volnék én, a nemes, a tündöklő, az értékek értéke, akkor hamarosan egy sárgöröngyöt választanátok s amiatt zsigerelnétek egymást. Akarva, nem-akarva érettem hajszolódtok, amíg éltek, sőt újabban a modern halottak még a sírba is elvisznek magukkal és miután a férgek leharabdálták ajkaikat, aranyfogaikkal vigyorognak az enyészetre. Csak az vethet meg, aki a becsvágyat megveti. Engem keresve találtátok meg az utat az igaz tudomány és élet felé. Nem is aljas fulajtárjaimra vagyok büszke, hanem erre. Meg arra, hogy egy költőt, aki maga a tökéletesség, az ami én a nemesfémek között, rólam neveztek el.

Ezüst

Én az arany szegény testvére, fáradt
Lelkemből a múlt halvány fénye árad.
Olyan vagyok, mint az ég késői ősszel,
Mint könny a csipkén, a holdfény s az ősz fej.
Emlékezem csupán, de már nem élek.
Ezért zenélek.

Platina

– Igénytelen külsőmmel többnyire vegykonyhák poros asztalán húzódom meg, olvaszthatatlan, tűzálló tégely alakjában. A legnagyobb szerelem hőfokát is állom. Mikor újabban a poklot a mai kor igényeihez képest átalakították, lemezeimmel vonták be padlóját, mennyezetét, falait. *Dante* még nem tudta, de elárulhatom, hogy azok az amerikai bankárok, akik a hamisbukás következményei elől repülőgépen menekülnek, platinakamrákban fognak égni örökkön-örökké.

Gyémánt

– Adamas! A legyőzhetetlen, a legkeményebb! Mindenkit megkarcolok. Ki karcol meg engem? Spinoza valaha Amszterdamban próbált köszörülni, kicsiszolni, de nem birt velem. Én köszörültem, csiszoltam ki őt. Tőlem tanulta gyémántlogikáját. Királyi koronákban villogok. Dollár-milliomosnők aszott keblén, hervatag fülcimpáiban, de mindig közönyösen, függetlenül tőlük, rájuk se hederítve. Másnap, hogyha elárverezik ékszerüket, éppúgy szikrázom egy ügynök sötét páncélszekrényében, hol nincs is közönségem. Az embereket túlélem. Századokon át folyton gazdát cserélek. Természetem a hűtlenség. Csak egyhez vagyok hű. A szegény üvegestóthoz. Annak kopott bőrtáskájában mindig találok egy darab gyémántkőt, mert ez az ablakvágó szerszámához szükséges. Hitvány gyémántrögöcske ez, igaz. De örökre nála marad.

Fordítóprogramok szerkezete

avagy

Mi történik Pascalban mikor F9-et nyomunk?

A magas szintű programozási nyelvek megjelenése maga után vonta a fordítóprogramok elméletének kialakulását. A számítógépek csak a gépi kódot fogadják el alapnyelvként. A gépi kód a processzor belső utasításkészlete, vagyis egyesek és nullák sorozata. A magas szintű nyelvekben való programozás szükségszerűvé teszi egy olyan program használatát, amely áthidalja az illető nyelv és a gépi kód közötti különbséget, vagyis az illető nyelv utasításait (forráskód, forrásprogram) gépi kódú utasításokra vagy valamilyen értelmezhető formára (tárgykód) fordítja. Ezt a fordítást háromféleképpen végezhetjük el.

Az első módszer az, hogy a tervezett nyelvben megírt programot lefordítjuk a számítógép gépi kódjára. A fordítást megvalósító program a *fordítóprogram* (*compiler*). Például *Pascal*, *C* nyelvek esetében.

A második módszer az, hogy készítünk egy olyan egységet, virtuális gépet, amely a tervezett nyelvet értelmezi. Ezt a módszert értelmezésnek, a megvalósítóját pedig *értelmezőnek* (*interpreter*) nevezzük. Például *FoxPro*, *BASIC* nyelvek esetében. Az értelmezőt hardver útján is meg lehet valósítani, ekkor *formulavezérlésű számítógépekről* beszélünk. Például készítettek olyan hardvert, amely a *Java* nyelv köztes kódját tudja értelmezni.

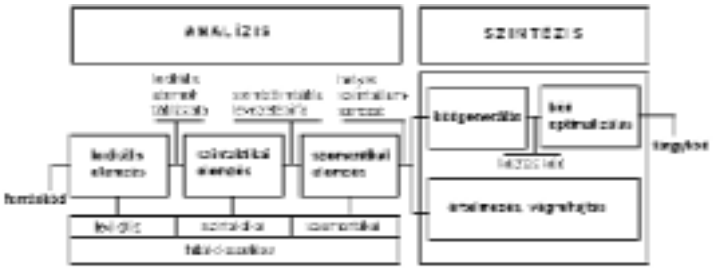
A harmadik módszer az, hogy a tervezett nyelvben megírt programot egy hasonló elvek szerint működő, már megírt, magas vagy alacsony szintű nyelvre fordítjuk és a későbbi műveleteket ezen nyelv fordítóprogramjára vagy értelmezőjére bízzuk. Egy ilyen fordítóprogramot *átalakítónak* (*translator*) nevezünk. Például a *Java* nyelv.

Elméleti szempontból az első és a harmadik módszer között nincs lényeges különbség, ezért a két módszert együtt vizsgáljuk.

A fordítóprogramok működése elméleti szempontból két különálló egységre bontható: az *analízisre* és a *szintézisre*. Az analízis fázisban a forrásszöveg kerül „elemzőasztalra", és három elemzési szempont (lexikális, szintaktikai és szemantikai elemzés) szerint vizsgáljuk a program helyességét. A szintézis fázisban a fordítóprogram kódot generál és optimalizál. Ha fordítóprogramról vagy átalakítóról beszélünk akkor a tárgykód a gépi kód vagy valamilyen köztes kód, ha értelmezőről beszélünk akkor a generált kód azonnal végre is hajtódik.

A különféle programozási nyelveknek tehát semmi értelmük nem lenne fordítóprogramok nélkül.

A fordítóprogramok működése tehát így ábrázolható:



Azt az időintervallumot, ami alatt a fordítás történt *fordítási időnek*, azt az időintervallumot, ami alatt a generált tárgykódot futtatjuk, *futási időnek* nevezzük. Fordítás esetén a fordítási idő és a futási idő teljesen elkülönített, diszjunkt időintervallumok. Értelmezés esetén a megfelelő elemzések után a beolvasott, helyes szimbólumokat azonnal kiértékeljük. A fordítási idő tehát egybeesik a futási idővel.

Parancssoros fordítók, a környezet fogalma

A fordítóprogramok kezdetben *parancssorosak* voltak, ez azt jelentette, hogy egyszerű utasításként meghívtuk őket az adott számítógép operációs rendszerének parancsértelmezőjében, megadtuk paraméterként a kívánt forráskódot tartalmazó állományt, esetleg kapcsolódirektívák segítségével különféle opciókat állíthattunk be, majd eredményként megkaptuk a tárgykódot, ha a forráskód helyes volt. Ha ez helytelen volt, akkor a fordítóprogram hibaüzenettel tért vissza, megjelölve a hiba előfordulási helyét is.

A forráskódot tartalmazó állományt valamilyen szövegszerkesztőben kellett elkészítenünk, hasonlóan, valamilyen szövegszerkesztő segítségével kellett kijavítani a hibákat is. A hibajavítás után megint meghívtuk a parancssoros fordítóprogramot mindaddig, míg meg nem kaptuk a tárgykódot. Gyakran megesett az is, hogy az egyes modulokat külön, önállóan kellett lefordítani, és ezután a lefordított köztes kódokat össze kellett szerkeszteni egy *linker* segítségével. A teljes fordításhoz pedig – hogy helyzetünket megkönnyítse – egy fordító szkriptet kellet írni.

A következő példa a *Turbo Pascal* parancssoros fordítóját mutatja be (*tpc.exe*).

```
Turbo Pascal  Version 7.0  Copyright (c) 1983,92 Borland International
Syntax: TPC [options] filename [options]
  -B = Build all units                -L = Link buffer on disk
  -D<syms> = Define conditionals      -M = Make modified units
  -E<path> = EXE/TPU directories      -O<path> = Object directories
  -F<seg>:<ofs> = Find error          -Q = Quiet compile
  -GD = Detailed map file             -T<path> = TPL/CFG directory
  -GP = Map file with publics         -U<path> = Unit directories
  -GS = Map file with segments        -V = Debug information in EXE
  -I<path> = Include directories      -$<dir> = Compiler directive
Compiler switches: -$<letter><state>  (defaults are shown below)
  A+ Word alignment       I+ I/O error checking   R- Range checking
  B- Full boolean eval    L+ Local debug symbols  S+ Stack checking
  D+ Debug information    N- 80x87 instructions   T- Typed pointers
  E+ 80x87 emulation      O- Overlays allowed     V+ Strict var-strings
  F- Force FAR calls      P- Open string params   X+ Extended syntax
  G- 80286 instructions   Q- Overflow checking
Memory sizes: -$M<stack>,<heapmin>,<heapmax>  (default: 16384,0,655360)
```

A fordítás pedig így történt:

```
C:\Apps\BP\BIN>tpc program.pas
Turbo Pascal  Version 7.0  Copyright (c) 1983,92 Borland International
PROGRAM.PAS(2)
2 lines, 1472 bytes code, 668 bytes data.

C:\Apps\BP\BIN>
```

Hasonlóan működik az *Assembly* nyelv fordítója is, itt azonban a szerkesztést külön lépésben kell elvégezni: a *tasm.exe* fordító segítségével az ASM állományt egy köztes kódra, OBJ állományra tudjuk lefordítani, majd egy vagy több OBJ állományt a *tlink.exe* szerkesztő segítségével tudunk összeszerkeszteni, futtatható COM vagy EXE állománnyá alakítani. Nagyobb projekt esetében, ha több modullal dolgozunk, a fordításra egy külön BAT állományt is kell írnunk, amely megkönnyíti a hívásokat.

Napjaink tendenciája, hogy a fordítóprogramokat *környezettel* lássuk el, mely integrálja a különböző elemeket. Legfontosabb kritérium, hogy a környezet egy szövegszerkesztővel rendelkezzen, amelyben meg tudjuk írni a forráskódot, közvetlenül lehessen hívni a fordítóprogramot vagy a szerkesztőt, a környezet tartalmazzon egy jól megírt kontextusfüggő súgórendszert is (*help*), amely a nyelvleírást és az egyes modulok, eljárások, függvények stb. bemutatását tartalmazza lehetőleg sok példaprogrammal.

Gondoljunk csak a *Turbo Pascal 6.0* környezetére, mennyire kényelmesebb benne dolgozni, mint parancssoros fordítás esetén.

Ezeket a környezeteket *IDE*-nek (*Integrated Development Environment*), *beágyazott fejlesztési környezeteknek* nevezzük.

Egy modern fordítóprogram környezete a következő elemeket tartalmazza:

- Szövegszerkesztő
- Fordítórendszer
- Szerkesztőrendszer (linker)
- Futtatórendszer
- Súgó
- Kódkiegészítők, sablonok
- Varázslók, kódgenerátorok
- Tervezőfelület (vizuális tervezés elősegítése: folyamatábrák, UML tervezési lehetőségek stb.)
- Projekt kezelése, egyszerre több forráskód-állomány szerkesztése
- Debugger, nyomkövető (töréspontok definiálása, részletes futtatás, változók értékeinek nyomon követése, kifejezések kiértékelése stb.)
- Szimbólumkövető
- Verem, regiszterek tartalmának kijelzése, gépi kód
- Adatbázis-tervező (relációk megadása)
- Csoport- és nemzetközi programozás támogatása
- Automatikus dokumentációkészítő
- Tennivalók listája (ToDo)
- Más környezeti eszközök, beágyazott lehetőségek (pl. ikon rajzolóprogramok stb.)

Elemzések

A fordítás első feladata a forrásszöveg beolvasása. Ez a leggyakoribb esetben valamilyen szövegszerkesztővel megírt fizikai állományként van jelen a háttértárolón, vagy környezet esetében a memóriában. A szépség és áttekinthetőség kedvéért a forrásszöveg megjegyzéseket, szóközöket, sorbarendezéseket, behúzásokat tartalmaz. A forrásszöveg beolvasásakor a legfontosabb feladat, hogy a forrásszöveget egy egységes, a későbbi felhasználás céljára egyszerű, egyértelmű, szimbolikus alakra hozzuk. Az ömlesztett forrásszövegből tehát fel kell ismerni a szimbolikus egységeket. A szimbolikus egységeket *elválasztó karakterek* határolják. Elválasztó karaktereknek nevezünk bizonyos fehér karaktereket (pl. TAB, Space, Enter) és bizonyos speciális szimbólumokat vagy szimbólum párokat (pl. (,), :=, <, >, stb.). A szimbolikus egységeket *lexikális atomoknak* nevezzük, és ezek felismerése, elkülönítése a lexikális elemző feladata. A lexikális atomok általában a következő részekből állnak: a *szimbólum azonosítója*, *a szimbólum típusa*, *előfordulási helye a forrásszövegben* a későbbi beazonosítás céljából. A lexikális atomokat egy rendezett táblázatba (szimbólumtábla vagy lexikális atomtáblázat) írjuk.

A szimbólumok típusa általában a következő:

- fehér karakter
- kulcsszó
- azonosító
- művelet

- speciális szimbólum
- elválasztó
- kulcsszó
- string konstans
- egész konstans
- valós konstans
- megjegyzés

Példa: Tekintsük a következő *Pascal* programrészt:

```
begin
  x := 10;
end.
```

A programrész beolvasása után a következő lexikális atomtáblázat generálódik:

azonosító	*típus*	*sor*	*karakter*
begin	<kulcsszó>	1	1
x	<azonosító>	2	3
:=	<művelet>	2	5
10	<szám>	2	8
;	<elválasztó>	2	10
end	<kulcsszó>	3	1
.	<elválasztó>	3	4

A lexikális atomokat a lexikális elemző ismeri fel. Ha a forrásszövegbe egy felismerhetetlen lexikális atom kerül, akkor a lexikális elemző hibajelzést generál. A lexikális elemző karaktereken operál és a karakterekből szimbólumsorozatokat állít elő. Feladata megmondani, hogy az előállított szimbólum eleme-e a nyelvnek vagy sem. Például ha a # jelt nem tartalmazza a nyelv ábécéje, karakterkészlete, és ez előfordul egy program forráskódjában, a lexikális elemző hibajelzést ad.

A szintaktikai elemző dönti el, hogy a lexikális elemző által előállított szimbólumsorozat megfelel-e a nyelv leírásának. Ellenkező esetben szintaktikai hibajelzést generál.

A hagyományos szintaktikai elemzők meg kell, hogy határozzák a programhoz tartozó szintaxisfát, ismerve a szintaxisfa gyökérelemét és a leveleit, elő kell, hogy állítsák a szintaxisfa többi elemét és éleit, vagyis meg kell, hogy határozzák a program egy levezetését. Ha ez sikerült, akkor azt mondjuk, hogy a program eleme a nyelvnek, vagyis szintaktikusan helyes.

A szintaxisfa belső részeinek a felépítésére több módszer létezik. Az egyik az, mikor a grammatika kezdőszimbólumból kiindulva építjük fel a szintaxisfát. Ezt felülről-lefelé történő elemzésnek nevezzük. Hasonlóan, ha a szintaxisfa felépítése a levelektől halad a gyökér felé, akkor ezt alulról-felfelé elemzésnek nevezzük.

A szintaktikus elemzések elmélete van a legtökéletesebben kidolgozva, a formális nyelvek elméletének köszönhetően. Számos módszerrel lehet szintaktikai elemzést végezni (teljes visszalépéses elemzés, korlátozott visszalépéses elemzés, LL(k), LR(k) elemzések, rekurzív leszállásos elemzés stb.).

A szintaktikai elemző tehát szimbólumsorozatokon operál és azt mondja meg, hogy a szimbólumok milyen sorrendje eredményez helyes programot. Például nem mindegy, hogy a lexikálisan helyes *if then else* szimbólumokat milyen sorrendben írjuk. Sem a *then else if*, sem az *else if, then* sorrend nem helyes.

A programozási nyelv szemantikája határozza meg a szimbólumsorozat értelmét. A szemantikai elemző a forrásszöveg értelmét, az adattípusok, műveletek kompatibilitását ellenőrzi. A szemantikai elemző dönti el, hogy a szintaktikailag helyes program a

fordítás szempontjából is valóban helyesnek tekinthető-e. Ellenkező esetben szemantikai hibajelzést generál.

A szemantika olyan megkötéseket tartalmaz, mint például a típusazonosság egy értékadó utasítás két oldalán, a szimbólumok érvényességi tartományának a megadása, vagy az, hogy egy eljárás definíciójában és hívásában a formális és aktuális paraméterek darabszáma és típusa meg kell, hogy egyezzen.

Ezeket a tulajdonságokat *statikus szemantikának* hívjuk. A hagyományos értelemben vett szemantikát a statikus szemantikától való megkülönböztetésre *dinamikus* vagy *run-time* szemantikának szokták nevezni. Dinamikus például az *a* := 1 értékadó utasítás szemantikája, az, hogy az *a*-nak megfelelő állapotkomponens értéke 1-re változik.

Pascalban a konstruktorok például értelemszerűen nem lehetnek virtuálisak, így egy *constructor Init; virtual;* programrész szemantikai hibajelzést eredményez.

Kódgenerálás

A fordítóprogramok a szintézis első lépésében a már elemzett, helyes forrásprogramból futtatható bináris kódot generálnak. A tárgykód gép és operációs rendszer függő, lényege az, hogy a programozási nyelv minden egyes utasítására, vezérlési struktúrájára külön meg kell adni, milyen kódot generáljon a fordítóprogram, és azt, hogy közben milyen előre megírt eljárásokat, függvényeket kell meghívjon (pl. kifejezés kiértékelő). Kódgenerálással és az elemzési algoritmusok kifejtésével itt nem foglalkozunk, ez a téma teljesen más hatáskörbe tartozik. Kódoptimalizálással is abból a meggondolásból foglalkozunk részletesebben, hogy a megismert optimalizálási módszereket már programírásnál fel tudjuk használni, így már a forráskód optimális, szebb, áttekinthetőbb lesz.

Kód optimalizálás

A fordítóprogram a generált tárgykódot optimalizálja. Az optimalizálás történhet tárhely vagy időbonyolultság szerint. Ennek érdekében a kódoptimalizálás a tárgykódban lévő azonos utasítás sorozatok felfedezésére és eljárásokban való elhelyezésükre, ciklusok független részeinek összevonására, egymást kiegészítő utasítások redukálására, optimális regiszterhasználatra szorítkozik. Lényege, hogy az optimalizált tárgykód gyorsabban fusson le, vagy kevesebb helyet foglaljon. Célja, hogy a tárgykód minősége javuljon. Az optimalizálással szemben támasztott legfontosabb követelmény a megbízhatóság, vagyis az optimalizált tárgykód ugyanazt kell mindig eredményezze, mint az eredeti tárgykód. Az optimalizálás nem jelenti az optimális program meghatározását. Ha például egy programág sohasem fut le, azt optimálisan jobb lenne teljesen kitörölni, de ilyen döntéseket a kódoptimalizáló nem hozhat.

Az optimalizálás lehet *gépfüggő* vagy *gépfüggetlen*. A gépfüggő optimalizálás a speciális regiszterhasználatot, az adott architektúrára jellemző alapműveletetek minél jobban történő kihasználását jelenti.

Az optimalizálás lehet globális vagy lokális optimalizálás.

A globális optimalizálás program-transzformációkkal jár: utasítások kiemelése ciklusból, konstansösszevonás, utasítások kiemelése elágazásból, elágazások összevonása, ciklusok összevonása, azonos részkifejezések egyszeri kiszámítása, kifejezések algebrai egyszerűsítése stb.

A lokális optimalizálás lokális gyorsításokkal jár: felesleges utasítások kihagyása, konstanskifejezések kiértékelése, eljárás behelyettesítése, ciklus kifejtése, késleltetett tárolás, hatékony nyelvi elemek használata, felesleges műveletek elhagyása stb.

Logikai értékadásra optimálisabb a következő alakot használni:

```
egyenlo := (a = b);
```

soha ne írjuk azt, hogy:

```
if (a = b) then egyenlo := true
           else egyenlo := false;
```

vagy azt, hogy:

```
if (a = b) = true    then egyenlo := true
                     else egyenlo := false;
```

A kódoptimalizáló összevonja a konstansokat, így az *a* := *b* + 1 + *c* + 3 + 4; utasításból *a* := *b* + *c* + 8; lesz.

A fordítóprogram, kódoptimalizálás során a konstansokat továbbterjeszti, így az:

```
a := 8;
b := a / 2;
c := b + 3;
```

program részletből a:

```
a := 8;
b := 4;
c := 7;
```

programrészlet lesz.

A fordítóprogram optimalizálja a ciklusokat. Az optimalizálás lényege, hogy egy ciklusmagban levő műveleteket gyorsabban végrehajtható művelettel vagy műveletekkel helyettesítjük. Például a szorzást összeadásokkal és eltolásokkal helyettesítjük. Vegyük például a következő Pascal *while* ciklust:

```
while i < n do
   begin
      i := i + d;
      a := i * j;
   end;
```

Az első átalakítás után temporális változók bevezetésével a következő formára hozható:

```
t1 := i * j;
t2 := d * j;
while i < n do
   begin
      i := i + d;
      t1 := t1 + t2;
      a := t1;
   end;
```

Látható, hogy a t_1 veszi át az *i* szerepét. Ha a ciklusváltozónak t_1-t választjuk, akkor végfeltételének értéke *n* * *j* lesz:

```
t1 := i * j;
t2 := d * j;
t3 := n * j;
while t1 < t3 do
   begin
      i := i + d;
      t1 := t1 + t2;
      a := t1;
   end;
```

A ciklusmag első utasítására már nincs szükség. Ha az eredeti ciklusmag legalább egyszer végrehajtódik, akkor a t_1 helyett az *a* is használható, ezért a ciklusmagból a második utasítás, és ezennel a *begin* és *end*; is törölhető:

```
a := i * j;
t2 := d * j;
t3 := n * j;
while a < t3 do
   a := a + t2;
```

Cikluskifejtéssel is néha javíthatunk a kód minőségén. Például a következő ciklus helyett:

```
for i := 1 to 10 do
if odd(i) then a[i] := 0;
```

egyszerűen

```
a[1] := 0;
a[3] := 0;
a[5] := 0;
a[7] := 0;
a[9] := 0;
```

írható. Ez az öt értékadás sokkal gyorsabban végrehajtódik, mint a ciklus és az elágazás, de nem minden esetben – meggondolandó, hogy mikor éri meg jobban használni.

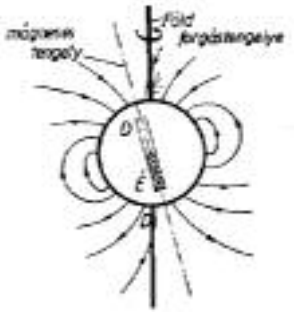
A hatékony nyelvi elemek kihasználása azt jelenti, hogy például *Pascalban i* := *i* + 5; helyett az *inc*(*i,* 5); eljárást használjuk, vagy halmazműveletek esetében az *include* és *exclude* eljárásokat használjuk. Ha *for* ciklussal keresünk egy értéket például egy tömbben, akkor ha megkaptuk, *break*-kel befejezhetjük a ciklust, adatstruktúrák lenullázását a *FillChar* eljárással végezzük stb.

Kovács Lehel

A Föld mágneses térerősségének mérése

Már az ókorban ismeretes volt, hogy a Földnek egy meghatározott mágneses tere van, amelyet a földi tájékozódásban jól fel lehetett használni. Ez a tény főleg a nagy távolságú szárazföldi vagy tengeri utazásoknál jelentett biztos tájékozódást.

Valószínűleg elsőként a kínaiak, Európában pedig a görögök ismerték fel, hogy a Földnek mágneses tere van. Ez azt jelenti, hogy a közel gömb alakú Föld mágneses szempontból egy óriás mágneses dipólusként fogható fel, melynek mágneses momentuma 10^9 Wbm. Ezt úgy is tekinthetjük mint egy mágnesrudat (lásd 1. ábrát), melynek egy jól meghatározott északi és déli pólushelye van. A Föld mágneses pólushelyei jól meghatározhatók akárcsak a földrajzi pólushelyek. A földrajzi észak-dél irányt a Föld forgástengelye jelenti.



1. ábra

Ez nem esik egybe a mágneses észak-dél iránnyal. A két irány által bezárt δ szöget a mágneses elhajlás vagy mágneses deklináció szögének nevezik.

A Föld mágneses terének eredetét napjainkig sem sikerült részleteiben tisztázni, valószínűleg több hatás együttesének tulajdonítható. Ennek megfelelően több hipotézis is igyekszik a földmágnesség okára magyarázatot adni. Az egyik ilyen hipotézis a telurikus áramok elmélete, amely feltételezi, hogy a földkéregben és a magmában különböző eredetű (galvanikus, termoelektromos) áramok mágneses tere hozza létre, melyhez hozzáadódik a Föld légkörében folyó elektromos áramok mágneses tere. Egy régebbi elmélet szerint a földkéregben lévő ferromágneses anyagok egy része mágnesezett állapotban van és ezek eredményezik a földmágnességet. Ezen elmélet szerint a kéregnek ezt a mágnesezett állapotát részben a Föld belsejében folyó áramok, részben külső kozmikus hatások hozták létre.