15 - Programok fordítása és végrehajtása

Fordítás és interpretálás, bytecode. Előfordító, fordító, szerkesztő. A make. Fordítási egység, könyvtárak. Szintaktikus és szemantikus szabályok. Statikus és dinamikus típusellenőrzés. Párhuzamos programozás.

1. Fordítás és interpretálás

Fordítás esetén a forráskódot a **fordítóprogram tárgyprogram**má alakítja. A tárgyprogramokból a **szerkesztés** során futtatható állomány jön létre. A fordítási és a futtatási idő élesen elkülönül.
Fordítható nyelvek például a C/C++, Ada.

Interpretáláskor a forráskódot az **interpreter** értelmezi, és azonnal végrehajtja. Ebben az esetben a fordítási és futási idő nem különül el.
Jellemezően a scriptnyelvek interpretálhatóak, például a JavaScript, Perl, PHP.

A fordítás előnye a gyorsabb végrehajtás és a forrás alaposabb ellenőrizhetősége. Hátránya azonban, hogy a forrásprogramot minden platformra külön le kell fordítani. Az interpretálás mellett szól még a rugalmasság (utasítások akár fordítási időben is összeállíthatóak), és hogy minden platformon azonnal futtatható, ahol megtalálható az interpreter.

Egyes nyelvekhez fordító és interpreter is létezik, ilyen például a Haskell.

2. Bytecode

A fordítás és interpretálás előnyeinek egyesítése és hátrányaik elhagyása érdekében a két eljárás „keresztezhető”. Ebben az esetben a fordítóprogram a forráskódot **bájtkód**ra fordítja le, amely a programozási nyelvhez tartozó **virtuális gép** gépi kódja. Az így létrejött program ezután minden platformon interpretálható, ahol megtalálható a virtuális gép.
Ilyen nyelvek például a Java és a C#.

3. Előfordító

Az előfordító (*preprocessor*) különböző szöveges változtatásokat hajt végre a forráskódon, és ezzel előkészíti azt a tényleges fordításra.

Feladatai közé tartozik általában többek között:

- a kommentek eltávolítása;
- meghívott fájlok beágyazása;
- makrók kiértékelése;
- szimbólumok kezelése, feltételes fordítás;
- a forrásfájlban fizikailag több sorban elhelyezkedő forráskód logikailag egy sorba történő csoportosítása – ha szükséges.

A programozó az előfordítóval általában **direktívák** útján közli a végrehajtandó tevékenységeket. Ezek a direktívák például C/C++ nyelven # karakterrel kezdődnek.

4. Fordító

A fordítási folyamat magába foglalja az előfordító által módosított forrásprogram **lexikális**, **szintaktikus** és **szemantikus** elemzését, a **kódgenerálás**t és a **kódoptimalizálás**t. A fordító (*compiler*) forráskódból **fordítási egység**enként a gép számára értelmezhető gépi kódot generál, ezek **a tárgyprogram**ok.

A fordítás egyes lépései általában nem különülnek el élesen, tehát például a lexikai elemző nem generál le egy különálló fájlt, hanem a kimenetét azonnal átadja a szintaktikai elemzőnek.
Igaz ez rendszerint az előfordító és a fordító kapcsolatára is.

5. Szerkesztő

A szerkesztő (*linker*) feladata, hogy a fordítási egységekből külön lefordított tárgyprogramokat (*object fájlokat*) egyetlen futtatható állománnyá összeszerkessze.

A szerkesztés fajtái.

6. A make

A ***make*** eszköz célja a nagyobb programok fordításának automatizálása. A *make* a specifikáció alapján meghatározza, hogy mely részeket kell újrafordítani, majd meghívja a megfelelő parancsokat.
A *make* használatához először létre kell hozni a ***Makefile*** -t, ami leírja az állományok közötti függőségeket, és a fájlok frissítésére szolgáló parancsokat. Egy programban például a végrehajtható fájl rendszerint az *object fájl*októl függ, azok pedig a forrás fájloktól.

A *make* program meghívása: `make`
Ekkor alapértelmezésként az aktuális könyvtárban található *Makefile* nevű fájlt kerül végrehajtásra.
Ha másik *Makefile*-t szeretnénk használni: `make -f Masik_Makefile`

A következőekben tekintsük át *Makefile* legfontosabb elemeit.

6.1. Explicit szabályok

Ezek határozzák meg, hogy mikor és hogyan kell lefordítani egy vagy több állományt. A szabályok három részből állnak: a **cél állomány**ból, a **függőségi listá**ból (azok az állományok, amelyektől a cél függ) és a **parancs**ból (fordító utasítás). A *make* a parancsot végrehajtja, ha a célállomány nem létezik vagy a függőségi listában van olyan állomány, amelyet később módosítottak, mint a cél állományt, ezáltal szükséges az újrafordítás. Az explicit szabály alakja:

```
Célok: függőségi lista; Parancs
```

A parancs vagy a függőségi listával van egy sorba pontosvesszővel elválasztva, vagy új sorba, de utóbbi esetben tabulátor karakterrel kell a sornak kezdődnie.

6.2. Implicit szabályok

Egy olyan állományra, amely szerepel valamelyik szabály feltételei között, de nem szerepel szabály céljaként, megpróbál alapértelmezett szabályt találni a *make*.
(Például a valami.o állományt a valami.cpp állományból lehet lefordítani.)

6.3. Változó definíciók

Gyakorlatilag makró helyettesítés történik.
Változó definiálása: `VÁLTOZÓ = ÉRTÉK`
Hivatkozás a változó értékére: `$(VÁLTOZÓ)`

6.4. Direktívák

Az előfordítási direktívákhoz hasonlóak, lehetőség van például más állományok beágyazására, feltételes fordításra, stb.

6.5. Kommentek

A # jel utáni karakterek az adott sor végéig.

Bővebben >>

7. Fordítási egység

Fordítási egység a nyelvnek az az egysége, amely a fordítóprogram egyszeri lefuttatásával, a program többi részétől elkülönülten lefordítható. Ha programunkat fordítási egységekre tagoljuk, akkor elkerülhetjük azt, hogy egyetlen kisebb módosítás miatt a teljes programot újra kelljen fordítani.

Mivel a fordító a neki átadott forrásfájlokat teljes egészében feldolgozza, ezért a fordítási egységeket úgy tudjuk kialakítani, hogy a forrásprogramot nem egy fájlban helyezzük el, hanem fordítási egységenként tagoljuk. Ezzel a módszerrel egyből a forráskód logikai tagolása is megtörténik, így a forrásszöveg könnyebben áttekinthetővé és megérthetővé válik.

A különböző fordítási egységek összekapcsolásáért a szerkesztő felel. A szerkesztő számos hibát, következetlenséget képes észrevenni, így a program nem lesz kevésbé megbízható amiatt, hogy nem egy forrásfájlból áll az egész. Lehetőség van arra is, hogy ne szerkesszünk minden fordítási egységet bele a futtatható állományba, hanem a szerkesztés dinamikusan történjen, így megosztott könyvtárakat tudunk létrehozni. A szerkesztés fajtái:

- **Statikus**: az *object fájlok*at fordítási időbe összeszerkesztjük a könyvtárakkal.
- **Dinamikus, betöltéskor** (*load-time*): fordítási időben úgynevezett import könyvtárakat használunk, ezek a megosztott könyvtárakra vonatkozó hivatkozásokat tartalmaznak, amiket majd az operációs rendszer a program betöltésekor kapcsol hozzá a futtatható fájlhoz. Ha valamelyik hivatkozott megosztott könyvtár hiányzik, a programot nem lehet betölteni.
- **Dinamikus, futtatáskor** (*run-time*): fordítási időben a megosztott könyvtárak betöltésére és az eljárások címeinek lekérdezésére vonatkozó rendszerhívások kerülnek a programba. A megosztott könyvtárak betöltése futás közben történik, amikor szükség van rájuk. Ezzel a megoldással lehetőség van arra, hogy a program a neki megfelelő verziójú könyvtárat megkeresse, vagy például a program indításkor ellenőrizze, hogy van-e egyáltalán ilyen.

8. Könyvtárak

A programkönyvtárnak az olyan alprogram-, modul-, osztály-, illetve adattípus-gyűjteményeket tekintjük, amelyek egy jól körülhatárolható szolgáltatáscsoportot megvalósító programkódot tartalmaznak, és egységes felületet (**interfész**t) biztosítanak a felhasználó programozók számára.

Mivel ezeket a programkódokat más, nagyobb programokba beépítve használják, fontos, hogy rendelkezzenek a következő tulajdonságokkal:

- helyesség (pontosan megoldja a feladatot),
- hatékonyság (gyors futási idő és kevés memória igény, bár e kettő egymás ellen dolgozik),
- megbízhatóság (rossz bemenet esetén a megfelelően kezelje azt),
- kiterjeszthetőség (továbbfejlesztése, módosítása egyszerű legyen),
- újrafelhasználhatóság (minél általánosabb legyen, így több feladat megoldására használható),
- jól dokumentáltság.

9. Szintaktikus és szemantikus szabályok

A szintaktikus szabályok a **környezetfüggetlen**, a szemantikus a **környezetfüggő** ellenőrzést teszik lehetővé. Környezetfüggetlen ellenőrzés a *token*ek érvényes sorrendjének ellenőrzése például, környezetfüggő pedig a típusellenőrzés többek között.

A szintaktikai elemzéshez *Chomsky 2*, a szemantikaihoz *Chomsky 1* féle nyelvtan szükséges.

10. Statikus és dinamikus típusellenőrzés

A típusellenőrzés az az eljárás, ami vagy fordítási időben (statikus ellenőrzés) vagy végrehajtási időben (dinamikus ellenőrzés) ellenőrzi típuskényszerítés szabályait, és szükség esetén végrehajtja a előírt művelet(ek)et.

A statikus ellenőrzés a fordítóprogram feladata. Ha a nyelv kikényszeríti a típushoz tartózó szabályok végrehajtását (ez általában a típuskonverziók végrehajtását jelenti, lehetőleg információvesztés nélkül), akkor a nyelv **erősen típusos**, ellenkező esetben **gyengén típusos**.

Statikus típusellenőrzés esetén az ellenőrzések fordítási időben történnek, így futás közben csak az értékeket kell tárolni. Ennek előnye a biztonságosság, mert futás közben nem történhet probléma.
Statikus típusellenőrzést használ például az Ada, C++, Haskell.

Dinamikus típusellenőrzésnél futási időben történik az ellenőrzés, így futás közben az értékek mellett a típusinformációkat is tárolni kell. A típusokat minden utasítás végrehajtása előtt ellenőrizni kell, típushiba esetén futási idejű hiba keletkezik. A dinamikus ellenőrzés előnye a hajlékonyság.
Dinamikus típusellenőrzést használ például az Lisp, Erlang.

Bizonyos feladatokhoz muszáj használni a dinamikus típusellenőrzés technikáit, ilyen például az objektumorientált nyelvekben a dinamikus kötés.

11. Párhuzamos programozás

A párhuzamos programok előnye, hogy több processzormaggal rendelkező számítógép esetén annak számítási kapacitásait jobban ki tudja használni, ezzel nagyobb hatékonyság, jobb erőforrás-kihasználtság érhető el, vagy több feladat futtatható egyszerre (felhasználó számára kényelmes).
Bizonyos feladatokat eleve könnyebb párhuzamos modell alkalmazásával elkészíteni.

Programunkban a párhuzamosságot több folyamat vagy több szál létrehozásával érhetjük el.
A **folyamat** az operációs rendszer szempontjából egy végrehajtás alatt álló program. Minden folyamathoz egy logikai processzor és egy logikai memória tartozik. A logikai memória tárolja a programkódot, a változókat és a konstansokat, a logikai processzor hajtja végre a programkódot.
Az operációs rendszer felelős a logikai processzor és a logikai memória fizikai processzorra és memóriára való leképezéséért. Multiprocesszoros rendszer esetén a logikai processzort több fizikai processzorra képezheti le. Az összefüggő logikai memóriát pedig a fizikai memória több nem összefüggő része vagy a háttértár szolgáltathatja.

A **szál**ak ezzel szemben egy folyamaton belül párhuzamosan futó programrészek. Minden szálnak saját logikai processzora van, és a folyamatokhoz hasonlóan versenyezhetnek a processzorért. A szálak logikai memóriája ezzel szemben közös, tehát az egy folyamaton belüli összes szálnak egyetlen logikai memóriája van.
A szálak alkalmazásának előnye, hogy a közös memória miatt szálak közötti átkapcsolás lényegesen gyorsabb, mint a folyamatok közötti átkapcsolás. Ez az előny azonban különös körültekintést igényel: amennyiben több szál egyszerre próbálja ugyanazt az erőforrást elérni, akkor az úgynevezett szál-keresztezési probléma lép fel, és a programban futási idejű hiba keletkezhet.