Juhász István

PROGRAMOZÁS 1

mobiDIÁK könyvtár

Juhász István

Programozás 1

mobiDIÁK könyvtár

SOROZATSZERKESZTŐ
Fazekas István

Juhász István

Programozás 1

Egyetemi jegyzet
Első kiadás

mobiDIÁK könyvtár

Debreceni Egyetem
Informatikai Intézet

Lektor

Pánovics János
Debreceni Egyetem

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

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

TARTALOMJEGYZÉK

ELŐSZÓ

Jelen jegyzet a Debreceni Egyetem Informatika tanár, Programozó matematikus és Programtervező matematikus szakán alapozó tárgy, a **Programozás 1** elméleti anyagát tartalmazza. A tantárgy előfeltétele **Az informatika alapjai** tárgy. Ez a jegyzet is sok helyen támaszkodik az ott elsajátított alapismeretekre. Az ajánlott tantervi háló szerint a tárggyal párhuzamosan kerül meghirdetésre az **Operációs rendszerek 1** és az **Adatszerkezetek és algoritmusok** című tárgy. Ezekkel nagyon szoros a kapcsolat, gyakoriak az áthivatkozások. A **Programozás 1** tárgyhoz közvetlenül kapcsolódik a **Programozás 2** tárgy, ezek együtt alkotnak szerves egészet.

A jegyzet megírásának időpontjában a tárgy gyakorlatán a C a kötelező nyelv. Ez magyarázza, hogy ezzel a nyelvvel részletesebben foglalkozunk.

A tantárgy gyakorlatán Kósa Márk és Pánovics János **Példatár a *Programozás 1* tárgyhoz** című elektronikus jegyzete használható.

1. BEVEZETÉS

1.1. Modellezés

Az ember már igen régóta törekszik a valós világ megismerésére. A valós világban mindenféle objektumok (személyek, tárgyak, intézmények, számítógépes programok) vannak – nevezzük ezeket *egyedeknek*. Az egyedeknek egyrészt rájuk jellemző *tulajdonságaik* vannak, másrészt közöttük bonyolult kapcsolatrendszer áll fenn. Az egyedek reagálnak a körülöttük lévő más egyedek hatásaira, kapcsolatba lépnek egymással, információt cserélnek – vagyis *viselkednek*. Az egyes konkrét egyedeket egymástól tulajdonságaik eltérő értékei, vagy eltérő viselkedésük alapján különböztetjük meg. Ugyanakkor viszont a valós világ egyedei, *közös* tulajdonságaik *és* viselkedésmódjuk alapján *kategorizálhatók*, *osztályozhatók*.

A valós világ túlságosan összetett ahhoz, hogy a maga teljességében megragadjuk, éppen ezért a humán gondolkodás az *absztrakción* alapszik és ennek segítségével modellekben gondolkodunk. Az absztrakció lényege, hogy kiemeljük a *közös*, *lényeges* tulajdonságokat és viselkedésmódokat, az eltérőeket, lényegteleneket pedig elhanyagoljuk. Ezáltal létrejön a valós világ modellje, amely már nem az egyes egyedekkel, hanem az egyedek egy csoportjával, *osztályával* foglalkozik.

Az ember modelleket használ, amikor egy megoldandó problémán gondolkodik, amikor beszélget valakivel, amikor eszközt tervez, amikor tanít, amikor tanul és amikor megpróbálja megérteni az itt leírtakat.

A modellalkotás képessége velünk születik. Amikor a gyermek megismerkedik a világgal, akkor igazában azt tanulja meg, hogy a számtalan egyedi problémát hogyan lehet kezelhető számú problémaosztályra leszűkíteni.

Egy modellel szemben három követelményt szoktak támasztani:

1. *Leképezés követelménye*: Léteznie kell olyan egyednek, amelynek a modellezését végezzük. Ez az „eredeti egyed”.

2. *Leszűkítés követelménye*: Az eredeti egyed nem minden tulajdonsága jelenik meg a modellben, csak bizonyosak.
3. *Alkalmazhatóság követelménye*: A modellnek használhatónak kell lennie, azaz a benne levont következtetéseknek igaznak kell lenniük, ha azokat visszavetítjük az eredeti egyedre.

Az 1. követelmény nem jelenti szükségszerűen az eredeti egyed aktuális létezését, az lehet megtervezett (pl. egy legyártandó gép), kitalált (pl. egy regényalak), vagy feltételezett (pl. egy baktérium a Marson).

A 2. követelmény miatt a modell mindig szegényebb, viszont kezelhető (míg az eredeti egyed gyakran nem).

A 3. követelmény az, amiért az egész modellt egyáltalán elkészítjük. Az eredeti egyed igen gyakran nem is elérhető számunkra, ezért vizsgálatainkat csak a modellben végezhetjük.

A számítógépek megjelenése lehetővé tette az emberi gondolkodás bizonyos elemeinek automatizálását. Az informatika a modellezés terén is alapvető jelentőségre tett szert. Az egyedek tulajdonságait számítógépen *adatokkal*, a viselkedésmódot pedig *programokkal* tudjuk kezelni – ezzel természetesen szintén egyfajta modellt megadva. Így beszélhetünk *adatmodellről* és *funkcionális modellről* (*eljárásmodellről*). Ez a megkülönböztetés azonban csak számítógépes környezetben lehetséges, hiszen a modell maga egy és oszthatatlan. Ugyancsak ebben a közelítésben említhetjük az *adatabsztrakciót* és a *procedurális absztrakciót*, mint az absztrakció megjelenési formáit az informatikában.

1.2. Alapfogalmak

A számítógépek programozására kialakult nyelveknek három szintjét különböztetjük meg:

– gépi nyelv
– assembly szintű nyelv
– magasszintű nyelv

A **Programozás 1** és **Programozás 2** tárgyak a magasszintű nyelvek eszközeivel, filozófiájával, használatával foglalkoznak. A magasszintű nyelven megírt programot *forrásprogramnak*, vagy *forrásszövegnek* nevezzük. A forrásszöveg összeállítására vonatkozó formai, „nyelvtani" szabályok összességét *szintaktikai* szabályoknak hívjuk. A tartalmi, értelmezési, jelentésbeli szabályok alkotják a *szemantikai* szabályokat. Egy magasszintű programozási nyelvet szintaktikai és szemantikai szabályainak együttese határoz meg.

Minden processzor rendelkezik saját gépi nyelvvel és csak az adott gépi nyelven írt programokat tudja végrehajtani. A magasszintű nyelven megírt forrásszövegből tehát valamilyen módon gépi nyelvű programokhoz kell eljutni. Erre kétféle technika létezik, a *fordítóprogramos* és az *interpreteres*.

A fordítóprogram egy speciális szoftver, amely a magasszintű nyelven megírt forrásprogramból gépi kódú *tárgyprogramot* állít elő. A fordítóprogram a teljes forrásprogramot egyetlen egységként kezeli és működése közben a következő lépéseket hajtja végre:

– lexikális elemzés
– szintaktikai elemzés
– szemantikai elemzés
– kódgenerálás

A lexikális elemzés során a forrásszöveget feldarabolja lexikális egységekre (l. 2.2. alfejezet), a szintaktikai elemzés folyamán ellenőrzi, hogy teljesülnek-e az adott nyelv szintaktikai szabályai. Tárgyprogramot csak szintaktikailag helyes forrásprogramból lehet előállítani. A tárgyprogram már gépi nyelvű, de még nem futtatható. Belőle futtatható programot a *szerkesztő* vagy *kapcsolatszerkesztő* készít. A futtatható programot a *betöltő* helyezi el a tárban, és adja át neki a vezérlést. A futó program működését a *futtató rendszer* felügyeli. Az ezekkel kapcsolatos részletes ismereteket az **Operációs rendszerek 1**, a **Nyelvek és automaták 1** és a **Fordítóprogramok** tárgyak tárgyalják. Bennünket a továbbiakban csak a fordítóprogram működése és a *fordítási idejű* események (a szintaktika miatt), továbbá a futtató rendszer tevékenysége és a *futási időhöz* kapcsolódó események (a szemantika miatt) érintenek.

A fordítóprogramok általánosabb értelemben tetszőleges nyelvről tetszőleges nyelvre fordítanak. A magasszintű nyelvek között is létezik olyan, amelyben olyan forrásprogramot lehet írni, amely tartalmaz nem nyelvi elemeket is. Ilyenkor egy *előfordító* segítségével először a forrásprogramból egy adott nyelvű forrásprogramot kell generálni, ami aztán már feldolgozható a nyelv fordítójával. Ilyen nyelv például a C.

Az interpreteres technika esetén is megvan az első három lépés, de az interpreter nem készít tárgyprogramot. Utasításonként (vagy egyéb nyelvi egységenként) sorra veszi a forrásprogramot, értelmezi azt, és végrehajtja. Rögtön kapjuk az eredményt, úgy, hogy lefut valamilyen gépi kódú rutin.

Az egyes programnyelvek vagy fordítóprogramosak, vagy interpreteresek, vagy együttesen alkalmazzák mindkét technikát.

Minden programnyelvnek megvan a saját szabványa, amit *hivatkozási nyelvnek* hívunk. Ebben pontosan definiálva vannak a szintaktikai és a szemantikai szabályok. A szintaktika leírásához valamilyen formalizmust alkalmaznak, a szemantikát pedig általában természetes emberi nyelven (pl. angolul) adják meg. A hivatkozási nyelv mellett (néha vele szemben) léteznek az *implementációk*. Ezek egy adott platformon (processzor, operációs rendszer) realizált fordítóprogramok vagy interpreterek. Sok van belőlük. Gyakran ugyanazon platformon is létezik több implementáció. A probléma az, hogy az implementációk egymással és a hivatkozási nyelvvel nem kompatibilisek. A magasszintű programozási nyelvek elmúlt 50 éves történetében napjainkig nem sikerült megoldani a *hordozhatóság* (ha egy adott implementációban megírt programot átviszek egy másik implementációba, akkor az ott fut és ugyanazt az eredményt szolgáltatja) problémáját.

Napjainkban a programok írásához grafikus *integrált fejlesztői környezetek* állnak rendelkezésünkre. Ezek tartalmaznak szövegszerkesztőt, fordítót (esetleg interpretert), kapcsolatszerkesztőt, betöltőt, futtató rendszert és belövőt.

1.3. A programnyelvek osztályozása

Imperatív nyelvek:

- Algoritmikus nyelvek: a programozó mikor egy programszöveget leír, algoritmust kódol, és ez az algoritmus működteti a processzort.
- A program *utasítások* sorozata.
- Legfőbb programozói eszköz a *változó*, amely a tár közvetlen elérését biztosítja, lehetőséget ad az abban lévő értékek közvetlen manipulálására. Az algoritmus a változók értékeit alakítja, tehát a program a hatását a tár egyes területein lévő értékeken fejti ki.
- Szorosan kötődnek a Neumann-architektúrához.
- Alcsoportjai:
  - *Eljárásorientált nyelvek*
  - *Objektumorientált nyelvek*

Deklaratív nyelvek:

- Nem algoritmikus nyelvek.
- Nem kötődnek olyan szorosan a Neumann-architektúrához, mint az imperatív nyelvek.
- A programozó csak a problémát adja meg, a nyelvi implementációkba van beépítve a megoldás megkeresésének módja.
- A programozónak nincs lehetősége memóriaműveletekre, vagy csak korlátozott módon.
- Alcsoportjai:
  - *Funkcionális (applikatív) nyelvek*
  - *Logikai nyelvek*

Máselvű nyelvek:

Olyan nyelveket sorolunk ebbe a kategóriába, amelyek máshová nem sorolhatók.

Nincs egységes jellemzőjük. Általában tagadják valamelyik imperatív jellemzőt.

1.4. A jegyzetben alkalmazott formális jelölésrendszer

A továbbiakban a szintaktikai szabályok formális leírásához az alábbi jelölésrendszert használjuk:

Terminális: íráskép, ha a jelek betűk, akkor nagybetűs alak.

Nem terminális: kisbetűs kategórianevek, ha több szóból állnak, akkor a szavak között aláhúzás jellel.

Alternatíva: |

Opció: [ ]

Iteráció: …, mindig az előtte álló szintaktikai elem akárhányszoros ismétlődését jelenti.

A fenti elemekkel szintaktikai szabályok formalizálhatók. Ezek bal oldalán egy nem terminális áll, jobb oldalán pedig egy tetszőleges elemsorozat. A két oldalt kettőspont választja el. A terminálisokat és nem terminálisokat `Courier New` betűtípussal szedtük. Ezt alkalmazzuk a konkrét forrásprogramok megadásánál is. Amennyiben a formális leíró karakterek részei az adott nyelvnek, akkor azokat a formalizálásnál vastagon szedjük.

Példa:

```
számjegy: { 0 |1 |2 |3 |4 |5 |6 | 7 |8 |9 }

egész_szám: [{ + | - }] számjegy...
```

1.5. A jegyzet tárgya

A **Programozás 1** tantárgy az eljárásorientált nyelvek eszközeit, fogalmait, filozófiáját, számítási modelljét tárgyalja (ezen a fogalmak többsége persze változatlan, vagy módosított formában megvan más nyelvosztályokban is). Konkrétan elemzi a legfontosabb, a gyakorlatban szerepet játszó nyelvek (FORTRAN, COBOL, PL/I, Pascal, Ada, C) egyes elemeit. De a jegyzet ***nem nyelvleírás***! Az említett nyelvek eszközeinek csak egy részét tárgyalja, gyakran azokat is leegyszerűsített, nem teljes formában. A cél az, hogy a programozási nyelvekben használható eszközökről egy modellszintű áttekintést, egy általános absztrakt fogalomrendszert kapjunk, amely keretek között aztán az egyes nyelvek konkrét megvalósításai elhelyezhetők. A konkrét nyelvi ismeretek az adott nyelvet tárgyaló könyvekből és papíralapú illetve elektronikus dokumentációkból sajátíthatók el.

De bármely nyelven programozni megtanulni elméletben, „papíron” nem lehet. Ehhez sok-sok programot kell megírni és lefuttatni!

Jelentőségük miatt kiemelt figyelmet fordítunk a C és az Ada nyelvekre.

Végül itt jegyezzük meg, hogy az eljárásorientált programozási nyelvek általában fordítóprogramosak, csak ritkán interpreteresek.

2. ALAPELEMEK

Ebben a fejezetben a programozási nyelvek alapeszközeit, alapfogalmait ismerjük meg.

2.1. Karakterkészlet

Minden program forrásszövegének legkisebb alkotórészei a *karakterek*. A forrásszöveg összeállításánál alapvető *a karakterkészlet*, ennek elemei jelenhetnek meg az adott nyelv programjaiban, ezekből állíthatók össze a bonyolultabb nyelvi elemek. Az eljárásorientált nyelvek esetén ezek a következők:

– lexikális egységek
– szintaktikai egységek
– utasítások
– programegységek
– fordítási egységek
– program

Minden nyelv definiálja a saját karakterkészletét. A karakterkészletek között lényeges eltérések lehetnek, de a programnyelvek általában a karaktereket a következő módon kategorizálják:

– betűk
– számjegyek
– egyéb karakterek

Minden programnyelvben betű az angol ABC 26 nagybetűje. A nyelvek továbbá gyakran betű kategóriájú karakternek tekintik az `_` , `$` , `#` , `@` karaktereket is. Ez viszont sokszor implementációfüggő. Abban már eltérnek a nyelvek0, hogy hogyan kezelik az angol ABC kisbetűit. Egyes nyelvek (pl. FORTRAN, PL/I) szerint ezek nem tartoznak a betű kategóriába, mások (pl. Ada, C, Pascal) szerint igen. Ezen utóbbi nyelvek különböznek abban, hogy azonosnak tekintik-e a kis- és nagybetűket (Pascal), vagy különbözőeknek (C). A nyelvek

túlnyomó többsége a nemzeti nyelvi betűket nem sorolja a betű kategóriába, néhány késői nyelv viszont igen. Tehát ezekben például lehet „magyarul” írni a programot.

A számjegyeket illetően egységes a nyelvek szemlélete, mindegyik a decimális számjegyeket tekinti számjegy kategóriájú karakternek.

Az egyéb karakterek közé tartoznak a műveleti jelek (pl. `+`, `-`, `*`, `/`), elhatároló jelek (pl. `[`, `]`, `.`, `:`, `{`, `}`, `'`, `"`, `;`), írásjelek (pl. `?`, `!`) és a speciális karakterek (pl. `~`). A program szövegében kitüntetett szerepet játszik a szóköz, mint egyéb karakter (l. 2.3. alfejezet).

A hivatkozási nyelv és az implementációk karakterkészlete eltérő is lehet. Minden implementáció mögött egy-egy konkrét kódtábla (EBCDIC, ASCII, UNICODE) áll. Ez meghatározza egyrészt azt, hogy egy- vagy több-bájtos karakterek kezelése lehetséges-e, másrészt értelmezi a karakterek sorrendjét. Ugyanis nagyon kevés olyan hivatkozási nyelv van (pl. Ada), amely definiálja a karakterek közötti sorrendet.

2.2. Lexikális egységek

A lexikális egységek a program szövegének azon elemei, melyeket a fordító a lexikális elemzés során felismer és tokenizál (közbenső formára hoz). Fajtái a következők:

– többkarakteres szimbólum
– szimbolikus nevek
– címke
– megjegyzés
– literálok

2.2.1. Többkarakteres szimbólumok

Olyan karaktersorozatok, amelyeknek a nyelv tulajdonít jelentést és ezek csak ilyen értelemben használhatók. Nagyon gyakran a nyelvben operátorok, elhatárolók lehetnek. Például a C-ben többkarakteres szimbólumok a következők: `++`, `--`, `&&`, `/*`, `*/`.

2.2.2. Szimbolikus nevek

Azonosító: Olyan karaktersorozat, amely betűvel kezdődik, és betűvel vagy számjeggyel folytatódhat. Arra való, hogy a program írója a saját programozói eszközeit megnevezze vele, és ezután ezzel hivatkozzon rá a program szövegében bárhol. A hivatkozási nyelvek általában nem mondanak semmit a hosszáról, az implementációk viszont értelemszerűen korlátozzák azt.

A következők szabályos C azonosítók (a C-ben az _ betű kategóriájú):

```
x
almafa
hallgato_azonosito
SzemelyNev
```

A következő karaktersorozatok viszont nem azonosítók:

`x+y`	a + nem megengedett karakter
`123abc`	betűvel kell kezdődnie

Kulcsszó (alapszó, fenntartott szó, védett szó, foglalt szó): Olyan karaktersorozat (általában azonosító jellegű felépítéssel), amelynek az adott nyelv tulajdonít jelentést, és ez a jelentés a programozó által nem megváltoztatható. Nem minden nyelv (pl. FORTRAN, PL/I) ismeri ezt a fogalmat. Az utasítások általában egy-egy jellegzetes kulcsszóval kezdődnek, a szakmai szleng az utasítást gyakran ezzel nevezi meg (pl. IF-utasítás). Minden nyelvre nagyon jellemzőek a kulcsszavai. Ezek gyakran hétköznapi angol szavak, vagy rövidítések. Az alapszavak soha nem használhatók azonosítóként.

A C-ben például alapszavak a következők:

```
if, for, case, break
```

Standard azonosító: Olyan karaktersorozat, amelynek a nyelv tulajdonít jelentést, de ez az alapértelmezés a programozó által megváltoztatható, átértelmezhető. Általában az

implementációk eszközeinek (pl. beépített függvények) nevei ilyenek. A standard azonosító használható az eredeti értelemben, de a programozó saját azonosítóként is felhasználhatja.

2.2.3. Címke

Az eljárásorientált nyelvekben a végrehajtható utasítások (l. 4. fejezet) megjelölésére szolgál, azért, hogy a program egy másik pontjáról hivatkozni tudjunk rá. Bármely végrehajtható utasítás megcímkézhető.

A címke maga egy speciális karaktersorozat, amely lehet előjel nélküli egész szám, vagy azonosító. A címke felépítése az egyes nyelvekben a következő:

– COBOL: nincs.

– FORTRAN: maximum 5 jegyű előjel nélküli egész szám.

– Pascal: A szabvány Pascalban a címke maximum 4 számjegyből álló elő nélküli egész szám. Egyes implementációkban ezen kívül lehet azonosító is.

– PL/I, C, Ada: azonosító.

Eléggé általános, hogy a címke az utasítás előtt áll és tőle kettőspont választja el. Az Adában viszont a címke az utasítás előtt a << és a >> többkarakteres szimbólumok között szerepel.

2.2.4. Megjegyzés

A megjegyzés egy olyan programozási eszköz, amely segítségével a programban olyan karaktersorozat helyezhető el, amely nem a fordítónak szól, hanem a program szövegét olvasó embernek. Ez olyan magyarázó szöveg, amely a program használatát segíti, működéséről, megírásának körülményeiről, a felhasznált algoritmusról, az alkalmazott megoldásokról ad információt. A megjegyzést a lexikális elemzés során a fordító ignorálja. A megjegyzésben a karakterkészlet bármely karaktere előfordulhat és minden karakter egyenértékű, csak önmagát képviseli, a karakter-kategóriáknak nincs jelentősége.

Egy megjegyzés forrásszövegben való elhelyezésére háromféle lehetőség van:

– A forrásszövegben elhelyezhetünk teljes megjegyzés sort (pl. FORTRAN, COBOL). Ekkor a sor első karaktere (pl. C) jelzi a fordítónak, hogy a sor nem része a kódnak.

– Minden sor végén elhelyezhetünk megjegyzést. Ekkor a sor első része fordítandó kódot, második része figyelmen kívül hagyandó karaktersorozatot tartalmaz. Például Adában a `--` jeltől a sor végéig tart a megjegyzés.

– Ahol a szóköz elhatároló jelként szerepel (l. 2. 3. alfejezet), oda tetszőleges hosszúságú megjegyzés elhelyezhető, tehát ekkor nem vesszük figyelembe a sor végét. Ekkor a megjegyzés elejét és végét jelölni kell egy-egy speciális karakterrel vagy többkarakteres szimbólummal. Például a Pascalban `{és}`, a PL/I-ben és a C-ben `/* és */` határolja ezt a fajta megjegyzést.

A nyelvek egy része több fajta megjegyzést is alkalmaz.

A jó programozási stílusban elkészített programok szövege nagyon sok megjegyzést tartalmaz.

2.2.6. Literálok (Konstansok)

A *literál* olyan programozási eszköz, amelynek segítségével fix, explicit értékek építhetők be a program szövegébe. A literáloknak két komponensük van: *típus* és *érték*. A literál mindig önmagát definiálja. A literál felírási módja (mint speciális karaktersorozat) meghatározza mind a típust, mind az értéket.

Az egyes programozási nyelvek meghatározzák saját literálrendszerüket, nézzük ezeket sorban:

FORTRAN:
Egész literál:

`[{+|-}] számjegy [ számjegy ]...`

A hétköznapi egész szám fogalmát veszi át: pl. `+28`, `-36`, `111`. Mögötte fixpontos belső ábrázolási mód áll.

Valós literálok:
Mögöttük lebegőpontos belső ábrázolási mód áll.

– Tizedestört valós literál:

`{[{ +|-}].előjel_nélküli_egész |egész.[előjel_nélküli_egész]}`

Tehát szerepel benne a tizedespont. Például: `+.01`, `-3.4`, `-3.0`, `.3`, `28`.

– Exponenciális valós literál:

`{ tizedestört | egész }{E | D}egész`

`E`: rövid lebegőpontos, `D`: hosszú lebegőpontos. A hétnél több számjegyű számot hosszú lebegőpontos formában ábrázolja. Például: `1E3`, `-2.2D28`.

Komplex literál:

`(valós,valós)`

Például: `(3.2,1.4)`, ami nem más, mint a `3.2+1.4i` komplex szám.

Hexadecimális literál:

`Zhexa_számjegy[ hexa_számjegy ]...`

Arra való, hogy karaktereket tudjunk kezelni a FORTRAN-ban.

Logikai literál:

`.TRUE.`

`.FALSE.`

Szöveges literál:

– Hollerith konstans:

`nHkarakter[karakter]...`

Az `n` előjel nélküli egész, amely a karakterek számát adja meg. Hossza tetszőleges lehet, de legalább egy karakter szükséges. Például: `4HALMA`, `6HALMAFA` .

– Sztring literál:

`'karakter[karakter]...'`

COBOL:

Numerikus literál:

A FORTRAN exponenciális valós literáljának felel meg. Egy numerikus literál hossza az ős COBOL-ban 18 számjegyre volt maximálva.

Alfanumerikus literál:

Hossza maximum 100 karakter. Alakja:

`"karakter[karakter]..."`

PL/I:

Aritmetikai literálok:

– Valós literálok:

– – Decimális fixpontos literál:

Belső ábrázolása decimális. A FORTRAN tizedestört valós literáljának felel meg.

– – Decimális lebegőpontos literál:

A FORTRAN exponenciális valós literáljának felel meg, annyi különbséggel, hogy nem szerepelhet benne `D`.

– – Bináris fixpontos literál:

Alakja:

`{bitsorozat | [bitsorozat].bitsorozat}B`

Például: `1011.11B`, `11B`, `.01B`

– – Bináris lebegőpontos literál:

Ábrázolása decimális. Alakja:

`bináris_fixpontosE [{+|-}]számjegy [számjegy ]B`

Például: `1.1E33B`

– Imaginárius konstans:

`valósI`

Lánc literálok:

– Karakterlánc:

`[(n)]'[karakter]...'`

Az `n` előjel nélküli egész. Például: `(2)'BA'`, ami nem más, mint `'BABA'`.

– Bitlánc:

`[(n)]'[bit]... 'B`

Például: `'111001'B`.

Pascal:

Egész literál:

A FORTRAN egészének felel meg.

Tizedestört:

A FORTRAN tizedestörtje, de a tizedespont mindkét oldalán szerepelnie kell számjegynek.

Exponenciális:

Ugyanaz, mint a FORTRAN-ban, annyi különbséggel, hogy itt csak `E` betű van.

Sztring:

`'[karakter]...'`

C:

Rövid egész literálok:

– Decimális egész:

Megfelel az eddigi egész literálnak.

– Oktális egész:

Nyolcas számrendszerbeli egész, kötelezően `0`-val kezdődik. Például `011`.

– Hexadecimális egész:

Tizenhatos számrendszerbeli egész, `0X`-el, vagy `0x`-el kezdődik. Például `0X11`.

Előjel nélküli egész:

Alakja:

`rövid_egész{U|`**`u`**`}`

Hosszú egész literál:

Alakja:

`{rövid_egész | előjel_nélküli_egész}{L|`**`l`**`}`

Valós literálok:

– Hosszú valós (kétszeres pontosságú valós):

Megfelel a FORTRAN valósának, de nincs `D`.

– Rövid valós (egyszeres pontosságú valós):

`hosszú_valós{`**`f`**`|F}`

– Kiterjesztett valós (háromszoros pontosságú valós):

`hosszú_valós{`**`l`**`|L}`

Karakter literál:

`'karakter'`

Az adott karakter belső kódját képviseli, számolni is lehet vele. Egyes implementációk megengedik, hogy az aposztrófok között több karakter álljon.

Sztring literál:

"[karakter]..."

Ada:

Numerikus literálok:

– Egész literál:

számjegy[[_]számjegy]... [{E|e}[+]számjegy[számjegy]...]

Például: 223_222e8.

– Valós literál:

Megfelel a Pascal valós literáljának.

– Bázisolt literál:

alap#egész[.előjel_nélküli_egész]# [{E|e}[+|-]számjegy[számjegy]...]

Az alap a 2-16 számrendszer alapszámát adja meg decimálisan, az esetleges kitevő részben a számjegyek decimálisak. A # jelek közötti számjegyek viszont az alap számrendszer számjegyei. Például: 8#123.56#e-45, 16#FF#.

Karakter literál:

'[karakter]'

Például: 's'.

Sztring literál:

"[karakter]..."

2.3. A forrásszöveg összeállításának általános szabályai

A forrásszöveg, mint minden szöveg, sorokból áll. Kérdés, hogy milyen szerepet játszanak a sorok a programnyelvek szempontjából.

Kötött formátumú nyelvek: A korai nyelveknél (FORTRAN, COBOL) alapvető szerepet játszott a sor. Egy sorban egy utasítás helyezkedett el, tehát a sorvége jelezte az utasítás végét. Ha egy utasítás nem fért el egy sorban, azt külön kellett jelezni (mintegy semlegesíteni a sorvége hatását). Több utasítás viszont soha nem állhatott egy sorban. A sor bizonyos pozícióira csak bizonyos programelemek kerülhettek. Tehát a programozónak kellett igazodnia a feszes szabályokhoz.

Szabad formátumú nyelvek: Ezeknél a nyelveknél a sornak és az utasításnak semmi kapcsolata nincs egymással. Egy sorba akárhány utasítás írható, egy utasítás akárhány sorban elhelyezhető. A sorban tetszőleges helyen jelenhetnek meg az egyes programelemek. A sorvége nem jelenti az utasítás végét. Éppen ezért ezek a nyelvek bevezetik az utasítás végjelet, ez elég általánosan a pontosvessző. Tehát a forrásszövegben két pontosvessző között áll egy utasítás.

Az eljárásorientált nyelvekben a program szövegében a lexikális egységeket alapszóval, standard azonosítóval, valamilyen elhatároló jellel (zárójel, kettőspont, pontosvessző, vessző, stb.), vagy ezek hiányában egy szóközzel el kell választani egymástól. A fordítóprogram ez alapján ismeri föl azokat a lexikális elemzés során. Az eljárásorientált nyelvekben tehát a szóköz általános elhatároló szerepet játszik. A szóköznek nincs kiemelt szerepe a megjegyzésben és a sztring, valamint karakter literálokban. Itt, mint minden karakter, csak önmagát képviseli. Ahol egy szóköz megengedett elhatárolóként, oda akárhányat is írhatunk. Egyéb elhatárolók mellett is állhat szóköz, ez növeli a forrásszöveg olvashatóságát. A FORTRAN-ban a forrásszövegben bárhol akárhány szóköz elhelyezhető, ugyanis a fordítás azzal kezdődik, hogy a fordító a szóközöket ignorálja.

2.4. Adattípusok

Az adatabsztrakció első megjelenési formája az adattípus a programozási nyelvekben. Az adattípus maga egy *absztrakt* programozási eszköz, amely mindig más, *konkrét* programozási eszköz egy *komponenseként* jelenik meg. Az adattípusnak *neve* van, ami egy azonosító.

A programozási nyelvek egy része ismeri ezt az eszközt, más része nem. Ennek megfelelően beszélünk *típusos* és *nem típusos* nyelvekről. Az eljárásorientált nyelvek típusosak.

Egy adattípust három dolog határoz meg, ezek:

– tartomány
– műveletek
– reprezentáció

Az adattípusok tartománya azokat az elemeket tartalmazza, amelyeket az adott típusú konkrét programozási eszköz fölvehet értékként. Bizonyos típusok esetén a tartomány elemei jelenhetnek meg a programban literálként.

Az adattípushoz hozzátartoznak azok a műveletek, amelyeket a tartomány elemein végre tudunk hajtani.

Minden adattípus mögött van egy megfelelő belső ábrázolási mód. A reprezentáció az egyes típusok tartományába tartozó értékek tárban való megjelenését határozza meg, tehát azt, hogy az egyes elemek hány bájtra és milyen bitkombinációra képződnek le.

Minden típusos nyelv rendelkezik beépített (standard) típusokkal.

Egyes nyelvek lehetővé teszik azt, hogy a programozó is definiálhasson típusokat. A saját típus definiálási lehetőség az adatabsztrakciónak egy magasabb szintjét jelenti, segítségével a valós világ egyedeinek tulajdonságait jobban tudjuk modellezni.

A saját típus definiálása általában szorosan kötődik az absztrakt adatszerkezetekhez (l. **Adatszerkezetek és algoritmusok**).

Saját típust úgy tudunk létrehozni, hogy megadjuk a tartományát, a műveleteit és a reprezentációját. Szokásos, hogy saját típust a beépített és a már korábban definiált saját típusok segítségével adjuk meg. Általános, hogy a reprezentáció megadásánál így járunk el.

Csak nagyon kevés nyelvben lehet saját reprezentációt megadni (pl. ilyen az Ada). Kérdés, hogy egy nyelvben lehet-e a saját típushoz saját műveleteket és saját operátorokat megadni. Van, ahol igen, de az is lehetséges, hogy a műveleteket alprogramok realizálják. A tartomány megadásánál is alkalmazható a visszavezetés technikája, de van olyan lehetőség is, hogy explicit módon adjuk meg az elemeket.

Az egyes adattípusok, mint programozási eszközök önállóak, egymástól különböznek. Van azonban egy speciális eset, amikor egy típusból (ez az *alaptípus*) úgy tudunk származtatni egy másik típust (ez lesz az *altípus*), hogy leszűkítjük annak tartományát, változatlanul hagyva műveleteit és reprezentációját. Az alaptípus és az altípus tehát nem különböző típusok.

Az adattípusoknak két nagy csoportjuk van:

A *skalár* vagy *egyszerű* adattípus tartománya atomi értékeket tartalmaz, minden érték egyedi, közvetlenül nyelvi eszközökkel tovább nem bontható. A skalár típusok tartományaiból vett értékek jelenhetnek meg literálként a program szövegében.

A *strukturált* vagy *összetett* adattípusok tartományának elemei maguk is valamilyen típussal rendelkeznek. Az elemek egy-egy értékcsoportot képviselnek, nem atomiak, az értékcsoport elemeihez külön-külön is hozzáférhetünk. Általában valamilyen absztrakt adatszerkezet programnyelvi megfelelői.

2.4.1. Egyszerű típusok

Minden nyelvben létezik az *egész* típus, sőt általában egész típusok. Ezek belső ábrázolása fixpontos. Az egyes egész típusok az ábrázoláshoz szükséges bájtok számában térnek el és nyilván ez határozza meg a tartományukat is. Néhány nyelv ismeri az *előjel nélküli egész* típust, ennek belső ábrázolása előjel nélküli (direkt).

Alapvetőek a *valós* típusok, belső ábrázolásuk általában lebegőpontos. A tartomány itt is az alkalmazott ábrázolás függvénye, ez viszont általában implementációfüggő.

Az egész és valós típusokra közös néven mint *numerikus* típusokra hivatkozunk. A numerikus típusok értékein a numerikus és hasonlító műveletek hajthatók végre.

A *karakteres* típus tartományának elemei karakterek, a *karakterlánc* vagy *sztring* típuséi pedig karaktersorozatok. Ábrázolásuk karakteres (karakterenként egy vagy két bájt, az alkalmazott kódtáblától függően), műveleteik a szöveges és hasonlító műveletek.

Egyes nyelvek ismerik a *logikai* típust. Ennek tartománya a hamis és igaz értékekből áll, műveletei a logikai és hasonlító műveletek, belső ábrázolása logikai.

Speciális egyszerű típus a *felsorolásos* típus. A felsorolásos típust saját típusként kell létrehozni. A típus definiálása úgy történik, hogy megadjuk a tartomány elemeit. Ezek *azonosítók* lehetnek. Az elemekre alkalmazhatók a hasonlító műveletek.

Egyes nyelvek értelmezik az egyszerű típusok egy speciális csoportját a *sorszámozott* típust. Ebbe a csoportba tartoznak általában az egész, karakteres, logikai és felsorolásos típusok. A sorszámozott típus tartományának elemei listát alkotnak, azaz van első és utolsó elem, minden elemnek van megelőzője (kivéve az elsőt) és minden elemnek van rákövetkezője (kivéve az utolsót). Tehát az elemek között egyértelmű sorrend értelmezett. A tartomány elemeihez kölcsönösen egyértelműen hozzá vannak rendelve a 0, 1, 2, ... sorszámok. Ez alól kivételt képeznek az egész típusok, ahol a tartomány minden eleméhez önmaga mint sorszám van hozzárendelve.

Egy sorszámozott típus esetén mindig értelmezhetők a következő műveletek:

– ha adott egy érték, meg kell tudni mondani a sorszámát és viszont

– bármely értékhez meg kell tudni mondani a megelőzőjét és a rákövetkezőjét

A sorszámozott típus az egész típus egyfajta általánosításának tekinthető.

Egy sorszámozott típus altípusaként lehet származtatni az *intervallum* típust.

2.4.2. Összetett típusok

Az eljárásorientált nyelvek két legfontosabb összetett típusa a *tömb* (melyet minden nyelv ismer) és a *rekord* (melyet csak a FORTRAN nem ismer).

A tömb típus a tömb absztrakt adatszerkezet (l. **Adatszerkezetek és algoritmusok**) megjelenése típus szinten. A tömb *statikus* és *homogén* összetett típus, vagyis tartományának

elemei olyan értékcsoportok, amelyekben az elemek száma ugyanannyi és az elemek azonos típusúak.

A tömböt, mint típust meghatározza:

- dimenzióinak száma
- indexkészletének típusa és tartománya
- elemeinek a típusa

Egyes nyelvek (pl. a C) nem ismerik a többdimenziós tömböket. Ezek a nyelvek a többdimenziós tömböket olyan egydimenziós tömbökként kezelik, amelyek elemei egydimenziós tömbök.

Többdimenziós tömbök reprezentációja lehet sor- vagy oszlopfolytonos. Ez általában implementációfüggő, a sorfolytonos a gyakoribb.

Ha van egy tömb típusú programozási eszközünk, akkor a nevével az összes elemre együtt, mint egy értékcsoportra tudunk hivatkozni (ez alól kivétel a mutatóorientáltsága miatt a C), az elemek sorrendjét a reprezentáció határozza meg. Az értékcsoport egyes elemeire a programozási eszköz neve után megadott indexek segítségével hivatkozunk. Az indexek a nyelvek egy részében szögletes, másik részében kerek zárójelek között állnak. Egyes nyelvek (pl. COBOL, PL/I) megengedik azt is, hogy a tömb egy adott dimenziójának összes elemét (pl. egy kétdimenziós tömb egy sorát) együtt hivatkozhassuk.

A nyelveknek a tömb típussal kapcsolatban a következő kérdéseket kell megválaszolniuk:

1. *Milyen típusúak lehetnek az elemek?*
   - Minden nyelv bármelyik skalár típust megengedi.
   - A modernebb nyelvek összetett típusokat is megengednek.
2. *Milyen típusú lehet az index?*
   - Minden nyelv megengedi, hogy egész típusú legyen.
   - A Pascalban és az Adában sorszámozott típusú is lehet.
3. *Amikor egy tömb típust definiálunk, hogyan kell megadni az indextartományt?*

- Lehet intervallum típusú értékkel (pl. Pascal, Ada), azaz meg kell adni az alsó és a felső határt.
- Más nyelveknél (pl. PL/I) az indextartomány alsó határa a nyelv által rögzített (általában 1), és csak a tartomány felső határát kell megadni.
- A nyelvek egy szűkebb csoportja szerint csak a felső határt kell megadni, de az alsót nem a nyelv rögzíti, hanem a programozó.
- Ritkán (pl. C) az adott dimenzióban lévő elemek darabszámát kell megadni, az indexek tartományát ez alapján a nyelv határozza meg.

4. *Hogyan lehet megadni az alsó és a felső határt illetve a darabszámot?*
   - Literállal vagy nevesített konstanssal (pl. FORTRAN, COBOL, Pascal), vagy konstans kifejezéssel (pl. C). Ezek a statikus tömbhatárokkal dolgozó nyelvek. Itt fordítási időben eldől az értékcsoport elemeinek darabszáma.
   - Kifejezéssel. (pl. PL/I, Ada). Ezek a dinamikus tömbhatárt alkalmazó nyelvek. Itt futási időben dől el a darabszám, de ezután természetesen az nem változik.

A tömb típus alapvető szerepet játszik az absztrakt adatszerkezetek folytonos ábrázolását megvalósító implementációknál.

A *rekord* típus a rekord absztrakt adatszerkezet (l. **Adatszerkezetek és algoritmusok**) megjelenése típus szinten. A rekord típus minden esetben heterogén, a tartományának elemei olyan értékcsoportok, amelyeknek elemei különböző típusúak lehetnek. Az értékcsoporton belül az egyes elemeket *mezőnek* nevezzük. Minden mezőnek saját, önálló neve (ami egy azonosító) és saját típusa van. A különböző rekord típusok mezőinek neve megegyezhet.

A nyelvek egy részében (pl. C) a rekord típus statikus, tehát a mezők száma minden értékcsoportban azonos. Más nyelvek esetén (pl. Ada) lehet egy olyan mezőegyüttes, amely minden értékcsoportban szerepel (a rekord fix része) és lehet egy olyan mezőegyüttes, amelynek mezői közül az értékcsoportokban csak bizonyosak szerepelnek (a rekord változó része). Egy külön nyelvi eszköz (a diszkriminátor) szolgál annak megadására, hogy az adott konkrét esetben a változó rész mezői közül melyik jelenjen meg.

Az ősnyelvek (pl. PL/I, COBOL) többszintű rekord típussal dogoznak. Ez azt jelenti, hogy egy mező feloszthatóújabb mezőkre, tetszőleges mélységig és típus csak a legalsó szintű mezőkhöz rendelhető, de az csak egyszerű típus lehet. A későbbi nyelvek (pl. Pascal, C, Ada) rekord típusa egyszintű, azaz nincsenek almezők, viszont a mezők típusa összetett is lehet.

Egy rekord típusú programozási eszköz esetén az eszköz nevével az értékcsoport összes mezőjére hivatkozunk egyszerre (a megadás sorrendjében).

Az egyes mezőkre külön *minősített névvel* tudunk hivatkozni, ennek alakja:

```
eszköznév.mezőnév
```

Az eszköz nevével történő *minősítésre* azért van szükség, mert a mezők nevei nem szükségszerűen egyediek.

A rekord típus alapvető szerepet játszik az input-outputnál.

2.4.3. Mutató típus

A mutató típus lényegében egyszerű típus, specialitását az adja, hogy tartományának elemei *tárcímek*. A mutató típus segítségével valósítható meg a programnyelvekben az indirekt címzés. A mutató típusú programozási eszköz értéke tehát egy tárbeli cím, így azt mondhatjuk, hogy az adott eszköz a tár adott területét címzi, az adott tárterületre „mutat”. A mutató típus egyik legfontosabb művelete a megcímzett tárterületen elhelyezkedő érték elérése.

A mutató típus tartományának van egy speciális eleme, amely nem valódi tárcím. Tehát az ezzel az értékkel rendelkező mutató típusú programozási eszköz „nem mutat sehova”. A nyelvek ezt az értéket általában beépített nevesített konstanssal kezelik (az Adában és a C-ben ennek neve NULL).

A mutató típus alapvető szerepet játszik az absztrakt adatszerkezetek szétszórt reprezentációját kezelő implementációknál.

2.5. A nevesített konstans

A nevesített konstans olyan programozási eszköz, amelynek 3 komponense van:

– név

– típus

– érték

A nevesített konstanst deklarálni kell.

A program szövegében a nevesített konstans a nevével jelenik meg, és az mindig az értékkomponenst jelenti. A nevesített konstans értékkomponense a deklarációnál eldől és nem változtatható meg a futás folyamán.

A nevesített konstans szerepe egyrészt az, hogy bizonyos sokszor előforduló értékeket „beszélő” nevekkel látunk el és ily módon az érték szerepkörére tudunk utalni a szövegben. Másrészt viszont, ha a program szövegében meg akarjuk változtatni ezt az értéket, akkor nem kell annak valamennyi előfordulását megkeresni és átírni, hanem elegendő egy helyen, a deklarációs utasításban végrehajtani a módosítást.

A nevesített konstanssal kapcsolatban az egyes nyelveknek a következő kérdéseket kell megválaszolniuk:

1. *Létezik-e a nyelvben beépített nevesített konstans?*
2. *A programozó definiálhat-e saját nevesített konstanst?*
3. *Ha igen, milyen típusút?*
4. *Hogyan adható meg a nevesített konstans értéke?*

A válaszok:

FORTRAN-ban és PL/I-ben nincs nevesített konstans, COBOL-ban pedig csak beépített van.

A C-ben van beépített nevesített konstans és a programozó többféleképpen tud létrehozni sajátot. A legegyszerűbb az előfordítónak szóló

```
#define név literál
```

makró használata. Ekkor az előfordító a forrásprogramban a `név` minden előfordulását helyettesíti a `literál`lal.

Pascalban és Adában van beépített nevesített konstans és a programozó is definiálhat saját nevesített konstanst, egyszerű és összetett típusút egyaránt. A Pascalban az értéket literállal, Adában kifejezés segítségével tudjuk megadni.

2.6. A változó

A változó olyan programozási eszköz, amelynek 4 komponense van:

– név
– attribútumok
– cím
– érték

A név egy azonosító. A program szövegében a változó mindig a nevével jelenik meg, az viszont bármely komponenst jelentheti. Szemlélhetjük úgy a dolgokat, hogy a másik három komponenst a névhez rendeljük hozzá.

Az attribútumok olyan jellemzők, amelyek a változó futás közbeni viselkedését határozzák meg. Az eljárásorientált nyelvekben (általában a típusos nyelvekben) a legfőbb attribútum a típus, amely a változó által felvehető értékek körét határolja be. Változóhoz attribútumok deklaráció segítségével rendelődnek. A deklarációnak különböző fajtáit ismerjük.

Explicit deklaráció: A programozó végzi explicit deklarációs utasítás segítségével. A változó teljes nevéhez kell az attribútumokat megadni. A nyelvek általában megengedik, hogy több változónévhez ugyanazokat az attribútumokat rendeljük hozzá.

Implicit deklaráció: A programozó végzi, betűkhöz rendel attribútumokat egy külön deklarációs utasításban. Ha egy változó neve nem szerepel explicit deklarációs utasításban, akkor a változó a nevének kezdőbetűjéhez rendelt attribútumokkal fog rendelkezni, tehát az azonos kezdőbetűjű változók ugyanolyan attribútumúak lesznek.

Automatikus deklaráció: A fordítóprogram rendel attribútumot azokhoz a változókhoz, amelyek nincsenek explicit módon deklarálva, és kezdőbetűjükhöz nincs attribútum rendelve egy implicit deklarációs utasításban. Az attribútum hozzárendelése a név valamelyik karaktere (gyakran az első) alapján történik.

Az eljárásorientált nyelvek mindegyike ismeri az explicit deklarációt, és egyesek csak azt ismerik. Az utóbbiak általánosságban azt mondják, hogy minden névvel rendelkező programozói eszközt explicit módon deklarálni kell.

A változó címkomponense a tárnak azt a részét határozza meg, ahol a változó értéke elhelyezkedik. A futási idő azon részét, amikor egy változó rendelkezik címkomponenssel, a változó *élettartamának* hívjuk.

Egy változóhoz cím rendelhető az alábbi módokon.

Statikus tárkiosztás: A futás előtt eldől a változó címe és a futás alatt az nem változik. Amikor a program betöltődik a tárba, a statikus tárkiosztású változók fix tárhelyre kerülnek.

Dinamikus tárkiosztás: A cím hozzárendelését a futtató rendszer végzi. A változó akkor kap címkomponenst, amikor aktivizálódik az a programegység, amelynek ő lokális változója, és a címkomponens megszűnik, ha az adott programegység befejezi a működését. A címkomponens a futás során változhat, sőt vannak olyan időintervallumok, amikor a változónak nincs is címkomponense.

A programozó által vezérelt tárkiosztás: A változóhoz a programozó rendel címkomponenst futási időben. A címkomponens változhat, és az is elképzelhető, hogy bizonyos időintervallumokban nincs is címkomponens. Három alapesete van:

– A programozó abszolút címet rendel a változóhoz, konkrétan megadja, hogy hol helyezkedjen el.
– Egy már korábban a tárban elhelyezett programozási eszköz címéhez képest mondja meg, hogy hol legyen a változó elhelyezve, vagyis relatív címet ad meg. Lehet, hogy a programozó az abszolút címet nem is ismeri.

– A programozó csak azt adja meg, hogy mely időpillanattól kezdve legyen az adott változónak címkomponense, az elhelyezést a futtató rendszer végzi. A programozó nem ismeri az abszolút címet.

Mindhárom esetben lennie kell olyan eszköznek, amivel a programozó megszüntetheti a címkomponenst.

A programozási nyelvek általában többféle címhozzárendelést ismernek, az eljárásorientált nyelveknél általános a dinamikus tárkiosztás. A változók címkomponensével kapcsolatos a *többszörös tárhivatkozás* esete. Erről akkor beszélünk, amikor két különböző névvel, esetleg különböző attribútumokkal rendelkező változónak a futási idő egy adott pillanatában azonos a címkomponense és így értelemszerűen az értékkomponense is. Így ha az egyik változó értékét módosítjuk, akkor a másiké is megváltozik. A korai nyelvekben (pl. FORTRAN, PL/I) erre explicit nyelvi eszközök álltak rendelkezésre, mert bizonyos problémák megoldása csak így volt lehetséges. A szituáció viszont előidézhető (akár véletlenül is) más nyelvekben is, és ez nem biztonságos kódhoz vezethet.

A változó értékkomponense mindig a címen elhelyezett bitkombinációként jelenik meg. A bitkombináció felépítését a típus által meghatározott reprezentáció dönti el.

Egy változó értékkomponensének meghatározására a következő lehetőségek állnak rendelkezésünkre:

Értékadó utasítás: Az eljárásorientált nyelvek leggyakoribb utasítása, az algoritmusok kódolásánál alapvető. Alakja az egyes nyelvekben:

FORTRAN:

```
változónév = kifejezés
```

COBOL:

```
MOVE érték TO változónév [, változónév ]...
```

PL/I:

```
változónév [, változónév ]... = kifejezés;
```

Pascal:

```
változónév := kifejezés
```

C:

```
változónév = kifejezés;
```

Ada:

```
változónév := kifejezés;
```

Az értékadó utasítás bal oldalán a változó neve általában a címkomponenst, kifejezésben az értékkomponenst jelenti. Az étrtékadó utasítás esetén a műveletek elvégzésének sorrendje implementációfüggő. Általában a baloldali változó címkomponense dől el először.

A típusegyenértékűséget (l. 3. fejezet) valló nyelvekben a kifejezés típusának azonosnak kell lennie a változó típusával, a típuskényszerítést valló nyelveknél pedig mindig a kifejezés típusa konvertálódik a változó típusára.

Input: A változó értékkomponensét egy perifériáról kapott adat határozza meg. Részletesen l. 13. fejezet.

Kezdőértékadás: Két fajtája van. *Explicit kezdőértékadásnál* a programozó explicit deklarációs utasításban a változó értékkomponensét is megadja. Amikor a változó címkomponenst kap, akkor egyben az értéket reprezentáló bitsorozat is beállítódik Megadható az értékkomponens literál vagy kifejezés segítségével.

A hivatkozási nyelvek egy része azt mondja, hogy mindaddig, amíg a programozó valamilyen módon nem határozza meg egy változó értékkomponensét, az *határozatlan*, tehát nem használható föl. Ez azért van, mert amikor egy változó címkomponenst kap, akkor az adott memóriaterületen tetszőleges bitkombináció („szemét") állhat, amivel nem lehet mit kezdeni. Van olyan hivatkozási nyelv, amely az *automatikus kezdőértékadás* elvét vallja. Ezeknél a nyelveknél, ha a programozó nem adott explicit kezdőértéket, akkor a címkomponens hozzárendelésekor a hivatkozási nyelv által meghatározott bitkombináció kerül beállításra („nullázódnak a változók"). A hivatkozási nyelvek harmadik része nem mond semmit erről a dologról. Viszont az implementációk túlnyomó része megvalósítja az automatikus kezdőértékadást, akár még a hivatkozási nyelv ellenében is.

2.7. Alapelemek az egyes nyelvekben

Turbo Pascal:

A Pascalban minden programozói eszközt explicit módon deklarálni kell.

A Turbo Pascal típusrendszere az alábbi:

egyszerű típusok
  sorszámozott
    előredefiniált
      karakteres (`char`)
      logikai (`boolean`)
      egész (`integer`, `shortint`, `longint`, `byte`, `word`)
    felsorolásos
    intervallum
  valós

`string` (egyszerűnek és összetettnek is tekinthető egyszerre, mint karakterlánc, illetve mint karakterek egydimenziós tömbje)

összetett típusok
  tömb (`array`)
  rekord (`record`)
  halmaz (`set`)
  állomány (`file`)
  objektum (`object`)
mutató (`pointer`)

A tömb típus megadásának formája:

`ARRAY [ intervallum [, intervallum ]... ] OF elemtípus`

Nevesített konstans deklarációja:

`CONST név=literál; [ név=literál; ]...`

Változó deklarációja:

```
VAR név [, név ]... : típus; [ név [, név ]...: típus; ]...
```

Saját típus létrehozása:

```
TYPE név=típusleírás; [ név =típusleírás; ]...
```

Az így létrehozott típus minden más típustól különbözni fog.

Ada:

A programozónak minden saját eszközt explicit módon deklarálni kell.

Az Ada típusrendszere az alábbi:

skalár típusok

- felsorolásos
- egész (`integer`)
- karakteres (`character`)
- logikai (`boolean`)
- valós (`float`)

összetett típusok

- tömb (`array`)
- rekord (`record`)

mutató típus(`access`)

privát típus (`private`)

A skalár típus sorszámozott típus. Az intervallum altípust az Ada a következőképpen képzi:

```
RANGE alsó_határ..felső_határ
```

Az explicit deklarációs utasítás a következőképpen néz ki:

```
név [, név ]... : [CONSTANT] típus [:=kifejezés];
```

Ha a CONSTANT alapszó szerepel, akkor nevesített konstanst, ha nem szerepel, akkor változót deklaráltunk. A kifejezés a nevesített konstansnál kötelező, ez definiálja az értékét. Változó esetén pedig ezzel lehet explicit kezdőértéket adni.

Például:

```
A: constant integer:=111;
B: constant integer:=A*22+56;
X: real;
Y: real:=1.0;
```

Saját, minden más típustól különböző típus deklarálása:

```
TYPE név IS típusleírás;
```

Altípus deklarálása:

```
SUBTYPE név IS típusleírás;
```

Például:

```
type BYTE is range 0..255;
subtype KISBETUK is character range 'a'..'z';
```

Felsorolásos típus létrehozása saját típusként:

```
type HONAPOK is (Januar, Februar, Marcius, Aprilis, Majus, Junius,
Julius, Augusztus, Szeptember, Oktober, November, December);

type NYARI_HONAPOK is (Junius, Julius, Augusztus);
```

Ebben az esetben ugyanaz az azonosító két felsorolásos típus tartományában is szerepel, tehát hivatkozáskor meg kell mondani, hogy melyik típus eleméről van szó. Az Ada ilyenkor minősít a típus nevével:

```
HONAPOK'ORD(Augusztus)
```

Az Adában dinamikusak a tömbhatárok. Definiálható olyan saját tömb típus, ahol nem adjuk meg előre az indexek tartományát, hanem majd csak akkor, amikor a deklarációban felhasználjuk a típust.

Például:

```
type T is array(integer<>,integer<>);
A:T(0..10,0..10);
```

C:

A C típusrendszere a következő:

aritmetikai típusok

- integrális típusok
  - egész (`int`, `short`[`int`], `long`[`int`])
  - karakter (`char`)
  - felsorolásos
- valós (`float`, `double`, `long double`)

származtatott típusok

- tömb
- függvény
- mutató
- struktúra
- union

`void` típus

Az aritmetikai típusok az egyszerű, a származtatottak az összetett típusok a C-ben. Az aritmetikai típusok tartományának elemeivel aritmetikai műveletek végezhetők. A karakter típus tartományának elemeit a belső kódok alkotják. Logikai típus nincs, hamisnak az `int 0` felel meg, minden más `int` értéket pedig igaznak tekint a C. A hasonlító műveletek igaz esetén `int 1` értéket adnak Az egészek és karakter típus előtt szerepeltethető `unsigned`

típusminősítő nem előjeles (direkt) ábrázolást, a `signed` előjeles ábrázolást jelöl. A struktúra egy fix szerkezetű rekord, a union egy olyan, csak változó részt tartalmazó rekord, amelynél minden konkrét esetben pontosan egyetlen mező van jelen. A `void` típus tartománya üres, így reprezentációja és műveletei sincsenek.

A felsorolásos típusok tartományai nem fedhetik át egymást. A tartomány elemei viszont `int` típusú nevesített konstansoknak tekinthetők. Az értékük egész literálokkal beállítható, de ha nincs explicit értékadás, akkor a felsorolás sorrendjében 0-ról indul az értékük és egyesével növekszik. Ha valamelyik elem értékét megadtuk és a következőét nem, akkor annak értéke az előző értékétől 1-el nagyobb érték lesz. Különböző elemekhez ugyanaz az érték hozzárendelhető. A felsorolásos típus megadásának formája:

```
ENUM [ név ] {azonosító [=konstans_kifejezés ]
  [, azonosító [=konstans_kifejezés ] ]... }[változólista];
```

Példa:

```
enum szinek {VOROS=11, NARANCS=9, SARGA=7, ZOLD=5, KEK=3, IBOLYA=3};
```

Az explicit deklarációs utasítás a következőképpen néz ki:

```
[ CONST ] típusleírás eszközazonosítás [ =kifejezés ]
            [, eszközazonosítás [ =kifejezés ]... ;
```

A `CONST` megadása esetén nevesített konstanst deklarálunk (ekkor a `kifejezés` annak értékét, a típusleírás annak típusát definiálja, és az `eszközazonosítás` azonosító lehet), egyébként viszont a `típusleírás` és az `eszközazonosítás` alapján meghatározott programozási eszközt. Ha ez változó, akkor a `kifejezés` segítségével explicit kezdőérték adható neki. Ha nem szerepel a `CONST`, akkor az `eszközazonosítás` helyén az alábbiak szerepelhetnek a megadott jelentéssel:

`azonosító : típusleírás` típusú változó

`(azonosító): típusleírás` visszatérési típussal rendelkező függvényt címző mutató típusú változó

`*azonosító : típusleírás` típusú eszközt címző mutató típusú változó

`azonosító(): típusleírás` visszatérési típusú függvény

`azonosító[]: típusleírás` típusú elemeket tartalmazó tömb típusú változó

és ezek tetszőleges kombinációja. A `típusleírás` ugyanezeket a konstrukciókat tartalmazhatja a típusnevek mellett.

Példa:

```
int i, *j, f(), *g(), a[17], *b[8];
```

Ebben a deklarációs utasításban az `i` egész típusú a `j` egészet címző mutató típusú, az `a` egészeket tartalmazó 17 elemű egydimenziós tömb típusú, a `b` egészeket címző mutató típusú elemeket tartalmazó 8 elemű egydimenziós tömb típusú változó; az `f` egész visszatérési típusú, a `g` pedig egészeket címző mutató visszatérési típusú függvény.

Típusleírás nevesítése:

```
TYPEDEF típusleírás név [,típusleírás név]... ;
```

Nem hoz létre új típust, csak típusnév szinonimát.

Struktúra deklarálása:

```
STRUCT [struktúratípus_név] {mező_deklarációk} [változólista]
```

Union deklarálása:

```
UNION [uniontípus_név] {mező_deklarációk} [változólista]
```

A C csak egydimenziós tömböket kezel. Az indexek darabszámát kell megadni és az index 0-tól `darabszám-1`-ig fut. A hivatkozási nyelv statikus tömbhatárokat ismer, de egyes implementációk dinamikusakat kezelnek.

A C a tömböt mindig mutató típusként kezeli. Egy tömb típusú változó neve egy olyan mutató típusú lesz, amely a tömb első elemét címzi.

A C-ben van automatikus deklaráció, ha egy függvénynévhez nem adunk típust, akkor az alapértelmezés szerint `int` lesz.

3. KIFEJEZÉSEK

A kifejezések szintaktikai eszközök. Arra valók, hogy a program egy adott pontján ott már ismert értékekből új értéket határozzunk meg. Két komponensük van, *érték* és *típus*.

Egy kifejezés formálisan a következő összetevőkből áll:

- *operandusok*: Az operandus literál, nevesített konstans, változó vagy függvényhívás lehet. Az értéket képviseli.
- *operátorok*: Műveleti jelek. Az értékekkel végrehajtandó műveleteket határozzák meg.
- *kerek zárójelek*: A műveletek végrehajtási sorrendjét befolyásolják. Minden nyelv megengedi a redundáns zárójelek alkalmazását.

A legegyszerűbb kifejezés egyetlen operandusból áll.

Attól függően, hogy egy operátor hány operandussal végzi a műveletet, beszélünk *egyoperandusú* (unáris), *kétoperandusú* (bináris), vagy *háromoperandusú* (ternáris) operátorokról.

A kifejezésnek három alakja lehet attól függően, hogy kétoperandusú operátorok esetén az operandusok és az operátor sorrendje milyen. A lehetséges esetek:

- *prefix*, az operátor az operandusok előtt áll (`* 3 5`)
- *infix*, az operátor az operandusok között áll (`3 * 5`)
- *postfix*, az operátor az operandusok mögött áll (`3 5 *`)

Az egyoperandusú operátorok általában az operandus előtt, ritkán mögötte állnak. A háromoperandusú operátorok általában infixek.

Azt a folyamatot, amikor a kifejezés értéke és típusa meghatározódik, a *kifejezés kiértékelésének* nevezzük. A kiértékelés során adott sorrendben elvégezzük a műveleteket, előáll az érték, és hozzárendelődik a típus.

A műveletek végrehajtási sorrendje a következő lehet:

- A műveletek felírási sorrendje, azaz balról-jobbra.
- A felírási sorrenddel ellentétesen, azaz jobbról-balra.

– Balról-jobbra a precedencia táblázat figyelembevételével.

Az infix alak nem egyértelmű (l. **Adatszerkezetek és algoritmusok**). Az ilyen alakot használó nyelvekben az operátorok nem azonos erősségűek. Az ilyen nyelvek operátoraikat egy *precedencia táblázatban* adják meg. A precedencia táblázat sorokból áll, az egy sorban megadott operátorok azonos erősségűek (prioritásúak, precedenciájúak), az előrébb szereplők erősebbek. Minden sorban meg van adva még a *kötési irány*, amely megmondja, hogy az adott sorban szereplő operátorokat milyen sorrendben kell kiértékelni, ha azok egymás mellett állnak egy kifejezésben. A kötési irány lehet *balról jobbra*, vagy *jobbról balra*.

Egy infix alakú kifejezés kiértékelése a következőképpen történik:

Indulunk a kifejezés elején (balról-jobbra szabály) és összehasonlítjuk az 1. és 2. operátor precedenciáját (ha csak egyetlen operátor van, akkor az általa kijelölt műveletet végezzük el, ha csak egyetlen operandus van, akkor annak értéke adja a kifejezés értékét, típusa a típusát). Ha a baloldali erősebb, vagy azonos erősségűek és a precedencia táblázat adott sorában balról jobbra kötési irány van, akkor végrehajtódik a baloldali operátor által kijelölt művelet, különben továbblépünk a kifejezésben (ha még van operátor) és hasonlítjuk a következő két operátor precedenciáját. Ez alapján az elsőnek elvégzendő művelet egyértelműen meghatározható, a folytatás viszont implementációfüggő. Ugyanis az első művelet végrehajtása után vagy visszalépünk a kifejezés elejére és megint az első operátorral kezdjük a további vizsgálatokat, vagy a kifejezésben továbblépünk és csak akkor lépünk vissza az elejére, amikor a kifejezés végére értünk.

> ***Megjegyzés**: A kifejezés értékének meghatározása természetesen futási idejű tevékenység. A fordítóprogramok általában az infix kifejezésekből postfix kifejezéseket állítanak elő, és ezek tényleges kiértékelése történik meg. A fentebb leírt lépések tehát igazában az infix kifejezések átírására vonatkoznak.*

A műveletek elvégzése előtt meg kell határozni az operandusok értékét. Ennek sorrendjét a hivatkozási nyelvek egy része szabályozza (általában előbb a baloldaliét), más része nem mond róla semmit, a C pedig azt mondja, hogy tetszőleges (tehát implementációfüggő).

Az infix kifejezések esetén kell használni a zárójeleket, amelyek a precedencia táblázat alapján következő végrehajtási sorrendet tudják felülbírálni. Egy bezárójelezett részkifejezést

mindig előbb kell kiértékelni. Egyes nyelvek a kerek zárójeleket is szerepeltetik a precedencia táblázatban, az első sorban.

A teljesen zárójelezett infix kifejezés egyértelmű, kiértékelésénél egyetlen sorrend létezik.

Az eljárásorientált nyelvek az infix alakot használják.

A kiértékelés szempontjából speciálisak azok a kifejezések, amelyekben logikai operátorok szerepelnek. Ezeknél ugyanis úgyis eldőlhet a kifejezés értéke, hogy nem végezzük el az összes kijelölt műveletet. Például egy ÉS művelet esetén, ha az első operandus értéke hamis, az eredmény hamis lesz, függetlenül a második operandus értékétől (bármilyen bonyolult részkifejezéssel is adtuk meg azt).

A nyelvek az ilyen kifejezésekkel kapcsolatban a következőket vallják:

– Az ilyen kifejezést is végig kell értékelni (pl. FORTRAN), ez a *teljes* kiértékelés.
– Csak addig kell kiértékelni a kifejezést, amíg egyértelműen el nem dől az eredmény. Ez a *rövidzár* kiértékelés (pl. PL/I).
– A nyelvben vannak rövidzár illetve nem rövidzár operátorok, a programozó döntheti el a kiértékelés módját (pl. Adában nem rövidzár : `and`, `or`; rövidzár: `and then`, `or else`).
– A kifejezés kiértékelését futási üzemmódként lehet beállítani (pl. Turbo Pascal).

A kifejezés típusának meghatározásánál kétféle elvet követnek a nyelvek. Vannak a *típusegyenértékűséget* és vannak a *típuskényszerítést* vallók. Ugyanez a kérdéskör fölmerül az értékadó utasításnál (l. 2.6. alfejezet) és a paraméterkiértékelésnél (l. 5.4. alfejezet) is.

A típusegyenértékűséget valló nyelvek azt mondják, hogy egy kifejezésben egy kétoperandusú vagy háromoperandusú operátornak csak *azonos* típusú operandusai lehetnek. Ilyenkor *nincs konverzió*, az eredmény típusa vagy az operandusok közös típusa, vagy azt az operátor dönti el (például hasonlító műveletek esetén az eredmény logikai típusú lesz).
A különböző nyelvek szerint két programozási eszköz típusa azonos, ha azoknál fönnáll a

– *deklaráció egyenértékűség*: az adott eszközöket azonos deklarációs utasításban, együtt, azonos típusnévvel deklaráltuk.

- *név egyenértékűség*: az adott eszközöket azonos típusnévvel deklaráltuk (esetleg különböző deklarációs utasításban).
- *struktúra egyenértékűség*: a két eszköz összetett típusú és a két típus szerkezete megegyezik (például két 10-10 egész értéket tartalmazó tömb típus, függetlenül az indexek tartományától).

A típuskényszerítés elvét valló nyelvek esetén különböző típusú operandusai lehetnek az operátornak. A műveletek viszont csak az azonos belső ábrázolású operandusok között végezhetők el, tehát különböző típusú operandusok esetén konverzió van. Ilyen esetben a nyelv definiálja, hogy egy adott operátor esetén egyrészt milyen típuskombinációk megengedettek, másrészt, hogy mi lesz a művelet eredményének a típusa.

A kifejezés kiértékelésénél minden művelet elvégzése után eldől az adott részkifejezés típusa és az utoljára végrehajtott műveletnél pedig a kifejezés típusa.

Egyes nyelvek (pl. Pascal, C) a numerikus típusoknál megengedik a típuskényszerítés egy speciális fajtáját még akkor is, ha egyébként a típusegyenértékűséget vallják. Ezeknél a nyelveknél beszélünk a *bővítés* és *szűkítés* esetéről. A bővítés olyan típuskényszerítés, amikor a konvertálandó típus tartományának minden eleme egyben eleme a céltípus tartományának is (pl. egész → valós). Ekkor a konverzió minden további nélkül, értékvesztés nélkül végrehajtható. A szűkítés ennek a fordítottja (pl. valós → egész), ekkor a konverziónál értékcsonkítás, esetleg kerekítés történik.
A nyelvek közül az Adában semmiféle típuskeveredés nem lehet, a PL/I viszont a teljes konverzió híve.

Konstans kifejezés

Azt a kifejezést, amelynek értéke fordítási időben eldől, amelynek kiértékelését a fordító végzi, *konstans kifejezésnek* hívjuk. Operandusai általában literálok és nevesített konstansok lehetnek.

3.1. Kifejezés a C-ben

A C egy alapvetően kifejezésorientált nyelv. Az aritmetikai típusoknál a típuskényszerítés elvét vallja.

A mutató típus tartományának elemeivel bizonyos aritmetikai műveletek elvégezhetők, azok előjel nélküli egészeknek tekinthetők.

A tömb típusú változó neve mutató típusú, tehát

```
int i;
int a[10];
```

esetén `a[i]` jelentése `*(a+i)`.

A C kifejezésfogalmának rekurzív definíciója a következő:

```
kifejezés:
{ elsődleges_kifejezés|
  balérték++ |
  balérték-- |
  ++balérték |
  --balérték |
  egy_operandusú_operátor kifejezés |
  SIZEOF(kifejezés) |
  SIZEOF(típusnév) |
  (típusnév)kifejezés |
  kifejezés két_operandusú_operátor kifejezés |
  kifejezés?kifejezés:kifejezés |
  balérték értékadó_operátor kifejezés |
  kifejezés,kifejezés }


elsődleges_kifejezés:

{ konstans |
  változó |
```

```
  (kifejezés) |
  függvénynév(aktuális_paraméter_lista) |
  tömbnév[kifejezés] |
  balérték.azonosító |
  elsődleges_kifejezés->azonosító}


balérték :
{ azonosító |
  tömbnév[kifejezés] |
  balérték.azonosító |
  elsődleges_kifejezés->azonosító |
  *kifejezés |
  (balérték)}
```

A C precedencia táblázata a következő:

```
( )   []   . ->                                  →
* & + - ! ~ ++ -- SIZEOF (típus)                 ←
* / %                                            →
+ -                                              →
>>   <<                                          →
<   >   <=   >=                                  →
== !=                                            →
&                                                →
^                                                →
|                                                →
&&                                               →
||                                               →
?:                                               →
= += -= *= /= %= >>= <<= &= ^= |=                ←
,                                                →
```

Az utolsó oszlop a kötési irányt mutatja.

A C kifejezés formális leírásában a következő operátorokra hivatkoztunk:

- `egy_operandusú_operátor`: a precedencia táblázat 2. sorának első 6 operátora
- `két_operandusú_operátor`: a precedencia táblázat 3. - 12. sorának operátorai
- `értékadó _operátor`: a 14. sor operátorai

Az egyes operátorok értelmezése:

`()`

Egyrészt a szokásos értelmű kiemelést, a precedencia felülírását szolgálja, másrészt a függvényoperátor.

`[]`

A tömboperátor.

`.`

A minősítő operátor, struktúra és union esetén használjuk, ha névvel minősítünk.

`->`

A mutatóval történő minősítés operátora.

`*`

Az indirekciós operátor. A mutató típusú operandusa által hivatkozott tárterületen elhelyezett értékéket adja.

`&`

Az operandusának címét adja meg.

`+`

Plusz előjel.

`-`

Mínusz előjel.

`!`

Egyoperandusú operátor, integrális és mutató típusú operandusokra alkalmazható. Ha az operandus értéke nem nulla, akkor az eredmény nulla, egyébként 1. Az eredmény `int` típusú lesz.

`~`

Az egyes komplemens operátora.

`++` és `--`

Operandusának értékét növeli illetve csökkenti 1-el.

```
int x,n;
n=5;
x=n++;
```

esetén `x` értéke `5` lesz, az értékadás az `n` korábbi értékével történik.

```
x=++n;
```

esetén pedig `6` lesz, a megnövelt értékkel történik az értékadás.

Az `n` értéke mindkét esetben növelődik.

`sizeof(kifejezés)`

A kifejezés típusának az ábrázolási hosszát adja bájtban.

`sizeof(típus)`

A `típus` ábrázoláshoz szükséges bájtok számát adja.

`(típus)`

Az explicit konverziós operátor.

`*`

Szorzás operátora.

`/`

Osztás operátora. Az egészek osztásakor egészosztás.

`%`

`M`aradékképzés operátora.

`+`

Összeadás operátora.

`-`

Kivonás operátora.

`>>` és `<<`

Léptető operátorok. A baloldali operandust lépteti a jobboldali által meghatározott számú bittel jobbra illetve balra.. Ha balra léptet, akkor 0-kat hoz be jobbról. Ha jobbra léptet, akkor az előjelbitet lépteti végig bal oldalon. Integrális típusú operandusokon működik.

`<`, `>`, `<=`, `>=`, `==`, `!=`

Hasonlító operátorok, eredményük `int` típusú, igaz esetben `1`, hamisnál `0`.

`&`, `^`, `|`

Nem rövidzár logikai operátorok (és, kizáró vagy, vagy), integrális típuson működnek, a megfelelő műveletet bitenként hajtják végre.

`&&` és `||`

Rövidzár logikai operátorok (és, vagy), `int` `0` és `1` értékeken működnek.

`? :`

Háromoperandusú operátor, az ún. feltételes operátor. Ha az első operandus értéke nem `0`, akkor a művelet eredményét a második operandus értéke határozza meg, egyébként a harmadiké.

Például az `(a>b)?a:b` kifejezés az `a` és `b` értéke közül a nagyobbikat adja.

`=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=`

Értékadó operátorok. Az `x operátor= y` kifejezés megfelel a következő értékadásnak: `x=(x)operátor(y)`. Az első operandus értékét írjuk felül.

`,`

A balról jobbra kiértékelést kényszeríti ki.

4. UTASÍTÁSOK

Az utasítások alkotják az eljárásorientált nyelveken megírt programok olyan egységeit, amelyekkel egyrészt az algoritmusok egyes lépéseit megadjuk, másrészt a fordítóprogram ezek segítségével generálja a tárgyprogramot. Két nagy csoportjuk van: a *deklarációs* és a *végrehajtható* utasítások.

A deklarációs utasítások mögött nem áll tárgykód. Ezen utasítások teljes mértékben a fordítóprogramnak szólnak, attól kérnek valamilyen szolgáltatást, üzemmódot állítanak be, illetve olyan információkat szolgáltatnak, melyeket a fordítóprogram felhasznál a tárgykód generálásánál. Alapvetően befolyásolják a tárgykódot, de maguk nem kerülnek lefordításra. A programozó a névvel rendelkező saját programozási eszközeit tudja deklarálni.

A végrehajtható utasításokból generálja a fordítóprogram a tárgykódot. Általában a magasszintű nyelvek végrehajtható utasításaiból több (néha sok) gépi kódú utasítás áll elő.

A végrehajtható utasításokat az alábbiak szerint csoportosíthatjuk:

1. Értékadó utasítás
2. Üres utasítás
3. Ugró utasítás
4. Elágaztató utasítások
5. Ciklusszervező utasítások
6. Hívó utasítás
7. Vezérlésátadó utasítások
8. I/O utasítások
9. Egyéb utasítások

A 3-7. utasítások az ún. *vezérlési szerkezetet megvalósító* utasítások. Az eljárásorientált nyelvek általában tartalmazzák az 1-5. utasításokat, egy részük pedig ismeri a 6-8. utasítás fajtákat. A legnagyobb eltérés az egyéb utasítások terén tapasztalható. A nyelvek egy részében nincs ilyen utasítás (pl. C), némelyikben viszont sok van belőlük (pl. PL/I).

Vegyük sorra a végrehajtható utasításokat.

4.1. Értékadó utasítás

Feladata beállítani vagy módosítani egy (esetleg több) változó értékkomponensét a program futásának bármely pillanatában. Ezt az utasítást már tárgyaltuk a változónál (l. 2.6. alfejezet).

4.2. Üres utasítás

Az eljárásorientált programnyelvek általában tartalmaznak üres utasítást, és vannak olyan nyelvek (a korai nyelvek), melyekben a szintaktika olyan, hogy elengedhetetlen ennek használata. Jelentősége viszont általánosságban abban áll, hogy segítségével egyértelmű programszerkezet alakítható ki.

Az üres utasítás hatására a processzor egy üres gépi utasítást hajt végre.

Egyes nyelvekben az üres utasításnak van külön alapszava (pl. Adában a `NULL`), máshol nincsen. Ez utóbbi nyelvekben az üres utasítást nem jelöljük (pl. két utasítás végjel között nem áll semmi).

4.3. Ugró utasítás

Az ugró (vagy feltétel nélküli vezérlésátadó) utasítás segítségével a program egy adott pontjáról egy adott címkével ellátott végrehajtható utasításra adhatjuk át a vezérlést. Általánosan használt alakja:

```
GOTO címke
```

A korai nyelvekben (FORTRAN, PL/I) `GOTO` nélkül gyakorlatilag nem lehet programot írni. A későbbi nyelvekben viszont minden program vezérlési szerkezete felírható a `GOTO` utasítás

használata nélkül, bár a nyelvek tartalmazzák azt. Az ugró utasítás nem fegyelmezett módon történő használata veszélyeket hordoz magában, mert nem biztonságos, átláthatatlan, struktúra nélküli kódot eredményezhet.

4.4. Elágaztató utasítások

4.4.1. Kétirányú elágaztató utasítás (feltételes utasítás)

A kétirányú elágaztató utasítás arra szolgál, hogy a program egy adott pontján két tevékenység közül válasszunk, illetve egy adott tevékenységet végrehajtsunk vagy sem. A nyelvekben meglehetősen általános a feltételes utasítás következő szerkezete:

```
IF feltétel THEN tevékenység [ELSE tevékenység ]
```

A `feltétel` egy logikai (vagy annak megfelelő típusú) kifejezés.

Kérdés, hogy egy nyelv mit mond a `tevékenység` megadásáról. Egyes nyelvek (pl. Pascal) szerint itt csak egyetlen végrehajtható utasítás állhat. Ha viszont a tevékenység olyan összetett, hogy csak több utasítással tudjuk leírni, akkor *utasítás zárójeleket* kell alkalmazni. Az utasítás zárójel a Pascal esetén `BEGIN` és `END`. Egy ilyen módon bezárójelezett utasítássorozatot hívunk *utasítás csoportnak*. Az utasítás csoport formálisan egyetlen utasításnak tekintendő. Más nyelvek (speciális szintaktikájuk miatt) megengedik, hogy a tevékenység leírására tetszőleges végrehajtható utasítássorozatot alkalmazzunk (pl. Ada). A kétirányú elágaztató utasításnál beszélünk rövid (nem szerepel az ELSE-ág) és hosszú (szerepel az ELSE-ág) alakról.

A kétirányú elágaztató utasítás szemantikája a következő:

Kiértékelődik a feltétel. Ha az igaz, akkor végrehajtódik a `THEN` utáni tevékenység, és a program az IF-utasítást követő utasításon folytatódik. Ha a feltétel értéke hamis, és van ELSE−ág, akkor az ott megadott tevékenység hajtódik végre, majd a program az IF-utasítást követő utasításon folytatódik, végül ha nincs ELSE-ág, akkor ez egy üres utasítás.

Az IF-utasítások tetszőlegesen egymásba ágyazhatók és természetesen akár az IF-ágban, akár az ELSE-ágban újabb feltételes utasítás állhat. Ilyenkor felmerülhet a „csellengő ELSE” problémája, amikoris a kérdés a következő: egy

```
IF ... THEN IF ... THEN ... ELSE ...
```

konstrukciójú feltételes utasítás esetén melyik IF-hez tartozik az ELSE-ág? Vagyis itt egy olyan rövid IF-utasítás áll, amelybe be van ágyazva egy hosszú, vagy pedig egy hosszú IF-utasítás THEN-ágában szerepel egy rövid?

A válasz többféle lehet:

a. A „csellengő ELSE” probléma kiküszöbölhető, ha mindig hosszú IF-utasítást írunk fel, úgy, hogy abban az ágban, amelyet elhagytunk volna, üres utasítás szerepel.

b. Ha a hivatkozási nyelv erről nem mond semmit, akkor a dolog implementációfüggő. Az implementációk többsége azt mondja, hogy egy szabad ELSE a legközelebbi, ELSE-el még nem párosított THEN-hez tartozik, vagyis az értelmezés bentről kifelé történik. Eszerint a fenti példánkban egy rövid IF-utasításba ágyaztunk egy hosszút.

c. A nyelv szintaktikája olyan, hogy egyértelmű a skatulyázás. Az Ada feltételes utasítása a következő:

```
IF feltétel THEN végrehajtható_utasítások
[ELSEIF feltétel THEN végrehajtható_utasítások ]...
[ELSE végrehajtható_utasítások ]
END IF;
```

Végül megjegyezzük, hogy a C-ben a feltétel zárójelek között szerepel és nincs `THEN` alapszó.

4.4.2. Többirányú elágaztató utasítás

A többirányú elágaztató utasítás arra szolgál, hogy a program egy adott pontján egymást kölcsönösen kizáró akárhány tevékenység közül egyet (vagy esetleg egyet sem)

végrehajtsunk. A tevékenységek közötti választást egy kifejezés értékei szerint tehetjük meg. Szintaktikája és szemantikája nyelvenként különböző. Nézzük a lehetséges megoldásokat.

Turbo Pascal:

```
CASE kifejezés OF
konstanslista : tevékenység
[ konstanslista : tevékenység ]...
[ ELSE tevékenység ]
END;
```

A `konstanslista` literálok, vagy intervallumok vesszővel elválasztott sorozata. Egy literál csak egy konstanslistában szerepelhet. A `kifejezés` és ennek megfelelően a `konstanslista` sorszámozott típusú lehet. Nem kötelező a kifejezés minden lehetséges értékére előírni tevékenységet. A `tevékenység` egy utasítás vagy egy utasítás csoport lehet.

Működése:

A `kifejezés` kiértékelődik, és az értéke a felírás sorrendjében hasonlításra kerül a konstansokkal. Ha van egyezés, végrehajtódik a megfelelő `konstanslista` utáni tevékenység, majd a CASE-utasítást követő utasítással folytatódik a program. Ha egyetlen konstanssal sincs egyezés, de van ELSE-ág, akkor végrehajtódik az abban magadott `tevékenység`, majd a CASE-utasítást követő utasítással folytatódik a program. Ha nincs ELSE-ág, akkor ez egy üres utasítás.

Ada:

```
CASE kifejezés1 IS
WHEN { kifejezés |tartomány| } [ |{kifejezés |tartomány }]...
     => végrehajtható_utasítások
[ WHEN { kifejezés |tartomány } [ |{kifejezés |tartomány }]...
     => végrehajtható_utasítások ]...
[ WHEN OTHERS => végrehajtható_utasítások ]
END CASE;
```

A WHEN-ágakban szereplő kifejezések és a tartományok értékeinek különbözniük kell. A `kifejezés1` skalár típusú lehet. A `kifejezés1` minden lehetséges értékére elő kell írni valamilyen tevékenységet.

Működése:

A `kifejezés1` kiértékelődik, és értéke a felírás sorrendjében hasonlításra kerül a WHEN-ágakban megadott kifejezések és tartományok értékeihez. Ha van egyezés, akkor végrehajtódnak a WHEN-ágban lévő utasítások, és a CASE-utasítást követő utasítással folytatódik a program. Ha nincs egyezés sehol, de van WHEN OTHERS ág, akkor az abban megadott utasítások hajtódnak végre és a CASE-utasítást követő utasítással folytatódik a program. Ha nincs WHEN OTHERS ág, akkor viszont futási hiba (kivétel) következik be. Ha valamely értékekre nem akarunk csinálni semmit, akkor egy olyan WHEN OTHERS ágat kell szerepeltetnünk, amely egy üres utasítást tartalmaz.

C:

```
SWITCH (kifejezés) {
CASE egész_konstans_kifejezés : [tevékenység]
[CASE egész_konstans_kifejezés : [tevékenység ]]...
[ DEFAULT: tevékenység]
};
```

A `kifejezés` típusának egészre konvertálhatónak kell lennie. A CASE-ágak értékei nem lehetnek azonosak. A `tevékenység` végrehajtható utasításokból állhat, vagy blokk lehet. A DEFAULT-ág akárhol elhelyezkedhet.

Működése:

Kiértékelődik a `kifejezés`, majd értéke a felírás sorrendjében hasonlításra kerül a CASE-ágak értékeivel. Ha van egyezés, akkor végrehajtódik az adott ágban megadott tevékenység, majd a program a következő ágakban megadott tevékenységeket is végrehajtja, ha még van ág. Ha nincs egyezés, de van DEFAULT-ág, akkor az ott megadott tevékenység hajtódik

végre, majd a program a következő ágakban megadott tevékenységeket is végrehajtja, ha még van ág. Ha nincs DEFAULT-ág, akkor ez egy üres utasítás. Tehát a C-ben külön utasítás kell ahhoz, hogy kilépjünk a SWITCH-utasításból, ha egy ág tevékenységét végrehajtottuk (l. BREAK-utasítás - 4.7. alfejezet).

PL/I:

```
SELECT [(kifejezés1)];
WHEN (kifejezés [, kifejezés ]...) tevékenység
[ WHEN (kifejezés [, kifejezés ]...) tevékenység ]...
[ OTHERWISE tevékenység ]
END;
```

A kifejezések típusa tetszőleges. A `tevékenység` egy utasítás, egy utasításcsoport, vagy egy blokk lehet.

Működése:

Ha szerepel a `kifejezés1`, akkor működése megegyezik az Adáéval. Tehát a PL/I-ben is a `kifejezés1` minden lehetséges értékére elő kell írni valamilyen tevékenységet. Ha nem szerepel, akkor a WHEN-ágakban megadott kifejezések értékét bitlánccá konvertálja és az első olyan ágat választja ki, amelynek a bitjei nem csupa nullák. Ha nincs ilyen, akkor ez egy üres utasítás.

A FORTRAN és a COBOL nem tartalmaz ilyen utasítást.

4.5. Ciklusszervező utasítások

A ciklusszervező utasítások lehetővé teszik, hogy a program egy adott pontján egy bizonyos tevékenységet akárhányszor megismételjünk.

Egy ciklus általános felépítése a következő:

fej

mag

vég

Az ismétlésre vonatkozó információk vagy a fejben vagy a végben szerepelnek.

A mag az ismétlendő végrehajtható utasításokat tartalmazza.

A ciklusok működésénél megkülönböztetünk két szélsőséges esetet. Az egyik az, amikor a mag egyszer sem fut le, ezt hívjuk *üres ciklusnak*. A másik az, amikor az ismétlődés soha nem áll le, ez a *végtelen ciklus*. A működés szerinti végtelen ciklus a programban nyilván szemantikai hibát jelent, hiszen az sohasem fejeződik be.

A programozási nyelvekben a következő ciklusfajtákat különböztetjük meg: *feltételes*, *előírt lépésszámú*, *felsorolásos*, *végtelen* és *összetett* ciklus.

Lássuk ezeket egyenként.

4.5.1. Feltételes ciklus

Ennél a ciklusnál az ismétlődést egy feltétel igaz vagy hamis értéke szabályozza. A feltétel maga vagy a fejben vagy a végben szerepel. Szemantikájuk alapján beszélünk kezdőfeltételes és végfeltételes ciklusról.

Kezdőfeltételes ciklus:

A feltétel a fejben jelenik meg.

Működése:

Kiértékelődik a feltétel. Ha igaz, végrehajtódik a ciklusmag, majd újra kiértékelődik a feltétel, és a ciklusmag mindaddig újra és újra lefut, amíg a feltétel hamissá nem válik. Tehát, ha egyszer beléptünk a magba, akkor ott kell valamikor olyan utasításnak végrehajtódnia, amely megváltoztatja a feltétel értékét.

A kezdőfeltételes ciklus lehet üres ciklus, ha a feltétel a legelső esetben hamis. Lehet végtelen ciklus is, ha a feltétel a legelső esetben igaz és mindig igaz is marad.

Végfeltételes ciklus:

A feltétel általában a végben van, de vannak nyelvek, amelyekben a fej tartalmazza azt.

Működése:

Először végrehajtódik a mag, majd ezután kiértékelődik a feltétel. Általában ha a feltétel hamis, újra végrehajtódik a mag. Tehát az ismétlődés mindaddig tart, míg a feltétel igazzá nem válik. Vannak viszont olyan nyelvek, amelyek igazra ismételnek. Nyilván itt is a magban kell gondoskodni a feltétel értékének megváltoztatásáról.

A végfeltételes ciklus soha nem lehet üres ciklus, a mag egyszer mindenféleképpen lefut. Végtelen ciklus viszont lehet, ha a feltétel értéke a második ismétlés után nem változik meg..

4.5.2. Előírt lépésszámú ciklus

Ennél a ciklusfajtánál az ismétlődésre vonatkozó információk (az ún. *ciklusparaméterek*) a fejben vannak. Minden esetben tartozik hozzá egy változó, a *ciklusváltozó*. A változó által felvett értékekre fut le a ciklusmag. A változó az értékeit egy tartományból veheti föl. Ezt a tartományt a fejben adjuk meg *kezdő*- és *végértékével*. A ciklusváltozó a tartománynak vagy minden elemét fölveheti (beleértve a kezdő- és végértéket is), vagy csak a tartományban szabályosan (ekvidisztánsan) elhelyezkedő bizonyos értékeket. Ekkor meg kell adni a tartományban a felvehető elemek távolságát meghatározó *lépésközt*. A változó az adott tartományt befuthatja növekvőleg, illetve csökkenőleg, ezt határozza meg az *irány*.

Az előírt lépésszámú ciklussal kapcsolatban a nyelveknek a következő kérdéseket kell megválaszolniuk:

1. *Milyen típusú lehet a ciklusváltozó?*

- Minden nyelv megengedi az egész típust.
- Egyes nyelveknél sorszámozott típusú lehet.
- Van néhány nyelv, amelyik megengedi a valósat is.
- A kezdőérték, a végérték és a lépésköz típusa meg kell, hogy egyezzen a ciklusváltozó típusával, vagy arra konvertálhatónak kell lennie.

2. *Milyen formában lehet megadni a kezdőértéket, végértéket és a lépésközt?*

   - Minden nyelv esetén megengedett a literál, változó és nevesített konstans.
   - Későbbi nyelveknél kifejezéssel is megadható.

3. *Hogyan határozódik meg az irány?*

   - A lépésköz előjele dönti el – ha pozitív, akkor növekvő, ha negatív, akkor csökkenő. Általában azok a nyelvek vallják ezt, melyekben a ciklusváltozó csak numerikus típusú lehet.
   - Külön alapszót kell használni.

4. *Hányszor értékelődnek ki a ciklusparaméterek?*

   - Általában egyszer, amikor a vezérlés a ciklushoz ér, és a ciklus működése alatt nem változnak.
   - Minden ciklusmag-végrehajtás előtt újra meghatározódnak.

5. *Hogyan fejeződhet be a ciklus?*

   - Szabályos lefutás
     - a ciklusparaméterek által meghatározott módon.
     - a ciklus magjában külön utasítással.
   - GOTO-utasítással, általában nem tekintjük szabályos befejezésnek.

6. *Mi lesz a ciklusváltozó értéke a ciklus lefutása után?*

   - Ha GOTO-val ugrunk ki a ciklusból, akkor a ciklusváltozó értéke az utoljára felvett érték lesz.
   - Szabályos befejezés esetén a hivatkozási nyelvek egy része nem nyilatkozik erről a kérdésről, másik része azt mondja, hogy a ciklusváltozó értéke határozatlan.
   - Az implementációk viszont a következőket mondják:
     - A ciklusváltozó értéke az az érték, amelyre utoljára futott le a ciklus.
     - A ciklusváltozó értéke az az érték, amire már éppen nem futott le a ciklus.
     - A ciklusváltozó értéke határozatlan.

Működését tekintve az előírt lépésszámú ciklus lehet előltesztelő, vagy hátultesztelő. A hivatkozási nyelveknek csak egy része definiálja ezt, ezért a működés gyakran implementációfüggő. Az implementációk többsége inkább az előltesztelő változatot valósítja meg.

Működés előltesztelő esetben:

A működés kezdetén meghatározódnak a *ciklusparaméterek*. Ezután a futtató rendszer megvizsgálja, hogy a megadott iránynak megfelelően a megadott tartomány nem üres-e. Ha üres (pl. a [10..1] tartomány növekvőleg), akkor ez egy üres ciklus. Különben a ciklusváltozó felveszi a kezdőértéket, és a mag lefut. Majd a futtató rendszer megvizsgálja, hogy az adott tartományban, az adott irányban, az adott lépésköznek megfelelően, a ciklusváltozó pillanatnyi értékéhez képest van-e még olyan érték, amit a ciklusváltozó felvehet. Ha van, akkor felveszi a következő ilyen értéket, és újra lefut a mag, ha nincs ilyen érték, akkor befejeződik a ciklus (szabályos befejezés).

Működés hátultesztelő esetben:

A működés kezdetén meghatározódnak a ciklusparaméterek. Ezután a ciklusváltozó felveszi a kezdőértéket, és a mag lefut. Majd a futtató rendszer megvizsgálja, hogy az adott tartományban, az adott irányban, az adott lépésköznek megfelelően, a ciklusváltozó pillanatnyi értékéhez képest van-e még olyan érték, amit a ciklusváltozó felvehet. Ha van, akkor felveszi a következő ilyen értéket, és újra lefut a mag, ha nincs ilyen érték, akkor befejeződik a ciklus.

Általában az eljárásorientált nyelvek megengedik, hogy a ciklusváltozónak értéket adjunk a magban.

Az előltesztelő előírt lépésszámú ciklus lehet üres ciklus, a hátultesztelő viszont nem. Egyes nyelvekben mindkét fajta lehet viszont végtelen ciklus (**miért?**).

4.5.3. Felsorolásos ciklus

A felsorolásos ciklus az előírt lépésszámú ciklus egyfajta általánosításának tekinthető. Van ciklusváltozója, amely explicit módon megadott értékeket vesz fel és minden felvett érték mellett lefut a mag. A ciklusváltozót és az értékeket a fejben adjuk meg, ez utóbbiakat kifejezéssel. A ciklusváltozó típusa általában tetszőleges. Nem lehet sem üres, sem végtelen ciklus.

4.5.4. Végtelen ciklus

A végtelen ciklus az a ciklusfajta, ahol sem a fejben, sem a végben nincs információ az ismétlődésre vonatkozóan. Működését tekintve definíció szerint végtelen ciklus, üres ciklus nem lehet. Használatánál a magban kell olyan utasítást alkalmazni, amelyik befejezteti a ciklust. Nagyon hatékony lehet eseményvezérelt alkalmazások implementálásánál.

4.5.5. Összetett ciklus

Az előző négy ciklusfajta kombinációiból áll össze. A ciklusfejben tetszőlegesen sok ismétlődésre vonatkozó információ sorolható föl, szemantikájuk pedig szuperponálódik. Nagyon bonyolult működésű ciklusok építhetők fel a segítségével.

A nyelvek egy részében vannak olyan vezérlésátadó utasítások, amelyeket bármilyen fajta ciklus magjában kiadva, a ciklus szabályos befejezését eredményezik. Végtelen ciklus szabályosan csak így fejeztethető be.

4.6. Ciklusszervező utasítások az egyes nyelvekben

FORTRAN:
Előírt lépésszámú ciklus:

```
      DO címke változó = k, v [,l]
      végrehajtható_utasítások
címke utasítás
```

A `változó`, `k`, `v`, `l` típusa csak egész lehet. A `k`, `v`, `l` (kezdőérték, végérték, lépésköz) csak literál vagy változó lehet, kifejezés nem. Ha nincs megadva az `l`, akkor a lépésköz alapértelmezett értéke 1, és az irány növekvő. Ha adott az `l`, akkor az irányt annak előjele adja meg. Implementációi általában hátultesztelőek. A ciklus végén nem lehet vezérlő utasítás, ilyenkor üres utasítást kell oda írni.

A későbbi verziókba bekerülnek a kezdő- és végfeltételes ciklusok is.

PL/I:

Benne minden ciklusfajta létezik.

Kezdőfeltételes ciklus:

```
DO WHILE(kifejezés);
végrehajtható_utasítások
END;
```

A `kifejezés` bitlánccá konvertálható kell legyen, akkor igaz, ha nem minden bit értéke 0.

Végfeltételes ciklus:

```
DO UNTIL(kifejezés);
végrehajtható_utasítások
END;
```

Előírt lépésszámú ciklus:

```
DO változó = k [TO v] [BY l];
végrehajtható_utasítások
END;
```

A `változó` aritmetikai vagy bitlánc típusú. Az irányt az `l` előjele dönti el. Ha nem szerepel a `TO v`, akkor végtelen ciklus. Ha nem szerepel a `BY l`, akkor a lépésköz 1.

Felsorolásos ciklus:

```
DO változó = kifejezés [, kifejezés ]... ;
végrehajtható_utasítások
END;
```

A `kifejezés` típusa tetszőleges.

Minden ciklusfajta kombinálható az összes többivel, így jönnek létre az összetett ciklusok.

Pascal:

Kezdőfeltételes ciklus:

```
WHILE feltétel DO végrehajtható_utasítás;
```

Végfeltételes ciklus:

```
REPEAT végrehajtható_utasítások UNTIL feltétel;
```

Előírt lépésszámú ciklus:

```
FOR változó := k { TO | DOWNTO } v DO végrehajtható_utasítás;
```

Előltesztelő. Az irányt alapszó dönti el, `TO` esetén növekvő, `DOWNTO` esetén csökkenő. A lépésközt a Pascal nem értelmezi.

Ada:

Kezdőfeltételes ciklus:

```
WHILE feltétel
LOOP
végrehajtható_utasítások
END LOOP;
```

Előírt lépésszámú ciklus:

```
FOR ciklusváltozó IN [REVERSE ] tartomány
LOOP
végrehajtható_utasítások
END LOOP;
```

Előltesztelő. `REVERSE` esetén csökkenőleg lépked végig a tartományon, ha nincs `REVERSE`, akkor növekvőleg. A `tartomány` sorszámozott típusú. Lépésközt az Ada nem értelmez. *A ciklusváltozó implicit módon a ciklus lokális változójaként deklarálódik a tartománynak megfelelő típussal és a ciklusmagban nevesített konstansként használható (tehát nem lehet neki értéket adni).*

Végtelen ciklus:

```
LOOP
végrehajtható_utasítások
END LOOP;
```

Az Adában az EXIT-utasítás segítségével minden ciklus magjából szabályosan ki tudunk lépni. A végtelen ciklusnál is ezt használjuk annak befejeztetésére.

C:

Kezdőfeltételes ciklus:

```
WHILE(feltétel) végrehajtható_utasítás;
```

A `feltétel` integrális vagy mutató típusú. Ha értéke nem `0`, akkor ismétel.

Végfeltételes ciklus:

```
DO végrehajtható_utasítás WHILE(feltétel);
```

Akkor ismétel, ha a `feltétel` értéke nem `0`.

FOR-ciklus:

```
FOR([kifejezés1]; [kifejezés2]; [kifejezés3]) végrehajtható_utasítás;
```

Ez a ciklus megfelel a következő kódnak:

```
kifejezés1;
WHILE(kifejezés2) {végrehajtható_utasítás; kifejezés3;}
```

A `kifejezés1` inicializáló kifejezés, kifejezésutasításként a ciklus működése előtt értékelődik ki. A `kifejezés2` felelős a ciklus befejezéséért, feltételként értelmeződik A `kifejezés3` kifejezésutasításként minden ciklusléps végén kiértékelődik. Ha a `kifejezés2` nem szerepel, akkor végtelen ciklus lesz. A végtelen ciklusból szabályosan a `BREAK` utasítással tudunk kilépni.

4.7. Vezérlő utasítások a C-ben

A C-ben három vezérlő utasítás van még az eddigiekben tárgyalt végrehajtható utasításokon kívül:

```
CONTINUE;
```

Ciklus magjában alkalmazható. A ciklus magjának hátralévő utasításait nem hajtja végre, hanem az ismétlődés feltételeit vizsgálja meg és vagy újabb ciklusléps kezd, vagy befejezi a ciklust.

```
BREAK;
```

Ciklus magjában, vagy többszörös elágaztató utasítás CASE-ágában helyezhető el. A ciklust szabályosan befejezteti, illetőleg kilép a többszörös elágaztató utasításból.

```
RETURN[ kifejezés];
```

Szabályosan befejezteti a függvényt és visszaadja a vezérlést a hívónak (l. 5.1. alfejezet).

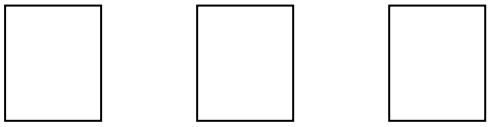
5. A PROGRAMOK SZERKEZETE

Az eljárásorientált programnyelvekben a program szövege többé-kevésbé független, szuverén részekre, ún. *programegységekre* tagolható.

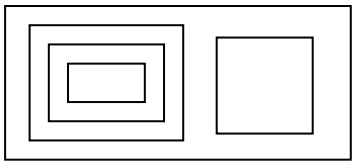
Ezen részekkel kapcsolatban a megválaszolandó kérdések a következők:

1. A *program teljes szövegét egyben kell-e lefordítani, vagy az feltördelhető önállóan fordítható részekre?*

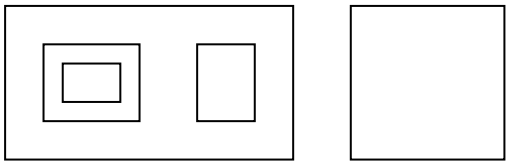
- Bizonyos nyelvekben a program fizikailag önálló részekből áll, melyek külön-külön fordíthatók. Ezek a részek mélységében nem strukturáltak.



- Más nyelvekben a programot egyetlen egységként kell lefordítani. Ilyenkor a program szövege mélységében strukturálható. A programegységek fizikailag nem függetlenek.



- Végül az előző kettő kombinációja is elképzelhető. Ezen nyelvekben fizikailag független, de tetszőleges belső struktúrával rendelkező programegységek léteznek.



2. *Ha a részek külön fordíthatók, mi alkothat egy önálló fordítási egységet?*

3. Milyen programegységek léteznek?

4. *Milyen a programegységek viszonya?*

5. *A programegységek hogyan kommunikálnak egymással?*

Az eljárásorientált nyelvekben az alábbi programegységek léteznek:

- alprogram
- blokk
- csomag
- taszk

5.1. Alprogramok

Az alprogram az eljárásorientált nyelvekben a procedurális absztrakció első megjelenési formája, alapvető szerepet játszik ebben a paradigmában, sőt meghatározója annak. Az alprogram mint absztrakciós eszköz egy bemeneti adatcsoportot képez le egy kimeneti adatcsoportra úgy, hogy egy specifikáció megadja az adatok leírását, de semmit nem tudunk magáról a tényleges leképezésről. Ismerjük a specifikációt, de nem ismerjük az implementációt.

Az alprogram, mint programozási eszköz az újrafelhasználás eszköze. Akkor alkalmazható, ha a program különböző pontjain ugyanaz a programrész megismétlődik. Ez az ismétlődő programrész kiemelhető, egyszer kell megírni, és a program azon pontjain, ahol ez a programrész szerepelt volna, csak *hivatkozni* kell rá – az alprogram az adott helyeken *meghívható*, *aktivizálható*.

Az alprogram attól lesz absztrakciós eszköz, hogy a kiemelt programrészt *formális paraméterekkel* látjuk el, vagyis *általánosabban* írjuk meg, mint ahogyan az adott helyeken szerepelt volna.

Formálisan az alprogram a következőképpen épül fel:

– fej vagy specifikáció
– törzs vagy implementáció

– vég

Az alprogram, mint programozási eszköz négy komponensből áll:

– név
– formális paraméter lista
– törzs
– környezet

A *név* egy azonosító, amely mindig a fejben szerepel. A *formális paraméter lista* is a specifikáció része. A formális paraméter listában azonosítók szerepelnek, ezek a törzsben saját programozási eszközök nevei lehetnek, és egy általános szerepkört írnak le, amelyet a hívás helyén konkretizálni kell az *aktuális paraméterek* segítségével.

A korai nyelvekben a formális paraméter listán csak a paraméterek nevei szerepelhettek, a későbbi nyelvekben viszont itt megadhatók további olyan információk, melyek a paraméterek futás közbeni viselkedését szabályozzák.

A formális paraméter lista kerek zárójelek között áll. A nyelvek egy része szerint a zárójelek a formális paraméter listához, mások szerint a névhez tartoznak.

A formális paraméter lista lehet üres is, ekkor *paraméter nélküli* alprogramról beszélünk.

A *törzsben* deklarációs és végrehajtható utasítások szerepelnek. A nyelvek egy része azt mondja, hogy ezeket el kell különíteni egymástól, tehát a törzsnek van egy deklarációs és egy végrehajtható része. Más nyelvek szerint viszont a kétféle utasítás tetszőlegesen keverhető.

Az alprogramban deklarált programozási eszközök az alprogram *lokális* eszközei, ezek nevei az alprogram *lokális nevei*. A lokális nevek az alprogramon kívülről nem láthatók, azokat az alprogram elrejti a külvilág elől. Ezzel szemben léteznek a *globális nevek*, melyeket nem az adott alprogramban deklaráltunk, hanem valahol rajta *kívül*, de a törzsben szabályosan hivatkozhatunk rájuk.

Egy alprogram *környezete* alatt a globális változók együttesét értjük.

Az alprogramoknak két fajtája van: *eljárás* és *függvény*.

Az eljárás olyan alprogram, amely valamilyen tevékenységet hajt végre. A hívás helyén ezen tevékenység eredményét használhatjuk fel. Az eljárás a hatását a paramétereinek vagy a környezetének megváltoztatásával illetve a törzsben elhelyezett végrehajtható utasítások által meghatározott tevékenység elvégzésével fejti ki.

A függvény olyan alprogram, amelynek az a feladata, hogy egyetlen értéket határozzon meg. Ez az érték általában tetszőleges típusú lehet. A függvény visszatérési értékének a típusa egy további olyan információ, amely hozzátartozik a függvény specifikációjához. A függvény visszatérési értékét mindig a neve hordozza, formálisan az közvetíti vissza a hívás helyére. A függvény törzsének végrehajtható utasításai a visszatérési érték meghatározását szolgálják.

Azt a szituációt, amikor a függvény megváltoztatja paramétereit, vagy környezetét, a függvény *mellékhatásának* nevezzük. A mellékhatást általában károsnak tartják.

Egy eljárást aktivizálni utasításszerűen lehet, azaz az eljáráshívás elhelyezhető bárhol, ahol végrehajtható utasítás állhat. Egyes nyelvekben van külön eljáráshívásra szolgáló alapszó (ez nagyon gyakran a `CALL`). Más nyelvekben nincs külön alapszó. Híváskor a vezérlés átadódik az eljárásra.

Formálisan a hívás a következőképpen néz ki:

```
[alapszó] eljárásnév(aktuális_paraméter_lista)
```

Egy eljárás *szabályosan* befejeződhet a különböző nyelvekben, ha

- elérjük a végét,
- külön utasítással befejeztetjük, ez bárhol kiadható az eljárás törzsében.

Szabályos befejeződés esetén a program a hívást követő utasításon folytatódik.

Általában nem szabályos befejezésnek tekintjük a következőket:

- A nyelvek általában megengedik, hogy az eljárásból GOTO-utasítással kilépjünk, a megadott címkén folytatva a programot.
- Van olyan eszköz a nyelvben, amely hatására a teljes program befejeződik, ekkor nyilván az eljárás maga is véget ér, és a vezérlés az operációs rendszerhez kerül vissza.

Függvényt meghívni csak *kifejezésben* lehet, a hívás alakja:

```
függvénynév(aktuális_paraméter_lista)
```

A függvényhívás után normális befejeződést feltételezve a vezérlés a kifejezésbe tér vissza és továbbfolytatódik annak a kiértékelése.

Egy függvény a következő módokon határozhatja meg a visszatérési értékét:

- A függvény törzsében változóként használható a függvény neve (pl. FORTRAN). A törzsben tetszőlegesen felhasználható és változtatható annak értéke. A visszatérési érték a legutoljára kapott érték lesz.
- A függvény törzsében a függvény nevéhez értéket kell hozzárendelni. A függvény neve azt az értéket hordozza, amit utoljára kapott.
- Külön utasítás szolgál a visszatérési érték meghatározására, amely egyben be is fejezteti a függvényt.

A függvény szabályosan befejeződhet, ha
- elérjük a végét és már van visszatérési érték,
- befejeztető utasítást alkalmazunk és már van visszatérési érték,
- olyan befejeztető utasítást alkalmazunk, amely egyben meghatározza a visszatérési értéket is.

Nem szabályos a befejeződés, ha
- elérjük a végét és nem határoztuk meg a visszatérési értéket,
- olyan befejeztető utasítást alkalmazunk, amely nem határozza meg a visszatérési értéket és más módon sincs megadva az,
- GOTO-utasítással lépünk ki.

Az utolsó esetben a vezérlés a címkére adódik, tehát egyáltalán nem térünk vissza a kifejezés kiértékeléséhez. Kerülendő megoldás, nem biztonságos kódot eredményez. Az első két esetben a vezérlés visszatér a kifejezés kiértékeléséhez, azonban a függvény visszatérési értéke határozatlan. Egyes nyelvekben (pl. C) ez bizonyos szituációkban nem jelent problémát, általában viszont szemantikai hiba.

Az eljárásorientált programozási nyelvekben megírt minden programban kötelezően lennie kell egy speciális programegységnek, amit *főprogramnak* hívunk. Ez alprogram jellegű, a betöltő neki adja át a vezérlést és az összes többi programegység működését ő koordinálja. Egy program szabályos befejeződése a főprogram befejeződésével történik meg, ekkor a vezérlés visszakerül az operációs rendszerhez.

5.2. Hívási lánc, rekurzió

Egy programegység bármikor meghívhat egy másik programegységet, az egy újabb programegységet, és így tovább. Így kialakul egy *hívási lánc*. A hívási lánc első tagja mindig a főprogram. A hívási lánc minden tagja *aktív*, de csak a legutoljára meghívott programegység *működik*. Szabályos esetben mindig az utoljára meghívott programegység fejezi be legelőször a működését, és a vezérlés visszatér az őt meghívó programegységbe. A hívási lánc futás közben dinamikusan épül föl és bomlik le.

Azt a szituációt, amikor egy aktív alprogramot hívunk meg, *rekurziónak* nevezzük.

A rekurzió kétféle lehet:

- közvetlen: egy alprogram önmagát hívja meg, vagyis a törzsben van egy hivatkozás saját magára.
- közvetett: a hívási láncban már korábban szereplő alprogramot hívunk meg.

A rekurzióval megvalósított algoritmus mindig átírható iteratív algoritmussá.

Egyes nyelvek nem ismerik a rekurziót (pl. FORTRAN), mások azt mondják, hogy minden alprogram alapértelmezett módon rekurzív, végül a nyelvek harmadik csoportjában a programozó döntheti el, hogy egy adott alprogram rekurzív legyen-e, vagy sem.

5.3. Másodlagos belépési pontok

Egyes nyelvek megengedik, hogy egy alprogramot meghívni ne csak a fejen keresztül lehessen, hanem a törzsben ki lehessen alakítani ún. *másodlagos belépési pontokat*, így vagy a

fejben megadott névvel vagy a másodlagos belépési pont nevével lehet hivatkozni az alprogramra. A másodlagos belépési pont képzése formálisan meg kell feleljen a specifikációnak. Ha függvényről van szó, akkor a típusnak meg kell egyeznie. Ha az adott alprogramba a fejen keresztül lépünk be, akkor az alprogram teljes törzse végrehajtódik, másodlagos belépési pont használata esetén a törzsnek csupán egy része hajtódik végre.

5.4. Paraméterkiértékelés

Paraméterkiértékelés alatt értjük azt a folyamatot, amikor egy alprogram hívásánál egymáshoz rendelődnek a formális- és aktuális paraméterek, és meghatározódnak azok az információk, amelyek a paraméterátadásnál a kommunikációt szolgáltatják.

A paraméterkiértékelésnél mindig a formális paraméter lista az elsődleges, ezt az alprogram specifikációja tartalmazza, egy darab van belőle. Aktuális paraméter lista viszont annyi lehet, ahányszor meghívjuk az alprogramot. Tehát az egymáshoz rendelésnél mindig a formális paraméter lista a meghatározó, mindig az aktuális paramétereket rendeljük a formálisakhoz.

A paraméterkiértékelésnek három aspektusa van, ezek az alábbi kérdésekre adnak választ.

1. *Melyik formális paraméterhez melyik aktuális paraméter fog hozzárendelődni?*

Ez történhet *sorrendi kötés* vagy *névszerinti kötés* szerint.

Sorrendi kötés esetén a formális paraméterekhez a felsorolás sorrendjében rendelődnek hozzá az aktuális paraméterek: az elsőhöz az első, a másodikhoz a második, és így tovább. Ezt a lehetőséget minden nyelv ismeri, és általában ez az alapértelmezés.

A névszerinti kötés esetén az aktuális paraméter listában határozhatjuk meg az egymáshoz rendelést úgy, hogy megadjuk a formális paraméter nevét és mellette valamilyen szintaktikával az aktuális paramétert. Ilyenkor lényegtelen a formális paraméterek sorrendje. Néhány nyelv ismeri.

Alkalmazható a sorrendi és névszerinti kötés kombinációja együtt is úgy, hogy az aktuális paraméter lista elején sorrendi kötés, utána névszerinti kötés van.

2. *Hány darab aktuális paramétert kell megadni?*

Lehetséges, hogy a formális paraméterek száma fix, a formális paraméter lista adott számú paramétert tartalmaz. Ekkor a paraméterkiértékelés kétféle módon mehet végbe:

- Az aktuális paraméterek számának meg kell egyeznie a formális paraméterek számával.
- Az aktuális paraméterek száma kevesebb lehet, mint a formális paraméterek száma. Ez csak érték szerinti paraméterátadási mód esetén lehetséges. Azon formális paraméterekhez, amelyekhez nem tartozik aktuális paraméter, a formális paraméter listában alapértelmezett módon rendelődik érték

Lehet olyan eset, amikor a formális paraméterek száma nem rögzített, tetszőleges. Ekkor az aktuális paraméterek száma is tetszőleges. Létezik olyan megoldás is, hogy a paraméterek számára van alsó korlát, tehát legalább ennyi aktuális paramétert szerepeltetni kell.

3. *Mi a viszony a formális és aktuális paraméterek típusai között?*

A nyelvek egyik része a típusegyenértékűséget vallja, ekkor az aktuális paraméter típusának azonosnak kell lennie a formális paraméter típusával. A nyelvek másik része a típuskényszerítés alapján azt mondja, hogy az aktuális paraméter típusának konvertálhatónak kell lennie a formális paraméter típusára.

5.5. Paraméterátadás

A paraméterátadás az alprogramok és más programegységek közötti kommunikáció egy formája. A paraméterátadásnál mindig van egy *hívó*, ez tetszőleges programegység és egy *hívott*, amelyik mindig alprogram. Kérdés, hogy melyik irányban és milyen információ mozog?

A nyelvek a következő paraméterátadási módokat ismerik:

– érték szerinti
– cím szerinti
– eredmény szerinti
– érték-eredmény szerinti
– név szerinti
– szöveg szerinti

Az *érték* szerinti paraméterátadás esetén a formális paramétereknek van címkomponensük a hívott alprogram területén. Az aktuális paraméternek rendelkeznie kell értékkomponenssel a hívó oldalon. Ez az érték meghatározódik a paraméterkiértékelés folyamán, majd átkerül a hívott alprogram területén lefoglalt címkomponensre. A formális paraméter kap egy kezdőértéket, és az alprogram ezzel az értékkel dolgozik a saját területén. Az információáramlás egyirányú, a hívótól a hívott felé irányul. A hívott alprogram semmit sem tud a hívóról, a saját területén dolgozik. Mindig van egy értékmásolás, és ez az érték tetszőleges bonyolultságú lehet. Ha egy egész adatcsoportot kell átmásolni, az hosszadalmas. Lényeges, hogy a két programegység egymástól függetlenül működik, és egymás működését az érték meghatározáson túl nem befolyásolják.

Az aktuális paraméter kifejezés lehet.

A *cím* szerinti paraméterátadásnál a formális paramétereknek nincs címkomponensük a hívott alprogram területén. Az aktuális paraméternek viszont rendelkeznie kell címkomponenssel a hívó területén. Paraméterkiértékeléskor meghatározódik az aktuális paraméter címe és átadódik a hívott alprogramnak, ez lesz a formális paraméter címkomponense. Tehát a meghívott alprogram a hívó területén dolgozik. Az információátadás kétirányú, az alprogram a hívó területéről átvehet értéket, és írhat is oda. Az alprogram átnyúl a hívó területre. Időben gyors, mert nincs értékmásolás, de veszélyes lehet, mivel a hívott alprogram hozzáfér a hívó területén lévő információkhoz, és szabálytalanul használhatja föl azokat.

Az aktuális paraméter változó lehet.

Az *eredmény* szerinti paraméterátadásnál a formális paraméternek van címkomponense a hívott alprogram területén, az aktuális paraméternek pedig lennie kell címkomponensének.

Paraméterkiértékeléskor meghatározódik az aktuális paraméter címe és átadódik a hívott alprogramnak, azonban az alprogram a saját területén dolgozik, és a futás közben nem használja ezt a címet. A működésének befejeztekor viszont átmásolja a formális paraméter értékét erre a címkomponensre. A kommunikáció egyirányú, a hívottól a hívó felé irányul. Van értékmásolás.

Az aktuális paraméter változó lehet.

Az *érték-eredmény* szerinti paraméterátadásnál a formális paraméternek van címkomponense a hívott területén és az aktuális paraméternek rendelkeznie kell érték- és címkomponenssel. A paraméterkiértékelésnél meghatározódik az aktuális paraméter értéke és címe és mindkettő átkerül a hívotthoz. Az alprogram a kapott értékkel, mint kezdőértékkel kezd el dolgozni a saját területén és a címet nem használja. Miután viszont befejeződik, a formális paraméter értéke átmásolódik az aktuális paraméter címére. A kommunikáció kétirányú, kétszer van értékmásolás.

Az aktuális paraméter változó lehet.

Név szerinti paraméterátadásnál az aktuális paraméter egy, az adott szövegkörnyezetben értelmezhető tetszőleges szimbólumsorozat lehet. A paraméterkiértékelésnél rögzítődik az alprogram szövegkörnyezete, itt értelmezésre kerül az aktuális paraméter, majd a szimbólumsorozat a formális paraméter nevének minden előfordulását felülírja az alprogram szövegében és ezután fut le az. Az információáramlás iránya az aktuális paraméter adott szövegkörnyezetbeli értelmezésétől függ.

A *szöveg* szerinti paraméterátadás a név szerintinek egy változata, annyiban különbözik tőle, hogy a hívás után az alprogram elkezd működni, az aktuális paraméter értelmező szövegkörnyezetének rögzítése és a formális paraméter felülírása csak akkor következik be, amikor a formális paraméter neve először fordul elő az alprogram szövegében a végrehajtás folyamán.

Alprogramok esetén típust paraméterként átadni nem lehet.

Egy adott esetben a paraméterátadás módját az alábbiak döntik el:

- a nyelv csak egyetlen paraméterátadási módot ismer (pl. C)

- a formális paraméter listában explicit módon meg kell adni a paraméterátadási módot (pl. Ada)
- az aktuális és formális paraméter típusa együttesen dönti el (pl. PL/I)
- a formális paraméter típusa dönti el (pl. FORTRAN)

Az alprogramok formális paramétereit három csoportra oszthatjuk:

- *Input* paraméterek: ezek segítségével az alprogram kap információt a hívótól (pl. érték szerinti paraméterátadás).
- *Output* paraméterek: a hívott alprogram ad át információt a hívónak (pl. eredmény szerinti paraméterátadás).
- *Input-output* paraméterek: az információ mindkét irányba mozog (pl. érték-eredmény szerinti paraméterátadás).

5.6. A blokk

A blokk olyan programegység, amely csak másik programegység belsejében helyezkedhet el, külső szinten nem állhat.

Formálisan a blokknak van *kezdete*, *törzse* és *vége*. A kezdetet és a véget egy-egy speciális karaktersorozat vagy alapszó jelzi. A törzsben lehetnek deklarációs és végrehajtható utasítások. Ugyanúgy mint az alprogramoknál, ezek az utasítások vagy tetszőlegesen keverhetők, vagy van külön deklarációs rész és végrehajtható rész. A blokknak nincs paramétere. A blokknak egyes nyelvekben lehet neve. Ez általában a kezdet előtt álló címke.

A blokk bárhol elhelyezhető, ahol végrehajtható utasítás állhat.

Blokkot aktivizálni vagy úgy lehet, hogy szekvenciálisan rákerül a vezérlés, vagy úgy, hogy GOTO-utasítással ráugrunk a kezdetére. Egy blokk befejeződhet, ha elértük a végét, vagy GOTO-utasítással kilépünk belőle, vagy befejeztetjük az egész programot a blokkban.

A blokkot az eljárásorientált nyelveknek csak egy része ismeri. Szerepe a nevek hatáskörének elhatárolásában van.

5.7. Hatáskör

A hatáskör a nevekhez kapcsolódó fogalom. Egy név *hatásköre* alatt értjük a *program szövegének azon részét*, ahol az adott név ugyanazt a programozási eszközt hivatkozza, tehát jelentése, felhasználási módja, jellemzői azonosak. A hatáskör szinonimája a *láthatóság*.

A név hatásköre az eljárásorientált programnyelvekben a programegységekhez illetve a fordítási egységekhez kapcsolódik.

Egy programegységben deklarált nevet a programegység *lokális nevének* nevezzük. Azt a nevet, amelyet nem a programegységben deklaráltunk, de ott hivatkozunk rá, *szabad névnek* hívjuk.

Azt a tevékenységet, mikor egy név hatáskörét megállapítjuk, *hatáskörkezelésnek* hívjuk. Kétféle hatáskörkezelést ismerünk, a *statikus* és a *dinamikus* hatáskörkezelést.

A statikus hatáskörkezelés fordítási időben történik, a fordítóprogram végzi. Alapja a programszöveg programegység szerkezete. Ha a fordító egy programegységben talál egy szabad nevet, akkor kilép a *tartalmazó* programegységbe, és megnézi, hogy a név ott lokális-e. Ha igen vége a folyamatnak, ha nem, akkor tovább lépked kifelé, egészen addig, amíg meg nem találja lokális névként, vagy el nem jut a legkülső szintre. Ha kiért a legkülső szintre, akkor két eset lehetséges:

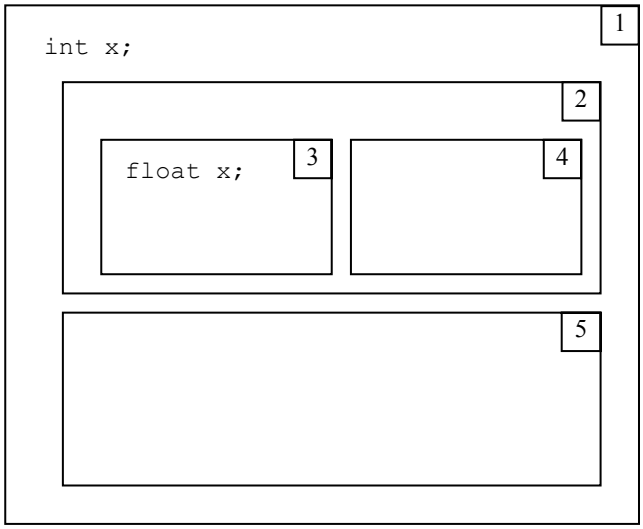
- A nyelvek egy része azt mondja, hogy a programozónak minden nevet deklarálni kell. Így ha egy név nem volt deklarálva, az fordítási hiba.
- A nyelvek másik része ismeri az automatikus deklarációt, és a névhez a fordító hozzárendeli az automatikus deklaráció szabályainak megfelelő attribútumokat. A név ilyenkor tehát a legkülső szint lokális neveként értelmeződik.

```
x:integer;
```

Statikus hatáskörkezelés esetén egy lokális név hatásköre az a programegység, amelyben deklaráltuk és minden olyan programegység, amelyet ez az adott programegység tartalmaz, hacsak a tartalmazott programegységekben a nevet nem deklaráltuk újra.

A hatáskör befelé terjed, kifelé soha. Egy programegység a lokális neveit bezárja a külvilág elől. Azt a nevet, amely egy adott programegységben nem lokális név, de onnan látható, *globális névnek* hívjuk. A globális név, lokális név relatív fogalmak. Ugyanaz a név az egyik programegység szempontjából lokális, egy másikban globális, egy harmadikban pedig nem is látszik.

Példa:



Az ábrán egymásba skatulyázott programegységek láthatók. A 1-es programegységben deklaráltunk egy `x` nevű `int` típusú változót, amely itt lokális. A 2-es, 4-es és 5-ös programegységben hivatkozhatunk az `x` névre, amely itt globális név. Az `x` nevet a 3-as programegységben újradeklaráltuk `float` típusúként. Így ez az újradeklarált név a 3-as programegységben már egy másik változót jelöl, ez itt lokális és másik programegységben nem látszik. Viszont a 3-as programegységben nem tudunk hivatkozni az `int` típusú `x` nevű változóra, mert az új deklaráció elfedi azt. Szemléletesen azt mondjuk, hogy az 1-es programegységben deklarált `x` hatáskörében „lyuk” keletkezik.

A dinamikus hatáskörkezelés futási idejű tevékenység, a futtató rendszer végzi. Alapja a hívási lánc. Ha a futtató rendszer egy programegységben talál egy szabad nevet, akkor a hívási láncon keresztül kezd el visszalépkedni mindaddig, amíg meg nem találja lokális névként, vagy a hívási lánc elejére nem ér. Ez utóbbi esetben vagy futási hiba keletkezik, vagy automatikus deklaráció következik be.

Dinamikus hatáskörkezelésnél egy név hatásköre az a programegység, amelyben deklaráltuk, és minden olyan programegység, amely ezen programegységből induló hívási láncban helyezkedik el, hacsak ott nem deklaráltuk újra a nevet. Újradeklarálás esetén a hívási lánc további elemeiben az újradeklarált eszköz látszik, nincs „lyuk a hatáskörben” szituáció.

Statikus hatáskörkezelés esetén a programban szereplő összes név hatásköre a forrásszöveg alapján egyértelműen megállapítható. Dinamikus hatáskörkezelésnél viszont a hatáskör futási időben változhat és más-más futásnál más-más lehet.

Az eljárásorientált nyelvek a statikus hatáskörkezelést valósítják meg. Általánosságban elmondható, hogy az alprogramok formális paraméterei az alprogram lokális eszközei, így neveik az alprogram lokális nevei. Viszont a programegységek neve a programegység számára globális. A kulcsszavak, mint nevek a program bármely pontjáról láthatók. A standard azonosítók, mint nevek azon programegységekből láthatók, ahol nem deklaráltuk újra őket.

A globális változók az eljárásorientált nyelvekben a programegységek közötti kommunikációt szolgálják.

5.8. Fordítási egység

Az eljárásorientált nyelvekben a program közvetlenül fordítási egységekből épül föl. Ezek olyan forrásszöveg-részek, melyek önállóan, a program többi részétől fizikailag különválasztva fordíthatók le. Az egyes nyelvekben a fordítási egységek felépítése igen eltérő lehet. A fordítási egységek általában hatásköri és gyakran élettartam definiáló egységek is.

5.9. Az egyes nyelvek eszközei

FORTRAN:

A FORTRAN fizikailag különálló programegységekből építi fel a programot, a programegységek külön fordíthatók, és nem skatulyázhatók egymásba. Csak az alprogramot ismeri. A FORTRAN-ban egy fordítási egység egyetlen programegység, vagy programegységek tetszőleges csoportja. Mivel így a FORTRAN-ban nincsenek globális változók, a programegységek egy speciális tárterületet használhatnak a kommunikációra, az ún. *közös adatmezőt*. A közös adatmezőt minden olyan programegység írhatja és olvashatja, amelyben szerepel a következő deklaráció:

`COMMON [/n/] a1 [(d1)] [, a2 [(d2)]]...`

Az a_i skalár vagy tömb típusú (d_i a dimenziódeklarációt jelenti) változó, amelynek típusát külön deklarációs utasításban kell megadni. Az `n` a közös adatmező neve. Közös adatmezőből akármennyi lehet. A tárnak ezen a speciális területén a felsorolás sorrendjében helyezi el a rendszer a változókat. A közös tárterület felosztása az egyes alprogramokban más-más lehet, sőt még a változók típusa is eltérhet. A megnevezetteknél annyi részre osztja a tárat, ahány név szerepel, ha a név hiányzik, akkor a tárterület végére rakja a változót.

A FORTRAN-ban van automatikus deklaráció.

A FORTRAN-ban a főprogramnak nincs külön kezdő utasítása.

Az eljárás alakja:

```
SUBROUTINE név[(formális_paraméter_lista)]
deklarációk
végrehajtható_utasítások
END
```

A formális paraméter listán csak nevek vannak, típusukat a deklarációs részben kell megadni. Hívása a `CALL` standard azonosítóval történik. Szabályosan befejeződik a `RETURN [n]` utasítás hatására, ahol `n` egy előjel nélküli egész.

A FORTRAN ismeri a másodlagos belépési pontot. Ha ilyet helyezünk el egy eljárásban, akkor a benne és az eljárásfejben megadott formális paraméterek eltérhetnek egymástól. Alakja:

```
ENTRY név[(formális_paraméter_lista)]
```

A paraméterkiértékelésnél sorrendi kötés, típusegyenértékűség és számbeli egyeztetés van.

A paraméterek a formális paraméter listán háromféleképpen jelenhetnek meg: A formális paraméter lehet egy azonosító. Ilyenkor ez a törzsben tetszőleges változó, vagy alprogram neveként szerepelhet. Ha az azonosító perjelek (/) között szerepel, akkor csak skalárváltozót nevezhet meg. Végül a formális paraméter lehet `*`.

Ha a formális paraméter skalár változó, akkor az aktuális paraméter kifejezés; ha tömbtípusú változó, akkor tömb típusú változó, vagy indexes változó; ha alprogram név, akkor alprogram név; és ha `*`, akkor a hívó alprogram egy címkéje lehet `&címke` formában. Ez utóbbi esetben alkalmazható a `RETURN n`, ami visszaadja a vezérlést az aktuális paraméter listában `n`-dikként megadott címkére.

A paraméterátadás a formális paramétertől függően:

- Tömb típus esetén cím szerinti.
- Skalár változóknál érték szerinti.
- Alprogram neve esetén név szerinti.
- `/azonosító/` esetén cím szerinti.
- * esetén cím szerinti.

A függvény alakja:

```
[típus] FUNCTION név(p1 [,p2]... )
deklarációk
végrehajtható_utasítások
END
```

A FORTRAN szerint kell legalább egy formális paraméter. Vagy a fejben deklaráljuk a függvény típusát, vagy külön a deklarációs részben. Másodlagos belépési pont van itt is, de a formális paraméter listának szigorúan egyeznie kell a másodlagos belépési pontban felsorolt paraméterekkel és a típusnak is azonosnak kell lenni. A formális paraméterek között a * nem szerepelhet. A függvény neve lokális változóként használható a törzsben. Befejezése a RETURN utasítással történik de ez nem határoz meg értéket, A törzsben kell értékadásról gondoskodni, és az utoljára adott értékkel tér vissza. Ha egy alprogram meg akar hívni egy függvényt, akkor a függvény nevét a megfelelő típussal ott is deklarálni kell.

A FORTRAN nem ismeri a rekurziót.

A faktoriálist kiszámító függvény:

```
      REAL FUNCTION FAKT(I)
      FAKT=1
      IF (I .EQ. 0 .OR. I .EQ. 1) RETURN
      DO 20 K=2,I
20    FAKT=FAKT*K
      RETURN
      END
```

PL/I:

A PL/I ismeri az alprogramot, a blokkot és a taszkot a programegységek közül. Az alprogramok lehetnek egymástól függetlenek, és tetszőlegesen egymásba skatulyázhatók. Itt is létezik főprogram, amely egy speciális alprogram. Fordítási egységet alkothat a főprogram, az alprogramok vagy ezek tetszőleges együttese.

Az alprogram alakja:

```
név:PROCEDURE [(formális_paraméter_lista)][OPTIONS(opciólista)]
   [RECURSIVE][RETURNS(attribútumok)];
utasítások
END [név];
```

Nem válik el a deklarációs és a végrehajtható rész, az utasítások tetszőlegesen keverhetők. A `név` címke jellegű. A `formális_paraméter_lista` csak a paraméterek neveit tartalmazza, deklarálni őket a törzsben kell. A formális paraméterek a törzsben változók, címkék, belépési pontok, vagy pedig állományok nevei lehetnek.

Az `opciólista` az adott alprogram futás közbeni viselkedését szabályozza. Ha itt a `MAIN` opció szerepel, akkor ez lesz a főprogram, és az összes többi opció nem szerepelhet. Ha nem ez az opció áll itt, akkor függvényről vagy eljárásról van szó. A programozó dönti el, hogy az rekurzív legyen-e vagy sem. Ha a `RECURSIVE` alapszó szerepel, akkor az adott alprogram rekurzív lesz, különben pedig nem.

Ha szerepel a `RETURNS`, akkor függvényről van szó, az `attribútumok` a visszatérési érték attribútumait adják meg.

Az eljárás hívása a `CALL` utasítással történik.

Egy eljárás szabályosan befejeződik, ha elérjük a végét, vagy valahol a törzsében kiadjuk `RETURN` utasítást.

Egy függvény a `RETURN(kifejezés);` utasítás hatására fejeződik be, és a `kifejezés` adja meg a függvény visszatérési értékét.

A PL/I-ben a másodlagos belépési pont alakja:

```
név:ENTRY [(formális_paraméter_lista)][RETURNS(attribútumok)];
```

Ha függvényről van szó, akkor az attribútumoknak és a formális paraméter listának meg kell egyezni, ha eljárásról, akkor nem.

A paraméterkiértékelésnél sorrendi kötés és számbeli egyeztetés, továbbá típuskényszerítés van.

A paraméterátadás, ha az aktuális paraméter:

– címke vagy a formális paraméterrel megegyező típusú változó, akkor cím szerinti,

– belépési pont vagy állomány neve, akkor név szerinti,

– minden más esetben pedig érték szerinti.

A blokk a következőképpen néz ki:

```
[címke:] BEGIN
utasítások
END [címke];
```

Itt is tetszőlegesen keverhetők a deklarációs- és végrehajtható utasítások.

A PL/I az élettartamot attribútumokkal határozza meg:

– STATIC: statikus tárkiosztás.

– AUTOMATIC: dinamikus tárkiosztás, ez az alapértelmezett.

– CONTROLLED: programozó által vezérelt tárkiosztás. A programozó a következő utasításokat használhatja ilyen esetben:

  ALLOCATE: a rendszer helyezi el a változót a tárban.

  FREE: memóriaterület felszabadítása.

– BASED: bázisolt változók alkalmazása. A címkomponenst a programozó rendeli a változóhoz egy korábban elhelyezett objektum címéhez képest. Ez relatív cím, abszolút címet általában nem kezel. Nem ismeri a programozó által vezérelt tárkiosztási módok közül az abszolút címes tárkiosztást.

A PL/I a statikus hatáskörkezelést vallja. Ha ugyanazt a nevet két különböző fordítási egységben az EXTERNAL attribútummal deklaráljuk úgy, hogy minden más attribútumuk megegyezik, akkor az ugyanaz a név. A legkülső szinten lévő alprogramok nevei alapértelmezés szerint EXTERNAL attribútumúak.

A PL/I gyakorlatilag minden attribútumra vonatkozóan rendelkezik automatikus deklarációval.

A faktoriálist kiszámító függvény iteratív változata:

```
FAKT:PROCEDURE(I);
F=1;
DO L=2 TO I;
F=F*L;
END;
RETURN(F);
END FAKT;
```

A faktoriálist kiszámító függvény rekurzív változata:

```
FAKT:PROCEDURE(I) RECURSIVE;
F=1;
IF I>1 THEN F=I*FAKT(I-1)
RETURN(F);
END FAKT;
```

Pascal:

A Pascalban a fordítási egység a főprogram. A programegységek közül csak az alprogramot ismeri. Egyes verziókban van csomag (pl. a Turbo Pascal unitja).

A főprogram alakja:

```
PROGRAM név[(környzeti_paraméterek)];
deklarációs_rész
BEGIN
végrehajtható_utasítások
END.
```

A főprogram is rendelkezik formális paraméter listával, a formális paraméterek száma nem fix. Ennek alapvető szerepe van az operációs rendszerrel történő kommunikációban. Az alprogramok a főprogram deklarációs részébe skatulyázandók. Az alprogramok felépítése teljesen hasonló elveket követ, beleértve az alprogramok skatulyázását is.

Az eljárás lakja:

```
PROCEDURE név[(formális_paraméter_lista)];
deklarációs_rész
BEGIN
végrehajtható_utasítások
END;
```

A függvény lakja:

```
FUNCTION név[(formális paraméter_lista)] : típus;
deklarációs_rész
BEGIN
végrehajtható_utasítások
END;
```

A formális paraméter lista paramétercsoportokból áll, melyeket pontosvessző választ el egymástól. Egy paramétercsoport alakja:

```
[ VAR ] azonosító [, azonosító]... : típus
```

Ha a paramétercsoportban szerepel a `VAR` kulcsszó, akkor a paraméterátadás cím szerinti, különben érték szerinti. A paraméterkiértékelésénél sorrendi kötés, számbeli- és típusegyeztetés van.

Az eljáráshívásra nincs külön alapszó.

A rekurzió alapértelmezett.

A függvény nevének a végrehajtható utasítások között értéket kell kapnia, és az utoljára kapott értékkel tér vissza, ha elérjük a függvény végét (szabályos befejeződés). Az eljárás is akkor fejeződik be szabályosan, ha elérjük a végét.

A Pascalban dinamikus élettartam-kezelés és programozó által vezérelt tárkiosztás van.

A Pascalban egy név csak a deklarációjától kezdve látszik.

A faktoriálist kiszámító függvény:

```
FUNCTION FAKT(I:INTEGER):REAL;
BEGIN
IF I=0 THEN FAKT:=1
ELSE FAKT:=FAKT(I-1)*I;
END;
```

Ada:

Minden programegységet ismer. Más nyelveknél a külön fordított egységek fizikailag önállóak, semmit nem tudnak egymásról, a kapcsolatszerkesztő dolga, hogy összeállítsa belőlük a programot. Az Ada fordító olyan, hogy fordítás közben van konzisztenciaellenőrzés. Nincs külön főprogram, implementációfüggő, hogy melyik alprogram indul el először.

Az eljárás alakja:

```
PROCEDURE név[(formális_paraméter_lista)]
IS
deklarációk
BEGIN
végrehajtható_utasítások
END [név];
```

A függvény alakja:

```
FUNCTION név[(formális_paraméter_lista)] RETURN típus
IS
deklarációk
BEGIN
végrehajtható_utasítások
END [név];
```

Az eljáráshívásra nincs külön alapszó. Az eljárás szabályosan befejeződik RETURN-utasítás hatására, vagy ha elérjük az `END`-et. A függvény befejeződik a `RETURN kifejezés;` utasítás hatására.

A formális paraméter lista paramétercsoportokból áll, amelyeket pontosvessző választ el. Egy paramétercsoport szerkezete a következő:

`név[,név]... : [mód] típus [:= kifejezés]`

A `mód` a paraméterátadás módját dönti el. Három alapszó szerepelhet itt: `IN`, `OUT`, `IN OUT`.

`IN` esetén a paraméterátadás érték szerinti, `OUT` esetén eredmény szerinti, `IN OUT` esetén pedig érték-eredmény szerinti. Ha a `mód` elmarad, akkor az alapértelmezés `IN`. `IN` módú paraméterek esetén szerepelhet a `kifejezés`, amely a paraméter inicializálásának szerepét játssza. A függvény paramétere csak `IN` módú lehet, tehát a függvény a paraméterét nem tudja megváltoztatni. De az Adában is van a függvénynek mellékhatása, mert a környezetét meg tudja változtatni.

A paraméterkiértékelésnél szigorú típusegyeztetés van. Alapértelmezett a sorrendi kötés, de lehetőség van a név szerinti kötésre is `formális_paraméter_név => aktuális_paraméter` alakban, az aktuális paraméter listán. Azon formális paraméterekhez, amelyekhez kifejezéssel kezdőértéket rendeltünk, nem kötelező aktuális paramétert megadni. Ilyenkor, ha megadtunk aktuális paramétert, akkor azzal, ha nem, akkor a kezdőértékkel kezd el működni az alprogram.

Példa eljárásspecifikációra:

```
PROCEDURE xy(C: IN INTEGER RANGE 1..89; D: IN INTEGER:=0)
```

Ezt az eljárást többféleképpen meghívhatjuk:

```
xy(2,9);
xy(2);
xy(D => 9, C => 2);
```

Az első és harmadik esetben `C=2` és `D=9`, a másodikban `C=2` és `D=0`, az inicializálás miatt. A harmadik esetben név szerinti kötés van, explicit módon megadjuk, hogy melyik formális paraméterhez melyik értéket rendeljük hozzá.

A rekurzió alapértelmezés.

A blokk alakja:

```
[blokknév:]
[DECLARE
deklarációk]
BEGIN
végrehajtható_utasítások
END [blokknév];
```

Ha `blokknév` szerepel a blokk előtt, akkor megadása kötelező az `END` után is.

A változók élettartama alapértelmezés szerint dinamikus.

A hatáskörkezelés statikus, de az Ada kezeli a „lyuk a hatáskörben” problémát. Ezt úgy oldja meg, hogy a globális eszközök nevét azon programegység nevével minősíti, amely programegységnek az lokális neve.

A faktoriálist kiszámító függvény:

```
FUNCTION FAKT(I : INTEGER) RETURN REAL IS
F : REAL:=1.0;
BEGIN
IF I>1  THEN F:=FAKT(I-1) * FLOAT(I);
END IF;
RETURN(F);
END FAKT;
```

C:

A C nyelv a függvényt és a blokkot ismeri. A függvények nem ágyazhatók be más programegységbe, a blokkok tetszőleges mélységben skatulyázhatók.

A blokk alakja:

```
{
deklarációk
végrehajtható_utasítások
}
```

A függvény alakja:

```
[típus] név([formális_paraméter_lista])
blokk
```

Ha nem szerepel a `típus`, akkor az alapértelmezés `int`. Ha `void` a `típus`, akkor lényegében egy eljárásról van szó. A főprogram is egy függvény, melynek neve `main()`.

Függvény befejeződhet az alábbi módokon:

- `RETURN kifejezés;` : ebben az esetben a `kifejezés` értéke lesz a függvény visszatérési értéke.
- `RETURN` : ha `void` típusú a függvény, akkor nem ad vissza értéket, egyébként határozatlan értékkel tér vissza.
- Ha eléri a záró **}**-t, ekkor is határozatlan értéket ad vissza.

A rekurzió alapértelmezés.

A formális paraméter listán a típust adhatjuk meg. A formális paramétereket vessző választja el. A C-ben a programozó tud nem fix paraméterszámú függvényt deklarálni úgy, hogy megad legalább egy formális paramétert és a formális paraméter listát … zárja. Az üres formális paraméter listát explicit módon jelölhetjük a `void` alapszó megadásával.

A paraméterkiértékelésnél sorrendi kötés, típuskényszerítés és fix paraméterszám esetén számbeli egyeztetés van.

A paraméterátadás érték szerinti.

A C-ben a fordítási egység a *forrásállomány*. Ez ún. *külső* deklarációkat (nevesített konstans, változó, típus, függvény) tartalmaz.

Egy forrásállományban más forrásállományokra az `#include` előfordítói utasítással hivatkozhatunk. Ez bemásolja a hivatkozás helyére a hivatkozott forrásállomány szövegét.

A C a hatáskör és élettartam szabályozására bevezeti a tárolási osztály attribútumokat, melyek a következők:

- `extern`: A fordítási egység szintjén deklarált nevek alapértelmezett tárolási osztálya, lokális neveknél explicit módon meg kell adni. Az ilyen nevek hatásköre a teljes program, élettartamuk a program futási ideje. Van automatikus kezdőértékük.
- `auto`: A lokális nevek alapértelmezett tárolási osztálya. Hatáskörkezelésük statikus, de csak a deklarációtól kezdve láthatók. Élettartamuk dinamikus. Nincs automatikus kezdőértékük.
- `register`: Speciális `auto`, amelynek értéke regiszterben tárolódik, ha van szabad regiszter, egyébként nincs különbség. A címkomponensére nem hivatkozhatunk.
- `static`: Bármely névnél explicit módon meg kell adni. Külső statikus név hatásköre a fordítási egység, lokális statikus név hatáskörkezelése statikus, de csak a deklarációtól kezdve látható. Élettartamuk a program futási ideje. Van automatikus kezdőértékük.

A faktoriálist kiszámító függvény C-ben:

```
long fakt(long n)
{
if (n<=1) return 1;
else return n*fakt(n-1);
}
```

6. ABSZTRAKT ADATTÍPUS

Az absztrakt adattípus olyan adattípus, amely megvalósítja a *bezárást* vagy *információ rejtést*. Ez azt jelenti, hogy ezen adattípusnál nem ismerjük a reprezentációt és a műveletek implementációját. Az adattípus ezeket nem mutatja meg a külvilág számára. Az ilyen típusú programozási eszközök értékeihez csak szabályozott módon, a műveleteinek specifikációi által meghatározott *interfészen* keresztül férhetünk hozzá. Tehát az értékeket véletlenül vagy szándékosan nem ronthatjuk el. Ez nagyon lényeges a *biztonságos programozás* szempontjából. Az absztrakt adattípus (angol rövidítéssel: ADT – Abstract Data Type) az elmúlt évtizedekben a programnyelvek egyik legfontosabb fogalmává vált és alapvetően befolyásolta a nyelvek fejlődését.

7. A CSOMAG

A csomag az a programegység, amely egyaránt szolgálja a procedurális és az adatabsztrakciót.

A procedurális absztrakció oldaláról tekintve a csomag programozási eszközök újrafelhasználható gyűjteménye. Ezek az eszközök:

- típus
- változó
- nevesített konstans
- kivétel
- alprogram
- csomag

Ezek az eszközök a csomag hatáskörén belül mindenhonnan tetszőlegesen hivatkozhatók.

A csomag mint programegység megvalósítja a bezárást, ezért alkalmas absztrakt adattípus implementálására.

A csomag az Adában jelenik meg. Az Ada csomagnak két része van: *specifikáció* és *törzs*.

Formálisan a specifikáció a következőképpen néz ki:

```
PACKAGE név IS
látható_deklarációs_rész
[PRIVATE
privát_deklarációs_rész]
END [név];
```

A `látható_deklarációs_rész` a csomagspecifikáció látható része. Az itt deklarált programozási eszközök hivatkozhatók a csomagon kívülről. Alprogramok esetén itt csak azok specifikációja állhat. A hivatkozás a csomag nevével való minősítéssel történik.

A `privát_deklarációs_ rész` kívülről nem elérhető, a csomag az itt deklarált eszközöket bezárja, elrejti a külvilág elől.

A csomag törzse opcionális. Ha viszont a specifikációban szerepel alprogramspecifikáció, akkor kötelező a törzs, és az alprogram teljes deklarációját itt kell megadni. A törzs a külvilág számára nem elérhető.

A csomag törzsének alakja:

```
PACKAGE BODY név IS
deklarációs_rész
[ BEGIN
végrehajtható_utasítások]
END [név] ;
```

Az Adában a csomag fordítható önállóan, vagy elhelyezhető lokákisan egy másik programegység deklarációs részében. Az utóbbi esetben a láthatóságát a statikus a hatáskörkezelésnek megfelelően a tartalmazó programegység szabályozza. Ha önállóan fordítjuk a csomagot, akkor a program más részei számára explicit módon kell láthatóvá tenni (l. 8.2. alfejezet).

A következő példa egy olyan csomag vázlatát mutatja be, amely a törtekkel való számolást teszi lehetővé.

```
package RACIONALIS_SZAM is
      type RACIONALIS is
      record
            SZAMLALO : integer;
            NEVEZO   : integer range 1..MAX_INTEGER;
      end record;
      function ”=” ( X,Y : RACIONALIS) return boolean;
      function ”+” ( X,Y : RACIONALIS) return RACIONALIS;
      function ”-” ( X,Y : RACIONALIS) return RACIONALIS;
      function ”*” ( X,Y : RACIONALIS) return RACIONALIS;
      function ”/” ( X,Y : RACIONALIS) return RACIONALIS;
end;
```

```
package body RACIONALIS_SZAM is
      procedure KOZOS_NEVEZO (X,Y: in out RACIONALIS) is ...
      function ”=” ...
      function ”+” ...
      function ”-” ...
      function ”*” ...
      function ”/” ...
end RACIONALIS_SZAM;
```

A csomag specifikációjának csak látható része van. Ebben szerepel egy saját típus definíció (`RACIONALIS`), amely a törtek értékeinek kezelésére való. A törtszámok reprezentációja egy rekord segítségével történik, a számlálót és a nevezőt egy-egy mezőben tároljuk. A reprezentáció meghatározza a tartományt is (a nevező pl. csak pozitív egész lehet). Van továbbá öt függvényspecifikáció, ezek adják meg a műveletek specifikációját.

Érdekességként jegyezzük meg, hogy az Ada lehetővé teszi, hogy a függvény neve ne csak azonosító legyen. A beépített operátorok túlterhelhetőek. Az Adában ugyanis idézőjelek közé tett speciális karaktereket is használhatunk a függvénynév megadásához. Példánkban a szokásos aritmetikai operátorokat terheltük túl.

A csomag specifikációjában van alprogramspecifikáció, tehát kötelező a törzs. Abban a teljes alprogram deklarációkat meg kell adni. A műveleteket úgy kell implementálni, hogy azok a megfelelő törtaritmetikára jellemző viselkedésmódot tükrözzék. Ezért szükséges a kívülről nem elérhető `KOZOS_NEVEZO` eljárás.

A fenti példánkban a `RACIONALIS` típus nem absztrakt adattípus. Műveleteinek implementációját ugyan elrejti, de a reprezentációját nem. A programozó önfegyelmére van bízva, hogy az ilyen típusú eszközök értékeit csak a csomagbeli függvények segítségével kezeli, vagy pedig a reprezentáló rekord mezőire külön-külön hivatkozik. Például a `SZAMLALO`-hoz minden további nélkül hozzá tudunk adni egy számot, a `NEVEZO`-től függetlenül.

Nézzük ezután, hogy az Ada csomagja hogyan szolgálja az adatabsztrakciót.

Az Adában a bezárást a privát (`PRIVATE`) típus és a csomag specifikációs részének kívülről nem látható része együttesen teszi lehetővé. A látható részben szerepelhet a privát

típusmegjelölés. A privát típusúnak deklarált objektumokra a nyelvbe beépített műveletek közül csak az egyenlőségvizsgálat (=) , és az értékadás (:=) alkalmazható. Tehát az így deklarált eszközök értékeit kezelő műveleteket a programozónak kell implementálnia. A privát típus reprezentációjáról nem tudunk semmit.

Ha van a látható részben privát típussal deklarált eszköz, akkor kötelező a nem látható rész szerepeltetése. Itt kell megadni a reprezentációra vonatkozó összes információt.

Korábbi példánkban a RACIONaLIS típusból csináljunk absztrakt adattípust. Ehhez a törzs változatlanul hagyása mellett a specifikációt kell átalakítanunk a következő módon:

```
package RACIONALIS_SZAM is
      type RACIONALIS is private;
      function ”=” ( X,Y : RACIONALIS) return boolean;
      function ”+” ( X,Y : RACIONALIS) return RACIONALIS;
      function ”-” ( X,Y : RACIONALIS) return RACIONALIS;
      function ”*” ( X,Y : RACIONALIS) return RACIONALIS;
      function ”/” ( X,Y : RACIONALIS) return RACIONALIS;
private
      type RACIONALIS is
      record
            SZAMLALO : integer;
            NEVEZO   : integer range 1..MAX_INTEGER;
      end record;
end;
```

A reprezentáció tehát átkerült a nem látható részbe, megvalósult a bezárás. Így most már csak a szabályos hozzáférés és műveletvégzés lehetséges.

Az Adában a privát típusnak létezik egy olyan változata, a korlátozott privát (LIMITED PRIVATE) típus, amelyre még az egyenlőségvizsgálat és az értékátadás beépített műveletek sem alkalmazhatók. Ennek használatánál tehát ezen műveleteket is a programozónak kell implementálnia.

A korlátozott privát típus alkalmazásával csomagunk vázlata a következőképpen alakul:

```
package RACIONALIS_SZAM is
      type RACIONALIS is limited private;
      function ”=” ( X,Y : RACIONALIS) return boolean;
      procedure ERTEKMASOLAS ( X : out RACIONALIS; Y : RACIONALIS);
      function ”+” ( X,Y : RACIONALIS) return RACIONALIS;
      function ”-” ( X,Y : RACIONALIS) return RACIONALIS;
      function ”*” ( X,Y : RACIONALIS) return RACIONALIS;
      function ”/” ( X,Y : RACIONALIS) return RACIONALIS;
private
      type RACIONALIS is
      record
            SZAMLALO : integer;
            NEVEZO   : integer range 1..MAX_INTEGER;
      end record;
end;

package body RACIONALIS_SZAM is
      procedure KOZOS_NEVEZO (X,Y: in out RACIONALIS) is ...
      function ”=” ...
      procedure ERTEKMASOLAS ...
      function ”+” ...
      function ”-” ...
      function ”*” ...
      function ”/” ...
end RACIONALIS_SZAM;
```

Ha egy csomagspecifikáció látható részében változókat deklarálunk, akkor azok a változók (amennyiben felhasználjuk őket a csomagban deklarált alprogramokban) `OWN` típusú változók lesznek, amelyeknek az a tulajdonsága, hogy két alprogramhívás közt megtartják értéküket (a változó befejezéskori értékét tehát felhasználhatom a következő alprogramhíváskor). Ezzel további kommunikációs lehetőséget biztosít az Ada az alprogramok között.

Itt jegyezzük meg, hogy a Turbo Pascal unitja csomag. Alakja:

```
UNIT név;
INTERFACE
     látható_deklarációk
IMPLEMENTATION
     nem_látható_deklarációk
BEGIN
     végrehajtható_utasítások
END.
```

8. AZ ADA FORDÍTÁSRÓL

8.1. Pragmák

A *pragmák* a program szövegében elhelyezkedő olyan utasítások, amelyek a fordító működését befolyásolják. A fordítóprogramnak szólnak, szolgáltatást kérnek tőle, valamilyen üzemmódot állítanak be, nem áll mögöttük közvetlen kód, de befolyásolhatják a kódot. A pragmák egy része a program szövegének bármely pontján elhelyezhető, másik része csak kötött helyen használható.

Egy pragma szerkezete a következő:

```
PRAGMA név [(paraméter_lista)];
```

Egy paraméter felépítése:

```
[ név => ] {azonosító | kifejezés }
```

A pragmák azon eszközrendszerek közé tartoznak, amelyeket az Ada hivatkozási nyelv csak részben szabályoz azaz az implementációk megvalósításai eltérhetnek egymástól. Az Ada rendszerekben általában mintegy 50 féle pragma van.

Lássunk közülük néhányat:

```
INTERFACE(programnyelv_neve, alprogram_név)
```

Az adott alprogram specifikációja után kell megadni, és azt jelzi, hogy az adott alprogram törzse az adott nyelven van megírva.

```
LIST ({ ON | OFF })
```

Fordítás közben a programszövegről lista készül a szabvány kimeneten (`ON`), vagy letiltjuk a listázást (`OFF`). Bárhol elhelyezhető.

Ha a fordító nem ismeri föl a pragma nevét, akkor ignorálja azt.

8.2. Fordítási egységek

Az Adában fordítási egység lehet:

- alprogram specifikáció
- alprogram törzs
- csomag specifikáció
- csomag törzs
- fordítási alegység
- valamint ezek tetszőleges kombinációja

Azokat a fordítási egységeket, amelyek nem függnek más fordítási egységtől (nem alegységei más fordítási egységnek), *könyvtári egységnek* hívja az Ada. Ilyen könyvtári egységet a programozó tetszőleges számban hozhat létre. A könyvtári egységeknek külön egyedi nevük van.

Ha valamely fordítási egységben használni akarok egy olyan eszközt, amely egy másik fordítási egységben van benne, akkor az adott eszköz specifikációjának lefordítva kell lennie, tehát előbb kell lefordítani, mint azt, amiben hivatkozunk rá. Tehát a főprogramot kell utoljára megírni. Az Ada fordító fordítási időben konzisztencia ellenőrzést is végez.

Ha a specifikáció módosul, újra kell fordítani azt a fordítási egységet, amelyben a specifikáció van, és azokat, amelyek hivatkoznak erre a módosult specifikációra. Ha csak az implementáció változik, csak az azt tartalmazó fordítási egységet kell újrafordítanunk. Ennek segítségével nagy programok fejlesztése mehet párhuzamosan, és a program módosítása is egyszerűbb, valamint nő a biztonságos programírás lehetősége.

Minden fordítási egység kezdete előtt meg kell adni egy ún. *környezeti előírást*. Ennek megadása a `WITH` utasítással történik, melyben azon könyvtári egységeket soroljuk fel, amelyekre az adott fordítási egységben hivatkozunk, amelynek eszközeit felhasználjuk a fordítási egységben.

Alakja:

```
WITH könyvtári_egység [, könyvtári_egység ]…;
```

Az így megadott könyvtári egységek alkotják az adott fordítási egység környezetét. Ez a fordítási egység ezen könyvtári egységek elemeit látja.

Az Adában 9 szabvány könyvtári egység van, ezek alkotják magát az Ada rendszert. Ezeket is szerepeltetni kell a környezeti előírásban.

Van viszont a `STANDARD` nevű könyvtári egység, amelyet a fordítóprogram minden fordítási egységhez automatikusan hozzáilleszt, ezt nem kell külön megadni. Ez tartalmazza az alapvető nyelvi eszközöket (pl. karakterkészlet, beépített típusok, stb.), melyek nélkül nem írható program.

Azokat a fordítási egységeket hívjuk fordítási alegységnek, amelyek önállóan nem léteznek, hanem egy másik fordítási egységhez kapcsolódnak.

Az Ada lehetővé teszi, hogy akármelyik szinten beágyazott alprogram, csomag, taszk törzsét ne a specifikációhoz kapcsolva adjuk meg, hanem ott csak jelezzük egy *csonk* segítségével, hogy a törzs egy másik fordítási egységben mint fordítási alegység lesz lefordítva. Egy fordítási alegység környezetét a csonk határozza meg, a fordítási alegységben a csonkot tartalmazó programegység nevére kell hivatkozni. A csonkot tartalmazó fordítási egységet mindig előbb kell lefordítani, mint a kapcsolódó fordítási alegységet. Tetszőleges egymáshoz kapcsolódó fordítási alegység sorozatot lehet létrehozni.

A csonkot a törzs helyett elhelyezett `SEPARATE` alapszó jelzi. Fordítási alegység elején a csonkot a `SEPARATE(név)` formában kell hivatkozni.

A következőkben fordítási egységekre látunk példákat:

```
-- egyetlen fordítási egység
procedure FELDOLGOZO is
      package D is
            HATAR : constant:=1000;
            TABLA : array (1..HATAR) of integer;
            procedure RESTART;
      end;
```

```
     package body D is
          procedure RESTART is
               begin
                    for N in 1..HATAR loop
                         TABLA(N):=N;
                    end loop;
               end;
    begin
        RESTART;
    end D;
    procedure Q(X:integer) is
    begin
                  ...
          D.TABLA(X):= D.TABLA(X)+1;
                  ...
    end Q;
begin
                 ...
    D.RESTART;
                 ...
end FELDOLGOZO;
```

Itt egy eljárást látunk, ezen belül van egy csomag és egy másik eljárás. Fordítás után egy `FELDOLGOZO` nevű könyvtári egység keletkezik.

Ez a programszöveg most egy fordítási egységet alkot, de feldarabolható három külön fordítási egységre a következőképpen:

```
-- első fordítási egység
package D is
      HATAR : constant:=1000;
      TABLA : array (1..HATAR) of integer;
      procedure RESTART;
end D;

-- második fordítási egység
```

```
package body D is
       procedure RESTART is
         begin
           for N in 1..HATAR loop
                 TABLA(N):=N;
           end loop;
         end;
begin
         RESTART;
end D;

-- harmadik fordítási egység
with D;
procedure FELDOLGOZO is
      procedure Q(X:integer) is
           begin
                      ...
                D.TABLA(X):= D.TABLA(X)+1;
                      ...
           end Q;
begin
                  ...
    D.RESTART;
                  ...
end FELDOLGOZO;
```

Miután a hivatkozott eszközök specifikációjának már lefordítottnak kell lenniük, ezért először fordítandó az első fordítási egység, majd a második és harmadik, tetszőleges sorrendben.

Példa fordítási alegységekre:

```
-- tartalmazó eljárás
procedure T is
    type REAL is digits 10;
    R,S : REAL:=1.0;
```

```
    package D is
       PI: constant:=3.14159;
       function F(X:REAL) return REAL;
       procedure G(X,Z:REAL);
    end;
    package body D is separate;  -- csonk
    procedure Q(U:in out REAL) is separate;  -- csonk
begin
    ...
    Q(R);
    ...
    D.G(R,S);
    ...
end T;


-- fordítási alegység
separate(T) -- hivatkozás a csonkra
    procedure Q(U : in out REAL) is
    begin
       ...
    end Q;


-- fordítási alegység
separate(T) -- hivatkozás a csonkra
    package body D is
          ...
       function F(X:REAL) return REAL is separate; -- csonk
       procedure G(Y,Z : real) is separate;  -- csonk
          ...
    end D;


-- fordítási alegység
separate(T.D) -- hivatkozás a fordítási alegységbeli csonkra
```

```
function F(X:REAL) return REAL is
    . . .
end F;
procedure G(Y,Z:REAL) is
    . . .
end G;
```

9. KIVÉTELKEZELÉS

A kivételkezelési eszközrendszer azt teszi lehetővé, hogy az operációs rendszertől átvegyük a megszakítások kezelését, felhozzuk azt a program szintjére. A *kivételek* olyan események, amelyek megszakítást okoznak. A *kivételkezelés* az a tevékenység, amelyet a program végez, ha egy kivétel következik be. *Kivételkezelő* alatt egy olyan programrészt fogunk érteni, amely működésbe lép egy adott kivétel bekövetkezte után, reagálva az eseményre.

A kivételkezelés az eseményvezérlés lehetőségét teszi lehetővé a programozásban.

Operációs rendszer szinten lehetőség van bizonyos megszakítások maszkolására. Ez a lehetőség megvan nyelvi szinten is. Egyes *kivételek figyelése letiltható* vagy *engedélyezhető*. Egy kivétel figyelésének letiltása a legegyszerűbb kivételkezelés. Ekkor az esemény hatására a megszakítás bekövetkezik, feljön programszintre, kiváltódik a kivétel, de a program nem vesz róla tudomást, fut tovább. Természetesen nem tudjuk, hogy ennek milyen hatása lesz a program további működésére, lehet, hogy az rosszul, vagy sehogy sem tudja folytatni munkáját.

A kivételeknek általában van *neve* (vagy egy kapcsolódó sztring, amely gyakran az eseményhez kapcsolódó üzenet szerepét játssza) és *kódja* (ami általában egy egész szám).

A kivételkezelés a PL/I-ben jelenik meg és az Ada is rendelkezik vele. A két nyelv kétfajta kivételkezelési filozófiát vall. A PL/I azt mondja, hogy ha egy program futása folyamán bekövetkezik egy kivétel, akkor az azért van, mert a program által realizált algoritmust nem készítettük föl az adott esemény kezelésére, olyan szituáció következett be, amelyre speciális módon kell reagálni. Ekkor keressük meg az esemény bekövetkeztének az okát, szüntessük meg a speciális szituációt és térjünk vissza a program normál működéséhez, folytassuk a programot ott, ahol a kivétel kiváltódott.

Az Ada szerint viszont, ha bekövetkezik a speciális szituáció, akkor hagyjuk ott az eredeti tevékenységet, végezzünk olyan tevékenységet, ami adekvát a bekövetkezett eseménnyel és ne térjünk vissza oda, ahol a kivétel kiváltódott.

A kivételkezelési eszközrendszerrel kapcsolatban a nyelveknek a következő kérdéseket kell megválaszolni:

1. *Milyen beépített kivételek vannak a nyelvben?*
2. *Definiálhat-e a programozó saját kivételt?*
3. *Milyenek a kivételkezelő hatásköri szabályai*?
4. *A kivételkezelés köthető-e programelemekhez (kifejezés, utasítás, programegység)?*
5. *Hogyan folytatódik a program a kivételkezelés után?*
6. *Mi történik, ha kivételkezelőben következik be kivétel?*
7. *Van-e a nyelvben beépített kivételkezelő?*
8. *Van-e lehetőség arra, hogy bármely kivételt kezelő (általános) kivételkezelőt írjunk?*
9. *Lehet-e parametrizálni a kivételkezelőt?*

Sem a PL/I-ben, sem az Adában nincs parametrizált és beépített kivételkezelő, a részleteket illetően pedig az alábbiakat mondják.

9.1. A PL/I kivételkezelése

A PL/I beépített kivételei a következők:

`CONVERSION`	konverziós hiba
`FIXEDOVERFLOW`	fixpontos túlcsordulás
`OVERFLOW`	lebegőpontos túlcsordulás
`UNDERFLOW`	lebegőpontos alulcsordulás
`ZERODIVIDE`	nullával való osztás
`SIZE`	mérethiba
`SUBSCRIPTRANGE`	indextúllépés
`STRINGRANGE`	
`STRINGSIZE`	
`CHECK[(azonosító)]`	nyomkövetés eszköze
`AREA`	címzési hiba
`ATTENTION`	külső megszakítás
`FINISH`	a program szabályos befejeződése

`ENDFILE(áll_név)`	állomány vége
`ENDPAGE(áll_név)`	lap vége
`KEY(áll_név)`	kulcshiba
`NAME(áll_név)`	
`RECORD(áll_név)`	
`TRANSMIT(áll_név)`	
`UNDEFINEDFILE(áll_név)`	
`PENDING(áll_név)`	
`ERROR`	általános kivétel

A programozó saját kivételt a

```
CONDITION(név)
```

formában tud deklarálni.

Az első öt beépített kivétel figyelése alapértelmezésben engedélyezett, de letiltható, a második öté letiltott, de engedélyezhető, a többié, és a programozói kivételeké mindig engedélyezett és soha nem tiltható le.

Minden utasítás előtt szerepelhet egy, a kivétel figyelésének letiltására vagy engedélyezésére vonatkozó előírás. Ha kerek zárójelek között megadunk tetszőleges számú kivételnevet, vesszővel elválasztva

```
(kivételnév [, kivételnév]…):utasítás
```

akkor ez az adott kivételek figyelésének engedélyezését jelenti.

Ha a kivételnév előtt szerepel a `NO`, akkor az adott kivétel letiltását írtuk elő.

Például:

```
(NOZERODIVIDE, SIZE):IF ...
```

Ha egy blokk vagy alprogram kezdő utasítása előtt szerepel az előírás, akkor az az adott teljes programegységre vonatkozik (melyen belül utasításonként felülbírálható), beleértve a tartalmazott programegységeket is. Ha olyan utasítás előtt áll, amelyben van kifejezés, akkor

csak a kifejezésre vonatkozik, nem pedig a teljes utasításra. Ha nincs kifejezés, akkor a teljes utasításra vonatkozik.

Egy kivétel explicit kiváltására a

```
SIGNAL kivételnév;
```

utasítás szolgál. Programozói kivétel csak így váltható ki.

A kivételkezelő alakja:

```
ON kivételnév végrehajtható_utasítás;
```

Természetesen a végrehajtható utasítás helyén állhat blokk.

A program szövegében bárhol elhelyezhető kivételkezelő.

A kivételkezelő hatásköre egy adott programegységben attól az időponttól kezdődik, amikor a vezérlés áthaladt rajta, és tart

- egy másik ugyanerre a névre kiadott kivételkezelőig (mely fölülírja az előző hatását),
- ugyanerre a névre kiadott `REVERT kivételnév;` utasításig (mely érvényteleníti a legutolsónak kiadott ON-utasítás hatását),
- vagy a programegység befejeződéséig,

beleértve a kivételkezelő hatáskörén belül meghívott minden egyes programegységet is.

A kivételkezelő hatásköre tehát dinamikus.

Ha egy programegységben bekövetkezik egy kivétel, akkor a futtató rendszer megnézi, hogy az adott kivétel figyelése engedélyezett-e vagy sem. Ha letiltott, akkor folytatódik tovább a programegység végrehajtása. Ha a kivétel figyelése engedélyezett, akkor megnézi a futtató rendszer, hogy ezen a ponton van-e olyan látható kivételkezelő, amely az adott kivétel nevét tartalmazza. Ha van ilyen, akkor lefut a kivételkezelő. Ha a kivételkezelőben van GOTO-utasítás, akkor a megadott címkéjű utasításon folytatódik a program. Ha nem szerepel GOTO-utasítás, akkor vagy azon az utasításra kerül vissza a vezérlés, amelyben bekövetkezett a kivétel, vagy a következő utasításra. Ez a kivételtől függ. Például `CONVERSION` esetén a kivételkezelés után újra ugyanaz az utasítás kerül végrehajtásra, amely kiváltotta ezt a

kivételt. Aritmetikai hibák esetén, illetve saját kivételnél pedig a kivételt kiváltó utasítást követő utasításon folytatódik a program futása.

Ha az adott programegységben nincs hatásos nevesített kivételkezelő, akkor visszalép a hívási láncon és a hívóban keres ilyet. Ha a hívási láncon visszafelé lépkedve sehol sem talál ilyet, akkor bekövetkezik az `ERROR` kivétel, és ezután erre próbál látható hatásos kivételkezelőt találni. Az ON ERROR kivételkezelő kezeli le az összes nem nevesített kivételt, ez tehát az általános kivételkezelő a PL/I-ben. Ha ilyen kivételkezelőt nem talál, akkor a vezérlés átkerül az operációs rendszerhez, a program nem kezelte az adott kivételt.

A kivételkezelőben bekövetkező kivételt a PL/I ugyanígy kezeli.

A PL/I rendelkezik a kivételkezelést elősegítő beépített, paraméter nélküli, csak kivételkezelőben hívható függvényekkel, az ún. ON-függvényekkel. Ezek segítenek behatárolni a kivételt kiváltó pontos eseményt, annak helyét, esetleg okát. Röviden áttekintünk közülük néhányat:

- `ONCODE`: A hiba kódját adja meg. Több olyan beépített kivétel (pl. `KEY`) van, amely egy eseménycsoportot nevez meg. Ekkor az egyedi eseményt csak a kódja alapján azonosíthatjuk.
- `ONCHAR`: Konverziós hibáknál, folyamatos módú átvitelnél megadja azt a karaktert, amely a hibát okozta. Ez a beépített függvény pszeudóváltozóként használható, azaz érték adható neki. Tehát kicserélhető a konverziós hibát okozó karakter, és újra lehet próbálkozni az I/O-val.
- `ONKEY`: I/O hibánál a hibát okozó rekord elsődleges kulcsát adja meg.
- `ONLOCK`: Azon alprogram nevével tér vissza, amelyben a kivétel bekövetkezett.

A kivételkezelő dinamikus hatáskörkezelésből problémák adódhatnak. A nevek hatáskörkezelése statikus, a kivételkezelőé dinamikus, ez ellentmondáshoz vezethet. A meghívott programegység örökli azon kivételkezelő hatását, amely a hívó programegységben hatásossá válik. Ez veszélyes lehet, mert nemlokális ugrásokat eredményezhet. Ha egy programegységben nem kezeljük az ott bekövetkezett kivételt, akkor lehet, hogy az egy, a

hívási láncban jóval korábban elhelyezkedő programegység olyan kivételkezelőjét aktivizálja, amely teljesen hibás reagálást eredményez.

A PL/I-ben a saját kivételek nagyon jól használhatók a belövésnél, de nem igazán hatékonyak futás közben.

Példa:

Egy szekvenciális állomány feldolgozásának mintája a PL/I-ben a következő lehet:

```
DECLARE F FILE;
1 S,
2 AZON PICTURE '9999',
3 EGYEB CHARACTER(91),
EOF BIT(1) INIT('0'B);
ON ENDFILE(F) EOF='1'B;
OPEN FILE(F);
READ FILE(F) INTO(S);
DO WHILE(¬EOF);
.
.
.
READ FILE(F) INTO(S);
END;
```

9.2. Az Ada kivételkezelése

Az Ada beépített kivételei általában eseménycsoportot neveznek meg. Ezek a következők:

- `CONSTRAINT_ERROR:` Olyan eseménycsoport, amely akkor következik be, ha valamilyen deklarációs korlátozást megpróbálunk túllépni. Például indexhatár átlépése.
- `NUMERIC_ERROR:` Aritmetikai hibák, alul- ill. túlcsordulás, 0-val való osztás, stb.
- `STORAGE_ERROR:` Tárhiba (minden allokálási probléma ide tartozik): a tárrész, amelyre hivatkoztunk nem áll rendelkezésre.

- TASKING_ERROR: Az adott taszkkal nem jöhet létre randevú.
- SELECT_ERROR: SELECT-utasítás hiba.

Alaphelyzetben minden kivétel figyelése engedélyezett, de egyes események figyelése (bizonyos ellenőrzések) letiltható. Erre egy pragma szolgál, melynek alakja:

```
SUPPRESS(név [,ON => { eszköznév |típus }] )
```

A név a letiltandó esemény neve. Egyetlen eseményt azonosít, nem egyezik meg a beépített kivételnevekkel. Lehetséges értékei:

- ACCESS_CHECK: címzés ellenőrzés
- DISCRIMINANT_CHECK: rekord diszkriminánsának ellenőrzése
- INDEX_CHECK: index ellenőrzés
- LENGTH_CHECK: hossz ellenőrzés
- RANGE_CHECK: tartomány ellenőrzés
- DIVISION_CHECK: nullával való osztás ellenőrzése
- OVERFLOW_CHECK: túlcsordulás ellenőrzés
- STORAGE_CHECK: tárhely rendelkezésre állásának ellenőrzése

Az eszköznév valamely programozói eszköz (pl. változó) nevét jelenti. Ha az opcionális rész nem létezik, akkor a teljes programra vonatkozik a letiltás, ha igen, akkor az ott megadott típusra, vagy az adott nevű eszközre.

Saját kivétel az EXCEPTION attribútummal deklarálható.

Kivételkezelő minden programegység törzsének végén, közvetlenül a záró END előtt helyezhető el, alakja:

```
EXCEPTION
      WHEN kivételnév [, kivételnév ]... => utasítások
      [ WHEN kivételnév [, kivételnév ]... => utasítások ]...
      [ WHEN OTHERS => utasítások ]
```

Az `utasítások` rész tetszőleges számú és fajtájú végrehajtható utasításból állhat. WHEN-ágból tetszőleges számú megadható, de legalább egy kötelező. WHEN OTHERS ág viszont legfeljebb egyszer szerepelhet, és utolsóként kell megadni. Ez a nem nevesített kivételek kezelésére való (általános kivételkezelés).

A kivételkezelő a teljes programegységben, továbbá az abból meghívott programegységekben látszik, ha azokban nem szerepel saját kivételkezelő. Tehát a kivételkezelő hatásköre az Adában is dinamikus, az a hívási láncon öröklődik.

Bármely kivételt explicit módon kiváltani a

```
RAISE kivételnév;
```

utasítással lehet. Programozói kivétel kiváltása csak így lehetséges.

Ha egy programegységben kiváltódik egy kivétel, akkor a futtató rendszer megvizsgálja, hogy az adott kivétel figyelése le van-e tiltva. Ha igen, akkor a program fut tovább, különben a programegység befejezi működését. Ezek után a futtató rendszer megnézi, hogy az adott programegységen belül van-e kivételkezelő. Ha van, akkor megnézi, hogy annak van-e olyan WHEN-ága, amelyben szerepel az adott kivétel neve. Ha van ilyen ág, akkor végrehajtja az ott megadott utasításokat. Ha ezen utasítások között szerepel a GOTO-utasítás, akkor a megadott címkén folytatódik a program. Ha nincs GOTO, akkor úgy folytatódik a program futása, mintha a programegység szabályosan fejeződött volna be. Ha a kivétel nincs nevesítve, megnézi, hogy van-e WHEN OTHERS ág. Ha van, akkor az ott megadott utasítások hajtódnak végre és a program ugyanúgy folytatódik mint az előbb. Ha nincs nevesítve a kivétel egyetlen ágban sem és nincs WHEN OTHERS ág, vagy egyáltalán nincs kivételkezelő, akkor az adott programegység *továbbadja* a kivételt. Ez azt jelenti, hogy a kivétel kiváltódik a hívás helyén, és a fenti folyamat ott kezdődik előlről. Tehát a hívási láncon visszafelé lépkedve keres megfelelő kivételkezelőt. Ha a hívási lánc elejére ér és ott sem talál kivételkezelőt, akkor a program a kivételt nem kezelte és a vezérlés átadódik az operációs rendszernek.

Kivételkezelőben kiváltott kivétel azonnal továbbadódik.

Csak a kivételkezelőben alkalmazható a

```
RAISE;
```

utasítás, amely újra kiváltja azt a kivételt, amely aktivizálta a kivételkezelőt. Ez viszont az adott kivétel azonnali továbbadását eredményezi.

Deklarációs utasításban kiváltódott kivétel azonnal továbbadódik. Csomagban bárhol bekövetkezett és ott nem kezelt kivétel beágyazott csomag esetén továbbadódik a tartalmazó programegységnek, fordítási egység szintű csomagnál viszont a főprogram félbeszakad.

A kivételkezelő dinamikus hatásköre itt is ugyanazokat a problémákat veti föl, mint a PL/I-ben. Az egyetlen különbség, hogy a hívási lánc programegységei szabályosan befejeződnek. Az Ada fordító nem tudja ellenőrizni a kivételkezelők működését.

Az Adában a saját kivételeknek alapvető szerepük van a programírásban, egyfajta kommunikációt tesznek lehetővé a programegységek között az eseményvezérlés révén.

Példa:

```
FUNCTION FAKT(N : NATURAL) RETURN FLOAT IS
BEGIN
IF N=1 THEN RETURN 1.0;
ELSE RETURN FLOAT(N)*FAKT(N-1);
END IF;
EXCEPTION
WHEN NUMERIC_ERROR => RETURN FLOAT_MAX;
END;
```

Ha olyan paraméterrel hívjuk meg a függvényt, amelyre a faktoriális értéke túl nagy, akkor kivételkezelés nélkül túlcsordulás következne be, így viszont a függvény a maximális lebegőpontos értékkel tér vissza.

10. GENERIKUS PROGRAMOZÁS

A generikus programozási paradigma az újrafelhasználhatóság és így a procedurális absztrakció eszköze. Ez a paradigma ortogonális az összes többi paradigmára, tehát bármely programozási nyelvbe beépíthető ilyen eszközrendszer. A generikus programozás lényege, hogy egy paraméterezhető forrásszöveg-mintát adunk meg. Ezt a *mintaszöveget* a fordító kezeli. A mintaszövegből aktuális paraméterek segítségével előállítható egy *konkrét* szöveg, ami aztán lefordítható. Az újrafelhasználás ott érhető tetten, hogy egy mintaszövegből tetszőleges számú konkrét szöveg generálható. És ami talán a leglényegesebb, hogy a mintaszöveg *típussal* is paraméterezhető.

Most az Ada lehetőségeit vizsgáljuk meg.

Az Ada *generikus* alakja:

```
GENERIC formális_paraméter_lista
törzs
```

A törzs egy teljes alprogram vagy csomag deklarációja, amiben szerepelnek a formális paraméterek. A generikus formális paraméterei változók, típusok és alprogramspecifikációk lehetnek. A `formális_paraméter_lista` alakja:

```
[{ változódeklaráció |
TYPE név IS {(<>) | RANGE <> | DELTA <> | DIGITS <> | tömbtípus
              |mutatótípus | [ LIMITED] PRIVATE }|
WITH alprogram_specifikáció [IS név ] |
WITH alprogram_specifikáció IS <> }; ]...
```

Konkrét alprogramot vagy csomagot ebből a mintaszövegből a következő utasítás segítségével lehet generáltatni:

```
{ PROCEDURE |FUNCTION|PACKAGE } generált_név
IS NEW generikus_név [(aktuális_paraméter_lista)];
```

A generikust így „hívjuk” meg. Ennek során lejátszódik a paraméterkiértékelés és a paraméterátadás.

A generikus formális paramétereinek száma mindig fix. A paraméterkiértékelésnél a sorrendi kötés az alapértelmezés, de alkalmazható a név szerinti kötés is. Az alprogramspecifikáció formális paraméterekhez az IS szerepeltetése esetén nem szükséges aktuális paramétert adni. Változóhoz azonos típusú konstans kifejezés, alprogramspecifikációhoz megfelelő specifikációjú eljárás- vagy függvénynevet adhatunk meg aktuális paraméternek. Típus formális paraméter esetén az aktuális paraméter rendre (a szintaktikai leírásban megadott sorrendet követve) egy sorszámozott, egész, fixpontos, lebegőpontos, tömb, mutató vagy tetszőleges típus neve lehet.

A paméterátadás változónál érték, típusnévnél név szerint történik. Alprogram specifikáció esetén, ha megadunk aktuális alprogram nevet, akkor a generált szövegben ez a név jelenik meg. Ha nem adunk meg aktuális alprogram nevet, akkor a generált név az `IS` után megadott `név` lesz, vagy `IS <>` esetén a generált név meg fog egyezni a formális paraméter nevével.

Példaként nézzük azt a generikus csomagot, ami a verem absztrakt adattípust implementálja:

```
generic
    MERET : integer;
    type ELEM is private;
    package VERMEK is
              type VEREM is limited private;
              procedure PUSH(S:in out VEREM; E:in ELEM);
              procedure POP(S:in out VEREM; E:out ELEM);
              TELE,URES : exception;
    private
              type VEREM is
                 record
                      HELY:array(1..MERET) of ELEM;
                      INDEX:integer range 0..MERET:=0;
                 end record;
    end;
```

```
package body VERMEK is
  procedure PUSH(S:in out VEREM; E:in ELEM) is
  begin
    if S.INDEX=MERET then raise TELE; end if;
    S.INDEX:= S.INDEX+1;
    S.HELY(S.INDEX):=E;
  end PUSH;
  procedure POP(S:in out VEREM; E:out ELEM) is
  begin
    if S.INDEX=0 then raise URES; end if;
    E:=S.HELY(S.INDEX);
    S.INDEX:=S.INDEX-1;
  end POP;
end VERMEK;
```

A generikusnak két formális paramétere van, a `MERET` a verem méretét, az `ELEM` a veremben tárolandó elemek típusát határozza meg. Ez utóbbi korlátozott privát típusú, tehát generáláskor bármilyen típusnév megadható hozzá aktuális paraméterként. Ezáltal tetszőleges típusú elemeket tartalmazó, tetszőleges méretű verem kezelését megvalósító konkrét csomag generálható belőle.

A vermet egydimenziós tömbbel reprezentáljuk. A verem LIFO viselkedését a két művelet (`PUSH` és `POP`) megfelelően implementálja. A két szélsőséges szituációt (a verem üres és tele van) saját kivételek kiváltásával jelezzük. A csomagban nincs kivételkezelés, tehát ezek a kivételek továbbadódnak a hívási környezetbe, hiszen az adott eseményt értelmesen csak ott lehet lekezelni.

A következőkben két konkrét veremkezelő csomag generálását láthatjuk:

```
package EGESZ_VEREM is new VERMEK(ELEM=>integer, MERET=>1024);
package LOGIKAI_VEREM is new VERMEK(100, boolean);
```

Az első esetben név szerinti kötést, a másodikban sorrendi kötést alkalmaztunk.

11. PÁRHUZAMOS PROGRAMOZÁS

A Neumann-architektúrán felépülő gépek szekvenciálisak: a processzor a programnak megfelelő sorrendben hajtja végre az utasításokat elemi lépésenként.

Egy processzor által éppen végrehajtott gépi kódú programot *folyamatnak* vagy *szálnak* hívunk. Ha ezek a működő kódok az erőforrásokat kizárólagosan birtokolják, akkor folyamatról, ha bizonyos erőforrásokat közösen birtokolhatnak, akkor szálakról beszélünk.

A párhuzamos programozás alapfogalmai (részletesen l. **Operációs rendszerek 1**):

Kommunikáció: A folyamatok kommunikálnak egymással, adatcserét folytatnak.

Szinkronizáció: A párhuzamosan futó folyamatoknak bizonyos időpillanatokban találkozniuk kell. Előfordul, hogy a szinkronizációs ponton keresztül történik adatcsere, a szinkronizációs ponton keresztül zajlik a kommunikáció. Például olyan információt vár az egyik a másiktól, ami nélkül nem tud továbbhaladni.

Konkurencia: A folyamatok vetélkednek az erőforrásokért.

Kölcsönös kizárás: Mivel a folyamatok kizárólagosan birtokolják az erőforrásokat, biztosítani kell, hogy amíg az egyik folyamat használja az erőforrást, addig a másik folyamat ne használhassa fel azt.

A párhuzamos programozási eszközrendszer először a PL/I-ben jelent meg. Létezik a Pascalnak és a C-nek is olyan változata, amely ebben az irányban bővíti tovább a nyelvet.

Azok az algoritmusok, amelyekkel eddig találkoztunk, szekvenciális algoritmusok, de léteznek párhuzamos algoritmusok is a problémák megoldására. Ezen algoritmusokon belül az egyszerre elvégezhető műveleteket egyszerre végezzük el.

A programozási nyelveknek a párhuzamos programozás megvalósításához rendelkezniük kell eszközzel:

– a folyamatok kódjának megadására,
– a folyamatok elindítására és befejeztetésére,
– a kölcsönös kizárás kérésére,
– a szinkronizációra,

– a kommunikáció megvalósítására,

– a folyamatok működésének felfüggesztésére,

– a folyamatok prioritásának meghatározására,

– a folyamatok ütemezésére.

12. A TASZK

Az Adában a taszk, mint programegység szolgál a párhuzamos programozás megvalósítására. A taszk tehát az a nyelvi eszköz, amely mögött folyamat áll. A taszk mint programegység önállóan nem létezik, csak egy másik programegységbe beágyazva jelenhet meg a program szövegében. A taszkot tartalmazó programegységet *szülőegység*nek hívjuk. Egy szülőegységen belül akárhány *testvértaszk* elhelyezhető. Ezek azonos szinten deklarált taszkok. A taszkok tetszőleges mélységben egymásba ágyazhatók. A szülőegység és a testvértaszkok törzse mögötti folyamatok működnek egymással párhuzamosan.

Többprocesszoros rendszerek esetén elképzelhető, hogy minden taszk más-más processzoron fut. Ez a valódi párhuzamosság. Egyprocesszoros rendszerek is programozhatók párhuzamos módon, ekkor az operációs rendszer szimulálja a párhuzamosságot. Ez a virtuális párhuzamosság.

Egy taszk akkor kezdi el a működését, amikor elindul a szülőegysége. Ez egy kezdeti szinkronizáció. Tehát az Adában az ütemezést a program szerkezete dönti el, vagyis lényegében a programozó ütemez.

Egy taszk befejezi a működését:

- ha elfogytak az utasításai,
- ha a szülőegysége vagy egy testvértaszkja befejezteti a működését az `ABORT név;` utasítással,
- explicit módon befejezteti saját működését külön utasítással.

A szülőegység akkor fejeződik be, ha ő, mint programegység befejeződött és ha az összes általa tartalmazott testvértaszk befejeződött. Ez egyfajta végszinkronizációs pont a szülőegység számára.

A taszknak két része van, *specifikáció* és *törzs*. Formálisan a következőképpen néz ki:

```
TASK [TYPE] név
[ IS entry_deklarációk
END [név]];
TASK BODY név IS
    [deklarációk ]
BEGIN
      végrehajtható_utasítások
      [ kivételkezelő ]
END [név];
```

A specifikációs részben ún. *entry-specifikációk* deklarálhatók, ezek segítségével belépési pontokat adhatunk meg. Ezek formálisan olyanok, mint az eljárások specifikációi, csak az alapszó `ENTRY`, nem `PROCEDURE`. Ezek a szinkronizáció eszközei az Adában.

Létrehozható taszk típus, ez egy korlátozott privát típusnak tekinthető.

A felhasználásuk módja szerint megkülönbözetünk két fajta taszkot. *Passzív* taszkok azok a taszkok, amelyek valamilyen szolgáltatást nyújtanak. Ezek specifikációs részében entry-specifikációk állnak, melyek leírják a szolgáltatás jellegét. *Aktív* taszkok azok a taszkok, amelyek igénybe veszik ezeket a szolgáltatásokat.

A szinkronizációt az Ada *randevúnak* hívja. Az aktív taszk egy entry-hívással képez egy randevúpontot. Ez formálisan megegyezik az eljáráshívással.

A passzív taszkon belül minden egyes entry-specifikációhoz meg kell adni legalább egy elfogadó utasítást, melynek alakja:

```
ACCEPT entry_név[(formális_paraméter_lista)]
[ DO végrehajtható_utasítások END [entry_név]];
```

A passzív taszk egy ilyen elfogadó utasítással képez egy randevúpontot. A passzív taszk által fölajánlott szolgáltatások a DO és END között vannak leírva.

Alaphelyzetben a randevú a következő módon megy végbe. A randevúhoz kell egy aktív taszk, amely meghív egy entryt, és egy passzív, amelyben a megfelelő elfogadó utasítás van. Elindul a két taszk, szekvenciálisan hajtja végre az utasításokat, amíg egy randevúponthoz

nem ér valamelyikük. Amelyik hamarabb ér a randevúponthoz, az bevárja a másikat, tehát addig a működését felfüggeszti. A randevú elsősorban a szinkronizáció eszköze az Adában, de van lehetőség a randevúban történő adatcserére is, erre szolgálnak a belépési pont formális paraméterei.

Ha mindkét taszk odaért a randevúponthoz, akkor az IN és IN OUT paraméterek esetén információ adódik át az aktív taszktól a passzív felé. Ezután, ha van DO-END rész, akkor az végrehajtódik, a randevú végén pedig az OUT és IN OUT paraméterek segítségével a passzív taszk felől mozog információ az aktív taszk felé. Tehát a randevúban szinkronizáció mindig van, kommunikáció és közös tevékenység pedig lehetséges.

A taszkok kommunikálhatnak a szülőegységben deklarált, a testvértaszkok számára globális változók segítségével is. Ezeket egyidőben használhatja ez összes testvértaszk. A közösen használt változókra a kölcsönös kizárást egy pragma segítségével biztosíthatjuk, ennek alakja:

```
SHARED(változó_név)
```

A szülőegység deklarációs részében kell megadni, ahol a változó deklarációja is szerepel.

Egy taszk specifikációs részében helyezhető el a következő pragma, melynek segítségével a taszkhoz prioritás rendelhető:

```
PRIORITY(kifejezés)
```

Egy alacsonyabb prioritású, vagy prioritás nélküli taszk soha nem akadályozhatja magasabb prioritású taszk munkáját.

Ha valamely passzív taszk egy adott szolgáltatását több aktív taszk akarja igénybe venni egyidejűleg, akkor az aktív taszkok egy prioritásos várakozási sorba kerülnek.

Minden taszk törzsében elhelyezhető a

```
DELAY kifejezés;
```

késleltető utasítás. A `kifejezés` nemnegatív, egész értékű, decimális számrendszerben értendő szám, amely a késleltetést adja meg másodpercben. Az adott taszk ennyi időre felfüggeszti a működését.

A randevú bekövetkezte a taszkokban befolyásolható a SELECT-utasítás segítségével. Ezen utasítás szerepe más-más az aktív és a passzív taszk esetén. Az aktív taszk ugyanis mindig

maga hív meg egy szolgáltatást jelentő belépési pontot, a passzív taszk viszont nem tudja sohasem, hogy egy elfogadó utasítás által felkínált szolgáltatást igénybe akar-e venni valamikor majd egy aktív taszk.

Az aktív taszkokban a SELECT-utasításnak két formája alkalmazható.

Feltételes randevúra szolgáló SELECT:

```
SELECT
  entry_hívás
  [végrehajtható_utasítások ]
ELSE
  végrehajtható_utasítások
END SELECT;
```

Ha az `entry_hívás` által kezdeményezett randevú azonnal létrejöhet, akkor végbemegy, ezután végrehajtódnak az esetlegesen megadott egyéb utasítások és a taszk kilép a SELECT-utasításból, ha viszont nem, akkor az aktív taszk nem vár, hanem „póttevékenységet” végez, azaz az ELSE-ágban lévő utasításokat hajtja végre és kilép a SELECT-utasításból.

Időzített randevúra szolgáló SELECT:

```
SELECT
  entry_hívás
  [végrehajtható_utasítások ]
ELSE
  késleltető_utasítás
  [végrehajtható_utasítások ]
END SELECT;
```

Ha az `entry_hívás` által kezdeményezett randevú azonnal létrejöhet, akkor végbemegy, ezután végrehajtódnak az esetlegesen megadott egyéb utasítások és a taszk kilép a SELECT-utasításból, ha viszont nem, akkor az aktív taszk várakozik a késleltető utasításban megadott ideig, miközben újra és újra megpróbál randevúzni és csak az adott idő letelte után hajtja végre az esetlegesen megadott „póttevékenységet” és lép ki a SELECT-utasításból.

A passzív taszkban elhelyezhető SELECT:

```
SELECT
    [ WHEN feltétel => ] alternatíva
    [ OR [WHEN feltétel => ] alternatíva ]...
    [ ELSE végrehajtható_utasítások ]
END SELECT;
```

Egy `alternatíva` alakja:

– elfogadó alternatíva:

```
elfogadó_utasítás [ végrehajtható_utasítások ]
```

– késleltető alternatíva:

```
késleltető_utasítás [ végrehajtható_utasítások ]
```

– befejeztető utasítás:

```
TERMINATE;
```

Legalább egy elfogadó alternatíva szükséges, de akármennyi lehet. A késleltető alternatívából bármennyi, befejeztetőből maximum egy szerepelhet. A késleltető és a befejeztető alternatíva kizárja egymást.

Egy alternatívát *nyíltnak* nevezünk, ha vagy nem szerepel előtte `WHEN feltétel`, vagy szerepel, de a feltétel igaz. Egyébként az alternatíva *zárt.*

Amikor egy ilyen SELECT-utasításhoz ér a passzív taszk, akkor

– Kiértékelődnek a feltételek és eldől, hogy mely alternatívák nyíltak és mely alternatívák zártak.

– Azon nyílt alternatívákban, melyekben DELAY-utasítás van, kiértékelődnek a megadott kifejezések és eldőlnek a várakozási idők.

– Egy nyílt elfogadó alternatíva kiválasztható, ha létezik olyan aktív taszk, amely ezzel a ponttal randevúzni akar (meghívta ezt az entryt). Ekkor a randevú végbemegy, végrehajtódnak az esetlegesen megadott egyéb utasítások és a taszk kilép a SELECT-utasításból. Ha egyszerre több kiválasztható elfogadó alternatíva van, bármelyik végrehajtódhat. Nem determinisztikus, hogy melyik randevú hajtódik végre, de mindig

csak egyetlen randevú mehet végbe.

- Egy nyílt késleltető alternatíva kiválasztható, ha nincs kiválasztható elfogadó alternatíva. Ha több kiválasztható késleltető alternatíva van, akkor a legkisebb várakozási idejűt választja ki. Ekkor a passzív taszk a megadott ideig várakozik, és közben vizsgálja, hogy nem futott-e be valamelyik nyílt elfogadó alternatívához randevú kérés. Ha igen végbemegy a randevú, ha nem, akkor a várakozási idő letelte után végrehajtja az esetlegesen megadott egyéb utasításokat, majd kilép a SELECT-utasításból.
- Nyílt befejeztető alternatíva akkor választható ki, ha a testvértaszkok, a szülőegység és minden, az adott taszk által tartalmazott taszk befejezte a működését. Ekkor a taszk befejezi a működését.
- Ha nincs kiválasztható nyílt alternatíva, és van ELSE-ág, akkor végrehajtódnak az ott megadott utasítások, és a taszk kilép a SELECT-utasításból. Ha nincsen ELSE-ág, akkor a taszk belefut egy végtelen várakozásba, és közben nézi, hogy egy nyílt alternatíva nem válik-e kiválaszthatóvá, vagy egy zárt nyílttá és kiválaszthatóvá.
- Ha minden alternatíva zárt, és van ELSE-ág, akkor végrehajtódnak az ott megadott utasítások, és a taszk kilép a SELECT-utasításból. Ha viszont nincs ELSE-ág, akkor bekövetkezik a `SELECT_ERROR` kivétel.

Aktív taszk csak olyan passzív taszkkal tud randevúzni, amelyik még működik. Ha egy olyan taszkkal akar randevúzni egy taszk, amelyikkel nem lehetséges (mert például már befejezte a működését), akkor kiváltódik a `TASKING_ERROR` kivétel.

A kivételkezelés szabályai a taszkok esetén némileg kiegészülnek. Ha egy kivétel randevúban következik be, akkor az a randevúban résztvevő mindkét taszkban kiváltódik. Viszont vannak olyan kivételek amelyek csak taszkban következhetnek be, így azok kezelését is csak taszkban lehet megoldani. Ezért az Ada azt mondja, hogy ha egy taszkban bekövetkezik egy kivétel és azt nem kezeljük, akkor az nem adódik tovább.

Példa:

Adva van egy folyamat, ami karaktereket állít elő a saját ütemének megfelelően. Van egy másik folyamat, ami felhasználja a termelt karaktereket szintén a saját ütemének megfelelően. Nincsenek szinkronban, párhuzamosan dolgoznak. Írjunk programot erre a problémára.

Írjunk egy termelő, egy fogyasztó aktív taszkot és egy passzív taszkot, amely fogadja, tárolja illetve átadja az karaktereket. Kell hozzájuk egy tartalmazó szülőegység, melynek egyetlen feladata a szinkronizáció. Legyen ez az alábbi blokk.

```
begin
  -- a taszkok deklarációja
  null;
end;
```

A termelő és a fogyasztó taszk törzsében legyen egy-egy végtelen ciklus:

```
loop
  -- a karakter előállítása
  BUFFER.WRITE(CHAR);
  exit when CHAR=END_OF_CHAR;
end loop;
```

```
loop
  BUFFER.READ(CHAR);
  -- a karakter feldolgozása
  exit when CHAR=END_OF_CHAR;
end loop;
```

A passzív taszk egy ciklikus reprezentációval kezelt sorban tárolja az adatokat:

```
task BUFFER is
  entry READ(C : out character);
  entry WRITE(C : in character);
end;
```

```
task body BUFFER is
   POOL_SIZE     : constant integer:=100;
   POOL          : array (1..POOL_SIZE) of character;
   COUNT         : integer range 0..POOL_SIZE:=0;
   IN_INDEX, OUT_INDEX : integer range 1..POOL_SIZE:=1;
begin
   loop
      select
         when COUNT < POOL_SIZE =>
          accept WRITE (C : in character) do
                 POOL(IN_INDEX):=C; end;
                 IN_INDEX:=IN_INDEX mod POOL_SIZE+1;
                 COUNT:=COUNT+1;
         or when COUNT > 0 =>
          accept READ(C : out character) do
                 C:=POOL(OUT_INDEX); end;
                 OUT_INDEX:= OUT_INDEX mod POOL_SIZE-1;
                 COUNT:=COUNT-1;
         or terminate;
      end select;
   end loop;
end BUFFER;
```

A törzs egy végtelen ciklus, amelyen belül egyetlen SELECT-utasítás van. A SELECT feladata a szinkronizáció. Három ága van, melyek közül a befejeztető alternatíva mindig nyílt. Elindul a szülőegység és azonnal be is fejezi a működését, várva hogy a taszkok is véget érjenek. A három taszk belefut a végtelen ciklusba. Az első randevút a termelő taszkkal hajtja végre a passzív taszk, majd kilép a SELECT-ből. Mivel végtelen ciklusról van szó, azonnal újból a SELECT-re kerül a vezérlés. A két aktív taszk előbb-utóbb kilép a végtelen ciklusból az EXIT-utasítással, és elér a törzsének a végére, befejezvén a működését. A szülőegység már várakozik, tehát kiválaszthatóvá válik a befejeztető alternatíva, a folyamatok lezárulnak.

13. INPUT/OUTPUT

Az I/O az a területe a programnyelveknek, ahol azok leginkább eltérnek egymástól. Az I/O plattform-, operációs rendszer-, implementációfüggő. Egyes nyelvek nem is tartalmaznak eszközt ennek megvalósítására, eleve az implementációra bízzák a megoldást.

Az I/O az az eszközrendszer a programnyelvekben, amely a perifériákkal történő kommunikációért felelős, amely az operatív tárból oda küld adatokat, vagy onnan vár adatokat. Az I/O középpontjában az *állomány* áll. A programnyelvi állományfogalom megfelel az absztrakt állományfogalomnak (l. **Adatszerkezetek és algoritmusok**). Egy programban a *logikai állomány* egy olyan programozási eszköz, amelynek neve van és amelynél az absztrakt állományjellemzők (rekordfelépítés, rekordformátum, elérés, szerkezet, blokkolás, rekordazonosító, stb.) attribútumként jelennek meg. A *fizikai állomány* pedig a szokásos operációs rendszer szintű, konkrét, a perifériákon megjelenő, az adatokat tartalmazó állomány.

Egy állomány *funkció* szerint lehet:

- *input* állomány: a feldolgozás előtt már léteznie kell, és a feldolgozás során változatlan marad, csak olvasni lehet belőle,
- *output* állomány: a feldolgozás előtt nem létezik, a feldolgozás hozza létre, csak írni lehet bele,
- *input-output* állomány: általában létezik a feldolgozás előtt és létezik a feldolgozás után is, de a tartalma megváltozik, olvasni és írni is lehet.

Az I/O során adatok mozognak a tár és a periféria között. A tárban is, és a periférián is van valamilyen ábrázolási mód. Kérdés, hogy az adatmozgatás közben történik-e konverzió. Ennek megfelelően létezik kétféle *adatátviteli mód*: a *folyamatos* (van konverzió) és a *bináris* vagy *rekord módú* (nincs konverzió).

A folyamatos módú adatátvitelnél a tárban és a periférián eltér a reprezentáció. Ebben az esetben a nyelvek a periférián az adatokat egy folytonos karaktersorozatnak tekintik, a tárban pedig a típusnak megfelelő belső ábrázolás által definiált bitsorozatok vannak. Az adatátvitel ekkor egyedi adatok átvitelét jelenti konverzióval. Olvasáskor meg kell mondania, hogy a

folytonos karaktersorozatot hogyan tördeljük fel olyan karaktercsoportokra, amelyek az egyedi adatokat jelentik, és hogy az adott karaktercsoport milyen típusú adatot jelent. Íráskor pedig rendelkezni kell arról, hogy a tárbeli, adott típusú adatot reprezentáló bitsorozatból a folytonos karaktersorozatban melyik helyen és hány karaktert alkotva jelenjen meg az egyedi adat.

A nyelvekben ezek megadására három alapvető eszközrendszer alakult ki:

- *formátumos módú adatátvitel*: minden egyes egyedi adathoz a formátumok segítségével explicit módon meg kell adni a kezelendő karakterek darabszámát és a típust.
- *szerkesztett módú adatátvitel*: minden egyes egyedi adathoz meg kell adni egy maszkot, amely szerkesztő és átviendő karakterekből áll. A maszk elemeinek száma határozza meg a kezelendő karakterek darabszámát, a szerkesztő karakterek megadják, hogy az adott pozíción milyen kategóriájú karakternek kell megjelennie, a többi karakter változtatás nélkül átvitelre kerül.
- *listázott módú adatátvitel*: itt a folytonos karaktersorozatban magában vannak a tördelést végző speciális karakterek, amelyek az egyedi adatokat elhatárolják egymástól, a típusra nézve pedig nincs explicit módon megadott információ.

A bináris adatátvitel esetén az adatok a tárban és a periférián ugyanúgy jelennek meg. Ez csak háttértárakkal való kommunikációnál jöhet szóba. Az átvitel alapja itt a rekord.

Ha egy programban állományokkal akarunk dolgozni, akkor a következőket kell végrehajtanunk:

1. *Deklaráció*: A logikai állományt mindig deklarálni kell az adott nyelv szabályainak megfelelően. El kell látni a megfelelő névvel és attribútumokkal. Minden nyelv definiálja, hogy milyen állományfogalommal dolgozik. Egyes nyelvek azt mondják, hogy a funkció is attribútum, tehát a deklarációnál eldől.
2. *Összerendelés*: Ennek során a logikai állománynak megfeleltetünk egy fizikai állományt. Ez a megfeleltetés vagy a program szövegében, nyelvi eszközzel történik (a fizikai állomány csak itt jelenik meg), vagy a program szövegén kívül, operációs rendszer szinten

végezzük azt el. Innentől kezdve csak a logikai állománynévvel dolgozunk, de a tevékenység mindig a mögötte álló fizikai állományra vonatkozik.

3. *Állomány megnyitása*: Egy állománnyal csak akkor tudunk dolgozni, ha megnyitottuk. Megnyitáskor operációs rendszer rutinok futnak le, ellenőrizve, hogy a logikai állomány attribútumai és a fizikai állomány jellemzői megfelelnek-e egymásnak. Egy állomány funkciója a megnyitásnál is eldőlhet bizonyos nyelvekben (pl. „inputra nyitunk"). Ekkor a program futása folyamán ugyanazt az állományt más-más funkcióra is megnyithatjuk.

4. *Feldolgozás*: Ha az állományt megnyitottuk, akkor abba írhatunk, vagy olvashatunk belőle. Az olvasást realizáló eszköznél meg kell adni a logikai állomány nevét és folyamatos módú adatátvitelnél egy tetszőleges változólistát. Ekkor a felsorolt változók értékkomponensüket az adott állományból kapják meg. Formátumos átvitelnél minden változóhoz egy formátumot, szerkesztettnél egy maszkot meg kell adni. Listázott átvitelnél a konverziót a változók típusa határozza meg. Bináris átvitelnél általában egy (ritkán több) változó adható meg, melynek rekord típusúnak kell lenni.

   A kiíró eszközrendszerben a logikai állomány neve mellett egy kifejezéslistát kell szerepeltetni. A kifejezések kiértékelődnek, és ezen értékek kiírásra kerülnek. A kifejezésekhez itt is egyenként szükségesek a formátumok, illetve a maszkok. Listázottnál a kifejezés típusa a meghatározó. Bináris átvitelnél a kifejezésnek rekordot kell szolgáltatnia.

5. *Lezárás*: A lezárás ismét operációs rendszer rutinokat aktivizál. Azért nagyon fontos, mert a könyvtárak információinak aktualizálása ilyenkor történik meg. Output és input−output állományokat le kell zárni, input állományokat pedig illik lezárni. A lezárás megszünteti a kapcsolatot a logikai állomány és a fizikai állomány között. A nyelvek általában azt mondják, hogy a program szabályos befejeződésekor az összes megnyitott állomány automatikusan lezáródik.

A programozási nyelvek a programozó számára megengedik azt, hogy input-output esetén ne állományokban gondolkozzon, hanem az írás-olvasást úgy képzelje el, hogy az közvetlenül valamelyik perifériával történik. Ezt hívjuk *implicit állománynak*. A megfelelő logikai és fizikai állomány most is létezik standard nevekkel és jellemzőkkel, de ezt a futtató rendszer

automatikusan kezeli. Tehát az implicit állományt nem kell deklarálni, összerendelni, megnyitni és lezárni. Az implicit input állomány a szabvány rendszerbemeneti periféria (általában a billentyűzet), az implicit output állomány a szabvány rendszerkimeneti periféria (általában a képernyő). A programozó bármely állományokkal kapcsolatos tevékenységet elvégezheti explicit módon (pl. az implicit output állományhoz hozzárendelheti a nyomtatót). Ha az író és olvasó eszközben nem adjuk meg a logikai állomány nevét, akkor a művelet az implicit állománnyal történik.

13.1. Az egyes nyelvek I/O eszközei

FORTRAN:

Szeriális, szekvenciális és direkt állományt tud kezelni. Eszközrendszere szegényes. Csak fix rekordformátumot tud kezelni. Ő vezeti be a formátumos adatátvitelt. A listázott átvitel a későbbi verziókba kerül be. Létezik benne a bináris adatátvitel, azonban nem rekordonként, hanem egyedi adatokként.

COBOL:

Erős I/O eszközrendszere van. Mindig konvertál. A COBOL vezeti be a szerkesztett átvitelt. Szeriális, szekvenciális, direkt, indexelt és invertált állományszerkezetet is ismer, de egyszerre csak egy másodlagos kulcs szerint tud keresni. Általában fix rekordformátumot kezel. Blokkolás lehetséges.

PL/I:

Kiemelkedően a legjobb állománykezelési eszközrendszerrel dolgozik. Az összes állományszerkezetet, adatátviteli módot, rekordformátumot és blokkolási lehetőséget ismeri és kezeli.

PASCAL:

Állománykezelési eszközrendszere szegényes. Fix és változó rekordformátumú szeriális állományt kezel. Az egyes implementációk ismerik a bináris és a folyamatos módú (a formátumos és a listázott egyfajta keverékeként) átvitelt, és esetlegesen közvetlen eléréssel dolgoznak. Blokkolás nincs. Nincs I/O utasítás, beépített alprogramokkal dolgozik.

C:

Az I/O eszközrendszer nem része a nyelvnek. Standard könyvtári függvények állnak rendelkezésére. Létezik a bináris és a folyamatos módú átvitel, ez utóbbinál egy formátumos és egy szerkesztett átvitel keverékeként. Szeriális szerkezetet kezel fix és változó rekordformátummal. Az I/O függvények minimálisan egy karakter vagy karaktercsoport, illetve egy bájt vagy bájtcsoport írását és olvasását teszik lehetővé.

Ada:

Minden perifériát tud nyelvi eszközökkel kezelni. Az Adának sem része az I/O, csomagok segítségével valósítja meg azt. Léteznek az alábbi csomagok:

- SEQUENTIAL_IO: Szekvenciális állományok kezelésére szolgál. Indexelt szekvenciális állománnyal is tud dolgozni.

- DIRECT_IO: Direkt állomány kezelésére szolgál.

- TEXT_IO: Szöveges állomány kezelésére szolgál.

- LOW_LEVEL_IO: Tetszőleges perifériával való kommunikációt biztosítása nyelvi eszközökkel.

- IO_EXCEPTIONS: Az I/O kivételek kezelésénél van jelentősége.

A logikai állomány deklarálása és fizikai állománnyal való összerendelése után a megnyitott állomány funkciója a programon belül dinamikusan változtatható, például egy kivétel bekövetkeztekor. Lezáráskor előírható, hogy az állomány megmaradjon vagy törlődjön.

14. IMPLEMENTÁCIÓS KÉRDÉSEK

Az eljárásorientált programozási nyelvek a rendelkezésükre álló memóriát általában a következő területekre osztják fel futás közben:

Statikus terület: ez tartalmazza a kódszegmenst és a futtató rendszer rutinjait.

Rendszer verem: tárolja az aktiváló rekordokat.

Dinamikus terület: A mutató típusú eszközökkel kezelt dinamikus konstrukciók helyezkednek el benne.

Sok nyelvi implementáció úgy kezeli a memóriát, hogy a szabad tárterület a verem és a dinamikus terület között van, tehát ezek egymás rovására növekszenek.

A kódszegmens a program gépi nyelvű utasításait, rendszerinformációkat és a literálok táblázatait tartalmazza.

Az eljárásorientált programozási nyelvek a programegységek futásidejű kezeléséhez, a hívási lánc implementálásához az ún. *aktiváló rekordot* használják. Ennek felépítése az alábbi:

Dinamikus kapcsoló: Ez egy mutató típusú mező, amely a *hívó* programegység aktiváló rekordját címzi. A hívási környezet érhető el vele és a programegység szabályos befejeződésekor az aktiváló rekord törlésénél van alapvető szerepe.

Statikus kapcsoló: Ez egy mutató típusú mező, amely a *tartalmazó* programegység aktiváló rekordját címzi. Statikus hatáskörkezelésnél ennek segítségével érhető el a tartalmazó környezet.

Visszatérési cím: A kódszegmens azon címe, ahol a programegység szabályos befejezése esetén a programot folytatni kell.

Lokális változók

Formális paraméterek (csak alprogram esetén)

Visszatérési érték (csak függvény esetén)

Az egyszerű típusú lokális változók számára a típusukhoz tartozó belső ábrázolásnak (fixpontos, lebegőpontos, logikai, karakteres, cím, felsorolásos) megfelelően foglalódik le a

tárterület. Az összetett típusúaknál már bonyolultabb a helyzet. Tömböknél általános, hogy a lefoglalt tárterület elején egy ún. tömbleíró helyezkedik el, amely tartamazza dimenziónként az indexek alsó és felső határát, az elemek típusát és az egy elem tárolásához szükséges bájtok számát. Ezután pedig jönnek az elemek sor- vagy oszlopfolytonosan. A rekord típusnál a mezők típusa dönt, ezek egymásután helyezkednek el. Változó hosszúságú rekordtípusnál a helyfoglalás a maximális méretre történik. Sztringeknél egyaránt szóba jöhet a fix és a változó hosszon való tárolás, az utóbbinál hosszmegadással, illetve végjellel. Halmaz esetén kevés elemszámnál karakterisztikus függvény, nagy elemszám esetén kulcstranszformációs táblázat alkalmazása a szokásos.

A formális paraméterek számára lefoglalt tárterület a paraméterátadástól függ. Érték szerinti esetben a formális paraméter típusának megfelelő tárterület szükséges. Cím és eredmény szerinti esetben egy cím tárolásához szükséges bájtmennyiség foglalódik le. Érték-eredmény szerintinél pedig az előző kettő. A név és szöveg szerinti paraméterátadás esetén ide egy paraméter nélküli rendszer rutin hívása kerül. Ez mindig lefut, amikor a formális paraméterre hivatkozás történik. Feladata a szövegkörnyezet meghatározása és abban az aktuális paraméter kiértékelése, aztán a formális paraméterek nevének felülírása.

Az aktiváló rekordok a veremben tárolódnak. A verem alján mindig a főprogram aktiváló rekordja van. Szabályos program befejezéskor a verem kiürül. Amikor meghívunk egy alprogramot, vagy blokkot, akkor felépül hozzá az aktiváló rekord és az a verem tetejére kerül (innentől aktív az adott programegység) és mindig az a programegység működik, amelynek aktiváló rekordja a verem tetején van. Szabályos befejeződéskor az aktiváló rekord törlődik.

Taszkok esetén (egy processzoron) egy „kaktusz” verem épül föl. A szülőegység aktiváló rekordja elhelyeződik a verem tejére és az általa meghívott nem taszk programegységek aktiváló rekordjai pedig fölé kerülnek. A testvértaszkok mindegyikéhez felépül egy-egy olyan verem, melynek az alján a szülőegység aktiváló rekordja van. Ezek a vermek egyidejűleg léteznek és tartalmazzák az adott taszk által létrehozott hívási lánc aktiváló rekordjait. A szülőegység aktiváló rekordja csak akkor törölhető, ha ninden testvértaszkjának verme kiürült.

IRODALOMJEGYZÉK

Bergin, T. J. – Gibson, R. G.: History of Programming Languages, Addison-Wesley, 1996.

Horowitz, E.: Magasszintű programnyelvek, Műszaki, 1987.

Kernighan, B. W. – Ritchie, D. M.: A C programozási nyelv, Műszaki, 2001.

Marcotty, M. – Ledgard, H.: The World of Programming Languages, Springer-Verlag, 1987.

Nyékiné Gaizler Judit(szerk.): Az Ada95 programozási nyelv, ELTE Eötvös Kiadó, 1998.

Nyékiné Gaizler Judit(szerk.): Programozási nyelvek, Kiskapu, 2003.

Pratt, T. W.: Programming Languages. Design and Implementation, Prentice Hall, 1984.

Pyle, I. C.: Az Ada programozási nyelv, Műszaki, 1987.

Sebesta, R. W.: Concepts of Programming Languages, Addison-Wesley, 2002.

Sethi, R.: Programming Languages, Concepts and Constructs, Addison-Wesley, 1996.

Teufel, B.: Organization of Programming Languages, Springer-Verlag, 1991.

The Programming Language Ada, Reference Manual, Lecture Notes in Computer Science 106, Springer-Verlag, 1981.

http://www.adahome.com/

http://www.engin.umd.umich.edu/CIS/course.des/cis400/algol/algol.html

http://cm.bell-labs.com/cm/cs/cbook/index.html

http://www.cobolreport.com/index.asp

http://www.fortran.com/

http://www.merlyn.demon.co.uk/pascal.htm

http://home.nycap.rr.com/pflass/pli.htm