Kritériumrendszer programozási nyelvek összehasonlító elemzésére

Criteria-system for Comparative Analyses of the Programming Languages

KOVÁCS Lehel István

Babeş-Bolyai Tudományegyetem, Kolozsvár
Számítógépes Rendszerek Tanszék

Abstract

60 years ago, in 1946, John von Neumann formulated the principles of large-scale computing machines. The machine-code was the only modality of programming the first generation of electronic computers constructed between 1946 and 1955. Between 1954 and 1958 appear the first high-level programming languages (FORTRAN, ALGOL). Using these programming languages the communication between man and computers evolves to the human side. The programs can be written in user-friendly mode, and translated to machine-code with compilers. We can speak about a very quick and spectacular evolution of programming languages the number of these exceeds 1000.

The aim of this article is to formulate a criteria-system for the classification and comparative analyses of the programming languages. This criteria-system can be a good methodological base for teaching the programming languages and searching the answer for the following question: which programming language provides the ideal elements and tools for solving the given problem.

Összefoglaló

60 évvel ezelőtt, 1946-ban Neumann János kidolgozta a korszerű számítógépek megépítésének alapelveit. 1946–1955 között megépültek az első generációs elektronikus számítógépek. Ezeket a kezdetleges számítógépeket a gépi kód közvetlen felhasználásával lehetett programozni. 1954–1958 között megjelentek a magas szintű programozási nyelvek (FORTRAN, ALGOL), amelyek segítségével emberközelibb formában lehet a programokat megírni, és azokat gépi kódra lefordítani. Ezt követően a programozási nyelvek gyors fejlődésnek indultak. Napjainkban több ezer programozási nyelvről beszélhetünk, és számuk növekszik.

Jelen cikkben a programozási nyelvek osztályozásához és összehasonlító elemzéséhez próbálunk felállítani egy kritériumrendszert, módszertani alapként az oktatáshoz és kutatáshoz, azon kérdés megválaszolására, hogy egy adott feladat megoldásához melyik programozási nyelv biztosít ideális eszközöket, elemeket.

Kulcsszavak: nyelvleírások, nyelvosztályok, történet, kapcsolatok, fordítóprogram, lexikális elemek, grammatikák, változók, konstansok, kifejezések, típusok, utasítások, kivételkezelés, programegységek, modulok, absztrakciós szintek

2000 Mathematics Subject Classification: **68N20**
1998 CR Categories and Descriptors: **D.3. [PROGRAMMING LANGUAGES]**

Bevezetés

A programozási nyelv egy jelölésmód, amelynek segítségével számítási folyamatokat írhatunk le, kódolni tudjuk az algoritmusokat ahhoz, hogy a számítógép megértse és végre tudja hajtani a megoldáshoz szükséges lépéseket. A programozási nyelv olyan számítógép által értelmezhető utasítások sorozata, mely adott feladat elvégzésének módját közli a számítógéppel. A számítógép által történő értelmezés nem direkt úton történik, hisz a számítógép csak a gépi kódot tudja értelmezni. Ez az indirekt út a *fordítóprogramok* jelenlétét teszi szükségesé, amelyek képesek az egyes programozási nyelvekben megfogalmazott feladatokat gépi kódra lefordítani.

A programozási nyelvek gyors és dinamikus fejlődést jártak be rövid – mintegy 60 éves – történelmük során, melyre az állandó megoldáskeresés, a feladat megfogalmazásának vagy megoldási lépéseinek lehető legjobb (egyszerűbb, egyértelműbb, optimális) kódolási módjának megtalálása volt jellemző.

Nem csoda, hogy e gyors fejlődés a programozási nyelvek hatalmas számához vezetett, számos speciális és általános nyelv alakult ki. A múlt századvég és az új századelő feladata ezeket a nyelveket osztályozni, kategorizálni, összehasonlító elemzések és szintézisek segítségével javaslatokat tenni, hogy a különböző programozási feladatok megoldására ki lehessen választani az ideális programozási nyelvet. Mindezek ellenére kevesen tették meg ezt a lépést, kevés összehasonlító tanulmány jelent meg, ezek inkább speciális esetekre, részletekre koncentrálnak, mintsem általánosan használható kritériumrendszert állítanának fel.

Kiindulva néhány alapmunkából: [1.], [2.], [3.], [4.], [5.], [6.], [7.], [8.], [9.], [10.], [11.], [12.], ezen cikk célja egy olyan kritériumrendszer felállítása, amely az osztályozási szempontok felállításán túl keretet biztosít az összehasonlító elemzések elkészítéséhez szükséges lépések, módszerek összefoglalására.

I. Programozási nyelvek osztályozása

Programozási nyelveket több szempont szerint is osztályozhatunk, különféle metszeteket készíthetünk, különböző nyelvosztályokat állíthatunk fel, de az egyes jellemzők közé éles határ nem húzható. *Hibrid nyelvekről* akkor beszélünk, ha az adott nyelv egy osztályozási szempont szerint több osztályba tartozik. Napjaink programozási nyelveinek többsége hibrid.

1.1. Amatőr és professzionális nyelvek

Az *amatőr programozási nyelvekre* az interaktivitás, a sok nyelvi elem, a gyors nyelvi fejlődés jellemző. Ezekben a nyelvekben a programok szerkezete egyszerű, és ezek speciális gépi tulajdonságokra épülnek rá (pl. *BASIC*, *Pascal*).

A *professzionális nyelvekre* a modularitás, a magasfokú stabilitás és a kevés nyelvi elem a jellemző. Ezen nyelvek igen hatékonyak, sok lehetőséggel bírnak és gépfüggetlen kódot generálnak, vagy a kód átvihető más architektúrájú gépekre is (pl. *C*, *Java*).

1.2. Emberközeliség

A *gépi nyelvek* használatával minden hardverlehetőség kihasználható, azonban a memóriacímeket, a memória-kiosztást és a programkódot önerőből kell megvalósítani (a memóriában lévő utasításkódokat közvetlenül a programozó adja meg). A megírt programok gépközeliek, közvetlenül a processzor utasításkészletére épülnek (pl. *Gépi kód*, *Assembly*).

Az *alacsony szintű nyelvek* géporientált nyelvek ugyan, de megjelennek a szimbolikus utasítások, azonosítók és címkenevek, megjelenik a feltételes vezérlésátadás fogalma, az eljárások és a visszatérések. Az adatokat deklarálni, definiálni lehet és a tárhely is ennek függvényében foglalódik le. Megjelennek a makrók és a direktívák. A közvetlen kódok helyett rövid, könnyen megjegyezhető szavakat alkalmazunk (mnemonikok) a könnyebb megjegyzés, a jobb átláthatóság kedvéért. Jobban áttekinthetők a címzési módok, a programokba megjegyzéseket szúrhatunk be, külön fordítható egységekkel dolgozhatunk (pl. *C*, *Assembly – Makró*).

A *magas szintű nyelvek* már feladatorientáltak, megjelenik a típus és a változó fogalma, kifejezések kiértékelésével komoly számításokat lehet elvégezni egyszerűen, megjelennek a ciklusok, elágazások. A nyelvek eljárásokat, függvényeket tudnak használni és komoly paraméterátadó mechanizmusokkal vannak felruházva (pl. *Pascal*, *Java*).

A *metanyelvekre* azért van szükségünk, hogy segítségükkel más nyelveket tudjunk leírni, ezáltal kizárható a különböző deklarációkban meghúzódó többértékűség (pl. *EBNF*, *BNF*).

1.3. Típusok használata

A *nem típusos nyelvek* esetében ha létezik is a változó fogalma, ez nincs kötve semmiféle típushoz (pl. *BASIC*, *JavaScript*).

A *típusos nyelveknél* megjelenik a típus fogalma, amely meghatározza, hogy a változó milyen értékeket vehet fel, mekkora memóriatartományra van szüksége, milyen műveletek végezhetők el vele stb. (pl. *C*).

A *szigorúan típusos nyelvek* esetében szigorú szabályok írják elő a típusok közötti átalakításokat, konverziókat (pl. *Pascal*).

1.4. Alapelvek szerint

Az *imperatív nyelvek* (I) osztályába a Neumann architektúrához szorosan kötődő algoritmikus nyelvek tartoznak, amelyeknél fő programozási egység az utasítás, és ezek egymásutánisága vezérli a processzort. A tár bizonyos területén lévő értékeket módosíthatjuk, így változókról beszélhetünk. A programozó mondja meg, hogy mit és hogyan kell csinálni (pl. *C*, *Pascal*).

Az *alkalmazás háttérnyelvek és makrónyelvek* (A) nagyobb alkalmazások hátterében feladatokat valósítanak meg, segítségükkel testre szabható, könnyebben, gyorsabban vezérelhetők lesznek az alkalmazások (pl. *Visual Basic for Application*, *AutoLisp*).

A *procedurális, strukturált nyelvek* (S) egy adott probléma megoldásának algoritmusát írják le. (pl. *BASIC*, *Algol*).

A *deklaratív nyelvek* (D) osztályába azok a matematikai logikára, vagy függvényhasználatra épülő, nem algoritmikus nyelvek tartoznak, amelyeknél a programozó csak a megoldandó feladatot írja le, a megoldást magát a rendszer végzi el. Ezeknél a nyelveknél nem létezik utasításfogalom, a tárhely értékeit nem lehet módosítani, nem léteznek adatok, vagy ezeknek teljesen más a szerepük (pl. *Prolog*, *Lisp*, *Miranda*, *Haskell*, *SML* – funkcionális, logikai nyelvek).

Az *applikatív nyelvek* (App) függvények változókra történő alkalmazásaival operálnak. Nincs mellékhatás (pl. *ML*)

A *funkcionális nyelvek* (F) magas szintű függvények használatára és operátor definíciókra épülnek. Az operátorok függvényeket manipulálnak, mintha azok egyszerű adatok lennének. (*Clean*, *Haskell*, *Miranda*, *SML*, *Hope*, *FP*).

A *definíciós nyelvek* (Def) olyan applikatív nyelvek, amelyeknél a megfeleltetések (értékadások) definíciókként vannak értelmezve (pl. *DAML*).

Az *egyszeres megfeleltetésű nyelvek* (SSL) esetén egy változó a láthatósági területén csak egyszer fordulhat elő a bal oldalon, egyszer vehet fel értéket (pl. *DAML*).

Az *adatfolyam (dataflow) nyelvek* (DF) az adatfolyam architektúrák programozási nyelvei (pl. *FOOD*, *TDFL*).

A *logikai nyelvek* (L) predikátumokra és relációkra épülnek. Tényekből szabályok segítségével következtetéseket tudnak levonni általában rezolúció-kalkulust használva (pl. *Prolog*).

A *megkötésorientált nyelvek* (C) a megoldandó feladatot megkötések sorozataként fogalmazzák meg és oldják meg (pl. *OCL*, *DAML+OIL*, *Eclipse*, *GUIdeLA*).

A *konkurrens nyelvek* (P) segítségével a párhuzamos, konkurrens, osztott, többszálas programokat tudjuk megfogalmazni (pl. *NESL*, *Orca*, *mpC*, *BLAZE*, *PARLANSE*).

A *folyamatszálas vagy veremalapú nyelvek* (T) alacsony szintű, konkrét feladatok megoldására szakosodott programozási nyelvek, melyek segítségével hatékonyan programozható a verem, vagy különböző grid-alapú osztott rendszerek (pl. *Vcode*, *Zero*).

A *matematikai vagy szimulációs nyelvek* (M) konkrét feladatosztályok megoldására szakosodtak. Többnyire matematikai számítások, szimulációs folyamatok végezhetők el velük igen nagy hatékonysággal. Kísérletek, folyamatok eredményeinek előrejelzését segítő eszközök (pl. *Mathematica*, *Matlab*, *Mathcad*).

A *szimbólum-feldolgozó nyelvek* (Sim) nagy mennyiségű szöveges információk, hosszú listaadatok értékelésére, elemzésére kidolgozott célnyelvek. Felhasználási területük az információkutatás, a matematikai kutatások. (pl. *Lisp*, *SPSS*)

A *formulakezelő nyelvek* (For) nagy pontosságot igénylő, bonyolult matematikai, műszaki számítások számítógép által történő elvégzéséhez biztosítanak hatékony segítséget, eszközöket (pl. *LotusScript*).

Az *objektumorientált nyelvek* (O) esetén magas absztrakciós szinten lévő osztályok és ezekből példányosított objektumok segítségével fogalmazhatjuk meg és oldhatjuk meg a feladatot. Az osztályok zárt egységnek tekintik az adatokat és az őket kezelő eljárásokat. Az objektumok egymással kommunikálnak (pl. *Java*, *Delphi*, *Sather*, *Modula*, *Smalltalk*, *Eiffel*, *Oberon*).

A *vizuális nyelvek* (V) esetén grafikus eszközök sokasága vehető igénybe az algoritmus leírására, megadására (pl. *ProGraph*).

A *negyedik generációs nyelvek* (4GL) nagyon magasszintű nyelvek, melyek segítségével a megoldandó feladat természetes nyelven, vagy diagrammok használatával fogalmazható meg. A fordítóprogram választja ki a megfelelő adatszerkezeteket vagy algoritmusokat (pl. *UML*, *RAD*).

A *lekérdező nyelvek* (QL) az adatbázis-programozás fő kommunikációs eszközei, interfészei (pl. *SQL*).

Az *adatbázis és szövegfeldolgozó nyelvek* (DB) segítségével adatokat, információkat strukturálhatunk, dolgozhatunk fel, módosíthatjuk, karbantarthatjuk az adatbázisban tárolt adatokat (pl. *SQL*, *ODL*, *TeX*, *HTML*).

A *specifikáló (leíró) nyelvek* (Spec) a szoftver vagy hardver tervezésének formális leírását szolgálják (pl. *VHDL*).

Az *assembly nyelvek* (Asm) a gépi kód szimbolikus jelölésére szolgálnak egy adott számítógép architektúrán (pl. *TASM*, *MASM*).

A *rendszer- és fordítóprogramok írására specializálódott nyelvek* (Sys) hatékonyan támogatják az alacsony szintű programozást és az operatív tár közvetlen kezelését, lehetővé teszik a bitenkénti műveletek végzését, de mégis rendelkeznek a magas szintű nyelvek előnyeivel (pl. *Gnu C*).

A *köztes nyelveket* (Int) a fordítóprogramok használják mint ábrázolás rendszert. Lehetnek szöveges vagy bináris formátumúak (pl. *Java ByteCode*, *OBJ*).

A *parancssor vagy szkriptnyelvek* (Com) segítségével operációs rendszer közeli feladatok írhatók le (pl. *BAT*, *Shell*).

A *kiterjeszthető nyelvek* (Ex) esetében a programozási feladat megoldásához a nyelv csak egy minimális alapot definiál. A hiányzó eszközöket a programozó maga állíthatja elő a már létező elemek felhasználásával, kombinálásával (pl. *XML*).

A *metanyelvek* (ML) más nyelvek deklarálására szolgálnak (pl. *BNF*, *EBNF*).

Az *egyéb, vagy más alapelvekre épülő nyelvek* (nemkonvencionális nyelvek – NCL) képezik az utolsó nagy nyelvosztályt. Ilyenek a különböző párhuzamos, adatfolyam, szisztolikus működést leíró nyelvek, vagy minden olyan nyelv, amelyet egy bizonyos speciális probléma megoldására tervezvtek.

1.5. Generációk szerint

Az elektronikus számítógépek nagy generációi tulajdonképpen meghatározták a programozási nyelvek generációit is. Ilyen értelemben beszélhetünk *első (1GL), második (2GL), harmadik (3GL), negyedik (4GL)* és *ötödik (5GL) generációs programozási nyelvekről*.

Az *első generációs nyelvek*et (kb. 1946–1955) a teljes mértékű processzorfüggőség jellemezte. Az utasítások bitsorozatok voltak, amelyeket a gép előlapján lévő kapcsolókkal lehetett megadni. Az első programozási nyelv a gépi kód volt. Ennek a nyelvnek az utasításait a számítógép képes volt közvetlenül, minden átalakítás nélkül végrehajtani. A nyelv erősen gépfüggő volt, hiszen minden gépen más és más utasításokat használt, az adott problémát minden géptípus esetén másképpen kellett leírni. A gépi kód nagy előnye a gyorsaság és az egységesség. A generáción belül komoly előrelépést jelentett az *Assembly* nyelvek megjelenése. A gépi kódú utasításokhoz egy-egy *mnemonikus* kódot rendeltek hozzá, a tárcímeket pedig a memória kezdetéhez viszonyított relatív címekkel számították. Az egyes memóriacímeket egy-egy szimbolikus névvel lehetett helyettesíteni.

A *második generációs nyelvek* megjelenése (kb. 1955–1963) tulajdonképpen a számítógépek alkalmazási területe kibővülésének eredménye. Felmerült az igény, hogy a programokat minél gyorsabban és hibamentesebben írják meg a programozók. A gépi kód és az assembly nehézkessége, géphez igazodása miatt nem volt erre alkalmas. A 60-as évek elején jelentek meg az első magas szintű programozási nyelvek, melyek már nem a számítógép sajátosságaihoz, hanem a problémához igazodtak. Egyetlen magas szintű programnyelvi utasítás több gépi kódú utasítást jelent. Ekkor jelennek meg a fordítóprogramok (*compiler*) és a szerkesztők (*linker*). Az első magas szintű programozási nyelv a *FORTRAN* volt.

A *harmadik generációs nyelvek* (kb. 1963–1973) már magas szintű programozási nyelvek, olyan nyelvek, amelyek nem egy konkrét feladatosztály megoldására specializálódtak, mint a második generációs nyelvek, hanem általánosak, univerzálisak voltak. Az új korszakot a procedurális programozási szemlélet és az adatok struktúrájának hangsúlyozása jellemezte. Ekkor jelennek meg az objektumorientált nyelvek is. A harmadik generációs nyelvek egyaránt alkalmasak az adatfeldolgozási és a számítási problémák megoldására, rendszerprogramozásra, emellett a program szerkezete egyszerű és követhető, formai előírásai nem túl szigorúak. A program tartalmaz egy főprogram részt, amely külön blokkokban több alprogramot foglalhat magába, az alprogramok egymásba ágyazhatók. Strukturált programozásról beszélhetünk (pl. *Algol*, *C*, *Pascal*, *BASIC*).

A *negyedik generációs nyelvek* (kb. 1973–) napjaink programozási eszközei. Bonyolult lekérdező nyelvek, programkód generátorok, interaktív fejlesztői környezetek, melyek már túl vannak a magas szintű nyelvek osztályán. A bonyolult információs rendszerek fejlesztése, központilag vezérelt számítógép-hálózatok programozása, döntéshozást támogató rendszerek megvalósítása túlmutatott a hagyományos programozási nyelvek által biztosított eszközök lehetőségein. Gyors alkalmazásfejlesztésre, vizuális paradigmára támaszkodó objektum- és komponensorientált kódgenerátorokra, magas hardverfüggetlenséget támogató nyelvekre van szükség. Sajnos, a program hatékonyságának növekedésével egyenes arányban nő a program hardver – elsősorban memória, tárhely – igénye is, a lefordított program mérete is (pl. *UML*, *Delphi*, *RAD*).

Az *ötödik generációs nyelvek* (1981–) alternatív irányvonalat képviselnek, valójában két fogalmat takarnak: egy gép közelebbi, de magas szintű nyelvet, amely tulajdonképpen a számítógép operációs rendszerét jelenti, és egy természetes nyelvet, amely során az ember és gép közötti kommunikáció („programírás") megvalósul (pl. *Prolog*, *KL1*, *természetes nyelvek*).

1.6. Számítási modellek szerint

Azon absztrakt modelleket követve, amelyeknek alapján az algoritmusokat végre kell hajtani, a feladatot meg kell oldani, a következő nagy paradigmákat különböztethetjük meg:

- *Az imperatív paradigma (IP):*
  - egyszerű operációs paradigma (SOP)
  - a Neumann-féle paradigma (NP)
  - az automata feldolgozás paradigmája (AP)
  - az adatbázis-kezelés paradigmája (DBP)
- *A deklaratív paradigma (DP)*
  - a funkcionális paradigma (FP)
  - a logikai paradigma (LP)
- *A párhuzamos és osztott paradigma (PDP)*
- *Az objektumorientált paradigma (OOP)*
- *A vizuális paradigma (VP)*
- *Az ötödik generációs paradigma (5GP)*
- *Alternatív paradigmák (AP)*

II. Imperatív programozási nyelvek elemzési szempontjai

Ha elemezni szeretnénk egy programozási nyelvet, vagy összehasonlító elemzéseket szeretnénk végezni, célszerű az alábbi kritériumrendszer szerint körbejárni a témát.

2.1. Nyelvleírások, könyvészet

Az elemzés első lépésében általánosságában szeretnénk megismerkedni a programozási nyelvvel:

- A nyelv céljai és specifikációja
- A nyelv rövid jellemzése
- Milyen feladatok megoldására specializálódott?
- Mennyire elterjedt a nyelv?
- Honnan lehet hozzáférni?
- Honlapok
- Létező fordítóprogramok
- Milyen dialektusokkal rendelkezik a nyelv?
- Jellemző példaprogram

2.2. Milyen nyelvosztályokba sorolható a nyelv?

Az előző fejezet alapján felállítjuk azokat a fő nyelvosztályokat, alosztályokat, amelyekbe besorolható a nyelv.

- Nyelvosztályok és alosztályok
- Hasonló nyelvek
- Paradigmák
- Hibrid nyelv-e vagy sem?

2.3. Története

Egy programozási nyelv története, létrehozásának körülményei sokat elárulhat a nyelv céljáról, specifikációjáról, fontosabb verziószámai pedig a fejlődéséről, korszerűsítéséről. Például igen érdekes az *Ada* nyelv története: 1975-ben az Amerikai Védelmi Minisztérium finanszírozásával megindult egy olyan komplex programozási nyelv elméletének kidolgozása, amely a kor legújabb kihívásait megoldotta. Az új kívánalmaknak megfelelő nyelv vázlatát STRAWMAN-nak (szalmabáb) nevezték el. Ezt felülvizsgálva az új változat a WOODENMAN (fabáb) nevet kapta. További vizsgálatok eredménye lett a TINMAN (ónbáb), majd az IRONMAN (vasbáb) jelentések. Ekkor versenyfelhívást tettek közzé, hogy ki tud egy olya nyelvet tervezni, ami a legközelebb áll az IRONMAN-ben szereplő leíráshoz. A négy induló közül a győztes a GREEN (zöld) csapat lett, amely a francia Cii-Honeywell Bull csoportja volt, amit Jean Ichbiah vezetett. A legújabb követelményeket STEELMAN-nak (acélbáb) nevezték el, és az ebből származó nyelvet *Ada* névre keresztelték Ada Augusta Byron (1815–1852) „az első programozó” tiszteletére. Az *Ada* potenciálisan a legfejlettebb nyelv lett a 80-as évek közepére, de szerepe ma messze nem akkora, mint várták volna.

- Kik tervezték?
- Mikor tervezték?
- Mi volt a terv neve?
- Miről kapta nevét a nyelv?
- Fontosabb verziószámok, bővítések
- Utolsó módosítás ideje
- A nyelv ősei
- Érdekességek a nyelvvel kapcsolatosan

2.4. Kapcsolat az operációs rendszerrel és a számítógéppel

A számítógép architektúrájával, az operációs rendszerrel fennálló kapcsolatokat vizsgáljuk:

- Architektúrafüggő-e a nyelv?
- Operációs rendszer függő-e a nyelv?
- Platformfüggetlen vagy átvihető, hordozható?
- Létezik-e köztes kód? Mi a szerkezete? Van-e virtuális gép?
- Milyen futtatható állományokat tud generálni (EXE, COM stb.)?
- Hogy veszi át a paramétereket a parancssorból?

Például a *Pascal* és az *Ada* csak függvények segítségével *(ParamCount, ParamStr*, illetve *Ada.Command_Line.Argument_Count, Ada.Command_Line.Argument)* tud operálni a parancssoron, a *C* és a *Java* a *main* függvény paraméterei által.

A *Pascal* átvihető nyelv például DOS és Linux között, a forráskódot kisebb-nagyobb módosításokkal le lehet fordítani (a Linux nem ismeri a CRT egységet), a *C* nyelv hordozható, a forráskód módosítás nélkül, vagy minimális módosítással lefordítható, a *Java* nyelv platformfüggetlen, a tárgykód lefut a különböző operációs rendszerek alatt a különböző architektúrákon.

2.5. A fordítóprogram

A hatékony tárgykódot, az interaktív, sőt feltételes fordítást a nyelv fordítóprogramja biztosítja. A fordítóprogram elemzésével a következő kérdésekre keresünk válaszokat:

- Parancssoros fordítóprogrammal rendelkezik-e?
- Milyen paraméterekkel kell meghívni?
- Milyen direktívákkal rendelkezik?
- Van-e pre- vagy posztprocesszálás (előfeldolgozás, utófeldolgozás)?
- Hány menetes fordítóprogramról beszélhetünk?
- Van-e külön szerkesztő (linker)?
- Optimalizál-e a fordítóprogram (Ha igen, akkor milyen elvek alapján)?
- Rendelkezik-e környezettel?
- Milyen tulajdonságokkal van a környezet felruházva?
    - Szövegszerkesztője
    - Fordítórendszere
    - Szerkesztőrendszere (linker)
    - Futtatórendszere

    - Súgó, kódkiegészítők, sablonok
    - Varázslók, kódgenerátorok
    - Tervezőfelület
    - Debugger, nyomkövető
    - Szimbólumkövető
    - Adatbázis-tervező
    - Támogatja-e a csoportprogramozást?
    - Más környezeti eszközök, beágyazott lehetőségek
- Fordítóprogram, értelmező, átalakító
- Hogyan kezeli a hibákat a fordítóprogram?
- Létezik-e formális helyességbizonyító?
- Milyen önellenőrző mechanizmusokkal rendelkezik?
- Mennyire gyors a fordító?

Például a *FoxPro* vagy *Logo* értelmezővel (interpreter) rendelkezik, amely értelmezi és végrehajtja a beírt utasításokat, programokat. A *Pascal* fordítóprogrammal (compiler) rendelkezik, a programokat elemzés után futtatható exe állománnyá fordítja, majd azt lehet futtatni. A *Java* átalakítóval (transzlátor) rendelkezik, a forráskódból köztes kódot állít elő, majd ezt a köztes kódot értelmezi a *Java Virtuális Gép* az adott architektúrán, operációs rendszeren.

Az *Assembly* parancssoros fordítóval rendelkezik, a *Pascal* parancssoros fordítóval, a 6.0-ás verziótól TurboVision környezettel, a *Delphi* fejlett környezettel rendelkezik. A *Java* fordítóhoz több környezet is létezik. Ezeket a környezeteket *IDE*-nek *(Integrated Development Environment)*, *beágyazott fejlesztési környezetek*nek nevezzük.

2.6. Lexikális elemek

A lexikális elemek összessége (kulcsszavak, azonosítók, konstansok, műveletek, speciális szimbólumok stb.) azt az eszköztárat képezi, amellyel a programozó direkt operál a programozás során. A következő kérdéseket kell megválaszolnunk:

- Milyen karakterek használhatók a nyelvben?
- Melyek a határoló jelek (szeparátorok)?
- Fehér karakterek
- Kis- és nagybetűk használata
- Azonosítók
    - Milyen karakterek használhatóak az azonosítók leírására?
    - Mi az azonosító szintaxisa?
    - Van-e hosszúsági megkötés az azonosítókra?
    - Vannak-e kulcsszavak?
    - Van-e különbség kulcsszó és az előre definiált szó között?
    - Vannak-e írásra vonatkozó konvenciók?
- Értékkonstansok
    - Milyen numerikus értékkonstansok vannak?
    - Milyen más alapok vannak a 10-esen kívül?
    - Mi az egész, illetve valós értékkonstansok szintaxisa?
    - Hogyan határoljuk a sztring értékkonstansokat?
    - Többsoros sztring értékkonstansok megengedettek-e?
    - Vezérlőkaraktereket használhatunk-e sztring értékkonstansokban, és hogyan?
- Megjegyzések
    - Van-e és hogyan használhatóak?
    - Megjegyzés a sor végéig?
    - Teljes sor megjegyzéssé alakítása?
    - Többsoros megjegyzés?
    - Egymásba ágyazható megjegyzések?
    - Dokumentációs megjegyzés?

Például az egész és valós konstansok esetén egyes nyelvek, pl. *Ada*, *Perl*, *Eiffel*, megengedik az aláhúzásjel („_”) használatát is százas elhatárolóként: *123_456, 1_000_000*. A *C*, *C++*, *Java*, *C#* nyelvekben a konstans után írt *l*, *L* vagy *u*, *U* betű tipizálja a számkonstanst: az `5L` *long* (hosszú egész) lesz, a `10u` *unsigned* (előjel nélküli) lesz. *Pascal*ban a valós szám kitevős alakjából elhagyható a tizedespont: *1E+2* a *100.0* valós

számot jelenti. *BASIC*-ben a tizedespont elé nem kell kitenni a nullát: *.5* a *0.5*-öt jelenti. Az *Ada* nyelvben egészeket is megadhatunk exponenciális alakban, ekkor a kitevő pozitív kell, hogy legyen: *1E3 = 1000*. Eiffelben nem szükséges a tizedespont mindkét oldalára számjegyet írni: *-1.* a mínusz egyes valós konstansot jelenti. *C++*-ban a *d*, *D* vagy *f*, *F* utótaggal pontosíthatjuk a valós konstans típusát (*double* vagy *float*). A *Java* nyelv definiálja a *NaN* konstanst (*Not a Number*), valamint a pozitív és negatív végteleneket jelölő *POSITIVE_INFINITY* és *NEGATIVE_INFINITY* konstansokat. Ezeknek véges szám hozzáadása vagy kivonása nem változtatja meg értéküket, szorzatuk a *NEGATIVE_INFINITY*-t eredményezi, összegük nem definiált. A nulla konstansnak előjelt is adhatunk: *+0.0* vagy *-0.0*. Az *1.0 / 0.0* eredménye a *POSITIVE_INFINITY*, míg az *1.0 / -0.0* eredménye a *NEGATIVE_INFINITY*. A *0.0 / 0.0* a *NaN*-t eredményezi.

2.7. Milyen grammatikákkal írható le a nyelv?

A fordítóprogram elméleti hátterére engednek következtetni az itt feltett kérdések:

- Lexikális elemek leírása
- Szintaktika
- Szemantika
- Speciális konstrukciók
- Ortogonalitás
- Egységesség
- Tömörség

Az *ortogonalitás* tulajdonsága azt jelenti, hogy minél több egymástól független elemet tartalmazzon a szintaktika, és ezeket bármely kombinációban lehessen alkalmazni (pl. a következő *C++* deklaráció: ***unsigned long int*** *valtozo;*). A nyelv néhány alaptulajdonsággal rendelkezik, ezen tulajdonságok mindegyike külön-külön is érthető, de együttes használatukkor is értelmes kifejezést kapunk. Így a nyelv könnyebben tanulható lesz, hisz csak kevés elemet kell megtanulni és ezeket kombinálni lehet, hátulütője viszont az, hogy a fordítóprogram logikailag zavaros, vagy kevéssé hatékony kombinációkat is le kell tudjon fordítani.

Az *egységesség* megköveteli, hogy a szemantikailag egységes elemek hasonló fogalmakat takarjanak, a *tömörség* pedig azt, hogy egy elem legyen szemantikailag használható több fogalom értelmezésére, különböző kontextusokban (pl. a „+” operátor stringekre, egészekre, valós számokra, halmazokra.)

2.8. Változók, szimbolikus konstansok

A tárhelyek címzésére, használatára szolgálnak a változók és a szimbolikus konstansok. A programozási nyelv módszereket kell hogy biztosítson mind az egyszerű, mind az összetett típusok konstansainak, változóinak a megadására, ezek későbbi használatára.

- Kell-e deklarálni a változókat?
- Vannak-e globális, lokális változók?
- Hogy lehet deklarálni a konstansokat?
- Léteznek-e speciális megkötések a nevekre?
- Írásra vonatkozó konvenciók
- A deklarációk szintaxisa
- A változóknak adhatunk-e kezdőértéket?
- Hogyan adjuk meg az összetett konstansok értékeit?
- Vannak-e dinamikus változók?
- Vannak-e közös referenciájú változók?
- Vannak-e automatikus változók, statikus változók?
- Regiszterekben tárolt változók
- Létezik-e a perzisztencia fogalma, milyen perzisztenciáról beszélhetünk?

2.9. Kifejezések

A *kifejezések* olyan programelemek, amelyek felhasználásával leírható a számítási folyamat. Szerkezeti és szintaktikai szempontból egy kifejezés *operátorok*ból és *operandusok*ból áll. Egy kifejezés kiértékelésének célja a számítási folyamat elvégzése, mely legtöbbször egy adott változó értékének a kiszámításával ekvivalens.

- Kifejezések szintaxisa
- Létezik-e mellékhatás?
- A logikai kifejezéseket hogyan értékeli ki (teljes, részleges)?

- Milyen műveleteket használhatunk?
- Vannak-e bitműveletek?
- Milyen típusokat téríthet vissza a kifejezés?
- A műveletek sorrendje és a kiértékelés iránya

Logikai kifejezéseknél gyakori a *nem teljes kiértékelés*. Ha egy logikai kifejezés csak *és* műveletekből áll, és az egyik operandusa *hamis*, vagy ha csak *vagy* műveletekből áll és az egyik operandusa *igaz*, illetve az előbbiek értelemszerű kombinációja a *tagadás* művelettel esetén, a kifejezést nem kell teljesen kiértékelnünk, megállhatunk az első olyan operandusnál, amelynél már egyértelmű lesz a kifejezés eredménye. Ilyen esetben vigyázni kell a különféle mellékhatásokkal, mert pl. a meg nem hívott függvények ezeket nem tudják kifejteni. Számos programozási nyelv erre lehetőséget biztosít, ekkor a *complete boolean evaluation* direktívát kell kikapcsolni. Pl. a (1 *és* 0) *és* ((1 *és* 1) *vagy* (0 *és* 1)) logikai kifejezés kiértékelése megállhat az első *és* utáni 0-nál.

Az *infix* jelölési módot a matematikából kölcsönöztük. A bináris operátor a két operandusa között állhat: $a_1\ o^2\ a_2$, pl. $x + y$. Ennek a jelölésmódnak az a hátránya, hogy a kiértékelése nem egyértelmű. Például az $x + y * a - b$ kifejezés esetén, ha semmiféle háttérismeretünk nincs, nem tudjuk eldönteni, hogy a műveleteket milyen sorrendben végezzük el. A háttérismeret, amire szükségünk van, a *művelet prioritása* vagy *precedenciája*, amely az elvégzés sorrendjét határozza meg. Hasonlóan szükségünk van az *asszociativitás* fogalmára is, amely megmondja, hogy több azonos prioritású művelet esetén melyiket kell elvégezni. A matematikai jelölésmódhoz hasonlóan, a prioritás és az asszociativitás megváltoztatására zárójeleket használhatunk. Például az (5 - 1) / (4 - 2) / (8 - 6) kifejezés esetén egyáltalán nem mindegy az asszociativitás. Ha jobbról, vagy balról végezzük el először az osztást 4 vagy 1 eredményhez jutunk!

2.10. Típusok

A *típus* egy olyan absztrakció, amely összefoglalja bizonyos entitások közös tulajdonságait: *kódolás, méret, szerkezet, szemantika*. Egy entitás típusa definiálja azt a halmazt, amelyből az entitás, mint változó, értékeket vehet fel, a memóriában lefoglalt hely méretét, és ugyanakkor definiálja azokat a műveleteket is, melyek az entitással elvégezhetők (szemantikai szinten). Az első programozási nyelvek típusként a primitív hardver típusokat használták, de támogattak néhány összetett típust is, azonban nem volt egy egységes ábrázolási mód kialakulva. A használt típuskonstrukciók függtek a hardvertől, a számítógép felépítésétől. Ezek a különbségek elsősorban a lebegőpontos számábrázolásban nyilvánultak meg.

Csak az 1960-as évek végén kezdték a típusokat absztrakcióként értelmezni, *C. Strachey* és *T. Standish* munkája hatására.

Egy típushoz *konstruktorok*at, *szelektorok*at és *predikátumok*at rendeltek. Ekkor születtek meg többek között a *Simula* és az *Algol68* nyelvek, amelyek már magasfokú típusszemlélettel bírtak: általános, nagy kifejezőerejű típusfogalmat használtak, szigorú *típusellenőrzés* mellett. Már ekkor kidolgozták és implementálták a *típusmegfelelési* (*kompatibilitási*, *compatibility*), *típuskiterjesztési* (*extension*) és *típusátalakítási* (*konverziós*, *conversion*) szabályokat.

Az *Ada83* nyelv újabb fontos mérföldkő volt a típusok terén. Definiálta a *típusértékhalmaz* és a *típusművelet* fogalmakat. Ezeken kívül lehetővé tette a típussal való paraméterezhetőséget, megszüntette a biztonsági réseket, megpróbálta szabályozni és explicitté tenni a szemantikailag szabálytalan programozási eszközöket.

A típusok elemzésekor a következő kérdésekre keresünk választ:

- Mit jelent a nyelvben az adattípus?
- Adattípusok szemantikája
- Írásra vonatkozó konvenciók
- Mik a beépített adattípusok?
  - Milyen elemi típusokkal rendelkezik a nyelv?
  - Milyen összetett típusokkal rendelkezik?
  - Szintaxis
- Szigorúan típusos-e a nyelv?
- Vannak-e a típusoknak egyéni jellemzői?
- Vannak-e beágyazott típusok?
- Rendezett típusok:
  - Kezdőértékük 0 vagy 1?
  - A logikai típus felsorolási típus-e, vagy önálló?
- Az egész és valós típusok számát a nyelv definíciója vagy az implementáció szabja meg?
- Milyen valós típusokat implementál a koprocesszor?
- Mutatók

    - Vannak-e típus nélküli pointerek?
    - Mire kellenek a mutatók?
        - Hogy ne kelljen nagy adatszerkezeteket átadni?
        - Dinamikus adatszerkezetek építéséhez?
        - Objektumorientált funkciókhoz?
    - Mutatóaritmetika
        - Mit jelent a „+” operátor mutatókra?
        - Hogyan történik a mutató és a mutatott objektumok értékadása?
        - Van-e, és mit jelent az egyenlőségvizsgálat?
    - Van-e mutató dereferencia művelet?
    - A szemétgyűjtés automatikus, vagy manuális?
    - Van-e sehová sem mutató mutató (nil, null)?
    - Lehet-e automatikus változóra pointer?
    - Lehet-e több mutató egy objektumra?
    - Van-e alprogramra mutató mutató?
- Lehet-e felhasználói típust deklarálni?
    - Szintaxis
    - Van-e altípus-definiálás?
    - Elérhetők-e az alábbi típuskonstrukciók, és ha igen, milyen tulajdonságokkal?
        - Tömb
            - Mi lehet az indexe?
            - Mi lehet az eleme?
            - Csak ugyanolyan típusú elemei lehetnek?
            - Van-e indextúlcsordulás-ellenőrzés?
            - Van-e kezdőérték-adás?
            - Van-e egyben értékadás?
            - Vannak-e dinamikus tömbök?
            - Vannak-e konstans tömbök?
            - Mikor dől el a mérete, a helyfoglalása?
            - Van-e többdimenziós tömb?
            - Van-e altömb (szelet) képzés?
            - Vannak-e speciális tömbök?
        - Rekord (direktszorzat)
            - Van-e kezdőérték-adás?
            - Van-e egyben értékadás?
            - Van-e rekord konstans?
            - Hogyan működik a kiválasztás művelet?
            - Vannak-e speciális rekordok?
        - Variáns rekord (unió)
            - Meg lehet-e állapítani, hogy a rekord melyik változat szerint volt kitöltve?
            - Ki lehet-e olvasni a kitöltésitől különböző változat szerint?
            - Vannak-e speciális variáns rekordok?
        - Halmaz
        - Állomány
            - Milyen állománytípusok vannak?
            - Milyen műveletek végezhetők állományokkal?
            - Vannak-e speciális állományok?
    - Vannak-e speciális típuskonstrukciók?
    - Vannak-e absztrakt adattípusok, típussablonok?
- Léteznek-e névtelen típusok?
- Létezik-e Variant típus?
- Mikor ekvivalens két típus?
    - Strukturális ekvivalencia esetén
        - A rekordok mezőneveit is figyelembe vették, vagy csak a struktúrájukat?
        - Számít-e a rekordmezők sorrendje?

        - Tömböknél elég-e az indexek számosságának egyenlőnek lenni, vagy az indexhatároknak is egyezniük kell?
    - Név szerinti ekvivalencia esetén
        - Deklarálhatók-e egy típushoz típusok, amelyekkel ekvivalens?
        - Névtelen tömb- illetve rekordtípusok ekvivalensek-e valamivel?
- Mikor kompatibilis két típus?
- Mi történik túlcsorduláskor?
- Típuskonverziók
    - Van-e, és hogyan működik
        - az automatikus konverzió?
        - az identitáskonverzió?
        - a bővítő konverzió?
        - a szűkítő konverzió?
        - a *toString* konverzió?

Érdekes a *Delphi Variant* típusa. A *Variant* típus olyan értékek tárolására szolgál, amelyeknek típusa nem ismeretes fordítási időben. Egy ilyen típusú változó tehát bármilyen értéket felvehet. A *Variant* típus 16 byte nagyságú helyet foglal a memóriában, ezen a helyen egy típusdeszkriptort és egy értéket, vagy egy mutatót egy értékre tárol. A *Variant* típus segítségével egész számokkal indexelhető dinamikus tömböket is létre lehet hozni. A tömbök elemei tetszőleges típusúak – akár tömbök is – lehetnek.

2.11. Utasítások, vezérlési szerkezetek

Az *utasítások* a program legalapvetőbb, algoritmikus részei. Az eredmény eléréséhez szükséges műveleteket – algoritmusokat – írják le. Az utasításokat általában kulcsszavak alkotják.

- Írásra vonatkozó konvenciók
- Egyszerű utasítások
    - Az értékadás egyszerű utasítás, vagy kifejezés utasítás?
    - Van-e többszörös értékadás?
    - Ki kell-e írni az üres utasítást?
    - Hogyan valósul meg az eljáráshívás?
    - Van-e ugróutasítás?
- Összetett utasítások, vezérlési szerkezetek
    - Van-e eseményvezérelt programozás?
    - Szekvencia
        - Terminátor vagy utasításelválasztó-e a „;” vagy más karakter?
    - Lehet-e blokkutasítást létrehozni és hogyan?
        - Lehet-e a blokk üres?
        - Elhelyezhető-e a blokkutasításban deklaráció?
        - Mi történik blokkból való kilépéskor?
    - Van-e makró-szubsztitúció, kódblokkok értelmezésére lehetőség?
    - Elágazás
        - Van-e kétirányú elágazás?
        - Hová tartozik az „else”?
        - Van-e aritmetikai elágazás?
        - Van-e többirányú elágazás?
        - Többirányú elágazás
            - Mi lehet a szelektor típusa?
            - Fel kell-e sorolni a szelektortípus minden lehetséges értékét?
            - Mi történik, ha fel nem sorolt értéket vesz fel a szelektor?
            - Rácsorog-e a vezérlés a következő kiválasztási ágakra is?
            - Diszjunktnak kell-e lennie a kiválasztási értékeknek?
            - Meg lehet-e adni intervallumot a szelektorértéknek?
            - Mi állhat a kiválasztási feltételben a következők közül?
                - egy érték
                - értékek felsorolása
                - intervallum
                - más is

- Ciklus
  - Vannak-e változó lépésszámú ciklusok?
    - Van-e elöl tesztelős ciklus?
    - Van-e hátul tesztelős ciklus?
    - A feltétel ciklusának logikai értéknek kell lennie, vagy más típusú is lehet?
    - Kell-e blokkot kijelölni a ciklusutasításnak?
  - Van-e fix lépésszámú ciklus?
    - A ciklusváltozó mely jellemzője állítható be a következők közül?
      - alsó érték
      - felső érték
      - lépésszám
    - Mi lehet a ciklusváltozó típusa?
    - Biztosított-e a ciklusmagon belül a ciklusváltozó változtathatatlansága?
    - A ciklushatárokat lehet-e dinamikusan változtatni?
    - Mi a ciklusváltozó hatásköre, definiált-e az értéke kilépéskor?
    - Lehet-e a ciklusutasításban ciklusváltozót deklarálni?
  - Van-e iterátor ciklus?
  - Van-e ciklusváltozó-iterátor?
  - Van-e általános ciklus, és hogy néz ki?
  - Léteznek-e a következő vezérlésátadó utasítások?
    - break
    - continue
    - kivételek
- Van-e hivatkozás utasítás?
- Vannak-e más, speciális utasítások?
- Az utasítások szintaxisa

Az üres utasítás jelölésére *COBOL*ban a ***NEXT SENTENCE***-t, *Pascal*ban a „;”-t, *Pyton*ban a ***pass*** kulcsszót használjuk akkor, ha szintaktikailag szükség van egy utasításra, de a programban nem kell semmit sem csinálni.

C++-ban a blokk fogalma sokkal többet fed, mint *Pascal*ban. A *Pascal* blokkdefinícióján kívül a következő elemeket tartalmazza: egy blokkon belül deklarált változó lokális az illető blokkra nézve; egy blokkból való kilépés alkalmával automatikusan meghívódik az összes blokkon belül használt objektum destruktora. *Ada*ban minden blokk elején újabb változókat deklarálhatunk, ezeknek külön cikkely van fenntartva, amit a ***declare*** kulcsszó vezet be.

Az elágazási utasítások valósították meg először a futás pillanatában történő döntést bizonyos feltételek függvényében. Ennek a megvalósításnak köszönhető, hogy ugyanaz az algoritmus különböző bemeneti értékek illetve részeredmények alapján önmagából más-más lineáris utasítássorozatot hajtson végre. Ettől az újítástól vált a lineárisan programozható algoritmust végrehajtó gép számítógéppé. Ez a megvalósítás Neumann Jánosnak tulajdonítható. Az első magas szintű nyelvben megjelent elágazás a *FORTRAN*-beli aritmetikai ***IF***: ***IF*** *(AritmetikaiKifejezés) E1, E2, E3*. Az elágazás az *AritmetikaiKifejezés* értékétől (negatív, nulla, pozitív) függ, és ennek alapján a programban az *E1, E2* vagy *E3*-as címkékre történik ugrás.

COBOL-ban a ciklusmag számára külön blokkot, alprogramot (paragrafust) kellett írni, és ezt a ***PERFORM*** utasítással lehetett meghívni. A *C#* bevezeti az *iterátor ciklust* is. Ezáltal lehetőség van olyan számlálásos, ciklusváltozóval ellátott ciklus megszervezésére, ahol a ciklusváltozó rendre felveszi egy előre megadott, felsorolható halmaz elemeinek értékeit (***foreach***). A *Pyton* érdekessége még, hogy a ciklus utasításoknak lehet egy ***else*** águk is. Ez az ág akkor hajtódik végre, ha a ciklus végighaladt a listán (***for*** esetén), illetve ha a feltétel hamissá vált (***when*** esetén), de nem hajtódik végre, ha a ciklust a ***break*** utasítással szakítottuk meg.

2.12. Kivételkezelés

A kivételek (*exception*) olyan hibás események, amelyek megszakítják az alkalmazás szabályszerű futását. Ilyenkor a vezérlés a kivételkezelőnek adódik át. A kivételkezelés nem egyszerű feladat, hiszen alkalmazásunk minden egyes forrássora potenciális hibaforrás is egyben. Ha már egy hibával szembekerültünk, célszerű azt kezelnünk, vagyis olyan tevékenységeket végeznünk, amelyekkel a hibák hatását eltüntethetjük vagy legalább „enyhíthetjük”. Ha egy hibát nem sikerül kezelnünk, szeretnénk annak helyéről, körülményeiről mindent tudni.

- Hiba- vagy kivételkezelést ad-e a nyelv?
- Milyen beépített kivételek vannak?
- Definiálhatunk-e saját kivételt?
- Milyen kivételkezelők vannak?
    - Ha kivétel lép fel, akkor...
    - Mindenképpen el kell végezni...
- Kivételkezelők szintaxisa
- Milyen programelemekhez köthető a kivételkezelő?
- Milyen hatáskörrel, élettartammal rendelkezik a kivételkezelő?
- Többszörös kivételek
- Hogyan folytatódik a program kivételkezelés után?
- Vannak-e beágyazott kivételkezelők?
- Van-e általános kivételkezelő?
- Van-e automatikus kivételkezelő?
- Párhuzamos környezetben vannak-e speciális kivételek?

2.13. Programegységek

A programozási nyelvek lehetőséget biztosítanak a programok bizonyos egységekre (*fordítási egységek*) való felosztására, klasszifikálására. Az utasításokat, műveleteket és adatokat tehát nem ömlesztve tartalmazza egy-egy forrásszöveg-állomány, hanem ezek valamilyen logikai vagy a programozó által meghatározott sorrendet követve rendezhetők a nyelv szintaxisának megfelelő állományokba. Ezek az állományok lehetnek moduláris egységek, vagyis külön-külön is van értelme mindegyiküknek, egymástól független egységek, vagy lehetnek olyan egységek, amelyek egymagukban semmit sem jelentenek, csak közös fordítás és láncolás után lesz meg az igazi értelmük.

- A program felépítése, hogy néz ki a főprogram?
- Minimális főprogram
- Milyen moduláris egységek léteznek, és ezek hogy néznek ki?
- Minimális egységek
- Az egységek szintaxisa
- Írásra vonatkozó konvenciók
- Létezik-e átlapoló egység (Overlay)?
- A vizuális elemek hogyan kötődnek az egységekhez?
- Lehet-e forrásszöveget inkludolni?
- Létrehozhatók-e DLL-ek, és hogyan?
- Lehet-e erőforrásokat (Resource) használni?
- Lehet-e külső OBJ állományt a programhoz szerkeszteni?
- Lehet-e más programozási nyelvben megírt alprogramokat használni?
- Vannak-e speciális egységek?

Az átlapoló egységek egymástól függetlenül végrehajtható programrészeket tartalmaznak. A memóriába egyszerre csak egy átlapoló rész töltődik be, és végrehajtás után felszabadul. A magasabb szintű programozási nyelvek megengedik az átlapoló egységek megírását. Lássuk, hogy valósul ez meg Borlad Pascalban: Az átlapoló egységek írása az Overlay unit (OVERLAY.TPU) használatával történik. Ez az egység tartalmazza az átlapolást kezelő függvényeket, eljárásokat, szimbólumokat. A {$O} direktíva engedélyezi vagy letiltja az átlapolásos kód generálását. A {$O EgységNév} direktíva pedig egy egységet egy overlay ágba irányít. Az OVR állományt az EXE állományhoz lehet másolni a copy DOS paranccsal (copy /b nev.exe+nev.ovr nev.exe), ha az Options / Debugger menüpontból a Standalone Off állapotban van és az OvrInit paramétere a ParamStr(0), vagyis az EXE állomány teljes elérési útvonala. Overlayt használó programot csak lemezre lehet fordítani.

2.14. Absztrakciós szintek

Az absztrakciós szintek a nyelv modularitását, strukturálhatóságát, a procedurális absztrahálás megvalósíthatóságát célozták meg. A procedurális absztrahálás egyike a legrégibb programozási eszközöknek, Charles Babbage már 1840-ben azt tervezte, hogy lyukkártyák egy csoportját fogja használni nagyobb számítások gyakrabban használt részeinél.

- Vannak-e alprogramok (eljárások, függvények)?
- Vannak-e függvények?
- Van-e különbség eljárás és függvény között?
- Írásra vonatkozó konvenciók
- Ki kell-e írni az üres paraméterlistát határoló jeleket?
- Hívási konvenciók
- Verem felépítése
- Paraméterátadás sorrendje
- Lehet-e alprogramokat egymásba ágyazni?
- Mik a láthatósági és a beágyazási területek?
- Hány belépési pontja lehet egy alprogramnak?
- Vannak-e korutinok?
- Engedélyezettek-e a mellékhatás eljárások, függvények esetén?
- Rekurzív hívások
- Megadhatók-e elő- és utófeltételek?
- A függvényeknek milyen visszatérési értékeik lehetnek, és hogyan jelöljük a visszatérést?
- Milyen paraméterátadási módokkal rendelkezik a nyelv?
- Vannak-e alapértelmezett értékek?
- A paraméterlista mérete lehet-e változó?
- Jó-e és meghatározható-e a formális-aktuális paraméterek közötti információáramlás?
- Az alprogram neve vagy szignaturája azonosítja ezt?
- Definiálhatók-e operátorok, ezek átlapolhatók-e?
- Léteznek-e sablonok?
- Lehet-e alprogrammal paraméterezni?
- Használhatók-e az alprogramok változókként?
- Lehet-e típussal paraméterezni?
- Van-e lehetőség generikus programozásra?
- Hogyan néznek ki a ki/bemeneti (I/O) műveletek?
- Van-e beágyazott assembly?
- Van-e beágyazott gépi kód értelmező?
- Kapcsolat az API-val
- Vannak-e más beágyazott lehetőségek?
- A kód újrafelhasználhatósága
- Ha hibrid nyelv, hogyan keverhetők a különböző paradigmák?
    - Imperatív
    - Objektumorientált
    - Funkcionális
    - Logikai
    - Párhuzamos és osztott
    - Vizuális
    - Ötödik generációs

Számos programozási nyelv nem tesz különbséget eljárás és függvény között. Például *C*, *C++*, *Java*, *C#*, ezekben a nyelvekben minden alprogram függvény – ha a visszatérési érték típusa üres (*void*), ezek eljárásoknak tekinthetők, de a nem üres típusú visszatérési értékű függvények is hívhatók egyszerű eljárásként. Más nyelvekben (pl. *Ada*, *Pascal*) éles a különbség az eljárás és a függvény között, olyanannyira, hogy külön kulcsszóval kell deklarálni őket.

Egyes programozási nyelvek esetében (pl. *C*, *C++*, *Java* stb.) az üres paraméterlistát határoló zárójeleket is ki kell tenni az alprogram neve után, más nyelvekben (*Pascal*, *Ada*) ezt nem szabad, vagy nem feltétlenül kell (*PL/I*) kitenni.

A koritunok olyan speciális alprogramok, amelyek szakaszosan adhatják át egymásnak a vezérlést. Egy korutin meghívhat egy másik korutint saját maga befejezése előtt, ekkor mindkettő szakaszosan fog futni. Másodszori meghívásnál ott fogja folytatni tevékenységét, ahol először abbahagyta – így a párhuzamosság látszatát kelti. A korutinok tehát tetszés szerint adogathatják át egymásnak a vezérlést, nincs külön hívási vermük, ezért a korutinok hívását inkább *folytatás*nak (*resume*) szokás nevezni. Kevés nyelv támogatja a korutinokat, pl. *SIMULA*, *Modula-2*. Korutinokat általában a ***coroutine*** kulcsszóval lehet deklarálni, és élet-

ciklusukban létezik két fontos pillanat: az első a *detach*, mikor az új korutint leválasztjuk a régiről (***detach***), a második a *transfer*, amikor átadjuk vagy visszaadjuk a vezérlést (***transfer***).

2.15. Végkövetkeztetések

Az elemzés utolsó szakaszában a megismert programozási nyelvről keltett benyomásainkat összegezzük:

- Megoldatlan problémák
- Vélemények a nyelvről
- Erőssége
- Gyengesége
- Továbbfejleszthetőség
- Megbízhatóság
- Általánosság

Könyvészet

[1.] Fischer, Alice E.; Grodzinsky, Frances S.: *The anatomy of programming languages*, Prentice-Hall, 1993.
[2.] Horowirtz, Ellis: *Fundamental Concepts of Programming Languages*, Computer Science Press, division of W.H. Freeman, New York, 1983.
[3.] Horowitz, Ellis (ed.): *Programming Languages: A Grand Tour*, Computer Science Press, Rockville, Maryland, 1984.
[4.] Horowitz, Ellis: *Magasszintű programnyelvek*, Műszaki Könyvkiadó, Budapest, 1987.
[5.] Pârv, Bazil; Vancea, Alexandru: *Fundamentele limbajelor de programare*, Ed. Albastră, Kolozsvár, 1996.
[6.] Sammet, J.: *Programming Languages: History and fundamentals*, Prentice Hall, Englewood Cliffs, NJ, 1969.
[7.] Scott, M.K.: *Programming Languages Pragmatics*, Morgan Kaufmann, San Francisco, 2000.
[8.] Sethi, R.: *Programming Languages: Concepts and Constructs*, Addison-Wesley, Reading Mass., 1996.
[9.] Bíró Ernő; Kovács Lehel: *A programozási nyelvek alapjai*, Komp-Press, Kolozsvár, 1997.
[10.] Kovács D. Lehel István: *Programozási nyelvek összehasonlító elemzése – Az objektumorientált paradigma*, Presa Universitară Clujeană, Cluj-Napoca, 2001.
[11.] Nyékyné Gaizler, Judit (ed.): *Programozási nyelvek*, Kiskapu Könyvkiadó, Budapest, 2003.
[12.] Kovács D. Lehel István: *Programozási nyelvek összehasonlító elemzése – A programozási nyelvek anatómiája*, Presa Universitară Clujeană, Cluj-Napoca, 2000. ISBN: 973-8095-63-8. p. 264., 2004.