkezelő ekkor a blokkban kiváltott kivételekre hatásos. A `when:` argumentuma egy kivételpéldány, a `do:` argumentuma egyargumentumú blokk, amely átadódik a kiváltott

kivételpéldánynak. A `when:do:` kivételkezelő ötszörös maximális értékig ismételhető egymás után.

Ha a fogadó blokkban kiváltódik egy kivétel, akkor a futtató rendszer a felírás sorrendjében megy végig a kivételkezelőkön és keres megfelelőt (olyat, amelyik `when:` utáni argumentuma megegyezik a kivétellel, vagy elődje annak). Az `EXAll` minden kivételnek elődje, tehát „elfog” bármilyen kivételt.

Ha van megfelelő kivételkezelő, akkor lefut a `do:` utáni blokk. Ebben többek között a következő üzenetek küldhetők a `Signal` példánynak:

<code>argument</code>	A kiváltáskor átadott első argumentum, vagy <code>nil</code> ha nincs ilyen.
<code>arguments</code>	A kiváltáskor átadott argumentumok.
<code>exitWith:</code>	A kivételkezelő blokk visszatérési értékét adja meg. Itt nem használható a <code>^</code> .
<code>signal</code>	Továbbadja a kivételt.
<code>signalWith:</code>	Egy paraméterrel továbbadja a kivételt.
<code>retry</code>	Újrafuttatja a kivételkezelő fogadó blokkját.
<code>handlesByDefault</code>	Lefuttatja az alapértelmezett kivételkezelőt.
<code>description</code>	Lekérdezi az értelmező sztringet. Értéke <code>'an exception has occurred'</code> , ha nem volt beállítva.

A `Smalltalk` a kivételkezelőben bekövetkező kivételt ugyanúgy kezeli, mint ami bárhol máshol következett be. Viszont egy kivétel mindig kezelésre kerül explicit vagy implicit módon (az alapértelmezett kivételkezelő miatt).

Az `ExceptionalEvent` alapértelmezett kivételkezelője a következő:

```
|anException|
(anException:=EXAll newChild)
    defaultHandler: [:signal |
        self error:
            'The exception was not expected at this time.'].
anException signal
```

Az `EXAll` kivétel a rendszerkivételek közé tartozik. Ilyen összesen négy van. Itt most még egyet említünk. Az `Object` üzenete az `error:`, melynek argumentuma egy sztring. Ez az üzenet az `EXError` rendszerkivételt váltja ki. Hatására elindul a `Smalltalk` belövője.

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. később) 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 három csoportját különbözteti meg, ezek a *kulcsszavak*, egyes *típusnevek* és az *előredefiniált nevek*. Az Eiffel foglalt szavai ábécé sorrendben a következők:

alias	all	and	as	<i>BIT</i>	<i>BOOLEAN</i>
<i>CHARACTER</i>	check	class	creation	<i>Current</i>	debug
deferred	do	<i>DOUBLE</i>	else	elsif	end
ensure	expanded	export	external	false	feature
from	frozen	if	implies	indexing	infix
inherit	inspect	<i>INTEGER</i>	invariant	is	like
local	loop	<i>NONE</i>	not	obsolete	old
once	or	<i>POINTER</i>	prefix	<i>REAL</i>	redefine
rename	require	rescue	<i>Result</i>	retry	select
separate	<i>STRING</i>	strip	then	true	undefine
unique	until	variant	when	xor	

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

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 metódusai meghatározzák a típus példányain 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 egy típus példányaként 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* típussal 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. Metódusai a szokásos hasonlítási műveletek.

A *NUMERIC* műveletei az összeadás, kivonás, szorzás, osztás infix, és a pozitív és negatív előjel, mint prefix műveletek. A metódusok formális paraméterei és visszatérési értékük *NUMERIC* típusú. Az öröklődés során a metódusokat az alaposztályok implementálják, a formális paramétereket és a visszatérési típust újradeklarálják, a szükséges specifikus metódusokat 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 belső kód (amely egy pozitív egész értékű) és a reprezentáló bitsorozat.

A *BOOLEAN* osztály metódusai 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.

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 unáris operátorai prefixek, bináris operátorai infixek. A precedencia táblázat a következőképpen néz ki:

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

Az egyes operátorok jelentése:

.	a minősítés operátora
old, strip	(l. 4.13.)
not	logikai tagadás
+, -	előjelek
^	hatványozás
*	szorzás
/	osztás
//	egészszorzás
\	maradékképzés
+	összeadás
-	kivonás
=, \=	az <i>ANY</i> osztály egyenlőségvizsgáló műveletei. $e=f$ azonos referencia típusok 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.
<, >, <=, >=	a <i>COMPARABLE</i> 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
;	elhatároló

A kifejezések kiértékelése balról jobbra a precedencia táblázat figyelembevételével történik.

4.5. Végrehajtható utasítások

Az Eiffelben az algoritmusok kódolására végrehajtható utasításokat használunk. Az értékadó utasítás általános alakját l. a 4.8. alfejezetben. A következőkben a vezérlési szerkezetet realizáló utasításokat tárgyaljuk.

4.5.1. Összetett utasítás

Alakja:

```
[utasítás] [[ ; ] utasítás]...
```

Az összetett utasításnál a pontosvessző opcionális elhatároló jelként szerepel, kiírni csak akkor kell, ha a követő utasítás zárójellel kezdődik. Az Eiffel konvenció 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
```

```

[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. 4.13. alfejezet.

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 egyben 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: Algoritmuskok"  
          "Meyer: Eiffel, The Language";  
keszult: 2004, augusztus, 31;  
utolso_modositas: 2004, 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* (parametrizált) 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, amelyek 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 *sstring*-et.

Egy osztály *eszközei attribútumok* és *rutinok* lehetnek. Ez utóbbiak a metódusok. Az attribútum vagy *változó*, vagy *nevesített konstans*, a rutin pedig *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. 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 explicit kezdőérték.

Egy rutin deklarációja a következőképpen néz ki:

[FROZEN] név [, [FROZEN] név]...
[(formális_paraméter_lista)] : típus
IS rutin_leírás

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 *sztring*-et 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 hatáskörkezelése 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 **DO** kulcsszóval indul, akkor az összetett utasítás minden hívásnál lefut, **ONCE** esetén viszont egy adott objektumra csak a munkamenet első hívásakor fut le. A további hívások hatástalanok. Ha függvényről van szó, a visszatérési érték mindig az első hívás visszatérési értéke lesz.

Függvény esetén létezik egy speciális, előredefiniált egyed (l. 4.8.), 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. 4.12.), típusegyenértékűséget (l. 4.11.) alkalmaz. A paraméterátadás érték szerinti. Ez alól kivétel a **POINTER** típus, amivel egy eszköz címét tudjuk átadni egy külső rutinnak.

Az *előfeltétel* és *utófeltétel* szerepét l. 4.13. alfejezet, a *kivételkezelő* pedig 4.14.-ben szerepel.

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

Az *öröklődés* formája:

INHERIT *szuperosztály_név* [*eszközüadaptáció*]
[[;] *szuperosztály_név* [*eszközüadaptáció*]]...

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 eszközüadaptáció alakja:

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

A sorrend kötött.

Az átnevezés formája:

```
RENAME örökölt_eszköznév AS új_eszköznév
[, 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 örököseinek.

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

```
EXPORT {kliens [, kliens]...} {ALL | eszközlista}
[, {kliens [, kliens]...}{ALL | eszközlista}]...
```

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 metódus implementálásánál nem kell a nevét újradefiniálni, hiszen ilyenkor a „definíálás” ebben az osztályban történik meg.

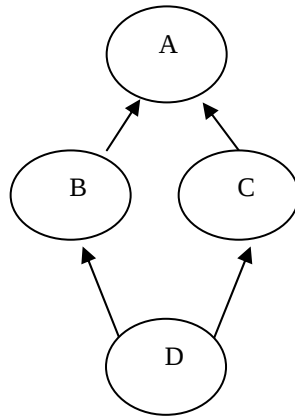
Az újradefiniálás alakja:

```
REDEFINE eszköznév_lista
```

A *szelekció* alakja:

```
SELECT rutinnév_lista
```

Az Eiffel a polimorf rutinok kezelésénél a dinamikus kötést alkalmazza. Ez általában elegendő is, ha megfelelően minősítünk, de problémát jelenthet az alábbi (*rombusz*) öröklődésnél:



Tételezzük fel, hogy *A*-ban van egy *f* rutin, amelyet *B* és *C* újradefiniál, ezeket öröklí *D* és átnevezi a névütközés elkerülése miatt *bf*-re és *cf*-re. Kérdés viszont ekkor, hogy melyik implementáció fut le, ha egy *A* objektumot helyettesítve egy *D* példánnyal, arra meghívjuk *f*-et? Ezt a problémát lehet úgy feloldani, hogy *D*-ben egy *select bf* „kitüntetett” implementációvá teszi a *bf*-et, és ekkor ez fog futni.

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 exportálódhatnak.

A *megjegyzés* egy elvárt megjegyzés.

A konstruktorok eljárások, ezeket az eszközkdeklarációban kell megadni.

Az INVARIANT-rész magyarázatát l. a 4.13. alfejezetben.

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. 4.12.). 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

alaptípus osztálynak a példánya, és 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 példányosításnál létrejön egy új objektum, kezdőállapotba 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 szemétgyűjtőgetőt alkalmaz. Ennek megvalósítása implementációfüggő.

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 „kezdet” és „vége”. Azt már láttuk, hogy az Eiffelben az ősoosztály 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 *NONE* 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 nem generikus osztály neve, vagy egy generikus osztálynév, aktuális generikus paraméterekkel. Tehát ekkor 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 Eiffelben 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*-ével.
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, a paraméterkiértékelésnél és az értékadásnál játszik alapvető szerepet az Eiffelben.

4.12. Tömbök és sztringek

A tömb értékek egy homogén sorozata, amely elemeit egész indexértékeken keresztül érhetjük el. A sztring egy speciális 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.

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

$$<< k_1, \dots, k_n >>$$

ahol k_i -k kifejezések, amelyek típusa egyenértékű.

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

Egy sztring literál 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.

Egy rutin elő- illetve utófeltételének 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, az utófeltételnek a működés befejeződésekor teljesülnie kell. Az utófeltételben szerepelhet az **old** unáris operátor, amely operandusának a rutinba való belépésénél meglevő értékét adja meg.

Az utófeltételben általában azt határozzuk meg formálisan, hogy milyen változásokat okoz a rutin lefutása. Hasznos lehet viszont az is, hogy leírjuk, 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**(*x*, *y*) kifejezést, akkor tulajdonképpen egy olyan tömböt kapunk, amely a **strip** után felsorolt mezőkből áll. **strip**() az összes mezőt tartalmazó tömböt eredményezi. Ekkor az

equal(**strip**(*x*, *y*), **old strip**(*x*, *y*))

igaz értéke azt jelenti, hogy a rutin nem változtathatja meg az *x* és *y* attribútum értékét. Az *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 **deferred class** *LISTA* osztály invariánsa lehet az alábbi:

```
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
```

Egy ciklus invariánsának 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 nem negatívra 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

4.14. Kivételkezelés

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 van egy

```
    default_rescue is  
        do  
        end
```

alapértelmezett kivételkezelő metódus, 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 újrainplementálható) 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 *Caught*, *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 az *EXCEPTIONS* különböző, egész típusú, a bekövetkezett kivétel kódját tartalmazó attribútumai segítségével dönthetjük el. Például ilyenek a *Precondition* (előfeltétel megsértése), *No_more_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 hajódnak végre,
- elindul a kivételkezelő,
- ha van benne *RETRY*-utasítás, akkor a rutin újra lefut (természetesen újra bekövetkezhets valamilyen kivétel és akkor ez ismétlődik rekurzívan),
- ha nincs *RETRY*-utasítás vagy kivétel következik be, 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.

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
    elsif 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 *lehetetlen*-t igazra állítja és sikeresen visszatér.

4.15. I/O

Az input-outputot 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 eszközköze közepesnek mondható.

5. A LOGIKAI PARADIGMA ÉS A PROLOG

A paradigma az 1970-es évek elején születik meg az első logikai programozási nyelv, a Prolog megkonstruálásával. A logikai paradigma a matematikai logika fogalom- és eszközrendszerén épül fel. A Prolog alapjait az elsőrendű predikátumkalkulus és a rezolúciós algoritmus képezi (l. **Az informatika logikai alapjai** című tárgy).

Egy logikai program nem más, mint egy absztrakt modellre vonatkozó *állítások* egy halmaza. Az állítások a modell elemeinek tulajdonságait és a közöttük levő kapcsolatokat formalizálják.

Az állítások egy konkrét kapcsolatot leíró részhalmazát *predikátumnak* nevezzük. Általánosságban egy logikai program lefuttatása egy, az állításokból következő tétel konstruktív bizonyítását jelenti. Ekkor a program állításai egy megoldási környezetet definiálnak, és ebben a környezetben tesszük fel a programnak a *kérdést* (vagy fogalmazzuk meg a *feladatot*), amire a választ egy *következtető gép* keresi meg.

A logikai programozási nyelvekben az állítás *tény* vagy *szabály* lehet. Az állításokat és a kérdéseket közös néven *mondatoknak* nevezzük. Egyes logikai nyelvekben a szigorúan vett logikai eszközökön túlmutató mondatok is lehetnek. A *deklarációk* a predikátumok alkalmazását pontosítják, a *direktívák* a program futtatási környezetét határozzák meg.

A Prolog egy általános célú magas szintű programozási nyelv. A teljes Prolog a logikai eszközrendszeren kívül tartalmaz *metalogikai* és *logikán kívüli* nyelvi elemeket is, továbbá be van ágyazva egy interaktív fejlesztői környezetbe. Mi most először a *tiszta* (absztrakt) Prolog általános működési mechanizmusát tárgyaljuk, és a fejezet végén teszünk egy kitekintést az egyéb elemekre.

Egy tiszta Prolog program felhasználói predikátumok együttese, amelyekben sehol sincs hivatkozás beépített predikátumra.

A Prolog egy nem típusos, interpreteres nyelv. Karakterkészlete a szokásos, a betűk közé az egyes változatok beleértik a nemzeti nyelvi betűket is. A kis- és nagybetűk különböznek.

Megjegyzést a % jel után helyezhetünk el, a sor végéig.

A Prologban a mondatokat . zárja.

A Prolog nyelv alapeleme a *term*, amely lehet *egyszerű* és *összetett*. Egy egyszerű term az vagy *állandó* vagy *változó*. Az állandó az *név* vagy *szám*.

A név egy kisbetűvel kezdődő azonosító, vagy a +, -, *, /, \, ^, <, >, =, ~, :, ., ?, @, #, &, \$ karakterekből álló karaktersorozat. Van négy foglalt név: ;, !, [], {}.

A szám egy olyan karaktersorozat, amely formálisan megfelel az eljárásorientált nyelvek *egész* és *valós* numerikus literáljának.

A változó speciális változó. Típusa nincs, címe nem hozzáférhető. Neve aláhúzásjellel vagy nagybetűvel kezdődő azonosító. Értékkomponensének kezelése speciális. A változó a matematikai egyenletek *ismeretlenjének* felel meg. Rá az *egyszeres értékadás* szabálya

vonatkozik. Egy változónak tehát vagy nincs értéke és ekkor a neve önmagát, mint karaktersorozatot képviseli hatáskörén belül mindenütt, vagy van értéke és ekkor a név mindenütt ezt az értékkomponenszt jelenti. Az értékkomponens nem írható felül. A tiszta Prologban egy változónak értéket a Prolog következtető gép adhat.

A változó hatásköre az a mondat, amelyikben szerepel a neve. Kivétel ez alól az a változó, amelynek neve `_` (ún. *névtelen változó*), amelynek minden előfordulása más-más változót jelöl.

Egy tiszta Prolog program futtatásának célja általában a kérdésekben szereplő változók lehetséges értékeinek meghatározása.

Általános Prolog konvenció, hogy az eredmény szempontjából érdektelen változók nevét aláhúzásjellel kezdjük.

Az összetett term alakja:

név(argumentum [, argumentum]...)

ahol az *argumentum* egy tetszőleges term, vagy egy aposztrófok közé zárt tetszőleges karaktersorozat lehet.

A tény egy összetett term és mint olyan, egy igaz állítás.

Egy szabály áll *fejből* és *törzsből* és közöttük valamilyen elhatároló áll (nálunk ez a `:` - lesz). A fej egy összetett term, a törzs egy vesszőkkel elválasztott összetett term sorozat (ezek predikátumok).

A szabály egy következtetési szabály: a fej akkor igaz, ha a törzs igaz. A vessző tehát itt egy rövidzár és műveletnek felel meg.

A kérdésnek csak törzse van.

A Prologban a deklarációk és direktívák

`:` - törzs

alakúak.

A Prolog állításaiban szereplő változók *univerzálisan*, a kérdésben szereplők viszont *egzisztenciálisan kvantáltak*.

Tehát a tények törzs nélküli szabályok, vagyis a törzs mindig *igaznak* tekinthető. A kérdés viszont fej nélküli szabály, azaz vagy azt kérdezzük, hogy a megoldási környezet mely elemei teszik igazzá a predikátumokat, vagy pedig csak egy „igen-nem” típusú kérdést teszünk föl.

A szabályok lehetnek rekurzívek.

Akárhány olyan szabály lehet, ahol a fej azonos, ilyenkor az *argumentumok* közötti kapcsolatot az egyes állítások által definiált kapcsolatok uniója határozza meg.

Példa:

apja(jános,ferenc). % ez egy tény

apja(ferenc,péter). % ez egy tény

nagyapja(X,Z):- apja(X,Y),apja(Y,Z). % ez egy szabály

nagyapja(Valaki,péter). % ez egy kérdés

A Prolog következtető gép a program futtatása során a memóriában egy *keresési fát* épít föl és jár be preorder módon. A fát teljes mértékben soha nem építi föl, mindig csak az aktuálisan kezelt út áll rendelkezésre, a bejárt csúcsot törli a feldolgozás után. A fa csúcsaiban a kérdés aktuális alakja áll, az éleket viszont az adott lépésben végrehajtott változóhelyettesítések címkézik.

A kérdés megválaszolásánál a tényeket és szabályokat a felírásuk sorrendjében használja fel, a megoldásnál alkalmazott technika pedig az *illesztés* és a *visszalépés*.

A megoldás lépései a következők:

1. A keresési fa gyökerében az eredeti kérdés áll. Induláskor ez az aktuális csúcs.
2. Ha az aktuális csúcsban a kérdés törzse üres, akkor megvan egy megoldás. Ezt kiírja a rendszer, és rákérdez, hogy a felhasználó akar-e további megoldásokat. Ha nem, akkor a programnak vége, ha igen, akkor folytatás 4-től.
3. Ha az aktuális csúcsban a kérdés törzse nem üres, akkor veszi a törzs első predikátumát, majd az első tényre végrehajt egy illesztést. Ha nem sikerül az illesztés, akkor veszi sorra a további tényeket és próbál azokra illeszteni. Ha sikerül valamelyik tényre illeszteni, akkor a fában létrehoz egy új csúcsot és abban a kérdés aktuális alakja úgy áll elő, hogy elhagyja az első predikátumot. Az élet címkézi az illesztéshez esetleg szükséges változóhelyettesítésekkel.

Ha egyetlen tényre sem sikerült illeszteni, akkor megpróbál illesztést találni a szabályok fejére. Ha van illeszkedés, akkor létrehoz egy új csúcsot a fában, az éleket ugyanúgy címkézi, és a kérdés új alakja úgy keletkezik, hogy a predikátumot felülírja az illeszkedő fejű szabály törzsével.

Ha nem illeszkedik egyetlen tény és egyetlen szabályfej sem, akkor a Prolog azt mondja, hogy zsákutcába jutott és a végrehajtás folytatódik 4-től, különben az új csúcs lesz az aktuális és folytatás 2-től.

4. Ez a visszalépés. Ha az aktuális csúcs a fa gyökere, akkor a program véget ér, nincs több megoldás (az eddigiekkel együtt esetleg egy sem). Különben törli a fában az aktuális csúcsot, és a megelőző csúcs lesz az aktuális. Egyben törli a két csúcsot összekötő él változóhelyettesítéseit. Ezután 3-tól folytatva megpróbál illesztést keresni az eddig felhasznált állításokat követő állítások segítségével.

Az illesztés algoritmus a következő:

1. Ha az illesztendő termsorozatok üresek, akkor vége (az illesztés sikeres), különben illesztí a két sorozat első elemeit 2 szerint, majd ha, azok illeszkednek, folytatódik az illesztés a sorozatok maradék elemeire 1 szerint.
2. Ha mindkét term állandó, akkor attól függően, hogy mint karaktersorozatok azonosak-e, az illesztés sikeres lesz, vagy meghiúsul.
3. Állandó és összetett term esetén az illesztés sikertelen lesz.
4. Két összetett term esetén az illesztés sikertelen, ha különbözik a nevük, vagy az argumentumaik száma. Különben az argumentumok sorozatai kerülnek illesztésre 1 szerint.
5. Ha mindkét term változó, bármelyik helyettesíthető a másikkal. Általában az állítás változói kapnak értéket.
6. Ha az egyik term változó, akkor az helyettesítődik a másik (állandó vagy összetett) termmel.

Nézzük meg a példánk végrehajtását. A keresési fa induláskor ilyen alakú:

nagyapja(Valaki, péter)

Az illesztési algoritmus 4-es pontja miatt a tényekre nem illeszthető a predikátum (eltér a név), de 4, 5, 6, miatt (változóhelyettesítések) a szabály fejére igen, a fa épül tovább (már figyelembe vettük a változóhelyettesítést és az egyszeres értékadást):

nagyapja(Valaki, péter)

↓ X ← Valaki
↓ Z ← péter

apja(Valaki, Y), apja(Y, péter)

Most az első predikátum 4 és 6 alapján illeszthető az első tényhez:

```

nagyapja(Valaki,péter)
  | X ← Valaki
  ↓ Z ← péter
  apja(Valaki,Y),apja(Y,péter)
    | Valaki ← jános
    ↓ Y ← ferenc
    apja(ferenc,péter)

```

4 és 2 alapján az első tény nem, de a második illeszkedik így:

```

nagyapja(Valaki,péter)
  | X ← Valaki
  ↓ Z ← péter
  apja(Valaki,Y),apja(Y,péter)
    | Valaki ← jános
    ↓ Y ← ferenc
    apja(ferenc,péter)
    ↓

```

A kérdés törzse üressé vált, megvan az első megoldás és ez jános. Ezt a Prolog kiírja, majd rákérdez a folytatásra. Ha folytatni akarjuk, akkor a visszalépések és a zsákutcák miatt a következő faszorozat alakul ki és több megoldást nem találunk:

```

nagyapja(Valaki,péter)
  | X ← Valaki
  ↓ Z ← péter
  apja(Valaki,Y),apja(Y,péter)
    | Valaki ← jános
    ↓ Y ← ferenc
    apja(ferenc,péter)

```

zsákutca

```

nagyapja(Valaki,péter)
  | X ← Valaki
  ↓ Z ← péter
  apja(Valaki,Y),apja(Y,péter)
    | Valaki ← ferenc           illesztés a 2. ténnyel
    ↓ Y ← péter
    apja(péter,péter)           zsákutca

nagyapja(Valaki,péter)
  | X ← Valaki
  ↓ Z ← péter
  apja(Valaki,Y),apja(Y,péter)  zsákutca

nagyapja(Valaki,péter)           zsákutca

```

A gyökérből kellene visszalépni, vége a futásnak, újabb megoldás nincs.

A predikátumok illesztése az eljárásorientált nyelvek eljáráshívásának felel meg. A hívás a predikátum, a meghívott eljárást illesztés választja ki, a paraméterkiértékelésnél sorrendi kötés és számbeli egyeztetés van, a paraméterátadást az illesztés helyettesíti és ezután a hívást felülírjuk a törzzsel (makrózás).

Ha az argumentum változó, az output paraméter, különben input paraméter.

Az „igen-nem” kérdés argumentumai között nem szerepel változó.

A gyakorlati problémák megoldásánál gyakran van szükség olyan megoldásokra (pl. számolni kell), amelyekre a tiszta logikai eszközök nem, vagy nem elég hatékonyan alkalmazhatók. Ezekre szolgálnak a Prolog *metalogikai* eszközei.

A Prologban van *kifejezés*, azonban ez csak bizonyos környezetekben kiértékelendő. A Prolognak vannak beépített operátorai (pl. aritmetikai és hasonlító operátorok), és a programozó is definiálhat saját operátorokat, sőt túlterhelheti a beépítetteket. Ezt direktívák segítségével szokás megtenni.

A Prologban a +, -, *, /, ** aritmetikai operátorok, a szokásos jelentéssel bírnak (az utolsó a hatványozás operátora). Ezek infix és prefix módon használhatók és természetesen szám operandusokon értelmezettek. Azonban ezek az operátorok összetett term nevek. Tehát a 2+3 kifejezés lényegében nem más, mint a +(2,3) összetett term. Ennek megfelelően egy ilyen kifejezés csak speciális szöveggörnyezetben kerül kiértékelésre ténylegesen.

Ilyen környezetet ad a kétoperandusú infix `is` operátor, amelynek jobb oldalán egy aritmetikai kifejezésnek kell állnia. Ez a kifejezés kiértékelődik, majd illesztésre kerül a bal oldali operandussal.

Ez az a szituáció, amikor az illesztés szabályainak alkalmazása miatt a programozó rendelhet értéket egy változóhoz. Ugyanis, ha az `is` bal oldalán egy érték nélküli változó áll, akkor az felveszi a kifejezés értékét.

Az illesztés miatt teljesen értelmetlen az `S is S+1` kifejezés. Ugyanis, ha `S`-nek van értéke, akkor az illesztés sikertelen lesz (`S` és `S+1` értéke nem azonos), ha nincs, akkor pedig hiba keletkezik (`S` nem létező értékéhez nem adhatunk hozzá 1-et).

A Prolog egyik beépített operátora a `!`, a *vágási operátor*. Operandus nélküli (tehát argumentum nélküli összetett term). Szerepe a megoldás keresésének vezérlésében van. Ha szerepel egy kérdés törzsében predikátumként, akkor az adott csomópontban „elvágja” a keresési fát. Ez annyit jelent, hogy ha a megoldás keresésénél ebből a csúcsból kellene visszalépni, akkor a program befejeződik. Szerepe alapvető a rekurzív szabályok alkalmazásánál, a végtelen rekurzió elkerülésénél.

Példaként lássuk a faktoriális kiszámításának szabályait:

```
faktoriális(0,X) :- !, X is 1.
```

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

Egy hatékony programozási nyelv nem lehet meg olyan eszközök nélkül, mint I/O, sztringkezelés, kivételkezelés, grafika. Ezek természetesen léteznek a Prologban is, azonban ezek logikán kívüli predikátumok, deklaratív értelmezésük nincs.

6. A FUNKCIONÁLIS PARADIGMA

A funkcionális paradigma középpontjában a *függvények* állnak. Egy funkcionális (vagy *applikatív*) nyelvben egy program típus-, osztály- és 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 eredendően interpreter alapúak, interaktívak, de tartalmazznak 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 *konfluensnek* 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 kifejezésekből áll.

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

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

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 beszélünk.

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 (ez lényegében szöveg szerinti paraméterátadás). 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 (ez lényegében név szerinti paraméterátadás).

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.

A funkcionális nyelvek eszközként tartalmaznak olyan függvényeket, melyek paramétere, vagy visszatérési értéke függvény (*funkcionálok*, vagy *magasabb rendű függvények*). Ez a procedurális 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 funkcionális nyelvek közül a Haskell egy erősen típusos, tisztán funkcionális, lusta kiértékelést megvalósító, a LISP egy imperatív eszközöket is tartalmazó, objektumorientált változattal (CLOS) is rendelkező, mohó kiértékelést valló 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. Két legelterjedtebb változata a Common Lisp és a Scheme. 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, párbeszédés üzemmódban 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 nagybetűket nem különbözteti meg.

A nyelvnek vannak foglalt szavai (pl. a speciális formák nevei). 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 alak jelenik meg.

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

A nyelv alap építőelemei az *atomok*. Egy atom lehet *numerikus atom* (vagy *szám*). Ez megfelel az eljárásorientált nyelvek numerikus literáljának. A LISP decimális számrendszert használ. A számok fajtái a következők:

- Valós
- Racionális
- Tört
- Egész
- Komplex

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 komplex szám zárójelekben szereplő valós és képzetes részből áll, előtte a # jellel.

A LISP a törtet belső ábrázolásban mindig *kanonikus* 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, 2/3, #(3.0 2.0), #(0 1).

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*. A nyelv kulcsszavai előtt &, vagy pedig : jel áll.

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, és nem más mint a hierarchikus lista (1. **Adatszerkezetek és algoritmusok**) absztrakt adatszerkezet nyelvi realizációja.

Példák listára:

```
() ; üres lista  
(Ez egy 5 elemű 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 ekkor kiértékelendő. A numerikus atom értéke önmaga, a változó értéke az aktuális érték. Lista esetén viszont 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* alakú kifejezés, az operátoroknak a függvénynevek, az operandusoknak az S-kifejezések felelnek meg.

A kifejezés értéke maga is S-kifejezés.

Egy változó értékül S-kifejezést vehet fel.

A LISP függvényei lehetnek fix és változó paraméterszámúak, így az operátorok egy része tetszőleges számú operandus esetén 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 paraméter kiértékelésnél mindig sorrendi kötés, fix paraméterek esetén 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 lehet *szimbolikus*, ekkor az aktuális paraméter nem értékelődik ki, hanem mint karaktersorozat kerül átadásra. Ezek a függvények meg tudják változtatni a paraméterüket, tehát **lehet** mellékhatásuk. Az ilyen függvényeket egyes LISP verziók *álfüggvényeknek* hívják. A CLOS ekkor *makróról* beszél, megkülönböztetve őket a *függvényektől*. A makrókról részletesebben l. 7.9. alfejezet.

7.2. CLOS beépített függvények

Aritmetikai 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

A további példákban mindenütt a `>` a promptjel, a LISP válasza az alatta levő sorban látható.

Példák:

```
> (- 3 2)
1
> (+ 3 4 5)
12
> (* 3 4 5)
60
```

Predikátumok

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

```
> ()
```

NIL

Sok LISP verzió az ún. általánosított logikai értékeket kezeli. Ezek azt mondják, 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:

```
> (integerp (/ 12 4))
```

T

```
> (ratiop (/ 12 5))
```

T

```
> (integerp (/ 12 5))
```

NIL

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 két 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

Logikai függvények

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

A not egy paraméterű, az and és az 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:

```
> (and)
```

```
T
```

```
> (or)
```

```
NIL
```

```
> (and 3)
```

```
3
```

```
> (and 0)
```

```
0
```

Feltételes konstrukciók

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

Az if speciális formának 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.

A cond makró nem fix paraméterszámú függvény, paramétereinek listák. Formája:

```
(COND [(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-lel tér vissza. Ugyancsak NIL lesz az értéke, ha paraméter nélkül hívjuk meg.

A quote speciális forma

Egy paramétere van, visszatérési értéke az 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 példa ekvivalens az alábbival:

```
> '(+ 5 6)
```

7.3. Nevesített konstans, változó, saját függvény

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

Példa:

```
(defconstant pi 3.14)
```

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

Példák:

```
> (defvar a 8)
```

A

```
> (defvar a)
```

A

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

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

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 a 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` makró visszatérési értéke egyes verziók szerint *határozatlan*, a CLOS-ban a függvény nevét adja vissza.

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 paraméter számú 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. 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 abszolút érték függvényt implementálja logikai függvények segítségével (az általánosított logikai értékek felhasználásával):

```
(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 (1- n))))))
```

7.4. Listák

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

Példák:

```
> (defvar a '(+ 2 3 4))
A
```

```

> 'a
A
> a
(+ 2 3 4)
> (car a)
+
> (cdr a)
(2 3 4)
> (car (cdr '(a b c)))
B

```

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

```

(car (car x)) = (caar x)
(car (cdr x)) = (cadr x)
(cdr (car 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 `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.

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 '((a b) 1 (x)))
3
> (reverse '((a b c) (d e)))
((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:

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

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:

```
> (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 alkalmazva rá a `cdr` függvényt, az utolsó mindig `NIL`-lél tér vissza. A *nem valódi 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.

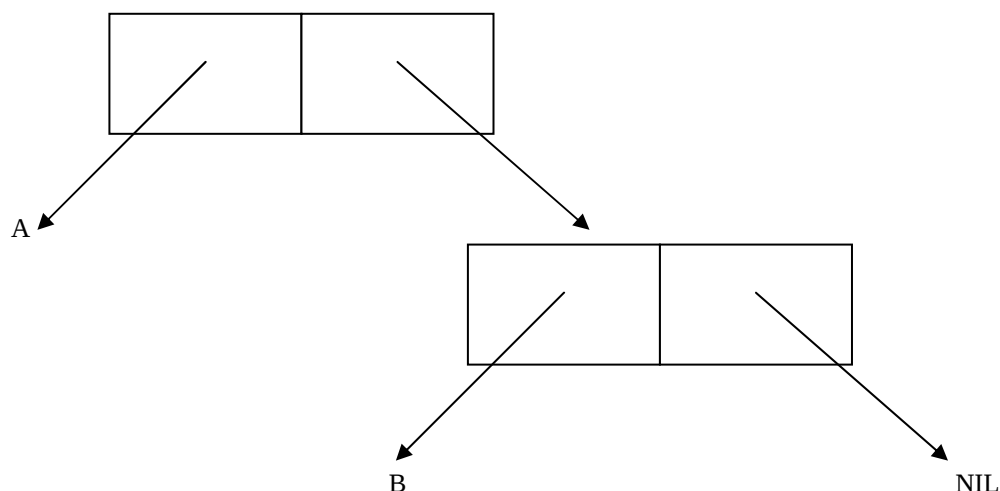
Az `endp` predikátum a lista végét teszteli. Értéke igaz, ha paramétere az üres lista és `NIL`, ha nem.

A listák memóriában történő ábrázolása a következőképpen történik:

A lista mindig egy pointer párból 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)
```

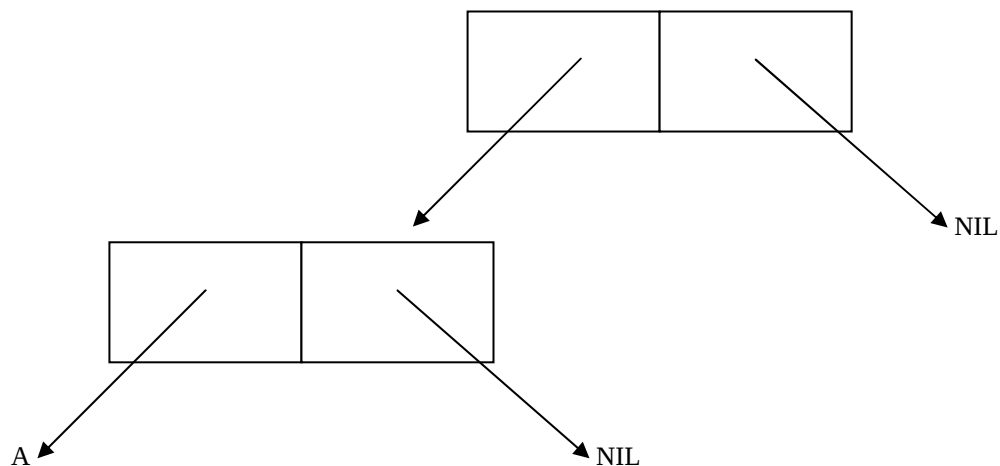
hatására az alábbi ábrázolású lista jön létre:



A

```
> (cons (cons 'a NIL) NIL)
((A))
```

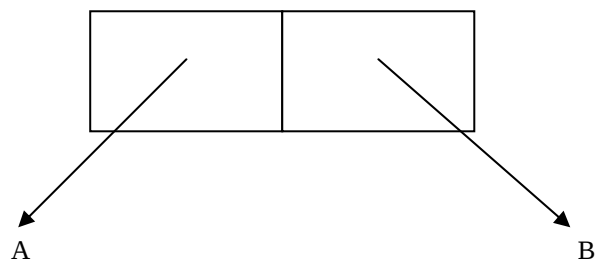
lista viszont a következőképpen néz ki:



A nem valódi

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

pontozott pár szerkezete viszont:



A lista feje és a nem valódi lista vége vagy atom, vagy lista. A valódi lista farka mindig lista. 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). Az utolsó paraméter lehet nem lista is.

Példák:

```
> (append '(a b) '(c d))  
(A B C D)  
> (append '(a b) c)  
(A B . C)
```

A listák kezelésére szolgálnak a következő predikátumok:

<code>symbolp</code>	igaz, ha paramétere szimbólum
<code>consp</code>	igaz, ha paramétere valódi lista
<code>atom</code>	igaz, ha paramétere nem valódi lista
<code>listp</code>	igaz, ha paramétere lista
<code>null</code>	igaz, ha paramétere az üres lista

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

Eszközök összehasonlítására szolgál az `eq` és `equalp` függvény. Az `eq` függvény akkor ad igaz értéket, ha pontosan azonos két eszköz (vagyis pontosan azonos memóriaterületen vannak!). Az `equalp` viszont akkor igaz, ha paramétereik mint 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
```

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_bovit (n lista)
  (cond ((numberp n) (cons n lista))
        (T lista)))
```

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

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

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

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

6. Rendszerezünk 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)))))
```

```
(defun beszur (ertek sorozat)
  (cond ((null sorozat) (list ertek))
        ((< ertek (car sorozat))(cons ertek sorozat))
        (T (cons (car sorozat)
                  (beszur ertek (cdr sorozat))))))
```

7.5. Lokális és globális eszközök

A CLOS-ban a nevesített konstansok, a függvények és a `defvar` makróval létrehozott változók nevei *globálisak*, mindenhol 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 `defparameter` 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])...)
  törzs)
```

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

Példa: Egy csak számokat tartalmazó, tetszőleges (akárhányszorosan egymásba ágyazott) 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, amelynek paraméterei: egy függvény formális paraméter listája és törzse. Ezzel tulajdonképpen egy lokális, a definíciójánál azonnal meghívásra is kerülő, a globális függvények közé fel nem veendő függvény használatára nyílik lehetőség.

Példa:

```
> ((lambda (n)(+ n 2)) 5)
7
```

Az `flet` makró lehetővé teszi saját függvényen belül *lokális nem rekurzív*, 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 megadása 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 öndefiníáló 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` (újsor).

A sztring karakterek egydimenziós tömbjeként értelmezendő. 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 predikátumok és egyéb függvények segítik. Ízelítőül néhány ezek közül:

<code>stringp</code>	igaz, ha paramétere sztring
<code>characterp</code>	igaz, ha paramétere karakter
<code>alphanumericp</code>	igaz, ha paramétere alfanumerikus karakter
<code>upper-case-p</code>	igaz, ha paramétere nagybetű
<code>char-code</code>	visszaadja a karakter ASCII kódját
<code>length</code>	megadja egy sztring hosszát
<code>concatenate</code>	összefűzi a sztringeket
<code>code-char</code>	visszaadja az adott kódú karaktert

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:

<code>*standard-input*</code>	alapértelmezett bemenet (billentyűzet)
<code>*standard-output*</code>	alapértelmezett kimenet (képernyő)
<code>*terminal-io*</code>	a felhasználói terminál
<code>*query-io*</code>	a felhasználói interakciók javallott adatfolyama
<code>*debug-io*</code>	az interaktív belövés adatfolyama
<code>*trace-output*</code>	a <code>trace</code> makró kimenete
<code>*error-output*</code>	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 szekvenciális kiértékelés megváltoztatását célozza.

A `prog1` makró visszatérési értéke az első paraméterének értéke, a `prog2` makróé pedig a második paraméterének értéke. A `progn` speciális forma az alapértelmezett viselkedést mutatja.

Példa:

```
>(prog1 (setf n 3) (setf n (1+ n)))
3
> n
4
```

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 listából 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 cikluslépés 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 nem üres 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 valódi lista hosszát meghatározó függvény rekurzív és iteratív változatát.

```
a. (defun r_length (l)
    (cond ((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 a **return** makró 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))
                  (incf szamlalo))))))
```

A CLOS ismeri a *blokk* fogalmát is, ezt a `block` speciális forma 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 egy (`return-from név érték`) speciális forma, 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 speciális forma, ami egy GOTO-utasításnak felel meg. Használhatjuk a `return` makró is.

Példa:

```
(defun fakt(n)
  (prog ((k n) (1 1))
    kovetkezo
    (cond ((zerop k) (return 1)))
    (setf l (* k 1))
    (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 megkülönböztethetetlen eszközökkel bővíti.

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

Az igazi nyelvkiterjesztő eszközök azonban a saját makrók.

Saját makró 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 függvélynél először kiértékelődik az összes paraméter, majd kiértékelődik a törzs. A makrónál először kiértékelődik a törzs, majd kiértékelődik az eredményül kapott új törzs.

Egy makródefinícióban szinte 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 ugyanezt 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 speciális forma, akkor a hozzátartozó kód alapján történik a lista kiértékelése.
2. Különben, ha a szimbólum egy makró neve, akkor végrehajtja a makrót, és kiértékeli az eredményt.
3. Különben a szimbólumot függvéynévként tekinti.

A CLOS speciális formái a következők:

<code>block</code>	<code>catch</code>	<code>declare</code>	<code>eval-when</code>
<code>flet</code>	<code>function</code>	<code>go</code>	<code>if</code>
<code>labels</code>	<code>let</code>	<code>let*</code>	<code>macrolet</code>
<code>multiple-value-call</code>	<code>multiple-value-prog1</code>	<code>progn</code>	<code>progv</code>
<code>quote</code>	<code>return-from</code>	<code>setq</code>	<code>tagbody</code>
<code>the</code>	<code>throw</code>	<code>unwind-protect</code>	

Itt jegyezzük meg, hogy a speciális formák egy része (pl. `let`, `labels`) makróként van megvalósítva.

Példák:

1. Írjunk makrót, amely megcseréli két paraméterének értékét.

```
(defmacro 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.

```
(defmacro ismetel (n &rest mag)
  `(do ((i ,n (- i 1)))
    ((<= i 0) nil)
    ,@mag))
```

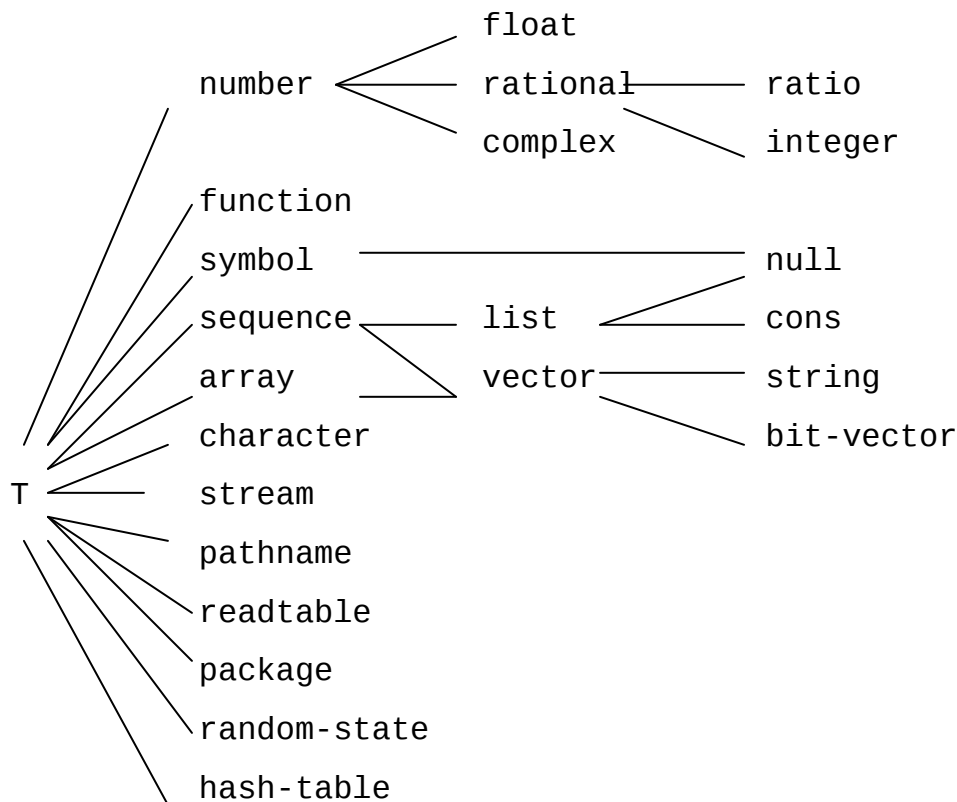
7.10. Objektumorientált eszközök

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

A CLOS-ban nincs bezárási eszközzrendszer. Az öröklődés többszörös.

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.

A beépített osztályok (típusok) hierarchiája a következő:



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:

<code>:reader</code>	az attribútum lekérdező metódusának a neve
<code>:writer</code>	az attribútum beállító metódusának a neve
<code>:accessor</code>	egy olyan metódus neve, amellyel az attribútumot egyaránt le lehet kérdezni és be lehet állítani
<code>:allocation</code>	értéke <code>:instance</code> (ez az alapértelmezett) vagy <code>:class</code> lehet. <code>:instance</code> esetén az adott attribútum értéke példányonként különbözhet, <code>:class</code> esetén viszont azonos (vagyis akkor ez egy osztály szintű attribútum)
<code>:initarg</code>	értéke egy szimbólum, amelynek segítségével az attribútum kezdőértéke a példányosításakor beállításra kerül

`:initform` az attribútum alapértelmezett értéke, 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 megadhatók a beállító és lekérdező metódusok nevei, ezek automatikusan létrejönnek. Az osztályhoz kapcsolódó további függvényeket, makrókat az osztálydefiníciótól függetlenül kell létrehozni.

Az *osztály_opciók* a következők lehetnek:

`:default-initargs` Az attribútumok kezdőértékét állítja be, az egyes attribútumoknál 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, ennek a `standard-class` alosztályának kell lennie.

`:documentation sztring` Dokumentációs megjegyzés.

Egy osztály példányosítása a `make-instance` generikus 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 igen lényeges az osztályok esetén, a viselkedésmódot alapvetően ezek segítségével adhatjuk meg.

Példák:

```
(defclass szemely( )
  ((nev :initarg :nev :reader szemely-neve)
   (eletkor :initform 0 :accessor szemely-kora))
  (:documentation "Saját osztály"))
```

A szemely osztálynak nincs szuperosztálya. Két attribútuma van, ezek közül a nev-hez csak lekérdező, a kor-hoz lekérdező és beállító metódus is van definiálva.

```
(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> ;implementációfüggő
> (szemely-kora x)
0
> (szemely-neve x)
KISS
> (setf (szemely-kora 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.

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

Hozzuk most létre a szemely egy alosztályát:

```
(defclass programozo (szemely)
  ((nyelvek :initform NIL :initarg :nyelvek
            :accessor nyelv))
```

```
(:documentation "Ez egy alosztály"))
(defun 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 személy osztály definícióját és definiáljuk azt át a következőképpen:

```
(defclass személy( )
  ((nev :initarg :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)
PROGRAMOZO
> (defvar y (személy_kons 'NAGY))
#<SZEMELY: NAGY (Kora: 0)>
> (munkakor y)
PROGRAMOZO
> (setf (munkakor x) 'Kereskedo)
KERESKEDO
> (munkakor y)
KERESKEDO
```

A CLOS *automatikusan* konvertálja a régi osztály példányait az új osztály példányaivá, átalakítva 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
- **standard-class**: a defclass makróval definiált saját osztályok alapértelmezett metaosztálya
- **structure-class**: a defstruct makróval definiált *rekordok* metaosztálya (nem foglalkozunk vele)

Egy osztály metaosztályát, vagy egy objektum definiáló osztályát a **class-of** segítségével kérdezhetjük le.

A CLOS-ban kitüntetett szerepű a `standard-object` osztály, amely a `standard-class` metaosztály példánya és az összes `standard-class` példány őssztálya.

A CLOS-ban a generikus függvények és a metódusok is objektumok (metaobjektumok), osztályaik a `standard-generic-function` és a `standard-method`.

A CLOS-ban lehetőség van a metaosztályok kiterjesztésére a *metaobject protocol* (MOP) segítségével. Ez a témakör meghaladja jelen jegyzet kereteit, részletesen lásd *G. Kiczales – J. des Rivières – D. G. Bobrow: The Art of The Metaobject Protocol, MIT Press, 1991.*

7.11. Általánosított függvények

A függvények a nyelvnek ugyanolyan eszközei, mint a szimbólumok vagy a listák. A `function` függvénynek 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, a 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 *funkcionáloknak*.

Példa:

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

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

```
(defun append (&rest listak)
  (cond ((endp listak) '())
        ((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 szintjét biztosítja. Például segítségével megírhatunk egy általános leválogató függvényt, amely egy lista elemei közül azokat írja át egy másik listába, amelyek egy predikátumot igazgá tesznek. A függvény:

```
(defun levalogat (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:

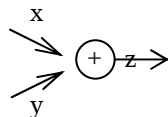
```
(defun negativ (lista) (levalogat lista 'minusp))
```

8. ADATVEZÉRELT PARADIGMA, ADATFOLYAM NYELVEK

Az adatvezérelt paradigma középpontjában az abszolút párhuzamosság áll. Tagadja a Neumann architektúrát, azt mondja, hogy az alapvető probléma a szekvenciális működésű processzor. A paradigma szerint minden algoritmust párhuzamos algoritmusként kell tekinteni, és az algoritmust realizáló programnak működés közben minden párhuzamosan végrehajtható tevékenységet egyszerre kell végrehajtania. Tehát totális párhuzamosságra kell törekedni.

A szekvenciális algoritmus leírásának egyik eszköze a folyamatábra. Az adatvezérelt paradigmában a párhuzamos algoritmust szintén egy irányított gráffal írjuk le. Azonban ez a gráf *aktív*, benne az élek mentén a nyilak irányában adatcsomagok, ún. *tokenek* mozognak. A gráf csomópontjai jelentik a tokenekkel végrehajtandó műveleteket. Az adatcsomagok tetszőleges adatszerkezetet reprezentálhatnak, és a műveletek tetszőleges bonyolultságúak lehetnek. A csomópontok figyelik a bevezető éleket, és ha azok mindegyikén megérkezett egy-egy token, akkor azonnal működésbe lépnek, elvégzik a hozzájuk rendelt műveletet, az eredményt (mint token) a kimenő élre rakják, és az továbbhalad a gráfban.

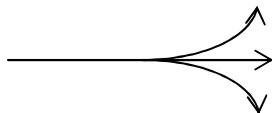
Az alábbi gráfrészlet



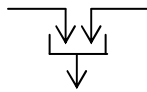
megfelel a $z=x+y$ értékadó utasításnak.

A gráf összeállításának megvannak a szintaktikai szabályai:

- Egyetlen csomópontot kivételével minden csomópontból egyetlen él indul, de egy él akárhányfelé ágazhat, tehát a token a gráf bármely másik csomópontjához eljuthat. Ezt ábrázolja a „villa”:



- A gráf különböző pontjaiból érkezhettek a tokenek ugyanahhoz a csomóponthoz, viszont élek nem végződhetnek élen. Szükséges egy gyűjtő szimbólum:



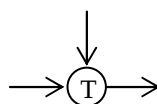
- A gráfban egyetlen olyan él lehet, amely nem csomópontból indul, ezen jut be a gráfba az input token, amely a bemenő adatokat tartalmazza.
- A gráfban akárhány olyan él lehet, amely nem csomópontból és gyűjtő szimbólumhoz vezet, hanem kivezet a gráfból. Ezek valamelyikén jelenik meg az output token, az algoritmus eredménye.

A paradigma algoritmikus, tehát kellene speciális, vezérlést leíró csomópontok:

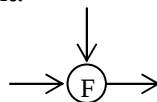
- Vizsgálat: tartalma egy feltétel, egy logikai értékű tokent állít elő, attól függően, hogy az érkező tokenre teljesül-e a feltétel vagy sem.



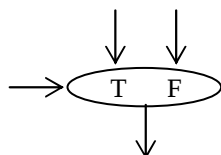
- Igaz kapu: két bemenő- és egy kimenő éle van. A bemenő élek egyike egy logikai értékű token, a másik értéke tetszőleges. Ha a logikai token értéke igaz, akkor a másik tokent átengedi, egyébként nem, a kapu lezár.



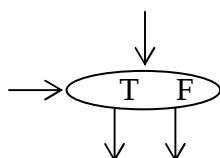
- Hamis kapu: hasonló az igaz kapuhoz, annyi különbséggel, hogy hamis esetben engedi át a másik input tokent.



- Kiválasztó: az igaz és hamis kapu kombinációja. Lényeges a szerkezet is: 2+1 bemenő és egy kimenő éle van. Az oldalsó 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 a T-be mutató tokent teszi a kimenetre, míg ha hamis, akkor a másikat.



- Elosztó: két bemenő tokenet és két kimenő tokenet tartalmaz (ez az egyetlen több kimenő éllel rendelkező csomópont). A bemenő tokenek közül az egyik, a vezérlő token csak logikai értéket hordozhat. Ha a vezérlő token értéke igaz, akkor a másik token a T-ből induló élen jelenik meg, ellenkező esetben a másodikon



Az általános csomópont ellipszis alakú, benne a hozzá tartozó művelet valamilyen formális leírása található.

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

Az adatvezérelt paradigmán belül többféle adatvezérelt modell létezik, attól függően, hogy milyen megszorításokat teszünk a gráfra, a tokenek felépítésére és azok gráfon belüli mozgására. Az egyes modellek más-más architektúrák és adatfolyam nyelvek tervezéséhez vezettek. A modellek a következők:

Alapmodell

A gráf aciklikus, csak horizontális párhuzamosságot enged meg, vertikálisat nem. Az egy input tokenel indított tokensorozat hullámfrontszerűen terjed végig a gráfon. Később indított tokensorozat nem előzhet meg korábban indított tokensorozatot. A hullámfront mereven összeköti a tokeneket, nem hajolhat el. Csak adatvezérlésen alapuló műveletek értelmezhetők, ciklus, rekurzió és egyéb programozási fogalmak nem. A modell determinisztikus.

Denis-féle modell

A gráf ciklikus. A hullámfrontok megmaradnak, de megköveteljük, hogy a gráfban egy csomópont csak akkor működhessen, ha nincs az output élen token, vagyis két hullámfront között mindig legyen csomópont. Itt is igaz, hogy később indult hullámfront nem előzhet meg

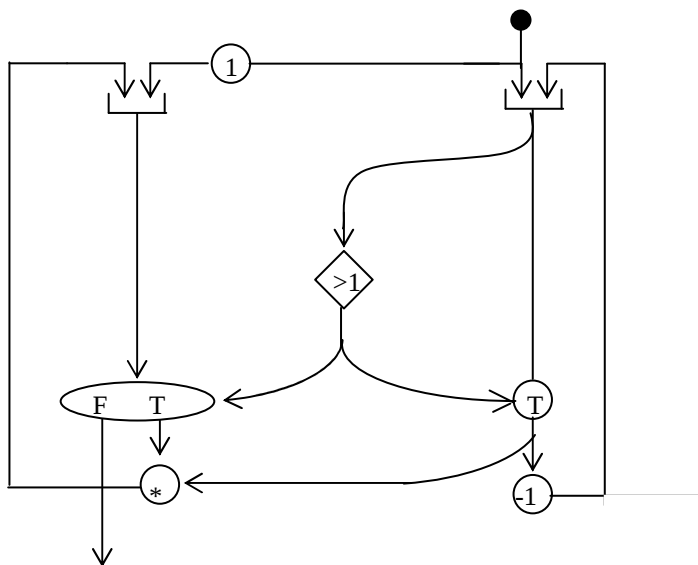
korábbi hullámfrontot. A gráf ciklusaiban egyszerre csak egy token tartózkodhat. A hullámfrontok tehát „elhajolhatnak”, ezáltal egy korlátozott vertikális párhuzamosság jöhet létre. Értelmezhetők az alapvető programnyelvi fogalmak. A modell determinisztikus.

Színezett modell

Jelenleg az egyik legfejlettebb adatvezérelt modell. Nem determinisztikus. Nincs hullámfront, viszont az ugyanazon input tokenel indított tokensorozatot azonos színűre festjük. Egy csomópont akkor működik, amikor az összes bemenő élen megjelennek az azonos színű tokenek (addig a más színűeket pufferelem). Ekkor végrehajtja a műveletet, és adott színű tokenek rak a kimenő éle. Értelmezhető minden programnyelvi fogalom. A párhuzamosság és adatvezéreltség totális, sőt, a problémaosztály minden feladata egyszerre oldható meg.

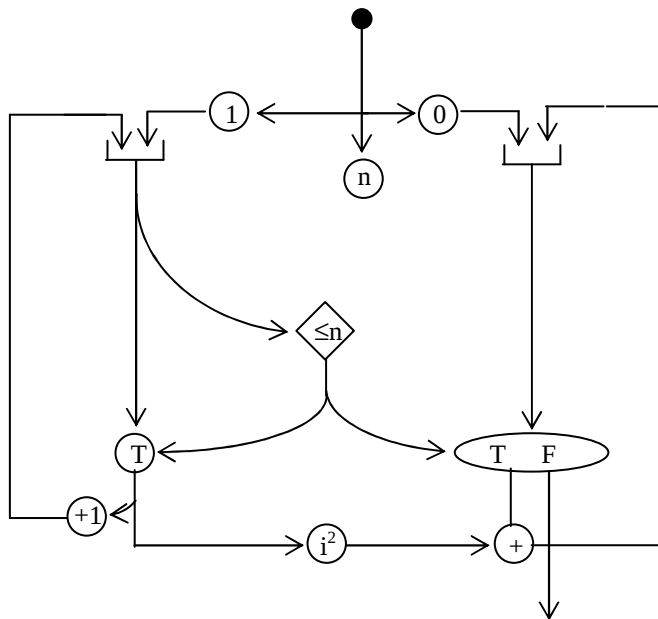
Lássunk most két színezett modellen alapuló gráfot!

Az $n!$ színezett gráfja:



Az első n természetes szám négyzetösszege:

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



Az adatfolyam nyelvek

Speciális algoritmikus nyelvek, amelyek valamelyik adatvezérelt modellhez lettek tervezve. Általános jellemzőik:

- Gyakran funkcionális tulajdonságaik vannak. Az általános műveletet leíró csomópontokat függvényekkel reprezentálják. Ezeknek a függvényeknek nincs mellékhatásuk. A paraméterátadás mindig érték szerinti. Általában nem használnak globális változókat, de ha mégis, akkor az egyszeres értékadás érvényes. A változó értékének módosítása nem megengedett. Csak lokális adattér van, az adatok azonnal felhasználásra kerülnek.
- Általában a rekurzió alapeszköz.
- Általában fordítóprogram-orientáltak, és a gépi kódjuk a megfelelő modell gráfja (grafikus gépi kód!). A hordozhatóság nem gond, hiszen minden azonos modellen felépülő nyelv ugyanazt a gépi kódot generálja. Ez a gráf matematikailag automatikusan könnyen kezelhető. A programhelyességbizonyítás automatikus, és nagyon egyszerű.
- A programfejlesztés nagyon egyszerű. Megírunk egy primitív programot, és azt transzformáljuk bizonyítottan helyes másik programmá automatizált módon.

Adatfolyam programnyelvekhez léteznek olyan fordítók, amelyek Neumann architektúrára fordítanak.

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

A VAL a Denis féle modellt megvalósító nyelv, a faktoriális számítás programja az alábbi:

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

A LUCID a színezett modellen alapul, a faktoriális programja:

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

Sajnos a paradigma mögé nem sikerült olyan architektúrát megalkotni, amely a gyakorlatban beváltotta volna a hozzáfűzött reményeket, ilyenek csak prototípus szinten léteznek.

IRODALOMJEGYZÉK

- Bergin, T. J. – Gibson, R. G.: History of Programming Languages, Addison-Wesley, 1996.
- Gábor András, Fazekas Imre: Java példatár, elektronikus jegyzet, 2004,
- Goldberg, A. – Robson, D.: Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading, 1983.
- A. Hejlsberg, S. Wiltamuth, P. Golde: The C# Programming Language. Addison Wesley, 2006.
- Horowitz, E.: Magasszintű programnyelvek, Műszaki, 1987.
- IBM Smalltalk Programmer's Reference. Introduction to Object-Oriented Programming with IBM Smalltalk.
- Juhász István: Magas szintű programozási nyelvek 1, elektronikus jegyzet, 2008,
- Marcotty, M. – Ledgard, H.: The World of Programming Languages, Springer-Verlag, 1987.
- Meyer, B.: EIFFEL. The Language, Prentice Hall, New York, 1992.
- Meyer, B.: Object-oriented Software Constructions, Prentice Hall, 1988.
- Nash, T.: C# 2008 (fordító: Endródi Gergely), Panem Könyvkiadó Kft, Budapest, 2009.
- Nyékiné Gaizler Judit(szerk.): Programozási nyelvek, Kiskapu, 2003.
- Nyékiné Gaizler Judit(szerk.): Java 2 útikalauz programozóknak, ELTE TTK Hallgatói Alapítvány, Budapest, 2000.
- Pratt, T. W.: Programming Languages. Design and Implementation, Prentice Hall, 1984.
- Sebesta, R. W.: Concepts of Programming Languages, Addison-Wesley, 2008.
- Sethi, R.: Programming Languages, Concepts and Constructs, Addison-Wesley, 1996.
- Slade, S.: Object-Oriented Common LISP, Prentice Hall, 1998.
- Steele, G. C.: Common Lisp. The Language, Digital Press, 1990.
- Teufel, B.: Organization of Programming Languages, Springer-Verlag, 1991.
- Watt D. A.: Programming Language Design Concepts, Wiley, 2006.
- Vég Csaba – Juhász István: Java –Start!, Logos 2000, Debrecen, 1999.

www.cetus-links.org/oo_clos.html
www.clos.org
www.eiffel.com
www.haskell.org
java.sun.com
msdn.microsoft.com/en-us/vcsharp/default.aspx
www.engin.umd.umich.edu/CIS/course.des/cis400/simula/simula.html
www.smalltalk.org
www.interactiva.org/Di/English/Dataflow/Computers/Programming/Languages
pauillac.inria.fr/~deransar/prolog/docs.html
www.sics.se/isl/sicstuswww/site/index.html