18 - Fordítóprogramok

Fordítóprogramok felépítése, az egyes komponensek feladata. A lexikális elemző működése, implementációja. Szintaktikus elemző algoritmusok csoportosítása, összehasonlítása; létrehozásuk és működésük vázlatos ismertetése. Az ATG-k szerepe és alapfogalmai. Kódgenerálás assemblyben alapvető imperatív vezérlési szerkezetekhez.

1. Fordítóprogramok felépítése

A programok fordítása **analízis**ből és **szintézis**ből áll.

1.1. Analízis

Az analízis átalakítja a **forrásszöveg**et a **belső reprezentáció**ra, és közben a szükséges ellenőrzéseket is elvégzi, a fordítás során itt léphet fel hiba.

1.1.1. Forrás-kezelő (*source handler*)

A forrás-kezelő bemenete a forrásfájl(ok), kimenete a forrásszöveg, mint karaktersorozat. Feladata a fájlok megnyitása, olvasása, pufferelése, az operációsrendszer-specifikus feladatok elvégzése. Ha a fordítóprogram készít listafájlt, akkor azt is kezeli.

1.1.2. Lexikális elemző (*scanner*)

A lexikális elemező bemente a forrás-kezelő által előállított karaktersorozat, kimenete szimbólumok (*token*ek) sorozata vagy lexikális hibák. Feladata a lexikális egységek (szimbólumok) felismerése, mint például: azonosítók, konstansok, kulcsszavak.

1.1.3. Szintaktikus elemző (*parser*)

A szintaktikus elemző bemenete a szimbólumsorozat, kimenete a szintaktikusan elemzett program, (például szintaxisfa formájában) vagy szintaktikus hibák. Feladata a program szerkezetének felismerése, a szerkezet ellenőrzése: megfelel-e a nyelv definíciójának?

1.1.4. Szemantikus elemző (*semantic analizer*)

A szemantikus elemző bemenete a szintaktikusan elemzett program (például szintaxisfa vagy szimbólumtábla formájában), kimenete a szemantikusan elemzett program vagy szemantikus hibák. Feladata a környezetfüggő ellenőrzések elvégzése, pl.: típusellenőrzés, változók láthatóságának ellenőrzése, eljárások hívása megfelel-e a szignatúrának?

1.2. Szintézis

A szintézis a belső reprezentációra hozott programból hozza létre a **tárgyprogram**ot.

1.2.1. Kódgenerátor (*code generator*)

A kódgenerátor bemenete a szemantikusan elemzett program, kimenete a tárgykód utasításai (pl. assembly, gépi kód). Feladata a forrásprogrammal ekvivalens tárgyprogram készítése.

1.2.2. Kódoptimalizáló (*optimizer*)

A kódoptimalizáló bemenete a tárgykód, kimenete az optimalizált tárgykód. Feladata valamilyen szempontok szerint jobb kód készítése (pl. futási sebesség növelése, méret csökkentése, felesleges utasítások megszüntetése, ciklusok kifejtése). Az optimalizáció végezhető az eredeti programon (szemantikus elemzés után) vagy a tárgykódon is (a kódgenerálás után).

1.2.3. Kód-kezelő (*code handler*)

A kód-kezelő bemenete az optimalizált tárgykód, kimenete a kimeneti fájlok, amiket például a háttértárolóra ír.

1.3. Fordítás menete

A fordítóprogram egyes komponensei egymást követik, az egyik kimenete a másik bemenete. Ez nem jelenti azt, hogy például a lexikális elemzőnek be kell fejeződnie a szintaktikus elemzés előtt, a szintaktikus elemző meghívhatja a lexikálisat, amikor egy újabb szimbólumra van szüksége. Ennek eredményeképp egy utasításnak akár a tárgykódja is elkészülhet, amikor a következő utasítás még be sincs olvasva.
A kódoptimalizálásnál viszont jellemzően újra át kell olvasni az egész tárgykódot (akár többször is). A **fordítás menetszáma** alatt azt értjük, ahányszor a programszöveget (vagy annak belső reprezentációit) végigolvassa a fordító a teljes fordítási folyamat során.

2. Lexikális elemző

A lexikális elemző megadható a **Chmosky 3. nyelvosztály**beli nyelvtanokkal, ami igen egyszerűvé teszi a megoldást. A Chmosky 3. nyelvosztállyal ekvivalens a **reguláris kifejezés**ek nyelvosztálya és a **véges determinisztikus automaták** nyelvosztálya. Ezt felhasználva a lexikális elemzőt magas szinten reguláris szabályokkal szokás megadni, például *Flex* szintaxissal. Ez már automatikusan átalakítható véges determinisztikus automatává, ami két féle módon is implementálható:

- **egymásba ágyazott `if` vagy `case` utasításokkal egy cikluson belül**: elágazunk a pillanatnyi állapot szerint, ezen belül pedig a következő karakter szerint. Az egyes ágakban beállítjuk a következő állapotot, és kezdjük előről.
- **táblázattal**: a táblázat soraihoz az állapotok, oszlopaihoz a karakterek vannak rendelve, celláiban a következő állapot sorszáma van. A következő állapot a pillanatnyi állapot sorában és az olvasott karakter oszlopában található.

A lexikális elemző **mohó algoritmus** alapján működik, a leghosszabb olyan karaktersorozatot ismeri fel szimbólumként, amelyre illeszkedik valamelyik reguláris szabály. Ha több szabály is illeszkedik rá, akkor azok sorrendje a meghatározó, a korábban megadott szabály a meghatározó.

A lexikális elemzőnek a felismert szimbólum fajtáján kívül egyéb információkat is továbbítania kell, például az azonosítók nevét vagy a konstansok értékét, mert ezekre a **szemantikus érték**ekre szükség lesz a szemantikus elemzésnél és a kódgenerálásnál.

Lexikális hiba akkor lép fel, ha a lexikális elemző egy karaktersorozatnak nem tud egy szimbólumot sem megfeleltetni. Ennek oka lehet például illegális karakter, elgépelt kulcsszó, kihagyott szimbólum, vagy hibás számformátum.

Nem célszerű hiba esetén az elemzést megszakítani, így valamilyen **hibaelfedő algoritmus**ra van szükség, amely lehetőséget biztosít arra, hogy az elemzés a lehető legkevesebb karakter kihagyásával folytatódjon. Illegális karakter esetén például a következő lehetőségeink vannak:

- a karakter eldobása
- a karakter helyettesítése szóközzel
- az éppen épített teljes szimbólum eldobása

Bővebben >>

3. Szintaktikus elemző

A szintaktikus elemző megadható a **Chmosky 2. nyelvosztály**beli, azaz környezetfüggetlen nyelvtanokkal. Megköveteljük ezen túl a nyelvtan **ciklusmentes**ségét (nincsen $A \Rightarrow^+ A$ levezetés), **redukált**ságát (nem tartalmaz „felesleges” nemterminálisokat) és **egyértelmű**ségét (minden mondathoz pontosan egy szintaxisfa tartozik).

A szintaktikus elemző algoritmusokat több szempont szerint is csoportosíthatjuk. **Legbal** levezetésnek azt nevezzük, ha mindig a bal szélső nemterminálist helyettesítjük egy szabály jobb oldalával. Hasonlóan definiálható a **legjobb** levezetés.

Az **elemzési irány**okat tekintve **felülről lefelé** és **alulról felfelé** elemzéseket különböztetünk meg. Előbbinél a startszimbólumból indulva, felülről lefelé építjük a szintaxisfát. A mondatforma baloldalán megjelenő terminálisokat illesztjük az elemzendő szimbólumsorozatra. Utóbbinál az elemzendő szimbólumsorozat összetartozó részeit helyettesítjük nemterminális szimbólumokkal (redukció) és így alulról, a startszimbólum felé építjük a fát.
A felülről lefelé elemzés egy legbal levezetés, az alulról felfelé elemzés pedig egy legjobb levezetés inverzét adja ki.

3.1. Felülről lefelé elemzések

Az egyik lehetséges stratégia a **visszalépéses keresés** (*backtrack*), azaz ha nem illeszkednek a szimbólumsorozatra a mondatforma baloldalán megjelenő terminálisok, lépjünk vissza, és válasszunk másik szabályt. Hátránya, hogy lassú, és ha hibás a szöveg, az csak túl későn derül ki.

A másik lehetőség az **előreolvasás**os stratégia, ez esetben olvassunk előre a szövegben valahány szimbólumot, és az alapján döntsünk az alkalmazandó szabályról. Az ilyen elemzéseket ***LL* elemzés**eknek (***L****eft to right, using a* ***L****eftmost derivation* – balról jobbra, legbal levezetéssel) nevezzük. Hátránya, hogy csak szűk nyelvosztályra alkalmazható.

Egy nyelvtant $\boldsymbol{LL(k)}$ grammatikának nevezünk, ha a levezetés tetszőleges pontján a szimbólumsorozat következő k terminálisa meghatározza az alkalmazandó levezetési szabályt. A gyakorlatban könnyen megvalósítható LL grammatikák:

- Egyszerű $LL(1)$: a szabályok jobboldala terminális szimbólummal kezdődik.
- ϵ-mentes $LL(1)$: ϵ-mentes $LL(1)$ nyelvtan.
- Általános $LL(1)$.

Az LL elemzés implementálása az **elemző táblázat** felírásával történik. Ezekhez az ϵ-mentes $LL(1)$ nyelvtanok esetében szükséges a $FIRST_1$, az általános $LL(1)$ nyelvtanok esetében a $FIRST_1$ és a $FOLLOW_1$ halmazok felírása.

A $\boldsymbol{FIRST_k(\alpha)}$ halmazt az α mondatformából levezethető terminális sorozatok k hosszúságú kezdőszeletei alkotják. Ha a sorozat hossza kisebb, mint k, akkor az egész sorozat eleme $FIRST_k(\alpha)$-nak, akár $\epsilon \in FIRST_k(\alpha)$ is előfordulhat.

A $\boldsymbol{FOLLOW_k(\alpha)}$ halmazt a levezetésekben az α mondatforma után előforduló k hosszúságú terminális sorozatok alkotják. Ha a sorozat hossza kisebb, mint k, akkor az egész sorozat eleme $FOLLOW_k(\alpha)$-nak, ha α után vége a szövegnek, akkor # $\in FIRST_k(\alpha)$.

Az $LL(k)$ elemzés vázlatos működése:

- ha a verem tetején terminális szimbólum van, és az egyezik a szimbólumsorozat következő karakterével, akkor *pop* és lépés a szimbólumsorozatban. Máskülönben szintaktikai hiba.
- ha a verem tetején nemterminális szimbólum (A) van, és van olyan $A \longrightarrow \alpha$ szabály, amelyre a szimbólumsorozat előreolvasott k eleme megfelel, akkor a veremben A helyére α kerül és a szabály alkalmazása bejegyzésre kerül a szintaxisfába. Máskülönben szintaktikai hiba.
- ha a verem üres és a szöveg végére értünk, akkor *OK*. Máskülönben szintaktikai hiba.

Bővebben >>

3.2. Alulról felfelé elemzések

Az alulról felfelé elemzéseknél is az egyik lehetséges stratégia a **visszalépéses keresés** (*backtrack*), azaz ha nem sikerül eljutni a startszimbólumig, lépjünk vissza, és válasszunk másik redukciót. Hátrányai ugyanazok, mint a felülről lefelé elemzésnél, hogy lassú, és ha hibás a szöveg, az csak túl későn derül ki.

Másik opció a **precedencia elemzés**ek használata, ekkor az egyes szimbólumok között precedenciarelációkat adunk meg, és ezek segítségével határozzuk meg a megfelelő redukciót. Ez az elemzési módszer ma már kevéssé alkalmazott, de az operátorokkal felépített kifejezések esetén természetes a használata.

A harmadik lehetőség az **előreolvasás**os stratégia, ez esetben olvassunk előre a szövegben valahány szimbólumot, és az alapján döntünk a redukcióról. Az ilyen elemzések ***LR* elemzés**nek (***L****eft to right, using a* ***R****ightmost derivation* – balról jobbra, legjobb levezetéssel) nevezzük. Előnye, hogy minden programozási nyelvhez lehet LR elemzőt készíteni, és majdnem mindegyikhez lehet gyorsabb, hatékonyabb LALR elemzőt konstruálni.

Egy nyelvtant $\boldsymbol{LR(k)}$ grammatikának nevezünk, ha a levezetés tetszőleges pontján k szimbólum előreolvasásával eldönthető, hogy mi legyen az elemzés következő lépése. Ez lehet **léptetés** (a szimbólumsorozat következő elemét a verem tetejére helyezzük és tovább lépünk) vagy **redukálás** (a verem tetején található szabály jobb oldalt helyettesítjük a bal oldalán álló nemterminálissal). Redukciókor mindig a nyelet – a mondatforma legbaloldalibb egyszerű részmondatát – redukáljuk.

Az LR elemező úgy implementálható, hogy egy véges determinisztikus automatát készítünk, amelynek az átmeneteit a verembe kerülő szimbólumok határozzák meg: léptetéskor terminális, redukáláskor nemterminális. Amikor az automata elfogadó állapotba jut, akkor kell redukálni.
A véges determinisztikus automatát úgy tudjuk megvalósítani, hogy az állapotokat kanonikus **halmaz**okkal reprezetáljuk.

A gyakorlatban könnyen megvalósítható LR grammatikák:

- $LR(0)$: előreolvasás nélkül tudunk dönteni az elemzés következő lépéséről. Bővebben >>
- $SLR(1)$: *Simple* (egyszerű) $LR(1)$, ilyenkor az előreolvasott szimbólum globális az egész grammatikára, a $FOLLOW_1$ halmazt használjuk. Bővebben >>
- $LR(1)$: az előreolvasási szimbólumot a kanonikus halmazok elemeihez külön-külön rendeljük hozzá, így nem léphet fel konfliktus olyankor, amikor a $FOLLOW_1$ halmaznak nem is minden eleme követheti a szabályt. Bővebben >>
- $LALR(1)$: ***L**ook**A**head* (előreolvasó) $LR(1)$, amely az $LR(1)$ elemzés hatékonyságát optimalizálja. Az $LR(1)$ grammatikák állapotainak száma jelentősen megnő az $LR(0)$ és az $SLR(1)$ grammatikák állapotainak számához képest az előreolvasási szimbólum miatt. A $LALR(1)$ nyelvtan az $LR(1)$ grammatika azon kanonikus halmazainak összevonásából készül, ahol az elemek csak az előreolvasási szimbólumban térnek el. Mivel megjelennek benne az előreolvasási szimbólumok, több nyelvtan lesz elemezhető vele, mint az $SLR(1)$ módszerrel, de nem minden $LR(1)$ grammatika esetén használható, mert redukálás/redukálás konfliktus léphet fel. A $LALR(1)$ elemző nem csak a kész $LR(1)$ elemzőből, hanem a nélkül is, hatékonyabban is generálható. Bővebben >>

4. Szemantikus elemző

Fordítási grammatikának azt nevezzük, ha a nyelvtan szabályait kiegészítjük a szemantikus elemzés tevékenységeivel. Ezeket az **akciószimbólum**ok segítségével tesszük meg, a nyelvtanban @ jellel kezdődnek. Az akciószimbólumok által jelölt szemantikus tevékenységeket szemantikus rutinoknak nevezzük.

Attribútum fordítási grammatikáról (*ATG*) akkor van szó, ha a nyelvtan szimbólumaihoz szemantikus típusokat (attribútumokat) rendelünk, ezekben tárolják a szemantikus rutinok a szükséges információkat. Az attribútum fajtája szerint három féle lehet:

- **Szintetizált attribútum**: a helyettesítési szabály bal oldalán áll abban a szabályban, amelyikhez az őt kiszámoló szemantikus rutin tartozik.
  Az információt a szintaxisfában alulról felfelé közvetíti.
- **Örökölt attribútum**: a helyettesítési szabály jobb oldalán áll abban a szabályban, amelyikhez az őt kiszámoló szemantikus rutin tartozik.
  Az információt a szintaxisfában felülről lefelé közvetíti.
- **Kitüntetett szintetizált attribútum**: olyan attribútum, amelyek terminális szimbólumhoz tartozik, és kiszámításához nem használunk fel más attribútumokat.
  Az információt általában a lexikális elemző szolgáltatja.

Az attribútum fordítási grammatikára a következő megkötések vonatkoznak:

- Egy adott szabályhoz tartozó feltételek csak a szabályban előforduló attribútumoktól függhetnek. (Ha egy feltétel nem teljesül, akkor **szemantikus hibát** kell jelezni!)
- A szemantikus rutinok csak annak a szabálynak az attribútumait használhatják és számíthatják ki, amelyikhez az őket reprezentáló akciószimbólum tartozik.
- Minden szintaxisfában minden attribútum értékét pontosan egy szemantikus rutin határozhatja meg.

Jól definiált attribútum fordítási grammatika olyan attribútum fordítási grammatika, amelyre igaz, hogy a grammatika által definiált nyelv mondataihoz tartozó minden szintaxisfában minden attribútum értéke egyértelműen kiszámítható.

Ha az $Y.b$ attribútumot kiszámoló szemantikus rutin használja az $X.a$ attribútumot, akkor $(X.a, Y.b)$ egy **direkt attribútumfüggőség**.
Ezek a függőségek a **függőségi gráf**ban ábrázolhatók. Jól definiált attribútum fordítási grammatikákhoz tartozó szintaxisfák függőségi gráfjaiban biztosan nincsenek körök!

Egy lehetséges **attribútumkiértékelő algoritmus** a **nemdeterminisztikus algoritmus**: a szintaxisfa minden attribútumához egy folyamatot rendelünk, a folyamat pedig figyeli, hogy az adott attribútum kiértékeléséhez szükséges összes attribútum értékét meghatározta-e már a többi folyamat. Ha igen, akkor a folyamat kiszámítja az adott attribútum értékét. Ha az attribútum fordítási grammatika jól definiált, akkor ez a program biztosan terminál és kiszámítja az összes attribútumértéket.
Azonban ennél hatékonyabb algoritmust szeretnénk.

Particionált, illetve **rendezett attribútum fordítási grammatikák** esetében megszorításokat teszünk a grammatikára, hogy a kiértékelés egyszerűbb legyen.

A kiértékelés sorrendjét általában nem lehet az attribútum-fordítási grammatikából meghatározni, mert függhet a konkrét szintaxisfától. A particionált *ATG* ezen úgy segít, hogy minden X szimbólum attribútumai szétoszthatók az $\mathcal{A}_1^X, \mathcal{A}_2^X, \dots \mathcal{A}_{m_X}^X$ diszjunkt halmazokba (*partíció*kba) úgy, hogy:

- az $\mathcal{A}_{m_X}^X, \mathcal{A}_{m_X-2}^X, \dots$ halmazokban csak szintetizált attribútumok,
- az $\mathcal{A}_{m_X-1}^X, \mathcal{A}_{m_X-3}^X, \dots$ halmazokban csak örökölt attribútumok vannak, és
- minden szintaxisfában az X attribútumai a halmazok növekvő sorrendjében meghatározhatók.

Ha ismerjük a partíciókat, akkor csak az attribútum fordítási grammatika szabályai alapján tudunk olyan programot írni, ami a megfelelő sorrendben kiszámítja a szintaxisfában lévő összes attribútumot.

A partíciókat általában nem könnyű meghatározni. A rendezett *ATG*-k esetén egy egyszerű algoritmussal meghatározhatók a partíciók. A halmazokat fordított sorrendben határozzuk meg olyan módon, hogy felváltva szintetizált és örökölt attribútumokat teszünk a halmazokba (utolsóba szintetizáltat, utolsó előttibe örököltet, stb.). Mindegyik halmazba azokat az attribútumokat tesszük, amikre csak olyan attribútumok kiszámításához van szükség, amelyeket már beosztottunk valamelyik halmazba. Ha ezzel az algoritmussal megfelelő partíciókat kapunk, akkor hívjuk a **nyelvtant rendezett attribútum fordítási grammatikának**.

Az ***S*-attribútum fordítási grammatika** olyan *ATG*, amelyben kizárólag szintetizált attribútumok vannak. A szemantikus információ a szintaxisfában a levelektől a gyökér felé terjed.
Jól illeszthető az alulról felfelé elemzésekhez.

Az ***L*-attribútum fordítási grammatika** olyan *ATG*, amelyben minden $A \longrightarrow X_1 X_2 \dots X_n$ szabályban az attribútumértékek az alábbi sorrendben meghatározhatóak:

- A örökölt attribútumai
- X_1 örökölt attribútumai
- X_1 szintetizált attribútumai
- X_2 örökölt attribútumai
- X_2 szintetizált attribútumai
- ….
- X_n örökölt attribútumai
- X_n szintetizált attribútumai
- A szintetizált attribútumai

Jól illeszkedik a felülről lefelé elemzésekhez.

Bővebben >>

5. Kódgenerátor

5.1. Értékadások fordítása

Az értékadás alakja:

$$assignment \longrightarrow variable\ operator\ expression$$

A generálandó kód:

```
; kifejezést az eax regiszterbe kiértékelő kód
mov [Változó], eax
```

A kifejezések kiértékelése nem tárgya a tételnek. A `Változó` az értékadás bal oldalán szereplő változó címkéje. Bonyolultabb adatszerkezetek lemásolását külön eljárás végzi (másoló konstruktor).

5.2. Egy ágú elágazás fordítása

Az egy ágú elágazás alakja:

$$statement \longrightarrow if\ condition\ then\ program\ end$$

A generálandó kód:

```
; a feltételt az al regiszterbe kiértékelő kód
cmp al, 1
jne near Vége
; a then-ág programjának kódja
Vége:
```

A feltételek kiértékelése nem tárgya a tételnek. Mivel a programban több elágazás is lehet, minden esetben egyedi címkéket kell generálni. Ezt könnyen megvalósíthatjuk például egy globális számláló használatával.

5.3. Több ágú elágazás fordítása

A több ágú elágazás alakja:

$statement \longrightarrow$
$if\ condition_1\ then\ program_1$
$elseif\ condition_2\ then\ program_2$
…
$elseif\ condition_n\ then\ program_n$
$else\ program_{n+1}\ end$

A generálandó kód:

```
; az 1. feltétel kiértékelése az al regiszterbe
cmp al, 1
jne near Feltétel_2
; az 1. ág programjának kódja
jmp Vége
; …
Feltétel_n:
; az n-edik feltétel kiértékelése az al regiszterbe
cmp al, 1
jne near Else
; az n-edik ág programjának kódja
jmp Vége
Else:
; az else ág programjának kódja
Vége:
```

5.4. *switch-case* utasítás fordítása

A *switch-case* utasítás alakja:

$statement \longrightarrow switch\ variable$
$case\ value_1 : program_1$
$case\ value_2 : program_2$
…
$case\ value_n : program_n$

A generálandó kód:

A generálandó kód hasonló egy több ágú elágazáshoz, ezért nem ismételjük meg. Azonban itt a feltételekre megszorítások vannak (csak `variable == value` alakúak lehetnek a feltételek, ahol a `value` konstans érték), ezért lehet hatékonyabb a kiértékelés.

Ügyelni kell arra, hogy a *switch-case* másként működik az egyes nyelvekben:

- **Ada stílus**: csak egy ág hajtódik végre.
- **C stílus**: az első teljesülő ágtól kezdve az összes végrehajtódik (hacsak nem használunk `break` utasítást az ágak végén).

5.5. Elöl tesztelő ciklus fordítása

Az elöl tesztelő ciklus alakja:

$$statement \longrightarrow while\ condition\ program\ end$$

A generálandó kód:

```
Eleje:
; a ciklusfeltétel kiértékelése az al regiszterbe
cmp al, 1
jne near Vége
; a ciklusmag programjának kódja
jmp Eleje
Vége:
```

5.6. Hátul tesztelő ciklus fordítása

A hátul tesztelő ciklus alakja:

$$statement \longrightarrow loop\ program\ while\ condition$$

A generálandó kód:

```
Eleje:
; a ciklusmag programjának kódja
; a ciklusfeltétel kiértékelése az al regiszterbe
cmp al,1
je near Eleje
```

5.7. Számláló ciklus fordítása

A számláló ciklus alakja:

$$statement \longrightarrow for\ variable\ from\ value_1\ to\ value_2\ program\ end$$

A generálandó kód:

```
; a „from” érték kiszámítása a [Változó] memóriahelyre
Eleje: ; a „to” érték kiszámítása az eax regiszterbe
cmp [Változó], eax
ja near Vége
; a ciklusmag kódja
inc [Változó]
jmp Eleje
Vége:
```

A generált kód hasonlít az elöl tesztelő cikluséhoz, ami nem véltetlen, hiszen minden számlálós ciklus átalakítható elöl tesztelő ciklussá.

Hatékonyabb megoldás lehet, ha a ciklusváltozót regiszterben tároljuk. Erre van processzor szintű támogatás is, a **`loop`** utasítás. A `loop Címke` kód csökkenti az `ecx` regiszter értékét eggyel, és ha még pozitív, akkor a címkére ugrik, egyébként pedig továbblép. A verem használatával kivédhetjük azt is, hogy a ciklusmag elállítsa az `ecx` regiszter értékét.

Bővebben a vezérlési szerkezetek és a kifejezések kódgenerálásáról >>

Leírás az alprogramokkal és a memóriakezeléssel kapcsolatos kódgeneráláshoz >>