

# Fordítóprogramok beugrókérdések

---

## 1) Mirol mire fordít a fordítóprogram?

Általában magas szintű programozási nyelvről gépi kódra.

## 2) Mirol mire fordít az assembler?

Assembly nyelvről gépi kódra.

## 3) Mi a különbség a fordítóprogram és az interpreter között?

Fordítóprogram: A forráskódot a fordítóprogram tárgyprogrammá alakítja. A tárgyprogramokból a szerkesztés során futtatható állomány jön létre.

Interpreter: A forráskódot az interpreter értelmezi és azonnal végrehajtja.

## 4) Mi a virtuális gép?

Olyan gép szoftveres megvalósítása, amelynek a bájtkód a gépi kódja.

## 5) Mi a különbség a fordítási és a futási idő között?

Fordítási idő: amikor a fordító dolgozik

Futási idő: amikor a program fut

## 6) Mi a feladata az analízisnek (a fordítási folyamat első fele) és milyen részfeladatokra bontható?

Az analízis elsődleges feladata, hogy meghatározza az egyes szimbolikus egységeket, konstansokat, változókat, kulcsszavakat, operátorokat ezt a feladatot a lexikális elemző végzi, ami egy karaktersorozatból szimbólum sorozatot generál.

Ez bekerül a szintaktikus elemzőbe, aminek a feladata a program struktúrájának felismerése, ellenőrzése: szimbólumok megfelelő sorrendben vannak-e? értelmes-e? a nyelvnek megfelel-e? stb. ezt megkapja a szemantikus elemző, mely eldönti, a kód a szükséges szemantikus tulajosságokkal rendelkezik-e?

## 7) Mi a feladata a szintézisnek (a fordítási folyamat második fele) és milyen részfeladatokra bontható?

Szintézis feladatainak részletezése:

az első lépés a kódgenerálás, melyet a kódgenerátor végez ezt követi a kódoptimalizálás.

## 8) A fordítóprogram mely részei adhatnak hibajelzést?

A lexikális, szintaktikus, szemantikus elemzők adhatnak hibát.

## 9) Mi a lexikális elemző feladata, bemenete, kimenete?

Feladat: lexikális egységek (szimbólumok) felismerése – azonosító, konstans, kulcsszó,...

input: karakterszorozat

output: szimbólumsorozat, lexikális hibák

## 10) Mi a szintaktikus elemző feladata, bemenete, kimenete?

Feladat: program szerkezetének felismerése, a szerkezet ellenőrzése: megfelel-e a nyelv definíciójának?

input: szimbólumsorozat

output: szintaxisfa, szintaktikus hibák

## 11) Mi a szemantikus elemző feladata, bemenete, kimenete?

Feladat: típusellenőrzés, változók láthatóságának ellenőrzése, eljárások hívása megfelel-e a szignatúrának?, stb.

input: szintaktikusan elemzett program

output: szemantikusan elemzett program, szemantikus hibák.

**12) Mi a kódgenerátor feladata, bemenete, kimenete?**

Feladat: forrásprogrammal ekvivalens tárgyprogram készítése  
input: szemantikusan elemzett program  
output: tárgykód utasításai (pl assembly, gépi kód)

**13) Mi a kódoptimalizáló feladata, bemenete, kimenete?**

Feladat: valamilyen szempontok szerint jobb kód készítése:  
- futási sebesség növelése, méret csökkentése,  
- felesleges utasítások megszüntetése, ciklusok kifejtése....  
input: tárgykód  
output: optimalizált tárgykód

**14) Mi a fordítás menetszáma?**

A fordítás annyi menetes, ahányszor a programszöveget (vagy annak belső reprezentációit) végigolvassa a fordító a teljes fordítási folyamat során.

**15) Milyen nyelvtanokkal dolgozik a lexikális elemző?**

reguláris, 3-as nyelvtanokkal

**16) Hogy épülnek fel a reguláris kifejezések?**

Elemek: üres halmaz, üres szöveget tartalmazó halmaz, egy karaktert tartalmazó halmaz  
Konstrukciós műveletek a fenti halmazokkal: konkatenáció, unió, lezárás  
További „kényelmi” műveletek: +, ?

**17) Hogy épülnek fel a véges determinisztikus automaták?**

Elemei: ábécé, állapotok halmazai, átmenetfüggvény, kezdőállapot, végállapotok halmaza

**18) Milyen elv szerint ismeri fel a lexikális elemző a lexikális elemeket?**

A lexikális elemző mindenkor leghosszabb karakterszorozatot ismeri fel.

**19) Mi a szerepe a lexikális elemek sorrendjének?**

A lexikális elemző megadásakor sorbarendezhetjük a szimbólumok definícióit.  
Ha egyszerre több szimbólum is felismerhető, a sorrendben korábbi lesz az eredmény.

**20) Mi a különbség a kulcsszavak és a standard szavak között?**

A standard szó felüldefiniálható, de a kulcsszavak nem.

**21) Mi az elofeldolgozó fázis feladata?**

- feljegyzi a makródefiníciókat,
- elvégzi a makróhelyettesítéseket,
- meghívja a lexikális elemzőt a beillesztett fájlokra,
- kiértékeli a feltételeket és dönt a kódrészletek beillesztéséről vagy törléséről.

**22) Mutass példát olyan hibára, amelyet a lexikális elemző fel tud ismerni és olyanra is, amelyet nem!**

Feltud ismerni:

- hibás számformátum (pl. 1.23.45)  
valamelyiket illegális karakternek lehet tekinteni

Nem tud felismerni:

- kihagyott szimbólum (pl. 1+a helyett 1a, a+1 helyett a1)  
ezeket csak a szintaktikus elemző tudja észrevenni

### **23) Mikor ciklusmentes egy nyelvtan?**

Egy nyelvtan ciklusmentes, ha nincs  $A \Rightarrow^+ A$  levezetés

Ellenpélda:  $A \rightarrow B, B \rightarrow A$

### **24) Mikor redukált egy nyelvtan?**

„nincsenek felesleges nemterminálisok”

- minden nemterminális szimbólum előfordul valamelyik mondatformában

- mindegyikből levezethető valamelyik terminális sorozat

ellenpélda:  $A \rightarrow aA$ , ha ez az egyetlen szabály  $A$ -ra

### **25) Mikor egyértelmu egy nyelvtan?**

Minden mondathoz pontosan egy szintaxisfa tartozik.

### **26) Mi a különbség a legbal és legjobb levezetés között?**

Legbal: Mindig a legbaloldalibb nemterminálist helyettesítjük.  $S \Rightarrow AB \Rightarrow aaB \Rightarrow aab$

Legjobb: Mindig a legjobboldalibb nemterminálist helyettesítjük.  $S \Rightarrow AB \Rightarrow Ab \Rightarrow aab$

### **27) Mi a különbség a felülről lefelé és az alulról felfelé elemzés között?**

Felülről lefelé:

A startszimbólumból indulva, felülről lefelé építjük a szintaxisfát. A mondatforma baloldalán megjelenő terminálisokat illesztjük az elemzendő szövegre.

Alulról felfelé:

Az elemzendő szöveg összetartozó részeit helyettesítjük nemterminális szimbólumokkal (redukció) és így alulról, a startszimbólum felé építjük a fát.

### **28) Mi az összefüggés az elemzési irányok és a legbal, illetve legjobb levezetés között?**

Felülről lefelé elemzés = Legbal levezetés

Alulról felfelé levezetés = Legjobb levezetés inverzával.

### **29) Milyen alapvető stratégiák használatosak a felülről lefelé elemzésekben?**

- visszalépéses keresés, előreolvasás

### **30) Milyen alapvető stratégiák használatosak az alulról felfelé elemzésekben?**

- visszalépéses keresés, precedenciák használata, előreolvasás

### **31) Definiáld a FIRST $k(\underline{\alpha})$ halmazt, és röviden magyarázd meg a definíciót!**

Az  $\alpha$  mondatformából levezethető terminális sorozatok  $k$  hosszúságú kezdőszeletei.

(ha a sorozat hossza kisebb mint  $k$ , akkor az egész sorozat eleme FIRST $k(\alpha)$ -nak, akár  $\epsilon \in \text{FIRST}_k(\alpha)$  is előfordulhat)

$$\text{FIRST}_k(\alpha) = \{x \mid \alpha \Rightarrow^* x\beta \wedge |x| = k\} \cup \{x \mid \alpha \Rightarrow^* x \wedge |x| < k\}$$

**32) Definiáld az LL(k) gramatikákat és röviden magyarázd meg a definíciót!**

A levezetés tetszőleges pontján a szöveg következő k terminálisa meghatározza az alkalmazandó levezetési szabályt.

Tetszőleges

$$S \Rightarrow^* wA\beta \Rightarrow w\alpha_1\beta \Rightarrow^* wx$$

$$S \Rightarrow^* wA\beta \Rightarrow w\alpha_2\beta \Rightarrow^* wy$$

levezetéspárra FIRST<sub>k</sub>(x) = FIRST<sub>k</sub>(y) esetén  $\alpha_1 = \alpha_2$ .

**33) Definiáld a FOLLOW<sub>k</sub>(\_) halmazt és röviden magyarázd meg a definíciót!**

$$\text{FOLLOW}_k(\alpha) = \{x \mid S \Rightarrow^* \beta\alpha\gamma \wedge x \in \text{FIRST}_k(\gamma) \setminus \{\epsilon\} \cup \{\#\mid S \Rightarrow^* \beta\alpha\}$$

FOLLOW<sub>k</sub>(α): a levezetésekben az α után előforduló k hosszúságú terminális sorozatok (ha a sorozat hossza kisebb mint k, akkor az egész sorozat eleme FOLLOW<sub>k</sub>(α)-nak,

ha α után vége lehet a szövegnek, akkor # ∈ FOLLOW<sub>k</sub>(α))

**34) Definiáld az egyszerű LL(1) gramatikát!**

Olyan LL(1) gramma, amelyben a szabályok jobboldala terminális szimbólummal kezdődik (ezért ε-mentes is). (Az összes szabály A → aa alakú.)

**35) Mi az egyszerű LL(1) gramatikáknak az a tulajdonsága, amire az elemző épül?**

Egyszerű LL(1) gramma esetén az azonos nemterminálhoz tartozó szabályok jobboldalai különböző terminálissal kezdődnek.

**36) Mit csinál az egyszerű LL(1) elemző, ha a verem tetején az A nemterminális van és a bemenet következő szimbóluma az a terminális?**

Ha van A → aa szabály:

- A helyére aa és bejegyzés a szintaxisfába
- különben: hiba

**37) Definiáld az ε -mentes LL(1) gramatikát!**

Olyan LL(1) gramma, amely ε-mentes. (Nincs A → ε szabály.)

**38) Mi az ε -mentes LL(1) gramatikáknak az a tulajdonsága, amire az elemző épül?**

ε-mentes LL(1) gramma esetén az egy nemterminálhoz tartozó szabályok jobboldalainak FIRST<sub>1</sub> halmazai diszjunktak.

**39) Mit csinál az  $\mathcal{E}$  -mentes LL(1) elemző, ha a verem tetején az A nemterminális van és a bemenet következő szimbóluma az a terminális?**

Ha van  $A \rightarrow \alpha$  szabály, amelyre  $a \in \text{FIRST1}(\alpha)$ :

- A helyére  $\alpha$  és bejegyzés a szintaxisfába
- különben: hiba

**40) Definiáld az LL(1) grammatikát!**

Tetszőleges

$$S \Rightarrow^* wA\beta \Rightarrow w\alpha_1\beta \Rightarrow^* wx$$

$$S \Rightarrow^* wA\beta \Rightarrow w\alpha_2\beta \Rightarrow^* wy$$

levezetéspárra  $\text{FIRST1}(x) = \text{FIRST1}(y)$  esetén  $\alpha_1 = \alpha_2$ .

**41) Mi az LL(1) grammatikáknak az a tulajdonsága, amire az elemző épül?**

$\text{FIRST1}(\alpha\text{FOLLOW1}(A))$  jelentése:  $\alpha$ -hoz egyenként konkatenáljuk

$\text{FOLLOW1}(A)$  elemeit és így kapott halmaz minden elemére alkalmazzuk a  $\text{FIRST1}$  függvényt.

**42) Mit csinál az LL(1) elemző, ha a verem tetején az A nemterminális van és a bemenet következő szimbóluma az a terminális?**

Ha van  $A \rightarrow \alpha$  szabály, amelyre  $a \in \text{FIRST1}(\alpha\text{FOLLOW1}(A))$ :

- A helyére  $\alpha$  és bejegyzés a szintaxisfába
- különben: hiba

**43) Milyen komponensei vannak az LL(1) elemzőknek?**

//táblázat miből épül fel

**44) Hogyan épülnek fel a rekurzív leszállásos elemzőben a nemterminális szimbólumokhoz rendelt eljárások?**

Nemterminális szimbólumokhoz rendelt eljárások a nyelvtani szabálytól függően tartalmaznak más eljárásokat amik nyelvtani szabályokat reprezentálnak, vagy terminális szimbólum levezetésekét, a szabálynak megfelelő sorrendben.

Ezek elágazásokban helyezkednek el az éppen aktuálisan olvasott szimbólum szempontjából.

**45) Mit jelentenek a léptetés és redukálás műveletek?**

Redukálás: Az elemzendő szöveg összetartozó részeit helyettesítjük nemterminális szimbólumokkal (redukció) és így alulról, a kezdőszimbólum felé építjük a fát.

Léptetés: Egyik állapotból a másikba lépünk az automatával.

**46) Mi a kiegészített grammatika és miért van rá szükség?**

Az elemzés végét arról fogjuk felismerni, hogy egy redukció eredménye a kezdőszimbólum lett.

Ez csak akkor lehet, ha a kezdőszimbólum nem fordul elő a szabályok jobboldalán.

Ezt nem minden grammaтика teljesíti, de mindegyik kiegészíthető:

legyen S' az új kezdőszimbólum

legyen S' → S egy új szabály

#### 47) Mi a nyél szerepe az alulról felfelé elemzésekben?

Nyél: a mondatformában a legbaloldalibb egyszerű részmondat.

Épp a nyelet kell megtalálni a redukcióhoz.

#### 48) Mond ki az LR(k) grammaтика definícióját és magyarázz meg!

Egy kiegészített grammaтика LR(k) grammaтика ( $k \geq 0$ ), ha

$$S \Rightarrow^* \alpha Aw \Rightarrow \alpha \beta w$$

$$S \Rightarrow^* \gamma Bx \Rightarrow \gamma \delta x$$

$\alpha \beta y = \gamma \delta x$  és FIRST $_k(w)$  = FIRST $_k(y)$  esetén

$\alpha = \gamma$ ,  $\beta = \delta$  és  $A = B$ .

#### 49) Hogyan határozza meg az LR(0) elemző véges determinisztikus automatája, hogy léptetni vagy redukálni kell?

Az átmeneteit a verembe kerülő szimbólumok határozzák meg:

- léptetéskor terminális
- redukáláskor nemterminális

amikor elfogadó állapotba jut, akkor kell redukálni!

#### 50) Hogy néz ki egy LR(0)-elem és mi a jelentése?

Ha  $A \rightarrow \alpha$  a grammaтика egy helyettesítési szabálya, akkor az  $\alpha = \alpha_1 \alpha_2$  tetszőleges felbontás esetén  $[A \rightarrow \alpha_1 \dots \alpha_n]$  a grammaтика egy LR(0) eleme.

#### 51) Milyen műveletek segítségével állítjuk elő a kanonikus halmazokat és mi ezeknek a szerepe?

closure (lezárás) és read (olvasás) műveletek segítségével állítjuk elő a kanonikus halmazokat.

Ezen műveletek alkalmazásával tudjuk meghatározni a kanonikus halmazokat => az adott állapotból melyik állapotba tudunk továbbmenni.

#### 52) Mi köze van az LR(0) kanonikus halmazoknak az LR(0) elemző véges determinisztikus automatájához?

A végállapotok azok a kanonikus halmazok, amelyekben olyan elemek vannak, ahol a pont a szabály végén van.

#### 53) Hogyan határozzuk meg az LR(0) elemző automatájában az átmeneteket?

closure read kanonikus halmazokat alakítunk ki, ha nem végállapot van a halmazban, akkor léptetünk.

#### 54) Hogyan határozzuk meg az LR(0) elemző automatájában a végállapotokat?

closure read kanonikus halmazokat alakítunk ki, ha végállapot van, akkor redukálunk.

#### 55) Mond ki az LR(0)-elemzés nagy tételeit!

Egy  $\gamma$  járható prefix hatására az elemző automatája a kezdőállapotból olyan állapotba kerül, amelyhez tartozó kanonikus halmaz éppen a  $\gamma$  járható prefixre érvényes LR(0) elemeket

tartalmazza.

**56) Milyen konfliktusok lehetnek az LR(0) elemző táblázatban?**

léptetés/redukálás konfliktus: az egyik elem léptetést, egy másik redukálást ír elő  
redukálás/redukálás konfliktus:

az egyik elem az egyik szabály szerinti, a másik egy másik szabály szerinti redukciót ír elő  
Az LR(0) tulajdonság biztosítja a táblázat konfliktusmentes kitöltését!

**57) Milyen esetben ír elo redukciót az SLR(1) elemzés?**

Ha az előreolvasott szimbólum benne van a szabályhoz tartozó nemterminális FOLLOW1 halmazában.

**58) Hogy néz ki egy LR(1)-elem és mi a jelentése?**

Ha  $A \rightarrow \alpha$  a grammatika egy helyettesítési szabálya, akkor az  $\alpha = \alpha_1 \alpha_2$  tetszőleges felbontás és a terminális szimbólum (vagy  $a = \#$ ) esetén  $[A \rightarrow \alpha_1 \alpha_2, a]$  a grammatika egy LR(1)-eleme.

$A \rightarrow \alpha_1 \alpha_2$  az LR(1) elem magja, a pedig az előreolvasási szimbóluma.

**59) Milyen esetben ír elo redukciót az LR(1) elemzés?**

Ha az aktuális állapot **i**, és az előreolvasás eredménye az **a** szimbólum:

- ha  $[A \rightarrow \alpha_i, a] \in I_i$  ( $A = S'$ ), akkor redukálni kell  $A \rightarrow \alpha$  szabály szerint,

**60) Miért van általában lényegesen több állapota az LR(1) elemzőknek, mint az LR(0) (illetve SLR(1)) elemzőknek?**

Egy szabályt több jel is követhet. Mivel előreolvasunk mindenkor egy jelet, ha különbözőket olvasunk, az különböző állapotot jelent. Így ugyanaz az LR(0) illetve SLR(1) állapot többször is előfordul más előreolvasási szimbólummal.

**61) Mikor nevezünk két LR(1) kanonikus halmazt összevonhatónak?**

Ha a kanonikus halmaz párok csak az előreolvasási szimbólumokban különböznek.

**62) Hogyan kapjuk meg az LALR(1) kanonikus halmazokat?**

Az egyesíthető LR(1) kanonikus halmazokat vonjuk össze!

**63) Milyen fajta konfliktus keletkezhet a halmazok összevonása miatt az LALR(1) elemző készítése során?**

Redukálás-redukálás konfliktus.

**64) Milyen lépésekkel áll az LR elemzők hibaelfedo tevékenysége?**

- Hiba detektálása esetén meghívja a megfelelő hibarutint.
- A verem tetejéről addig töröl, amíg olyan állapotba nem kerül, ahol lehet az error szimbólummal lépni.
- A verembe lépteti az error szimbólumot.
- Az bemeneten addig ugorja át a soron következő terminálisokat, amíg a hibaalternatíva építését folytatni nem tudja.

**65) Mi az if □ then □ else probléma?**

A if ( F ) if ( S ) U else U részmondat hibaelfedo tevékenysége.

Nem egyértelmű a nyelvtan!

Problémát okoz mindegyik elemző esetén.

**66) Hogyan kell értelmezni a gyakorlatban az egymásba ágyazott elágazásokat, ha az az if □ then □ else probléma miatt nem egyértelmu?**

„Az else ág az öt közvetlenül megelőző if utasításhoz tartozik.”

**67) Hogyan oldják meg az if □ then □ else problémát az LR elemzők?**

Léptetéssel: Így az else az öt közvetlenül megelőző if utasításhoz fog tartozni.

**68) Mire kell figyelni programozási nyelvek tervezésekor, ha el akarjuk kerülni az if □ then □ else problémát?**

Vezessünk be if utasítás végét jelző kulcsszót.

**69) Milyen információkat tárolunk a szimbólumtáblában a szimbólumokról (fajtájuktól függetlenül)?**

Szimbólum neve,

Szimbólum attribútumai: definíció adatai, típus, tárgyprogram-beli cím, def. helye a forrásprogramban, szimbólumra hivatkozások a forrásprogramban.

**70) Milyen információkat tárolunk a szimbólumtáblában a változókról?**

típus

módosító kulcsszavak: const, static ...

címe a tárgyprogramban (függ a változó tárolási módjától)

**71) Milyen információkat tárolunk a szimbólumtáblában a függvényekről?**

paraméterek típusa

visszatérési típus

módosítók

címe a tárgyprogramban

**72) Milyen információkat tárolunk a szimbólumtáblában a típusokról?**

egyszerű típusok: méret

rekord: mezők nevei és típusleírói

tömb: elem típusleírója, index típusleírója, méret

intervallum-típus: elem típusleírója, minimum, maximum

unio-típus: a lehetséges típusok leírói, méret

**73) Milyen információkat tárolunk a szimbólumtáblában az osztályokról?**

Hasonlatos, mint a típusoknál a rekord.

Mezők nevei és típusleírói.

**74) Mi a szimbólumtábla két alapvető művelete és mikor használja ezeket a fordítóprogram?**

keresés: szimbólum használatakor,

beszúrás:

- új szimbólum megjelenésekor
- tartalmaz egy keresést is: „Volt-e már deklarálva?”

-

**75) Mi a változó hatóköre?**

hatókör: „Ahol a deklaráció érvényben van.”

**76) Mi a változó láthatósága?**

láthatóság: „Ahol hivatkozni lehet rá a nevével.”

- része a hatókörnek
- az elfedés miatt lehet kisebb, mint a hatókör

**77) Mi a változó élettartama?**

„Amíg memóriaterület van hozzárendelve.”

**78) Hogyan kezeljük változó hatókörét és láthatóságát szimbólumtáblával?**

A szimbólumokat egy verembe tesszük.

Keresés:

- a verem tetejéről indul
- az első találatnál megáll

Blokk végén a hozzá tartozó szimbólumokat töröljük.

**79) Milyen szerkezeteket szimbólumtáblákat ismersz?**

verem-, faszerkezetű-, hash-szerezetű-

**80) Miben tér el a névterek és blokkok kezelése a szimbólumtáblában?**

A using direktíva használatakor az importált névtér szimbólumait be kell másolni a verembe (vagy legalább hivatkozást tenni a verembe erre a névterre).

**81) Miért nem a szintaktikus elemző végzi el a szemantikus elemzés feladatait?**

A szintaktikus elemző környezetfüggetlen nyelvtannal dolgozik, míg a szemantikus környezetfüggővel, utóbbi több hibát kiszűr.

**82) Mi a különbség a statikus és a dinamikus típusozás között?**

Statikus: a kifejezésekhez fordítási időben a szemantikus elemzés rendel típust.

Dinamikus: a típusellenőrzés futási időben történik.

**83) Mi a különbség a típusellenőrzés és a típuslevezetés között?**

Típusellenőrzés:

- minden típus a deklarációkban adott
- a kifejezések egyszerű szabályok alapján típusozhatók
- egyszerűbb fordítóprogram, gyorsabb fordítás
- kényelmetlenebb a programozónak

Típuslevezetés, típuskikövetkeztetés:

- a változók, függvények típusait (általában) nem kell megadni
- a típusokat fordítóprogram „találja ki” a definíciójuk,
- használatuk alapján
- bonyolultabb fordítóprogram, lassabb fordítás
- kényelmesebb a programozónak

**84) Mi a fordítóprogram teendője típuskonverzió esetén?**

A típusellenőrzés során át kell írni a kifejezés típusát ha szükséges, akkor a tárgykódba generálni kell a konverziót elvégző utasításokat.

**85) Mik az akciós szimbólumok?**

A típusellenőrzés során át kell írni a kifejezés típusát ha szükséges, akkor a tárgykódba generálni kell a konverziót elvégző utasításokat.

**86) Mik az attribútumok?**

A nyelvtan szimbólumaihoz rendelt szemantikus típusok.

**87) Hogyan kapnak értéket az attribútumok?**

A szimbólumokhoz hozzárendeljük őket.

**88) Mi a szintetizált attribútum?**

A helyettesítési szabály bal oldalán áll abban a szabályban, amelyikhez az öt kiszámoló szemantikus rutin tartozik.

**89) Mi a kitüntetett szintetizált attribútum?**

Olyan attribútumok, amelyek terminális szimbólumokhoz tartoznak és kiszámításukhoz nem használunk fel más attribútumokat.

#### **90) Mi az örökolt attribútum?**

A helyettesítési szabály jobb oldalán áll abban a szabályban, amelyikhez az öt kiszámoló szemantikus rutin tartozik.

#### **91) Mivel egészítjük ki a nyelvtan szabályait attribútum fordítási grammatikák esetében?**

Megszorítással, hogy a kiértékelés egyszerűbb legyen:

⇒ partcionált attribútum fordítási grammatikák

⇒ rendezett attribútum fordítási grammatikák

#### **92) Mi a direkt attribútumfüggőség?**

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 függősségi gráfban ábrázolhatók.

#### **93) Mi az S □ ATG? Milyen elemzésekhez illeszkedik?**

Olyan attribútum fordítási grammatika, amelyben kizárolag szintetizált attribútumok vannak. A szemantikus információ a szintaxisfában a levelektől a gyökér felé terjed. Jól illeszthető az alulról felfelé elemzésekhez!

#### **94) Mi az L □ ATG? Milyen elemzésekhez illeszkedik?**

Olyan attribútum fordítási grammatika, amelyben minden  $A \rightarrow X_1 X_2 \dots X_n$  szabályban az attribútumértékek az alábbi sorrendben meghatározhatók:

A örökolt attribútumai

X1 örökolt attribútumai

X1 szintetizált attribútumai

X2 örökolt attribútumai

X2 szintetizált attribútumai

...

Xn örökolt attribútumai

Xn szintetizált attribútumai

A szintetizált attribútumai

Jól illeszkedik a felülről lefelé elemzésekhez.

#### **95) Mi az assembly?**

Programozási nyelvek egy csoportja.

#### **96) Mi az assembler?**

Az assembly programok fordítóprogramja.

#### **97) Milyen fobb regisztereket ismersz (általános célú, veremkezeléshez, adminisztratív célra)? eax, ebx, ecx, edx, esi, edi, ebp, esp, ...**

**98) Mi köze van egymáshoz az eax, ax, al, ah regisztereknek?**

eax 32 bitből áll: felső 16 bitnek nincs neve, alsó 16 bit: **ax**, ennek részei: felső bájt: **ah**, alsó bájt: **al**

**99) Milyen aritmetikai utasításokat ismersz assemblyben?**

inc, dec, add, sub, mul, div

**100) Mutasd be a logikai értékek egy lehetséges ábrázolását és a műveleteik megvalósítását assemblyben!**

**and al,bl** :Bitenkénti „és” művelet.

**or eax,dword [C]** :Bitenkénti „vagy” művelet.

**xor word [D],bx** :Bitenkénti „kizáró vagy” művelet.

**not bl** :Bitenkénti „nem” művelet.

**101) Milyen feltételes ugró utasításokat ismersz?**

je: „equal” - ugorj, ha egyenlő

jne: „not equal” - ugorj, ha nem egyenlő

jb: „below” - ugorj, ha kisebb ≡ jnae: „not above or equal” - nem nagyobb egyenlő

ja: „above” - ugorj, ha nagyobb ≡ jnbe: „not below or equal” - nem kisebb egyenlő

jnb: „not below” - nem kisebb ≡ jae: „above or equal” - nagyobb egyenlő

jna: „not above” - nem nagyobb ≡ jbe: „below or equal” - kisebb egyenlő

Ha előjeles egészekkel számolunk: jl („less”), jg („greater”), jnl, jng, jle, jge, . . .

**102) Hogyan kapják meg a feltételes ugró utasítások a cmp utasítás eredményét?**

A cmp eredményéppen egy flaget 0-ra vagy 1-re állít és azt vizsgálja meg az ugró utasítás.

**103) Milyen veremkezelő utasításokat ismersz assemblyben, és hogyan működnek ezek?**

push eax: eax-et a verembe dobja. pop ebx: a verem tetején levőt ebx-be teszi.

**104) Melyik utasításokkal lehet alprogramot hívni és alprogramból visszatérni assemblyben?**

call: alprogram meghívása, ret: visszatérés

**105) Hogyan generálunk kódot egyszerű típusok értékadásához?**

a kifejezést az eax regiszterbe kiértékelő kód

**mov [Változó],eax**

**106) Hogyan generálunk kódot egy ágú elágazáshoz?**

a feltételt az al regiszterbe kiértékelő kód:

cmp al,1

je Then

jmp Vége

Then: a then-ág programjának kódja

Vége:

**107) Hogyan generálunk kódot több ágú elágazáshoz?**

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:

**108) Hogyan generálunk kódot előtesztelő ciklushoz?**

Eleje: a ciklusfeltétel kiértékelése az al regiszterbe

  cmp al,1

  jne near Vége

  a ciklusmag programjának kódja

  jmp Eleje

  Vége:

**109) Hogyan generálunk kódot háltutesztelő ciklushoz?**

Eleje: a ciklusmag programjának kódja

  a ciklusfeltétel kiértékelése az al regiszterbe

  cmp al,1

  je near Eleje

**110) Hogyan generálunk kódot for ciklushoz?**

a „from” érték kiszámítása a [Változó] memória helyre

  Eleje: a „to” érték kiszámítása az eax regiszterbe

  cmp [Változó],eax

  ja near Vége

  a ciklusmag kódja

  inc [Változó]

  jmp Eleje

  Vége:

**111) Hogyan generáljuk kezdoérték nélküli statikus változó definíciójának assembly kódját?**

section .bss

; a korábban definiált változók...

Lab12: resd 1 ; 1 x 4 bájtnyi terület

**112) Hogyan generáljuk kezdoértékkal rendelkezo statikus változó definíciójának assembly kódját?**

section .data

; a korábban definiált változók...

Lab12: dd 5 ; 4 bájton tárolva az 5-ös érték

**113) Hogyan generáljuk aritmetikai kifejezés kiértékelésének assembly kódját? (konstans, változó, beépített függvény)**

konstans: mov eax,25

változó:mov eax,[X]

beépített függvény:

  ; a 2. kifejezés kiértékelése eax-be

  push eax

  ; az 1. kifejezés kiértékelése eax-be

  pop ebx

  add eax,ebx

**114) Mutasd meg a különbséget a minden részkifejezést kiértékelő és a rövidzárás logikai operátorok assembly kódja között!**

(skip – ezt kikell keresni)

**115) Hogyan generáljuk a goto utasítás assembly kódját?**

goto Lab; => jmp Lab

**116) Miért nehéz a break utasítás kódgenerálását megoldani S □ATG használata esetén?**

A Vége címkét a ciklus feldolgozásakor generáljuk, pedig szükség van rá a break kódjában is! Azaz ez a címke egy örökolt attribútum...

**117) Mit csinál a call és a ret utasítás?**

A **call** Címke az eip regiszter tartalmát a verembe teszi, ez a call utáni utasítás címe, visszatérési címnek nevezzük.

Átadja a vezérlést a Címke címkéhez mint egy ugró utasítás

A **ret** kiveszi a verem legfelső négy bajtját és az eip regiszterbe teszi mint egy pop utasítás. A program a veremben talált címnél folytatódik

**118) Hogyan adjuk át assemblyben az alprogramok paramétereit és hol lesz a lefutás után a visszatérési érték (C stílus esetén)?**

a paramétereket a verembe kell tenni a call utasítás előtt

C stílusú paraméterátadás: fordított sorrendben tesszük a verembe

- az utolsó kerül legalulra
- az első a verem tetejére

az ejárásból való visszatérés után a hívó állítja vissza a vermet és a visszatérési érték: az eax regiszterbe kerül

**119) Hogyan épül fel az aktivációs rekord?**

lokális változók <= esp

előző aktivációs rekord bázispointere <= ebp

visszatérési cím

1. paraméter
2. paraméter .....

**120) Mi a bázismutató és melyik regisztert szoktuk erre a célra felhasználni?**

Aktivációs rekordban az előző aktivációs rekordra mutató pointer, **ebp** regiszter.

**121) Hol tároljuk alprogramok lokális változóit?**

A verem tetején tároljuk őket.

**122) Mi a különbség az érték és a hivatkozás szerinti paraméterátadás assembly kódja között?**

érték szerint:

- a paraméterértékeket másoljuk a verembe
  - ha az alprogram módosítja, az nem hat az átadott változóra
- hivatkozás szerint
- az átadandó változóra mutató pointert kell a verembe tenni
  - az alprogramban a lokális változó kiértékelése is módosul

**123) Milyen csoportokba oszthatók a változók tárolásuk szerint és a memória mely részeiben tároljuk az egyes csoportokba tartozó változókat?**

statikus memóriakezelés:

- a .data vagy .bss szakaszban
  - globális vagy statikusnak deklarált változók
  - előre ismerni kell a változók méretét, darabszámát
- dinamikus memóriakezelés
- blokk-szerkezethez kötődő, lokális változók: verem
  - tetszőleges élettartamú változók: heap memória

**124) Mi a különbség a lokális és a globális optimalizálás között?**

lokális: kis programrészletek átalakítása

globális: a teljes program szerkezetét kell vizsgálni

**125) Mit jelent a gépfüggo optimalizálás?**

gépfüggő optimalizálás: az adott architektúra sajátosságait használja ki

**126) Mit nevezünk alapblokknak és melyik optimalizálás során van szerepe?**

Egy programban egymást követő utasítások sorozatát alapblokknak nevezzük, ha:

- az első utasítás kivételével egyik utasítására sem lehet távolról átadni a vezérlést
- az utolsó utasítás kivételével nincs benne vezérlés-átadó utasítás
- az utasítás-sorozat nem bővíthető a fenti két szabály megsértése nélkül

Lokális optimalizálás során van szerepe: egy alapblokkon belüli átalakításnál.

**127) Mutass példát konstansok összevonására!**

a:= 1+b+3+4; helyett a:= b+8;

**128) Mutass példát konstans továbbterjesztésére!**

a:= 6;	helyett	a:=6;
b:= a/2;		b:=3;
c:=b+5;		c:=8;

**129) Mutass példát azonos kifejezések többszöri kiszámításának elkerülésére!**

x := 20 - (a * b);	helyett	t:=a*b;
y := (a * b) ^ 2;		x:=20-t;
		y:=t^2;

**130) Mi az ablakoptimalizálási technika lényege?**

egyszerre csak egy néhány utasításnyi részt vizsgálunk a kódból , majd a vizsgált részt előre megadott mintákkal hasonlítjuk össze. Ha illeszkedik, akkor a mintához megadott szabály szerint átalakítjuk és ezt az „ablakot” végigcsúsztatjuk a programon.

**131) Mi a különbség a kódkiemelés és a kódsüllyesztés között?**

Kódkiemelésnél a blokk elé visszük ki a változót kódsüllyesztésnél mögé.

**132) Hogyan változtatja a program sebességét és méretét a ciklusok kifejtése?**

Mérlegelni kell, hogy a méret és a sebesség mennyire fontos...

**133) Mi a frekvenciaredukálás lényege?**

Költséges utasítások „átköltözése” ritkábban végrehajtó alapblokkba.

**134) Mutass példát eros redukcióra!**

Helyett

```
for( int i=a; i<b; i+=c ){  
cout << 3*i;  
}
```

```
int t1 = 3*a;  
int t2 = 3*c;  
for( int i=a; i<b; i+=c ){  
cout << t1; t1 +=t2;  
}
```

by Varga Mátyás (VAMQAAI.ELTE)

2011.01.12.

Megjegyzés: A várható kérdések **javarészét** lefedi, ha tudod ezeket, jó eséllyel meglesz a beugró, de előfordulhat pár másik kérdés is. 2010-2011-1 félévnek előadás diái alapján kidolgozva Dévai Gergelynél.

# Fordítóprogramok (A,C,T szakirány)

## Feladatgyűjtemény

ELTE IK

### 1 Lexikális elemzés

1. Add meg reguláris nyelvtannal, reguláris kifejezéssel és véges determinisztikus automatával a következő lexikális elemeket!
  - (a) egész szám (legalább egy számjegy 0-9-ig)
  - (b) olyan egész szám, amely több számjegy esetén nem kezdődhet nullával
  - (c) előjeles egész szám (opcionális + vagy - az elején)
  - (d) törtszám (tizedespont előtt legalább egy számjegy)
2. Add meg reguláris kifejezéssel és véges determinisztikus automatával a következő lexikális elemeket!
  - (a) azonosító (betűvel kezdődik, számmal vagy betűvel vagy \_ jellel folytatódik)
  - (b) egysoros megjegyzés // -től a sor végéig
  - (c) többsoros megjegyzés /\* -tól \*/ -ig
  - (d) sztring
    - "alma"
    - "a \" egy idézőjel a sztringben"
    - "a \\ egy backslash a sztringben"
    - backslash után csak idézőjel vagy backslash állhat
  - (e) fehér szóközök (legalább egy space, tab vagy sorvége)
3. Rajzold fel a lexikális elemző véges determinisztikus automatáját, ha a következő szimbólumokat szeretnénk felismerni:

++ += -- -= -> >= >> >>= <<=

4. A következőkben az ábécé a  $\{a, b\}$  halmaz.
  - (a) Ha a változók egybetűsek, ab és aba kulcsszavak, milyen lexikális elemekre kell felbontani az abab szöveget?
  - (b) Ha a változók egybetűsek, aa, ab és aba kulcsszavak, milyen lexikális elemekre kell felbontani az aabaa szöveget?
  - (c) Hogyan kell ezt lexikális elemző generátor (pl. *flex*) segítségével implementálni?
5. Készíts programot a + ++ += és - szimbólumok felismeréséhez (lexikális elemző generátor használata nélkül)!
  - (a) Készísd el az egyetlen szimbólum felismerésére képes automata implementációját!
  - (b) Fejleszd tovább a programot egy teljes lexikális elemzővé, ami több egymást követő szimbólumot is képes felismerni!
6. Lexikális elemző generátor (pl. *flex*) segítségével készíts programot az alábbi feladatokra!
  - (a) a teljes bemenetet a kimenetre másolni

- (b) a szövegben a *username* szót a saját felhasználói nevedre cserélni  
 (c) programkód  $\rightsquigarrow$  HTML konverter
  - *tab*  $\rightsquigarrow$  &nbsp;
  - *sorvége*  $\rightsquigarrow$  <BR>
  - < $\rightsquigarrow$  &lt;
  - > $\rightsquigarrow$  &gt;
 (d) minden sor végén kiírni a sor számát és hosszát  
 (e) minden szóhoz kiírni a pozícióját (sor, oszlop) és magát a szót
7. Egy nyelvben a *begin* és az *end* kulcsszavak, a változók betűkből és számjegyekből állnak, de csak betűvel kezdődhetnek, a számkonstansok pedig számjegyekből állhatnak. A fehér szóközök szóközökből, tabokból és sorvégékből állhatnak.
- (a) Milyen lexikális elemekre bontja ennek a nyelvnek a lexikális elemzője az alábbi szöveget?
- ```
alma123 beginend begin 12
```
- (b) Ha ezt az elemzőt *flex* segítségével írjuk, milyen sorrendben kell megadni az egyes lexikális elemek reguláris kifejezéseit?
8. Egyes programozási nyelvekben a hexadecimális (16-os számrendszerben ábrázolt számok) a következő alakúak:
- a "számjegyeik": 0, 1, ..., 9, *A*, *B*, ..., *F*
  - a szám nem kezdődhet betűvel (*A*, *B*, ..., *F*)
  - a szám végén a *h* karakter áll
- Példák helyes hexadecimális számokra: 12AF*h* 123*h* 0*h* 0DE*h*  
 Példák helytelen hexadecimális számokra: DE*h* D1*h* *h* 0hh 123
- (a) Írd fel az ilyen hexadecimális számokat leíró reguláris kifejezést!  
 (b) Rajzold fel az ilyen hexadecimális számokat felismerő véges determinisztikus automatát!  
 (c) Sorszámozd az automata állapotait! Írd fel azt az állapot sorozatot, amin az automata keresztülmegy, miközben a 0DE*h* számot felismeri!
9. Egy nyelvben a változók betűkből és számjegyekből állnak, és felkijáltójellel kell kezdődniük. A számkonstansok számjegyekből, a fehér szóközök pedig szóközökből, tabokból és sorvégékből állhatnak. A nyelvben értékkedás és elágazás van az alábbi példaprogram szerint:
- ```
!x1 := 5
ha !x1 paros akkor !2y := 4
ha !2y paratlan akkor !x1 := 2
```
- Az elágazás feltétele csak párosságot és páratlanságot tud vizsgálni, az elágazás belsejében pontosan egy értékkedás lehet.
- (a) Határozd meg a nyelv lexikális elemeit és írd le őket reguláris kifejezéssel!  
 (b) Milyen sorrendben kell ezeket megadni *flex*-ben és miért így?  
 (c) Készíts a változóneveket felismerő véges determinisztikus automatát!
10. Készíts reguláris kifejezéseket és véges determinisztikus automatákat az alábbi lexikális elemekhez!
- (a) *a*, *b* és *c* betűkből és pontokból álló nem üres sorozat  
 (b) *a*, *b* és *c* betűkből és pontokból álló nem üres sorozat, amely nem kezdődhet és nem végződhet ponttal és két pont nem állhat egymás mellett

- (c)  $a$ ,  $b$  és  $c$  betűkből és pontokból álló tetszőleges sorozat, amelyben az egymás mellett álló betűknek ábécé-sorrendben kell lennie
- példák helyes elemekre:  $\epsilon \dots a b.a. .aabcc.bc$
  - példák helytelen elemekre:  $ba .c.ba.$
11. Egy nyelv kulcsszavai legyenek **abba** és **abc**, változónevei pedig a 10. feladat (c) pontjának megfelelő lexikális elemek.
- Milyen sorrendben kell feltüntetni ezeket a reguláris kifejezéseket Flex-ben?
  - Milyen lexikális elemekre bontja az elemző az alábbi szövegeket?
    - abbaabc
    - abbaabc.
    - .abba.abc

## 2 LL elemzések

1. Dönts el, hogy az alábbi nyelvtanok közül melyik egyszerű LL(1)-es!

- $S \rightarrow aSA \mid A$   
 $A \rightarrow bA \mid a$
- $S \rightarrow aSA \mid bA$   
 $A \rightarrow \epsilon \mid bA$
- $S \rightarrow aSA \mid bA$   
 $A \rightarrow a$

Amelyik igen, ahhoz készítsd el az elemző táblázatot és elemezd az alábbi szövegeket!

- aabaaa
- aabbaa
- aabaa

2. Dönts el, hogy az alábbi nyelvtanok közül melyik  $\epsilon$ -mentes LL(1)-es!

- $S \rightarrow aSA \mid A$   
 $A \rightarrow bA \mid a$
- $S \rightarrow SA \mid Bc$   
 $A \rightarrow aA \mid c$   
 $B \rightarrow b \mid A$
- $S \rightarrow dSdS \mid A \mid eB$   
 $A \rightarrow a \mid BA$   
 $B \rightarrow b \mid c$

Amelyik igen, ahhoz készítsd el az elemző táblázatot! Válassz egy olyan szöveget, amelynek levezetéséhez mindegyik szabályt pontosan egyszer kell felhasználni és elemezd!

3. Milyen mondati vannak az alábbi grammatika által definiált nyelvnek?

$$\begin{aligned} S &\rightarrow [A] \\ A &\rightarrow \epsilon \mid nB \\ B &\rightarrow \epsilon \mid ,nB \end{aligned}$$

Mutasd meg, hogy ez egy LL(1) nyelvtan, készítsd el az elemző táblázatot, és elemezz egy 5 szimbólumból álló mondatot. (Hány 5 szimbólumból álló mondat van a nyelvben?)

4. Elemzőt szeretnénk készíteni egy nyelvhez, aminek egy példaprogramja:

```

if ?a1 then
    ?a1 := false
endif
while ?a3 do
    ?a3 := ?a2
    ?a2 := ?a1
done

```

A nyelvben minden változó `?`  jellel kezdődik, deklarálni nem kell őket, mindegyik logikai típusú (`true`, `false`). A nyelvben van értékadás, elágazás és ciklus. Az értékadás baloldalán változó, jobboldalán változó, `true` vagy `false` állhat. A ciklusok és elágazások feltétele minden egyetlen változó. Az egész program és a ciklusok, elágazások törzse is lehet üres.

- (a) Határozd meg a lexikális elemeket és írj hozzájuk reguláris kifejezéseket!
  - (b) Írd fel a nyelvtan szabályait!
  - (c) Ellenőrizd, hogy LL(1)-e a nyelvtan!
  - (d) Készíts hozzá rekurzív leszállásos elemzőt!
5. C++ függvénydeklarációt szeretnénk szintaktikus elemzővel elemezni.  
A lexikális elemek legyenek: `a` `(` `)` `*` `:`  
A nyelvtan szabályai pedig:
- $$S \rightarrow \underline{aa}(L);$$
- $$L \rightarrow \epsilon \mid \underline{aa}F$$
- $$F \rightarrow \epsilon \mid , \underline{aa}F$$
- (a) Igaz-e, hogy a grammatika egyszerű LL(1),  $\epsilon$ -mentes LL(1), általános LL(1)?
  - (b) Készítsd el hozzá az elemző táblázatot és elemezd az `aa(aa,aa)`; szöveget!
  - (c) Milyen eljárás tartozik a nyelvtan rekurzív leszállásos elemzőjében az  $L$  nemterminális szimbólumhoz?
6. Készíts elemzőt a 9. feladatban adott nyelvhez!
- (a) Írd fel a nyelv nyelvtanát!
  - (b) Készíts LL elemzőt a nyelvtanhöz!
  - (c) Elemezd a `ha 2 paros akkor !x := 0` mondatot!
    - Írd fel hozzá a lexikális elemző kimenetét!
    - Elemezd az elemző táblázat segítségével!
7. Egy olyan programozási nyelvhez szeretnénk szintaktikus elemzőt írni, amelyikben három lexikális elem van: `begin`, `end` és `skip`.  
A nyelv szintaxisát a következő nyelvtan adja meg:
- $$S \rightarrow \underline{skip} \mid BS$$
- $$B \rightarrow \underline{begin} S \underline{end}$$
- (a) Igaz-e, hogy a grammatika egyszerű LL(1),  $\epsilon$ -mentes LL(1), általános LL(1)?
  - (b) Készítsd el hozzá az elemző táblázatot és elemezd a `begin skip end skip` szöveget!

### 3. LR elemzések

1. Írj fel nyelvtanokat az alábbi nyelvi elemek szintaxisának leírásához! Elemzőgenerátor (pl. `bisonc++`) segítségével készíts hozzájuk szintaktikus elemző programokat!
- (a) Lista  
`[ alma, körte, szilva ]`

Terminális szimbólumok: nyitó- és csukó zárójel, vessző, listaelem

- (b) C stílusú függvénydeklarációk sorozata

```
char betuje( string s, int index );
int osszeg( int x, int y );
```

Terminális szimbólumok: azonosító, nyitó- és csukó zárójel, vessző

(A típusok, függvények és paraméterek neve egyaránt azonosító.)

- (c) Blokkstrukturált nyelv *skip* utasítással.

```
begin
  skip
  begin end
end
begin
  begin skip end
  skip skip skip
end
```

Terminális szimbólumok: begin, end és skip

- (d) A nulladrendű logika nyelve

```
(a <-> b) -> (!a | b) & true
```

Terminális szimbólumok: true, false, változónév, nyitó- és csukó zárójelek, továbbá az öt logikai összekötőjel a precedencia szerint csökkenő sorrendben: negáció, konjunkció, diszjunkció, implikáció, ekvivalencia

2. Adott az alábbi nyelvtan:

$$\begin{aligned} S &\rightarrow \underline{x} \mid AS \\ A &\rightarrow \underline{a} S \underline{b} \end{aligned}$$

- (a) Alakítsd át kiegészített nyelvtanná!
- (b) Számold ki az  $LR(0)$  kanonikus halmazait!
- (c) Rajzold fel az elemző automatáját és törlsd ki az elemző táblázatot!
- (d) Elemezd az  $axbx$  szöveget!

3. Adott a következő nyelvtan:

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow \epsilon \mid \underline{\text{skip}} S \mid BS \\ B &\rightarrow \underline{\text{begin}} S \underline{\text{end}} \end{aligned}$$

- (a) Mutasd meg, hogy a nyelvtan nem  $LR(0)$ ! Milyen konfliktusok vannak az  $LR(0)$  elemző táblázatában?
- (b) Igaz-e, hogy a nyelvtan  $SLR(1)$ ? Számold ki a szükséges *FOLLOW* halmazokat és törlsd ki az  $SLR(1)$  elemző táblázatot!
- (c) Elemezd a  $\text{skip begin begin end}$  szöveget és figyeld meg, hol jön rá az elemző a hibára!

4. Adott a következő nyelvtan:

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow \underline{id} \mid \underline{id} + S \end{aligned}$$

- (a) Mutasd meg, hogy a nyelvtan nem  $LR(0)$ ! Milyen konfliktusok vannak az  $LR(0)$  elemző táblázatában?
- (b) Igaz-e, hogy a nyelvtan  $SLR(1)$ ? Számold ki a szükséges *FOLLOW* halmazokat és törlsd ki az  $SLR(1)$  elemző táblázatot!
- (c) Elemezd az  $\underline{id} + + \underline{id}$  szöveget és figyeld meg, hol jön rá az elemző a hibára!

5. Adott a következő nyelvtan:

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow B \sqsubseteq J \mid J \\ B &\rightarrow * J \mid \underline{id} \\ J &\rightarrow B \end{aligned}$$

- (a) Mutasd meg, hogy a nyelvtan nem  $SLR(1)$ !
- (b) Igaz-e, hogy a nyelvtan  $LR(1)$ ?
- (c) Készítsd el az  $LR(1)$  elemző táblázatot és elemezd az id=\*id szöveget!
6. Készíts  $LALR(1)$  elemzőt az alábbi nyelvtanhoz!
- $$S' \rightarrow S$$
- $$S \rightarrow \underline{x} \mid \underline{y} S \underline{y}$$
7. Adott a következő nyelvtan:
- $$S' \rightarrow S$$
- $$S \rightarrow A\underline{b} \mid \underline{a}c \mid \underline{d}A\underline{c}$$
- $$A \rightarrow \underline{a} \mid \underline{a}A$$
- (a) Számítsd ki az  $LR(0)$  kanonikus halmazokat!
- (b) Add meg az összes nemterminálishez a  $FOLLOW_1$  halmazaikat!
- (c) Az előzőek segítségével töltsd ki az  $SLR(1)$  elemző táblázatot! Jelöld a táblázatban a konfliktusokat!
- (d) Számítsd ki az  $LR(1)$  kanonikus halmazokat!
- (e) Melyek az összevonható halmazok?
- (f) Töltsd ki az  $LALR(1)$  elemző táblázatot!
- (g) Elemezd az ab mondatot!

## 4 Assembly programozás

1. Definiálj olyan adatterületet, ami az X és Y címkéktől kezdve 4-4 bájton a 10 és 5 értékeket tárolja, és a Z címkétől kezdve 4 bájtot lefoglal kezdeti érték nélkül.  
Írj olyan asszembly kódrészletet, ami
- megnöveli az X címkétől kezdődő 4 bájt értékét
  - átmásolja az Y-tól kezdődő 4 bájtot az X-től kezdődő 4 bájtba
  - kiszámolja a szorzatukat a Z-től kezdődő 4 bájtba
  - bitenként összeéseli őket és az eredményt a Z-től kezdődő 4 bájtba teszi
2. Írj olyan kódrészletet, ami az abszolút értékét számolja ki eax értékének!
3. Írj olyan kódrészletet, ami bekér két számot és a másodikat kiírja annyiszor, amennyi az első értéke.
4. Írj olyan kódrészletet, ami kiszámolja két pozitív szám legnagyobb közös osztóját!
5. Írj programot, ami egészeket olvas be, tárolja őket tömbben tömbben, majd kiírja a tömb tartalmát.
6. Írj programot, ami tömbben tárolt számokat összegez.
7. Írj programot, ami egy karaktertömbben megszámolja az 'a' karaktereket.
8. Írj programot, ami egészek tömbjében megkeresi a legnagyobb számot.
9. Írj programot, ami bekér egy nemnegatív egész számot, majd átalakítja szöveggé (10-es számrendszer szerint). Az eredményt egy karaktertömbbe kell írni majd kiírni a képernyőre.

## 5 Kódgenerálás

- Írd fel a szimbólumtábla tartalmát (változó neve, típusa, címkéje) majd rögzített kódgenerálási szabályok alapján add meg a generált assembly kódot az alábbi kódrészlethez:

```
int x;  
int y;  
x = 2*x+y;
```

- Rögzített kódgenerálási szabályok alapján add meg a generált assembly kódot az alábbi kódrészlethez:

```
while(x < 10)  
    if( b )  
        x = x+1;  
    else  
        x = 10;
```

**Fordítóprogramok célja és szerkezete**

Fordítóprogramok előadás (A,C,T szakirány)

1 Fordítóprogramok előadás (A,C,T szakirány) Fordítóprogramok célja és szerkezete

Fordítóprogramról általában Fordítóprogramok célja

Miért van szükség fordítóprogramokra?

Így kényelmes programozni

```
int sum = 0;
for( int i=0; i<len; ++i )
    sum += t[i];
```

2 Fordítóprogramok előadás (A,C,T szakirány) Fordítóprogramok célja és szerkezete

Fordítóprogramról általában Fordítóprogramok célja

Miért van szükség fordítóprogramokra?

Így kényelmes programozni

```
int sum = 0;
for( int i=0; i<len; ++i )
    sum += t[i];
```

Ezt tudja végrehajtani a számítógép

```
B9 00 00 00 00
B8 00 00 00 00
81 F9 0A 00 00 00
7D 06
03 04 8B
41
EB F2
```

2 Fordítóprogramok előadás (A,C,T szakirány) Fordítóprogramok célja és szerkezete

Fordítóprogramról általában Fordítóprogramok célja

Miért van szükség fordítóprogramokra?

- magasszintű programozási nyelvek
  - könnyebb programozni
  - közlebb a megoldandó problémához
  - platform-függetlenség
- gépi kód
  - gépközeli (numerikus utasítások, regiszterek, memóriahivatkozások ...)
  - erősen platform-függő
  - optimizált
- Fordítóprogramokat általában a kettő közötti átalakításra használunk.

3 Fordítóprogramok előadás (A,C,T szakirány) Fordítóprogramok célja és szerkezete

Fordítóprogramról általában Assembly, gépi kód

Assembly és assembler

- az assembly nyelvben
  - a gépi kód utasításaihoz neveket (*mnemonikokat*) rendelünk
  - a regiszterekre névvel hivatkozunk
  - az egyes memoriacímeket címkékkel azonosítjuk
  - gyakran közülös állomás a magasszintű nyelvről gépi kódra történő fordítás során
- assembler: assembly nyelvről gépi kódra fordít

4 Fordítóprogramok előadás (A,C,T szakirány) Fordítóprogramok célja és szerkezete

Fordítóprogramról általában Assembly, gépi kód

Assembly és assembler

Gépi kód	Assembly
----------	----------

```
B9 00 00 00 00
B8 00 00 00 00
81 F9 0A 00 00 00
7D 06
03 04 8B
41
EB F2
```

```
mov ecx,0
mov eax,0
eleje:
cmp ecx,10
jge vege
add eax,[ebx+4*ecx]
inc ecx
jmp eleje
vege:
```

5 Fordítóprogramok előadás (A,C,T szakirány) Fordítóprogramok célja és szerkezete

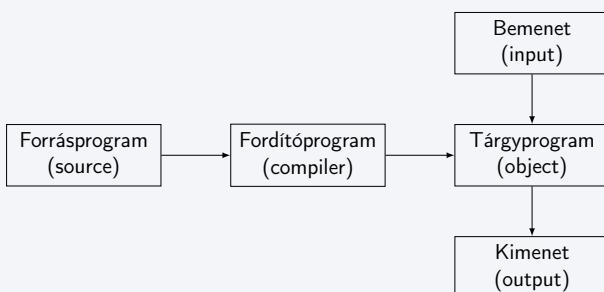
## Példák

- magasszintű nyelvek fordítóprogramjai (compiler)
  - gcc, javac, ghc ...
- assembly nyelvek fordítóprogramjai (assembler)
  - nasm, masm, gas ...
- nyelvek közötti fordítás (translator)
  - Fortran  $\Rightarrow$  C, latex2html, dvips ...
- egyéb fordítóprogramok
  - latex
  - hardware leíró nyelvből áramkörök tervezés

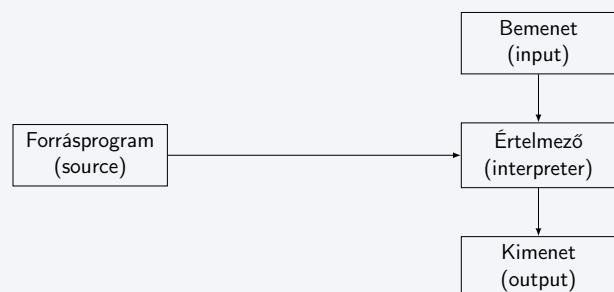
## Fordítás és interpretálás

- Fordítás:** A forráskódot a fordítóprogram **tárgyprogrammá** alakítja. A tárgyprogramokból a szerkesztés során **futtatható állomány** jön létre.
  - Fordítási idő:** amikor a fordító dolgozik
  - Futási idő:** amikor a program fut
- Interpretálás:** A forráskódot az interpreter értelmezi és azonnal végrehajtja.
  - a fordítási és futási idő nem különbözik

## Fordítás



## Interpretálás



## Fordítás vs. interpretálás

### Fordítás

- gyorsabb végrehajtás
- a forrás alaposabb ellenőrzése
- minden platformra külön-külön le kell fordítani
- C++, Ada, Haskell ...

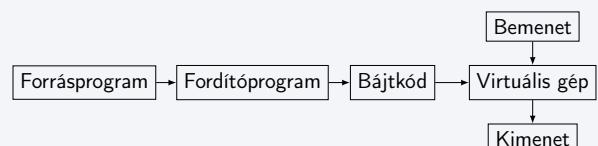
### Interpretálás

- rugalmasabb (pl. utasítások fordítási időben történő összeállítása)
- minden platformon azonnal futtatható, ahol az interpreter rendelkezésre áll
- Perl, Php, Javascript ...

## Fordítás + interpretálás

Fordítás és interpretálás egymásra építve (pl. Java):

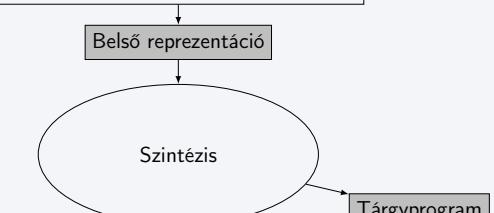
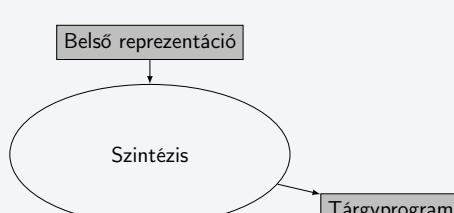
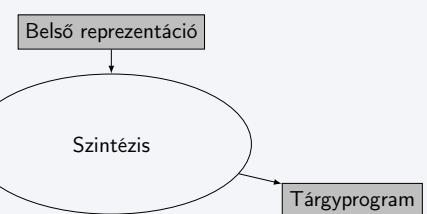
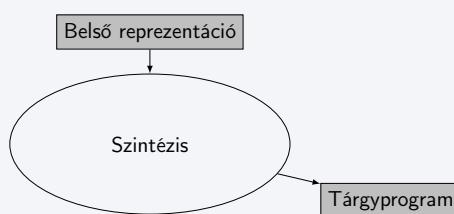
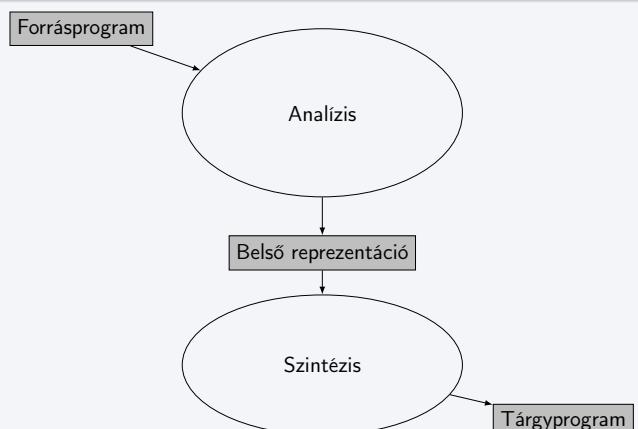
- A forráskód fordítása **bájtkódra**.
- A bájtkód interpretálása **virtuális géppel**.



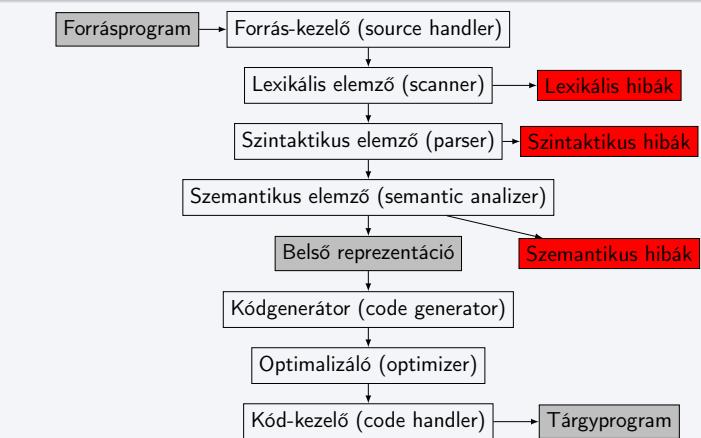
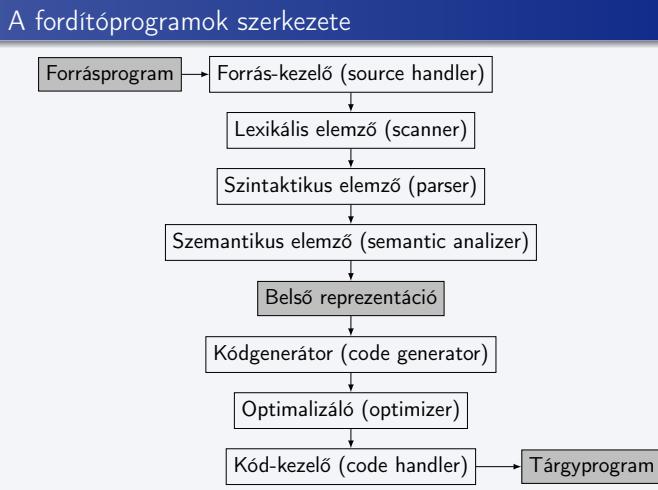
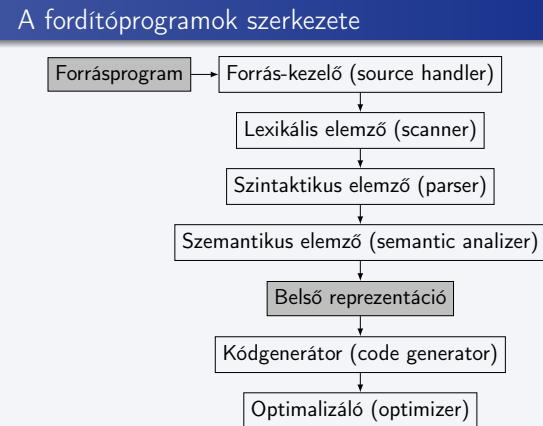
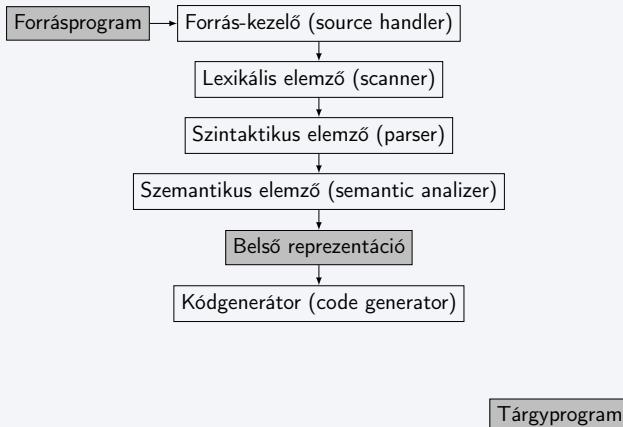
**Virtuális gép:** olyan gép szoftveres megvalósítása, amelynek a bájtkód a gépi kódja.

## Fejlődés

- 1957, az első Fortran compiler: 18 emberévnyi munka
- azóta fejlődött a formális nyelvek és automaták elmélete
- ma: a fordítóprogramok létrehozásának egy része automatizálható
  - a programszöveget (szintaktikusan) elemző program: automatikusan generálható
  - a szemantikus ellenőrzést (pl. típusok) és kódgenerálást végző program: nem generálható automatikusan, de nem túl bonyolult
  - kódoptimalizálás: nehéz feladat
- Példa: elemzőgenerátorok



## A fordítóprogramok szerkezete



## Forrás kezelő

- bemenet: fájl(ok)
- kimenet: karakterszorozat
- feladata:
  - fájlok megnyitása
  - olvasás
  - pufferelés
  - az operációsrendszer-specifikus feladatok elvégzése
  - ha a fordítóprogram készít listafájlt, akkor ezt is kezeli

## Példa

input.cpp  $\Rightarrow$   $x_1 = x_1 + 1;$ 

## Lexikális elemző

- bement: karakterszorozat
- kimenet: szimbólumsorozat, lexikális hibák
- feladata:
  - lexikális egységek (szimbólumok) felismerése: azonosító, konstans, kulcsszó...

## Példa

 $x_1 = x_1 + 1;$   
 $\Rightarrow$   
 változó[x] operátor[=] változó[x] operátor[+] konstans[1] utasításvége

## Szintaktikus elemző

- bemenet: szimbólumsorozat
- kimenet: szintaxisfa, szintaktikus hibák
- feladata:
  - a program szerkezetének felismerése
  - a szerkezet ellenőrzése: megfelel-e a nyelv definíciójának?



## Szemantikus elemző

- bemenet: szintaktikusan elemzett program (szintaxisfa, szimbólumtábla, ...)
- kimenet: szemantikusan elemzett program, szemantikus hibák
- feladata:
  - típusellenőrzés
  - változók láthatóságának ellenőrzése
  - eljárások hívása megfelel-e a szignatúrának?
  - stb.

### Szintaktikus hibák

```
x = x 1;
x = x + ;
if x==0) {}
```

### Szemantikus hibák

```
if( "alma" == 3.14 ) {}
void f(int x) {} f(3,4);
{ int y; } y++;
```

## Kódgenerátor

- bemenet: szemantikusan elemzett program
- kimenet: tárgykód utasításai (pl. assembly, gépi kód)
- feladata:
  - a forrásprogrammal ekvivalens tárgyprogram készítése

### Az $x = x + 1$ ; utasítás egy lehetséges megvalósítása

```
mov eax,1
push eax
mov eax,[x]
pop ebx
add eax,ebx
mov [x],eax
```

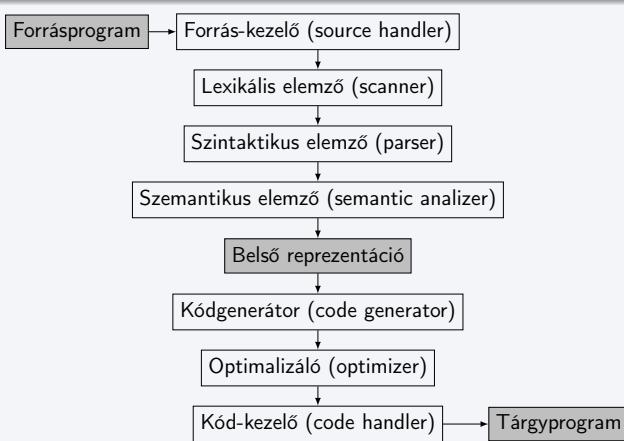
## Kódpotimalizáló

- bemenet: tárgykód
- kimenet: optimalizált tárgykód
- feladata: valamelyen szempontok szerint jobb kód készítése
  - pl. futási sebesség növelése, méret csökktése
  - pl. felesleges utasítások megszüntetése, ciklusok kifejtése...
- optimalizáció végezhető
  - az eredeti programon (szemantikus elemzés után)
  - a tárgykódon (a kódgenerálás után)

### Az $x = x + 1$ ; utasítás kódjának optimalizálása

```
mov eax,1
push eax
mov eax,[X]
pop ebx
add eax,ebx
mov [X],eax
      ==>
inc dword [X]
```

## Emlékeztető: a fordítás komponensei



## A fordítás menetei

- az egyes komponensek egymást követik
  - az egyik kimenete a másik bemenete
- ez nem jelenti azt, hogy pl. a lexikális elemzőnek be kell fejezni a szintaktikus elemzés előtt
  - a szintaktikus elemző meghívhatja a lexikálisat, amikor egy újabb szimbólumra van szüksége
- eredmény: 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

A fordítás annyi menetes, ahányszor a programszöveget (vagy annak belső reprezentációját) végigolvassa a fordító a teljes fordítási folyamat során.

## A lexikális elemzés

Fordítóprogramok előadás (A,C,T szakirány)

## A lexikális elemzés helye



1 Fordítóprogramok előadás (A,C,T szakirány) A lexikális elemzés

2 Fordítóprogramok előadás (A,C,T szakirány) A lexikális elemzés

## Elemzési lépések szétválasztása

- lexikális elemzés: megadható reguláris nyelvtannal (3-as)
- szintaktikus elemzés: megadható környezetfüggetlen nyelvtannal (2-es)
- szemantikus elemzés: környezetfüggő (1-es)
  - pl. egy változó használatának helyessége függhet a deklarációjától, azaz a környezettől

A három lépés szétválasztásának oka, hogy **egyszerű feladathoz ne használunk bonyolult eszközöket**.

## Reguláris nyelvtan

Reguláris nyelvtanokban (Chomsky 3) a szabályok a következő alakúak lehetnek:

### Jobbrekurzív eset

$$\begin{aligned} A &\rightarrow a \\ A &\rightarrow aB \\ A &\rightarrow \epsilon \end{aligned}$$

### Balrekurzív eset

$$\begin{aligned} A &\rightarrow a \\ A &\rightarrow Ba \\ A &\rightarrow \epsilon \end{aligned}$$

3 Fordítóprogramok előadás (A,C,T szakirány) A lexikális elemzés

4 Fordítóprogramok előadás (A,C,T szakirány) A lexikális elemzés

## Reguláris nyelvtan

Reguláris nyelvtanokban (Chomsky 3) a szabályok a következő alakúak lehetnek:

### Jobbrekurzív eset

$$\begin{aligned} A &\rightarrow a \\ A &\rightarrow aB \\ A &\rightarrow \epsilon \end{aligned}$$

### Balrekurzív eset

$$\begin{aligned} A &\rightarrow a \\ A &\rightarrow Ba \\ A &\rightarrow \epsilon \end{aligned}$$

#### Példa: változónevek leírása

$$\begin{aligned} V &\rightarrow \underline{a} F \mid \underline{b} F \mid \dots \\ F &\rightarrow \epsilon \mid \underline{a} F \mid \underline{b} F \mid \dots \mid \underline{1} F \mid \underline{2} F \mid \dots \end{aligned}$$

## Reguláris kifejezés

- kifejezőreje azonos a reguláris nyelvtanokéval
- alapelemek:
  - üres halmaz
  - üres szöveget tartalmazó halmaz
  - egy karaktert tartalmazó halmaz
- konstruktív műveletek:
  - konkatenáció
  - unió:  $\mid$
  - lezárás:  $*$
- további „kényelmi” műveletek:  $+, ?$

### Példák (flex szintaxissal)

változónév: `[a-zA-Z] [a-zA-Z0-9]*`

egész szám: `(\+|\-)?`  $[0-9]^+$

törtszám: `[0-9]^+.\^+[0-9]^*`

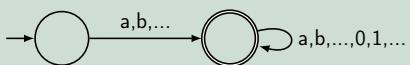
4 Fordítóprogramok előadás (A,C,T szakirány) A lexikális elemzés

5 Fordítóprogramok előadás (A,C,T szakirány) A lexikális elemzés

## Véges determinisztikus automaták

- kifejezőreje azonos a reguláris nyelvtanokéval és a reguláris kifejezésekével
- elemei:
  - ábécé
  - állapotok halmaza
  - átmennetfüggvény
  - kezdőállapot
  - végállapotok halmaza

### Változónevek elfogadása automatával



## Véges determinisztikus automata implementációja

- egymásba ágyazott if vagy case utasításokkal egy cikluson belül
  - elágazunk a pillanatnyi állapot szerint
  - ezen belül elágazunk a következő karakter szerint
  - az egyes ágakban beállítjuk a következő állapotot és kezdjük elóról
- 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 működése

- Az abc bemenet esetén a, ab és abc is legális változónév. Melyiket kellene felismerni?
  - A lexikális elemző minden a lehető leghosszabb karaktersorozatot ismeri fel.

## A lexikális elemző működése

- Az abc bemenet esetén a, ab és abc is legális változónév. Melyiket kellene felismerni?
  - A lexikális elemző minden a lehető leghosszabb karaktersorozatot ismeri fel.
- A while input megfelel a változónév definíciójának és egy kulcsszó is egyben. Melyiket kellene felismerni?
  - A lexikális elemző megadásakor sorbarendezhetjük a szimbólumok definíciót. Ha egyszerre több szimbólum is felismerhető, a sorrendben korábbi lesz az eredmény. (Tehát a kulcsszavak definícióját kell előre venni.)

## Kulcsszavak és standard szavak

### Kulcsszó

A kulcsszavaknak előre adott jelentésük van, és ez nem definiálható felül.

## Kulcsszavak és standard szavak

### Kulcsszó

A kulcsszavaknak előre adott jelentésük van, és ez nem definiálható felül.

### Standard szó

A standard szavaknak előre adott jelentésük van, de ez felüldefiniálható.

## Kulcsszavak és standard szavak

### Kulcsszó

A kulcsszavaknak előre adott jelentésük van, és ez nem definiálható felül.

### Standard szó

A standard szavaknak előre adott jelentésük van, de ez felüldefiniálható.

- Ha a kulcsszavakat is véges determinisztikus automatával akarjuk felismerni, nagyon nagy méretű automatát kaphatunk.
- Jobb módszer egy táblázatban tárolni őket: akárhányszor a lexikális elemző egy azonosítót ismer fel, meg kell nézni, hogy benne van-e ebben a táblázatban. (Ha igen, akkor kulcsszó, különben azonosító.)

9

Fordítóprogramok előadás (A,C,T szakirány)

A lexikális elemzés

## Előreolvasás

A lexikális elemző időnként több karaktert is előreolvas a szimbólum felismeréséhez.

10

Fordítóprogramok előadás (A,C,T szakirány)

A lexikális elemzés

## Előreolvasás

A lexikális elemző időnként több karaktert is előreolvas a szimbólum felismeréséhez.

④ példa: az egyes szimbólomok egymás prefixei

- egész szám: [0-9]+
- valós szám: [0-9]+."[0-9]+
- 3.14 ⇒ valós szám
- 3.x ⇒ egész szám, majd lexikális hiba
- az elemző megjegyzí a legutóbbi érvényes állapotot

## Előreolvasás

A lexikális elemző időnként több karaktert is előreolvas a szimbólum felismeréséhez.

④ példa: az egyes szimbólomok egymás prefixei

- egész szám: [0-9]+
- valós szám: [0-9]+."[0-9]+
- 3.14 ⇒ valós szám
- 3.x ⇒ egész szám, majd lexikális hiba
- az elemző megjegyzí a legutóbbi érvényes állapotot

④ példa: Fortran (a szóközöknek semmi szerepe nem volt!)

- D0 10 I = 1.1000 (ez egy értékkedés a D010I változónak)
- D0 10 I = 1,1000 (ez egy ciklus)

10

Fordítóprogramok előadás (A,C,T szakirány)

A lexikális elemzés

10

Fordítóprogramok előadás (A,C,T szakirány)

A lexikális elemzés

## Előreolvasás

A lexikális elemző időnként több karaktert is előreolvas a szimbólum felismeréséhez.

④ példa: az egyes szimbólomok egymás prefixei

- egész szám: [0-9]+
- valós szám: [0-9]+."[0-9]+
- 3.14 ⇒ valós szám
- 3.x ⇒ egész szám, majd lexikális hiba
- az elemző megjegyzí a legutóbbi érvényes állapotot

④ példa: Fortran (a szóközöknek semmi szerepe nem volt!)

- D0 10 I = 1.1000 (ez egy értékkedés a D010I változónak)
- D0 10 I = 1,1000 (ez egy ciklus)
- megoldás: D0 / [0-9]+ [a-zA-Z0-9]\* = [a-zA-Z0-9]\* ,
  - a '/' az előreolvasási operátor
  - r/s jelentése: ismert fel r-t, de csak ha s követi
  - r felismerése után s visszakerüli a bemenetbe
  - a leghosszabb karakterszorozatról való döntéskor | r | + | s | számít

## Szemantikus értékek és szimbólumtábla

- A lexikális elemzőnek a felismert szimbólum fajtáján kívül egyéb információkat is továbbítania kell.

- változó: a változó neve
- konstans: a konstans értéke

- Ezekre a **szemantikus értékekre** szemantikus elemzéshez és a kódgeneráláshoz van szükség.

- A változókat és azok adatait a **szimbólumtáblába** kell felírni.
  - Ezt általában a szintaktikus elemző teszi meg a változó deklarációjának felismerésekor.

10

Fordítóprogramok előadás (A,C,T szakirány)

A lexikális elemzés

11

Fordítóprogramok előadás (A,C,T szakirány)

A lexikális elemzés

## Direktívák

### Példa direktívára

```
#include "my.h"
#define valami 42
#ifndef FELTETEL
int akarmi() { return valami; }
#endif
```

Célszerű egy előfeldolgozó fázis beiktatása a lexikális és a szintaktikus elemzés közé, ami

- feljegyzi a makródefiníciókat,
- elvégzi a makróhelyettesítéseket,
- meghívja a lexikális elemzést a beillesztett fájlokra,
- kiértékeli a feltételeket és dönt a kód részletek beillesztéséről vagy törléséről.

## Hibatípusok és javítási lehetőségek

- illegális karakter (pl. add?ress, ? legyen az illegális karakter)
  - az éppen épített szimbólum eldobása és folytatás a következő karaktertől (eredmény: ress azonosító)
  - a karakter kihagyása (eredmény: address azonosító)
  - a karakter helyettesítése szóközzel (eredmény: add és ress azonosítók)

## Hibatípusok és javítási lehetőségek

- illegális karakter (pl. add?ress, ? legyen az illegális karakter)
  - az éppen épített szimbólum eldobása és folytatás a következő karaktertől (eredmény: ress azonosító)
  - a karakter kihagyása (eredmény: address azonosító)
  - a karakter helyettesítése szóközzel (eredmény: add és ress azonosítók)
- elgépelt kulcsszó (pl. while helyett wile whille wjile)
  - a lexikális elemző azonosítónak fogja felismerni, de egy ügyes szintaktikus elemző kijavíthatja a hibát
  - azokban a nyelveken, ahol a kulcsszavakat speciális módon jelölnek, a lexikális elemző is felismerheti és javíthatja a hibát
    - speciális jelölés lehet pl. adott karakterrel való kezdés, zárójelezés, nagybetű használata
    - a szintaxiskiemelés (pl. vastagítás, színezés) nem használható erre a célra, mert az láthatatlan a lexikális elemző számára!

## Hibatípusok és javítási lehetőségek

- kihagyott szimbólum (pl. 1+a helyett 1a, a+1 helyett a1)
  - ezeket csak a szintaktikus elemző tudja észrevenni

## Hibatípusok és javítási lehetőségek

- kihagyott szimbólum (pl. 1+a helyett 1a, a+1 helyett a1)
  - ezeket csak a szintaktikus elemző tudja észrevenni
- hibás számformátum (pl. 1.23.45)
  - valamelyiket illegális karakternek lehet tekinteni

## Hibatípusok és javítási lehetőségek

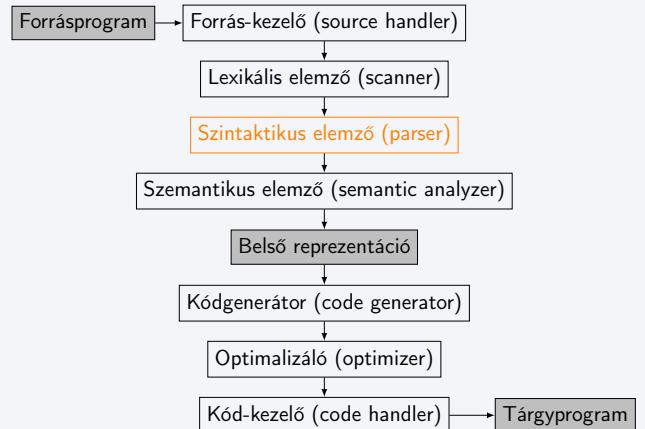
- kihagyott szimbólum (pl. 1+a helyett 1a, a+1 helyett a1)
  - ezeket csak a szintaktikus elemző tudja észrevenni
- hibás számformátum (pl. 1.23.45)
  - valamelyiket illegális karakternek lehet tekinteni
- befejezetlen megjegyzések és sztringek (pl. 'alma ..., /\* megjegyzés ...)
  - könnyen az egész további program megjegyzésbe kerülhet
  - sor végén, illetve fájl végén lehet jelezni a hibát

## A szintaktikus elemzés

Fordítóprogramok előadás (A,C,T szakirány)

A szintaktikus elemzésről általában

## A szintaktikus elemzés helye



1 Fordítóprogramok előadás (A,C,T szakirány)

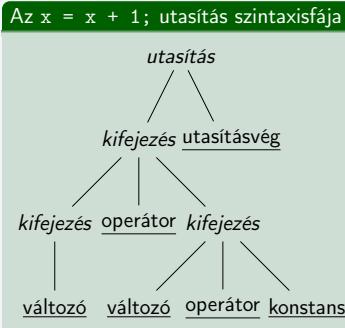
A szintaktikus elemzés

2 Fordítóprogramok előadás (A,C,T szakirány)

A szintaktikus elemzés

## Szintaktikus elemzés

- bemenet:  
szimbólumsorozat
- kimenet:  
szintaxisfa,  
szintaktikus hibák
- feladata:
  - a program szerkezetének felismerése
  - a szerkezet ellenőrzése:  
megfelel-e a nyelv definíciójának?



3 Fordítóprogramok előadás (A,C,T szakirány)

A szintaktikus elemzés

4 Fordítóprogramok előadás (A,C,T szakirány)

A szintaktikus elemzés

Jelölések és fogalmak

## Környezetfüggetlen nyelvtanok

- a szintaktikus elemzés:  
környezetfüggetlen nyelvtannal (Chomsky 2)
- $A \rightarrow \alpha$  alakú szabályok

„az  $A$  szimbólum a környezetétől függetlenül helyettesíthető  $\alpha$ -ra”

## Környezetfüggetlen nyelvtanok

- a szintaktikus elemzés:  
környezetfüggetlen nyelvtannal (Chomsky 2)
- $A \rightarrow \alpha$  alakú szabályok

„az  $A$  szimbólum a környezetétől függetlenül helyettesíthető  $\alpha$ -ra”

Jelölés	Jelentés
$a, b, c\dots$	terminális szimbólum (ezek a lexikális elemek, tokenek)
$A, B, C\dots$	nemterminális szimbólum
$X, Y, Z\dots$	terminális vagy nemterminális
$x, y, z\dots$	terminális szimbólumok sorozata
$\alpha, \beta, \gamma\dots$	terminális vagy nemterminális szimbólumok sorozata

## Levezetések és kapcsolódó fogalmak

Jelölés	Jelentés
$A \rightarrow \alpha$	szabály
$A \Rightarrow \alpha$	egy lépéses levezetés (1 szabály alk.)
$A \Rightarrow^* \alpha$	nulla, egy vagy több lépéses levezetés
$A \Rightarrow^+ \alpha$	legalább egy lépésből álló levezetés

4 Fordítóprogramok előadás (A,C,T szakirány)

A szintaktikus elemzés

5 Fordítóprogramok előadás (A,C,T szakirány)

A szintaktikus elemzés

## Levezetések és kapcsolódó fogalmak

Jelölés	Jelentés
$A \rightarrow \alpha$	szabály
$A \Rightarrow \alpha$	egy lépéses levezetés (1 szabály alk.)
$A \Rightarrow^* \alpha$	nulla, egy vagy több lépéses levezetés
$A \Rightarrow^+ \alpha$	legalább egy lépésből álló levezetés

- **mondat:** a startszimbólumból levezethető terminális sorozat ( $S \Rightarrow^* x$ )
- **mondatforma:** a startszimbólumból jelből levezethető bármilyen sorozat ( $S \Rightarrow^* \alpha$ )
- **részmondat:**  $\beta$  az  $\alpha_1\beta\alpha_2$  mondatforma részmondata, ha  $S \Rightarrow^* \alpha_1A\alpha_2 \Rightarrow^+ \alpha_1\beta\alpha_2$ .
- **egyszerű részmondat:**  $\beta$  az  $\alpha_1\beta\alpha_2$  mondatforma egyszerű részmondata, ha  $S \Rightarrow^* \alpha_1A\alpha_2 \Rightarrow \alpha_1\beta\alpha_2$ .

## Követelmények

- **ciklusmentesség:** nincs  $A \Rightarrow^+ A$  levezetés

- ellenpélda:
 
$$\begin{aligned} A &\rightarrow B \\ B &\rightarrow A \end{aligned}$$

## Követelmények

- **ciklusmentesség:** nincs  $A \Rightarrow^+ A$  levezetés
  - ellenpélda:
 
$$\begin{aligned} A &\rightarrow B \\ B &\rightarrow A \end{aligned}$$
- **redukáltság:** „nincsenek felesleges nemterminálisok”
  - minden nemterminális szimbólum előfordul valamelyik mondatformában
  - mindegyikból levezethető valamely terminális sorozat
  - ellenpélda:  $A \rightarrow aA$ , ha ez az egyetlen szabály  $A$ -ra

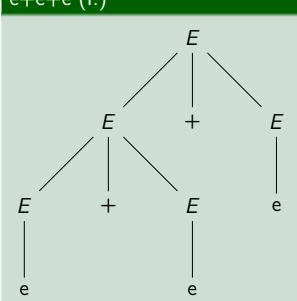
## Követelmények

- **ciklusmentesség:** nincs  $A \Rightarrow^+ A$  levezetés
  - ellenpélda:
 
$$\begin{aligned} A &\rightarrow B \\ B &\rightarrow A \end{aligned}$$
- **redukáltság:** „nincsenek felesleges nemterminálisok”
  - minden nemterminális szimbólum előfordul valamelyik mondatformában
  - mindegyikból levezethető valamely terminális sorozat
  - ellenpélda:  $A \rightarrow aA$ , ha ez az egyetlen szabály  $A$ -ra
- **egyértelműség:** minden mondathoz pontosan egy szintaxisfa tartozik

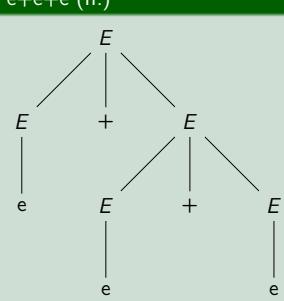
## Példa nem egyértelmű nyelvtanra

$$E \rightarrow e \mid E + E$$

e+e+e (I.)



e+e+e (II.)



## Legbal és legjobb levezetések

- **Legbal:** minden a *legbaloldalibb* nemterminálist helyettesítjük
- **Legjobb:** minden a *legjobboldalibb* nemterminálist helyettesítjük

## Legbal és legjobb levezetések

- **Legbal:** minden a *legbaloldalibb* nemterminálist helyettesítjük

### Legbal levezetés

$$S \Rightarrow AB \Rightarrow aaB \Rightarrow aab$$

- **Legjobb:** minden a *legjobboldalibb* nemterminálist helyettesítjük

### Legjobb levezetés

$$S \Rightarrow AB \Rightarrow Ab \Rightarrow aab$$

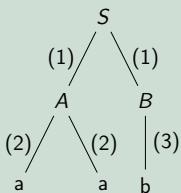
## Elemzési irányok

- **Felülről lefelé:** A startszimbólumból indulva, felülről lefelé építjük a szintaxisfát. A mondatforma baloldalán megjelenő terminálisokat illesztjük az elemzendő szövegre.

- **Alulról felfelé:** Az elemzendő szöveg összetartozó részeit helyettesítjük nemterminális szimbólumokkal (redukció) és így alulról, a startszimbólum felé építjük a fát.

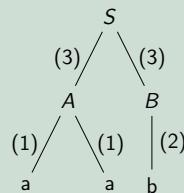
## Elemzési irányok

### Felülről lefelé



$$S \rightarrow^{(1)} AB \rightarrow^{(2)} aaB \rightarrow^{(3)} aab$$

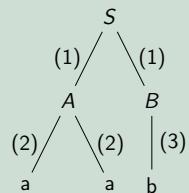
### Alulról felfelé



$$aab \leftarrow^{(1)} Ab \leftarrow^{(2)} AB \leftarrow^{(3)} S$$

## Elemzési irányok

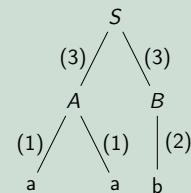
### Felülről lefelé



$$S \rightarrow^{(1)} AB \rightarrow^{(2)} aaB \rightarrow^{(3)} aab$$

• Ez egy *legbal* levezetés!

### Alulról felfelé



$$aab \leftarrow^{(1)} Ab \leftarrow^{(2)} AB \leftarrow^{(3)} S$$

• Ez egy *legjobb* levezetés inverze!

## Felülről lefelé elemzések

- Probléma: „Melyik helyettesítési szabályt kell alkalmazni?”

## Felülről lefelé elemzések

- Probléma: „Melyik helyettesítési szabályt kell alkalmazni?”

### Stratégiák:

- visszalépéses keresés (backtrack): ha nem illeszkednek a szövegre a mondatforma baloldalán megjelenő terminálisok, lépjünk vissza, és válasszunk másik szabályt  
⇒ **visszalépéses felülről lefelé elemzések** (nem tananyag)

- lassú
- ha hibás a szöveg, az csak túl későn derül ki

## Felülről lefelé elemzések

- Probléma: „Melyik helyettesítési szabályt kell alkalmazni?”
- Stratégiák:
  - visszalépéses keresés (backtrack): ha nem illeszkednek a szövegre a mondatforma baloldalán megjelenő terminálisok, lépjünk vissza, és válasszunk másik szabályt  
⇒visszalépéses felülről lefelé elemzések (nem tananyag)
    - lassú
    - ha hibás a szöveg, az csak túl későn derül ki
  - előreolvasás: olvassunk előre a szövegen valahány szimbólumot, és az alapján döntsünk az alkalmazandó szabályról  
⇒LL elemzések
    - csak szűk nyelvosztályra alkalmazható

## Alulról felfelé elemzések: „Mit redukálunk?”

## Alulról felfelé elemzések: „Mit redukálunk?”

- visszalépéses keresés (backtrack): ha nem sikerül eljutni a startszimbólumig, lépjünk vissza, és válasszunk másik redukciót  
⇒visszalépéses alulról felfelé elemzések (nem tananyag)
  - lassú
  - ha hibás a szöveg, az csak túl későn derül ki

## Alulról felfelé elemzések: „Mit redukálunk?”

- visszalépéses keresés (backtrack): ha nem sikerül eljutni a startszimbólumig, lépjünk vissza, és válasszunk másik redukciót  
⇒visszalépéses alulról felfelé elemzések (nem tananyag)
  - lassú
  - ha hibás a szöveg, az csak túl későn derül ki
- precedenciák használata: az egyes szimbólumok között adjunk meg precedenciarelációkat és ennek segítségével határozzuk meg a megfelelő redukciót  
⇒precedencia elemzések (nem tananyag)
  - ma már kevessé használt
  - operátorokkal felépített kifejezések esetén természetes a használata

## Alulról felfelé elemzések: „Mit redukálunk?”

- visszalépéses keresés (backtrack): ha nem sikerül eljutni a startszimbólumig, lépjünk vissza, és válasszunk másik redukciót  
⇒visszalépéses alulról felfelé elemzések (nem tananyag)
  - lassú
  - ha hibás a szöveg, az csak túl későn derül ki
- precedenciák használata: az egyes szimbólumok között adjunk meg precedenciarelációkat és ennek segítségével határozzuk meg a megfelelő redukciót  
⇒precedencia elemzések (nem tananyag)
  - ma már kevessé használt
  - operátorokkal felépített kifejezések esetén természetes a használata
- előreolvasás: olvassunk előre a szövegen valahány szimbólumot, és az alapján döntünk a redukcióról  
⇒LR elemzések
  - minden programozási nyelvhez lehet (LR) elemzőt készíteni
  - majdnem mindenhez lehet gyors (LALR) elemzőt készíteni

## LL elemzések

Fordítóprogramok előadás (A,C,T szakirány)

Az LL elemzésekéről általában

## LL elemzések

- felülről lefelé elemzés
- alapötlet:  $k$  terminális szimbólum előreolvasásával döntünk az alkalmazandó szabályról
- név: Left to right, using a Leftmost derivation (balról jobbra, legbal levezetéssel)

## Ellenpélda

Nem minden grammatika esetén alkalmazható!

Nem elemezhető LL módszerrel

$$\begin{aligned} S &\rightarrow A \mid B \\ A &\rightarrow a \mid aA \\ B &\rightarrow ab \mid aBb \end{aligned}$$

Az LL elemzésekéről általában

## Ellenpélda

Nem minden grammatika esetén alkalmazható!

Nem elemezhető LL módszerrel

$$\begin{aligned} S &\rightarrow A \mid B \\ A &\rightarrow a \mid aA \\ B &\rightarrow ab \mid aBb \end{aligned}$$

Akárhogy rögzítjük az előreolvasandó szimbólumok számát ( $k$ ), a

$$\underbrace{aa\dots a}_{k}\dots$$

inputra nem tudjuk eldönten, hogy  $S \rightarrow A$  vagy  $S \rightarrow B$  a jó választás.

## FIRST halmazok

Az LL elemzésekéről általában

$FIRST_k(\alpha)$ : az  $\alpha$  mondatformából levezethető terminális sorozatok  $k$  hosszúságú kezdőszeletei

(ha a sorozat hossza kisebb mint  $k$ , akkor az egész sorozat eleme  $FIRST_k(\alpha)$ -nak, akár  $\epsilon \in FIRST_k(\alpha)$  is előfordulhat)

Az LL elemzésekéről általában

## FIRST halmazok

$FIRST_k(\alpha)$ : az  $\alpha$  mondatformából levezethető terminális sorozatok  $k$  hosszúságú kezdőszeletei

(ha a sorozat hossza kisebb mint  $k$ , akkor az egész sorozat eleme  $FIRST_k(\alpha)$ -nak, akár  $\epsilon \in FIRST_k(\alpha)$  is előfordulhat)

Definíció:  $FIRST_k(\alpha)$

$$\begin{aligned} FIRST_k(\alpha) = \{x \mid \alpha \Rightarrow^* x\beta \wedge |x| = k\} \cup \\ \{x \mid \alpha \Rightarrow^* x \wedge |x| < k\} \end{aligned}$$

## LL( $k$ ) grammatikák

LL( $k$ ) grammatika: a levezetés tetszőleges pontján a szöveg következő  $k$  terminálisa meghatározza az alkalmazandó levezetési szabályt

## LL( $k$ ) grammatikák

LL( $k$ ) grammatika: a levezetés tetszőleges pontján a szöveg következő  $k$  terminálisa meghatározza az alkalmazandó levezetési szabályt

Definíció: LL( $k$ ) grammatica

Tetszőleges

$$S \Rightarrow^* wA\beta \Rightarrow w\alpha_1\beta \Rightarrow^* wx$$

$$S \Rightarrow^* wA\beta \Rightarrow w\alpha_2\beta \Rightarrow^* wy$$

levezetéspárra  $FIRST_k(x) = FIRST_k(y)$  esetén  $\alpha_1 = \alpha_2$ .

## LL elemzések

- példa szoftverek:
  - ANTLR parser generátor: <http://www.antlr.org/>
  - Coco/R: <http://www.ssw.uni-linz.ac.at/coco/>
- gyakorlatban: az LL(1) elemzést könnyű megvalósítani
  - egyszerű LL(1)
  - epszilonmentes LL(1)
  - általános

Definíció: Egyszerű LL(1)

Olyan LL(1) grammatika, amelyben a szabályok jobboldala terminális szimbólummal kezdődik (ezért  $\epsilon$ -mentes is). (Az összes szabály  $A \rightarrow a\alpha$  alakú.)

Következmény: Egyszerű LL(1) grammatika esetén az azonos nemterminálhoz tartozó szabályok jobboldalai különböző terminálissal kezdődnek.

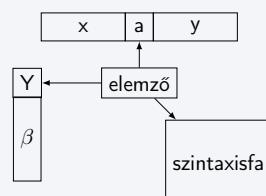
### Tétel

Egy grammatika pontosan akkor egyszerű LL(1), ha

- csak  $A \rightarrow a\alpha$  alakú szabályai vannak
- és ha  $A \rightarrow a_1\alpha_1$  és  $A \rightarrow a_2\alpha_2$  két különböző szabály, akkor  $a_1 \neq a_2$ .

(A fordítóprogramok könyvben ez szerepel definícióként!)

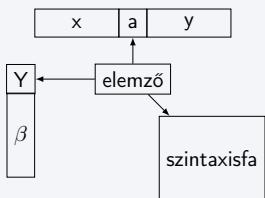
## Egyszerű LL(1) elemzés



- $xay$ : bemenet

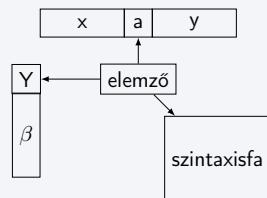
- $Y\beta$ : aktuális mondatforma (veremben tároljuk)

## Egyszerű LL(1) elemzés



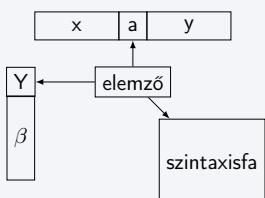
- ha a verem tetején terminális szimbólum van:
  - ha egyezik a szöveg következő karakterével: pop és lépés a szövegen
  - különben: hiba

## Egyszerű LL(1) elemzés



- ha a verem tetején nemterminális szimbólum ( $A$ ) van:
  - ha van  $A \rightarrow a\alpha$  szabály: A helyére  $a\alpha$  és bejegyzés a szintaxisfába
  - különben: hiba

## Egyszerű LL(1) elemzés



- ha a verem üres:
  - ha a szöveg végére értünk: OK
  - különben hiba

## Elemző táblázat

Az egyes akciók táblázatba foglalhatók.

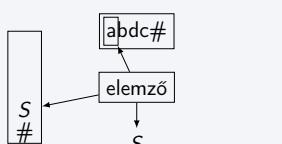
**Példa nyelvtan**

$S \rightarrow aS$ (1)	$ $	$bAc$ (2)
$A \rightarrow bAc$ (3)	$ $	$d$ (4)

- #: veremalja és fájlvége
- üres helyek: hiba

	a	b	c	d	#
S	$S \rightarrow^{(1)} aS$	$S \rightarrow^{(2)} bAc$			
A		$A \rightarrow^{(3)} bAc$		$A \rightarrow^{(4)} d$	
a	pop				
b		pop			
c			pop		
d				pop	
#					OK

## Példa: abdc

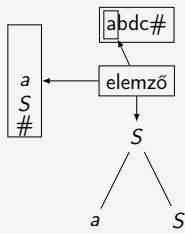


•  $(abdc\#, S\#, \epsilon)$

## Példa: abdc

	a	b	c	d	#
S	$S \rightarrow^{(1)} aS$	$S \rightarrow^{(2)} bAc$			
A		$A \rightarrow^{(3)} bAc$		$A \rightarrow^{(4)} d$	
a	pop				
b		pop			
c			pop		
d				pop	
#					OK

Példa: abdc

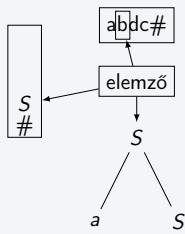


- $(abdc\#, S\#, \epsilon)$
- $(abdc\#, aS\#, 1)$

Példa: abdc

	a	b	c	d	#
S	$S \rightarrow^{(1)} aS$	$S \rightarrow^{(2)} bAc$			
A		$A \rightarrow^{(3)} bAc$		$A \rightarrow^{(4)} d$	
a	pop				
b		pop			
c			pop		
d				pop	
#					OK

Példa: abdc

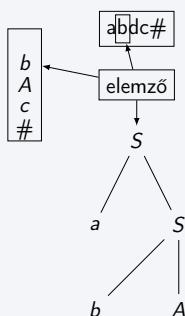


- $(abdc\#, S\#, \epsilon)$
- $(abdc\#, aS\#, 1)$
- $(bdc\#, S\#, 1)$

Példa: abdc

	a	b	c	d	#
S	$S \rightarrow^{(1)} aS$	$S \rightarrow^{(2)} bAc$			
A		$A \rightarrow^{(3)} bAc$		$A \rightarrow^{(4)} d$	
a	pop				
b		pop			
c			pop		
d				pop	
#					OK

Példa: abdc

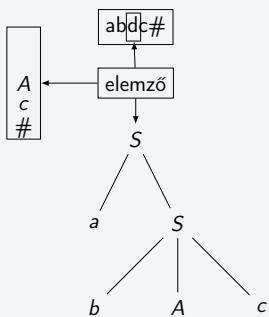


- $(abdc\#, S\#, \epsilon)$
- $(abdc\#, aS\#, 1)$
- $(bdc\#, S\#, 1)$
- $(bdc\#, bAc\#, 12)$

Példa: abdc

	a	b	c	d	#
S	$S \rightarrow^{(1)} aS$	$S \rightarrow^{(2)} bAc$			
A		$A \rightarrow^{(3)} bAc$		$A \rightarrow^{(4)} d$	
a	pop				
b		pop			
c			pop		
d				pop	
#					OK

Példa: abdc

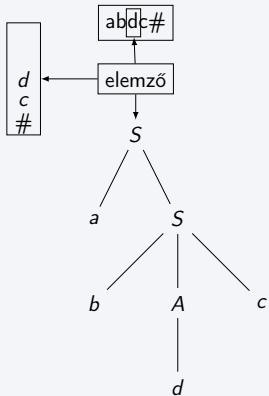


- $(abdc\#, S\#, \epsilon)$
- $(abdc\#, aS\#, 1)$
- $(bdc\#, S\#, 1)$
- $(bdc\#, bAc\#, 12)$
- $(dc\#, Ac\#, 12)$

Példa: abdc

	a	b	c	d	#
S	$S \rightarrow^{(1)} aS$	$S \rightarrow^{(2)} bAc$			
A		$A \rightarrow^{(3)} bAc$		$A \rightarrow^{(4)} d$	
a	pop				
b		pop			
c			pop		
d				pop	
#					OK

Példa: abdc

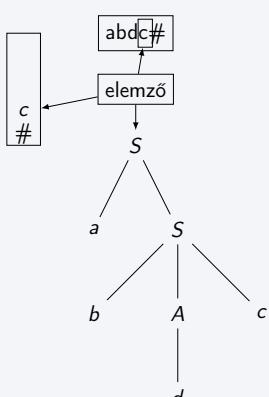


- $(abdc\#, S\#, \epsilon)$
- $(abdc\#, aS\#, 1)$
- $(bdc\#, S\#, 1)$
- $(bdc\#, bAc\#, 12)$
- $(dc\#, Ac\#, 12)$
- $(dc\#, dc\#, 124)$

Példa: abdc

	a	b	c	d	#
S	$S \rightarrow^{(1)} aS$	$S \rightarrow^{(2)} bAc$			
A		$A \rightarrow^{(3)} bAc$		$A \rightarrow^{(4)} d$	
a	pop				
b		pop			
c			pop		
d				pop	
#					OK

Példa: abdc

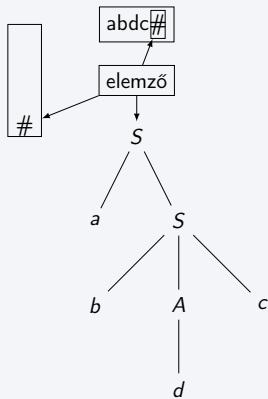


- $(abdc\#, S\#, \epsilon)$
- $(abdc\#, aS\#, 1)$
- $(bdc\#, S\#, 1)$
- $(bdc\#, bAc\#, 12)$
- $(dc\#, Ac\#, 12)$
- $(dc\#, dc\#, 124)$
- $(c\#, c\#, 124)$

Példa: abdc

	a	b	c	d	#
S	$S \rightarrow^{(1)} aS$	$S \rightarrow^{(2)} bAc$			
A		$A \rightarrow^{(3)} bAc$		$A \rightarrow^{(4)} d$	
a	pop				
b		pop			
c			pop		
d				pop	
#					OK

## Példa: abdc

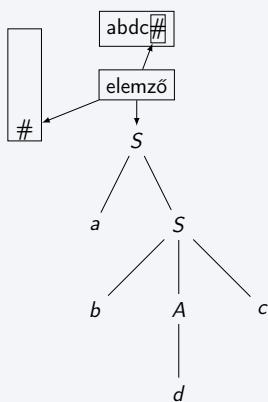


- ( $abdc\#, S\#, \epsilon$ )
- ( $abdc\#, aS\#, 1$ )
- ( $bdc\#, S\#, 1$ )
- ( $bdc\#, bAc\#, 12$ )
- ( $dc\#, Ac\#, 12$ )
- ( $dc\#, dc\#, 124$ )
- ( $c\#, c\#, 124$ )
- ( $\#, \#, 124$ )

## Példa: abdc

	a	b	c	d	#
S	$S \rightarrow^{(1)} aS$	$S \rightarrow^{(2)} bAc$			
A		$A \rightarrow^{(3)} bAc$		$A \rightarrow^{(4)} d$	
a	pop				
b		pop			
c			pop		
d				pop	
#					OK

## Példa: abdc



- ( $abdc\#, S\#, \epsilon$ )
- ( $abdc\#, aS\#, 1$ )
- ( $bdc\#, S\#, 1$ )
- ( $bdc\#, bAc\#, 12$ )
- ( $dc\#, Ac\#, 12$ )
- ( $dc\#, dc\#, 124$ )
- ( $c\#, c\#, 124$ )
- ( $\#, \#, 124$ )
- OK

## ε-mentes LL(1) nyelvtan

## Definíció: ε-mentes LL(1)

Olyan LL(1) grammatika, amely ε-mentes.  
(Nincs  $A \rightarrow \epsilon$  szabály.)

## ε-mentes LL(1) nyelvtan

## Definíció: ε-mentes LL(1)

Olyan LL(1) grammatika, amely ε-mentes.  
(Nincs  $A \rightarrow \epsilon$  szabály.)

Következmény: ε-mentes LL(1) grammatika esetén az egy nemterminálishoz tartozó szabályok jobboldalainak  $FIRST_1$  halmazai diszjunktak.

## Tétel

Egy grammatika pontosan akkor ε-mentes LL(1), ha

- ε-mentes
- és ha  $A \rightarrow \alpha_1$  és  $A \rightarrow \alpha_2$  két különböző szabály, akkor  $FIRST_1(\alpha_1) \cap FIRST_1(\alpha_2) = \emptyset$ .

(A fordítóprogramok könyvben ez szerepel definícióként.)

## ε-mentes LL(1) elemzés

- ha a verem tetején terminális szimbólum van:

- ha egyezik a szöveg következő karakterével:  
pop és lépés a szövegben
- különben: hiba

## ε-mentes LL(1) elemzés

- ha a verem tetején terminális szimbólum van:
  - ha egyezik a szöveg következő karakterével:  
pop és lépés a szövegen
  - különben: hiba
- ha a verem tetején nemterminális szimbólum ( $A$ ) van (és a szövegen a következik):
  - ha van  $A \rightarrow \alpha$  szabály, amelyre  $a \in FIRST_1(\alpha)$ :  
 $A$  helyére  $\alpha$  és bejegyzés a szintaxisfába
  - különben: hiba

## ε-mentes LL(1) elemzés

- ha a verem tetején terminális szimbólum van:
  - ha egyezik a szöveg következő karakterével:  
pop és lépés a szövegen
  - különben: hiba
- ha a verem tetején nemterminális szimbólum ( $A$ ) van (és a szövegen a következik):
  - ha van  $A \rightarrow \alpha$  szabály, amelyre  $a \in FIRST_1(\alpha)$ :  
 $A$  helyére  $\alpha$  és bejegyzés a szintaxisfába
  - különben: hiba
- ha a verem üres:
  - ha a szöveg végére értünk: OK
  - különben hiba

- ugyanolyan szerkezetű, mint az egyszerű LL(1)-es
- az  $A \rightarrow \alpha$  szabályt az  $A$  sorába és a  $FIRST_1(\alpha)$  elemeinek oszlopába kell beírni

## Példa nyelvtan és a FIRST halmazok

$$\begin{array}{l} S \rightarrow aS \mid A \\ A \rightarrow bAc \mid d \end{array}$$

$$\begin{array}{ll} FIRST_1(aS) = \{a\} & FIRST_1(A) = \{b, d\} \\ FIRST_1(bAc) = \{b\} & FIRST_1(d) = \{d\} \end{array}$$

	a	b	c	d	#
S	$S \xrightarrow{(1)} aS$	$S \xrightarrow{(2)} A$		$S \xrightarrow{(2)} A$	
A		$A \xrightarrow{(3)} bAc$		$A \xrightarrow{(4)} d$	
a	pop				
b		pop			
c			pop		
d				pop	
#					OK

## LL(1)

## Definíció: LL(1) grammatika

## Tetszőleges

$$S \Rightarrow^* wA\beta \Rightarrow w\alpha_1\beta \Rightarrow^* wx$$

$$S \Rightarrow^* wA\beta \Rightarrow w\alpha_2\beta \Rightarrow^* wy$$

levezetéspárra  $FIRST_1(x) = FIRST_1(y)$  esetén  $\alpha_1 = \alpha_2$ .

## LL(1)

## Definíció: LL(1) grammatika

## Tetszőleges

$$S \Rightarrow^* wA\beta \Rightarrow w\alpha_1\beta \Rightarrow^* wx$$

$$S \Rightarrow^* wA\beta \Rightarrow w\alpha_2\beta \Rightarrow^* wy$$

levezetéspárra  $FIRST_1(x) = FIRST_1(y)$  esetén  $\alpha_1 = \alpha_2$ .

## Probléma:

$A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$  szabályban előfordulhat, hogy  $\alpha_i \Rightarrow^* \epsilon$  valamelyik  $\alpha_i$ -re.

Ezért az előreolvasott szimbólum a  $S \Rightarrow^* wA\beta$  levezetésesetén a  $\beta$ -ból származhat.

Meg kell tehát vizsgálni azt is, hogy milyen terminálisok követhetik az  $A$ -t!

## FOLLOW halmazok

$FOLLOW_k(\alpha)$ : a levezetésekben az  $\alpha$  után előforduló  $k$  hosszúságú terminális sorozatok

(ha a sorozat hossza kisebb mint  $k$ , akkor az egész sorozat eleme  $FOLLOW_k(\alpha)$ -nak,  
ha  $\alpha$  után vége lehet a szövegnek, akkor  $\# \in FOLLOW_k(\alpha)$ )

## FOLLOW halmazok

**FOLLOW<sub>k</sub>( $\alpha$ )**: a levezetésekben az  $\alpha$  után előforduló  $k$  hosszúságú terminális sorozatok

(ha a sorozat hossza kisebb mint  $k$ , akkor az egész sorozat eleme FOLLOW<sub>k</sub>( $\alpha$ )-nak,  
ha  $\alpha$  után vége lehet a szövegnek, akkor  $\# \in FOLLOW_k(\alpha)$ )

Definíció: FOLLOW<sub>k</sub>( $\alpha$ )

$$FOLLOW_k(\alpha) = \{x \mid S \Rightarrow^* \beta\alpha\gamma \wedge x \in FIRST_k(\gamma) \setminus \{\epsilon\} \cup \{\#\mid S \Rightarrow^* \beta\alpha\}$$

## LL(1) elemzést megalapozó téTEL

## TéTEL

Egy grammata pontosan akkor LL(1) grammata, ha bármely  $A \rightarrow \alpha_1$  és  $A \rightarrow \alpha_2$  különböző szabályok esetén  $FIRST_1(\alpha_1 FOLLOW_1(A)) \cap FIRST_1(\alpha_2 FOLLOW_1(A)) = \emptyset$ .

**FIRST<sub>1</sub>( $\alpha FOLLOW_1(A)$ ) jelentése**:  $\alpha$ -hoz egyenként konkatenáljuk FOLLOW<sub>1</sub>( $A$ ) elemeit és az így kapott halmaz minden elemére alkalmazzuk a FIRST<sub>1</sub> függvényt.

## Példa

## Példa nyelvtan és a FIRST, FOLLOW halmazok

$$\begin{aligned} S &\rightarrow aS \mid A \\ A &\rightarrow bAc \mid d \mid \epsilon \end{aligned}$$

$$\begin{aligned} FOLLOW_1(S) &= \{\#\} \\ FOLLOW_1(A) &= \{c, \#\} \end{aligned}$$

$$\begin{aligned} FIRST_1(aS FOLLOW_1(S)) &= \{a\} \\ FIRST_1(A FOLLOW_1(S)) &= \{b, d, \#\} \end{aligned}$$

$$\begin{aligned} FIRST_1(bAc FOLLOW_1(A)) &= \{b\} \\ FIRST_1(d FOLLOW_1(A)) &= \{d\} \\ FIRST_1(\epsilon FOLLOW_1(A)) &= \{c, \#\} \end{aligned}$$

## LL(1) elemző táblázat

Mint az egyszerű és az  $\epsilon$ -mentes LL(1) esetén. Az  $A \rightarrow \alpha$  szabályt az  $A$  sorába és a  $FIRST_1(\alpha FOLLOW_1(A))$  halmaz oszlopáiba kell írni.

	a	b	c	d	#
S	$S \xrightarrow{(1)} aS$	$S \xrightarrow{(2)} A$		$S \xrightarrow{(2)} A$	$S \xrightarrow{(2)} A$
A		$A \xrightarrow{(3)} bAc$	$A \xrightarrow{(5)} \epsilon$	$A \xrightarrow{(4)} d$	$A \xrightarrow{(5)} \epsilon$
a	pop				
b		pop			
c			pop		
d				pop	
#					OK

## LL(1) elemzés

- ha a verem tetején terminális szimbólum van:

- ha egyezik a szöveg következő karakterével: pop és lépés a szövegben
- különben: hiba

## LL(1) elemzés

- ha a verem tetején terminális szimbólum van:

- ha egyezik a szöveg következő karakterével: pop és lépés a szövegben
- különben: hiba

- ha a verem tetején nemterminális szimbólum ( $A$ ) van (és a szövegben a következik):

- ha van  $A \rightarrow \alpha$  szabály, amelyre  $a \in FIRST_1(\alpha FOLLOW_1(A))$ :  $A$  helyére  $\alpha$  és bejegyzés a szintaxisfába
- különben: hiba

## LL(1) elemzés

- ha a verem tetején terminális szimbólum van:
  - ha egyezik a szöveg következő karakterével: pop és lépés a szövegben
  - különben: hiba
- ha a verem tetején nemterminális szimbólum ( $A$ ) van (és a szövegben a következik):
  - ha van  $A \rightarrow \alpha$  szabály, amelyre  $a \in FIRST_1(\alpha FOLLOW_1(A))$ :  
A helyére  $\alpha$  és bejegyzés a szintaxisfába
  - különben: hiba
- ha a verem üres:
  - ha a szöveg végére értünk: OK
  - különben hiba

## Rekurzív leszállás

- az (általános) LL(1) elemzés egy másik implementációja
- minden nemterminálishoz egy eljárást készítünk
- az eljárásokon keresztül a futási idejű verem valósítja meg az elemzés vermét

## Az elfogad eljárás

A terminális szimbólumok ellenőrzéséhez:

```
void elfogad( terminalis t )
{
    if( aktualis_szimbolum == t )
        aktualis_szimbolum = lexikalis_elemzo.kovetkezo();
    else
        hiba(...);
}
```

(A lexikális elemző kovetkezo függvénye visszaadja a soron következő lexikális elemet.)

 $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$  programja

```
void A()
{
    if( aktualis_szimbolum ∈ FIRST_1(α1 FOLLOW1(A)))
    {
        // alfa_1 programja
    }
    ...
    else if( aktualis_szimbolum ∈ FIRST_1(αn FOLLOW1(A)))
    {
        // alfa_n programja
    }
    else
    {
        hiba(...);
    }
}
```

## A szabályalternatívák programja

Az  $\alpha = \epsilon$  alternatíva programja a „skip”.

Az  $\alpha = X_1 X_2 \dots X_n$  alternatíva programja az  $X_1, X_2, \dots, X_n$  szimbólumokhoz tartozó utasítások szekvenciája.

- ha  $X_i = a$  (terminális), akkor az  $X_i$ -hez tartozó utasítás  
`elfogad(a);`
- ha  $X_i = B$  (nemterminális), akkor  
`B();`

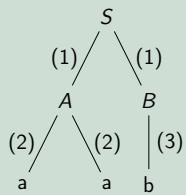
- hiba detektálása esetén:
  - segítő hibajelzést kell adni
  - meg kell próbálni folytatni az elemzést (az elemzőt szinkronizálni kell a bemenettel)
- a szinkronizáció szimbólumok kihagyását jelenti egy olyan szimbólumig, ahonnan folytatni lehet az elemzést
  - „pánik módszer”: az elemző pánikszerűen menekül a hiba helyétől
- például: a rekurzív leszállás eljárásai elején megvizsgálhatjuk, hogy megfelelő-e az aktualis\_szimbolum, és szükség esetén addig ugorjuk át a szimbólumokat a bemeneten, amíg megfelelő nem lesz

## LR elemzések (az LR(0) elemzés)

Fordítóprogramok előadás (A,C,T szakirány)

## Emlékeztető: elemzési irányok

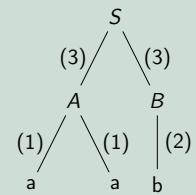
### Felülről lefelé



$$S \xrightarrow{(1)} AB \xrightarrow{(2)} aaB \xrightarrow{(3)} aab$$

- Ez egy *legbal* levezetés!

### Alulról felfelé



$$aab \xleftarrow{(1)} Ab \xleftarrow{(2)} AB \xleftarrow{(3)} S$$

- Ez egy *legjobb* levezetés inverze!

## Emlékeztető: alulról felfelé elemzések

- Az elemzendő szöveg összetartozó részeit helyettesítjük nemterminális szimbólumokkal (redukció) és így alulról, a kezdőszimbólum felé építjük a fát.
- Fő kérdés: „Mit redukálunk?”

## Emlékeztető: alulról felfelé elemzési stratégiák

- **visszalépéses keresés (backtrack):** ha nem sikerül eljutni a startszimbólumig, lépjünk vissza, és válasszunk másik redukciót  
⇒visszalépéses alulról felfelé elemzések (nem tananyag)
  - lassú
  - ha hibás a szöveg, az csak túl későn derül ki

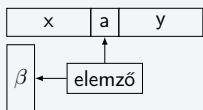
## Emlékeztető: alulról felfelé elemzési stratégiák

- **visszalépéses keresés (backtrack):** ha nem sikerül eljutni a startszimbólumig, lépjünk vissza, és válasszunk másik redukciót  
⇒visszalépéses alulról felfelé elemzések (nem tananyag)
  - lassú
  - ha hibás a szöveg, az csak túl későn derül ki
- **precedenciák használata:** az egyes szimbólumok között adjunk meg precedenciarelációkat és ennek segítségével határozzuk meg a megfelelő redukciót  
⇒precedencia elemzések (nem tananyag)
  - ma már kevessé használt
  - operátorokkal felépített kifejezések esetén természetes a használata

## Emlékeztető: alulról felfelé elemzési stratégiák

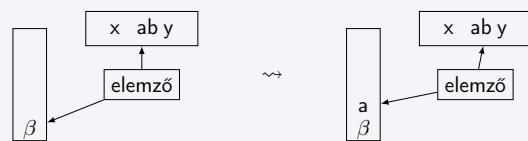
- **visszalépéses keresés (backtrack):** ha nem sikerül eljutni a startszimbólumig, lépjünk vissza, és válasszunk másik redukciót  
⇒visszalépéses alulról felfelé elemzések (nem tananyag)
  - lassú
  - ha hibás a szöveg, az csak túl későn derül ki
- **precedenciák használata:** az egyes szimbólumok között adjunk meg precedenciarelációkat és ennek segítségével határozzuk meg a megfelelő redukciót  
⇒precedencia elemzések (nem tananyag)
  - ma már kevessé használt
  - operátorokkal felépített kifejezések esetén természetes a használata
- **előreolvasás:** olvassunk előre a szövegen valahány szimbólumot, és az alapján döntsünk a redukcióról  
⇒LR elemzések
  - minden programozási nyelvhez lehet (LR) elemzöt készíteni
  - majdnem mindenhez lehet gyors (LALR) elemzöt készíteni

## Az LR elemző felépítése



- $xay$ : bemenet
- $\beta$ : aktuális mondatforma egy része (veremben tároljuk)
- két lehetséges művelet: léptetés és redukálás
- *LR*: Left to right, using a Rightmost derivation  
(Balról jobbra, legjobb vezetéssel)

## Léptetés



A bemenet következő szimbólumát a verem tetejére helyezzük.

## Redukálás

Az  $A \rightarrow \alpha$  szabály szerinti redukció.



A verem tetején lévő szabály-jobboldalt helyettesítjük a megfelelő nemterminális szimbólummal.

## Kiegészített grammatika

- Az elemzés végét arról fogjuk felismerni, hogy egy redukció eredménye a kezdőszimbólum lett.
- Ez csak akkor lehet, ha a kezdőszimbólum nem fordul elő a szabályok jobboldalán.

## Kiegészített grammatika

- Az elemzés végét arról fogjuk felismerni, hogy egy redukció eredménye a kezdőszimbólum lett.
- Ez csak akkor lehet, ha a kezdőszimbólum nem fordul elő a szabályok jobboldalán.
- Ezt nem minden grammatika teljesíti, de mindegyik kiegészíthető:
  - legyen  $S'$  az új kezdőszimbólum
  - legyen  $S' \rightarrow S$  egy új szabály
- az *LR* elemzésekhez minden kiegészített nyelvtanokat fogunk használni

## Mit kell redukálni?

## Mit kell redukálni?

- **Egyszerű részmondat** (emlékeztető):  $\alpha$  a  $\gamma\alpha\beta$  mondatforma egyszerű részmondata, ha  $S \Rightarrow^* \gamma A \beta \Rightarrow \gamma\alpha\beta$ .

## Mit kell redukálni?

- **Egyszerű részmondat** (emlékeztető):  $\alpha$  a  $\gamma\alpha\beta$  mondatforma egyszerű részmondata, ha  $S \Rightarrow^* \gamma A \beta \Rightarrow \gamma\alpha\beta$ .
- **Nyél:** a mondatformában a legbaloldalibb egyszerű részmondat.

## Mit kell redukálni?

- **Egyszerű részmondat** (emlékeztető):  $\alpha$  a  $\gamma\alpha\beta$  mondatforma egyszerű részmondata, ha  $S \Rightarrow^* \gamma A \beta \Rightarrow \gamma\alpha\beta$ .
- **Nyél:** a mondatformában a legbaloldalibb egyszerű részmondat.
- **Épp a nyelet kell megtalálni a redukcióhoz.**

## A nyél meghatározása

- **Probléma: „Mi a nyél?”**
  - léptetni vagy redukálni kell?
  - ha több lehetőség is van, melyik szabály szerint kell redukálni?

## A nyél meghatározása

- **Probléma: „Mi a nyél?”**
  - léptetni vagy redukálni kell?
  - ha több lehetőség is van, melyik szabály szerint kell redukálni?
- **$LR(k)$  grammatika:**  $k$  szimbólum előreolvasásával eldönthető, hogy mi legyen az elemzés következő lépése.

Magyarázat az  $LR(k)$  definíciójához

- Tegyük fel, hogy léptetésekkel és redukálásokkal eljutottunk az  $\alpha\beta w$  mondatformához, és itt  $\beta$  a nyél:  $S \Rightarrow^* \alpha A w \Rightarrow \alpha\beta w$ .

Magyarázat az  $LR(k)$  definíciójához

- Tegyük fel, hogy léptetésekkel és redukálásokkal eljutottunk az  $\alpha\beta w$  mondatformához, és itt  $\beta$  a nyél:  $S \Rightarrow^* \alpha Aw \Rightarrow \alpha\beta w$ .
- Tegyük fel, hogy egy ugyanígy kezdődő mondatforma, az  $\alpha\beta y$  felbontható  $\alpha\beta y = \gamma\delta x$  módon, és ebben  $\delta$  a nyél, azaz  $S \Rightarrow^* \gamma Bx \Rightarrow \gamma\delta x$ .

Magyarázat az  $LR(k)$  definíciójához

- Tegyük fel, hogy léptetésekkel és redukálásokkal eljutottunk az  $\alpha\beta w$  mondatformához, és itt  $\beta$  a nyél:  $S \Rightarrow^* \alpha Aw \Rightarrow \alpha\beta w$ .
- Tegyük fel, hogy egy ugyanígy kezdődő mondatforma, az  $\alpha\beta y$  felbontható  $\alpha\beta y = \gamma\delta x$  módon, és ebben  $\delta$  a nyél, azaz  $S \Rightarrow^* \gamma Bx \Rightarrow \gamma\delta x$ .
- Az  $LR(k)$  tulajdonság azt mondja, hogy  $w$ -ból és  $y$ -ból előreolvasva  $k$  szimbólumot, egyértelműen előntható az elemzés következő lépése.

Magyarázat az  $LR(k)$  definíciójához

- Tegyük fel, hogy léptetésekkel és redukálásokkal eljutottunk az  $\alpha\beta w$  mondatformához, és itt  $\beta$  a nyél:  $S \Rightarrow^* \alpha Aw \Rightarrow \alpha\beta w$ .
- Tegyük fel, hogy egy ugyanígy kezdődő mondatforma, az  $\alpha\beta y$  felbontható  $\alpha\beta y = \gamma\delta x$  módon, és ebben  $\delta$  a nyél, azaz  $S \Rightarrow^* \gamma Bx \Rightarrow \gamma\delta x$ .
- Az  $LR(k)$  tulajdonság azt mondja, hogy  $w$ -ból és  $y$ -ból előreolvasva  $k$  szimbólumot, egyértelműen előntható az elemzés következő lépése.
- Ezért ha  $FIRST_k(w) = FIRST_k(y)$ , akkor  $\alpha\beta w$  és  $\alpha\beta y$  esetén is ugyanazt kell csinálni:
  - mivel  $\alpha\beta w$  esetén az  $A \rightarrow \beta$  szabály szerint redukáltunk,
  - ugyanezt kellett csinálni  $\alpha\beta y$  esetén is,
  - vagyis  $\alpha = \gamma$ ,  $\beta = \delta$ ,  $A = B$  és  $y = x$ .

 $LR(k)$  definíciójaDefiníció:  $LR(k)$  grammatika

Egy kiegészített grammata  $LR(k)$  grammata ( $k \geq 0$ ), ha

$$S \Rightarrow^* \alpha Aw \Rightarrow \alpha\beta w$$

$$S \Rightarrow^* \gamma Bx \Rightarrow \gamma\delta x$$

$\alpha\beta y = \gamma\delta x$  és  $FIRST_k(w) = FIRST_k(y)$  esetén

$\alpha = \gamma$ ,  $\beta = \delta$  és  $A = B$ .

 $LR(k)$  definíciójaDefiníció:  $LR(k)$  grammatika

Egy kiegészített grammata  $LR(k)$  grammata ( $k \geq 0$ ), ha

$$S \Rightarrow^* \alpha Aw \Rightarrow \alpha\beta w$$

$$S \Rightarrow^* \gamma Bx \Rightarrow \gamma\delta x$$

$\alpha\beta y = \gamma\delta x$  és  $FIRST_k(w) = FIRST_k(y)$  esetén

$\alpha = \gamma$ ,  $\beta = \delta$  és  $A = B$ .

**LR(0) grammata:** előreolvasás nélkül előntható az elemzés következő lépése.

## Példa

## Grammatika

- 1  $S' \rightarrow S$
- 2  $S \rightarrow aAd$
- 3  $A \rightarrow bA$
- 4  $A \rightarrow c$

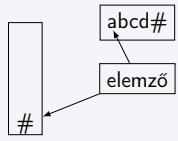
$$S' \Rightarrow S \Rightarrow aAd \Rightarrow abAd \Rightarrow abc$$

## Példa

## Grammatika

- ❶  $S' \rightarrow S$
- ❷  $S \rightarrow aAd$
- ❸  $A \rightarrow bA$
- ❹  $A \rightarrow c$

$$S' \Rightarrow S \Rightarrow aAd \Rightarrow abAd \Rightarrow abcd$$



## Példa

## Grammatika

- ❶  $S' \rightarrow S$
- ❷  $S \rightarrow aAd$
- ❸  $A \rightarrow bA$
- ❹  $A \rightarrow c$

$$S' \Rightarrow S \Rightarrow aAd \Rightarrow abAd \Rightarrow abcd$$

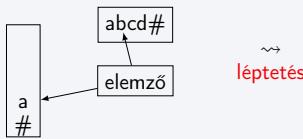


## Példa

## Grammatika

- ❶  $S' \rightarrow S$
- ❷  $S \rightarrow aAd$
- ❸  $A \rightarrow bA$
- ❹  $A \rightarrow c$

$$S' \Rightarrow S \Rightarrow aAd \Rightarrow abAd \Rightarrow abcd$$



## Példa

## Grammatika

- ❶  $S' \rightarrow S$
- ❷  $S \rightarrow aAd$
- ❸  $A \rightarrow bA$
- ❹  $A \rightarrow c$

$$S' \Rightarrow S \Rightarrow aAd \Rightarrow abAd \Rightarrow abcd$$

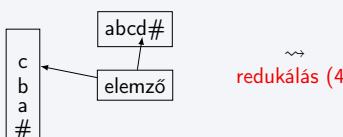


## Példa

## Grammatika

- ❶  $S' \rightarrow S$
- ❷  $S \rightarrow aAd$
- ❸  $A \rightarrow bA$
- ❹  $A \rightarrow c$

$$S' \Rightarrow S \Rightarrow aAd \Rightarrow abAd \Rightarrow abcd$$

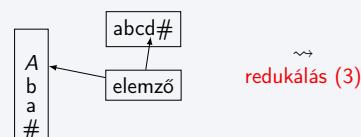


## Példa

## Grammatika

- ❶  $S' \rightarrow S$
- ❷  $S \rightarrow aAd$
- ❸  $A \rightarrow bA$
- ❹  $A \rightarrow c$

$$S' \Rightarrow S \Rightarrow aAd \Rightarrow abAd \Rightarrow abcd$$



## Példa

## Grammatika

- ①  $S' \rightarrow S$
- ②  $S \rightarrow aAd$
- ③  $A \rightarrow bA$
- ④  $A \rightarrow c$

$$S' \Rightarrow S \Rightarrow aAd \Rightarrow abAd \Rightarrow abcd$$

léptetés



## Példa

## Grammatika

- ①  $S' \rightarrow S$
- ②  $S \rightarrow aAd$
- ③  $A \rightarrow bA$
- ④  $A \rightarrow c$

$$S' \Rightarrow S \Rightarrow aAd \Rightarrow abAd \Rightarrow abcd$$

redukálás (2)



## Példa

## Grammatika

- ①  $S' \rightarrow S$
- ②  $S \rightarrow aAd$
- ③  $A \rightarrow bA$
- ④  $A \rightarrow c$

$$S' \Rightarrow S \Rightarrow aAd \Rightarrow abAd \Rightarrow abcd$$

elfogadás

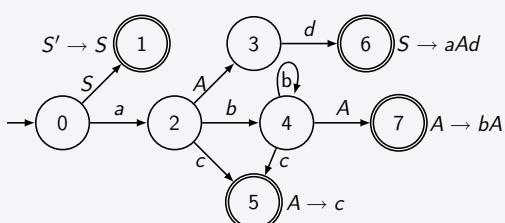


## Léptetés vagy redukálás?

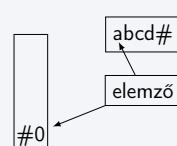
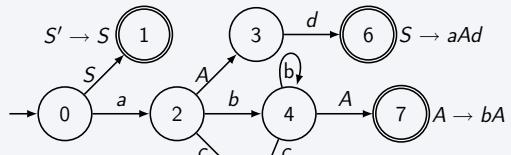
- LR(0) elemzés: előreolvasás nélkül kell döntenünk!

## Léptetés vagy redukálás?

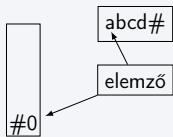
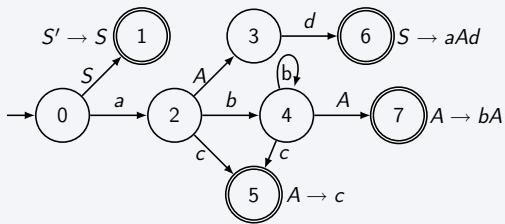
- LR(0) elemzés: előreolvasás nélkül kell döntenünk!
- Ötlet: készítsünk egy véges determinisztikus automatát
  - az átmeneteit a verembe kerülő szimbólumok határozzák meg
    - léptetéskor terminális
    - redukáláskor nemterminális
  - amikor elfogadó állapotba jut, akkor kell redukálni



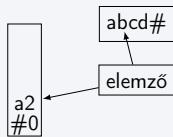
## Példa



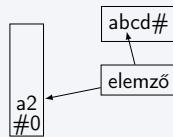
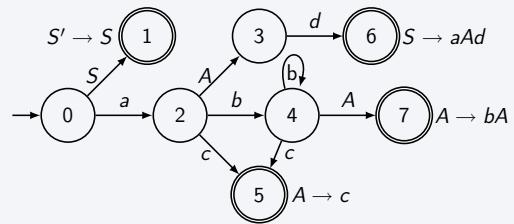
## Példa



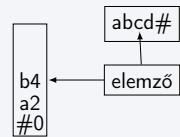
~~&gt; léptetés



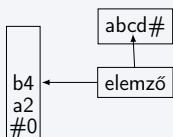
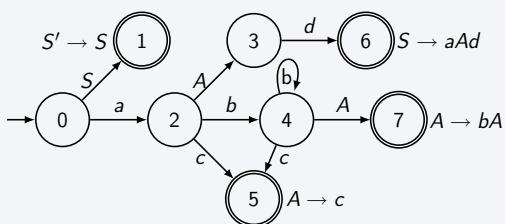
## Példa



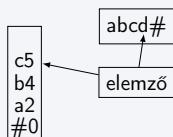
~~&gt; léptetés



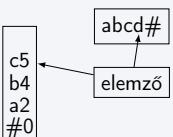
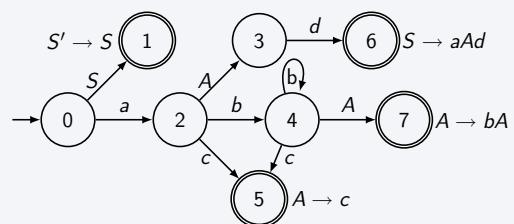
## Példa



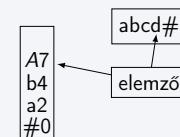
~~&gt; léptetés



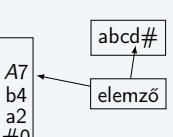
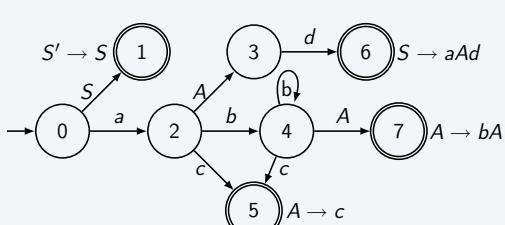
## Példa



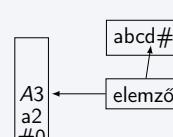
~~&gt; redukálás



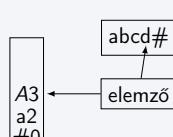
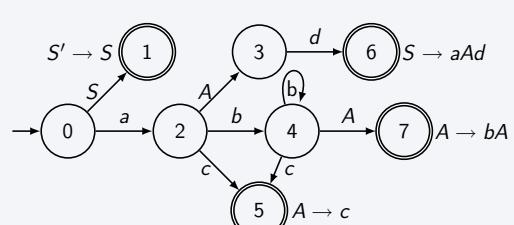
## Példa



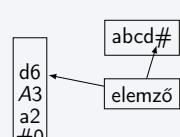
~~&gt; redukálás



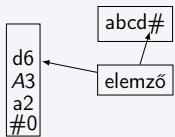
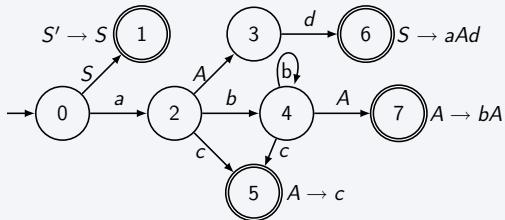
## Példa



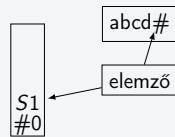
~~&gt; léptetés



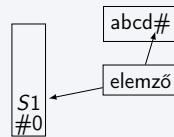
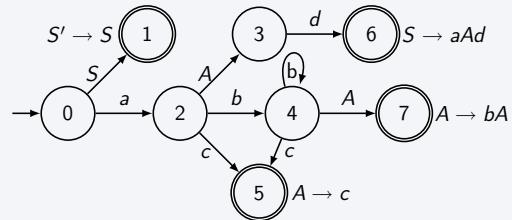
## Példa



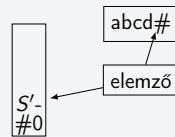
redukálás



## Példa



elfogadás



## LR(0) elemek

- Hogyan építsük fel az automatát?

## LR(0) elemek

- Hogyan építsük fel az automatát?
- Az automata állapotai azt mutatják meg, hogy melyik szabály építésében hol tartunk.
- Például:  $[S \rightarrow a.Ad]$  jelentse azt, hogy
  - az a szimbólumot már elemeztük,
  - az Ad rész még hátra van.

## LR(0) elemek

- Hogyan építsük fel az automatát?
- Az automata állapotai azt mutatják meg, hogy melyik szabály építésében hol tartunk.
- Például:  $[S \rightarrow a.Ad]$  jelentse azt, hogy
  - az a szimbólumot már elemeztük,
  - az Ad rész még hátra van.

## Definíció: LR(0) elem

Ha  $A \rightarrow \alpha$  a grammatika egy helyettesítési szabálya, akkor az  $\alpha = \alpha_1\alpha_2$  tetszőleges felbontás esetén  $[A \rightarrow \alpha_1.\alpha_2]$  a grammatika egy LR(0) eleme.

## LR(0) elemek

- Hogyan építsük fel az automatát?
- Az automata állapotai azt mutatják meg, hogy melyik szabály építésében hol tartunk.
- Például:  $[S \rightarrow a.Ad]$  jelentse azt, hogy
  - az a szimbólumot már elemeztük,
  - az Ad rész még hátra van.

## Definíció: LR(0) elem

Ha  $A \rightarrow \alpha$  a grammatika egy helyettesítési szabálya, akkor az  $\alpha = \alpha_1\alpha_2$  tetszőleges felbontás esetén  $[A \rightarrow \alpha_1.\alpha_2]$  a grammatika egy LR(0) eleme.

- Ha a szabály jobboldala  $n$  szimbólumot tartalmaz, akkor  $n + 1$  darab LR(0)-elem tartozik hozzá.

$A \rightarrow .aAd, A \rightarrow a.Ad, A \rightarrow aA.d, A \rightarrow aAd.$

## A lezárás művelet

- Az automata egy állapotához több  $LR(0)$ -elem is tartozhat.
  - Ezeket a halmazokat fogjuk **kanonikus halmazoknak** hívni.

## A lezárás művelet

- Az automata egy állapotához több  $LR(0)$ -elem is tartozhat.
  - Ezeket a halmazokat fogjuk **kanonikus halmazoknak** hívni.
- Milyen  $LR(0)$  elemek tartoznak egy halmazba?
  - Például: Ha  $[A \rightarrow a.Ad]$  állapotban vagyunk, akkor az  $A \rightarrow bA$  és  $A \rightarrow c$  szabályokat kezdhetjük építeni, azaz  $[A \rightarrow .bA]$  és  $[A \rightarrow .c]$  is hozzátarozik a halmazhoz.

## A lezárás művelet

- Az automata egy állapotához több  $LR(0)$ -elem is tartozhat.
  - Ezeket a halmazokat fogjuk **kanonikus halmazoknak** hívni.
- Milyen  $LR(0)$  elemek tartoznak egy halmazba?
  - Például: Ha  $[A \rightarrow a.Ad]$  állapotban vagyunk, akkor az  $A \rightarrow bA$  és  $A \rightarrow c$  szabályokat kezdhetjük építeni, azaz  $[A \rightarrow .bA]$  és  $[A \rightarrow .c]$  is hozzátarozik a halmazhoz.

### Definíció: lezárás (closure)

Ha  $\mathcal{I}$  a grammatika egy  $LR(0)$  elemhalmaza, akkor  $closure(\mathcal{I})$  a legszűkebb olyan halmaz, amely az alábbi tulajdonságokkal rendelkezik:

- $\mathcal{I} \subseteq closure(\mathcal{I})$
- ha  $[A \rightarrow \alpha.B\gamma] \in closure(\mathcal{I})$ , és  $B \rightarrow \beta$  a grammatika egy szabálya, akkor  $[B \rightarrow .\beta] \in closure(\mathcal{I})$ .

## A olvasás művelet

- Hogyan lépünk át az automata egyik állapotából a másikba?

## Az olvasás művelet

- Hogyan lépünk át az automata egyik állapotából a másikba?
- Ha  $[A \rightarrow a.Ad]$  állapotban vagyunk, és A kerül a verem tetejére, akkor  $[A \rightarrow aA.d]$  állapotba jutunk.

## Az olvasás művelet

- Hogyan lépünk át az automata egyik állapotából a másikba?
- Ha  $[A \rightarrow a.Ad]$  állapotban vagyunk, és A kerül a verem tetejére, akkor  $[A \rightarrow aA.d]$  állapotba jutunk.

### Definíció: olvasás (read)

Ha  $\mathcal{I}$  a grammatika egy  $LR(0)$  elemhalmaza,  $X$  pedig terminális vagy nemterminális szimbóluma, akkor  $read(\mathcal{I}, X)$  a legszűkebb olyan halmaz, amely az alábbi tulajdonsággal rendelkezik:

- ha  $[A \rightarrow \alpha.X\beta] \in \mathcal{I}$ , akkor  $closure([A \rightarrow \alpha.X.\beta]) \subseteq read(\mathcal{I}, X)$ .

## LR(0) kanonikus halmazok

Definíció: LR(0) kanonikus halmazok

- ❶  $\text{closure}([S' \rightarrow .S])$  a grammatika egy kanonikus halmaza.
- ❷ Ha  $\mathcal{I}$  a grammatika egy kanonikus elemhalmaza,  $X$  egy terminális vagy nemterminális szimbóluma, és  $\text{read}(\mathcal{I}, X)$  nem üres, akkor  $\text{read}(\mathcal{I}, X)$  is a grammatika egy kanonikus halmaza.
- ❸ Az első két szabálytal az összes kanonikus halmaz előáll.

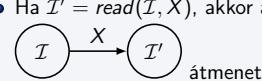
## LR(0) kanonikus halmazok

Definíció: LR(0) kanonikus halmazok

- ❶  $\text{closure}([S' \rightarrow .S])$  a grammatika egy kanonikus halmaza.
- ❷ Ha  $\mathcal{I}$  a grammatika egy kanonikus elemhalmaza,  $X$  egy terminális vagy nemterminális szimbóluma, és  $\text{read}(\mathcal{I}, X)$  nem üres, akkor  $\text{read}(\mathcal{I}, X)$  is a grammatika egy kanonikus halmaza.
- ❸ Az első két szabálytal az összes kanonikus halmaz előáll.

Az automata felépítése:

- A  $\text{closure}([S' \rightarrow .S])$  legyen a kezdőállapot.
- Ha  $\mathcal{I}' = \text{read}(\mathcal{I}, X)$ , akkor az automatában legyen



átmenet.

- A végállapotok azok a kanonikus halmazok, amelyekben olyan elemek vannak, ahol a pont a szabály végén van.

## Példa

Példa grammatika

- ❶  $S' \rightarrow S$
- ❷  $S \rightarrow aAd$
- ❸  $A \rightarrow bA$
- ❹  $A \rightarrow c$

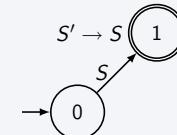


$$\mathcal{I}_0 = \text{closure}([S' \rightarrow .S]) = \{[S' \rightarrow .S], [S \rightarrow .aAd]\}$$

## Példa

Példa grammatika

- ❶  $S' \rightarrow S$
- ❷  $S \rightarrow aAd$
- ❸  $A \rightarrow bA$
- ❹  $A \rightarrow c$



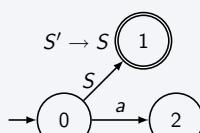
$$\mathcal{I}_0 = \text{closure}([S' \rightarrow .S]) = \{[S' \rightarrow .S], [S \rightarrow .aAd]\}$$

$$\mathcal{I}_1 = \text{read}(\mathcal{I}_0, S) = \text{closure}([S' \rightarrow S.]) = \{[S' \rightarrow S.]\}$$

## Példa

Példa grammatika

- ❶  $S' \rightarrow S$
- ❷  $S \rightarrow aAd$
- ❸  $A \rightarrow bA$
- ❹  $A \rightarrow c$



$$\mathcal{I}_0 = \text{closure}([S' \rightarrow .S]) = \{[S' \rightarrow .S], [S \rightarrow .aAd]\}$$

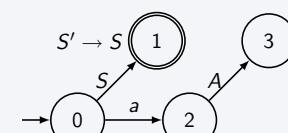
$$\mathcal{I}_1 = \text{read}(\mathcal{I}_0, S) = \{[S' \rightarrow S.]\}$$

$$\begin{aligned} \mathcal{I}_2 &= \text{read}(\mathcal{I}_0, a) = \text{closure}([S \rightarrow a.Ad]) \\ &= \{[S \rightarrow a.Ad], [A \rightarrow .bA], [A \rightarrow .c]\} \end{aligned}$$

## Példa

Példa grammatika

- ❶  $S' \rightarrow S$
- ❷  $S \rightarrow aAd$
- ❸  $A \rightarrow bA$
- ❹  $A \rightarrow c$



$$\mathcal{I}_0 = \text{closure}([S' \rightarrow .S]) = \{[S' \rightarrow .S], [S \rightarrow .aAd]\}$$

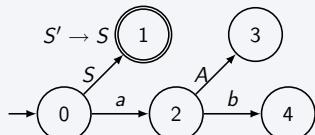
$$\mathcal{I}_1 = \text{read}(\mathcal{I}_0, S) = \{[S' \rightarrow S.]\}$$

$$\begin{aligned} \mathcal{I}_2 &= \text{read}(\mathcal{I}_0, a) = \{[S \rightarrow a.Ad], [A \rightarrow .bA], [A \rightarrow .c]\} \\ \mathcal{I}_3 &= \text{read}(\mathcal{I}_2, A) = \text{closure}([S \rightarrow aA.d]) = \{[S \rightarrow aA.d]\} \end{aligned}$$

## Példa

## Példa grammatika

- ❶  $S' \rightarrow S$
- ❷  $S \rightarrow aAd$
- ❸  $A \rightarrow bA$
- ❹  $A \rightarrow c$



$$\mathcal{I}_0 = \text{closure}([S' \rightarrow .S]) = \{[S' \rightarrow .S], [S \rightarrow .aAd]\}$$

$$\mathcal{I}_1 = \text{read}(\mathcal{I}_0, S) = \{[S' \rightarrow S.] \}$$

$$\mathcal{I}_2 = \text{read}(\mathcal{I}_0, a) = \{[S \rightarrow a.Ad], [A \rightarrow .bA], [A \rightarrow .c] \}$$

$$\mathcal{I}_3 = \text{read}(\mathcal{I}_2, A) = \{[S \rightarrow a.A.d] \}$$

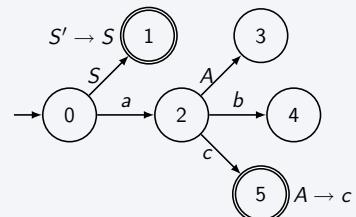
$$\mathcal{I}_4 = \text{read}(\mathcal{I}_2, b) = \text{closure}([A \rightarrow b.A])$$

$$= \{[A \rightarrow b.A], [A \rightarrow .bA], [A \rightarrow .c] \}$$

## Példa

## Példa grammatika

- ❶  $S' \rightarrow S$
- ❷  $S \rightarrow aAd$
- ❸  $A \rightarrow bA$
- ❹  $A \rightarrow c$



$$\mathcal{I}_0 = \text{closure}([S' \rightarrow .S]) = \{[S' \rightarrow .S], [S \rightarrow .aAd]\}$$

$$\mathcal{I}_1 = \text{read}(\mathcal{I}_0, S) = \{[S' \rightarrow S.] \}$$

$$\mathcal{I}_2 = \text{read}(\mathcal{I}_0, a) = \{[S \rightarrow a.Ad], [A \rightarrow .bA], [A \rightarrow .c] \}$$

$$\mathcal{I}_3 = \text{read}(\mathcal{I}_2, A) = \{[S \rightarrow a.A.d] \}$$

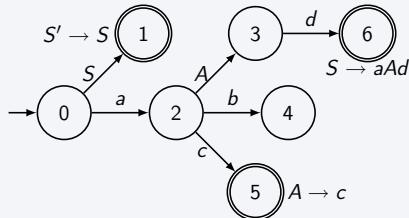
$$\mathcal{I}_4 = \text{read}(\mathcal{I}_2, b) = \{[A \rightarrow b.A], [A \rightarrow .bA], [A \rightarrow .c] \}$$

$$\mathcal{I}_5 = \text{read}(\mathcal{I}_2, c) = \text{closure}([A \rightarrow c.]) = \{[A \rightarrow c.] \}$$

## Példa

## Példa grammatika

- ❶  $S' \rightarrow S$
- ❷  $S \rightarrow aAd$
- ❸  $A \rightarrow bA$
- ❹  $A \rightarrow c$



$$\mathcal{I}_0 = \text{closure}([S' \rightarrow .S]) = \{[S' \rightarrow .S], [S \rightarrow .aAd]\}$$

$$\mathcal{I}_1 = \text{read}(\mathcal{I}_0, S) = \{[S' \rightarrow S.] \}$$

$$\mathcal{I}_2 = \text{read}(\mathcal{I}_0, a) = \{[S \rightarrow a.Ad], [A \rightarrow .bA], [A \rightarrow .c] \}$$

$$\mathcal{I}_3 = \text{read}(\mathcal{I}_2, A) = \{[S \rightarrow a.A.d] \}$$

$$\mathcal{I}_4 = \text{read}(\mathcal{I}_2, b) = \{[A \rightarrow b.A], [A \rightarrow .bA], [A \rightarrow .c] \}$$

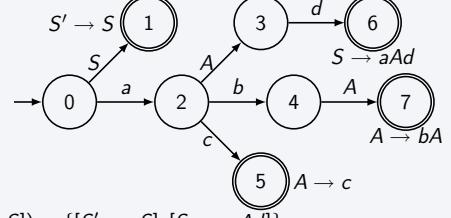
$$\mathcal{I}_5 = \text{read}(\mathcal{I}_2, c) = \{[A \rightarrow c.]\}$$

$$\mathcal{I}_6 = \text{read}(\mathcal{I}_3, d) = \text{closure}([S \rightarrow aAd.]) = \{[S \rightarrow aAd.] \}$$

## Példa

## Példa grammatika

- ❶  $S' \rightarrow S$
- ❷  $S \rightarrow aAd$
- ❸  $A \rightarrow bA$
- ❹  $A \rightarrow c$



$$\mathcal{I}_0 = \text{closure}([S' \rightarrow .S]) = \{[S' \rightarrow .S], [S \rightarrow .aAd]\}$$

$$\mathcal{I}_1 = \text{read}(\mathcal{I}_0, S) = \{[S' \rightarrow S.] \}$$

$$\mathcal{I}_2 = \text{read}(\mathcal{I}_0, a) = \{[S \rightarrow a.Ad], [A \rightarrow .bA], [A \rightarrow .c] \}$$

$$\mathcal{I}_3 = \text{read}(\mathcal{I}_2, A) = \{[S \rightarrow a.A.d] \}$$

$$\mathcal{I}_4 = \text{read}(\mathcal{I}_2, b) = \{[A \rightarrow b.A], [A \rightarrow .bA], [A \rightarrow .c] \}$$

$$\mathcal{I}_5 = \text{read}(\mathcal{I}_2, c) = \{[A \rightarrow c.]\}$$

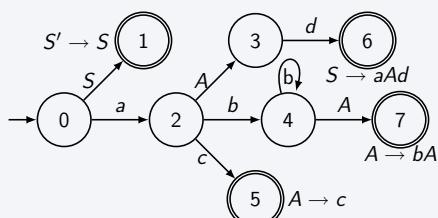
$$\mathcal{I}_6 = \text{read}(\mathcal{I}_3, d) = \{[S \rightarrow aAd.] \}$$

$$\mathcal{I}_7 = \text{read}(\mathcal{I}_4, A) = \text{closure}([A \rightarrow bA.]) = \{[A \rightarrow bA.] \}$$

## Példa

## Példa grammatika

- ❶  $S' \rightarrow S$
- ❷  $S \rightarrow aAd$
- ❸  $A \rightarrow bA$
- ❹  $A \rightarrow c$



$$\mathcal{I}_4 = \text{read}(\mathcal{I}_2, b) = \{[A \rightarrow b.A], [A \rightarrow .bA], [A \rightarrow .c] \}$$

$$\mathcal{I}_5 = \text{read}(\mathcal{I}_2, c) = \{[A \rightarrow c.]\}$$

...

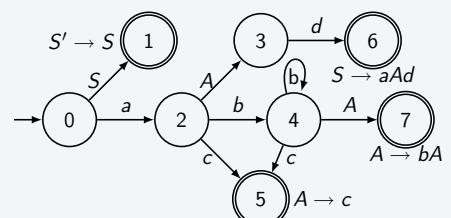
$$\text{read}(\mathcal{I}_4, b) = \text{closure}([A \rightarrow b.A])$$

$$= \{[A \rightarrow b.A], [A \rightarrow .bA], [A \rightarrow .c] \} = \mathcal{I}_4$$

## Példa

## Példa grammatika

- ❶  $S' \rightarrow S$
- ❷  $S \rightarrow aAd$
- ❸  $A \rightarrow bA$
- ❹  $A \rightarrow c$



$$\mathcal{I}_4 = \text{read}(\mathcal{I}_2, b) = \{[A \rightarrow b.A], [A \rightarrow .bA], [A \rightarrow .c] \}$$

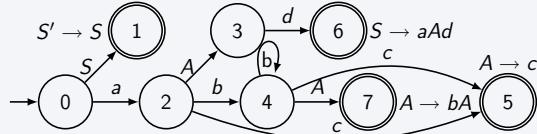
$$\mathcal{I}_5 = \text{read}(\mathcal{I}_2, c) = \{[A \rightarrow c.]\}$$

...

$$\text{read}(\mathcal{I}_4, b) = \mathcal{I}_4$$

$$\text{read}(\mathcal{I}_4, c) = \text{closure}([A \rightarrow c.]) = \{[A \rightarrow c.] \} = \mathcal{I}_5$$

## LR(0) elemző táblázat



állapot	akció	$S$	$A$	a	b	c	d
0	léptetés	1		2			
1	OK						
2	léptetés	3		4	5		
3	léptetés					6	
4	léptetés	7		4	5		
5	redukálás ( $A \rightarrow c$ )						
6	redukálás ( $S \rightarrow aAd$ )						
7	redukálás ( $A \rightarrow bA$ )						

## Véges-e az elemző létrehozása?

- a grammatikának véges sok szabálya van
- véges sok LR(0) eleme van

## Véges-e az elemző létrehozása?

- a grammatikának véges sok szabálya van
- véges sok LR(0) eleme van
- a closure függvény kiszámítása véges sok lépében befejeződik
- a read függvény kiszámítása véges sok lépében befejeződik

## Véges-e az elemző létrehozása?

- a grammatikának véges sok szabálya van
- véges sok LR(0) eleme van
- a closure függvény kiszámítása véges sok lépében befejeződik
- a read függvény kiszámítása véges sok lépében befejeződik
- a véges sok LR(0)-elem hatványhalmaza is véges
- a lehetséges kanonikus halmazok száma is véges

## Véges-e az elemző létrehozása?

- a grammatikának véges sok szabálya van
- véges sok LR(0) eleme van
- a closure függvény kiszámítása véges sok lépében befejeződik
- a read függvény kiszámítása véges sok lépében befejeződik
- a véges sok LR(0)-elem hatványhalmaza is véges
- a lehetséges kanonikus halmazok száma is véges

Az elemző táblázat (az automata) létrehozása véges sok lépében befejeződik.

## Helyes-e az elemző?

- Az automata pontosan akkor jut-e végállapotba, ha redukálni kell?
- A megfelelő a redukciót írja-e elő?

## Járható prefix

### Definíció: járható prefix

Ha az  $\alpha\beta x$  mondatforma nyele  $\beta$ , akkor az  $\alpha\beta$  prefixeit az  $\alpha\beta x$  járható prefixeinek nevezzük.

## Járható prefix

### Definíció: járható prefix

Ha az  $\alpha\beta x$  mondatforma nyele  $\beta$ , akkor az  $\alpha\beta$  prefixeit az  $\alpha\beta x$  járható prefixeinek nevezzük.

- A járható prefixeket olvassuk végig a nyél végének eléréséhez.
- Példa:  $S' \Rightarrow S \Rightarrow aAd \Rightarrow abAd \Rightarrow abcd$   
Az  $abAd$  mondatforma nyele a  $bA$ .  
A járható prefixei:  $a$ ,  $ab$ ,  $abA$

## Járható prefix

### Definíció: járható prefix

Ha az  $\alpha\beta x$  mondatforma nyele  $\beta$ , akkor az  $\alpha\beta$  prefixeit az  $\alpha\beta x$  járható prefixeinek nevezzük.

- A járható prefixeket olvassuk végig a nyél végének eléréséhez.
- Példa:  $S' \Rightarrow S \Rightarrow aAd \Rightarrow abAd \Rightarrow abcd$   
Az  $abAd$  mondatforma nyele a  $bA$ .  
A járható prefixei:  $a$ ,  $ab$ ,  $abA$
- Maximális járható prefix:** olyan járható prefix, amihez nem lehet újabb szimbólumot hozzávenni, hogy járható prefixet kapunk.
- Egy járható prefix épp akkor maximális, ha a végén van a nyél.

## Járható prefixre érvényes $LR(0)$ elemek

### Definíció: járható prefixre érvényes $LR(0)$ elemek

A grammatika egy  $[A \rightarrow \alpha.\beta]$   $LR(0)$ -eleme érvényes a  $\gamma\alpha$  járható prefixre nézve, ha

$$S' \Rightarrow^* \gamma A x \Rightarrow \gamma \alpha \beta x$$

## Járható prefixre érvényes $LR(0)$ elemek

### Definíció: járható prefixre érvényes $LR(0)$ elemek

A grammatika egy  $[A \rightarrow \alpha.\beta]$   $LR(0)$ -eleme érvényes a  $\gamma\alpha$  járható prefixre nézve, ha

$$S' \Rightarrow^* \gamma A x \Rightarrow \gamma \alpha \beta x$$

- Az érvényes  $LR(0)$  elemek az adott járható prefix „lehetséges folytatásait” adják meg.
- Példa: az  $ab$  járható prefixre érvényes  $LR(0)$ -elemek:  $[A \rightarrow b.A]$ ,  $[A \rightarrow .bA]$ ,  $[A \rightarrow .c]$ .  
 $S' \Rightarrow S \Rightarrow aAd \Rightarrow abAd$   
 $S' \Rightarrow S \Rightarrow aAd \Rightarrow abAd \Rightarrow ab\underline{b}Ad$   
 $S' \Rightarrow S \Rightarrow aAd \Rightarrow abcd$
- A maximális járható prefixekre érvényes  $LR(0)$  elemek azok, ahol a pont a szabály végén van.

## Az $LR(0)$ elemzés helyessége

### Tétel: az $LR(0)$ elemzés nagy tétele

Egy  $\gamma$  járható prefix hatására az elemző automatája a kezdőállapotból olyan állapotba kerül, amelyhez tartozó kanonikus halma az éppen a  $\gamma$  járható prefixre érvényes  $LR(0)$  elemeket tartalmazza.

## Az $LR(0)$ elemzés helyessége

### Tétel: az $LR(0)$ elemzés nagy tétele

Egy  $\gamma$  járható prefix hatására az elemző automatája a kezdőállapotból olyan állapotba kerül, amelyhez tartozó kanonikus halmaz éppen a  $\gamma$  járható prefixre érvényes  $LR(0)$  elemeket tartalmazza.

- Az elemző tehát addig fog léptetést előírni, amíg a veremben lévő járható prefix maximális nem lesz.
- Ha a járható prefix maximális, akkor az elemző redukálást ír elő.

## Konfliktusok a táblázatban

- Hol használtuk ki, hogy  $LR(0)$  a grammaтика?

## Konfliktusok a táblázatban

- Hol használtuk ki, hogy  $LR(0)$  a grammaтика?
- **Konfliktus:** Ha egy  $\mathcal{I}_k$  kanonikus halmaz alapján nem lehet egyértelműen eldönteni, hogy az adott állapotban milyen akciót kell végrehajtani.
  - léptetés/redukálás konfliktus: az egyik elem léptetést, egy másik redukálást ír elő
  - redukálás/redukálás konfliktus: az egyik elem az egyik szabály szerinti, a másik egy másik szabály szerinti redukciót ír elő

## Konfliktusok a táblázatban

- Hol használtuk ki, hogy  $LR(0)$  a grammaтика?
- **Konfliktus:** Ha egy  $\mathcal{I}_k$  kanonikus halmaz alapján nem lehet egyértelműen eldönteni, hogy az adott állapotban milyen akciót kell végrehajtani.
  - léptetés/redukálás konfliktus: az egyik elem léptetést, egy másik redukálást ír elő
  - redukálás/redukálás konfliktus: az egyik elem az egyik szabály szerinti, a másik egy másik szabály szerinti redukciót ír elő
- **Az  $LR(0)$  tulajdonság biztosítja a táblázat konfliktusmentes kitöltését!**

## LR elemzések (SLR(1) és LR(1) elemzések)

Fordítóprogramok előadás (A,C,T szakirány)

### Emlékeztető

## Emlékeztető: $LR(0)$ elemzés

- A lexikális elemző által előállított szimbólumsorozatot balról jobbra olvassuk, a szimbólumokat az elemző vermébe tesszük.

### Emlékeztető

## Emlékeztető: $LR(0)$ elemzés

- A lexikális elemző által előállított szimbólumsorozatot balról jobbra olvassuk, a szimbólumokat az elemző vermébe tesszük.
- Léptetés: egy új szimbólumot teszünk a bemenetről a verem tetejére.
- Redukálás: a verem tetején lévő szabály-jobboldalt helyettesítjük a szabály bal oldalán álló nemterminálissal.

### Emlékeztető

## Emlékeztető: $LR(0)$ elemzés

- A lexikális elemző által előállított szimbólumsorozatot balról jobbra olvassuk, a szimbólumokat az elemző vermébe tesszük.
- Léptetés: egy új szimbólumot teszünk a bemenetről a verem tetejére.
- Redukálás: a verem tetején lévő szabály-jobboldalt helyettesítjük a szabály bal oldalán álló nemterminálissal.
- $LR(0)$ : az alkalmazandó műveletről előreolvasás nélkül döntünk.
- A háttérben egy véges determinisztikus automata működik:
  - az automata átmeneteit a verem tetejére kerülő szimbólumok határozzák meg
  - ha az automata végállapotba jut, redukálni kell
  - egyéb állapotban pedig léptetni

### Emlékeztető

## Emlékeztető: $LR(0)$ elemzés

- Az automata állapotai a *kanonikus halmazok*.
  - „Melyik szabály építésében hol tartunk éppen?”
  - elemei az  $LR(0)$ -elemek

### Kanonikus halmaz és jelentése

A  $\{[S \rightarrow a.Ad], [A \rightarrow .bA], [A \rightarrow .c]\}$  kanonikus halmaz jelentése:

„Az adott állapotban az  $S \rightarrow a.Ad$ ,  $A \rightarrow bA$  és  $A \rightarrow c$  szabályok jobboldalait építhetjük. A  $S \rightarrow a.Ad$  szabályból az a szimbólumot már elemezük, az  $Ad$  rész még hátra van. A másik két szabály építése most kezdődhet.”

### Emlékeztető

## Emlékeztető: $LR(0)$ elemzés

- Lezáras (closure)* művelet: segítségével adhatók meg az egy kanonikus halmazba tartozó  $LR(0)$ -elemek.

### Lezáras

$$\text{closure}([S \rightarrow a.Ad]) = \{[S \rightarrow a.Ad], [A \rightarrow .bA], [A \rightarrow .c]\}$$

## Emlékeztető: LR(0) elemzés

- Lezáras (closure) művelet: segítségével adhatók meg az egy kanonikus halmazba tartozó LR(0)-elemek.

## Lezáras

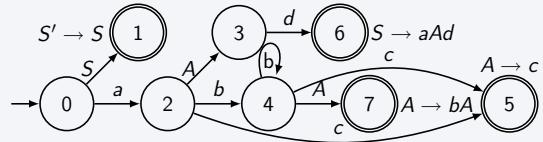
$$\text{closure}([S \rightarrow a.Ad]) = \{[S \rightarrow a.Ad], [A \rightarrow .bA], [A \rightarrow .c]\}$$

- Olvasás (read) művelet: megadja, hogy egy kanonikus halmazból egy adott szimbólum olvasásával melyik kanonikus halmazba jutunk. Ezek lesznek az automata átmenetei.

## Olvasás

$$\begin{aligned} \text{read}(\{[S \rightarrow a.Ad], [A \rightarrow .bA], [A \rightarrow .c]\}, b) &= \\ &= \text{closure}([A \rightarrow b.A]) = \\ &= \{[A \rightarrow b.A], [A \rightarrow .bA], [A \rightarrow .c]\} \end{aligned}$$

## Emlékeztető: LR(0) elemzés



...

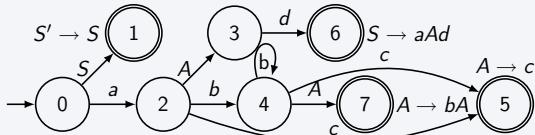
$$I_2 = \{[S \rightarrow a.Ad], [A \rightarrow .bA], [A \rightarrow .c]\}$$

...

$$\begin{aligned} I_4 &= \text{read}(I_2, b) = \{[A \rightarrow b.A], [A \rightarrow .bA], [A \rightarrow .c]\} \\ I_5 &= \text{read}(I_4, c) = \text{read}(I_4, c) = \{[A \rightarrow c]\} \end{aligned}$$

Elfogadó állapot: „a hozzá tartozó elemeknek a végén van a pont”

## Emlékeztető: LR(0) elemzés



állapot	akció	$S$	$A$	$a$	$b$	$c$	$d$
0	léptetés	1		2			
1	OK						
2	léptetés		3	4	5		
3	léptetés					6	
4	léptetés		7	4	5		
5	redukálás ( $A \rightarrow c$ )						
6	redukálás ( $S \rightarrow aAd$ )						
7	redukálás ( $A \rightarrow bA$ )						

## Konfliktusok

- Az LR(0) tulajdonság biztosította, hogy a táblázat egy cellájába sem kerül két különböző műveletet, azaz a táblázat konfliktusmentes.
- Mi történik, ha nem LR(0) a grammatika?

## Konfliktusok

- Az LR(0) tulajdonság biztosította, hogy a táblázat egy cellájába sem kerül két különböző műveletet, azaz a táblázat konfliktusmentes.
- Mi történik, ha nem LR(0) a grammatika?

## A helyes zárójelezés

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow \epsilon \mid (S)S \end{aligned}$$

## Konfliktusok

- Az LR(0) tulajdonság biztosította, hogy a táblázat egy cellájába sem kerül két különböző műveletet, azaz a táblázat konfliktusmentes.
- Mi történik, ha nem LR(0) a grammatika?

## A helyes zárójelezés

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow \epsilon \mid (S)S \\ S' &\Rightarrow S \Rightarrow (S)S \Rightarrow ((S)S) \Rightarrow (\dots)S \Rightarrow (\dots) \end{aligned}$$

## Konfliktusok

- Az LR(0) tulajdonság biztosította, hogy a táblázat egy cellájába sem kerül két különböző műveletet, azaz a táblázat *konfliktusmentes*.
- Mi történik, ha nem LR(0) a grammatika?

## A helyes zárójelezés

 $S' \rightarrow S$   
 $S \rightarrow \epsilon \mid (S)S$ 
 $S' \Rightarrow S \Rightarrow (S)S \Rightarrow (S) \Rightarrow ()$   
 $S' \Rightarrow S \Rightarrow (S)S \Rightarrow (S) \Rightarrow ((S)S) \Rightarrow ((S)) \Rightarrow ((())$ 

Az piros részek elolvasása után:

- az első esetben  $S \rightarrow \epsilon$  szerinti redukciót kell végrahajtani,
- a második esetben léptetni kell.

Előreolvasás nélkül nem tudunk dönteneni, nem LR(0) grammatika.

## Példa: helyes zárójelezés

## Példa grammatika

- ①  $S' \rightarrow S$   
 ②  $S \rightarrow \epsilon \mid (S)S$



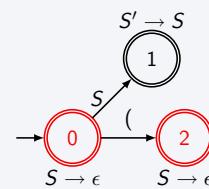
$\mathcal{I}_0 = closure([S' \rightarrow .S]) = \{[S' \rightarrow .S], [S \rightarrow .], [S \rightarrow .(S)S]\}$

## Példa: helyes zárójelezés

## Példa: helyes zárójelezés

## Példa grammatika

- ①  $S' \rightarrow S$   
 ②  $S \rightarrow \epsilon \mid (S)S$



$\mathcal{I}_0 = closure([S' \rightarrow .S]) = \{[S' \rightarrow .S], [S \rightarrow .], [S \rightarrow .(S)S]\}$

$\mathcal{I}_1 = read(\mathcal{I}_0, S) = \{[S' \rightarrow S.] \}$

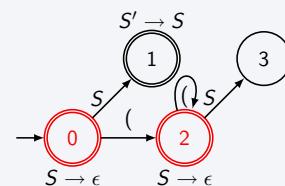
$\mathcal{I}_2 = read(\mathcal{I}_0, ()) = \{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\}$

## Példa: helyes zárójelezés

## Példa: helyes zárójelezés

## Példa grammatika

- ①  $S' \rightarrow S$   
 ②  $S \rightarrow \epsilon \mid (S)S$



$\mathcal{I}_0 = closure([S' \rightarrow .S]) = \{[S' \rightarrow .S], [S \rightarrow .], [S \rightarrow .(S)S]\}$

$\mathcal{I}_1 = read(\mathcal{I}_0, S) = \{[S' \rightarrow S.] \}$

$\mathcal{I}_2 = read(\mathcal{I}_0, ()) = \{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\}$

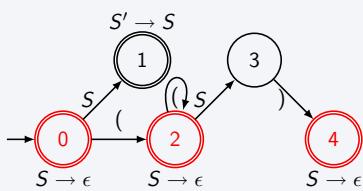
$\mathcal{I}_3 = read(\mathcal{I}_2, S) = \{[S \rightarrow (S.)S] \}$

$read(\mathcal{I}_2, ()) = \{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\} = \mathcal{I}_2$

## Példa: helyes zárójelezés

## Példa grammatika

- ➊  $S' \rightarrow S$
- ➋  $S \rightarrow \epsilon \mid (S)S$

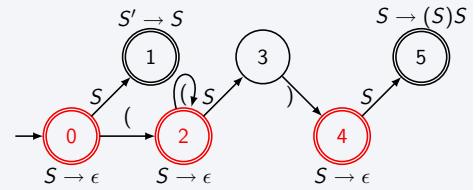


$\mathcal{I}_0 = \text{closure}([S' \rightarrow .S]) = \{[S' \rightarrow .S], [S \rightarrow .], [S \rightarrow .(S)S]\}$   
 $\mathcal{I}_1 = \text{read}(\mathcal{I}_0, S) = \{[S' \rightarrow S.\}\}$   
 $\mathcal{I}_2 = \text{read}(\mathcal{I}_0, ()) = \{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\}$   
 $\mathcal{I}_3 = \text{read}(\mathcal{I}_2, S) = \{[S \rightarrow (S.)S]\}$   
 $\text{read}(\mathcal{I}_2, ()) = \{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\} = \mathcal{I}_2$   
 $\mathcal{I}_4 = \text{read}(\mathcal{I}_3, ()) = \{[S \rightarrow (S).S], [S \rightarrow .], [S \rightarrow .(S)S]\}$

## Példa: helyes zárójelezés

## Példa grammatika

- ➊  $S' \rightarrow S$
- ➋  $S \rightarrow \epsilon \mid (S)S$

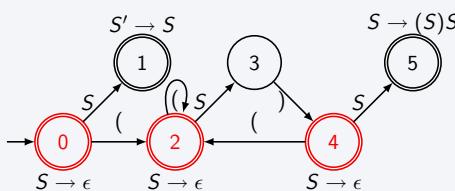


$\mathcal{I}_0 = \text{closure}([S' \rightarrow .S]) = \{[S' \rightarrow .S], [S \rightarrow .], [S \rightarrow .(S)S]\}$   
 $\mathcal{I}_1 = \text{read}(\mathcal{I}_0, S) = \{[S' \rightarrow S.\}\}$   
 $\mathcal{I}_2 = \text{read}(\mathcal{I}_0, ()) = \{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\}$   
 $\mathcal{I}_3 = \text{read}(\mathcal{I}_2, S) = \{[S \rightarrow (S.)S]\}$   
 $\text{read}(\mathcal{I}_2, ()) = \{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\} = \mathcal{I}_2$   
 $\mathcal{I}_4 = \text{read}(\mathcal{I}_3, ()) = \{[S \rightarrow (S).S], [S \rightarrow .], [S \rightarrow .(S)S]\}$   
 $\mathcal{I}_5 = \text{read}(\mathcal{I}_4, S) = \{[S \rightarrow (S)S.\}\}$

## Példa: helyes zárójelezés

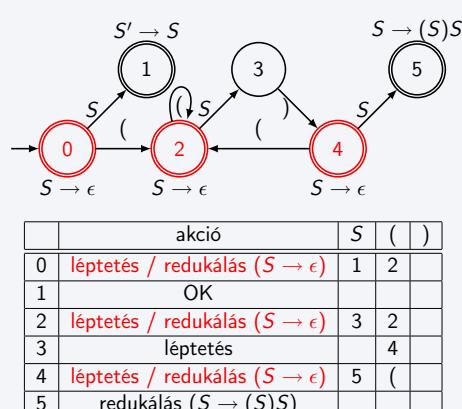
## Példa grammatika

- ➊  $S' \rightarrow S$
- ➋  $S \rightarrow \epsilon \mid (S)S$



$\mathcal{I}_0 = \text{closure}([S' \rightarrow .S]) = \{[S' \rightarrow .S], [S \rightarrow .], [S \rightarrow .(S)S]\}$   
 $\mathcal{I}_1 = \text{read}(\mathcal{I}_0, S) = \{[S' \rightarrow S.\}\}$   
 $\mathcal{I}_2 = \text{read}(\mathcal{I}_0, ()) = \{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\}$   
 $\mathcal{I}_3 = \text{read}(\mathcal{I}_2, S) = \{[S \rightarrow (S.)S]\}$   
 $\text{read}(\mathcal{I}_2, ()) = \{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\} = \mathcal{I}_2$   
 $\mathcal{I}_4 = \text{read}(\mathcal{I}_3, ()) = \{[S \rightarrow (S).S], [S \rightarrow .], [S \rightarrow .(S)S]\}$   
 $\mathcal{I}_5 = \text{read}(\mathcal{I}_4, S) = \{[S \rightarrow (S)S.\}\}$   
 $\text{read}(\mathcal{I}_4, ()) = \{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\} = \mathcal{I}_2$

## Konfliktusok az LR(0) elemző táblázatban



## Az SLR(1) elemzés alapötlete

- Olvassunk előre egy szimbólumot!
- léptessünk, ha az automata tud lépni az előreolvasott szimbólummal
- redukáljunk, ha az előreolvasott szimbólum benne van a szabályhoz tartozó nemterminális  $FOLLOW_1$  halmazában

## Az SLR(1) elemzés alapötlete

- Olvassunk előre egy szimbólumot!
- léptessünk, ha az automata tud lépni az előreolvasott szimbólummal
- redukáljunk, ha az előreolvasott szimbólum benne van a szabályhoz tartozó nemterminális  $FOLLOW_1$  halmazában

## A helyes zárójelezés

$S' \Rightarrow S \Rightarrow (S)S \Rightarrow (S) \Rightarrow ()$   
 $S' \Rightarrow S \Rightarrow (S)S \Rightarrow (S) \Rightarrow ((S)S) \Rightarrow (( )) \Rightarrow (())$

$\mathcal{I}_2 = \{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\}$

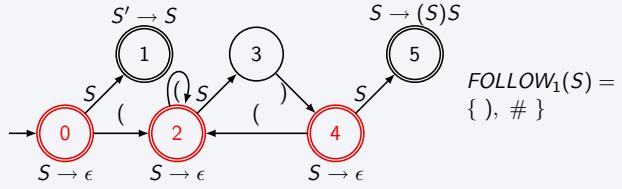
Az piros részek elolvasása után  $\mathcal{I}_2$  állapotban van az automata:

- az első esetben  $S \rightarrow \epsilon$  szerinti redukciót kell végrahajtani, mert  $) \in FOLLOW_1(S)$ .
- a második esetben léptetni kell, mert  $[S \rightarrow .(S)S] \in \mathcal{I}_2$ .

## Az SLR(1) elemzés szabályai

- Ha az aktuális állapot  $i$ , és az előreolvasás eredménye az a szimbólum:
  - ha  $[A \rightarrow \alpha.a\beta] \in \mathcal{I}_i$  és  $\text{read}(\mathcal{I}_i, a) = \mathcal{I}_j$ , akkor léptetni kell, és átlépni a  $j$  állapotba,
  - ha  $[A \rightarrow \alpha.] \in \mathcal{I}_i$  ( $A \neq S'$ ) és  $a \in FOLLOW_1(A)$ , akkor redukálni kell  $A \rightarrow \alpha$  szabály szerint,
  - ha  $[S' \rightarrow S] \in \mathcal{I}_i$  és  $a = \#$ , akkor el kell fogadni a szöveget,
  - minden más esetben hibát kell jelezni.
- Ha az  $i$  állapotban  $A$  kerül a verem tetejére:
  - ha  $\text{read}(\mathcal{I}_i, A) = \mathcal{I}_j$ , akkor át kell lépni a  $j$  állapotba,
  - egyébként hibát kell jelezni.

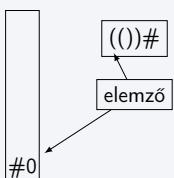
## A helyes zárójelezés SLR(1) elemző táblázata



	(	)	#	S
0	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	1
1			OK	
2	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	3
3		léptetés, 4		
4	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	5
5		redukálás ( $S \rightarrow (S)S$ )	redukálás ( $S \rightarrow (S)S$ )	

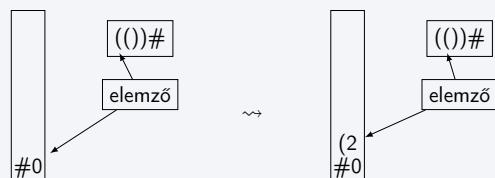
## Példa

	(	)	#	S
0	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	1
1			OK	
2	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	3
3		léptetés, 4		
4	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	5
5		redukálás ( $S \rightarrow (S)S$ )	redukálás ( $S \rightarrow (S)S$ )	



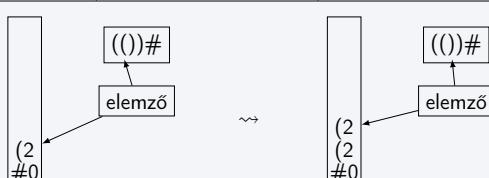
## Példa

	(	)	#	S
0	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	1
1			OK	
2	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	3
3		léptetés, 4		
4	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	5
5		redukálás ( $S \rightarrow (S)S$ )	redukálás ( $S \rightarrow (S)S$ )	



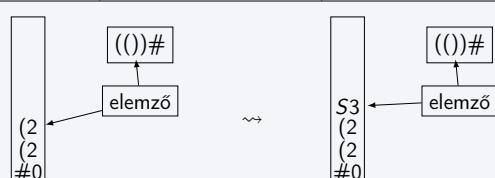
## Példa

	(	)	#	S
0	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	1
1			OK	
2	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	3
3		léptetés, 4		
4	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	5
5		redukálás ( $S \rightarrow (S)S$ )	redukálás ( $S \rightarrow (S)S$ )	



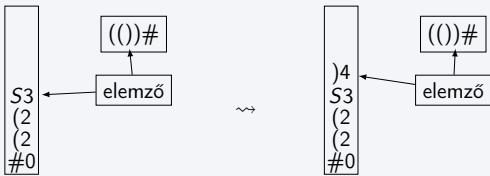
## Példa

	(	)	#	S
0	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	1
1			OK	
2	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	3
3		léptetés, 4		
4	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	5
5		redukálás ( $S \rightarrow (S)S$ )	redukálás ( $S \rightarrow (S)S$ )	

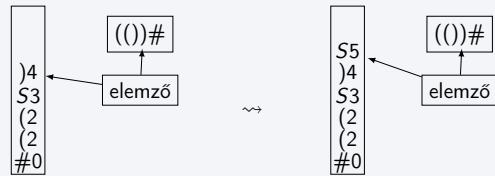


## Példa

	(	)	#	S
0	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	1
1			OK	
2	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	3
3		léptetés, 4		
4	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	5
5		redukálás ( $S \rightarrow (S)S$ )	redukálás ( $S \rightarrow (S)S$ )	

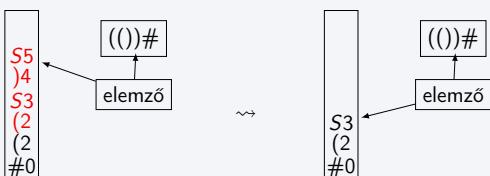


	(	)	#	S
0	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	1
1			OK	
2	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	3
3		léptetés, 4		
4	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	5
5		redukálás ( $S \rightarrow (S)S$ )	redukálás ( $S \rightarrow (S)S$ )	

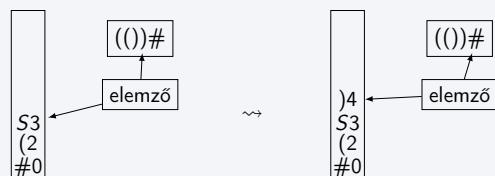


Példa

	(	)	#	S
0	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	1
1			OK	
2	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	3
3		léptetés, 4		
4	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	5
5		redukálás ( $S \rightarrow (S)S$ )	redukálás ( $S \rightarrow (S)S$ )	

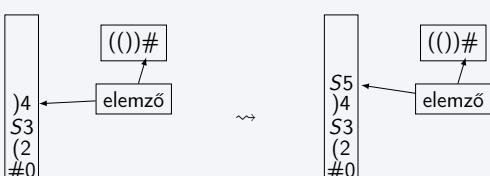


	(	)	#	S
0	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	1
1			OK	
2	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	3
3		léptetés, 4		
4	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	5
5		redukálás ( $S \rightarrow (S)S$ )	redukálás ( $S \rightarrow (S)S$ )	

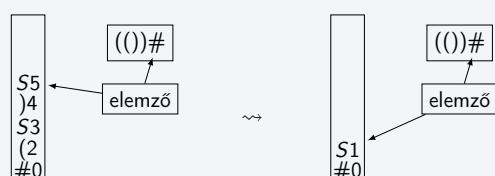


Példa

	(	)	#	S
0	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	1
1			OK	
2	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	3
3		léptetés, 4		
4	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	5
5		redukálás ( $S \rightarrow (S)S$ )	redukálás ( $S \rightarrow (S)S$ )	

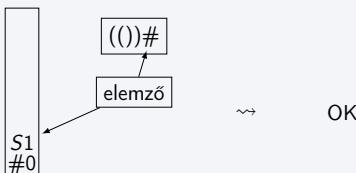


	(	)	#	S
0	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	1
1			OK	
2	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	3
3		léptetés, 4		
4	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	5
5		redukálás ( $S \rightarrow (S)S$ )	redukálás ( $S \rightarrow (S)S$ )	



## Példa

	(	)	#	S
0	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	1
1			OK	
2	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	3
3		léptetés, 4		
4	léptetés, 2	redukálás ( $S \rightarrow \epsilon$ )	redukálás ( $S \rightarrow \epsilon$ )	5
5		redukálás ( $S \rightarrow (S)S$ )	redukálás ( $S \rightarrow (S)S$ )	



## SLR(1) grammatika

## Definíció: SLR(1) grammatika

Egy kiegészített grammatika  $SLR(1)$  grammatika, ha az  $SLR(1)$  elemző táblázata konfliktusmentes.

- elnevezés: „Simple LR”
- jobb, mint az  $LR(0)$
- a valódi programnyelvek nyelvtanai általában nem  $SLR(1)$  nyelvtanok

## Probléma az SLR(1) elemzéssel

## Példa grammatika

$S' \rightarrow S$	Egy program
$S \rightarrow U \mid E$	az egy utasítás vagy egy értékadás.
$U \rightarrow a$	Egy utasítás az egy azonosító szimbólum.
$E \rightarrow V = V$	Egy értékadás változó legyen egyenlő változó alakú.
$V \rightarrow a$	Egy változó az egy azonosító szimbólum.

## Probléma az SLR(1) elemzéssel

## Példa grammatika

$S' \rightarrow S$	Egy program
$S \rightarrow U \mid E$	az egy utasítás vagy egy értékadás.
$U \rightarrow a$	Egy utasítás az egy azonosító szimbólum.
$E \rightarrow V = V$	Egy értékadás változó legyen egyenlő változó alakú.
$V \rightarrow a$	Egy változó az egy azonosító szimbólum.

$$\mathcal{I}_0 = \{[S' \rightarrow .S], [S \rightarrow .U], [S \rightarrow .E], [U \rightarrow .a], [E \rightarrow .V = V], [V \rightarrow .a]\}$$

...

$$\mathcal{I}_4 = \text{read}(\mathcal{I}_0, a) = \{[U \rightarrow a], [V \rightarrow a]\}$$

## Probléma az SLR(1) elemzéssel

## Példa grammatika

$S' \rightarrow S$	Egy program
$S \rightarrow U \mid E$	az egy utasítás vagy egy értékadás.
$U \rightarrow a$	Egy utasítás az egy azonosító szimbólum.
$E \rightarrow V = V$	Egy értékadás változó legyen egyenlő változó alakú.
$V \rightarrow a$	Egy változó az egy azonosító szimbólum.

## Probléma az SLR(1) elemzéssel

$$\mathcal{I}_4 = \text{read}(\mathcal{I}_0, a) = \{[U \rightarrow a], [V \rightarrow a]\}$$

$$FOLLOW_1(U) = \{\#\} \text{ és } FOLLOW_1(V) = \{=, \#\}$$

Redukálás / redukálás konfliktus!

## Probléma az SLR(1) elemzéssel

$$\mathcal{I}_4 = \text{read}(\mathcal{I}_0, a) = \{[U \rightarrow a], [V \rightarrow a]\}$$

$\text{FOLLOW}_1(U) = \{\#\}$  és  $\text{FOLLOW}_1(V) = \{=, \#\}$   
Redukálás / redukálás konfliktus!

Az  $a\#$  szövegből az  $a$  elolvasása után az SLR(1) elemző nem tud dönteni a  $U \rightarrow a$  és  $V \rightarrow a$  szerinti redukciók között, mert a következő szimbólum ( $\#$ ) benne van az  $U$  és a  $V \text{ FOLLOW}_1$  halmazában is.

## Probléma az SLR(1) elemzéssel

$$\mathcal{I}_4 = \text{read}(\mathcal{I}_0, a) = \{[U \rightarrow a], [V \rightarrow a]\}$$

$\text{FOLLOW}_1(U) = \{\#\}$  és  $\text{FOLLOW}_1(V) = \{=, \#\}$   
Redukálás / redukálás konfliktus!

Az  $a\#$  szövegből az  $a$  elolvasása után az SLR(1) elemző nem tud dönteni a  $U \rightarrow a$  és  $V \rightarrow a$  szerinti redukciók között, mert a következő szimbólum ( $\#$ ) benne van az  $U$  és a  $V \text{ FOLLOW}_1$  halmazában is.

Pedig a szöveg elején a  $V$ -t csak az  $=$  szimbólum követheti...

Az  $LR(1)$  elemzés alapötlete

- a  $\text{FOLLOW}_1$  halmaz globális az egész grammaticákra
- előfordulhat, hogy egy adott állapotban a  $\text{FOLLOW}_1$  halmaznak nem minden eleme követheti a szabályt

Az  $LR(1)$  elemzés alapötlete

- a  $\text{FOLLOW}_1$  halmaz globális az egész grammaticákra
- előfordulhat, hogy egy adott állapotban a  $\text{FOLLOW}_1$  halmaznak nem minden eleme követheti a szabályt
- Vegyük hozzá az  $LR(0)$  elemekhez azokat a szimbólumokat, amik követhetik a szabályt az adott állapotban!

 $LR(1)$  elemekDefiníció:  $LR(1)$  elem

Ha  $A \rightarrow \alpha$  a grammatica egy helyettesítési szabálya, akkor az  $\alpha = \alpha_1\alpha_2$  tetszőleges felbontás és a terminális szimbólum (vagy  $a = \#$ ) esetén  $[A \rightarrow \alpha_1.\alpha_2, a]$  a grammatica egy  $LR(1)$ -eleme.

$A \rightarrow \alpha_1.\alpha_2$  az  $LR(1)$  elem magja, a pedig az előreolvasási szimbóluma.

 $LR(1)$  elemekDefiníció:  $LR(1)$  elem

Ha  $A \rightarrow \alpha$  a grammatica egy helyettesítési szabálya, akkor az  $\alpha = \alpha_1\alpha_2$  tetszőleges felbontás és a terminális szimbólum (vagy  $a = \#$ ) esetén  $[A \rightarrow \alpha_1.\alpha_2, a]$  a grammatica egy  $LR(1)$ -eleme.

$A \rightarrow \alpha_1.\alpha_2$  az  $LR(1)$  elem magja, a pedig az előreolvasási szimbóluma.

- $[V \rightarrow a., =]$  jelentése: a  $V \rightarrow a$  szabály építését befejeztük és a szabályt az  $=$  szimbólum követheti.

## A lezárás művelet

- Ha  $[V \rightarrow .V = V, \#]$  állapotban vagyunk, akkor a  $V \rightarrow a$  szabályt kezdhetjük építeni, amit az = szimbólum követhet.
- Tehát az adott kanonikus halmazhoz  $[V \rightarrow .a, =]$  is hozzátarozik.

## A lezárás művelet

- Ha  $[V \rightarrow .V = V, \#]$  állapotban vagyunk, akkor a  $V \rightarrow a$  szabályt kezdhetjük építeni, amit az = szimbólum követhet.
- Tehát az adott kanonikus halmazhoz  $[V \rightarrow .a, =]$  is hozzátarozik.

## Definíció: lezárás (closure)

Ha  $\mathcal{I}$  a grammatika egy **LR(1)** elemhalmaza, akkor  $closure(\mathcal{I})$  a legszűkebb olyan halmaz, amely az alábbi tulajdonságokkal rendelkezik:

- $\mathcal{I} \subseteq closure(\mathcal{I})$
- ha  $[A \rightarrow \alpha.B\gamma.a] \in closure(\mathcal{I})$ , és  $B \rightarrow \beta$  a grammatika egy szabálya, akkor  $\forall b \in FIRST_1(\gamma a)$  esetén  $[B \rightarrow .\beta.b] \in closure(\mathcal{I})$

## Az olvasás művelet

- Ha  $[V \rightarrow .V = V, \#]$  állapotban vagyunk, és  $V$  kerül a verem tetejére, akkor  $[V \rightarrow V. = V, \#]$  állapotba jutunk.

## LR(1) kanonikus halmazok

## Definíció: LR(1) kanonikus halmazok

- $closure([S' \rightarrow .S, \#])$  a grammatika egy kanonikus halmaza.
- Ha  $\mathcal{I}$  a grammatika egy kanonikus elemhalmaza,  $X$  pedig terminális vagy nemterminális szimbóluma, akkor  $read(\mathcal{I}, X)$  nem üres, akkor  $read(\mathcal{I}, X)$  a grammatika egy kanonikus halmaza.
- Az első két szabálytal a összes kanonikus halmaz előáll.

## Az LR(1) elemzés szabályai

- Ha az aktuális állapot  $i$ , és az előreolvasás eredménye az  $a$  szimbólum:
  - ha  $[A \rightarrow \alpha.a\beta, b] \in \mathcal{I}_i$  és  $read(\mathcal{I}_i, a) = \mathcal{I}_j$ , akkor léptetni kell, és átlépni a  $j$  állapotba,
  - ha  $[A \rightarrow \alpha., a] \in \mathcal{I}_i$  ( $A \neq S'$ ), akkor redukálni kell  $A \rightarrow \alpha$  szabály szerint,
  - ha  $[S' \rightarrow S., \#] \in \mathcal{I}_i$  és  $a = \#$ , akkor el kell fogadni a szöveget,
  - minden más esetben hibát kell jelezni.
- Ha az  $i$  állapotban  $A$  kerül a verem tetejére:
  - ha  $read(\mathcal{I}_i, A) = \mathcal{I}_j$ , akkor át kell lépni a  $j$  állapotba,
  - egyébként hibát kell jelezni.

## Példa

## Példa grammatika

$$S' \rightarrow S \quad S \rightarrow U \mid E \quad U \rightarrow a \quad E \rightarrow V = V \quad V \rightarrow a$$

## Példa

## Példa grammatika

$$S' \rightarrow S \quad S \rightarrow U \mid E \quad U \rightarrow a \quad E \rightarrow V = V \quad V \rightarrow a$$

$$\begin{aligned} \mathcal{I}_0 &= closure([S' \rightarrow .S, \#]) = \\ &= \{[S' \rightarrow .S, \#], [S \rightarrow .U, \#], [S \rightarrow .E, \#], [U \rightarrow .a, \#], \\ &\quad [E \rightarrow .V = V, \#], [V \rightarrow .a, =]\} \end{aligned}$$

## Példa

## Példa grammatika

$$S' \rightarrow S \quad S \rightarrow U \mid E \quad U \rightarrow a \quad E \rightarrow V = V \quad V \rightarrow a$$

$$\begin{aligned} \mathcal{I}_0 &= closure([S' \rightarrow .S, \#]) = \\ &= \{[S' \rightarrow .S, \#], [S \rightarrow .U, \#], [S \rightarrow .E, \#], [U \rightarrow .a, \#], \\ &\quad [E \rightarrow .V = V, \#], [V \rightarrow .a, =]\} \end{aligned}$$

$$\mathcal{I}_1 = read(\mathcal{I}_0, S) = \{[S' \rightarrow S., \#]\}$$

## Példa

## Példa grammatika

$$S' \rightarrow S \quad S \rightarrow U \mid E \quad U \rightarrow a \quad E \rightarrow V = V \quad V \rightarrow a$$

$$\begin{aligned} \mathcal{I}_0 &= closure([S' \rightarrow .S, \#]) = \\ &= \{[S' \rightarrow .S, \#], [S \rightarrow .U, \#], [S \rightarrow .E, \#], [U \rightarrow .a, \#], \\ &\quad [E \rightarrow .V = V, \#], [V \rightarrow .a, =]\} \end{aligned}$$

$$\mathcal{I}_1 = read(\mathcal{I}_0, S) = \{[S' \rightarrow S., \#]\}$$

$$\mathcal{I}_2 = read(\mathcal{I}_0, U) = \{[S \rightarrow U., \#]\}$$

## Példa

## Példa grammatika

$$S' \rightarrow S \quad S \rightarrow U \mid E \quad U \rightarrow a \quad E \rightarrow V = V \quad V \rightarrow a$$

$$\begin{aligned} \mathcal{I}_0 &= closure([S' \rightarrow .S, \#]) = \\ &= \{[S' \rightarrow .S, \#], [S \rightarrow .U, \#], [S \rightarrow .E, \#], [U \rightarrow .a, \#], \\ &\quad [E \rightarrow .V = V, \#], [V \rightarrow .a, =]\} \end{aligned}$$

$$\mathcal{I}_1 = read(\mathcal{I}_0, S) = \{[S' \rightarrow S., \#]\}$$

$$\mathcal{I}_2 = read(\mathcal{I}_0, U) = \{[S \rightarrow U., \#]\}$$

$$\mathcal{I}_3 = read(\mathcal{I}_0, E) = \{[S \rightarrow E., \#]\}$$

## Példa

## Példa grammatika

$$S' \rightarrow S \quad S \rightarrow U \mid E \quad U \rightarrow a \quad E \rightarrow V = V \quad V \rightarrow a$$

$$\begin{aligned} \mathcal{I}_0 &= closure([S' \rightarrow .S, \#]) = \\ &= \{[S' \rightarrow .S, \#], [S \rightarrow .U, \#], [S \rightarrow .E, \#], [U \rightarrow .a, \#], \\ &\quad [E \rightarrow .V = V, \#], [V \rightarrow .a, =]\} \end{aligned}$$

$$\mathcal{I}_1 = read(\mathcal{I}_0, S) = \{[S' \rightarrow S., \#]\}$$

$$\mathcal{I}_2 = read(\mathcal{I}_0, U) = \{[S \rightarrow U., \#]\}$$

$$\mathcal{I}_3 = read(\mathcal{I}_0, E) = \{[S \rightarrow E., \#]\}$$

$$\mathcal{I}_4 = read(\mathcal{I}_0, a) = \{[U \rightarrow a., \#], [V \rightarrow a., =]\} \text{ Nincs konfliktus!}$$

## Példa

## Példa grammatika

$$S' \rightarrow S \quad S \rightarrow U \mid E \quad U \rightarrow a \quad E \rightarrow V = V \quad V \rightarrow a$$

$\mathcal{I}_0 = closure([S' \rightarrow .S, \#]) =$   
 $= \{[S' \rightarrow .S, \#], [S \rightarrow .U, \#], [S \rightarrow .E, \#], [U \rightarrow .a, \#],$   
 $[E \rightarrow .V = V, \#], [V \rightarrow .a, =]\}$

$\mathcal{I}_1 = read(\mathcal{I}_0, S) = \{[S' \rightarrow S, \#]\}$

$\mathcal{I}_2 = read(\mathcal{I}_0, U) = \{[S \rightarrow U, \#]\}$

$\mathcal{I}_3 = read(\mathcal{I}_0, E) = \{[S \rightarrow E, \#]\}$

$\mathcal{I}_4 = read(\mathcal{I}_0, a) = \{[U \rightarrow a, \#], [V \rightarrow a, =]\}$  Nincs konfliktus!

$\mathcal{I}_5 = read(\mathcal{I}_0, V) = \{[E \rightarrow V = V, \#]\}$

## Példa

## Példa grammatika

$$S' \rightarrow S \quad S \rightarrow U \mid E \quad U \rightarrow a \quad E \rightarrow V = V \quad V \rightarrow a$$

$\mathcal{I}_0 = closure([S' \rightarrow .S, \#]) =$   
 $= \{[S' \rightarrow .S, \#], [S \rightarrow .U, \#], [S \rightarrow .E, \#], [U \rightarrow .a, \#],$   
 $[E \rightarrow .V = V, \#], [V \rightarrow .a, =]\}$

$\mathcal{I}_1 = read(\mathcal{I}_0, S) = \{[S' \rightarrow S, \#]\}$

$\mathcal{I}_2 = read(\mathcal{I}_0, U) = \{[S \rightarrow U, \#]\}$

$\mathcal{I}_3 = read(\mathcal{I}_0, E) = \{[S \rightarrow E, \#]\}$

$\mathcal{I}_4 = read(\mathcal{I}_0, a) = \{[U \rightarrow a, \#], [V \rightarrow a, =]\}$  Nincs konfliktus!

$\mathcal{I}_5 = read(\mathcal{I}_0, V) = \{[E \rightarrow V = V, \#]\}$

$\mathcal{I}_6 = read(\mathcal{I}_5, =) = \{[E \rightarrow V = .V, \#], [V \rightarrow .a, =]\}$

## Példa

## Példa grammatika

$$S' \rightarrow S \quad S \rightarrow U \mid E \quad U \rightarrow a \quad E \rightarrow V = V \quad V \rightarrow a$$

$\mathcal{I}_0 = closure([S' \rightarrow .S, \#]) =$   
 $= \{[S' \rightarrow .S, \#], [S \rightarrow .U, \#], [S \rightarrow .E, \#], [U \rightarrow .a, \#],$   
 $[E \rightarrow .V = V, \#], [V \rightarrow .a, =]\}$

$\mathcal{I}_1 = read(\mathcal{I}_0, S) = \{[S' \rightarrow S, \#]\}$

$\mathcal{I}_2 = read(\mathcal{I}_0, U) = \{[S \rightarrow U, \#]\}$

$\mathcal{I}_3 = read(\mathcal{I}_0, E) = \{[S \rightarrow E, \#]\}$

$\mathcal{I}_4 = read(\mathcal{I}_0, a) = \{[U \rightarrow a, \#], [V \rightarrow a, =]\}$  Nincs konfliktus!

$\mathcal{I}_5 = read(\mathcal{I}_0, V) = \{[E \rightarrow V = V, \#]\}$

$\mathcal{I}_6 = read(\mathcal{I}_5, =) = \{[E \rightarrow V = .V, \#], [V \rightarrow .a, =]\}$

$\mathcal{I}_7 = read(\mathcal{I}_6, V) = \{[E \rightarrow V = V, \#]\}$

## Példa

## Példa grammatika

$$S' \rightarrow S \quad S \rightarrow U \mid E \quad U \rightarrow a \quad E \rightarrow V = V \quad V \rightarrow a$$

$\mathcal{I}_0 = closure([S' \rightarrow .S, \#]) =$   
 $= \{[S' \rightarrow .S, \#], [S \rightarrow .U, \#], [S \rightarrow .E, \#], [U \rightarrow .a, \#],$   
 $[E \rightarrow .V = V, \#], [V \rightarrow .a, =]\}$

$\mathcal{I}_1 = read(\mathcal{I}_0, S) = \{[S' \rightarrow S, \#]\}$

$\mathcal{I}_2 = read(\mathcal{I}_0, U) = \{[S \rightarrow U, \#]\}$

$\mathcal{I}_3 = read(\mathcal{I}_0, E) = \{[S \rightarrow E, \#]\}$

$\mathcal{I}_4 = read(\mathcal{I}_0, a) = \{[U \rightarrow a, \#], [V \rightarrow a, =]\}$  Nincs konfliktus!

$\mathcal{I}_5 = read(\mathcal{I}_0, V) = \{[E \rightarrow V = V, \#]\}$

$\mathcal{I}_6 = read(\mathcal{I}_5, =) = \{[E \rightarrow V = .V, \#], [V \rightarrow .a, =]\}$

$\mathcal{I}_7 = read(\mathcal{I}_6, V) = \{[E \rightarrow V = V, \#]\}$

$\mathcal{I}_8 = read(\mathcal{I}_6, a) = \{[V \rightarrow a, \#]\}$

## Az elemző táblázat kitöltése

$\mathcal{I}_0 = \{[S' \rightarrow .S, \#], [S \rightarrow .U, \#], [S \rightarrow .E, \#], [U \rightarrow .a, \#],$   
 $[E \rightarrow .V = V, \#], [V \rightarrow .a, =]\}$

$read(\mathcal{I}_0, S) = \mathcal{I}_1$

	x	=	#	S	U	E	V
0			1				
1							
2							
3							
4							
5							
6							
7							
8							

## Az elemző táblázat kitöltése

$\mathcal{I}_0 = \{[S' \rightarrow .S, \#], [S \rightarrow .U, \#], [S \rightarrow .E, \#], [U \rightarrow .a, \#],$   
 $[E \rightarrow .V = V, \#], [V \rightarrow .a, =]\}$

$read(\mathcal{I}_0, U) = \mathcal{I}_2$

	x	=	#	S	U	E	V
0				1	2		
1							
2							
3							
4							
5							
6							
7							
8							

## Az elemző táblázat kitöltése

 $\mathcal{I}_0 = \{[S' \rightarrow .S, \#], [S \rightarrow .U, \#], [S \rightarrow .E, \#], [U \rightarrow .a, \#], [E \rightarrow .V = V, \#], [V \rightarrow .a, =]\}$ 
 $\text{read}(\mathcal{I}_0, E) = \mathcal{I}_3$ 

	x	=	#	S	U	E	V
0				1	2	3	
1							
2							
3							
4							
5							
6							
7							
8							

## Az elemző táblázat kitöltése

 $\mathcal{I}_0 = \{[S' \rightarrow .S, \#], [S \rightarrow .U, \#], [S \rightarrow .E, \#], [U \rightarrow .a, \#], [E \rightarrow .V = V, \#], [V \rightarrow .a, =]\}$ 
 $\text{read}(\mathcal{I}_0, V) = \mathcal{I}_5$ 

	a	=	#	S	U	E	V
0	léptetés, 4			1	2	3	5
1							
2							
3							
4							
5							
6							
7							
8							

## Az elemző táblázat kitöltése

 $\mathcal{I}_2 = \{[S \rightarrow U., \#]\}$ 

	a	=	#	S	U	E	V
0	léptetés, 4			1	2	3	5
1		OK					
2		redukálás, $S \rightarrow U$					
3							
4							
5							
6							
7							
8							

## Az elemző táblázat kitöltése

 $\mathcal{I}_0 = \{[S' \rightarrow .S, \#], [S \rightarrow .U, \#], [S \rightarrow .E, \#], [U \rightarrow .a, \#], [E \rightarrow .V = V, \#], [V \rightarrow .a, =]\}$ 
 $\text{read}(\mathcal{I}_0, a) = \mathcal{I}_4$ 

	a	=	#	S	U	E	V
0	léptetés, 4			1	2	3	
1							
2							
3							
4							
5							
6							
7							
8							

## Az elemző táblázat kitöltése

 $\mathcal{I}_1 = \{[S' \rightarrow S., \#]\}$ 

	a	=	#	S	U	E	V
0	léptetés, 4			1	2	3	5
1		OK					
2							
3							
4							
5							
6							
7							
8							

## Az elemző táblázat kitöltése

 $\mathcal{I}_3 = \{[S \rightarrow E., \#]\}$ 

	a	=	#	S	U	E	V
0	léptetés, 4			1	2	3	5
1		OK					
2		redukálás, $S \rightarrow U$					
3		redukálás, $S \rightarrow E$					
4							
5							
6							
7							
8							

## Az elemző táblázat kitöltése

$$\mathcal{I}_4 = \{[U \rightarrow a., \#], [V \rightarrow a., =]\}$$

	a	=	#	S	U	E	V
0	léptetés, 4			1	2	3	5
1		OK					
2		redukálás, $S \rightarrow U$					
3		redukálás, $S \rightarrow E$					
4		redukálás, $U \rightarrow a$					
5							
6							
7							
8							

## Az elemző táblázat kitöltése

$$\mathcal{I}_4 = \{[U \rightarrow a., \#], [V \rightarrow a., =]\}$$

	a	=	#	S	U	E	V
0	léptetés, 4						1 2 3 5
1		OK					
2		redukálás, $S \rightarrow U$					
3		redukálás, $S \rightarrow E$					
4		redukálás, $V \rightarrow a$	redukálás, $U \rightarrow a$				
5							
6							
7							
8							

## Az elemző táblázat kitöltése

$$\mathcal{I}_5 = \{[E \rightarrow V. = V, \#]\}$$

$\text{read}(\mathcal{I}_5, =) = \mathcal{I}_6$

	a	=	#	S	U	E	V
0	léptetés, 4			1	2	3	5
1		OK					
2		redukálás, $S \rightarrow U$					
3		redukálás, $S \rightarrow E$					
4		redukálás, $V \rightarrow a$	redukálás, $U \rightarrow a$				
5		léptetés, 6					
6							
7							
8							

## Az elemző táblázat kitöltése

$$\mathcal{I}_6 = \{[E \rightarrow V = .V, \#], [V \rightarrow .a, \#]\}$$

$\text{read}(\mathcal{I}_6, V) = \mathcal{I}_7$

	a	=	#	S	U	E	V
0	léptetés, 4						1 2 3 5
1		OK					
2		redukálás, $S \rightarrow U$					
3		redukálás, $S \rightarrow E$					
4		redukálás, $V \rightarrow a$	redukálás, $U \rightarrow a$				
5		léptetés, 6					
6							7
7							
8							

## Az elemző táblázat kitöltése

$$\mathcal{I}_6 = \{[E \rightarrow V = .V, \#], [V \rightarrow .a, \#]\}$$

$\text{read}(\mathcal{I}_6, a) = \mathcal{I}_8$

	a	=	#	S	U	E	V
0	léptetés, 4			1	2	3	5
1		OK					
2		redukálás, $S \rightarrow U$					
3		redukálás, $S \rightarrow E$					
4		redukálás, $V \rightarrow a$	redukálás, $U \rightarrow a$				
5		léptetés, 6					
6	léptetés, 8						7
7							
8							

## Az elemző táblázat kitöltése

$$\mathcal{I}_7 = \{[E \rightarrow V = V, \#]\}$$

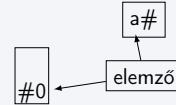
	a	=	#	S	U	E	V
0	léptetés, 4						1 2 3 5
1		OK					
2		redukálás, $S \rightarrow U$					
3		redukálás, $S \rightarrow E$					
4		redukálás, $V \rightarrow a$	redukálás, $U \rightarrow a$				
5		léptetés, 6					
6	léptetés, 8						7
7							
8							

## Az elemző táblázat kitöltése

$$\mathcal{I}_7 = \{[V \rightarrow a, \#]\}$$

	a	=	#	S	U	E	V
0	léptetés, 4			1	2	3	5
1		OK					
2		redukálás, $S \rightarrow U$					
3		redukálás, $S \rightarrow E$					
4		redukálás, $V \rightarrow a$	redukálás, $U \rightarrow a$				
5		léptetés, 6					
6	léptetés, 8						7
7		redukálás, $E \rightarrow V = V$					
8		redukálás, $V \rightarrow a$					

	a	=	#	S	U	E	V
0	léptetés, 4						1 2 3 5
1		OK					
2		redukálás, $S \rightarrow U$					
3		redukálás, $S \rightarrow E$					
4		redukálás, $V \rightarrow a$	redukálás, $U \rightarrow a$				
5		léptetés, 6					
6	léptetés, 8						7
7		redukálás, $E \rightarrow V = V$					
8		redukálás, $V \rightarrow a$					



## Az a# szöveg elemzése

	a	=	#	S	U	E	V
0	léptetés, 4			1	2	3	5
1		OK					
2		redukálás, $S \rightarrow U$					
3		redukálás, $S \rightarrow E$					
4		redukálás, $V \rightarrow a$	redukálás, $U \rightarrow a$				
5		léptetés, 6					
6	léptetés, 8						7
7		redukálás, $E \rightarrow V = V$					
8		redukálás, $V \rightarrow a$					



## Az a# szöveg elemzése

	a	=	#	S	U	E	V
0	léptetés, 4						1 2 3 5
1		OK					
2		redukálás, $S \rightarrow U$					
3		redukálás, $S \rightarrow E$					
4		redukálás, $V \rightarrow a$	redukálás, $U \rightarrow a$				
5		léptetés, 6					
6	léptetés, 8						7
7		redukálás, $E \rightarrow V = V$					
8		redukálás, $V \rightarrow a$					



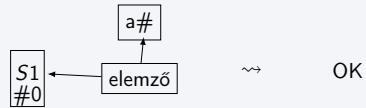
## Az a# szöveg elemzése

	a	=	#	S	U	E	V
0	léptetés, 4			1	2	3	5
1		OK					
2		redukálás, $S \rightarrow U$					
3		redukálás, $S \rightarrow E$					
4		redukálás, $V \rightarrow a$	redukálás, $U \rightarrow a$				
5		léptetés, 6					
6	léptetés, 8						7
7		redukálás, $E \rightarrow V = V$					
8		redukálás, $V \rightarrow a$					



## Az a# szöveg elemzése

	a	=	#	S	U	E	V
0	léptetés, 4						1 2 3 5
1		OK					
2		redukálás, $S \rightarrow U$					
3		redukálás, $S \rightarrow E$					
4		redukálás, $V \rightarrow a$	redukálás, $U \rightarrow a$				
5		léptetés, 6					
6	léptetés, 8						7
7		redukálás, $E \rightarrow V = V$					
8		redukálás, $V \rightarrow a$					



Az  $LR(1)$  elemző táblázat konfliktusmentessége

## Tétel

Egy grammatika pontosan akkor  $LR(1)$  grammatika, ha az  $LR(1)$  elemző táblázatai konfliktusmentesek.

Az  $LR(1)$  elemző létrehozása véges

- az  $LR(1)$ -elemek száma véges
- a closure és read függvények kiszámítása véges
- a kanonikus halmazok száma véges
- a kanonikus halmazok kiszámítása véges

Az  $LR(1)$  elemző a megadott módon véges sok lépében és teljesen automatikusan létrehozható.

Járható prefix, érvényes  $LR$ -elem

- járható prefix: a mondatformának olyan prefixei, amelyek legfeljebb a nyél végéig tartanak
  - ezeket járja be az elemző automata
  - a maximális járható prefixnek a végén ott a teljes nyél

Járható prefix, érvényes  $LR$ -elem

- járható prefix: a mondatformának olyan prefixei, amelyek legfeljebb a nyél végéig tartanak
  - ezeket járja be az elemző automata
  - a maximális járható prefixnek a végén ott a teljes nyél
- járható prefixre érvényes  $LR$ -elemek: a járható prefix „lehetséges folytatásai”
  - melyik szabályok építésében hol tarthatunk egy adott járható prefix elemzése után?

Járható prefix, érvényes  $LR$ -elem

- járható prefix: a mondatformának olyan prefixei, amelyek legfeljebb a nyél végéig tartanak
  - ezeket járja be az elemző automata
  - a maximális járható prefixnek a végén ott a teljes nyél
- járható prefixre érvényes  $LR$ -elemek: a járható prefix „lehetséges folytatásai”
  - melyik szabályok építésében hol tarthatunk egy adott járható prefix elemzése után?

Definíció: Járható prefixre érvényes  $LR(1)$ -elem

A grammatika egy  $[A \rightarrow \alpha.\beta, a]$   $LR(1)$ -eleme érvényes a  $\gamma\alpha$  járható prefixre nézve, ha

$$S' \Rightarrow^* \gamma A x \Rightarrow \gamma \alpha \beta x,$$

és  $x \neq \epsilon$  esetén a az  $x$  első szimbóluma,  $x = \epsilon$  esetén pedig  $a = \#$ .

Az  $LR(1)$ -elemzés nagy tétele

Egy  $\gamma$  járható prefix hatására az elemző automatája a kezdőállapotból olyan állapotba kerül, amelyhez tartozó kanonikus halmaz éppen a  $\gamma$  járható prefixre érvényes  $LR(1)$  elemeket tartalmazza.

## Az $LR(1)$ elemzés helyessége

### Az $LR(1)$ -elemzés nagy tétele

Egy  $\gamma$  járható prefix hatására az elemző automatája a kezdőállapotból olyan állapotba kerül, amelyhez tartozó kanonikus halmaz éppen a  $\gamma$  járható prefixre érvényes  $LR(1)$  elemeket tartalmazza.

- Összefoglalva:

- az  $LR(1)$  elemző automatikusan és véges lépésekben generálható a nyelvtan szabályaiból
- minden  $LR(1)$  grammatika elemezéséhez használható
- az elemző minden lépést írja elő a fenti tételek miatt

- Probléma:

- túl sok állapota van...

## LR elemzések (LALR(1) elemzés)

Fordítóprogramok előadás (A,C,T)

1

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Emlékeztető

- $LR(0)$  elemzés

- léptetés:  $[S \rightarrow (S.)S]$
- redukálás:  $\{[S \rightarrow (S)S]\}$
- konfliktus:  $\{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\}$

## Emlékeztető

- $LR(0)$  elemzés

- léptetés:  $[S \rightarrow (S.)S]$
- redukálás:  $\{[S \rightarrow (S)S]\}$
- konfliktus:  $\{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\}$

- $SLR(1)$  elemzés

Az  $\{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\}$  állapotban:

- léptetés: ( hatására
- redukálás: ) vagy # hatására, mert ezek elemei  $FOLLOW_1(S)$ -nek

## Emlékeztető

- $LR(0)$  elemzés

- léptetés:  $[S \rightarrow (S.)S]$
- redukálás:  $\{[S \rightarrow (S)S]\}$
- konfliktus:  $\{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\}$

- $SLR(1)$  elemzés

Az  $\{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\}$  állapotban:

- léptetés: ( hatására
- redukálás: ) vagy # hatására, mert ezek elemei  $FOLLOW_1(S)$ -nek

- $LR(1)$  elemzés

Az  $\{[S \rightarrow (.S)S, \#], [S \rightarrow ..], [S \rightarrow .(S)S, \#]\}$  állapotban:

- léptetés: ( hatására
- redukálás: ) hatására

2

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

2

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## $SLR(1)$ és $LR(1)$ állapotok száma

### $SLR(1)$ ( $LR(0)$ ) kanonikus halmazok

- $$\begin{aligned} I_0 &= closure([S' \rightarrow .S]) = \{[S' \rightarrow .S], [S \rightarrow .], [S \rightarrow .(S)S]\} \\ I_1 &= read(I_0, S) = \{[S' \rightarrow S.\}\} \\ I_2 &= read(I_0, ()) = \{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\} \\ I_3 &= read(I_2, S) = \{[S \rightarrow (S)S]\} \\ \text{read}(I_2, ()) &= \{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\} = I_2 \\ I_4 &= read(I_3, ()) = \{[S \rightarrow (S).S], [S \rightarrow ..], [S \rightarrow .(S)S]\} \\ I_5 &= read(I_4, S) = \{[S \rightarrow (S)S]\} \end{aligned}$$

## $SLR(1)$ és $LR(1)$ állapotok száma

### $SLR(1)$ ( $LR(0)$ ) kanonikus halmazok

- $$\begin{aligned} I_0 &= closure([S' \rightarrow .S]) = \{[S' \rightarrow .S], [S \rightarrow ..\#], [S \rightarrow .(S)S, \#\}\} \\ I_1 &= read(I_0, S) = \{[S' \rightarrow S.\#\}\} \\ I_2 &= read(I_0, ()) = \{[S \rightarrow (.S)S, \#\}, [S \rightarrow ..\#], [S \rightarrow .(S)S, \#\}\} \\ I_3 &= read(I_2, S) = \{[S \rightarrow (S)S, \#\}\} \\ \text{read}(I_2, ()) &= \{[S \rightarrow (.S)S, \#\}, [S \rightarrow ..\#], [S \rightarrow .(S)S, \#\}\} = I_2 \\ I_4 &= read(I_3, S) = \{[S \rightarrow (S)S]\} \end{aligned}$$

### $LR(1)$ kanonikus halmazok

- $$\begin{aligned} I_0 &= closure([S' \rightarrow .S, \#\}) = \{[S' \rightarrow .S, \#\}, [S \rightarrow ..\#], [S \rightarrow .(S)S, \#\}\} \\ I_1 &= read(I_0, S) = \{[S' \rightarrow S.\#\}\} \\ I_2 &= read(I_0, ()) = \{[S \rightarrow (.S)S, \#\}, [S \rightarrow ..\#], [S \rightarrow .(S)S, \#\}\} \\ I_3 &= read(I_2, S) = \{[S \rightarrow (S)S, \#\}\} \\ \text{read}(I_2, ()) &= \{[S \rightarrow (.S)S, \#\}, [S \rightarrow ..\#], [S \rightarrow .(S)S, \#\}\} = I_2 \\ \dots & \end{aligned}$$

Az előreolvasási szimbólumok miatt nő az állapotok száma!

3

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

4

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## SLR(1) és LR(1) állapotok száma

### LR(1) kanonikus halmazok

$\mathcal{I}_0 = closure([S' \rightarrow .S, \#]) = \{[S' \rightarrow .S, \#], [S \rightarrow .., \#], [S \rightarrow .(S)S, \#]\}$   
 $\mathcal{I}_1 = read(\mathcal{I}_0, S) = \{[S' \rightarrow S., \#]\}$   
 $\mathcal{I}_2 = read(\mathcal{I}_0, ()) = \{[S \rightarrow .(S)S, \#], [S \rightarrow .., ], [S \rightarrow .(S)S, )]\}$   
 $\mathcal{I}_3 = read(\mathcal{I}_2, S) = \{[S \rightarrow .(S)S, \#]\}$   
 $\mathcal{I}_4 = read(\mathcal{I}_2, ()) = \{[S \rightarrow .(S)S, )], [S \rightarrow .., ], [S \rightarrow .(S)S, )]\}$   
 $\mathcal{I}_5 = read(\mathcal{I}_3, ()) = \{[S \rightarrow (S).S, \#], [S \rightarrow .., \#], [S \rightarrow .(S)S, \#]\}$   
 $\mathcal{I}_6 = read(\mathcal{I}_4, S) = \{[S \rightarrow (S).S, ]\}$   
 $read(\mathcal{I}_4, ()) = \{[S \rightarrow .(S)S, )], [S \rightarrow .., ], [S \rightarrow .(S)S, )]\} = \mathcal{I}_4$   
 $\mathcal{I}_7 = read(\mathcal{I}_5, S) = \{[S \rightarrow (S).S, \#]\}$   
 $read(\mathcal{I}_5, ()) = \{[S \rightarrow .(S)S, \#], [S \rightarrow .., ], [S \rightarrow .(S)S, )]\} = \mathcal{I}_2$   
 $\mathcal{I}_8 = read(\mathcal{I}_6, ()) = \{[S \rightarrow (S).S, )], [S \rightarrow .., ], [S \rightarrow .(S)S, )]\}$   
 $\mathcal{I}_9 = read(\mathcal{I}_8, S) = \{[S \rightarrow (S).S, ]\}$   
 $read(\mathcal{I}_8, ()) = \{[S \rightarrow .(S)S, )], [S \rightarrow .., ], [S \rightarrow .(S)S, )]\} = \mathcal{I}_4$

- SLR(1) (és LR(0)): 6 állapot; LR(1): 10 állapot
- Valódi programnyelveknél:  
SLR(1): néhány száz; LR(1): néhány ezer

5

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Egyesíthető LR(1) kanonikus halmazok

Egyes kanonikus halmaz párok csak az előreolvasási szimbólumokban különböznek.

### LR(1) kanonikus halmazok

$\mathcal{I}_0 = closure([S' \rightarrow .S, \#]) = \{[S' \rightarrow .S, \#], [S \rightarrow .., \#], [S \rightarrow .(S)S, \#]\}$   
 $\mathcal{I}_1 = read(\mathcal{I}_0, S) = \{[S' \rightarrow S., \#]\}$   
 $\mathcal{I}_2 = read(\mathcal{I}_0, ()) = \{[S \rightarrow .(S)S, \#], [S \rightarrow .., ], [S \rightarrow .(S)S, )]\}$   
 $\mathcal{I}_3 = read(\mathcal{I}_2, S) = \{[S \rightarrow .(S)S, \#]\}$   
 $\mathcal{I}_4 = read(\mathcal{I}_2, ()) = \{[S \rightarrow .(S)S, ), [S \rightarrow .., ], [S \rightarrow .(S)S, )]\}$   
 $\mathcal{I}_5 = read(\mathcal{I}_3, ()) = \{[S \rightarrow (S).S, \#], [S \rightarrow .., \#], [S \rightarrow .(S)S, \#]\}$   
 $\mathcal{I}_6 = read(\mathcal{I}_4, S) = \{[S \rightarrow (S).S, ]\}$   
 $\mathcal{I}_7 = read(\mathcal{I}_5, S) = \{[S \rightarrow (S).S, \#]\}$   
 $\mathcal{I}_8 = read(\mathcal{I}_6, ()) = \{[S \rightarrow (S).S, ), [S \rightarrow .., ], [S \rightarrow .(S)S, )]\}$   
 $\mathcal{I}_9 = read(\mathcal{I}_8, S) = \{[S \rightarrow (S).S, ]\}$

Ezeket a halmazokat **egyesíthetőnek** nevezük.

6

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## LALR(1) kanonikus halmazok

Az egyesíthető kanonikus halmazokat vonjuk össze!

### LALR(1) kanonikus halmazok

$\mathcal{K}_0 = \mathcal{I}_0 = \{[S' \rightarrow .S, \#], [S \rightarrow .., \#], [S \rightarrow .(S)S, \#]\}$   
 $\mathcal{K}_1 = \mathcal{I}_1 = \{[S' \rightarrow S., \#]\}$   
 $\mathcal{K}_2 = \mathcal{I}_2 \cup \mathcal{I}_4 = \{[S \rightarrow .(S)S, \#], [S \rightarrow .., ], [S \rightarrow .(S)S, )], [S \rightarrow .(S)S, ]\}$   
 $\mathcal{K}_3 = \mathcal{I}_3 \cup \mathcal{I}_6 = \{[S \rightarrow (S).S, \#], [S \rightarrow (S).S, ]\}$   
 $\mathcal{K}_4 = \mathcal{I}_5 \cup \mathcal{I}_8 = \{[S \rightarrow (S).S, \#], [S \rightarrow .., \#], [S \rightarrow .(S)S, \#], [S \rightarrow (S).S, )], [S \rightarrow .., ], [S \rightarrow .(S)S, )]\}$   
 $\mathcal{K}_5 = \mathcal{I}_7 \cup \mathcal{I}_9 = \{[S \rightarrow (S).S, \#], [S \rightarrow (S).S, ]\}$

Az egyesített kanonikus halmazokat **LALR(1) kanonikus halmazoknak** nevezünk.

7

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## LALR(1) elemzés

- Az LALR(1) elemzőnek ugyanannyi állapota van, mint az SLR(1) (és LR(0)) emelzőknek.
- Mivel megjelennek benne az előreolvasási szimbólumok, több nyelvtan lesz elemezhető vele, mint SLR(1) módszerrel.
- De nem minden LR(1) grammátika esetén használható!
- Név: LookAhead LR  
(Előreolvasó LR)

## Lehetséges problémák az egyesítések miatt

- Előfordulhat-e léptetés-léptetés konfliktus (azaz, hogy két különböző állapotba kellene lépni ugyanazon szimbólum hatására)?
  - Ha  $\mathcal{I}_m$  és  $\mathcal{I}_n$  halmazok egyesíthetők, akkor  $read(\mathcal{I}_m, X)$  és  $read(\mathcal{I}_n, X)$  is egyesíthető.
  - Léptetés-léptetés konfliktus nem fordulhat elő!

9

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Lehetséges problémák az egyesítések miatt

- Előfordulhat-e léptetés-léptetés konfliktus (azaz, hogy két különböző állapotba kellene lépni ugyanazon szimbólum hatására)?
  - Ha  $\mathcal{I}_m$  és  $\mathcal{I}_n$  halmazok egyesíthetők, akkor  $read(\mathcal{I}_m, X)$  és  $read(\mathcal{I}_n, X)$  is egyesíthető.
  - Léptetés-léptetés konfliktus nem fordulhat elő!
- Előfordulhat-e léptetés-redukálás konfliktus?
  - Tegyük fel, hogy  $[A \rightarrow \alpha.a\beta, b]$  és  $[B \rightarrow \gamma.., a]$  elemei egy egységesített halmaznak.
  - Nézzük meg azt az LR(1) kanonikus halmazt, amelyikből  $[B \rightarrow \gamma.., a]$ -t kaptuk. Ebben benne kell lennie egy  $A \rightarrow \alpha.a\beta$  magú elemnek is, csak esetleg más előreolvasási szimbólummal.
  - Viszont ha  $[A \rightarrow \alpha.a\beta, b']$  és  $[B \rightarrow \gamma.., a]$  elemei egy LR(1) kanonikus halmaznak, akkor már itt léptetés-redukálás konfliktus van, azaz nem LR(1)-es a grammáti!
  - Léptetés-redukálás konfliktus sem fordulhat elő!

9

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Lehetséges problémák az egyesítések miatt

- Redukálás-redukálás konfliktus: előfordulhat!

Példa  $LR(1)$ , de nem  $LALR(1)$  grammatikára

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow aAd \mid bBd \mid aBe \mid bAe \\ A &\rightarrow c \\ B &\rightarrow c \end{aligned}$$

- $ac$  járható prefixre érvényes elemek:  $\{[A \rightarrow c., d], [B \rightarrow c., e]\}$
- $bc$  járható prefixre érvényes elemek:  $\{[A \rightarrow c., e], [B \rightarrow c., d]\}$

10

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Lehetséges problémák az egyesítések miatt

- Redukálás-redukálás konfliktus: előfordulhat!

Példa  $LR(1)$ , de nem  $LALR(1)$  grammatikára

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow aAd \mid bBd \mid aBe \mid bAe \\ A &\rightarrow c \\ B &\rightarrow c \end{aligned}$$

- $ac$  járható prefixre érvényes elemek:  $\{[A \rightarrow c., d], [B \rightarrow c., e]\}$
- $bc$  járható prefixre érvényes elemek:  $\{[A \rightarrow c., e], [B \rightarrow c., d]\}$
- az egyesítés után:
  - $\{[A \rightarrow c., d], [B \rightarrow c., e], [A \rightarrow c., e], [B \rightarrow c., d]\}$
  - redukálás-redukálás konfliktus!

10

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## $LALR(1)$ elemzés, $LALR(1)$ grammatika

- az  $LALR(1)$  elemző táblázat kitöltése megegyezik az  $LR(1)$  táblázat kitöltésével

11

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## $LALR(1)$ elemzés, $LALR(1)$ grammatika

- az  $LALR(1)$  elemző táblázat kitöltése megegyezik az  $LR(1)$  táblázat kitöltésével
- az elemzés menete is azonos

11

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## $LALR(1)$ elemzés, $LALR(1)$ grammatika

- az  $LALR(1)$  elemző táblázat kitöltése megegyezik az  $LR(1)$  táblázat kitöltésével
- az elemzés menete is azonos

Definíció:  $LALR(1)$  grammatika

Egy grammatika  $LALR(1)$  grammatika, ha az  $LALR(1)$  elemző táblázata konfliktusmentesen kitölthető.

A programnyelvek többsége leírható  $LALR(1)$  nyelvtannal.

11

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## $LALR(1)$ elemző táblázat

$$\begin{aligned} \mathcal{K}_0 &= \{[S' \rightarrow .S, \#], [S \rightarrow .., \#], [S \rightarrow .(S)S, \#]\} \\ \mathcal{K}_1 &= \{[S' \rightarrow S., \#]\} \\ \mathcal{K}_2 &= \{[S \rightarrow (.S)S, \#], [S \rightarrow .., ]\}, [S \rightarrow .(S)S, ]], [S \rightarrow .(S)S, )]\} \\ \mathcal{K}_3 &= \{[S \rightarrow (S).S, \#], [S \rightarrow .., ]\}, [S \rightarrow .(S)S, )]\} \\ \mathcal{K}_4 &= \{[S \rightarrow (S).S, \#], [S \rightarrow .., ], [S \rightarrow .(S)S, \#], \\ &\quad [S \rightarrow (S).S, )], [S \rightarrow .., ), [S \rightarrow .(S)S, )]\} \\ \mathcal{K}_5 &= \{[S \rightarrow (S)S, \#], [S \rightarrow (S).S, )]\} \end{aligned}$$

	(	)	#	S
0	léptetés, 2		redukálás, $S \rightarrow \epsilon$	1
1			OK	
2	léptetés, 2	redukálás, $S \rightarrow \epsilon$		3
3		léptetés, 4		
4	léptetés, 2	redukálás, $S \rightarrow \epsilon$	redukálás, $S \rightarrow \epsilon$	5
5		redukálás, $S \rightarrow (S)S$	redukálás, $S \rightarrow (S)S$	

12

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## SLR(1) vs. LALR(1)

SLR(1)	(	)	#	S
0	léptetés, 2	redukálás, $S \rightarrow \epsilon$	redukálás, $S \rightarrow \epsilon$	1
1			OK	
2	léptetés, 2	redukálás, $S \rightarrow \epsilon$	redukálás, $S \rightarrow \epsilon$	3
3		léptetés, 4		
4	léptetés, 2	redukálás, $S \rightarrow \epsilon$	redukálás, $S \rightarrow \epsilon$	5
5		redukálás, $S \rightarrow (S)S$	redukálás, $S \rightarrow (S)S$	

LALR(1)	(	)	#	S
0	léptetés, 2		redukálás, $S \rightarrow \epsilon$	1
1			OK	
2	léptetés, 2	redukálás, $S \rightarrow \epsilon$		3
3		léptetés, 4		
4	léptetés, 2	redukálás, $S \rightarrow \epsilon$	redukálás, $S \rightarrow \epsilon$	5
5		redukálás, $S \rightarrow (S)S$	redukálás, $S \rightarrow (S)S$	

13

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## LALR(1) elemző hatékonyabb generálása

- nagyon időigényes létrehozni az összes LR(1) kanonikus halmazt és megkeresni közöttük az összevonhatókat

## LALR(1) elemző hatékonyabb generálása

- nagyon időigényes létrehozni az összes LR(1) kanonikus halmazt és megkeresni közöttük az összevonhatókat
- egyszerűbb módszer:
  - az LR(0) kanonikus halmazokból indulunk (de azoknak is csak a lényeges elemeit fogjuk tárolni)
  - az előreolvasási szimbólumokat utólag határozzuk meg

14

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## A kanonikus halmazok törzse

### Definíció: LR(0) kanonikus halmazok törzse

Egy LR(0) kanonikus halmaz törzse azokat az elmeket tartalmazza, amelyek nem a lezáras művelet hatására kerültek bele a halmazba. Az  $\mathcal{I}_0$  halmaz esetén ez a  $[S' \rightarrow .S]$  elemet, a többi halmaz esetén pedig azokat az elmeket jelenti, amelyekben a pont nem a szabály jobboldalának elején áll.

## A kanonikus halmazok törzse

### Definíció: LR(0) kanonikus halmazok törzse

Egy LR(0) kanonikus halmaz törzse azokat az elmeket tartalmazza, amelyek nem a lezáras művelet hatására kerültek bele a halmazba. Az  $\mathcal{I}_0$  halmaz esetén ez a  $[S' \rightarrow .S]$  elemet, a többi halmaz esetén pedig azokat az elmeket jelenti, amelyekben a pont nem a szabály jobboldalának elején áll.

### Példa: helyes zárójelzés

$$\begin{aligned}\mathcal{I}_0 &= \{[S' \rightarrow .S], [S \rightarrow .], [S \rightarrow .(S)S]\} \\ \mathcal{I}_1 &= \{[S' \rightarrow S.\] \\ \mathcal{I}_2 &= \{[S \rightarrow (.S)S], [S \rightarrow .], [S \rightarrow .(S)S]\} \\ \mathcal{I}_3 &= \{[S \rightarrow (S.)S]\} \\ \mathcal{I}_4 &= \{[S \rightarrow (S).S], [S \rightarrow .], [S \rightarrow .(S)S]\} \\ \mathcal{I}_5 &= \{[S \rightarrow (S)S.\]\end{aligned}$$

15

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## A kanonikus halmazok törzse

- Jelölések:

- $\mathcal{I}_j^*$ : az  $\mathcal{I}_j$  kanonikus halmaz törzse
- $read^*(\mathcal{I}_j^*, X)$ : a  $read(\mathcal{I}_j, X)$  törzse

### Példa: helyes zárójelzés

$$\begin{aligned}\mathcal{I}_0^* &= \{[S' \rightarrow .S]\} \\ \mathcal{I}_1^* &= \{[S' \rightarrow S.\] \\ \mathcal{I}_2^* &= \{[S \rightarrow (.S)S]\} \\ \mathcal{I}_3^* &= \{[S \rightarrow (S.)S]\} \\ \mathcal{I}_4^* &= \{[S \rightarrow (S).S]\} \\ \mathcal{I}_5^* &= \{[S \rightarrow (S)S.\]\end{aligned}$$

16

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

- $\mathcal{I}_0^* = \{[S' \rightarrow .S]\}$
- ha  $[A \rightarrow \alpha.X\beta] \in closure(\mathcal{I}_j^*)$ , akkor  
 $[A \rightarrow \alpha X.\beta] \in read^*(\mathcal{I}_j^*, X)$

- $\mathcal{I}_0^* = \{[S' \rightarrow .S]\}$
- ha  $[A \rightarrow \alpha.X\beta] \in closure(\mathcal{I}_j^*)$ , akkor  
 $[A \rightarrow \alpha X.\beta] \in read^*(\mathcal{I}_j^*, X)$

**Példa**

Mivel  $[S \rightarrow (.S)S] \in \mathcal{I}_2^*$  és  $[S \rightarrow .(S)S] \in closure([S \rightarrow (.S)S])$ , ezért  $[S \rightarrow (.S)S] \in read(\mathcal{I}_2^*, ())$ .

$$\mathcal{I}_0^* = \{[S' \rightarrow .S]\}$$

$$\begin{aligned}\mathcal{I}_0^* &= \{[S' \rightarrow .S]\} \\ \mathcal{I}_1^* &= read^*(\mathcal{I}_0^*, S) = \{[S' \rightarrow S.] \}\end{aligned}$$

$$\begin{aligned}\mathcal{I}_0^* &= \{[S' \rightarrow .S]\} \\ \mathcal{I}_1^* &= read^*(\mathcal{I}_0^*, S) = \{[S' \rightarrow S.] \} \\ \mathcal{I}_2^* &= read^*(\mathcal{I}_0^*, ()) = \{[S \rightarrow (.S)S] \}\end{aligned}$$

$$\begin{aligned}\mathcal{I}_0^* &= \{[S' \rightarrow .S]\} \\ \mathcal{I}_1^* &= read^*(\mathcal{I}_0^*, S) = \{[S' \rightarrow S.] \} \\ \mathcal{I}_2^* &= read^*(\mathcal{I}_0^*, ()) = \{[S \rightarrow (.S)S] \} \\ \mathcal{I}_3^* &= read^*(\mathcal{I}_2^*, S) = \{[S \rightarrow (S.)S] \}\end{aligned}$$

## Példa

$$\begin{aligned}\mathcal{I}_0^* &= \{[S' \rightarrow .S]\} \\ \mathcal{I}_1^* &= \text{read}^*(\mathcal{I}_0^*, S) = \{[S' \rightarrow S.] \} \\ \mathcal{I}_2^* &= \text{read}^*(\mathcal{I}_0^*, ()) = \{[S \rightarrow (.S)S] \} \\ \mathcal{I}_3^* &= \text{read}^*(\mathcal{I}_2^*, S) = \{[S \rightarrow (S.)S] \} \\ \text{read}^*(\mathcal{I}_2^*, ()) &= \{[S \rightarrow (.S)S] \} = \mathcal{I}_2^*\end{aligned}$$

18

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

$$\begin{aligned}\mathcal{I}_0^* &= \{[S' \rightarrow .S]\} \\ \mathcal{I}_1^* &= \text{read}^*(\mathcal{I}_0^*, S) = \{[S' \rightarrow S.] \} \\ \mathcal{I}_2^* &= \text{read}^*(\mathcal{I}_0^*, ()) = \{[S \rightarrow (.S)S] \} \\ \mathcal{I}_3^* &= \text{read}^*(\mathcal{I}_2^*, S) = \{[S \rightarrow (S.)S] \} \\ \text{read}^*(\mathcal{I}_2^*, ()) &= \{[S \rightarrow (.S)S] \} = \mathcal{I}_2^* \\ \mathcal{I}_4^* &= \text{read}^*(\mathcal{I}_3^*, ()) = \{[S \rightarrow (S).S] \}\end{aligned}$$

18

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

$$\begin{aligned}\mathcal{I}_0^* &= \{[S' \rightarrow .S]\} \\ \mathcal{I}_1^* &= \text{read}^*(\mathcal{I}_0^*, S) = \{[S' \rightarrow S.] \} \\ \mathcal{I}_2^* &= \text{read}^*(\mathcal{I}_0^*, ()) = \{[S \rightarrow (.S)S] \} \\ \mathcal{I}_3^* &= \text{read}^*(\mathcal{I}_2^*, S) = \{[S \rightarrow (S.)S] \} \\ \text{read}^*(\mathcal{I}_2^*, ()) &= \{[S \rightarrow (.S)S] \} = \mathcal{I}_2^* \\ \mathcal{I}_4^* &= \text{read}^*(\mathcal{I}_3^*, ()) = \{[S \rightarrow (S).S] \} \\ \mathcal{I}_5^* &= \text{read}^*(\mathcal{I}_4^*, S) = \{[S \rightarrow (S)S.] \}\end{aligned}$$

18

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

$$\begin{aligned}\mathcal{I}_0^* &= \{[S' \rightarrow .S]\} \\ \mathcal{I}_1^* &= \text{read}^*(\mathcal{I}_0^*, S) = \{[S' \rightarrow S.] \} \\ \mathcal{I}_2^* &= \text{read}^*(\mathcal{I}_0^*, ()) = \{[S \rightarrow (.S)S] \} \\ \mathcal{I}_3^* &= \text{read}^*(\mathcal{I}_2^*, S) = \{[S \rightarrow (S.)S] \} \\ \text{read}^*(\mathcal{I}_2^*, ()) &= \{[S \rightarrow (.S)S] \} = \mathcal{I}_2^* \\ \mathcal{I}_4^* &= \text{read}^*(\mathcal{I}_3^*, ()) = \{[S \rightarrow (S).S] \} \\ \mathcal{I}_5^* &= \text{read}^*(\mathcal{I}_4^*, S) = \{[S \rightarrow (S)S.] \} \\ \text{read}^*(\mathcal{I}_4^*, ()) &= \{[S \rightarrow (.S)S] \} = \mathcal{I}_2^*\end{aligned}$$

18

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Örökolt és generált előreolvasási szimbólumok

- Mivel  $[S' \rightarrow .S, \#] \in \mathcal{I}_0$ , ezért  $[S' \rightarrow S., \#] \in \text{read}(\mathcal{I}_0, S)$ .
  - Ilyenkor a  $\#$  szimbólum öröklődik az  $[S' \rightarrow .S, \#]$  elemről az  $[S' \rightarrow S., \#]$  elemre.

19

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Örökolt és generált előreolvasási szimbólumok

- Mivel  $[S' \rightarrow .S, \#] \in \mathcal{I}_0$ , ezért  $[S' \rightarrow S., \#] \in \text{read}(\mathcal{I}_0, S)$ .
  - Ilyenkor a  $\#$  szimbólum öröklődik az  $[S' \rightarrow .S, \#]$  elemről az  $[S' \rightarrow S., \#]$  elemre.
- Mivel  $[S \rightarrow (.S)S, \#] \in \mathcal{I}_2$ , ezért  $[S \rightarrow (S.)S, ] \in \mathcal{I}_2$ , és így  $[S \rightarrow (S)S, ] \in \text{read}(\mathcal{I}_2, ())$ .
  - Ilyenkor a  $)$  szimbólum spontán generálható a  $[S \rightarrow (S)S, ]$  elemhez.

19

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Előreolvasási szimbólumok meghatározása

- Legyen  $\emptyset$  egy olyan szimbólum, ami nem szerepel a grammatikában!
- Ha  $[A \rightarrow \alpha.\beta] \in \mathcal{I}_j^*$ , akkor határozzuk meg  $closure([A \rightarrow \alpha.\beta, \emptyset])$  elemeit!

20

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Előreolvasási szimbólumok meghatározása

- Legyen  $\emptyset$  egy olyan szimbólum, ami nem szerepel a grammatikában!
- Ha  $[A \rightarrow \alpha.\beta] \in \mathcal{I}_j^*$ , akkor határozzuk meg  $closure([A \rightarrow \alpha.\beta, \emptyset])$  elemeit!
  - Ha  $[B \rightarrow \gamma.X\delta, a] \in closure([A \rightarrow \alpha.\beta, \emptyset])$  és  $a \neq \emptyset$ , akkor a  $read^*(\mathcal{I}_j^*, X)$  halmaz  $[B \rightarrow \gamma X.\delta]$  eleméhez az  $a$  spontán generálható.

20

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Előreolvasási szimbólumok meghatározása

- Legyen  $\emptyset$  egy olyan szimbólum, ami nem szerepel a grammatikában!
- Ha  $[A \rightarrow \alpha.\beta] \in \mathcal{I}_j^*$ , akkor határozzuk meg  $closure([A \rightarrow \alpha.\beta, \emptyset])$  elemeit!
  - Ha  $[B \rightarrow \gamma.X\delta, a] \in closure([A \rightarrow \alpha.\beta, \emptyset])$  és  $a \neq \emptyset$ , akkor a  $read^*(\mathcal{I}_j^*, X)$  halmaz  $[B \rightarrow \gamma X.\delta]$  eleméhez az  $a$  spontán generálható.
  - Ha  $[B \rightarrow \gamma.X\delta, \emptyset] \in closure([A \rightarrow \alpha.\beta, \emptyset])$ , akkor az előreolvasási szimbólumok öröklődnek az  $\mathcal{I}_j^*$  halmaz  $[A \rightarrow \alpha.\beta]$  elemétől a  $read^*(\mathcal{I}_j^*, X)$  halmaz  $[B \rightarrow \gamma X.\delta]$  eleméhez.

20

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

Törzs	Elem	Spontán generálható	Honnan örökölt?
$\mathcal{I}_0^*$	$[S' \rightarrow .S]$		
$\mathcal{I}_1^*$	$[S' \rightarrow S.]$		
$\mathcal{I}_2^*$	$[S \rightarrow (.S)S]$		
$\mathcal{I}_3^*$	$[S \rightarrow (S.)S]$		
$\mathcal{I}_4^*$	$[S \rightarrow (S).S]$		
$\mathcal{I}_5^*$	$[S \rightarrow (S)S.]$		

21

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

$$closure([S' \rightarrow .S, \emptyset]) = \\ = \{[S' \rightarrow .S, \emptyset], [S \rightarrow .., \emptyset], [S \rightarrow .(S)S, \emptyset]\}$$

Törzs	Elem	Spontán generálható	Honnan örökölt?
$\mathcal{I}_0^*$	$[S' \rightarrow .S]$		
$\mathcal{I}_1^*$	$[S' \rightarrow S.]$	$\mathcal{I}_0^*$ -beli $[S' \rightarrow .S]$	
$\mathcal{I}_2^*$	$[S \rightarrow (.S)S]$	$\mathcal{I}_0^*$ -beli $[S' \rightarrow .S]$	
$\mathcal{I}_3^*$	$[S \rightarrow (S.)S]$		
$\mathcal{I}_4^*$	$[S \rightarrow (S).S]$		
$\mathcal{I}_5^*$	$[S \rightarrow (S)S.]$		

22

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

$$closure([S \rightarrow (.S)S, \emptyset]) = \\ = \{[S \rightarrow (.S)S, \emptyset], [S \rightarrow ..], [S \rightarrow .(S)S, ]\}$$

Törzs	Elem	Spontán generálható	Honnan örökölt?
$\mathcal{I}_0^*$	$[S' \rightarrow .S]$		
$\mathcal{I}_1^*$	$[S' \rightarrow S.]$		$\mathcal{I}_0^*$ -beli $[S' \rightarrow .S]$
$\mathcal{I}_2^*$	$[S \rightarrow (.S)S]$		$\mathcal{I}_0^*$ -beli $[S' \rightarrow .S]$
$\mathcal{I}_3^*$	$[S \rightarrow (S.)S]$		$\mathcal{I}_2^*$ -beli $[S \rightarrow (.S)S]$
$\mathcal{I}_4^*$	$[S \rightarrow (S).S]$		
$\mathcal{I}_5^*$	$[S \rightarrow (S)S.]$		

23

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

$$\text{closure}([S \rightarrow (S.)S], \emptyset) = \\ = \{[S \rightarrow (S.)S, \emptyset]\}$$

Törzs	Elem	Spontán generálható	Honnan örökölt?
$\mathcal{I}_0^*$	$[S' \rightarrow .S]$		
$\mathcal{I}_1^*$	$[S' \rightarrow S.]$		$\mathcal{I}_0^*$ -beli $[S' \rightarrow .S]$
$\mathcal{I}_2^*$	$[S \rightarrow (.S)S]$	)	$\mathcal{I}_0^*$ -beli $[S' \rightarrow .S]$
$\mathcal{I}_3^*$	$[S \rightarrow (S.)S]$		$\mathcal{I}_2^*$ -beli $[S \rightarrow (.S)S]$
$\mathcal{I}_4^*$	$[S \rightarrow (S).S]$		$\mathcal{I}_3^*$ -beli $[S \rightarrow (S.)S]$
$\mathcal{I}_5^*$	$[S \rightarrow (S)S.]$		

24

Fordítóprogramok előadás (A,C,T) LR elemzések (LALR(1) elemzés)

## Példa

$$\text{closure}([S \rightarrow (S).S], \emptyset) = \\ = \{[S \rightarrow (S).S, \emptyset], [S \rightarrow ., \emptyset], [S \rightarrow .(S)S, \emptyset]\}$$

Törzs	Elem	Spontán generálható	Honnan örökölt?
$\mathcal{I}_0^*$	$[S' \rightarrow .S]$		
$\mathcal{I}_1^*$	$[S' \rightarrow S.]$		$\mathcal{I}_0^*$ -beli $[S' \rightarrow .S]$
$\mathcal{I}_2^*$	$[S \rightarrow (.S)S]$	)	$\mathcal{I}_0^*$ -beli $[S' \rightarrow .S]$ $\mathcal{I}_4^*$ -beli $[S \rightarrow (S).S]$
$\mathcal{I}_3^*$	$[S \rightarrow (S.)S]$		$\mathcal{I}_2^*$ -beli $[S \rightarrow (.S)S]$
$\mathcal{I}_4^*$	$[S \rightarrow (S).S]$		$\mathcal{I}_3^*$ -beli $[S \rightarrow (S.)S]$
$\mathcal{I}_5^*$	$[S \rightarrow (S)S.]$		$\mathcal{I}_4^*$ -beli $[S \rightarrow (S).S]$

25

Fordítóprogramok előadás (A,C,T) LR elemzések (LALR(1) elemzés)

## Az előreolvasási szimbólumok meghatározása

- 1. menet:
  - az  $[S' \rightarrow .S]$  elemhez felvesszük a # előreolvasási szimbólumot
  - minden elemhez felvesszük a spontán generálódó szimbólumait
- további menetek:
  - az öröklési szabályok szerint továbbterjesztjük a szimbólumokat

26

Fordítóprogramok előadás (A,C,T) LR elemzések (LALR(1) elemzés)

## Példa

Törzs	Elem	1. menet
$\mathcal{I}_0^*$	$[S' \rightarrow .S]$	#
$\mathcal{I}_1^*$	$[S' \rightarrow S.]$	
$\mathcal{I}_2^*$	$[S \rightarrow (.S)S]$	)
$\mathcal{I}_3^*$	$[S \rightarrow (S.)S]$	
$\mathcal{I}_4^*$	$[S \rightarrow (S).S]$	
$\mathcal{I}_5^*$	$[S \rightarrow (S)S.]$	

27

Fordítóprogramok előadás (A,C,T) LR elemzések (LALR(1) elemzés)

## Példa

Törzs	Elem	1. menet	2. menet
$\mathcal{I}_0^*$	$[S' \rightarrow .S]$	#	#
$\mathcal{I}_1^*$	$[S' \rightarrow S.]$		#
$\mathcal{I}_2^*$	$[S \rightarrow (.S)S]$	)	), #
$\mathcal{I}_3^*$	$[S \rightarrow (S.)S]$		)
$\mathcal{I}_4^*$	$[S \rightarrow (S).S]$		
$\mathcal{I}_5^*$	$[S \rightarrow (S)S.]$		

28

Fordítóprogramok előadás (A,C,T) LR elemzések (LALR(1) elemzés)

## Példa

Törzs	Elem	1. menet	2. menet	3. menet
$\mathcal{I}_0^*$	$[S' \rightarrow .S]$	#	#	#
$\mathcal{I}_1^*$	$[S' \rightarrow S.]$		#	#
$\mathcal{I}_2^*$	$[S \rightarrow (.S)S]$	)	), #	), #
$\mathcal{I}_3^*$	$[S \rightarrow (S.)S]$		)	), #
$\mathcal{I}_4^*$	$[S \rightarrow (S).S]$			)
$\mathcal{I}_5^*$	$[S \rightarrow (S)S.]$			

29

Fordítóprogramok előadás (A,C,T) LR elemzések (LALR(1) elemzés)

Törzs	Elem	Spontán generálható	Honnan örökölt?
$\mathcal{I}_0^*$	$[S' \rightarrow .S]$		
$\mathcal{I}_1^*$	$[S' \rightarrow S.]$		$\mathcal{I}_0^*$ -beli $[S' \rightarrow .S]$
$\mathcal{I}_2^*$	$[S \rightarrow (.S)S]$	)	$\mathcal{I}_0^*$ -beli $[S' \rightarrow .S]$ , $\mathcal{I}_4^*$ -beli $[S \rightarrow (S).S]$
$\mathcal{I}_3^*$	$[S \rightarrow (S.)S]$		$\mathcal{I}_2^*$ -beli $[S \rightarrow (.S)S]$
$\mathcal{I}_4^*$	$[S \rightarrow (S).S]$		$\mathcal{I}_3^*$ -beli $[S \rightarrow (S.)S]$
$\mathcal{I}_5^*$	$[S \rightarrow (S)S.]$		$\mathcal{I}_4^*$ -beli $[S \rightarrow (S).S]$

## Példa

Törzs	Elem	1. menet	2. menet	3. menet	4. menet
$\mathcal{I}_0^*$	$[S' \rightarrow .S]$	#	#	#	#
$\mathcal{I}_1^*$	$[S' \rightarrow S.]$		#	#	#
$\mathcal{I}_2^*$	$[S \rightarrow (.S)S]$	)	), #	), #	), #
$\mathcal{I}_3^*$	$[S \rightarrow (S.)S]$	)	), #	), #	
$\mathcal{I}_4^*$	$[S \rightarrow (S).S]$		)	), #	
$\mathcal{I}_5^*$	$[S \rightarrow (S)S.]$				)

30

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

Törzs	Elem	1. menet	2. menet	3. menet	4. menet	5. menet
$\mathcal{I}_0^*$	$[S' \rightarrow .S]$	#	#	#	#	#
$\mathcal{I}_1^*$	$[S' \rightarrow S.]$		#	#	#	#
$\mathcal{I}_2^*$	$[S \rightarrow (.S)S]$	)	), #	), #	), #	), #
$\mathcal{I}_3^*$	$[S \rightarrow (S.)S]$	)	), #	), #	), #	), #
$\mathcal{I}_4^*$	$[S \rightarrow (S).S]$		)	), #	), #	), #
$\mathcal{I}_5^*$	$[S \rightarrow (S)S.]$			)		), #

31

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

Törzs	Elem	Előreolvasási szimbólumok
$\mathcal{I}_0^*$	$[S' \rightarrow .S]$	#
$\mathcal{I}_1^*$	$[S' \rightarrow S.]$	#
$\mathcal{I}_2^*$	$[S \rightarrow (.S)S]$	), #
$\mathcal{I}_3^*$	$[S \rightarrow (S.)S]$	), #
$\mathcal{I}_4^*$	$[S \rightarrow (S).S]$	), #
$\mathcal{I}_5^*$	$[S \rightarrow (S)S.]$	), #

$$\mathcal{K}_0^* = \{[S' \rightarrow .S, \#]\}$$

$$\mathcal{K}_1^* = \{[S' \rightarrow S., \#]\}$$

$$\mathcal{K}_2^* = \{[S \rightarrow (.S)S, ], [S \rightarrow (.S)S, \#]\}$$

$$\mathcal{K}_3^* = \{[S \rightarrow (S.)S, ], [S \rightarrow (S.)S, \#]\}$$

$$\mathcal{K}_4^* = \{[S \rightarrow (S).S, ], [S \rightarrow (S).S, \#]\}$$

$$\mathcal{K}_5^* = \{[S \rightarrow (S)S, ][S \rightarrow (S)S, \#]\}$$

Így az egyesített kanonikus halmazok törzsét kaptuk meg!

32

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## LR(1) kanonikus halmazok törzse

Definíció: LR(1) kanonikus halmazok törzse

Egy (egyesített) LR(1) kanonikus halmaz törzse azokat az elmeket tartalmazza, amelyek nem a lezárást művelet hatására kerültek bele a halmazba.

Az  $\mathcal{I}_0$  halmaz esetén ez a  $[S' \rightarrow .S, \#]$  elemet, a többi halmaz esetén pedig azokat az elmeket jelenti, amelyekben a pont nem a szabály jobboldalának elején áll.

## LR(1) kanonikus halmazok törzse

Definíció: LR(1) kanonikus halmazok törzse

Egy (egyesített) LR(1) kanonikus halmaz törzse azokat az elmeket tartalmazza, amelyek nem a lezárást művelet hatására kerültek bele a halmazba.

Az  $\mathcal{I}_0$  halmaz esetén ez a  $[S' \rightarrow .S, \#]$  elemet, a többi halmaz esetén pedig azokat az elmeket jelenti, amelyekben a pont nem a szabály jobboldalának elején áll.

- Cél: az egyesített kanonikus halmazok törzsei segítségével kitölteni az LALR(1) elemző táblázatot

## Az elemző táblázat kitöltése

- $A \rightarrow \alpha$  redukció felismerése:

33

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

34

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Az elemző táblázat kitöltése

- $A \rightarrow \alpha$  redukció felismerése:

- ha  $\alpha \neq \epsilon$ , akkor  $[A \rightarrow \alpha, a]$  eleme a törzsnek  
 $\Rightarrow$  az a szimbólumhoz kell írni a redukciót

34

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Az elemző táblázat kitöltése

- $A \rightarrow \alpha$  redukció felismerése:

- ha  $\alpha \neq \epsilon$ , akkor  $[A \rightarrow \alpha, a]$  eleme a törzsnek  
 $\Rightarrow$  az a szimbólumhoz kell írni a redukciót
- ha  $\alpha = \epsilon$ , akkor az olyan a szimbólumokhoz tartozik a redukció, amelyekre
  - $[B \rightarrow \beta, C\gamma, b]$  eleme az adott törzsnek,
  - $C \Rightarrow^* A\delta$ , és
  - $a \in FIRST_1(\delta\gamma b)$ .

34

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Az elemző táblázat kitöltése

- $A \rightarrow \alpha$  redukció felismerése:

- ha  $\alpha \neq \epsilon$ , akkor  $[A \rightarrow \alpha, a]$  eleme a törzsnek  
 $\Rightarrow$  az a szimbólumhoz kell írni a redukciót
- ha  $\alpha = \epsilon$ , akkor az olyan a szimbólumokhoz tartozik a redukció, amelyekre
  - $[B \rightarrow \beta, C\gamma, b]$  eleme az adott törzsnek,
  - $C \Rightarrow^* A\delta$ , és
  - $a \in FIRST_1(\delta\gamma b)$ .
- az a szimbólumhoz a  $j$  állapotban léptetést kell előírni, ha
  - $read(\mathcal{K}_j^*, a) = \mathcal{K}_k^*$

34

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

	(	)	#	S
0				
1				
2				
3				
4				
5				

35

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

	(	)	#	S
0	léptetés, 2			
1				
2				
3				
4				
5				

$$read(\mathcal{K}_0^*, ()) = \mathcal{K}_2^*$$

36

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

	(	)	#	S
0	léptetés, 2		redukálás, S → ε	
1				
2				
3				
4				
5				

$$[S' \rightarrow .S, \#] = \mathcal{K}_2^* \text{ és } S \Rightarrow^* S \text{ és } \# \in FIRST_1(\#)$$

37

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

	(	)	#	S
0	léptetés, 2		redukálás, $S \rightarrow \epsilon$	1
1				
2				
3				
4				
5				

$$read(\mathcal{K}_0^*, S) = \mathcal{K}_1^*$$

## Példa

	(	)	#	S
0	léptetés, 2		redukálás, $S \rightarrow \epsilon$	1
1			OK	
2				
3				
4				
5				

$$[S' \rightarrow S., \#] \in \mathcal{K}_1^*$$

38

Fordítóprogramok előadás (A,C,T) LR elemzések (LALR(1) elemzés)

## Példa

	(	)	#	S
0	léptetés, 2		redukálás, $S \rightarrow \epsilon$	1
1			OK	
2	léptetés, 2			
3				
4				
5				

$$read(\mathcal{K}_2^*, ()) = \mathcal{K}_2^*$$

## Példa

	(	)	#	S
0	léptetés, 2		redukálás, $S \rightarrow \epsilon$	1
1			OK	
2	léptetés, 2	redukálás, $S \rightarrow \epsilon$		
3				
4				
5				

$$[S \rightarrow (\cdot S)S, \cdot] \in \mathcal{K}_2^* \text{ és } S \Rightarrow^* S \text{ és } ) \in FIRST_1(\ )S )$$

40

Fordítóprogramok előadás (A,C,T) LR elemzések (LALR(1) elemzés)

## Példa

	(	)	#	S
0	léptetés, 2		redukálás, $S \rightarrow \epsilon$	1
1			OK	
2	léptetés, 2	redukálás, $S \rightarrow \epsilon$		3
3				
4				
5				

$$read(\mathcal{K}_2^*, S) = \mathcal{K}_3^*$$

## Példa

	(	)	#	S
0	léptetés, 2		redukálás, $S \rightarrow \epsilon$	1
1			OK	
2	léptetés, 2	redukálás, $S \rightarrow \epsilon$		3
3		léptetés, 4		
4				
5				

$$read(\mathcal{K}_3^*, )) = \mathcal{K}_4^*$$

42

Fordítóprogramok előadás (A,C,T) LR elemzések (LALR(1) elemzés)

43

Fordítóprogramok előadás (A,C,T) LR elemzések (LALR(1) elemzés)

## Példa

	(	)	#	S
0	léptetés, 2		redukálás, $S \rightarrow \epsilon$	1
1			OK	
2	léptetés, 2	redukálás, $S \rightarrow \epsilon$		3
3		léptetés, 4		
4	léptetés, 2			
5				

$$read(\mathcal{K}_4^*, ()) = \mathcal{K}_2^*$$

44

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

	(	)	#	S
0	léptetés, 2		redukálás, $S \rightarrow \epsilon$	1
1			OK	
2	léptetés, 2	redukálás, $S \rightarrow \epsilon$		3
3		léptetés, 4		
4	léptetés, 2	redukálás, $S \rightarrow \epsilon$		
5				

$$[S \rightarrow (S).S, ()] \in \mathcal{K}_4^* \text{ és } S \Rightarrow^* S \text{ és } ) \in FIRST_1( )$$

45

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

	(	)	#	S
0	léptetés, 2		redukálás, $S \rightarrow \epsilon$	1
1			OK	
2	léptetés, 2	redukálás, $S \rightarrow \epsilon$		3
3		léptetés, 4		
4	léptetés, 2	redukálás, $S \rightarrow \epsilon$	redukálás, $S \rightarrow \epsilon$	
5				

$$[S \rightarrow (S).S, \#] \in \mathcal{K}_4^* \text{ és } S \Rightarrow^* S \text{ és } ) \in FIRST_1( \# )$$

46

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

	(	)	#	S
0	léptetés, 2		redukálás, $S \rightarrow \epsilon$	1
1			OK	
2	léptetés, 2	redukálás, $S \rightarrow \epsilon$		3
3		léptetés, 4		
4	léptetés, 2	redukálás, $S \rightarrow \epsilon$	redukálás, $S \rightarrow \epsilon$	5
5				

$$read(\mathcal{K}_4^*, S) = \mathcal{K}_5^*$$

47

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

	(	)	#	S
0	léptetés, 2		redukálás, $S \rightarrow \epsilon$	1
1			OK	
2	léptetés, 2	redukálás, $S \rightarrow \epsilon$		3
3		léptetés, 4		
4	léptetés, 2	redukálás, $S \rightarrow \epsilon$	redukálás, $S \rightarrow \epsilon$	5
5		redukálás, $S \rightarrow (S)S$		

$$[S \rightarrow (S)S., ()] \in \mathcal{K}_5^*$$

48

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

## Példa

	(	)	#	S
0	léptetés, 2		redukálás, $S \rightarrow \epsilon$	1
1			OK	
2	léptetés, 2	redukálás, $S \rightarrow \epsilon$		3
3		léptetés, 4		
4	léptetés, 2	redukálás, $S \rightarrow \epsilon$	redukálás, $S \rightarrow \epsilon$	5
5		redukálás, $S \rightarrow (S)S$	redukálás, $S \rightarrow (S)S$	

$$[S \rightarrow (S)S., \#] \in \mathcal{K}_5^*$$

49

Fordítóprogramok előadás (A,C,T)

LR elemzések (LALR(1) elemzés)

- *Hibajelzés*: a táblázatok üres cellái hibarutinokra hivatkoznak
  - a hibaüzeneteket az adott nyelvtan és nyelv alapján kell kitalálni
  - nem automatizálható

- *Hibajelzés*: a táblázatok üres cellái hibarutinokra hivatkoznak
  - a hibaüzeneteket az adott nyelvtan és nyelv alapján kell kitalálni
  - nem automatizálható
- *Hibaelfedés*: az elemző szinkronizálja a vermet az inputtal és folytatja az elemzést

- ❶ Vezessünk be egy speciális terminális szimbólumot: *error*
- ❷ A „fontos” szabályokhoz (pl. kifejezés, utasítás, blokk) adjunk hozzá egy *hibaalternatívát*:
 
$$\text{Statement} \rightarrow \text{Assignment} ; | \dots | \text{error} ;$$

$$\text{Block} \rightarrow \text{begin StatementList end} | \text{begin error end}$$
- ❸ Az így kiegészített grammatikához készítsük el az elemzőt!

- ❶ Hiba detektálása esetén meghívja a megfelelő hibarutint.
- ❷ A verem tetejéről addig törl, amíg olyan állapotba nem kerül, ahol lehet az *error* szimbólummal lépni.
- ❸ A verembe lépteti az *error* szimbólumot.
- ❹ Az bemeneten addig ugorja át a soron következő terminálisokat, amíg a hibaalternatíva építését folytatni nem tudja.

## Az if-then-else probléma

Fordítóprogramok előadás (A, C, T szakirány)

## Az if-then-else probléma

C/C++ if utasítás

```
if(b1) if(b2) x++; else y++;
```

## Az if-then-else probléma

C/C++ if utasítás

```
if(b1) if(b2) x++; else y++;
```

Mit jelent?

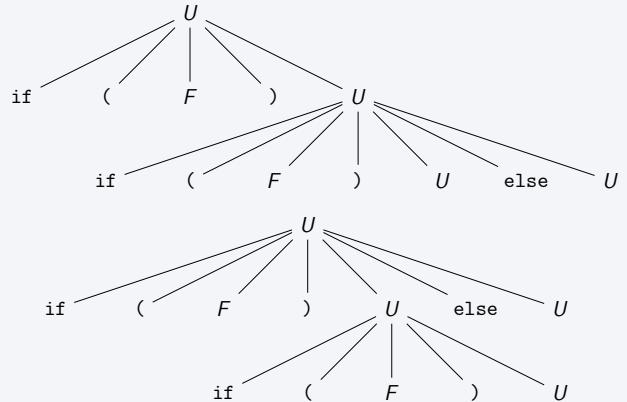
### Egyik lehetőség

```
if(b1)
  if(b2)
    x++;
  else
    y++;
```

### Másik lehetőség

```
if(b1)
  if(b2)
    x++;
  else
    y++;
```

Nyelvtan:  $U \rightarrow \dots | if(F)U | if(F)U\ else U | \dots$



Nyelvtan:  $U \rightarrow \dots | if(F)U | if(F)U\ else U | \dots$

## Rekurzív leszállás

$U \rightarrow \dots | if(F)U | if(F)U\ else U | \dots$

```
void U()
{
  if( aktualis == if_szimbolum )
  {
    elfogad( if_szimbolum );
    elfogad( nyitozarojel );
    F();
    elfogad( csukozarojel );
    U();
    if( aktualis == else_szimbolum )
    {
      elfogad( else_szimbolum );
      U();
    }
  }
  else if( ... )
  ...
}
```

- A  $if(F)if(S)U\ else U$  részmondathoz több szintaxisfa is tartozik.
- **Nem egyértelmű a nyelvtan!**
- Problémát okoz mindegyik elemző esetén.
- A gyakorlatban:

„Az else ág az őt közvetlenül megelőző if utasításhoz tartozik.”

## Rekurzív leszállás

```
if(b1) if(b2) x++; else y++;  
void U()  
{  
    if( aktualis == if_szimbolum )  
    {  
        elfogad( if_szimbolum );  
        elfogad( nyitozarojel );  
        F();  
        elfogad( csukozarojel );  
        U();  
        if( aktualis == else_szimbolum )  
        {  
            elfogad( else_szimbolum );  
            U();  
        }  
    }  
    else if( ... )  
    ...  
}
```

## Rekurzív leszállás

```
if(b1) if(b2) x++; else y++;  
void U()  
{  
    if( aktualis == if_szimbolum )  
    {  
        elfogad( if_szimbolum );  
        elfogad( nyitozarojel );  
        F();  
        elfogad( csukozarojel );  
        U();  
        if( aktualis == else_szimbolum )  
        {  
            elfogad( else_szimbolum );  
            U();  
        }  
    }  
    else if( ... )  
    ...  
}
```

## Rekurzív leszállás

```
if(b1) if(b2) x++; else y++;  
void U()  
{  
    if( aktualis == if_szimbolum )  
    {  
        elfogad( if_szimbolum );  
        elfogad( nyitozarojel );  
        F();  
        elfogad( csukozarojel );  
        U();  
        if( aktualis == else_szimbolum )  
        {  
            elfogad( else_szimbolum );  
            U();  
        }  
    }  
    else if( ... )  
    ...  
}
```

## Rekurzív leszállás

```
if(b1) if(b2) x++; else y++;  
void U()  
{  
    if( aktualis == if_szimbolum )  
    {  
        elfogad( if_szimbolum );  
        elfogad( nyitozarojel );  
        F();  
        elfogad( csukozarojel );  
        U();  
        if( aktualis == else_szimbolum )  
        {  
            elfogad( else_szimbolum );  
            U();  
        }  
    }  
    else if( ... )  
    ...  
}
```

## Rekurzív leszállás

```
if(b1) if(b2) x++; else y++;  
void U()  
{  
    if( aktualis == if_szimbolum )  
    {  
        elfogad( if_szimbolum );  
        elfogad( nyitozarojel );  
        F();  
        elfogad( csukozarojel );  
        U();  
        if( aktualis == else_szimbolum )  
        {  
            elfogad( else_szimbolum );  
            U();  
        }  
    }  
    else if( ... )  
    ...  
}
```

## Rekurzív leszállás

```
if(b1) if(b2) x++; else y++;  
void U()  
{  
    if( aktualis == if_szimbolum )  
    {  
        elfogad( if_szimbolum );  
        elfogad( nyitozarojel );  
        F();  
        elfogad( csukozarojel );  
        U();  
        if( aktualis == else_szimbolum )  
        {  
            elfogad( else_szimbolum );  
            U();  
        }  
    }  
    else if( ... )  
    ...  
}
```

## Rekurzív leszállás

```
if(b1) if(b2) x++; else y++;  
void U()  
{  
    if( aktualis == if_szimbolum )  
    {  
        elfogad( if_szimbolum );  
        elfogad( nyitozarojel );  
        F();  
        elfogad( csukozarojel );  
        U();  
        if( aktualis == else_szimbolum )  
        {  
            elfogad( else_szimbolum );  
            U();  
        }  
        else if( ... )  
        ...  
    }  
}
```

12 Fordítóprogramok előadás (A, C, T szakirány) Az if-then-else probléma

## Rekurzív leszállás

```
if(b1) if(b2) x++; else y++;  
void U()  
{  
    if( aktualis == if_szimbolum )  
    {  
        elfogad( if_szimbolum );  
        elfogad( nyitozarojel );  
        F();  
        elfogad( csukozarojel );  
        U();  
        if( aktualis == else_szimbolum )  
        {  
            elfogad( else_szimbolum );  
            U();  
        }  
        else if( ... )  
        ...  
    }  
}
```

13 Fordítóprogramok előadás (A, C, T szakirány) Az if-then-else probléma

## Rekurzív leszállás

```
if(b1) if(b2) x++; else y++;  
void U()  
{  
    if( aktualis == if_szimbolum )  
    {  
        elfogad( if_szimbolum );  
        elfogad( nyitozarojel );  
        F();  
        elfogad( csukozarojel );  
        U();  
        if( aktualis == else_szimbolum )  
        {  
            elfogad( else_szimbolum );  
            U();  
        }  
        else if( ... )  
        ...  
    }  
}
```

14 Fordítóprogramok előadás (A, C, T szakirány) Az if-then-else probléma

## Rekurzív leszállás

```
if(b1) if(b2) x++; else y++;  
void U()  
{  
    if( aktualis == if_szimbolum )  
    {  
        elfogad( if_szimbolum );  
        elfogad( nyitozarojel );  
        F();  
        elfogad( csukozarojel );  
        U();  
        if( aktualis == else_szimbolum )  
        {  
            elfogad( else_szimbolum );  
            U();  
        }  
        else if( ... )  
        ...  
    }  
}
```

15 Fordítóprogramok előadás (A, C, T szakirány) Az if-then-else probléma

## Rekurzív leszállás

```
if(b1) if(b2) x++; else y++;  
void U()  
{  
    if( aktualis == if_szimbolum )  
    {  
        elfogad( if_szimbolum );  
        elfogad( nyitozarojel );  
        F();  
        elfogad( csukozarojel );  
        U();  
        if( aktualis == else_szimbolum )  
        {  
            elfogad( else_szimbolum );  
            U();  
        }  
        else if( ... )  
        ...  
    }  
}
```

16 Fordítóprogramok előadás (A, C, T szakirány) Az if-then-else probléma

## Rekurzív leszállás

```
if(b1) if(b2) x++; else y++;  
void U()  
{  
    if( aktualis == if_szimbolum )  
    {  
        elfogad( if_szimbolum );  
        elfogad( nyitozarojel );  
        F();  
        elfogad( csukozarojel );  
        U();  
        if( aktualis == else_szimbolum )  
        {  
            elfogad( else_szimbolum );  
            U();  
        }  
        else if( ... )  
        ...  
    }  
}
```

17 Fordítóprogramok előadás (A, C, T szakirány) Az if-then-else probléma

## Rekurzív leszállás

```
if(b1) if(b2) x++; else y++;
void U()
{
    if( aktualis == if_szimbolum )
    {
        elfogad( if_szimbolum );
        elfogad( nyitozarojel );
        F();
        elfogad( csukozarojel );
        U();
        if( aktualis == else_szimbolum )
        {
            elfogad( else_szimbolum );
            U();
        }
    }
    else if( ... )
    ...
}
```

18 Fordítóprogramok előadás (A, C, T szakirány) Az if-then-else probléma

## Rekurzív leszállás

```
if(b1) if(b2) x++; else y++;
void U()
{
    if( aktualis == if_szimbolum )
    {
        elfogad( if_szimbolum );
        elfogad( nyitozarojel );
        F();
        elfogad( csukozarojel );
        U();
        if( aktualis == else_szimbolum )
        {
            elfogad( else_szimbolum );
            U();
        }
    }
    else if( ... )
    ...
}
```

19 Fordítóprogramok előadás (A, C, T szakirány) Az if-then-else probléma

## Rekurzív leszállás

```
if(b1) if(b2) x++; else y++;
void U()
{
    if( aktualis == if_szimbolum )
    {
        elfogad( if_szimbolum );
        elfogad( nyitozarojel );
        F();
        elfogad( csukozarojel );
        U();
        if( aktualis == else_szimbolum )
        {
            elfogad( else_szimbolum );
            U();
        }
    }
    else if( ... )
    ...
}
```

20 Fordítóprogramok előadás (A, C, T szakirány) Az if-then-else probléma

## LALR(1) elemzés

- Az LALR(1) elemzésnél konfliktushoz vezet az *if* utasítás előbb látott nyelvtana.

21 Fordítóprogramok előadás (A, C, T szakirány) Az if-then-else probléma

## LALR(1) elemzés

- Az LALR(1) elemzésnél konfliktushoz vezet az *if* utasítás előbb látott nyelvtana.
- Az egyik kanonikus halmaz:  
 $\mathcal{K}_i = \{[U \rightarrow \text{if } (F) U., \text{else}], [U \rightarrow \text{if } (F) U., \#], [U \rightarrow \text{if } (F) U. \text{ else } U., \text{else}], [U \rightarrow \text{if } (F) U. \text{ else } U., \#]\}$
- Ez egy léptetés-redukálás konfliktus.

## LALR(1) elemzés

- Az LALR(1) elemzésnél konfliktushoz vezet az *if* utasítás előbb látott nyelvtana.
- Az egyik kanonikus halmaz:  
 $\mathcal{K}_i = \{[U \rightarrow \text{if } (F) U., \text{else}], [U \rightarrow \text{if } (F) U., \#], [U \rightarrow \text{if } (F) U. \text{ else } U., \text{else}], [U \rightarrow \text{if } (F) U. \text{ else } U., \#]\}$
- Ez egy léptetés-redukálás konfliktus.
- Feloldása: léptetni kell!
  - Így az *else* az ót közvetlenül megelőző *if* utasításhoz fog tartozni.

21 Fordítóprogramok előadás (A, C, T szakirány) Az if-then-else probléma

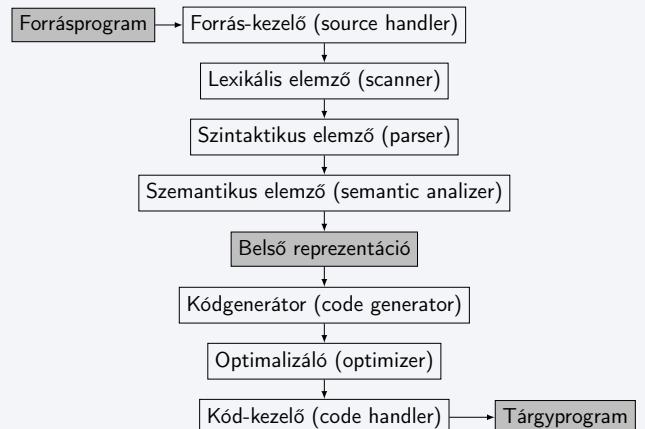
- Megoldás: az *if* utasítás végét jelző kulcsszó bevezetése.

```
    if(b1)
    {
if b1 then          if(b2)
                  {
if b2 then          x++;
                  {
else
                  y := y+1;
end if;
end if;           else
                  {
                  y++;
}
}
```

## Szimbólumtábla-kezelés

Fordítóprogramok előadás (A, C, T szakirány)

## Emlékeztető: a fordítás lépései



## Információáramlás

### Programszöveg

$x = y + 2;$

- Lexikális elemző: lexikális elemek sorozatát állítja elő

### Lexikális elemek

változó operátor változó operátor számkonstans utasításvég

- Szintaktikus elemző: felépíti a szintaxisfát

## Információáramlás

### Szemantikus elemző:

- deklarálva van-e az  $x$  és  $y$  változó?
- $x$  és  $y + 2$  kifejezések típusa azonos-e?
- stb.

- Kódgenerálás: utasítások generálása a tárgyprogram nyelvén, amelyek megvalósítják az értékkedést.

## Információáramlás

### Szemantikus elemző:

- deklarálva van-e az  $x$  és  $y$  változó?
- $x$  és  $y + 2$  kifejezések típusa azonos-e?
- stb.

- Kódgenerálás: utasítások generálása a tárgyprogram nyelvén, amelyek megvalósítják az értékkedést.

A szemantikus elemzéshez és a kódgeneráláshoz nem elég a lexikális elemek sorozata és a szintaxis!

## Információáramlás

### A szemantikus elemzés és a kódgenerálás számára szükséges kiegészítő információk:

- konstansok értéke
- változók típusa
- függvények, operátorok szignatúrája
- a tárgykódba már bekerült változók, függvények címe
- részkifejezések típusa, hozzárendelt tárgykód
- stb.

## Információáramlás

- A szemantikus elemzés és a kódgenerálás számára szükséges kiegészítő információk:
  - konstansok értéke
  - változók típusa
  - függvények, operátorok szignatúrája
  - a tárgykódba már bekerült változók, függvények címe
  - részkifejezések típusa, hozzárendelt tárgykód
  - stb.
- Ezeket az információkat két módon tároljuk:
  - **szimbólumtábla**
  - szemantikus értékek

## Szimbólumtáblában tárolt információk

- **szimbólum neve**
- **szimbólum attribútumai:**
  - definíció adatai
  - típus
  - tárgyprogram-beli cím
  - definíció helye a forrásprogramban
  - szimbólumra hivatkozások a forrásprogramban

## Szimbólum neve

- a szemantikus elemző és a kódgenerátor a szimbólumokat a nevükkel azonosítja
- a nevek általában változó hosszúságúak lehetnek
  - érdemes dinamikus memoriában tárolni
  - a tábla csak egy mutatót tartalmaz
  - pl. C++ `std :: string` választás esetén pont ez történik...

## Definíció adatai

- **változó**
  - típus
  - módosító kulcsszavak: `const`, `static` ...
  - címe a tárgyprogramban (függ a változó tárolási módjától)

## Definíció adatai

- **változó**
  - típus
  - módosító kulcsszavak: `const`, `static` ...
  - címe a tárgyprogramban (függ a változó tárolási módjától)
- **függvény, eljárás, operátor**
  - paraméterek típusa
  - visszatérési típus
  - módosítók
  - címe a tárgyprogramban

## Definíció adatai

- **típus (típusleíró, típusdeszkriptor)**
  - egyszerű típusok: méret
  - rekord: mezők nevei és típusleírói
  - tömb: elem típusleírása, index típusleírása, méret
  - intervallum-típus: elem típusleírása, minimum, maximum
  - unio-típus: a lehetséges típusok leírói, méret

## Tárgyprogram-beli cím

- A változók címét a definiáláskor határozza meg a fordító.
  - globális és statikus változók:
    - az adatszegmens kezdetétől számított relatív cím
    - az utoljára elhelyezett változó utáni szabad helyre
  - lokális változók (alprogramban vagy belső blokkban deklarálvá):
    - a veremben kap helyet
    - az aktuális veremkereten belüli relatív cím
    - az ebben a blokkban utoljára deklarált változó után
  - dinamikusan foglalt változók
    - a heap memoriában kapnak helyet
    - címük csak a program futásakor derül ki
    - mutatókkal hivatkozunk rájuk, csak azonak van „neve”
    - deklarált mutatók az előző két kategóriába esnek

## Tárgyprogram-beli cím

- Az alprogramok tárgyprogram-beli címét is definiáláskor határozza meg a fordító.
  - kódszegmensen belüli relatív cím
  - az utoljára definiált alprogram utáni szabad helyre

## Hivatkozások a szimbólumra

- deklaráció, definíció és hivatkozások sorainak száma a forrásprogramban
- hasznos lehet:
  - jó hibaüzenetek generálásához
  - kereszthivatkozás-lista létrehozásához  
„Hol melyik változót használjuk?”)

## (Túl) egyszerű példa

### Forrásprogram

```

1. double d;
2. double fv( int i )
3. {
4.   int j = i*i;
5.   return d*j;
6. }
```

Név	Fajta	Típus	Param.	Def.	Hivatk.

## (Túl) egyszerű példa

### Forrásprogram

```

1. double d;
2. double fv( int i )
3. {
4.   int j = i*i;
5.   return d*j;
6. }
```

Név	Fajta	Típus	Param.	Def.	Hivatk.
"d"	változó	double		1	

## (Túl) egyszerű példa

### Forrásprogram

```

1. double d;
2. double fv( int i )
3. {
4.   int j = i*i;
5.   return d*j;
6. }
```

Név	Fajta	Típus	Param.	Def.	Hivatk.
"d"	változó	double		1	
"fv"	függvény	double	int	2	
"i"	paraméter	int		2	

## (Túl) egyszerű példa

### Forrásprogram

```
1. double d;  
2. double fv( int i )  
3. {  
4.     int j = i*i;  
5.     return d*j;  
6. }
```

Név	Fajta	Típus	Param.	Def.	Hivatk.
"d"	változó	double		1	
"fv"	függvény	double	int	2	
"i"	paraméter	int		2	4
"j"	változó	int		4	

## (Túl) egyszerű példa

### Forrásprogram

```
1. double d;  
2. double fv( int i )  
3. {  
4.     int j = i*i;  
5.     return d*j;  
6. }
```

Név	Fajta	Típus	Param.	Def.	Hivatk.
"d"	változó	double		1	5
"fv"	függvény	double	int	2	
"i"	paraméter	int		2	4
"j"	változó	int		4	5

## A szimbólumtábla műveletei

- keresés
  - szimbólum használatakor
- beszúrás
  - új szimbólum megjelenésekor
  - tartalmaz egy keresést is: „Volt-e már deklarálva?”

Ezek hatékonysága nagy mértékben befolyásolja a fordítóprogram sebességét!

## Hatókör, láthatóság, élettartam

- hatókör: „Ahhol a deklaráció érvényben van.”

## Hatókör, láthatóság, élettartam

- hatókör: „Ahhol a deklaráció érvényben van.”
- láthatóság: „Ahhol hivatkozni lehet rá a nevével.”
  - része a hatókörnek
  - az elfedés miatt lehet kisebb, mint a hatókör

## Hatókör, láthatóság, élettartam

- hatókör: „Ahhol a deklaráció érvényben van.”
- láthatóság: „Ahhol hivatkozni lehet rá a nevével.”
  - része a hatókörnek
  - az elfedés miatt lehet kisebb, mint a hatókör
- élettartam: „Amíg memóriaterület van hozzárendelve.”

## Példa: hatókör

```
int x = 2;
cout << x << endl;
{
    cout << x << endl;
    int x = 3;
    cout << x << endl;
}
cout << x << endl;
```

## Példa: láthatóság

```
int x = 2;
cout << x << endl;
{
    cout << x << endl;
    int x = 3;
    cout << x << endl;
}
cout << x << endl;
```

## Hatókör és láthatóság kezelése szimbólumtáblával

- A szimbólumokat egy verembe tesszük.
- Keresés:
  - a verem tetejéről indul
  - az első találatnál megáll
- Blokk végén a hozzá tartozó szimbólumokat töröljük.

## Hatókör és láthatóság kezelése szimbólumtáblával

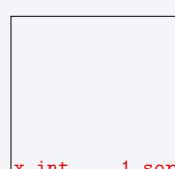
- A szimbólumokat egy verembe tesszük.
- Keresés:
  - a verem tetejéről indul
  - az első találatnál megáll
- Blokk végén a hozzá tartozó szimbólumokat töröljük.

1. int x = 2;
2. cout << x << endl;
3. {
4. cout << x << endl;
5. int x = 3;
6. cout << x << endl;
7. }
8. cout << x << endl;

## Hatókör és láthatóság kezelése szimbólumtáblával

- A szimbólumokat egy verembe tesszük.
- Keresés:
  - a verem tetejéről indul
  - az első találatnál megáll
- Blokk végén a hozzá tartozó szimbólumokat töröljük.

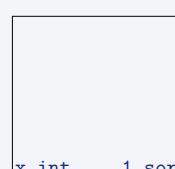
```
1. int x = 2;
2. cout << x << endl;
3. {
4.   cout << x << endl;
5.   int x = 3;
6.   cout << x << endl;
7. }
8. cout << x << endl;
```



## Hatókör és láthatóság kezelése szimbólumtáblával

- A szimbólumokat egy verembe tesszük.
- Keresés:
  - a verem tetejéről indul
  - az első találatnál megáll
- Blokk végén a hozzá tartozó szimbólumokat töröljük.

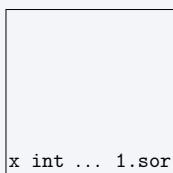
```
1. int x = 2;
2. cout << x << endl;
3. {
4.   cout << x << endl;
5.   int x = 3;
6.   cout << x << endl;
7. }
8. cout << x << endl;
```



## Hatókör és láthatóság kezelése szimbólumtáblával

- A szimbólumokat egy verembe tesszük.
- Keresés:
  - a verem tetejéről indul
  - az első találatnál megáll
- Blokk végén a hozzá tartozó szimbólumokat töröljük.

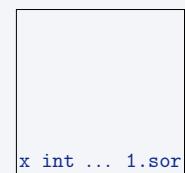
```
1. int x = 2;
2. cout << x << endl;
3. {
4.   cout << x << endl;
5.   int x = 3;
6.   cout << x << endl;
7. }
8. cout << x << endl;
```



## Hatókör és láthatóság kezelése szimbólumtáblával

- A szimbólumokat egy verembe tesszük.
- Keresés:
  - a verem tetejéről indul
  - az első találatnál megáll
- Blokk végén a hozzá tartozó szimbólumokat töröljük.

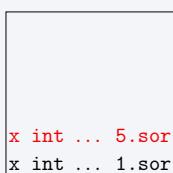
```
1. int x = 2;
2. cout << x << endl;
3. {
4.   cout << x << endl;
5.   int x = 3;
6.   cout << x << endl;
7. }
8. cout << x << endl;
```



## Hatókör és láthatóság kezelése szimbólumtáblával

- A szimbólumokat egy verembe tesszük.
- Keresés:
  - a verem tetejéről indul
  - az első találatnál megáll
- Blokk végén a hozzá tartozó szimbólumokat töröljük.

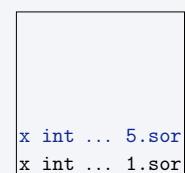
```
1. int x = 2;
2. cout << x << endl;
3. {
4.   cout << x << endl;
5.   int x = 3;
6.   cout << x << endl;
7. }
8. cout << x << endl;
```



## Hatókör és láthatóság kezelése szimbólumtáblával

- A szimbólumokat egy verembe tesszük.
- Keresés:
  - a verem tetejéről indul
  - az első találatnál megáll
- Blokk végén a hozzá tartozó szimbólumokat töröljük.

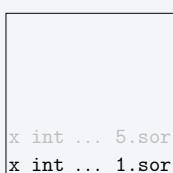
```
1. int x = 2;
2. cout << x << endl;
3. {
4.   cout << x << endl;
5.   int x = 3;
6.   cout << x << endl;
7. }
8. cout << x << endl;
```



## Hatókör és láthatóság kezelése szimbólumtáblával

- A szimbólumokat egy verembe tesszük.
- Keresés:
  - a verem tetejéről indul
  - az első találatnál megáll
- Blokk végén a hozzá tartozó szimbólumokat töröljük.

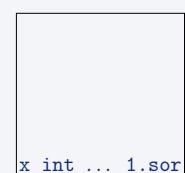
```
1. int x = 2;
2. cout << x << endl;
3. {
4.   cout << x << endl;
5.   int x = 3;
6.   cout << x << endl;
7. }
8. cout << x << endl;
```



## Hatókör és láthatóság kezelése szimbólumtáblával

- A szimbólumokat egy verembe tesszük.
- Keresés:
  - a verem tetejéről indul
  - az első találatnál megáll
- Blokk végén a hozzá tartozó szimbólumokat töröljük.

```
1. int x = 2;
2. cout << x << endl;
3. {
4.   cout << x << endl;
5.   int x = 3;
6.   cout << x << endl;
7. }
8. cout << x << endl;
```



## Verem szimbólumtábla

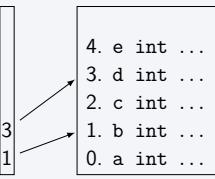
- „Meddig kell a szimbólumokat törölni a blokk végén?”
  - Számon kell tartani a blokk kezdetét a szimbólumtábla vermében!

## Verem szimbólumtábla

- „Meddig kell a szimbólumokat törölni a blokk végén?”
  - Számon kell tartani a blokk kezdetét a szimbólumtábla vermében!
- Ötlet: **blokk-index vektor**
  - ez is egy verem
  - blokk kezdetén: *set*
    - az új blokk első szimbólumára mutató pointert teszünk a blokk-index vektorba
  - blokk végén: *reset*
    - a blokk-index vektor tetején lévő pointer mutatja, hogy meddig kell törölni az elemeket
    - a szimbólumok törlése után a blokk-index vektor legfelső elemét is törölni kell
- Név: **verem szimbólumtábla**

## Példa

```
int a
{
    int b;
    int c;
    {
        int d;
        int e;
        ...
    }
}
```



## Hatókonyság-növelés

- verem-szimbólumtábla:
  - keresés: lineáris
  - beszúrás (mivel tartalmaz egy keresést is): lineáris

## Hatókonyság-növelés

- verem-szimbólumtábla:
  - keresés: lineáris
  - beszúrás (mivel tartalmaz egy keresést is): lineáris
- hatékonyabb adatszerkezetek:
  - kiegyensúlyozott fa:**  
keresés és beszúrás is logaritmikus
  - hash-tábla:**  
keresés és beszúrás is konstans műveletigényű  
(jól megválasztott hash-függvény esetén)

## Faszerkezetű szimbólumtábla

- Minden blokkhoz egy fa tartozik.
  - A szimbólumtábla sorai a fa csúcsaiban helyezkednek el.
- A fákat egy veremben tároljuk.
  - Új blokk kezdetén egy új (üres) fát teszünk a verembe.
  - A blokk szimbólumait ebbe a fába láncoljuk be.
  - Időnként kiegyensúlyozzuk a fát.  
(Ha a nyelvben van külön deklarációs része a blokkoknak, akkor elég ennek a végén kiegyensúlyozni.)
  - A blokk végén a teljes fát törölni kell a veremből.
- Keresés:
  - A verem tetején lévő fában kezdjük.
  - Továbblépünk az előző fára, ha addig nem volt találat.
  - Az első előfordulásig keresünk.

## Hash-szerkezetű szimbólumtábla

- szimbólumtábla sorai: veremben (mint a verem-szimbólumtábla esetén)
  - van blokk-index vektor is
- hash-függvény: a szimbólum nevét egy hash-értékre képezi le
- hash-tábla: a hash értékekhez hozzárendeli a legutóbbi ilyen hash-kódú szimbólum pozícióját a veremben
  - ha még nem volt ilyen kódú szimbólum: nullptr vagy speciális érték
  - ha több ilyen kódú szimbólum is van: láncolni kell őket a legutóbbitól a régebbiek felé

## Keresés a hash-szerkezetű szimbólumtáblában

- a hash-függvénnel meghatározzuk a keresendő szimbólum hash-értékét
- a hash-táblából megkapjuk az ilyen kódú szimbólumok láncolt listáját
- végigmegyünk a listán az első találatig

## Beszúrás a hash-szerkezetű szimbólumtáblába

- a szimbólumot (és attribútumait) a verem tetejére helyezzük
- a hash-függvénnel meghatározzuk a beszúrandó szimbólum hash-értékét
- a hash táblában a kapott hash értékhez bejegyezzük a verem-beli pozíciót
- ha volt már ilyen hash-kódú szimbólum, akkor az új szimbólumhoz beláncoljuk

## Törlés a hash-szerkezetű szimbólumtáblából

- blokk végén a blokk-index vektor tetején lévő elem mutatja, hogy meddig kell törlni a szimbólumokat
- egy szimbólum törlése:
  - előállítjuk a hash-értékét
  - a hash-táblába a kapott hash-értékhez beírjuk a törlendő elemhez láncolt következő elem pozícióját (ha nincs hozzá láncolva elem, akkor a hash táblába az üres lista jelzés kerül)
  - eltávolítjuk az elemet a verem tetejéről
- a blokk összes szimbólumának törlése után a blokk-index vektor legfelső elemét is töröljük

## Minősített nevek kezelése

```
namespace a
{
    int x = 1;
    namespace b
    {
        int x = 2;
        void print()
        {
            cout << a::x << x << endl;
        }
    }
}
```

## Minősített nevek kezelése

```
namespace a
{
    int x = 1;
    namespace b
    {
        int x = 2;
        void print()
        {
            cout << a::x << x << endl;
        }
    }
}
```

A (külső) x és b szimbólumokhoz fel kell jegyezni a szimbólumtáblába, hogy az a névtérhez tartoznak.

## Minősített nevek kezelése

```
namespace a
{
    int x = 1;
    namespace b
    {
        int x = 2;
        void print()
        {
            cout << a::x << x << endl;
        }
    }
}
```

A belső `x` és `print` szimbólumokhoz fel kell jegyezni a szimbólumtáblába, hogy a `b` névtérhez tartoznak.

## Minősített nevek kezelése

```
namespace a
{
    int x = 1;
    namespace b
    {
        int x = 2;
        void print()
        {
            cout << a::x << x << endl;
        }
    }
}
```

A nem minősített nevű szimbólumok keresése nem változik.

## Minősített nevek kezelése

```
namespace a
{
    int x = 1;
    namespace b
    {
        int x = 2;
        void print()
        {
            cout << a::x << x << endl;
        }
    }
}
```

Az `a :: x` szimbólum keresésekor olyan `x`-et kell keresni a szimbólumtáblában, amihez fel van jegyezve, hogy az a névtérben van.

## Szimbólumok importálása

```
namespace a
{
    int x = 1;
}
using namespace a;
int main()
{
    cout << x << endl;
    return 0;
}
```

- A névterek szimbólumait a veremből nem törölni kell, hanem feljegyezni egy másik tárterületre.
- A `using` direktíva használatakor az importált névtér szimbólumait be kell másolni a verembe (vagy legalább hivatkozást tenni a verembe erre a névtérre).

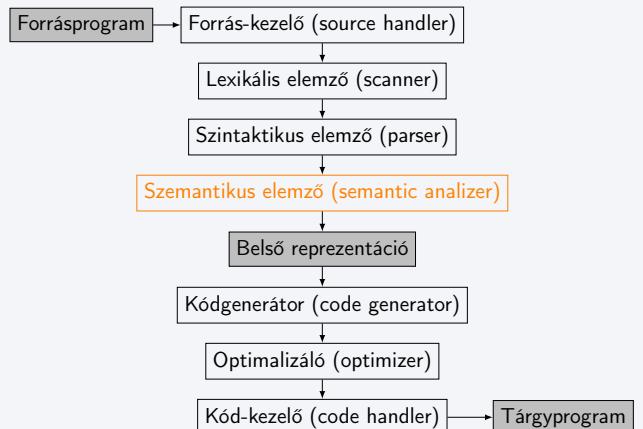
## Osztályok

- A rekordokhoz hasonló: típusleírót kell készíteni hozzá.
- Láthatóság szabályozása: a mezőkhöz és tagfüggvényekhez fel kell jegyezni, hogy milyen a láthatóságuk (`public`, `protected`, `private`).
- Az osztályok névteret is alkotnak: (statikus) adattagjai minősített névvel is elérhetők, ilyenkor az előbb látott technikát lehet használni.

## A szemantikus elemzés feladatai

Fordítóprogramok előadás (A, C, T szakirány)

## A szemantikus elemzés helye



1 Fordítóprogramok előadás (A, C, T szakirány)

A szemantikus elemzés feladatai

2 Fordítóprogramok előadás (A, C, T szakirány)

A szemantikus elemzés feladatai

## A szemantikus elemzés feladatai

A szemantikus elemzés jellemzően a környezetfüggő ellenőrzéseket valósítja meg.

- deklarációk kezelése
  - változók
  - függvények, eljárások, operátorok
  - típusok
- láthatósági szabályok
- aritmetikai ellenőrzések
- a program szintaxisának környezetfüggő részei
- típusellenőrzés
- stb.

A szemantikus elemzés erősen függ a konkrét programozási nyelvtől!

## Deklarációk és láthatósági szabályok

### Többszörös deklaráció

```
int x = 1;
cout << x;
int x = 2;
cout << x;
```

### Elfedés

```
int x = 1;
cout << x;
{
    int x = 2;
    cout << x;
}
```

### Adatelrejtés

```
class sajat
{
    public:
        int x;
    private:
        int y;
}
...
sajat s;
cout << s.x;
cout << s.y;
```

3 Fordítóprogramok előadás (A, C, T szakirány)

A szemantikus elemzés feladatai

4 Fordítóprogramok előadás (A, C, T szakirány)

A szemantikus elemzés feladatai

## Deklarációk és láthatósági szabályok

- Jól megválasztott szimbólumtáblával megoldható:
  - verem szerkezetű szimbólumtábla (+ keresőfa vagy hash-tábla)
  - minden blokkhoz egy új szint a veremben
  - a keresés a verem tetejéről indul
- Lásd az előző előadás anyagát!

## Aritmetikai ellenőrzések

- pl. a nullával osztás eseténként kiszűrhető:

### Konstanssal osztás

```
a = b / 0;
// Itt lehet hibát vagy
// figyelmeztetést adni!
```

### Változóval osztás

```
cin >> c;
a = b / c;
// Itt nem...
```

- hasonlóan kiszűrhető konstansok esetén a túl- vagy alulcsordulás

5 Fordítóprogramok előadás (A, C, T szakirány)

A szemantikus elemzés feladatai

6 Fordítóprogramok előadás (A, C, T szakirány)

A szemantikus elemzés feladatai

## A szintaxis környezetfüggő részei

Példa: Ada eljárásdefiníciók

```
procedure eljarasom is
  ...
end eljarasom
```

## A szintaxis környezetfüggő részei

Példa: Ada eljárásdefiníciók

```
procedure eljarasom is
  ...
end eljarasom
```

A környezetfüggetlen nyelvtan nem tudja leírni a szabályt:

*EljDef → procedure Azonosito is Program end Azonosito*

## A szintaxis környezetfüggő részei

Példa: Ada eljárásdefiníciók

```
procedure eljarasom is
  ...
end eljarasom
```

A környezetfüggetlen nyelvtan nem tudja leírni a szabályt:

*EljDef → procedure Azonosito is Program end Azonosito*

Ez megengedi ezt a programszöveget is:

Hibás eljárásdefiníció

```
procedure egyik is
  ...
end masik
```

**Ezt a hibát a szemantikus elemzésnek kell megtalálnia.**

## Típusellenőrzés

- első közelítésben a kifejezések típusozhatóságának ellenőrzése

*s.length() + 1*

```
s          :: string
s.length() :: int
1          :: int
s.length() + 1 :: int
```

*s + 1*

```
s      :: string
1      :: int
s + 1 :: hiba
```

## Típusellenőrzés

- első közelítésben a kifejezések típusozhatóságának ellenőrzése

*s.length() + 1*

```
s          :: string
s.length() :: int
1          :: int
s.length() + 1 :: int
```

*s + 1*

```
s      :: string
1      :: int
s + 1 :: hiba
```

- mára nagyon kifejező típusrendszerek léteznek

- sablon (generikus) típusok
- altípusok
- öröklődés
- minél több információ tárolása a típusban  
(pl. string hossza, gráf párossága stb.)

## A típusok szerepe

- időnként a típushibás utasításokhoz nem lehet értelmesen kódot generálni

- pl. különböző méretű rekordok közötti értékadás

- a típusellenőrzés számos elírást felderít

```
string s;
if( s = 'hello' )
// ...
```

## Statikus vs. dinamikus típusozás

- **Statikus:** a kifejezésekhez *fordítási időben* a szemantikus elemzés rendel típust
  - az ellenőrzések fordítási időben történnek
  - futás közben csak az értékeket kell tárolni
  - futás közben „nem történhet baj”
  - előny: biztonságosabb
  - pl.: Ada, C++, Haskell ...
- **Dinamikus:** a típusellenőrzés *futási időben* történik
  - futás közben az értékek mellett típusinformációt is kell tárolni
  - minden utasítás végrehajtása előtt ellenőrizni kell a típusokat
  - típushiba esetén futási idejű hiba keletkezik
  - előny: hajlékonyabb
  - pl.: Lisp, Erlang ...

## Statikus és dinamikus típusozás

Bizonyos feladatokhoz használni kell a dinamikus típusellenőrzés technikáit:

- objektumorientált nyelvekben a *dinamikus kötés*
- Java *instanceof* operátora

## Típusellenőrzés vs. típuslevezetés

```
C++  
int fac( int n )  
{  
    if( n == 0 )  
        return 1;  
    else  
        return n * fac(n-1);  
}
```

```
Haskell  
fac n =  
    if n == 0  
    then 1  
    else n * fac (n-1)
```

## Típusellenőrzés vs. típuslevezetés

### Típusellenőrzés:

- minden típus a deklarációban adott
- a kifejezések egyszerű szabályok alapján típusozhatók
- egyszerűbb fordítóprogram, gyorsabb fordítás
- kényelmetlenebb a programozónak

## Típusellenőrzés vs. típuslevezetés

### Típusellenőrzés:

- minden típus a deklarációban adott
- a kifejezések egyszerű szabályok alapján típusozhatók
- egyszerűbb fordítóprogram, gyorsabb fordítás
- kényelmetlenebb a programozónak

### Típuslevezetés, típuskikövetkeztetés:

- a változók, függvények típusait (általában) nem kell megadni
- a típusokat fordítóprogram „találja ki” a definíciójuk, használatuk alapján
- bonyolultabb fordítóprogram, lassabb fordítás
- kényelmesebb a programozónak

## Automatikus típuskonverziók

```
void f1( double d ) {}  
void f2( int i ) {}  
int main()  
{  
    int i;  
    double d;  
    f1(i); // ez megy  
    f2(i);  
    f1(d);  
    f2(d); // ez problémás ...  
    return 0;  
}
```

### Warning

warning: passing 'double' for argument 1  
to 'void f2(int)',

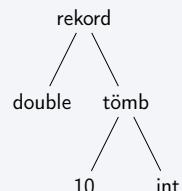
- C++ példa:
  - *int ~> double* automatikusan konvertálódik
  - *double ~> int* figyelmeztetést vált ki (elvesztjük a törrészt)
- ha nincs automatikus típuskonverzió:
  - ki kell írni: `f2( (int)d );`
  - kényelmetlenebb, de biztonságosabb
- ha nagyon sok automatikus típuskonverzió van:
  - időnként meglepő eredmény születhet
  - kényelmesebb, de veszélyes lehet

- alaptípusok
  - pl. `int, double, char ...`
- összetett típusok
  - tömb
  - rekord (osztály ...)
  - unió

- alaptípusok
  - pl. `int, double, char ...`
- összetett típusok
  - tömb
  - rekord (osztály ...)
  - unió

Összetett típusok fa-struktúrája:

```
struct T
{
    double d;
    int t[10];
};
```



Mikor tekintsünk két típust ekvivalénsnek?

#### • szerkezeti (strukturális) ekvivalencia:

„Két típus egyenlő, ha az őket leíró fa azonos.”

- bonyolultabb az ellenőrzés
- nem minden fejezi ki a programozói szándékot
  - `struct Ember { string nev; int eletkor; };`
  - `struct Uzenet { string szoveg; int azonosito; };`

#### • név szerinti ekvivalencia:

„Két típus egyenlő, ha a nevük azonos.”

- egyszerűbb az ellenőrzés (`==`)
- időnként kényelmetlen lehet a programozónak

- altípus: általában valamilyen megszorítást tesz az alaptípusra

- pl. a természetes szám típus altípusa az egész szám típusnak

#### • származtatott típus:

- öröklődéssel jön létre az alaptípusból
- a specializáció sokféle lehet:
  - új adattag
  - új metódus
  - meglévő metódus felüldefiniálása

**Upcast:** Altípus vagy származtatott típus mindenhol használható, ahol az alaptípus megengedett.

**Downcast:** Az alaptípus általában nem használható a származtatott vagy altípus helyett.

Az öröklődési vagy altípus kapcsolatok a szokásos módon ábrázolhatók fával (többszörös öröklődés esetén gráffal):

**Upcast**

```

void f( Kutya k )
{ ... }

int main()
{
    Vizsla v;
    f( v ); // gond nélkül megy
}
  
```

**Upcast**

```

void f( Kutya k )
{ ... }

int main()
{
    Vizsla v;
    f( v ); // gond nélkül megy
}
  
```

Típusellenőrzés: a fában felfelé kell keresni, hogy van-e olyan ősosztály (alaptípus), ami használható az adott helyen.

## Szemantikus elemzés (attribútum fordítási grammatikák)

Fordítóprogramok előadás (A, C, T szakirány)

## A szemantikus elemzés elmélete

- a nyelvtan szabályait kiegészítjük a szemantikus elemzés tevékenységeivel  
⇒ [fordítási grammatikák](#)

## A szemantikus elemzés elmélete

- a nyelvtan szabályait kiegészítjük a szemantikus elemzés tevékenységeivel  
⇒ [fordítási grammatikák](#)
- a nyelvtan szimbólumaihoz szemantikus típusokat (attribútumokat) rendelünk  
⇒ [attribútum fordítási grammatikák](#)

## A szemantikus elemzés elmélete

- a nyelvtan szabályait kiegészítjük a szemantikus elemzés tevékenységeivel  
⇒ [fordítási grammatikák](#)
- a nyelvtan szimbólumaihoz szemantikus típusokat (attribútumokat) rendelünk  
⇒ [attribútum fordítási grammatikák](#)
- megvizsgáljuk, hogy mikor lehet kiértékelni az attribútumértékeket  
⇒ [jól definiált attribútum fordítási grammatikák](#)

## A szemantikus elemzés elmélete

- a nyelvtan szabályait kiegészítjük a szemantikus elemzés tevékenységeivel  
⇒ [fordítási grammatikák](#)
- a nyelvtan szimbólumaihoz szemantikus típusokat (attribútumokat) rendelünk  
⇒ [attribútum fordítási grammatikák](#)
- megvizsgáljuk, hogy mikor lehet kiértékelni az attribútumértékeket  
⇒ [jól definiált attribútum fordítási grammatikák](#)
- megszorításokat teszünk a grammikára, hogy a kiértékelés egyszerűbb legyen  
⇒ [particionált attribútum fordítási grammatikák](#)  
⇒ [rendezett attribútum fordítási grammatikák](#)

## A szemantikus elemzés elmélete

- a nyelvtan szabályait kiegészítjük a szemantikus elemzés tevékenységeivel  
⇒ [fordítási grammatikák](#)
- a nyelvtan szimbólumaihoz szemantikus típusokat (attribútumokat) rendelünk  
⇒ [attribútum fordítási grammatikák](#)
- megvizsgáljuk, hogy mikor lehet kiértékelni az attribútumértékeket  
⇒ [jól definiált attribútum fordítási grammatikák](#)
- megszorításokat teszünk a grammikára, hogy a kiértékelés egyszerűbb legyen  
⇒ [particionált attribútum fordítási grammatikák](#)  
⇒ [rendezett attribútum fordítási grammatikák](#)
- megvizsgáljuk, hogyan illeszthetők ezek a módszerek a tanult szintaktikus elemzőkhöz

- A grammatika szabályaiban jelöljük, hogy milyen szemantikus elemzési tevékenységekre van szükség.
  - Ezeket hívjuk **akciósziimbólumoknak**.
  - A nyelvtanban @ jelrel kezdődnek.
  - Az akciósziimbólumok által jelölt szemantikus tevékenységeket **szemantikus rutinoknak** nevezzük.

- A grammatika szabályaiban jelöljük, hogy milyen szemantikus elemzési tevékenységekre van szükség.
  - Ezeket hívjuk **akciósziimbólumoknak**.
  - A nyelvtanban @ jelrel kezdődnek.
  - Az akciósziimbólumok által jelölt szemantikus tevékenységeket **szemantikus rutinoknak** nevezzük.
- Az ilyen módon kiegészített grammatikát **fordítási grammatikának** nevezzük.

## Példa I.

## Értékadó utasítás

$\text{Értékadás} \rightarrow \text{Változó} := \text{Kifejezés } @\text{Ellenőrzés}$

Az  $@\text{Ellenőrzés}$  akciósziimbólum a következő tevékenységet jelöli:

- ellenőrizni kell, hogy a változó típusa és a kifejezés típusa megfelelnek-e egymásnak  
(egyenlők-e, vagy a kifejezés típusa konvertálható-e a változó típusára)
- ellenőrizni kell, hogy a változónak szabad-e értéket adni  
(pl. nem konstans változó-e stb.)

## Példa II.

## Változódeklaráció

$\text{Deklaráció} \rightarrow \text{Típus } \text{Változó } @\text{Feljegyzés}$

A  $@\text{Feljegyzés}$  akciósziimbólum a következő tevékenységet jelöli:

- ellenőrizni kell, hogy a deklaráció nem ütközik-e egy korábbival  
(használni kell a szimbólumtáblát)
- a változó adatait be kell tenni a szimbólumtáblába

## Attribútumok

- Hol találják a szemantikus rutinok a szükséges információkat?  
Hova írják a kiszámolt eredményt?
  - a szemantikus rutinok önállóan is gondoskodhatnak a paraméterátadásukról  
(például külön vermet kezelhetnek)
  - jobb megoldás: a szemantikus információkat a szintaktikus elemző szimbólumaihoz csatolva tároljuk

## Attribútumok

- Hol találják a szemantikus rutinok a szükséges információkat?  
Hova írják a kiszámolt eredményt?
  - a szemantikus rutinok önállóan is gondoskodhatnak a paraméterátadásukról  
(például külön vermet kezelhetnek)
  - jobb megoldás: a szemantikus információkat a szintaktikus elemző szimbólumaihoz csatolva tároljuk
- A szimbólumokhoz **attribútumokat** rendelünk.  
Ezek jelzik, hogy a szimbólumhoz milyen szemantikus értékek (attribútumértékek) kapcsolódnak.
- Jelölés:  $A.x, y$   
Az A szimbólumhoz az x és y attribútumokat rendeljük.

## Példa I.

### Kifejezés

$Kifejezés_0.t \rightarrow Kifejezés_1.t \pm Kifejezés_2.t @TípusEllenőrzés$   
 $Kifejezés.t \rightarrow \underline{\text{konstans}.t} @\text{KonstansKifejezés}$

- A Kifejezés szimbólum különböző előfordulásait indexeléssel különböztetjük meg.

## Példa I.

### Kifejezés

$Kifejezés_0.t \rightarrow Kifejezés_1.t \pm Kifejezés_2.t @TípusEllenőrzés$   
 $Kifejezés.t \rightarrow \underline{\text{konstans}.t} @\text{KonstansKifejezés}$

- A Kifejezés szimbólum különböző előfordulásait indexeléssel különböztetjük meg.
- A  $@TípusEllenőrzés$  által jelzett szemantikus rutin:
  - $Kifejezés_0.t := \text{int}$

7

Fordítóprogramok előadás (A, C, T szakirány)

Szemantikus elemzés (attribútum fordítási grammatikák)

7

Fordítóprogramok előadás (A, C, T szakirány)

Szemantikus elemzés (attribútum fordítási grammatikák)

## Példa I.

### Kifejezés

$Kifejezés_0.t \rightarrow Kifejezés_1.t \pm Kifejezés_2.t @TípusEllenőrzés$   
 $Kifejezés.t \rightarrow \underline{\text{konstans}.t} @\text{KonstansKifejezés}$

- A Kifejezés szimbólum különböző előfordulásait indexeléssel különböztetjük meg.
- A  $@TípusEllenőrzés$  által jelzett szemantikus rutin:
  - $Kifejezés_0.t := \text{int}$
- A  $@\text{KonstansKifejezés}$  által jelzett szemantikus rutin:
  - $Kifejezés.t := \underline{\text{konstans}.t}$

## Példa I.

### Kifejezés

$Kifejezés_0.t \rightarrow Kifejezés_1.t \pm Kifejezés_2.t @TípusEllenőrzés$   
 $Kifejezés.t \rightarrow \underline{\text{konstans}.t} @\text{KonstansKifejezés}$

- A Kifejezés szimbólum különböző előfordulásait indexeléssel különböztetjük meg.
- A  $@TípusEllenőrzés$  által jelzett szemantikus rutin:
  - $Kifejezés_0.t := \text{int}$
- A  $@\text{KonstansKifejezés}$  által jelzett szemantikus rutin:
  - $Kifejezés.t := \underline{\text{konstans}.t}$
- Szemantikus ellenőrzések:  
 $Kifejezés_1.t = \text{int}$  és  $Kifejezés_2.t = \text{int}$  teljesülnek-e?

7

Fordítóprogramok előadás (A, C, T szakirány)

Szemantikus elemzés (attribútum fordítási grammatikák)

7

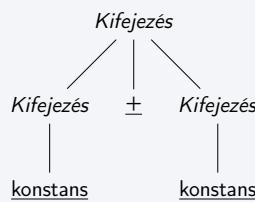
Fordítóprogramok előadás (A, C, T szakirány)

Szemantikus elemzés (attribútum fordítási grammatikák)

## A szemantikus információ terjedése I.

- Példaszöveg:  $1 + 2$

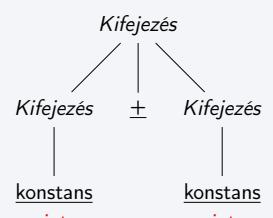
- Kezdetben a konstansok típusai ismertek.
- $Kifejezés.t := \underline{\text{konstans}.t}$
- $Kifejezés_0.t := \text{int}$
- Ellenőrzések:  
 $Kifejezés_1.t = \text{int} ?$   
 $Kifejezés_2.t = \text{int} ?$



## A szemantikus információ terjedése I.

- Példaszöveg:  $1 + 2$

- Kezdetben a konstansok típusai ismertek.
- $Kifejezés.t := \underline{\text{konstans}.t}$
- $Kifejezés_0.t := \text{int}$
- Ellenőrzések:  
 $Kifejezés_1.t = \text{int} ?$   
 $Kifejezés_2.t = \text{int} ?$



8

Fordítóprogramok előadás (A, C, T szakirány)

Szemantikus elemzés (attribútum fordítási grammatikák)

9

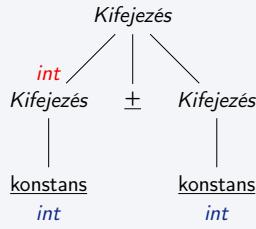
Fordítóprogramok előadás (A, C, T szakirány)

Szemantikus elemzés (attribútum fordítási grammatikák)

## A szemantikus információ terjedése I.

- Példaszöveg:  $1 + 2$

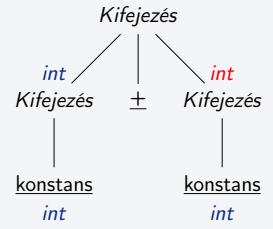
- Kezdetben a konstansok típusai ismertek.
- $Kifejezés.t := \underline{\text{konstans}.t}$
- $Kifejezés_0.t := \underline{\text{int}}$
- Ellenőrzések:  
 $Kifejezés_1.t = \underline{\text{int}} ?$   
 $Kifejezés_2.t = \underline{\text{int}} ?$



## A szemantikus információ terjedése I.

- Példaszöveg:  $1 + 2$

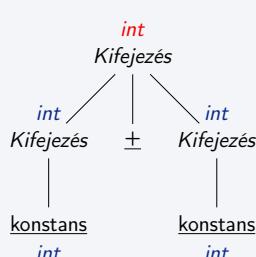
- Kezdetben a konstansok típusai ismertek.
- $Kifejezés.t := \underline{\text{konstans}.t}$
- $Kifejezés_0.t := \underline{\text{int}}$
- Ellenőrzések:  
 $Kifejezés_1.t = \underline{\text{int}} ?$   
 $Kifejezés_2.t = \underline{\text{int}} ?$



## A szemantikus információ terjedése I.

- Példaszöveg:  $1 + 2$

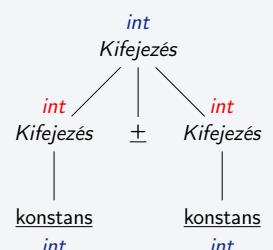
- Kezdetben a konstansok típusai ismertek.
- $Kifejezés.t := \underline{\text{konstans}.t}$
- $Kifejezés_0.t := \underline{\text{int}}$
- Ellenőrzések:  
 $Kifejezés_1.t = \underline{\text{int}} ?$   
 $Kifejezés_2.t = \underline{\text{int}} ?$



## A szemantikus információ terjedése I.

- Példaszöveg:  $1 + 2$

- Kezdetben a konstansok típusai ismertek.
- $Kifejezés.t := \underline{\text{konstans}.t}$
- $Kifejezés_0.t := \underline{\text{int}}$
- Ellenőrzések:  
 $Kifejezés_1.t = \underline{\text{int}} ?$   
 $Kifejezés_2.t = \underline{\text{int}} ?$



## Példa II.

### Deklarációs lista

Deklaráció  $\rightarrow$  típusnév.t Változólista.t @Beállít-1  
 Változólista.t  $\rightarrow$  változó.t Folytatás.t @Beállít-2  
 Folytatás0.t  $\rightarrow$  vége | vessző változó.t Folytatás1.t @Beállít-3

## Példa II.

### Deklarációs lista

Deklaráció  $\rightarrow$  típusnév.t Változólista.t @Beállít-1  
 Változólista.t  $\rightarrow$  változó.t Folytatás.t @Beállít-2  
 Folytatás0.t  $\rightarrow$  vége | vessző változó.t Folytatás1.t @Beállít-3

- Kezdetben a típusnév.t attribútum értéke ismert egyedül.

## Példa II.

### Deklarációs lista

Deklaráció → típusnév.t Változólista.t @Beállít-1  
 Változólista.t → változó.t Folytatás.t @Beállít-2  
 Folytatás<sub>0</sub>.t → vége | vessző változó.t Folytatás<sub>1</sub>.t @Beállít-3

- Kezdetben a típusnév.t attribútum értéke ismert egyedül.
- @Beállít-1
  - Változólista.t := típusnév.t

## Példa II.

### Deklarációs lista

Deklaráció → típusnév.t Változólista.t @Beállít-1  
 Változólista.t → változó.t Folytatás.t @Beállít-2  
 Folytatás<sub>0</sub>.t → vége | vessző változó.t Folytatás<sub>1</sub>.t @Beállít-3

- Kezdetben a típusnév.t attribútum értéke ismert egyedül.
- @Beállít-1
  - Változólista.t := típusnév.t
- @Beállít-2
  - változó.t := Változólista.t és bejegyzés a szimbólumtáblába
  - Folytatás.t := Változólista.t

## Példa II.

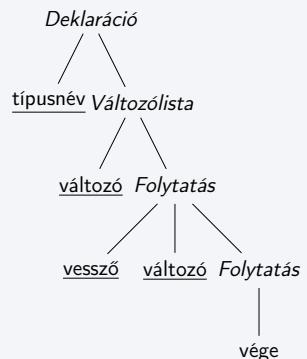
### Deklarációs lista

Deklaráció → típusnév.t Változólista.t @Beállít-1  
 Változólista.t → változó.t Folytatás.t @Beállít-2  
 Folytatás<sub>0</sub>.t → vége | vessző változó.t Folytatás<sub>1</sub>.t @Beállít-3

- Kezdetben a típusnév.t attribútum értéke ismert egyedül.
- @Beállít-1
  - Változólista.t := típusnév.t
- @Beállít-2
  - változó.t := Változólista.t és bejegyzés a szimbólumtáblába
  - Folytatás.t := Változólista.t
- @Beállít-3
  - változó.t := Folytatás<sub>0</sub>.t és bejegyzés a szimbólumtáblába
  - Folytatás<sub>1</sub>.t := Folytatás<sub>0</sub>.t

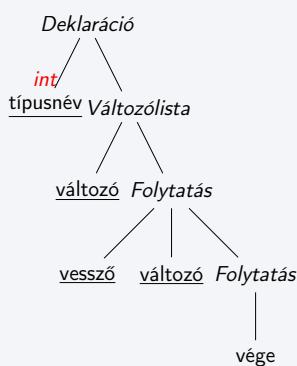
## A szemantikus információ terjedése II.

- Példaszöveg: int a,b;



## A szemantikus információ terjedése II.

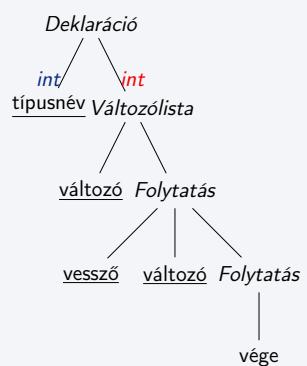
- Példaszöveg: int a,b;
- Kezdetben a típusnév.t attribútum értéke ismert.
- Változólista.t := típusnév.t
- változó.t := Változólista.t
- Folytatás.t := Változólista.t
- változó.t := Folytatás<sub>0</sub>.t
- Folytatás<sub>1</sub>.t := Folytatás<sub>0</sub>.t



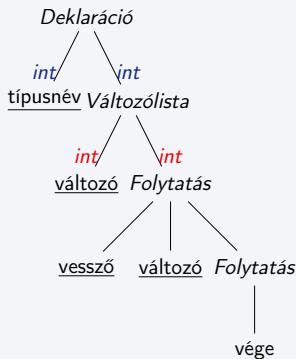
## A szemantikus információ terjedése II.

- Példaszöveg: int a,b;

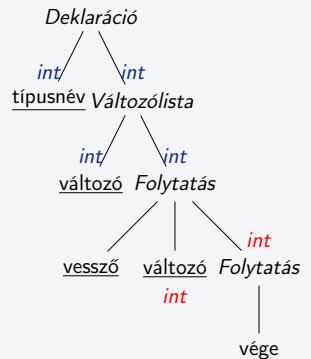
- Kezdetben a típusnév.t attribútum értéke ismert.
- Változólista.t := típusnév.t
- változó.t := Változólista.t
- Folytatás.t := Változólista.t
- változó.t := Folytatás<sub>0</sub>.t
- Folytatás<sub>1</sub>.t := Folytatás<sub>0</sub>.t



- Példaszöveg: int a,b;
  - Kezdetben a típusnév.t attribútum értéke ismert.
  - Változólista.t := típusnév.t
  - változó.t := Változólista.t
  - Folytatás.t := Változólista.t
  - változó.t := Folytatásq.t
  - Folytatás1.t := Folytatásq.t



- Példaszöveg: int a,b;
  - Kezdetben a típusnév.t attribútum értéke ismert.
  - Változólista.t := típusnév.t
  - változó.t := Változólista.t
  - Folytatás.t := Változólista.t
  - változó.t := Folytatás0.t
  - Folytatás1.t := Folytatás0.t



## Attribútum fajták

- Szintetizált attribútum:  
A helyettesítési szabály *bal oldalán* áll abban a szabályban, amelyikhez az őt kiszámoló szemantikus rutin tartozik.
    - például:  
szabály:  $Kifejezés_0.t \rightarrow Kifejezés_1.t \pm Kifejezés_2.t$   
semantikus rutin:  $Kifejezés_0.t := \text{int}$
    - Az információt a szintaxisfában *aljáról felfelé* közvetíti.

## Attribútum fajták

- **Örökítő attribútum:**  
A helyettesítési szabály jobb oldalán áll abban a szabályban, amelyikhez az őt kiszámoló szemantikus rutin tartozik.
    - például:  
szabály: Változólista.t → változó.t *Folytatás.t*  
semantikus rutin: változó.t := Változólista.t
    - Az információt a szintaxisfában *felülről lefelé* közvetíti.

Attribútum fajták

- **Kitüntetett szintetizált attribútum:**  
Olyan attribútumok, amelyek terminális szimbólumokhoz tartoznak és kiszámításukhoz nem használunk fel más attribútumokat.
    - például:  
szabály: *Kifejezés*. $t \rightarrow \text{konstans}.$ *t*
    - Az információt általában a lexikális elemző szolgáltatja

Attribútum fordítási grammatika (ATG)

- Egészítsük ki egy fordítási grammatika szimbólumait *attribútumokkal*, a szabályokat *feltételekkel*, valamint rendeljünk *szemantikus rutinokat* az akciószimbólumokhoz a következő szabályok betartásával:

## Attribútum fordítási gramatika (ATG)

- Egészítük ki egy fordítási gramatika szimbólumait *attribútumokkal*, a szabályokat *feltételekkel*, valamint rendeljünk *szemantikus rutinokat* az akciósimbólumokhoz a következő szabályok betartásával:
  - Egy adott szabályhoz tartozó feltételek csak a szabályban előforduló attribútumoktól függhetnek.  
(Ha egy feltétel nem teljesül, akkor szemantikus hibát kell jelezni!)

## Attribútum fordítási gramatika (ATG)

- Egészítük ki egy fordítási gramatika szimbólumait *attribútumokkal*, a szabályokat *feltételekkel*, valamint rendeljünk *szemantikus rutinokat* az akciósimbólumokhoz a következő szabályok betartásával:
  - 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 akciósimbólum tartozik.

23 Fordítóprogramok előadás (A, C, T szakirány)

Szemantikus elemzés (attribútum fordítási gramatikák)

23 Fordítóprogramok előadás (A, C, T szakirány) Szemantikus elemzés (attribútum fordítási gramatikák)

## Attribútum fordítási gramatika (ATG)

- Egészítük ki egy fordítási gramatika szimbólumait *attribútumokkal*, a szabályokat *feltételekkel*, valamint rendeljünk *szemantikus rutinokat* az akciósimbólumokhoz a következő szabályok betartásával:
  - 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 akciósimbólum tartozik.
  - Minden szintaxisfában minden attribútumértéket pontosan egy szemantikus rutin határozhat meg.

Így **attribútum fordítási gramatikát** kapunk.

## Kiszámíthatóság

- Nem minden attribútum fordítási gramatikában lehet kiszámolni az összes attribútum értékét.
  - Például ha  $X.a$  kiszámításához szükség van  $Y.b$ -re és  $Y.b$  kiszámításához szükség van  $X.a$ -ra...

### Definíció: Jól definiált attribútum fordítási gramatika

Olyan attribútum fordítási gramatika, amelyre igaz, hogy a gramatika által definiált nyelv mondataihoz tartozó minden szintaxisfában minden attribútum értéke egyértelműen kiszámítható.

23 Fordítóprogramok előadás (A, C, T szakirány)

Szemantikus elemzés (attribútum fordítási gramatikák)

24 Fordítóprogramok előadás (A, C, T szakirány) Szemantikus elemzés (attribútum fordítási gramatikák)

## Direkt függőségek

### Definíció: Direkt attribútumfüggőségek

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 gráfban* ábrázolhatók.

## Direkt függőségek

### Definíció: Direkt attribútumfüggőségek

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 gráfban* ábrázolhatók.

Jól definiált attribútum fordítási gramatikákhoz tartozó szintaxisfák függőségi gráfjában biztosan *nincsenek körök!*

25 Fordítóprogramok előadás (A, C, T szakirány)

Szemantikus elemzés (attribútum fordítási gramatikák)

25 Fordítóprogramok előadás (A, C, T szakirány) Szemantikus elemzés (attribútum fordítási gramatikák)

## Egy attribútumkiértékelő algoritmus

Nemdeterminisztikus algoritmus:

- a szintaxisfa minden attribútumához egy folyamatot rendelünk
- a folyamat 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

## Egy attribútumkiértékelő algoritmus

Nemdeterminisztikus algoritmus:

- a szintaxisfa minden attribútumához egy folyamatot rendelünk
- a folyamat 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 grammaтика jól definiált, akkor ez a program biztosan terminál és kiszámítja az összes attribútumértéket.

De ennél hatékonyabb algoritmust szeretnénk...

## Particionált ATG

- A kiértékelés sorrendjét általában nem lehet az attribútum-fordítási grammaтикаból meghatározni. (Függet a konkrét szintaxisfától.)

## Particionált ATG

- A kiértékelés sorrendjét általában nem lehet az attribútum-fordítási grammaтикаból meghatározni. (Függet a konkrét szintaxisfától.)
- **Particionált attribútum fordítási grammaтика:** minden  $X$  szimbólum attribútumai szétoszthatók az  $A_1^X, A_2^X, \dots, A_{m_X}^X$  diszjunkt halmazokba (*partíciókba*) úgy, hogy
  - $A_{m_X}^X, A_{m_X-2}^X, \dots$  halmazokban csak szintetizált attribútumok
  - $A_{m_X-1}^X, A_{m_X-3}^X, \dots$  halmazokban csak örökolt attribútumok vannak, és
  - minden szintaxisfában az  $X$  attribútumai a halmazok növekvő sorrendjében meghatározhatók.
- Ha ismerjük a particiókat, akkor csak az attribútum fordítási grammaтика szabályai alapján tudunk olyan programot írni, ami a megfelelő sorrendben kiszámítja a szintaxisfában lévő összes attribútumot.

## Rendezett ATG

- A particiókat általában nem könnyű meghatározni.

## Rendezett ATG

- A particiókat általában nem könnyű meghatározni.
- **Rendezett attribútum fordítási grammaтика:** egyszerű algoritmussal meghatározhatók a particiók.
  - a halmazokat fordított sorrendben határozzuk meg
  - felváltva szintetizált és örökolt attribútumokat teszünk a halmazokba (utolsóba szintetizált, utolsó előtérbe örökolt, stb.)
  - mindegyikbe azokat az attribútumokat tessük, amikre csak olyan attribútumok kiszámításához van szükség, amelyeket már beosztottunk valamelyik halmazba
- Ha ezzel az algoritmussal megfelelő particiókat kapunk, akkor hívjuk a nyelvtant *rendezett attribútum fordítási grammaтикаnak*.

- Olyan attribútum fordítási grammaika, amelyben kizárolag szintetizált attribútumok vannak.
- A szemantikus információ a szintaxisfában a levelektől a gyökér felé terjed.
- Jól illeszthető az alulról felfelé elemzésekhez!

2

int

- léptetés
- kitüntetett szintetizált attribútum: konstans típusa

$$\begin{array}{c} \text{int} \\ E \\ | \\ 2 \end{array}$$

- redukció  $E.t \rightarrow \underline{\text{konstans}}.t$  szabály alapján
- szemantikus rutin:  $E.t := \underline{\text{konstans}}.t$

$$\begin{array}{c} \text{int} \\ E \\ | \\ 2 \quad + \\ \text{int} \end{array}$$

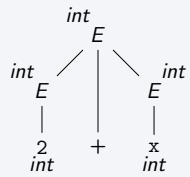
- léptetés

$$\begin{array}{c} \text{int} \\ E \\ | \\ 2 \quad + \quad x \\ \text{int} \end{array}$$

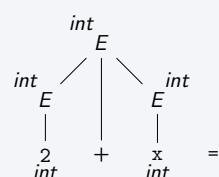
- léptetés
- kitüntetett szintetizált attribútum: változó típusa

$$\begin{array}{c} \text{int} \quad \text{int} \\ E \quad E \\ | \quad | \\ 2 \quad x \\ \text{int} \end{array}$$

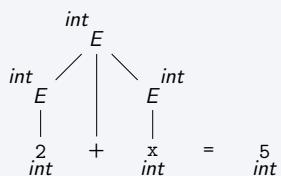
- redukció  $E.t \rightarrow \underline{\text{változó}}.t$  szabály alapján
- szemantikus rutin:  $E.t := \underline{\text{változó}}.t$



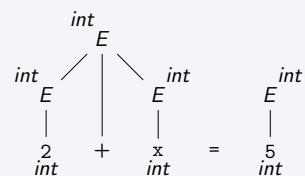
- redukció  $E_0.t \rightarrow E_1.t \pm E_2.t$  szabály alapján
- szemantikus rutin:  $E_0.t := \text{int}$
- ellenőrzés:  $E_1.t = \text{int}$  és  $E_2.t = \text{int}$  ?



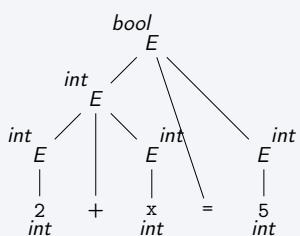
- léptetés



- léptetés
- kitüntetett szintetizált attribútum: konstans típusa



- redukció  $E.t \rightarrow \underline{\text{konstans}.t}$  szabály alapján
- szemantikus rutin:  $E.t := \underline{\text{konstans}.t}$



- redukció  $E_0.t \rightarrow E_1.t \equiv E_2.t$  szabály alapján
- szemantikus rutin:  $E_0.t := \text{bool}$
- ellenőrzés:  $E_1.t = E_2.t$  ?

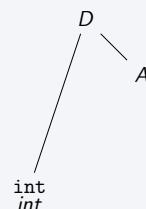
- redukció esetén a szabály jobb oldalának attribútumértékei már ismertek  
(hiszen nincsenek örökölt attribútumok)
- a szabályhoz rendelt szemantikus rutin feladata a baloldalon álló szimbólum szintetizált attribútumainak meghatározása

- esetenként a nyelvtan átalakítása segíthet kiküszöbölni az örökolt attribútumokat
- szimbólumtáblában vagy más globális változóban tárolható az örökítendő attribútumérték
- eltárolható az a részfa, ahol örökolt attribútumok vannak; mikor meghatározható az értékük, akkor be kell járni a részfát és pótolni a hiányzó attribútumokat; ez a részfa gyökeréhez tartozó szabály szemantikus rutinjában implementálható

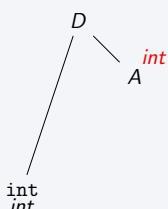
- Olyan attribútum fordítási grammatika, amelyben minden  $A \rightarrow X_1 X_2 \dots X_n$  szabályban az attribútumértékek az alábbi sorrendben meghatározhatók:
  - $A$  örökolt attribútumai
  - $X_1$  örökolt attribútumai
  - $X_1$  szintetizált attribútumai
  - $X_2$  örökolt attribútumai
  - $X_2$  szintetizált attribútumai
  - ...
  - $X_n$  örökolt attribútumai
  - $X_n$  szintetizált attribútumai
  - $A$  szintetizált attribútumai
- Jól illeszkedik a felülről lefelé elemzésekhez.

$D$

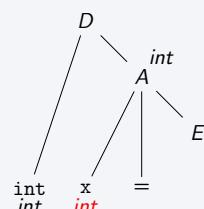
- $D$ -ből indulva felülről lefelé levezetünk egy deklarációt
- $D$ -nek most nincs örökolt attribútuma



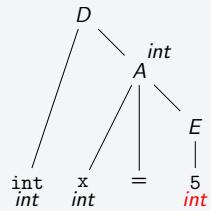
- $D \rightarrow \underline{\text{típusnév}.d} A.d$  szabály alkalmazása
- kitüntetett szintetizált attribútum: a típusnév által azonosított típus



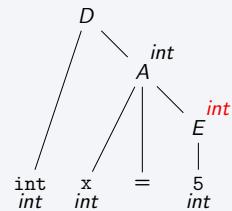
- szemantikus rutin:  $A.d := \underline{\text{típusnév}.d}$
- ez egy örökolt attribútum



- $A.d \rightarrow \underline{\text{változó}.d} \underline{=} E.t$  szabály alkalmazása
- szemantikus rutin:  $\underline{\text{változó}.d} := A.d$  (örökolt)



- $E.t \rightarrow \text{konstans}.t$  szabály alkalmazása
- kitüntetett szintetizált attribútum: konstans típusa



- szemantikus rutin:  $E.t := \text{konstans}.t$  (szintetizált)
- ellenőrzés:  $E.t = A.d$  ?

- a  $A \rightarrow X_1 \dots X_n$  szabályhoz tartozó eljárás
  - formális paraméterei legyenek az  $A$  örökölt attribútumai
  - visszatérési értéke az  $A$  szintetizált attribútumai
- az eljárások végrehajtása épp az L-ATG attribútumkiértékelési sorrendjét adja

## Gyakorlatias assembly bevezető

Fordítóprogramok előadás (A, C, T szakirány)

## Mi az assembly?

- programozási nyelvek egy csoportja
- gépközeli:
  - az adott processzor utasításai használhatóak
  - általában nincsenek programkonstrukciók, típusok, osztályok stb.
- a futtatható programban pontosan azok az utasítások lesznek, amit a programba írunk
  - lehet optimalizálni
  - lehet olyan trükköket használni, amit a magasabb szintű nyelvek nem engednek meg

## Sokfélé assembly van...

- különféle architektúrák (processzorok) utasításkészlete különbözik
  - van olyan architektúra, ahol gépi utasítás van külön egy bináris keresőfa kiegysúlyozására
  - Intel architektúrán ilyen nincs...
- egy architektúrán belül is lehet különböző szintaxisú assembly nyelvet definiálni
  - NASM assembly: `mov eax,0`
  - GAS assembly: `movl $0, %eax`

## Mit fogunk mi használni?

- Intel x86 architektúra (386, 486, Pentium, ...)
  - mindenki számára könnyen elérhető
- NASM assembly
  - tiszta, könnyen érthető a szintaxisa

## Assembly programok fordítása

- az assembly programok fordítóprogramja az *assembler*
- a magas szintű nyelvek fordítóprogramjaihoz képest:
  - az elemzés része egyszerűbb
    - általában nincsenek típusok, bonyolult szintaktikus elemek stb.
  - a kódgenerálás bonyolultabb
    - az assembler gépi kódra fordít
    - a többi fordítóprogram általában assemblyre vagy egy másik magasszintű nyelvre

## A NASM fordítóprogramja

### A fordítás lépései

```
$ ls  
io.c programom.asm  
$ nasm -f elf -o programom.o programom.asm  
$ ls  
io.c programom.asm programom.o  
$ gcc -o programom programom.o io.c  
$ ls  
io.c programom programom.asm programom.o
```

- nasm: fordítás (asszembblálás)
- gcc: szerkesztés (linkelés)  
(a C fordítóprogramját használjuk)
- io.o: egy object file, amit C-ben írtam és az ebben lévő függvényeket meghívjuk a majd programunkból

## Az asszembolás

```
$ nasm -f elf -o programom.o programom.asm
```

- **-f elf:** megadjuk az object fájl formátumát (elf)
- **-o programom.o:** az eredményként keletkező object fájl neve
- egyéb kapcsolók: nasm -h

## regiszterek

- kis méretű tárolóhelyek a processzoron
- külön nevük van: eax, ebx, ecx, ...

## memória

- itt tárolhatók a program által használt adatok
- itt tárolódik maga a program kódja is
- címzéssel lehet hivatkozni rá: [tömb+4]

## Regiszterek adattároláshoz, számoláshoz

- eax: „*accumulator*” - elsősorban aritmetikai számításokhoz
- ebx: „*base*” - ebben szokás tömbök, rekordok kezdőcímét tárolni
- ecx: „*counter*” - számlálókat szokás tárolni benne (pl. for jellegű ciklusokhoz)
- edx: „*data*” - egyéb adatok tárolása; aritmetikai számításokhoz segédregiszter
- esi: „*source index*” - sztringmásolásnál a forrás címe
- edi: „*destination index*” - sztringmásolásnál a cél címe

Ezek egymással felcserélhetők, de célszerű a megadott feladatokra használni őket, ha lehet.

## Regiszterek a program vermének kezeléséhez

- esp: „*stack pointer*” - a program verménék a tetejét tartja nyilván
- ebp: „*base pointer*” - az aktuális alprogramhoz tartozó verem-részt tartja nyilván

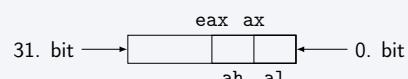
Ezeket csak a veremkezeléshez érdemes használni. Meggondolatlan elállításuk hibához vezethet!

## Adminisztratív regiszterek

- eip: „*instruction pointer*” - a következő végrehajtandó utasítás címe
- eflags: *jelzőbitek* - pl. összehasonlítások eredményeinek tárolása

Ezeket közvetlenül nem tudjuk olvasni és írni.

## A regiszterek felépítése



- eax 32 bitból áll; két része van:
  - a „felső” 16 bitnek nincs külön neve
  - az „alsó” 16 bit: ax; ennek részei:
    - a „felső” bájt: ah
    - az „alsó” bájt: al
- ugyanígy épülnek fel: ebx, ecx, edx
- esi, edi, esp és ebp regiszterekben csak az alsó 16 bitnek van külön neve: si, di, sp, bp

## Az adatterület megadása

- a programban helyet foglalunk az adatainak:
  - ha csak helyet akarunk foglalni nekik: .bss szakasz
  - ha kezdeti értéket is akarunk adni: .data szakasz

### Adatterület megadása

```
section .bss
A: resb 4      ; négy bájt lefoglalása
B: resb 2      ; két bájt lefoglalása

section .data
C: db 1,2,3,4  ; négyeszer 1 bájt
       ; az 1, 2, 3, 4 értékekkel
D: dd 42       ; egyszer 4 bájt a 42 értékkel
```

## Az adatterület megadása

### Adatterület megadása

```
section .bss
A: resb 4      ; négy bájt lefoglalása
B: resb 2      ; két bájt lefoglalása

section .data
C: db 1,2,3,4  ; négyeszer 1 bájt
       ; az 1, 2, 3, 4 értékekkel
D: dd 42       ; egyszer 4 bájt a 42 értékkel
```

A	A+1	A+2	A+3	B	B+1		
?	?	?	?	?	?		
C	C+1	C+2	C+3	D	D+1	D+2	D+3
1	2	3	4	42	0	0	0

## Adatterület megadása

- resb: „reserve byte” - egy bájt
- resw: „reserve word” - két bájt (egy gépi szó)
- resd: „reserve double word” - négy bájt (egy gépi duplaszó)
- db: „define byte” - egy bájt kezdőértékkel
- dw: „define word” - két bájt kezdőértékkel
- dd: „define double word” - négy bájt kezdőértékkel

## Memória hivatkozás

C	C+1	C+2	C+3	D	D+1	D+2	D+3
1	2	3	4	42	0	0	0

- byte [C]: 1 bájt a C címtől  
értéke: 1
- byte [C+1]: 1 bájt a C+1 címtől  
értéke: 2
- word [C+1]: 2 bájt a C+1 címtől  
értéke:  $256^1 \cdot 3 + 256^0 \cdot 2 = 770$
- dword [D]: 4 bájt a D címtől  
értéke:  $256^3 \cdot 0 + 256^2 \cdot 0 + 256^1 \cdot 0 + 256^0 \cdot 42 = 42$

## Adatmozgatás: mov

- mov eax,ebx  
ebx értékét eax-be tölti
- mov eax,dword [D]  
a D címtől kezdődő 4 bájt értékét eax-be tölti  
(a dword itt el is hagyható, mert eax-ból kiderül, hogy 4 bájtot kell mozogni)
- mov dword [D],eax  
eax értékét a D-től kezdődő 4 bájtba tölti
- mov al,bh  
bh értékét al-be tölti
- mov byte [C],al  
al értékét a C című bájtba tölti
- mov byte [C],byte [D]  
**Hibás! Nincs memoriából memóriába mozgatás.**

## Aritmetikai műveletek: inc, dec, add, sub

- inc eax  
az eax értékét megnöveli eggyel  
(lehet memória hivatkozás is az operandus)
- dec word [C]  
a C címtől kezdődő 2 bájt értékét csökkenti eggyel  
(lehet regiszter is az operandus)
- add eax,dword [D]  
eax-hez adj a D címtől kezdődő 4 bájt értékét
- sub edx,ecx  
levonja edx-ból ecx-et
- Az add és a sub utasításokra ugyanaz a szabály, mint a mov-ra:  
**legfeljebb az egyik operandus lehet memória hivatkozás.**

## Aritmetikai műveletek: mul, div

### mul ebx

Megszorozza eax értékét ebx értékével.

Az eredmény:

- edx-be kerülnek a nagy helyértékű bajtok
- eax-be az alacsony helyértékűek

### div ebx

Elosztja a  $256^4$  edx+eax értéket ebx értékével.

Az eredmény:

- eax-be kerül a hányados
- edx-be a maradék

- Ha előjeles egész számokkal dolgozunk, akkor az imul és idiv utasításokat kell használni.

## Logikai műveletek: and, or, xor, not

### and al, bl

Bitenkénti „és” művelet.

### Bitenkénti "és"

Ha előtte al= 12 = 0000 1100<sub>2</sub> és bl= 6 = 0000 0110<sub>2</sub>, akkor utána al= 0000 0100<sub>2</sub> = 4.

### or eax,dword [C]

Bitenkénti „vagy” művelet.

### xor word [D],bx

Bitenkénti „kizáró vagy” művelet.

### not bl

Bitenkénti „nem” művelet.

### Bitenkénti "nem"

Ha előtte bl= 9 = 0000 1001<sub>2</sub>, akkor utána bl= 1111 0110<sub>2</sub> = 246.

## Igaz, hamis értékek

A magasszintű programozási nyelvek *logikai* (bool) típusának egy lehetséges megvalósítása:

- 1 bájton tároljuk
- hamis érték: 0
- igaz érték: 1
- logikai és: and utasítás
- logikai vagy: or utasítás
- tagadás: not és and 1

## Ugró utasítás: jmp

### Végtelen ciklus ugró utasítással

eleje: inc ecx  
jmp eleje

### Utasítások átugrása

```
mov ecx,0  
jmp vege  
add ecx,ebx  
inc ecx  
vege: mov edx,ecx
```

## Feltételesek ugró utasítások

- cmp: „compare” - két érték összehasonlítása
- feltételesek ugró utasítások: „ugorj a megadott címkére, ha ...”

Ha eax=0, ugorj az A címkére!

```
cmp eax,0  
je A
```

Ha eax≠0, ugorj az A címkére!

```
cmp eax,0  
jne A
```

Ha eax<ebx, ugorj az A címkére!

```
cmp eax,ebx  
jb A
```

Ha eax>ebx, ugorj az A címkére!

```
cmp eax,ebx  
ja A
```

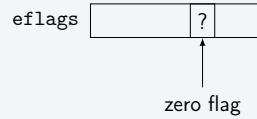
## Feltételesek ugró utasítások

- je: „equal” - ugorj, ha egyenlő
- jne: „not equal” - ugorj, ha nem egyenlő
- jb: „below” - ugorj, ha kisebb  
≡ jnae: „not above or equal” - nem nagyobb egyenlő
- ja: „above” - ugorj, ha nagyobb  
≡ jnbe: „not below or equal” - nem kisebb egyenlő
- jnb: „not below” - nem kisebb  
≡ jae: „above or equal” - nagyobb egyenlő
- jna: „not above” - nem nagyobb  
≡ jbe: „below or equal” - kisebb egyenlő
- Ha előjeles egészszámokkal számolunk:  
jl („less”), jg („greater”), jnl, jng, jle, jge, ...

## Feltételes ugró utasítások működése

```
mov eax,10
cmp eax,10
je A
eax [ ] ?
```

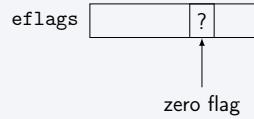
- Az eax regiszter értéke 10 lesz.
- Mivel eax=10, az összehasonlítás beállítja a zero flag-et 1-re.
- A je akkor ugrik, ha a zero flag=1.



## Feltételes ugró utasítások működése

```
mov eax,10
cmp eax,10
je A
eax [ ] 10
```

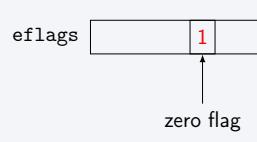
- Az eax regiszter értéke 10 lesz.
- Mivel eax=10, az összehasonlítás beállítja a zero flag-et 1-re.
- A je akkor ugrik, ha a zero flag=1.



## Feltételes ugró utasítások működése

```
mov eax,10
cmp eax,10
je A
eax [ ] 10
```

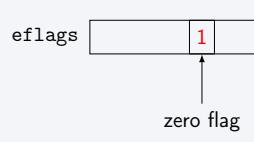
- Az eax regiszter értéke 10 lesz.
- Mivel eax=10, az összehasonlítás beállítja a zero flag-et 1-re.
- A je akkor ugrik, ha a zero flag=1.



## Feltételes ugró utasítások működése

```
mov eax,10
cmp eax,10
je A
eax [ ] 10
```

- Az eax regiszter értéke 10 lesz.
- Mivel eax=10, az összehasonlítás beállítja a zero flag-et 1-re.
- A je akkor ugrik, ha a zero flag=1.



## Feltételes elágazás

Ha eax<10, akkor eax:=eax+1.

```
cmp eax,10
jnb vege
inc eax
vege:
```

Ha eax<10, akkor eax:=eax+1, különben eax:=eax-1.

```
cmp eax,10
jnb hamis_ag
inc eax
jmp vege
hamis_ag: dec eax
vege:
```

## Ciklus

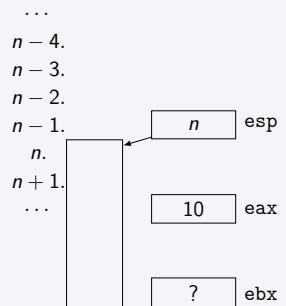
Amíg eax<10, eax:=eax+1.

```
eleje:    cmp eax,10
          jnb vege
          inc eax
          jmp eleje
vege:
```

- Minden futó programhoz hozzá van rendelve egy saját memóriaterület, a program verme.
- Ebben lehet tárolni a lokális változókat.
- Ebben adjuk át a paramétereket alprogramhívás esetén.
- Ebbe kerül bele a visszatérési cím, azaz, hogy hova kell visszatérni az alprogram végén.

```
push eax
pop ebx
```

- eax értéke bekerül a verembe (4 bájt)
- a verem tetején lévő (4 bájtos) érték bemásolódik ebx-be, a veremmutató visszaáll



```
push eax
pop ebx
```

- eax értéke bekerül a verembe (4 bájt)
- a verem tetején lévő (4 bájtos) érték bemásolódik ebx-be, a veremmutató visszaáll

The diagram illustrates the state of the stack after a push eax instruction. The stack grows downwards. At the top of the stack, there is a box labeled '10'. An arrow points from a box labeled 'n - 4' to the stack. Below the stack are boxes labeled 'esp', 'eax', and '?'.

```
push eax
pop ebx
```

- eax értéke bekerül a verembe (4 bájt)
  - a verem tetején lévő (4 bájtos) érték bemásolódik ebx-be, a veremmutató visszaáll
- 
- The diagram illustrates the state of the stack after a push eax instruction. The stack grows downwards. At the top of the stack, there is a box labeled '10'. An arrow points from a box labeled 'n' to the stack. Below the stack are boxes labeled 'esp', 'eax', and '10'.

- Csak 2 vagy 4 bájtot lehet a verembe tenni (vagy kivenni).
  - Tehát pl. **push ah** hibás!
- A verem tetején lévő adat pop művelet esetén nem törlődik a memoriából, csak lemosolódik, és a veremmutató megváltozik.
- Ha csak a veremmutató visszaállítása a cél, akkor lehet
 

```
pop ebx
      helyett
      add esp,4
```

 utasítást használni.
  - Ez visszaállítja a veremmutatót, de nem másolja seholra a verem tetején lévő értéket.

**A kiir eljárás hívása**

```
push eax ; Betesszük a verembe a paramétert.
call kiir ; Meghívjuk az eljárást.
           ; Az eljárás a veremből használhatja
           ; az átadott paramétert.
add esp,4 ; Visszaállítjuk a veremmutatót
           ; a push előtti állapotba.
```

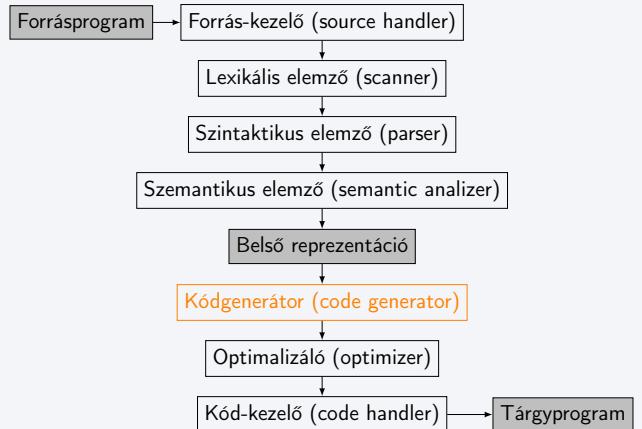
- Szabályozni kell, hogy a *hívó* és a *hívott* kód részlet pontosan hogyan kommunikál egymással:
  - Milyen sorrendben kerülnek a verembe a paraméterek?
  - Kiszedi-e az alprogram a paramétereket a veremből, vagy ez a hívó kód részlet dolga?
  - Hol kapja meg a hívó a függvények visszatérési értékét?
  - Melyik regisztereket változtathatja meg az alprogram?
- A C nyelv hívási konvenciói:
  - A paramétereket **fordított sorrendben** kell a verembe tenni, azaz az első paraméter lesz legfelül.
  - Az alprogram **bennt hagyja a paramétereket** a veremben.
  - A visszatérési érték az **eax** regiszterbe kerül.
  - Az alprogram csak az **eax, edx, ecx** regisztereket módosíthatja. (*Intel ABI* konvenció.)

## Kódgenerálás I. (kifejezések és vezérlési szerkezetek)

Fordítóprogramok előadás (A,C,T szakirány)

2008. őszi félév

## A kódgenerálás helye a fordítási folyamatban



## A kódgenerálás feladata

- a szintaktikusan és szemantikusan elemzett programot tárgykóddá alakítja
- a valóságban általában szorosan összekapcsolódik a szemantikus elemzéssel
- bonyolultabb esetekben:
  - ① a forrásprogram egyszerű utasításokká alakítása
  - ② gépfüggetlen kódoptimalizálás
  - ③ az egyszerű utasítások továbbfordítása assemblyre vagy gépi kódra
  - ④ gépfüggő kódoptimalizálás

## Ebben az előadásban...

- Intel 8086-os architektúrára,
- NASM assembly nyelvre fogjuk fordítani
- az imperatív programozási nyelvek leggyakrabban előforduló konstrukcióit.

## Értékadások fordítása

- alakja:  
*assignment* → *variable assignment operator expression*
- generálandó kód:

### Értékadást megalosító kód

- ① a kifejezést az eax regiszterbe kiértékelő kód
- ② mov [Változó],eax

- megjegyzések:
  - a kifejezések kiértékelésével később foglalkozunk
  - Változó az értékadás bal oldalán szereplő változó címkéje
  - bonyolultabb adatszerkezetek lemásolását külön eljárás végzi (*másoló konstruktör*)

## Összetett típusokra vontakozó értékadások

- bizonyos típusokra (pl. rekordok, tömbök) könnyen lehet alapértelmezett másoló eljárást készíteni:
  - mezőkénti / elemenkénti másolás
- láncolt adatszerkezetek másolási stratégiái:
  - rekurzívan minden lemásolunk
  - csak a legfelső szinten történik másolás, a mutatók által hivatkozott memóriaterületek közösek lesznek
  - megadjuk a lehetőséget a programozónak, hogy maga írja meg a másoló konstruktort

## Egy ágú elágazás fordítása

- alakja: *statement → if condition then program end*

### Egy ágú elágazás kódja

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

- a feltételek kiértékelésével később foglalkozunk
- itt a *logikai igaz* értéket az 1 reprezentálja
- `jne` Vége utasítás: maximum 128 bájt távolságra tud ugrani; az then-ág ennél hosszabb is lehet ⇒ `jne near` Vége
- a programban több elágazás is lehet
- ⇒ minden esetben **egyedi címkéket** kell generálni!

## Ha a feltételes ugrás csak rövid lehet

- ha a feltételes ugrásból kihagyjuk a `near-t`, akkor csak rövidet tud ugrani
- a `jmp` (feltétel nélküli) ugrás viszont alapértelmezetten hosszú ugrás
- így egy alternatív megoldás:

### Egy ágú elágazás kódja - másik megoldás

- a feltételt az al regiszterbe kiértékelő kód
- `cmp al,1`
- `je Then`
- `jmp Vége`
- `Then:` a then-ág programjának kódja
- Vége:

- ez a trükk a többi programkonstrukció esetén is alkalmazható

## Címkék generálása

- Egyedi címkékre van szükség:
  - elágazások
  - ciklusok
  - változó- és alprogramdefiníciók fordításakor.
- Egy lehetséges megoldás:
  - Lab1, Lab2, Lab3, ...
  - Egy számlálót tartunk fent, amit minden alkalommal inkrementálunk.
  - Új címke: a számláló értékét kell a végére koncatenálni:

```
stringstream ss;
ss << "Lab" << szamlalo++;
string cimkenev = ss.str();
```

## Több ágú elágazás alakja

```
statement →
if condition1 then program1
elseif condition2 then program2
...
elseif conditionn then programn
else programn+1 end
```

## Több ágú elágazás kódja

- 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:

## A switch-case utasítás fordítása

- alakja:

```
statement → switch variable
case value1 : program1
...
case valuen : programn
```
- a generálandó kód hasonló egy több ágú elágazáshoz
- mivel itt a feltételekre megszorítások vannak, lehet hatékonyabb a kiértékelés
  - csak `variable == value` alakúak a feltételek, ahol a `value` konstans érték
- a switch-case másként működik az egyes nyelvekben:
  - Ada stílus: csak egy ág hajtódkik végre
  - C stílus: az első teljesülő ágtól kezdve az összes végrehajtódkik (hacsak nem használunk `break` utasítást az ágak végén)

## A switch-case utasítás fordítása (Ada stílus)

```
❶ cmp [Változó],Érték_1  
❷ jne near Feltétel_2  
❸ első ág programjának kódja  
❹ jmp Vége  
❺ Feltétel_2: cmp [Változó],Érték_2  
❻ ...  
❼ Feltétel_n: cmp [Változó],Érték_n  
➋ jne near Vége  
⩿ n-edik ág programjának kódja  
⩾ Vége:
```

## A switch-case utasítás fordítása (C stílus)

```
❶ cmp [Változó],Érték_1  
❷ je near Program_1  
❸ cmp [Változó],Érték_2  
❹ je near Program_2  
❺ ...  
❻ cmp [Változó],Érték_n  
❼ je near Program_n  
⩿ jmp Vége  
⩾ Program_1: az 1. ág programjának kódja  
⩾ ...  
⩾ Program_n: az n-edik ág programjának kódja  
⩾ Vége:
```

## Elöl tesztelő ciklus fordítása

- alakja: *statement → while condition program end*
- generálandó kód:

### Elöl tesztelő ciklus kódja

```
❶ Eleje: a ciklusfeltétel kiértékelése az al regiszterbe  
❷ cmp al,1  
❸ jne near Vége  
❹ a ciklusmag programjának kódja  
❺ jmp Eleje  
❻ Vége:
```

## Hátul tesztelő ciklus fordítása

- alakja: *statement → loop program while condition*
- generálandó kód:

### Hátul tesztelő ciklus kódja

```
❶ Eleje: a ciklusmag programjának kódja  
❷ a ciklusfeltétel kiértékelése az al regiszterbe  
❸ cmp al,1  
❹ je near Eleje
```

## For ciklus fordítása

- alakja:  
*statement → for variable from value<sub>1</sub> to value<sub>2</sub> program end*
- hasonlítható a while ciklusok fordításához
  - hiszen minden for ciklus átalakítható while ciklussá

### For ciklus kódja

```
❶ a „from” érték kiszámítása a [Változó] memóriahezre  
❷ Eleje: a „to” érték kiszámítása az eax regiszterbe  
❸ cmp [Változó],eax  
❹ ja near Vége  
❺ a ciklusmag kódja  
❻ inc [Változó]  
❼ jmp Eleje  
⩿ Vége:
```

## Ciklusváltozó tárolása regiszterben

- hatékonyabb lehet a kód, ha a ciklusváltozót regiszterben tároljuk
- van processzor-szintű támogatás is erre: loop utasítás

### Példa a loop utasításra

```
mov ecx,10 ; 10-szer fogjuk végrehajtani  
Eleje:  
; ide kerül a ciklusmag  
loop Eleje ; csökkenti ecx-et,  
; ha még pozitív: visszaugrik,  
; ha nulla lett: továbblép
```

- Vigyázat:** lehet, hogy a ciklusmag elállítja az ecx regiszert!
  - vagy gondoskoni kell róla, hogy ne állítsa el
  - vagy körül kell venni a ciklusmagot:

**Az ecx regiszter eleme**

```
push ecx
; ciklusmag
pop ecx
```

- sokféle változó van: statikus, lokális, dinamikusan allokat...
- most a **statikus** változókkal foglalkozunk
  - globális változók
  - kifejezetten statikusnak deklarált változók  
(pl. C++ static kulcsszó)
- a statikus változó a .data vagy .bss szakaszban kapnak helyet
- a többi változóval a következő előadáson foglalkozunk

- kezdőérték nélkül: int x;

**Kezdőérték nélküli változódefinició fordítása**

```
section .bss
; a korábban definiált változók...
Lab12: resd 1 ; 1 x 4 bájtnyi terület
```

- kezdőértékkel: int x=5;

**Kezdőértékkel adott változódefinició fordítása**

```
section .data
; a korábban definiált változók...
Lab12: dd 5 ; 4 bájton tárolva az 5-ös érték
```

Minden változóhoz **új címkét** kell generálni és **fel kell jegyezni** a szimbólumtáblába a változó attribútumai közé!

**1. próbálkozás (*kifejezés<sub>1</sub>+kifejezés<sub>2</sub>*)**

```
; a 2. kifejezés kiértékelése eax-be
mov ebx,eax
; az 1. kifejezés kiértékelése eax-be
add eax,ebx
```

**Ez hibás!** Ha a részkifejezésekben is használjuk ebx-et, elállítjuk az értékét...

**Megoldás:**

**2. próbálkozás (*kifejezés<sub>1</sub>+kifejezés<sub>2</sub>*)**

```
; a 2. kifejezés kiértékelése eax-be
push eax
; az 1. kifejezés kiértékelése eax-be
pop ebx
add eax,ebx
```

- A példákban az eax regiszterbe értékeljük ki a kifejezéseket.
- 1. eset: egyetlen konstans értékből álló kifejezés (pl. 25)

**Konstans kiértékelése**

```
mov eax,25
```

- 2. eset: egyetlen változóból álló kifejezés (pl. x)

**Változó kiértékelése**

```
mov eax,[X] ; ahol X a változó címkeje
```

- 3. eset: összetett kifejezés
  - (pl. fibonacci(25) + factorial(x) )
  - általános megoldás: függvényhívásokat teszünk a kódba
  - a beépített függvényekhez (+,-,\*,/,&&,||,!,...) lehet hatékonyabban is

- cél: az al regiszterbe kiértékelni a logikai kifejezés eredményét
- 1. eset: <, >, =, ... operátorok

***kifejezés<sub>1</sub> < kifejezés<sub>2</sub>* kiértékelése**

```
; a 2. kifejezés kiértékelése az eax regiszterbe
push eax
; az 1. kifejezés kiértékelése az eax regiszterbe
pop ebx
cmp eax,ebx
jb Kisebb
mov al,0 ; hamis
jmp Vége
Kisebb:
mov al,1 ; igaz
Vége:
```

## Logikai kifejezések fordítása

- 2. eset: *és*, *vagy*, *nem*, *kizárvagy*, ... műveletek (hasonlóan a +, -, ... kiértékeléséhez)

### *kifejezés<sub>1</sub>* és *kifejezés<sub>2</sub>* kiértékelése

```
; a 2. kifejezés kiértékelése az al regiszterbe  
push ax      ; nem lehet 1 bájtot a verembe tenni!  
; az 1. kifejezés kiértékelése az al regiszterbe  
pop bx      ; bx-nek a bl részében van,  
                ; ami nekünk fontos  
and al,bl
```

- de ha az 1. hamis, akkor már nem is kell kiértékelni a 2-dikat...

25

Fordítóprogramok előadás (A,C,T szakirány)

Kódgenerálás I. (kifejezések és vezérlési szerkezetek)

## Rövidzás logikai operátorok

- Ha *kifejezés<sub>1</sub>* && *kifejezés<sub>2</sub>* kifejezésben az első hamis, akkor a másodikat garantáltan nem értékeli ki.
  - fontos lehet: (a != 0) && (c == b / a)

### *kifejezés<sub>1</sub>* és *kifejezés<sub>2</sub>* kiértékelése

```
; az 1. kifejezés kiértékelése az al regiszterbe  
cmp al,0  
je Vége  
push ax  
; a 2. kifejezés kiértékelése az al regiszterbe  
mov bl,al  
pop ax  
and al,bl  
Vége:
```

26

Fordítóprogramok előadás (A,C,T szakirány)

Kódgenerálás I. (kifejezések és vezérlési szerkezetek)

## Kódgenerálás és *S* – ATG

- Emlékeztető: *S*-attribútum fordítási grammatika
  - csak szintetizált („alulról felfelé” terjedő) attribútumok
  - jól illeszkedik az alulról felfelé elemzésekhez (*bison++*)

27

Fordítóprogramok előadás (A,C,T szakirány)

Kódgenerálás I. (kifejezések és vezérlési szerkezetek)

## Kódgenerálás és *S* – ATG

- Emlékeztető: *S*-attribútum fordítási grammatica
  - csak szintetizált („alulról felfelé” terjedő) attribútumok
  - jól illeszkedik az alulról felfelé elemzésekhez (*bison++*)
- Az eddig látott konstrukciók kényelmesen beilleszthetők a szemantikus elemzésbe:
  - a szimbólumoknak egy attribútuma lesz a hozzájuk generált kód
  - az összetettebb konstrukciók kódjához (eddig) csak kombinálni kellett a részeihez generált kódokat
    - pl. az elágazás kódja az egyes ágak kódja kiegészítve néhány összehasonlítással és ugrással

27

Fordítóprogramok előadás (A,C,T szakirány)

Kódgenerálás I. (kifejezések és vezérlési szerkezetek)

## Kódgenerálás és *S* – ATG

- Emlékeztető: *S*-attribútum fordítási grammatica
  - csak szintetizált („alulról felfelé” terjedő) attribútumok
  - jól illeszkedik az alulról felfelé elemzésekhez (*bison++*)
- Az eddig látott konstrukciók kényelmesen beilleszthetők a szemantikus elemzésbe:
  - a szimbólumoknak egy attribútuma lesz a hozzájuk generált kód
  - az összetettebb konstrukciók kódjához (eddig) csak kombinálni kellett a részeihez generált kódokat
    - pl. az elágazás kódja az egyes ágak kódja kiegészítve néhány összehasonlítással és ugrással
- Bonyolultabb a kódgenerálás a *nem strukturált* konstrukciók esetén.
  - goto, break, kivételkezelés

27

Fordítóprogramok előadás (A,C,T szakirány)

Kódgenerálás I. (kifejezések és vezérlési szerkezetek)

## A goto utasítás fordítása

### Forrásprogram

```
Lab: x++;  
     ...  
     goto Lab;
```

Vigyázni kell a következőkre:

- A felhasználó címkeje nehogy egybe essen egy generált címkével.
  - vagy generálni kell a felhasználó címkeje helyett is egy újat, és feljegyezni a szimbólumtáblába
  - vagy olyan címkéket generálni, amit a felhasználó nem írhat a forrásprogramba
- Ha pl. push ecx ... pop ecx utasításokkal körülvett ciklusmagból történik a kiugrás, helyre kell állítani a vermet...
- Még bonyolultabb, ha alprogramkból is ki lehet ugrani...

### Generálandó kód

```
Lab: inc [X]  
     ...  
     jmp Lab
```

28

Fordítóprogramok előadás (A,C,T szakirány)

Kódgenerálás I. (kifejezések és vezérlési szerkezetek)

## A break utasítás fordítása

- A break utasítás a legbelso ciklusból / elágazásból ugrik ki.
- Példa:

### Forrásszöveg

```
while( b )  
{  
    x++;  
    break;  
}
```

### Generálandó kód

```
Eleje: mov al,[B]  
        cmp al,1  
        jne Vége  
        inc [X]  
        jmp Vége  
        jmp Eleje
```

Vége:

## A break utasítás fordítása

- Alulról felfelé elemzéskor...
  - a ciklusmag kódját kell először generálni (a break kódját is)
  - a ciklus kódját később

## A break utasítás fordítása

- Alulról felfelé elemzéskor...
  - a ciklusmag kódját kell először generálni (a break kódját is)
  - a ciklus kódját később
- Probléma:
  - a Vége címkét a ciklus feldolgozásakor generáljuk,
  - pedig szükség van rá a break kódjában is!
  - Azaz ez a címke egy örökölt attribútum...

## A break utasítás fordítása

- Alulról felfelé elemzéskor...
  - a ciklusmag kódját kell először generálni (a break kódját is)
  - a ciklus kódját később
- Probléma:
  - a Vége címkét a ciklus feldolgozásakor generáljuk,
  - pedig szükség van rá a break kódjában is!
  - Azaz ez a címke egy örökölt attribútum...
- Megoldás lehet:
  - kihagyni a generált kódban a címke helyét
  - megjegyezni, hogy volt-e a ciklusmagban break (ez szintetizált attribútum!)
  - ha volt, akkor a ciklus generálásakor kitöltsük a hiányzó címkét

## A break és az L – ATG-k

- Emlékeztető: L-attribútum fordítási grammaika
  - az attribútumok először felülről lefelé, majd a szabályban balról jobbra, végül felfelé terjednek
  - jól illeszkedik az LL elemzésekhez (pl. rekurzív leszállás)
- A break fordításához szükséges örökölt attribútum nem okoz gondot L – ATG esetén
  - ciklus szimbólumnál lefelé haladva generáljuk a Vége címkét
  - a ciklusmag kódjának generálásakor a címke már rendelkezésre áll
  - a ciklusmag feldolgozása után felfelé haladva a szintaxisában elérhető a ciklus kódja

## Példa: rekurzív leszállás és a break

```
Kod ciklus_utasitas()  
{  
    Cimke eleje = cimkegenerator.ujcimke();  
    Cimke vege = cimkegenerator.ujcimke();  
    Kod ciklusmag_kodja;  
    ...  
    if( aktualis_token == break_token )  
        ciklusmag_kodja = break_utasitas( vege );  
    ...  
    return ...;  
}  
  
Kod break_utasitas( Cimke c )  
{  
    elfogad( break_token );  
    return new Kod("jmp " + c);  
}
```

(Kicsit csaltunk: a ciklus eljárásában közvetlenül nem jelenne meg a break, hanem a ciklusmag egy utasítássorozat, ami utasításokból áll és egy utasítás lehet break is...)

## Kódgenerálás II. (alprogramok, memóriakezelés)

Fordítóprogramok előadás (A,C,T szakirány)

2008. őszi félév

## Alprogramok fordítása

- függvény, eljárás, metódus
- paraméterátadási formák
  - érték szerint, hivatkozás szerint...
- tagfüggvények

## Függvény, eljárás

- **függvény:**
  - csak „bemenő” paraméterek
  - visszatérési érték
  - nincs mellékhatása
- **eljárás:**
  - „ki- és bemenő” paraméterek
  - jellemzően nincs visszatérési érték
- *C, C++, Java, ...*: nincs ilyen megkülönböztetés

### Példa: függvény

```
int duplaja( int n )
{
    return 2*n;
}
```

### Példa: eljárás

```
void duplaz( int & n )
{
    n *= 2;
}
```

## Alprogramok írása assemblyben

- paraméter nélküli alprogramok
- paraméterátadás
- lokális változók

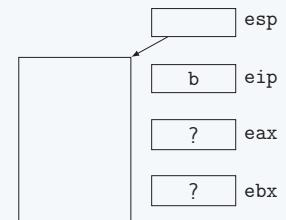
## A call és ret utasítások

- **call Címke**
  - az eip regiszter tartalmát a verembe teszi
    - ez a call utáni utasítás címe
    - visszatérési címnek nevezik
  - átadja a vezérlést a Címke címkéhez
    - mint egy ugró utasítás
- **ret**
  - kiveszi a verem legfelső négy bájtját és az eip regiszterbe teszi
    - mint egy pop utasítás
    - a program a veremben talált címnél folytatódik

### Példa: paraméter nélküli alprogram

```
a:      ...
b:      call nulláz
c:      ...

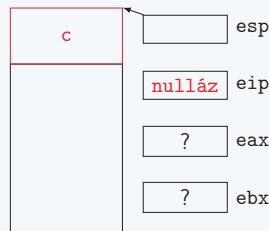
nulláz: mov eax,0
d:      mov ebx,0
e:      ret
```



## Példa: paraméter nélküli alprogram

```
a:    ...
b:    call nulláz
c:    ...

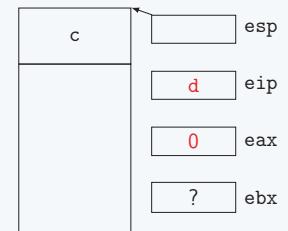
nulláz: mov eax,0
d:    mov ebx,0
e:    ret
```



## Példa: paraméter nélküli alprogram

```
a:    ...
b:    call nulláz
c:    ...

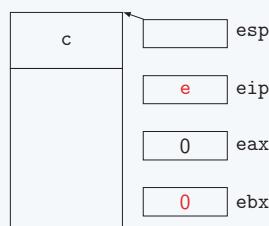
nulláz: mov eax,0
d:    mov ebx,0
e:    ret
```



## Példa: paraméter nélküli alprogram

```
a:    ...
b:    call nulláz
c:    ...

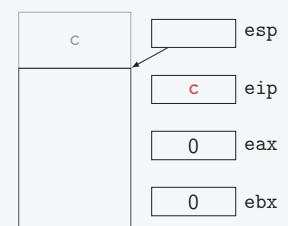
nulláz: mov eax,0
d:    mov ebx,0
e:    ret
```



## Példa: paraméter nélküli alprogram

```
a:    ...
b:    call nulláz
c:    ...

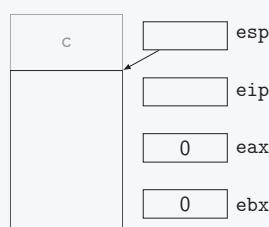
nulláz: mov eax,0
d:    mov ebx,0
e:    ret
```



## Példa: paraméter nélküli alprogram

```
a:    ...
b:    call nulláz
c:    ...

nulláz: mov eax,0
d:    mov ebx,0
e:    ret
```



## Paraméterek átadása

- a paramétereket a verembe kell tenni a call utasítás előtt
- C stílusú paraméterátadás: fordított sorrendben tessük a verembe
  - az utolsó kerül legalulra
  - az első a verem tetejére
- az ejárásból való visszatérés után a hívó állítja vissza a vermet
- visszatérési érték: az eax regiszterbe kerül

## Példa: paraméterátadás (1. változat)

```

a:      ...
b:      push dword 5
c:      call duplája
d:      add esp,4
e:      ...

duplája: mov eax,[esp+4]
f:      add eax,[esp+4]
g:      ret

```

## Példa: paraméterátadás (1. változat)

```

a:      ...
b:      push dword 5
c:      call duplája
d:      add esp,4
e:      ...

duplája: mov eax,[esp+4]
f:      add eax,[esp+4]
g:      ret

```

## Példa: paraméterátadás (1. változat)

```

a:      ...
b:      push dword 5
c:      call duplája
d:      add esp,4
e:      ...

duplája: mov eax,[esp+4]
f:      add eax,[esp+4]
g:      ret

```

## Példa: paraméterátadás (1. változat)

```

a:      ...
b:      push dword 5
c:      call duplája
d:      add esp,4
e:      ...

duplája: mov eax,[esp+4]
f:      add eax,[esp+4]
g:      ret

```

## Példa: paraméterátadás (1. változat)

```

a:      ...
b:      push dword 5
c:      call duplája
d:      add esp,4
e:      ...

duplája: mov eax,[esp+4]
f:      add eax,[esp+4]
g:      ret

```

## Példa: paraméterátadás (1. változat)

```

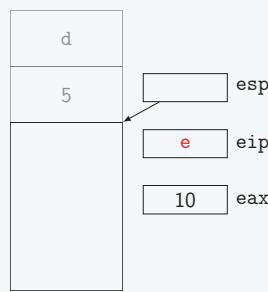
a:      ...
b:      push dword 5
c:      call duplája
d:      add esp,4
e:      ...

duplája: mov eax,[esp+4]
f:      add eax,[esp+4]
g:      ret

```

## Példa: paraméterátadás (1. változat)

```
a:      ...
b:      push dword 5
c:      call duplája
d:      add esp,4
e:      ...
duplája: mov eax,[esp+4]
f:      add eax,[esp+4]
g:      ret
```



## Paraméterátadás – 1. változat

- hivatkozás a paraméterekre (ha mindegyik 4 bájtos):
  - 1.: [esp+4] 2.: [esp+8] ...

## Paraméterátadás – 1. változat

- hivatkozás a paraméterekre (ha mindegyik 4 bájtos):
  - 1.: [esp+4] 2.: [esp+8] ...
- **probléma:** Időnként használni akarjuk a vermet az alprogram belsejében.  
Például:
  - egyes regiszterek elmentése
  - lokális változóknak helyfoglalás
- Ilyenkor változik a paraméterekre való hivatkozás módja!

### Példa: veremhasználat alprogram belsejében

```
alprogram: ; itt [esp+4] az első paraméter
           push ecx
           ; itt már [esp+8]
           pop ecx
           ; itt mégint [esp+4]
           ret
```

## Paraméterátadás – 2. változat

- **Megoldási ötlet:** Használjuk az ebp regisztert az esp helyett:
  - az alprogram elején ebp megkapja esp értékét
  - közben esp akár hogyan változhat
  - a paramétereket minden ugyanúgy érjük el: [ebp+4], [ebp+8], ...

### Példa: ebp használata a paraméterek eléréséhez

```
alprogram: mov ebp,esp
           ; az első paraméter: [ebp+4]
           push ecx
           ; ugyanúgy [ebp+4]
           pop ecx
           ; továbbra is [ebp+4]
           ret
```

## Paraméterátadás – 2. változat

- **probléma:** Alprogramhívás egy alprogram belsejében: elállítja ebp értékét!

### Alprogramhívás alprogramban

```
egyik: mov ebp,esp
       ; első paraméter: [ebp+4]
       call masik ; ez elállítja ebp-t
       ; itt ebp-t már nem tudjuk használni
       ret

masik: mov ebp,esp ; itt állítjuk el ebp-t
       ; ...
       ret
```

## Paraméterátadás – 3. (végső) változat

- **Megoldás:** Mentsük el ebp értékét az alprogram elején a verembe és állítsuk vissza a végén!

### Alprogramhívás alprogramban

```
egyik: push ebp
       mov ebp,esp
       ; első paraméter: [ebp+4]
       call masik ; ez megőrzi ebp értékét
       ; itt is [ebp+8] az első paraméter
       pop ebp
       ret

masik: push ebp ; itt mentjük el ebp-t
       mov ebp,esp
       ; ...
       pop ebp ; és itt állítjuk vissza
       ret
```

## Paraméterátadás – 3. (végső) változat

- Hivatkozás a paraméterekre: [ebp+8], [ebp+12], ...
- [ebp]: az ebp elmentett értéke
- [ebp+4]: a visszatérési cím
- Akár rekurzív is lehet az alprogram...

## Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
ret

```

## Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
ret

```

## Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
ret

```

## Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
ret

```

## Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
ret

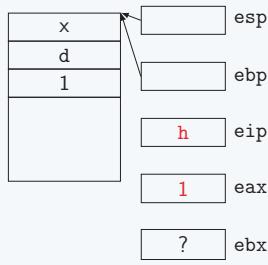
```

## Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
ret

```

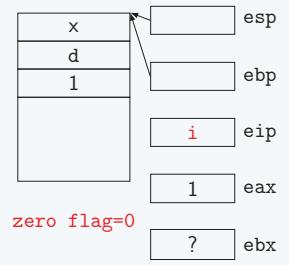


## Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
ret

```

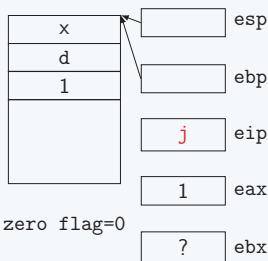


## Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
ret

```

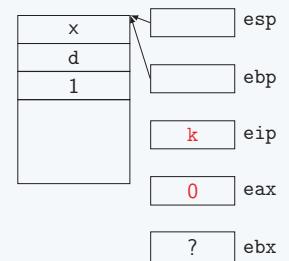


## Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
ret

```

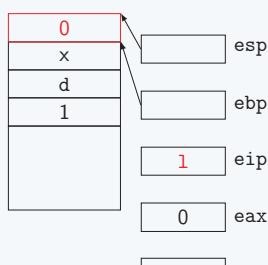


## Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
ret

```

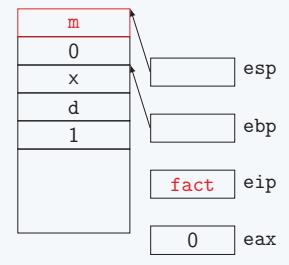


## Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
ret

```

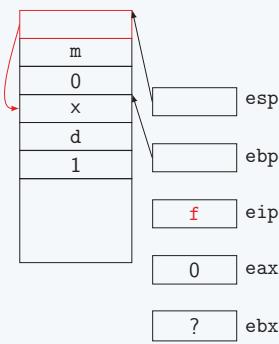


### Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
ret

```

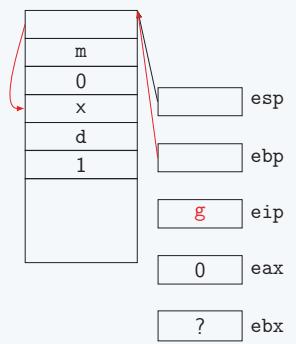


### Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
ret

```

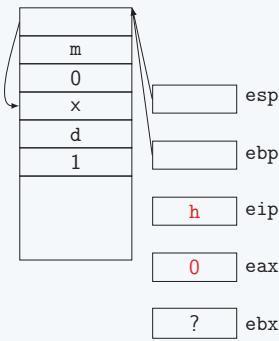


### Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
ret

```

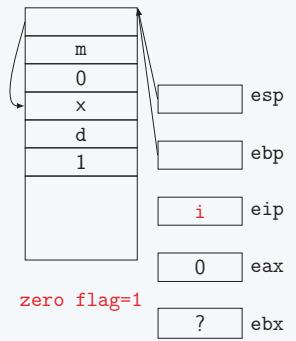


### Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
ret

```



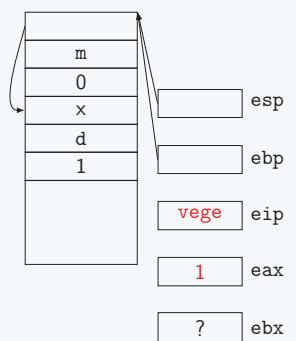
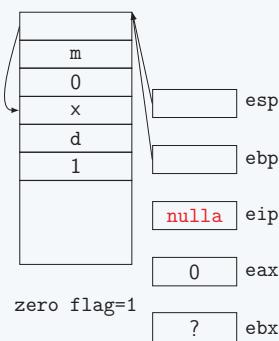
zero flag=1

### Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
ret

```

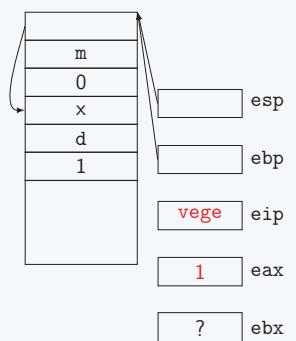


### Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
ret

```

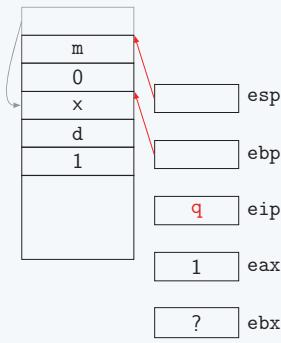


### Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
q: ret

```

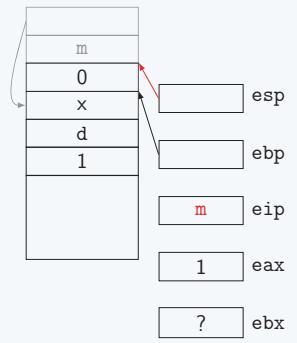


### Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
q: ret

```

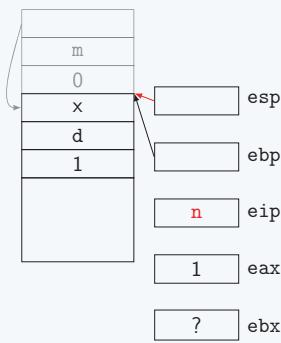


### Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
q: ret

```

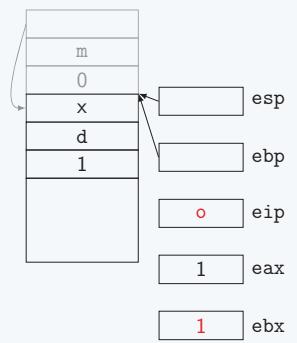


### Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
q: ret

```

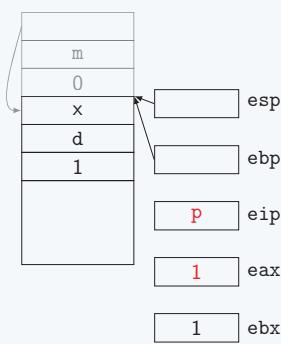


### Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
q: ret

```

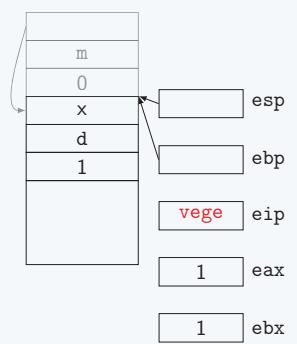


### Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
q: ret

```

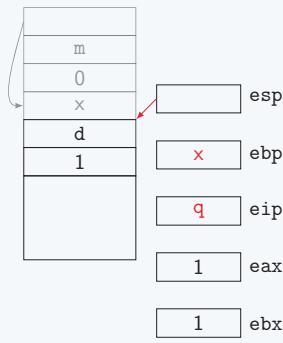


## Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
q: ret

```

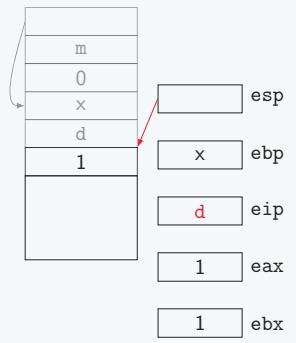


## Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
q: ret

```

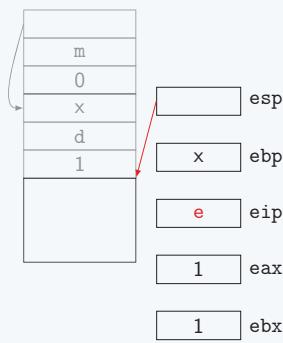


## Paraméterátadás (3. változat)

```

a: ...
b: push dword 1
c: call fact
d: add esp,4
e: ...
fact: push ebp
f: mov ebp,esp
g: mov eax,[ebp+8]
h: cmp eax,0
i: je nulla
j: dec eax
k: push eax
l: call fact
m: add esp,4
n: mov ebx,[ebp+8]
o: mul ebx
p: jmp vege
nulla: mov eax,1
vege: pop ebp
q: ret

```



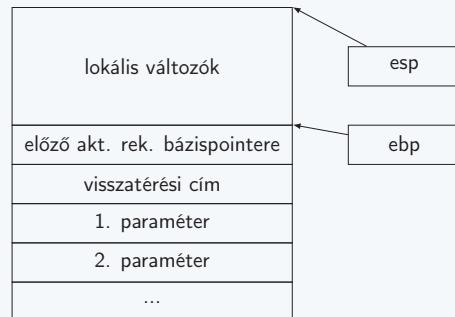
## Paraméterátadás – megjegyzések

- minden éppen futó alprogram-példányhoz tartozik egy résza veremből
  - aktivációs rekord / verem-keret (stack frame)
- az ebp regiszter segítségével érhetők el az aktuális aktivációs rekord részei bázis pointer
- az ebp-ból indulva egy láncolt listára vannak felfűzve az aktivációs rekordok
  - ez teszi lehetővé az eggyel korábbi aktivációs rekordhoz való visszatérést az alprogram végén

## Lokális változók

- a lokális változók a verem tetejére kerülnek
- hivatkozás a lokális változóra (ha mindenky 4 bájtos):
  - [ebp-4], [ebp-8], ...
- az esp a lokális változók területe „fölé” mutat
  - lehet veremműveleteket és alprogram-hívásokat végrehajtani a lokális változók felülírása nélkül

## Az aktivációs rekord felépítése



## Alprogramok sémája

```

alprogram: push ebp
          mov ebp,esp
          sub esp,'a lokális változók mérete'
          ; az alprogram törzse
vége:    add esp,'a lokális változók mérete'
          pop ebp
          ret

```

- egyedi címkeket kell generálni
- a lokális változók összmérete a törzs szintetizált attribútuma lehet

- összesíteni kell az alprogramban használt lokális változókat
  - ha minden az elején kell definiálni, akkor könnyű
  - fel kell venni őket a szimbólumtáblába
  - mindenkihez ki kell számolni, hogy hol fog elhelyezkedni: [ebp-?]
- az összméretükre szükség lesz az alprogram
 

```
sub esp,'a lokális változók mérete'
```

 soránál

- kiegészítjük a kifejezéskiértékelés fordítását egy új esettel: „ha a kifejezés egyetlen lokális változó...”
- a szimbólumtáblából kiolvassuk a pozícióját: *p*
- a generálandó kód:

## Hivatkozás lokális változóra

```
mov eax,[ebp-p]
```

## A 'return kifejezés' utasítás kódja

```
; a kifejezés kiértékelése eax-be
jmp vége
```

- itt is tudni kell az alprogram végét jelző címkét (hasonló problémák, mint a break esetén)

## Alprogramok sémája

```

; utolsó paraméter kiértékelése eax-be
push eax
;
; ...
; 1. paraméter kiértékelése eax-be
push eax
call alprogram
add esp,'a paraméterek összhossza'

```

- érték szerint:
  - a paraméterértékeket másoljuk a verembe
  - ha az alprogram módosítja, az nem hat az átadott változóra
- hivatkozás szerint
  - az átadandó változóra mutató pointert kell a verembe tenni
  - az alprogramban a lokális változó kiértékelése is módosul:
 

```
mov eax,[ebp+p]
          mov eax,[eax]
```

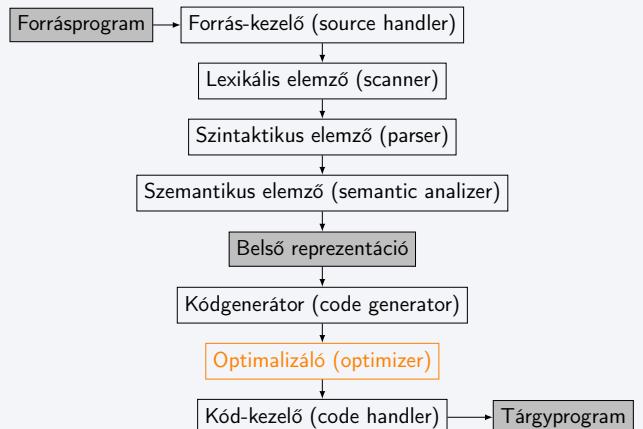
- statikus memóriakezelés:
  - a .data vagy .bss szakaszban
  - globális vagy statikusnak deklarált változók
  - előre ismerni kell a változók méretét, darabszámát
- dinamikus memóriakezelés
  - blokk-szerkezethez kötődő, lokális változók: *verem*
  - tetszőleges élettartamú változók: *heap memória*

- két alapvető művelet: *allokálás* és *felszabadítás*
- az operációs rendszer vagy egy programkönyvtár végzi
- a szabad és foglalt területeket nyilván kell tartani
  - allokáláskor valamilyen stratégia szerint egy szabad területet kell lefoglalni
  - a felszabadított memóriát hozzá kell tenni a szabad területhez
- assemblyból lehet hívni a C malloc és free függvényeit

## Kódoptimalizálás

Fordítóprogramok előadás (A,C,T szakirány)

## A fordítóprogramok szerkezete



1 Fordítóprogramok előadás (A,C,T szakirány) Kódoptimalizálás

2 Fordítóprogramok előadás (A,C,T szakirány) Kódoptimalizálás

## A szintézis menete „valójában”

- ① Optimalizálási lépések végrehajtása az eredeti programon (vagy annak egyszerűsített változatán).
- ② Kódgenerálás.
- ③ Gépfüggő optimalizálás végrehajtása a generált kódon.

## A kódoptimalizálás célja

- Hatékonyabb program létrehozása:
  - nagyobb sebesség
  - kisebb méret
- **Ezek a célok időnként ellentmondanak egymásnak!**

3 Fordítóprogramok előadás (A,C,T szakirány) Kódoptimalizálás

4 Fordítóprogramok előadás (A,C,T szakirány) Kódoptimalizálás

## Követelmény a kódoptimalizálással szemben

- „Az optimalizált programnak ugyanúgy kell működnie, mint az eredetinek.” - Ez sok minden jelenthet!
  - ugyanarra a bemenetre ugyanazt a kimenetet adja?
  - eseményvezérelt környezetben ugyanúgy viselkedik?
  - párhuzamos környezetben ugyanúgy viselkedik?
  - stb.
- Az adott nyelv szemantikája (jelentése) dönti el, hogy egy optimalizálási lépés megengedhető-e.

## Kódoptimalizálási lépések osztályozása

- Mit optimalizálunk?
  - az eredeti programot (vagy annak egyszerűsített változatát):  
itt még vannak ciklusok, elágazások, kifejezéskiértékelés stb.
  - a tárgyprogramot: jellemzően assembly kódot
- Mi az átalakítás hatóköre?
  - lokális: kis programrészletek átalakítása
  - globális: a teljes program szerkezetét kell vizsgálni
- Mit használ ki az átalakítás?
  - általános optimalizálási stratégiák: algoritmusok javítása
  - gépfüggő optimalizálás: az adott architektúra sajátosságait használja ki

5 Fordítóprogramok előadás (A,C,T szakirány) Kódoptimalizálás

6 Fordítóprogramok előadás (A,C,T szakirány) Kódoptimalizálás

## Lokális optimalizálás: alapblokk fogalma

### Definíció: alapblokk

Egy programban egymást követő utasítások sorozatát alapblokknak nevezzük, ha

- az első utasítás kivételével egyik utasítására sem lehet távolról átadni a vezérlést  
(assembly programokban: ahová a `jmp`, `call`, `ret` utasítások „ugranak”; magas szintű nyelvekben: eljárások, ciklusok eleje, elágazások ágainak első utasítása, `goto` utasítások célpontjai)
- az utolsó utasítás kivételével nincs benne vezérlés-átadó utasítás  
(assembly programban: `jmp`, `call`, `ret` magas szintű nyelvekben: elágazás vége, ciklus vége, eljárás vége, `goto`)
- az utasítás-sorozat nem bővíthető a fenti két szabály megsértése nélkül

## Alapblokk szerepe az optimalizálásban

### A definíció következményei:

- egy alapblokknak pontosan egy belépési pontja van (az első utasítás)
- az utasításai szekvenciálisan hajtódnak végre
- ha rá kerül a vezérlés, akkor az összes utasítása pontosan egyszer hajtódnak végre

### • Lokális optimalizálás: egy alapblokkon belüli átalakítások

## Alapblokok meghatározása

- Jelöljük meg:
  - a program első utasítását
  - azokat az utasításokat, amelyekre távolról át lehet adni a vezérlést
  - a vezérlés-átadó utasításokat követő utasításokat
- minden megjelölt utasításhoz tartozik egy alapblokk, ami a következő megjelölt utasításig (vagy az utolsó utasításig) tart.

## Alapblokok: példa

```
main: mov eax,[Cimke1]
      cmp eax,[Cimke2]
      jz igaz
      dec dword [Cimke1]
      inc dword [Cimke2]
      jmp vege
igaz: inc dword [Cimke1]
      dec dword [Cimke2]
vege: ret
```

## Tömörítés

- Cél: minél kevesebb konstans és konstans értékű változó legyen!
- konstansok összevonása: a fordítási időben kiértékelhető kifejezések kiszámítása

### Eredeti kód

```
a := 1 + b + 3 + 4;
```

### Optimalizált kód

```
a := 8 + b;
```

- konstans továbbterjesztése: a fordítási időben kiszámítható értékű változók helyettesítése az értékükkel

### Eredeti kód

```
a := 6;
b := a / 2;
c := b + 5;
```

### Optimalizált kód

```
a := 6;
b := 3;
c := 8;
```

## Azonos kifejezések többszöri kiszámításának elkerülése

### Eredeti kód

```
x := 20 - (a * b);
y := (a * b) ^ 2;
```

### Optimalizált kód

```
t := a * b;
x := 20 - t;
y := t ^ 2;
```

- Ez csak látszólag növeli a program méretét!

- az `a*b` kifejezés kiértékelése is több assembly utasítás
- a `t` változó lehet egy regiszter is

- Megvalósítás a gyakorlatban:

- az utasításokból egy címkézett, irányított körmentes gráfot építünk,
- ebből generálható az optimalizált kód

## Változó továbbterjesztése

## Eredeti kód

```
x := a + b;
y := x;
z := y;
```

## Optimalizált kód

```
x := a + b;
y := x;
z := x;
```

- Ha az y változóra a továbbiakban már nincs szükség, akkor  $y := x$  törölhető!
- (De ez a törlés már globális optimalizálás...)
- Ez is megoldható az előzőleg említett gráfos módszerrel.

## Ablakoptimalizálás

- Ez egy módszer a lokális optimalizálás egyes fajtához.

## • Ablak:

- egyszerre csak egy néhány utasításnyi részt vizsgálunk a kódóból
- a vizsgált részt előre megadott mintákkal hasonlítjuk össze
- ha illeszkedik, akkor a mintához megadott szabály szerint átalakítjuk
- ezt az „ablakot” végigcsúsztatjuk a programon

## • Az átalakítások megadása:

- $\{minta \rightarrow helyettesítés\}$  szabályhalmazzal
- a mintában lehet paramétereket is használni

## Ablakoptimalizálás: példa

- ablak mérete: 1 utasítás
- szabályhalmaz:
  $\{\text{mov } reg, 0 \rightarrow \text{xor } reg, reg$   
 $\text{add } reg, 0 \rightarrow ; \text{ elhagyható}\}$

## Eredeti kód

```
add eax,0
mov ebx,eax
mov ecx,0
```

## Optimalizált kód

## Ablakoptimalizálás: példa

- ablak mérete: 1 utasítás
- szabályhalmaz:
  $\{\text{mov } reg, 0 \rightarrow \text{xor } reg, reg$   
 $\text{add } reg, 0 \rightarrow ; \text{ elhagyható}\}$

## Eredeti kód

```
add eax,0
mov ebx,eax
mov ecx,0
```

## Optimalizált kód

; elhagyott utasítás

## Ablakoptimalizálás: példa

- ablak mérete: 1 utasítás
- szabályhalmaz:
  $\{\text{mov } reg, 0 \rightarrow \text{xor } reg, reg$   
 $\text{add } reg, 0 \rightarrow ; \text{ elhagyható}\}$

## Eredeti kód

```
add eax,0
mov ebx,eax
mov ecx,0
```

## Optimalizált kód

; elhagyott utasítás

```
mov ebx,eax
```

## Ablakoptimalizálás: példa

- ablak mérete: 1 utasítás
- szabályhalmaz:
  $\{\text{mov } reg, 0 \rightarrow \text{xor } reg, reg$   
 $\text{add } reg, 0 \rightarrow ; \text{ elhagyható}\}$

## Eredeti kód

```
add eax,0
mov ebx,eax
mov ecx,0
```

## Optimalizált kód

; elhagyott utasítás

```
mov ebx,eax
xor ecx,ecx
```

## Tipikus egyszerűsítések ablakoptimalizáláshoz

- felesleges műveletek törlése: nulla hozzáadása vagy kivonása
- egyszerűsítések: nullával szorzás helyett a regiszter törlése
- nulla mozgatása helyett a regiszter törlése
- regiszterbe töltés és ugyanoda visszaírás esetén a visszaírás elhagyható
- utasításmétlések törlése: ha lehetséges, az ismétlések törlése

## Globális optimalizálás

- a teljes program szerkezetét meg kell vizsgálni
- ennek módszere az [adatáram-analízis](#):
  - Mely változók értékeit számolja ki egy adott alapblokk?
  - Mely változók értékeit melyik alapblokk használja fel?
- lehetővé teszi:
  - az azonos kifejezések többszöri kiszámításának kiküszöbölését akkor is, ha különböző alapblokokban szerepelnek
  - konstansok és változók továbbterjesztését alapblokkok között is
  - elágazások, ciklusok optimalizálását

## Kódkiemelés

**Eredeti kód**

```
if( x < 10 )
{
    a = 0;
    b++;
}
else
{
    b--;
    a = 0;
}
```

**Optimalizált kód**

```
a = 0;
if( x < 10 )
{
    b++;
}
else
{
    b--;
}
```

## Kódcsülyesztés

**Eredeti kód**

```
if( x < 10 )
{
    x = 0;
    b++;
}
else
{
    b--;
    x = 0;
}
```

**Optimalizált kód**

```
if( x < 10 )
{
    b++;
}
else
{
    b--;
}
x = 0;
```

Mi lenne, ha itt *kódkiemelést* alkalmaznánk?

## Ciklusok kifejtése

**Eredeti kód**

```
for( int i=0; i<4; ++i )
{
    a += t[i];
}
```

**Optimalizált kód**

```
a += t[0];
a += t[1];
a += t[2];
a += t[3];
```

Mérlegelní kell, hogy a méret és a sebesség mennyire fontos...

## Parciális kifejtés

**Eredeti kód**

```
for( int i=0; i<4; ++i )
{
    a += t[i];
}
```

**Optimalizált kód**

```
for( int i=0; i<4; i+=2 )
{
    a += t[i];
    a += t[i+1];
}
```

## Ciklusok összevonása

### Eredeti kód

```
for( int i=0; i<4; ++i )
{
    t1[i] = 0;
}
for( int i=0; i<4; ++i )
{
    t2[i] = 1;
}
```

### Optimalizált kód

```
for( int i=0; i<4; i++ )
{
    t1[i] = 0;
    t2[i] = 1;
}
```

## Frekvenciaredukálás

- költséges utasítások „átköltözöttetése” ritkábban végrehajtódó alapblokkba
- példa:
  - ciklusinvariánsnak nevezük azokat a kifejezéseket, amelyeknek a ciklus minden lefutásakor azonos az értékük
  - a ciklusinvariánsok (esetenként) kiemelhetők a ciklusból

### Eredeti kód

```
cin >> a;
cin >> h;
while( h > 0 )
{
    cout << h*h*sin(a);
    cin >> h;
}
```

### Optimalizált kód

```
cin >> a;
cin >> h;
double s = sin(a);
while( h > 0 )
{
    cout << h*h*s;
    cin >> h;
}
```

## Erős redukció

- a ciklusban lévő költséges művelet (legtöbbször szorzás) kiváltása kevésbé költségessel

### Eredeti kód

```
for( int i=a; i<b; i+=c )
{
    cout << 3*i;
}
```

### Optimalizált kód

```
int t1 = 3*a;
int t2 = 3*c;
for( int i=a; i<b; i+=c )
{
    cout << t1;
    t1 += t2;
}
```

## Funkcionális programozási nyelvek fordítása

Fordítóprogramok előadás (A,C,T szakirány)

## Funkcionális programozási nyelvek

### • programok futása:

- funkcionális eset: függvények kiszámítása
- imperatív eset: állapotváltozások sorozata
- tiszta függvény: olyan függvény, amelynek nincs mellékhatása
- tiszta funkcionális nyelv: csak tiszta függvények írhatók benne
  - akár imperatív nyelveken is lehet „tiszta funkcionális” stílusban programozni
  - az imperatív nyelvekben megszokott változók nem léteznek
- példák: Haskell, Clean, Erlang, OCaml, F#, Lisp, Scheme, ...

## Funkcionális nyelvek sajátosságai

- (általában) nagyon kifinomult a típusrendszerük
  - többalakúság (polimorf típusrendszer)
  - típusellenőrzés helyett típusvezetés
  - típusosztályok
- a függvények „teljes jogú tagjai a nyelvnek” („first class citizens”):
  - függvényeket is lehet paraméterként átadni
  - függvény is lehet visszatérési érték
- lusta (lazy) / szigorú (strict) kiértékelés:
  - lusta: csak akkor értékel ki egy kifejezést, amikor arra valóban szükség van
  - szigorú: minden paramétert kiértékel még a függvényhívás előtt (az imperatív nyelvek szigorúak)

## Funkcionális nyelvek fordítása

- lexikális / szintaktikus / szemantikus elemzés eredménye egy transzformált program, amely
  - típusozott
  - néhány egyszerű nyelvi konstrukcióból épül fel
- ezen a transzformált programon optimalizációs lépéseket hajtunk végre
- kódgenerálás: általában C-re
  - így nem kell újraírni a C fordítóban meglévő sok kifinomult imperatív jellegű optimalizációt
- a végrehajtáshoz szükséges egy futtatókörnyezet is

## A margó szabály

- az utasítások végét és a blokkszerkezetet a sorok behúzása definíálja

### Margó szabálytal

```
f x y = a + b where
  a =
    x * y
  b = x - y
  g x = 2 * x
```

### Explicit jelöléssel

```
f x y = a + b where {
  a =
    x * y;
  b = x - y;
  g x = 2 * x;
```

## A margó szabály megvalósítása

- a lexikális elemző feladata
- minden lexikális elemhez megjegyezzük, hogy hányadik oszlopban van az első karaktere (a fehér szóközök is figyelembe kell venni)
- egy verembe beletesszük a legelső utasítás behúzását
- bizonyos kulcsszavak után (Haskellben pl. where, of) beszűrünk egy „{” tokenet és a következő token behúzását a verembe tesszük
- minden sortörésnél meg kell vizsgálni az új sor behúzást:
  - ha egyenlő a verem tetején lévő értékkel, akkor beszűrünk egy „;”-t
  - ha nagyobb a behúzás, akkor nincs teendő
  - ha kisebb a behúzás, akkor beszűrünk egy „;”-t, majd addig szűrünk be egy-egy „}”-t és veszünk ki egy-egy értéket a veremből, amíg a verem tetején nagyobb érték van az aktuális sor behúzásánál

## Polimorf típusok

- A típusok lehetnek „több alakúak”:

- pl. egészek listája, karakterek listája, listák listája...

```
[1,2,3] :: [Int]
['x'] :: [Char]
[[5], [1,2]] :: [[Int]]
[] :: [a]
[][] :: [[b]]
```

- A példában az a és b típusváltozók.

- Típuslevezetéskor nem a típusok *egyenlőségét* kell vizsgálni, hanem az *egyesíthetőségét*.

- Egyesíthetőség (kb.): A típusváltozók helyettesítésével azonos alakúra hozható-e a két típus?

7

Fordítóprogramok előadás (A,C,T szakirány)

Funkcionális programozási nyelvek fordítása

## Példák típuslevezetésre

- A [[‘x’, ‘y’], []] kifejezés típusát szeretnénk levezetni.

‘x’ :: Char  
‘y’ :: Char

- Char és Char egyesíthetők és az eredmény Char, ezért:

['x', 'y'] :: [Char]

- Az üres lista típusa:

[] :: [a]

- [Char] és [a] egyesíthetők az a → Char helyettesítéssel, ezért:  
[['x', 'y'], []] :: [[Char]]

- A [[‘x’, ‘y’], ‘z’]] kifejezés esetén egyesíteni kellene a [Char] és Char típusokat, ami nem lehetséges. ⇒ Tipushiba!

8

Fordítóprogramok előadás (A,C,T szakirány)

Funkcionális programozási nyelvek fordítása

## Szigorú és lusta kiértékelés

- C-ben kiértékelődik minden paraméter:

```
int f( int x, int y )
{
    return x + 1;
}
int a = 4321;
int b = f( a/2, a*515341 )
```

- Haskellben csak az első paraméter értékelődik ki:

```
f x y = x + 1
a = 4321
b = f (a/2) (a*515341)
```

- Nem csak hatékonysági kérdés!

- f( 5, a/0 ) futási idejű hiba C-ben
- f 5 (a/0) hiba nélkül lefut, és 6-ot ad Haskellben

9

Fordítóprogramok előadás (A,C,T szakirány)

Funkcionális programozási nyelvek fordítása

## Függvények mint paraméterek vagy visszatérési értékek

- A map egy lista minden elemére végrehajtja az f függvényt:

map f [] = []
map f (első:többi) = (f első) : (map f többi)

- Az addToAll egy olyan függvényt ad eredményül, ami egy lista minden elemét növeli n-nel:

addToList n = map (n+)

- A függvények tehát adatokként kezelhetők, azaz teljes jogú tagjai a nyelvnek.

- Ez C-ben elég nehézkesen, függvénypointerekkel oldható csak meg.

10

Fordítóprogramok előadás (A,C,T szakirány)

Funkcionális programozási nyelvek fordítása

## A lusta kiértékelés és a függvények teljes jogú tagságának megvalósítása

- Az f (a/2) (a\*515341) hívásban „kiértékeletlenül” kell átadni a paramétereket, hogy az f függvény dönthesse el, melyikre van ténylegesen szüksége.

```
f x y = x + 1
```

- A map függvény első paraméterében egy függvényt kell átadnunk, ami szintén egy „kiértékeletlen” kifejezés.
- Az addToList függvény eredménye egy függvény, amit még alkalmaznunk kell egy listára, hogy kiértékelhető legyen.
- Hogyan lehet kiértékeletlen kifejezéseket kezelni?**

## A „kiértékeletlen kifejezések” kezelése

- Készítünk egy objektumot, amely tartalmaz

- egy pointert a függvényre
- további pointereket a függvény meglévő paramétereire

- Az ilyen objektumok (closure) átadhatók függvényeknek, visszaadhatók visszatérési értékként.

- Ha minden paraméter rendelkezésre áll egy closure-ben és szükséges válik a kiértékelése, akkor elvégezhető a művelet.

11

Fordítóprogramok előadás (A,C,T szakirány)

Funkcionális programozási nyelvek fordítása

12

Fordítóprogramok előadás (A,C,T szakirány)

Funkcionális programozási nyelvek fordítása

- Egy closure-ben a paraméterek is és a függvény is lehet további kiértékeletlen kifejezés.
- Ezért egy teljes kifejezés ábrázolható egy fával, általánosabb esetben egy gráffal.
- Kódgenerálás a modern lusta kiértékelésű funkcionális nyelveken írt programokhoz:
  - a programban lévő main függvényből felépítünk egy gráfot
  - a programban lévő függvénydefiníciókból egy-egy gráfátíró szabályt készítünk
- A programok lefutása:
  - a futtatórendszer minden lépésben kiválaszt egy megfelelő részt a gráfból
  - átírja valamelyik szabály alapján
  - ezt addig ismételgeti, amíg
    - a gráf egy adottá egyszerűsödött (ez a végeredmény)
    - vagy már nem egyszerűsíthető tovább

# Kitlei Róbert

## Assembly programozás

Lektorálta: Dévai Gergely

ELTE, 2007

# Tartalomjegyzék

Bevezető.....	3
Alapvető adatok.....	4
Logikai értékek.....	4
Előjel nélküli egész számok.....	4
Előjeles egész számok.....	5
Törtszámok.....	5
Karakterek, stringek.....	6
Feladatok.....	7
Az architektúra alapjai.....	8
A memória szervezése.....	8
A regiszterek.....	10
Az utasítások szerkezete.....	10
Konstansok.....	11
A memória címzése.....	11
Adatmozgatás.....	12
Logikai utasítások.....	12
Bitláptető és bitforgató utasítások.....	12
Aritmetikai utasítások.....	13
Vezérlésátadás.....	13
Programozási konstrukciók megvalósítása.....	14
Változók deklarációja, definíciója.....	14
Értékadás.....	14
Goto.....	14
Kifejezések.....	15
Szekvencia.....	16
Elágazás.....	16
Rövidzáras operátorok.....	17
Elöltesztelős ciklus.....	18
Hátultesztelős ciklus.....	19
Rögzített lépésszámú ciklus.....	20
Leszámláló ciklus.....	21
Feladatok.....	22
A futási idejű verem szerepe: rekurzív alprogramhívások megvalósítása.....	25
A futási idejű verem használata.....	25
Konkrét példa a futási idejű verem használatára: a faktoriális( 2 ) hívás megvalósítása.....	26
Konkrét példa a futási idejű verem használatára: a faktoriális függvény kódja.....	26
A veremkeret általános szerkezete.....	26
Parancssori paraméterátadás.....	27
A rendszerszolgáltatások elérése: fájlkezelés.....	28
Feladatok.....	29
Makrók.....	31
Egyisoros makrók.....	31
Többsoros makrók.....	33
Feltételes fordítás makrókkal.....	35
Feladatok.....	37
A program fordításának menete.....	39
A gépi kód szerkezetéről.....	40
CISC-elvű architektúrák.....	41
RISC architektúrák.....	41
Listafájl.....	42
Hibakeresés a programban.....	43
Gyakran előforduló hibák.....	45
A feladatok megoldásai.....	49
Értékek ábrázolása.....	49
Utasítások.....	50
Parancssori paraméterátadás, rendszerszolgáltatások.....	57
Makrók.....	60
Irodalom.....	62

## Bevezető

Az **assembly** olyan nyelvek gyűjtőneve, amelyek segítségével számítógépes programok alacsony szintű, közvetlenül a processzor által értelmezhető működése írható le emberek számára olvasható, szöveges formában. Az egyes számítógép-architektúrákon<sup>1</sup> található assembly nyelvek nagymértékben különböznek, mivel az architektúrák hardveres felépítése is eltérő. A számítógép-családok általában az architektúra minden előírását megtartják a későbbi modellekben is – azaz visszafele kompatibilisek –, emiatt a korábbi modellekre írt programok változtatás nélkül futtathatóak a későbbieken is.

A munkafüzet főleg az **x86 architektúra** assembly nyelvén foglalkozik, azon belül is a 32 bites szervezésű processzorokéval (ezt szokták IA-32 vagy x86-32 architektúrának is nevezni). Az architektúrába tartozó legismertebb gépek a 8086, a 80386 (rövidebben 386-os) és a Pentium. Napjainkban az architektúrába tartozó processzorok két legnépszerűbb gyártója az AMD és az Intel. A gépek elterjedtsége miatt a munkafüzetben leírtak könnyen kipróbálhatóak a gyakorlatban is.

Bizonyos esetekben elkerülhetetlen, hogy az assembly programozás során igénybe vegyük az **operációs rendszer** szolgáltatásait. Assembly szintről ezek eltérően kezelendőek, ezért itt is választanunk kell. A munkafüzet a **Linux** rendszerek néhány fontos szolgáltatását írja le. Ez a választás abból a szempontból nem elsőrendű, hogy az architektúra határozza meg az assembly nyelv jellegét, azonban ha azt szeretnénk, hogy működjenek is a programjaink, például tudunk a képernyőre írni, ismernünk kell ennek is a módját, ez pedig az operációs rendszeren múlik.

Ha eldöntöttük, melyik **platformra** (architektúrára, azon belül pedig operációs rendszerre) szeretnénk programokat fejleszteni, ki kell választanunk, melyik **assemblerrel** és **szerkesztőprogrammal** (**linkerrel**) dolgozzunk. Az assembler az a program, ami az assembly nyelvű forráskódot egy közbülső formátumra, **tárgykódra** fordítja, a szerkesztő pedig egy vagy több tárgykódból előállítja a gép által közvetlenül értelmezhető **futtatható állományt**. A legtöbb platformon több assembler és linker is elérhető. A munkafüzetben részletesen a **nasm** assemblerrel és a **gcc** szerkesztőprogrammal fogunk foglalkozni. A nasm fontos tulajdonságai, hogy a benne írt forráskód szintaxisa könnyen olvasható, az assembler maga pedig ingyenesen elérhető több platformra is. A **gcc**-t mint szerkesztőprogramot szintén elterjedtsége miatt választottuk.

A munkafüzet célja az assembly általános jellegének áttekintése egy adott assembly nyelv és környezet segítségével. Ennek érdekében a munkafüzet sok egyszerűsítést tartalmaz: az architektúra számos elemét, amelyeket a munkafüzetben vázolt alapelevek alapján immár könnyű megérteni, nem mutatjuk be. Szintén kihagyjuk az architektúra specifikus elemeit, amelyek rendszerprogramozás során szükségesek. A részletek megtalálhatóak a megfelelő dokumentációs anyagokban, amelyekhez a hivatkozást lásd a munkafüzet végén.

---

<sup>1</sup> számítógép-architektúra: a számítógéppel szemben támasztott hardveres követelmények leírása, kiemelten a processzor belső struktúrájának, ezen keresztül annak programozásának alapvetése. Az azonos architektúrával rendelkező számítógépeket számítógép-családoknak nevezzük.

## Alapvető adatelemek

A számítógépek adatábrázolásának alapeleme a bit. Ez egy olyan tároló, amely egy adott időpillanatban két lehetséges állapot közül az egyiket vesszi fel: nullát vagy egyet tartalmaz, közönségi szóhasználatban „*le van kapcsolva*” és „*fel van kapcsolva*”. minden adatot bitekkel reprezentálunk; a következőkben áttekintjük, hogy milyen adatokat szoktak ábrázolni biteken, illetve hogyan értelmezendőek ezek az adatok, ha csak az ábrázolt alakjukkal találkozunk. Mivel a hardver szintjén csak maguk a bitek léteznek, a programozó felelőssége, hogy pontosan tudja a program minden pontján, melyik bit és bitcsoport milyen adatot ábrázol, illetve melyik adatszerkezet része.

### Logikai értékek

Egy bitnek két állása van, ezért nagyon könnyen lehet egy biten egy logikai értéket ábrázolni. A konvenciók szerint a 0 a hamis, az 1 az igaz; a szóhasználatban ezeket egymással felcserélhetően használják, pl. „0 vagy 1 értéke igaz” ahelyett, hogy „hamis vagy igaz értéke igaz” .

A logikai értékekkel különböző műveleteket lehet végezni, amelyeket táblázatos alakban szoktak megadni. A műveleteket általában nem egyes bitekre alkalmazzák, hanem bitvektorokra, pozícióinként.

<b>not</b>	0	1
	1	0

<b>and</b>	0	1
0	0	0
1	0	1

<b>or</b>	0	1
0	0	1
1	1	1

<b>xor</b>	0	1
0	0	1
1	1	0

<b>Példa</b>	<b>not</b>	1101 0101	1010 0010	0011 1000	0100 1001
		0010 1010	0111 1101	1001 0001	1111 0110
			0010 0000	1011 1001	1011 1111

### Előjel nélküli egész számok

<b>bitek állása</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>0</b>
<b>a bit sorszáma</b>	7	6	5	4	3	2	1	0
<b>a bit helyi értéke</b>	$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$

A számok bitjeit a legkisebb helyi értéket ábrázoló bittől (a legalsó bittől) szokták a magasabb helyi értéket ábrázoló (felső) bitek felé, nullával kezdődően számoznak. Ez azért logikus, mert így a pozíciót egy kettes alapú hatvány kitevőjeként értelmezve rögtön adódik a bit helyi értéke, azaz, hogy számként „mennyit ér” a bit.

A fenti nyolcbites szám értéke:  $0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 01100010_2 = 98_{10}$ . Az alsó index a szám számrendszerét jelzi, ezt minden tizes számrendszerben adjuk meg.

A kettes számrendszerbeli számokat szokták **binárisnak**, a tízes számrendszerbelieket **decimálisnak** nevezni. Gyakran használatos még a tízenháros, más néven **hexadecimális** (rövidítve **hexa**) számrendszer is, mert ezzel rövidebben le lehet írni a bináris számokat, és ha szükség van rá, kényelmesen oda-vissza lehet alakítani őket a két alak között. A hexadecimális számjegyek: 0-tól 9-ig mint a decimálisban, utána pedig a latin ábécé betűi: A, B, C, D, E, F (tízes számrendszerben az értékük sorra 10, 11, 12, 13, 14, 15). A könnyű egymásba alakíthatóság abból következik, hogy a bitek négyes csoportja pont egy hexadecimális számjegyet kódol.

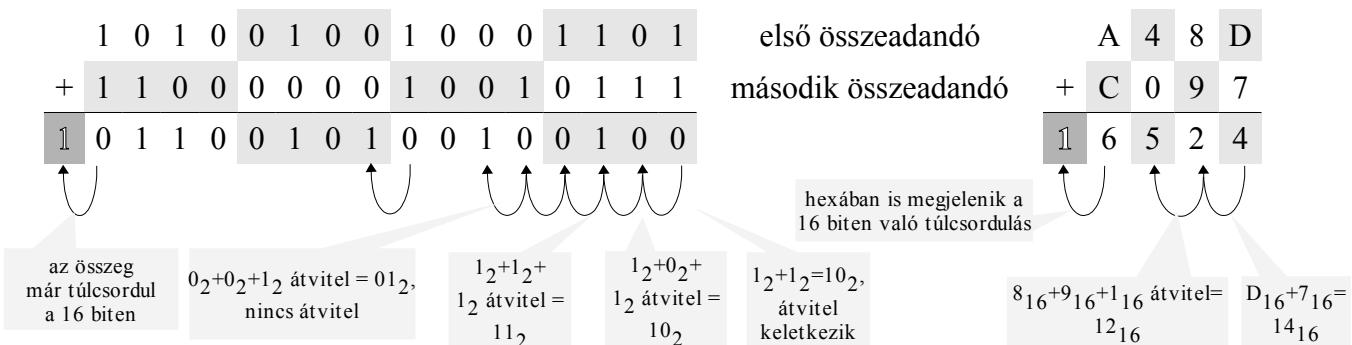
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Tízes számrendszerből kettesbe a következő algoritmussal konvertálhatunk. Írjuk le magát a számot. Osszuk el kettővel a számot. Az osztási maradékot írjuk a szám mellé, a hányados egészrészét pedig alá. Folytassuk az eljárást addig, amíg el nem jutunk nulláig.

Ekkor a bináris szám a kapott osztási maradékok sorozata *alulról felfelé olvasva*. Azért nem felülről lefelé, mert minden egyes osztás egy kettes szorzótényezővel bővíti az új bináris számjegyet, vagyis az n-edik sorban levő bináris számjegy a  $2^n$ -es helyi értékhez tartozik. Azaz felülről lefelé a bitek sorszámai rendre 0, 1, 2, ..., tehát fordítva, alulról felfelé olvasva kapjuk meg a számot. Jelen esetben  $42_{10} = 101010_2$ .

42	0
21	1
10	0
5	1
2	0
1	1

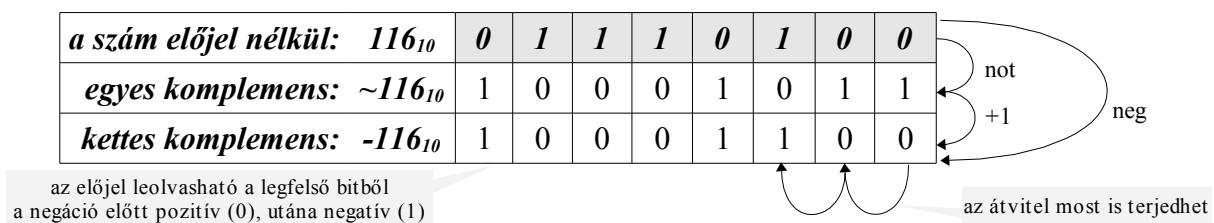
Bináris számokat összeadni a tízes számrendszerben megszokottakhoz hasonlóan lehet. Elindulunk az utolsó pozícióról, és összeadjuk az ott található két számjegyet. A két számjegy összege lesz a két szám összegének utolsó számjegye. Innen a következő pozícióra lépünk, és folytatjuk az összeadást, azonban innentől még egy számjegyet is hozzá kell adnunk az adott pozícion szereplőkhez. Ez az átvitel, ami 1, ha az előző összeadás túlcordult, vagyis az eredménye már nem fert el egy számjegyben, különben pedig 0.



## Előjeles egész számok

A nemnegatív számok alakja ebben az ábrázolásban megegyezik az előjel nélküli ábrázolás szerintivel. Az összes lehetséges ilyen szám közül most azonban csak azokat a számokat tekintjük érvényes pozitív számnak, amelyeknek a legfelső bitje nulla.

A negatív számokat a következőképpen kaphatjuk meg a pozitív számokból. Írjuk fel az előjel nélkül tekintett számot binárisan, alkalmazzuk rá bitenként a not műveletet, majd növeljük meg egyet a kapott számot. Ennek a műveletnek **negáció** vagy **negálás** a neve. Az előjelet a legfelső bit mutatja. Ezzel az eljárással van egy olyan szám, amit nem kapunk meg egyetlen nemnegatív számból kiindulva sem: az egyes bittel kezdődő, onnan végig csupa nullát tartalmazó szám. Ennek az értéke egyetlen kisebb, mint a fent leírt módon ábrázolható számok közül a legkisebb; n bites szám esetén az értéke pontosan  $2^{n-1}$ .



Tekinthetjük úgy is, hogy az n biten ábrázolt számok a modulo  $2^n$  maradékosztályokat adják ki: az előjel nélküliek a  $0..2^n-1$  reprezentánsokkal, az előjelek a  $-2^{n-1}..2^{n-1}-1$  reprezentánsokkal.

Előjeles számok összeadását a nemnegatív számokhoz hasonlóan végezhetjük el. Kivonáshoz adjuk hozzá a számhoz az összeandandó negáltját.

## Törtszámok

A törtszámok ábrázolását, amely általában ún. lebegőpontos formátumban történik, és az ezek kezelésére való utasításokat nem mutatjuk be részletesen a munkafüzetben. A lebegőpontos számok ábrázolását a numerikus analízis tárgyalja.

## Karakterek, stringek

Minden egyéb adathoz hasonlóan a karaktereket is bitekkel kell kódolnunk. Először is rögzítenünk kell, milyen karaktereket szeretnénk ábrázolni: ez a **karakterkészlet**. minden karakternek adunk egy sorszámot, ez a **karakter kódja**. Ahhoz, hogy különböző számítógépeken is ugyanazt a tartalmat kódolja ugyanaz a szöveg, meg kell állapodnunk a karakterek **kódolásában**: melyik karaktert milyen bitsorozattal ábrázoljuk. Egyszerűbb a helyzetünk, ha minden karakter kódja azonos számú bitet tartalmaz (pl. nyolc bites ASCII, UCS-2), de helytakarékkossági okokból lehet változó bithosszon is kódolni (pl. UTF-8). minden esetben követelmény azonban, hogy az ábrázolt formából egyértelműen meg lehessen állapítani, mi a karakter kódja, abból pedig azt, hogy melyik karakterről van szó.

A számítástechnika korai éveiben nem volt szabványosítva, hogy a karaktereket hogyan kell kódolni, ezért a különböző architektúrajú gépek között nehéz volt adatcsereit bonyolítani. Ezen sokat segített az 1967-ben megjelent ASCII kódolás, ami 7 bitben rögzítette a karakterkódok hosszát. Az ASCII karakterkészlete tartalmazza a számjegyeket, a latin kis- és nagybetűket, vezérlőkaraktereket, pl. sorvége, csengő, valamint különböző írásjeleket, pl. szóköz, pont, kérdőjel, aláhúzás. Később kiderült, hogy a felhasználóknak szüksége van a fentieken kívül a saját nyelükben előforduló, a latintól eltérő karakterekre is, ezért többféleképpen kiegészítették 8 bitesre: az újonnan nyert pozíciókon az egyes régiók karakterei kerültek, pl. az ISO-8859-1 a nyugat-európai, az ISO-8859-2 a közép- és kelet-európai nyelvek (köztük a magyar) speciális karaktereit tartalmazza.

Ez jónár évig elégnek tűnt, azonban több probléma is adódott. Egyrészt ezek a karakterkódolások csak korlátozottan tettek lehetővé, hogy többféle nyelvet lehessen egy dokumentumon belül használni. Amennyiben nem volt feltüntetve, melyik kódolást alkalmazza a dokumentum, téves választás esetén egyes karakterek helytelenül jelentek meg. További hiányossága volt ennek a rendszernek, hogy a lefedett nyelvek még mindig meglehetősen kevesen voltak: a távol-keleti nyelveket nem támogatta egyik kiterjesztés sem, ezekhez ismét más kódrendszer születtek.

A jelenlegi legátfogóbb karakterkódolási szabvány az 1991 óta létező Unicode, illetve ISO/IEC 10646. A kettő számunkra nem számottevő különbségektől eltérően megegyezik: a karakterkészlet és a kódolás ugyanaz a két rendszerben. Ez a karakterkészlet arra törekszik, hogy a világ összes létező és volt karaktert ábrázolni lehessen vele, azonban fenntartja a kompatibilitást az ASCII ISO-8859-1-es kiterjesztével is, melyet a Unicode teljesen tartalmaz. Az Unicode-hoz többfajta kódolás létezik: a legismertebb a változó kódhosszú UTF-8. Ennek megvan az előnye, hogy ha olyan szöveget kódolunk el vele, amely csak legfeljebb 127-es kódú ASCII karaktereket tartalmaz, akkor a kapott szöveg megegyezik az ASCII kódolással kapott szöveggel. Jelenleg a szabvány 5.0-ás verziója a legfrissebb, és folyamatosan fejlesztik.

karakterkód		vezérlőkarakter	karakterkód		karakter
0	00	szöveg vége	32	20	szóköz
10	0A	új sor	48 .. 57	30 .. 39	0 .. 9
13	0D	kocsi vissza	65 .. 90	41 .. 5A	A .. Z <sup>2</sup>
			97 .. 122	61 .. 7A	a .. z

ASCII szövegekben a sorvégét Unix rendszereken **0xA**, Internetes protokollokban és Windows alatt **0xD 0xA** jelzi; a Unicode szabványban mindenki, sőt, más kombinációk is jelezhetik a sor végét.

Szövegek kódolásánál a karakterek kódjai sorban következnek egymás után. Kétféle konvenció van arra nézve, meddig tart a szöveg: a „C konvenció” szerint az első 0 kódú karakterig, a „Pascal konvenció” szerint pedig a szöveg kezdete előtt el van tárolva a hossza is.

<sup>2</sup> Egyéb kódolásokban (pl. az IBM nagygépeken használatos EBCDIC) nem garantált, hogy a karakterek kódjai hézagtalanul követik egymást.

## Feladatok

1. Mekkora értékek tárolhatóak egy bitvektorban? Add meg általánosan is!

bitek száma	számrendszer	előjel nélkül		előjelesen	
		legkisebb érték	legnagyobb érték	legkisebb érték	legnagyobb érték
8	hexadecimális				
	decimális				
16	hexadecimális				
	decimális				
32	hexadecimális				
	decimális				
$4 \cdot n$	hexadecimális				
	decimális				

2. Töltsd ki az üres cellákat a táblázatban!

bitek száma	bináris	decimális		hexadecimális
		előjel nélküli	előjeles	
8	1100 1111			
8				6B
8			-24	
16	1001 0110 0001 1110			
16				20 A5
16			-11 721	

3. Végezd el az alábbi műveleteket!

$$\begin{array}{r} 1001 \ 1101 \\ \text{and} \ 1100 \ 0111 \end{array}$$

$$\begin{array}{r} 0110 \ 1110 \\ \text{or} \ 1000 \ 0000 \end{array}$$

$$\begin{array}{r} 0101 \ 1010 \\ \text{xor} \ 1111 \ 0001 \end{array}$$

$$0101 \ 0110 \ 1101 \ 1101_2 + 1C \ E3_{16} = \underline{\hspace{2cm}}_2 = \underline{\hspace{2cm}}_{10} = \underline{\hspace{2cm}}_{16}$$

$$16 \ 537_{10} - 0000 \ 0011 \ 1010 \ 1100_2 = \underline{\hspace{2cm}}_2 = \underline{\hspace{2cm}}_{10} = \underline{\hspace{2cm}}_{16}$$

$$54 \ 32_{16} \text{ or } 38 \ 529_{10} = \underline{\hspace{2cm}}_2 = \underline{\hspace{2cm}}_{10} = \underline{\hspace{2cm}}_{16}$$

4. Írd le a saját nevedet ASCII kódolással!

5. Írd le a „Hello Világ” szöveget ASCII kódolással!

## Az architektúra alapjai

### A memória szervezése

A memória bitekből épül fel, de a biteket nem tudjuk külön-külön elérni: a legkisebb egység a nyolc bit együtteséből felépülő **oktet**, amit általában **byte**-nak neveznek. A byte-ok sorban helyezkednek el a memóriában, mindenek rendelkezik egy 32 bites sorszámmal, amit a byte **memóriacímének** nevezünk. Ez a **lineáris memóriamodell**; a hardver és az operációs rendszer elfedi előlünk a bonyolultabb **ssegmentált modellt**.

A processzor és a memória közötti adatforgalom három fő vezetékkötéget segítségével zajlik, amelyeket **síneknek** vagy **buszoknak** nevezünk. A memóriához való hozzáférés menete vázlatosan: a processzor először elküldi a **címsín**, hányas sorszámú cím tartalmához fogunk hozzáérni, és a **vezérlősínen** (I/O buszon) a járulékos információkat, például az adatforgalom irányát<sup>3</sup> és az átvitelre váró adat méretét, ami lehet **byte**, **szó** (**word**, 2 byte) vagy **duplaszó** (**dword**, double word, 4 byte). Ezután maga az adatátvitel következik az **adatsín**en.

Mivel a vezetékek egyszerre csak egy bitnyi információt tudnak szállítani, és azt is csak egy irányban, a processzor kommunikációs képességeit behatárolja, hogy az egyes sínek hány vezetéket tartalmaznak. Az adatsín szélességét az architektúra **gépi szóhosszának** hívják. A processzoron belül, a regiszterekben tárolható adatok méretét **belső szóhossznak** nevezzük. Az x86 architektúrán belül a 32 bites szervezésű gépek gépi és belső szóhossza, valamint a címsín szélessége egyaránt 32 bit. Az architektúra korai tagjaiban a gépi szó még 16 bites volt, a „szó” ebből a történelmi okból maradt meg 386-ostól felfelé ekkorának. Tehát a 32 bites processzorokon a gépi szó a duplaszó, és ha tehetjük, akkor ekkora méretű adatokkal dolgozunk.<sup>4</sup>

Az architektúra egy, a programozó számára kevésbé intuitív tulajdonsága, hogy a byte-nál nagyobb méretű adatok byte-jait ún. **little endian** sorrendben tárolja. Ez azt jelenti, hogy a legkisebb helyi értékű byte kerül a legkisebb címre, az alulról következő byte a soron következő címre stb. Ez azt jelenti, hogy a szám leírt alakjához képest, ha a memóriát balról jobbra növekedő címeken képzeljük el, éppen fordított sorrendbe kerülnek az adatok. Más architektúrák éppen ellenkezőleg, **big endian** sorrendben írnak a memóriába (akkor a helyi értékek csökkennek a címek növekedtével). Egyes architektúrákon átprogramozható futás közben, melyik módon működjenek.

Mivel a memóriacímek is csak számok, ezért a memóriában magában is el tudunk tárolni egy másik memóriacímét. Amikor egy 32 bites adatot nem pusztán számként, hanem memóriacímként értelmezünk, akkor az adatot **mutatónak** (**pointernek**) nevezzük, és ezek segítségével építhetőek fel láncolt adatszerkezetek.

Példa	C15AEF4C	C15AEF4D	C15AEF4E	C15AEF4F	a byte memóriacíme
...	0101 0000 50	1110 1111 EF	0101 1010 5A	1100 0001 C1	byte értéke (bináris) byte értéke (hexa)

Little endian: a fenti C15AEF4C címen található duplaszó értéke nem 50EF5AC1, hanem C15AEF50.  
Mutató: ha mutatóként értelmezzük a duplaszót, akkor az utána következő byte-ra mutat.

Az adatok az **adatszegmensben** helyezkednek el, aminek a kezdetét a **section .data** direktíva vezet be. Adatot úgy lehet elhelyezni, hogy először leírjuk, mekkora méretű adatokkal foglalkozunk, majd leírjuk magukat az adatokat vesszőkkel elválasztva. A lehetséges adatmérétek: **db**, **dw** és **dd**, sorra byte, szó és duplaszó méretű adatokat jelentenek.

Az elhelyezett adatokat érdemes **címkével** is ellátni, mert ez megkönnyíti később a hozzáférést. A címkét a sorban csak szóközök és tabulátorok előzhetik meg; a címke neve tartalmazhat karaktereket, számjegyeket (számjeggyel nem kezdődhet) és pontot. A címke végére lehet egy kettőspontot tenni, de ez nem része a címke nevének. Amennyiben sok hasonló adatot szeretnénk elhelyezni, az adatméréket jelzése elő írhatunk egy **times mennyiséget** előtagot (a *mennyiséget* egy konstans, pl. 15), ami a megadott mennyiséget helyez le az adatokból.

<sup>3</sup> processzor → memória vagy memória → processzor

<sup>4</sup> néhány éve megjelentek a legújabb, 64 bites processzorok is, amelyeken a gépi szó mérete 64 bit

## Példa

```
section .data

cimke dd 1, 2, 0AF4B551Ch
        times 4 db 1 ; adat elhelyezése külön címke nélkül
        db 1, 1, 1, 1 ; hatása megegyezik az előző soréval: négy byte-nyi egyes
szoveg db "ASCII szoveg", 0xA, 0
```

A fentiek hatására az alábbi byte-ok keletkeznek. Aláhúzás jelöli az egy egységként definiált byte-okat.

01	00	00	00	02	00	00	00	1C	55	4B	AF	01	01	01	01
01	01	01	01	41	53	43	49	49	20	73	7A	6F	76	65	67
														0A	00

Sokszor előfordul, hogy tudjuk, szükségünk lesz valamekkora adatterületre, de annak tartalmát kezdetben még nem töltjük fel. Ezek számára az inicializálatlan adatszegmensben tarthatunk fenn helyet. Ennek a szegmensnek a kezdetét a **section .bss** direktíva jelzi. Itt nem helyezünk el adatokat, csak tárterületet tartunk fenn számukra: **resb mennyiségek, resw mennyiségek, resd mennyiségek** leírásával jelezhetjük, hogy az aktuális pozíciótól kezdődően adott mennyiségű byte-ra (resw esetén kétszer, resd esetén négyszer annyira) lesz szükségünk. Itt szintén érdemes címkéket alkalmazni.

A programkód írása során az inicializált és inicializálatlan adatszegmensbe tartozó címkék használata nem különbözik. Lényeges különbség viszont, hogy a program futásának a kezdetén ezeknek a területeknek a tartalma nem ismert, és amíg nem töltöttük fel, ismeretlen értéket (társzemétet) tartalmaznak.

## Példa

```
section .bss

tomb resb 1024 ; 1024 inicializálatlan byte lefoglalása, a kezdetének címkéje tomb
fajlleiro resd 1 ; a fajlleiro címkén 4 byte érhető el
```

A nasm címkéi kétfajták lehetnek. A **globális címkék** karakterrel kezdődnek. A **lokális címkék** ponttal kezdődnek, és a teljes alakjukat úgy kaphatjuk meg, ha hozzáfűzzük a lokális nevet az előtte utoljára definiált globális címke nevéhez. Ez azért hasznos, mert ugyanazt a nevet újra fel lehet használni: globális nevet általában az alprogramok belépései pontjai és az adatszerkezetek szoktak kapni, lokális nevet az alprogramokban előforduló vezérlési szerkezeteknek és az adatszerkezetek komponenseinek adunk. Amíg egy globális címke hatókörében vagyunk, az alatta definiált lokális címkére a lokális névvel lehet hivatkozni, azonban egy másik globális címke hatókörében ki kell írni a lokális címke teljes nevét – ámbár ha más globális címke alól szeretnénk a címkére hivatkozni, akkor érdemes megfontolni, hogy globális nevet adjunk neki.

## Példa

```
globalisCimke
.lokalisCimke ; teljes alakja: globalisCimke.lokalisCimke
```

*Fontos: a memória nem „tud” arról, hogy mi hogyan szeretnénk értelmezni a tartalmát, nekünk kell arra ügyelnünk, hogy minden megfelelően kezeljük azt. Ha pl. egy adatelem méretét dw adja meg, majd megpróbálunk belőle byte hosszan olvasni, akkor nem kapunk hibát, hanem sikeresen kiolvassuk az ott elhelyezkedő szó alsó felét (mivel a little endian tárolás miatt az kerül az első byte-ra).*

## A regiszterek

A legtöbbet használt adattárolók a **regiszterek**. Ezek a memória kiemelt szereppel rendelkező részei, melyek a memória többi részétől elkülönülnek mind fizikailag (a processzor belséjében helyezkednek el, és ezért nagyon gyorsan elérhetők), mind kezelésüket tekintve.

a regiszterek részeinek nevei

A programozás során nyolc **általános célú regisztert** használhatunk. Ezek 32 bitesek, neveik **eax, ebx, ecx, edx, esi, edi, esp** és **ebp**. Mindegyiknek elérhető külön regiszterként az alsó fele (a 15..0-adik bitekből álló részregiszter): **ax, bx, cx, dx, si, di, sp** és **bp**. Ezek közül az első négy regiszter felső (15..8-adik bit) és alsó fele (7..0-adik bit) is egy-egy névvel ellátott, 8 bites regiszter: **ah, al, bh, bl, ch, cl, dh, dl**<sup>5</sup>. A regiszterek közül leggyakrabban a 32 biteseket fogjuk használni, és esetenként, például karakteres adatok kezelésére, a 8 biteseket.

Van két olyan regiszter, amelyet nem lehet közvetlenül elérni, azonban a számítógép működéséhez elengedhetetlen a létek. Az **utasításmutató**, **eip** (*instruction pointer*), azt mutatja, hogy melyik memóriacímről folytatódik a végrehajtás, vagyis a memória melyik címéről kell a processzornak felolvasnia a következő utasítás kódját. A tartalma automatikusan változik, általában a közvetlenül az előző végrehajtott utasítás utáni címre mutat, kivéve az ugró utasítások esetén, amikor az ugrás céljának címét kapja értékül.

A **jelzőbitek regisztere** (eflags) az utoljára elvégzett számításról, illetve a processzor aktuális állapotáról tárol információkat.<sup>6</sup> Bitjei közül az alábbiak változhatnak meg aritmetikai műveletek eredménye szerint.

Jel	Név		Mikor egy az értéke?	Jel	Név		Mikor egy az értéke?
Z	zero	nulla	Az eredmény nulla.	S	sign	előjel	A szám negatív előjelesen.
C	carry	átvitel	Előjel nélküli (túl-/alul-)csordulás.	O	overflow	túlcordulás	Előjeles (túl-/alul-)csordulás.
P	parity	paritás	Az eredmény páros.				

## Az utasítások szerkezete

**Az utasítások** a gép működésének elemi egységei: vezérlik a processzort, milyen műveletet végezzen az adatokkal, illetve melyik utasítással folytassa a végrehajtást. Felépítésüköt tekintve egy **mnemoniknak** nevezett kulcsszóból és megfelelő számú, vesszővel elválasztott **operandusból** (paraméterből) állnak. Az operandusok számát a mnemonik után alsó indexben jelöljük az utasítások bemutatása során. Egy sorba legfeljebb egyetlen utasítás írható. Az utasítások a **kódszegmensben** találhatóak, ennek a kezdetét a **section .text** direktíva jelzi.

A kétoperandusú utasítások, ha az utasítás leírása nem mond mást, az általuk elvégzett művelet eredményét az első operandusban tárolják el, ezért arra **célként**, a második operandusra pedig **forrásként** fogunk hivatkozni. Az operandusok lehetséges kombinációi a munkafüzet végén levő táblázatban vannak összefoglalva. Két kivételtől (movsx, movzx) eltekintve teljesül továbbá minden ismertetett kétoperandusú utasításra, hogy *a két operandus hosszának meg kell egyeznie*: mindkettőnek vagy byte-osnak, vagy szavasnak, vagy duplaszavasnak kell lennie.

<sup>5</sup> Az 'e' betű a regiszterek elején azt jelöli, hogy ezek a 16 bites regiszterek kiterjesztett (extended) változatai, mert a számítógépcsalád első gépein még csak a 16 bites regiszterek voltak megtalálhatóak. A 'h' és 'l' a „felső” és „alsó” (high és low) szavak rövidítései.

6 Egyes architektúrákon (pl. IBM 360) az utasításmutató regiszter is tartalmaz jelzőbiteket.

## Konstansok

Szám konstans leírásakor el kell döntenünk, milyen számrendszerben ábrázoljuk.

- Ha **decimális** konstansról van szó, nem kell külön jelölést alkalmaznunk, csak leírjuk a számot: **157**.
  - Bináris** konstans esetén egy 'b' betűt írunk a szám végére: **001011101b**.
  - Hexadecimális** konstansokat kétféleképpen tudunk írni.
    - A '**0x**' **prefix** után írjuk le a konstanst: **0xABCD**.
    - A konstans után '**h**' **posztfixet** illesztünk: **8CDh**.
      - Ha a szám betűvel kezdődik, nulla prefixet kell írnunk a szám elő: **0F352h**.
- Enélkül ugyanis előfordulhat, hogy valamelyik regiszter, például ah nevével ütközik a leírt alak.

**Karakterstring** konstansokat aposztrófok ('') vagy idézőjelek ("") közé írhatunk. Ezek értéke megegyezik azzal a bináris száméval, amit úgy kapunk, hogy a karakterek nyolc bites ASCII kódjait összefűzzük. Itt, a numerikus értékekkel ellentétben, nem számít, hogy little vagy big endian üzemmódú-e a számítógép, minden abban a sorrendben tárolódnak a byte-ok, ahogy a stringben szerepelnek.

Lehetőség van arra, hogy a konstansokból kifejezéseket alkossunk, amelyeket a nasm assembler *fordítási időben*, 32 biten ábrázolva kiértékel. A következő operátorok állnak a rendelkezésünkre, csökkenő precedencia-sorrendben.  
*Fontos: ez csak konstansokra vonatkozik; regisztereket és memóriatartalmakat tartalmazó kifejezéseket csak utasításokkal tudunk kiszámítani.*

zárójelek	( )
bitenkénti negáció	~
negatív előjel	-
szorzás	*
összeadás, kivonás	+ -
bitenkénti eltolás	<< >>
bitenkénti és, vagy, kizárá vagy	&   ^

A konstans hosszának értelmezésében szerepe lehet annak is, hogy mi a másik paraméter: **mov eax, 4** ugyanazt jelenti, mint **mov eax, 0x00000004**, azaz eax teljes tartalmát átírja, a második bit 1 lesz, a többi 0.

Példa	mov ecx, 183	mov esi, 0x554C	mov eax, ebx + 5	mov dl, [adat] / 2	Hibás
	mov dl, 101101b	mov ah, 0Ah	mov cl, Eh		
	mov al, 'a'	mov ax, 5 + 8 / 4			

## A memória címzése

A memória tartalmának hozzáférésére sokféle módszer közül válogathatunk. Általánosan igaz mindegyiknél, hogy azt a címet, ahol az adatelem elkezdődik a memóriában, szöges zárójelek közé írjuk; ezt az alakot **memóriatartalomnak** nevezzük. A memória-hozzáférésnél fontos, hogy meg kell adnunk azt is, milyen hosszúságú adattal dolgozunk, mert önmagában a cím csak azt mutatja meg, hol kezdődik a memóriában az adat. Ezt a **byte**, **word** vagy **dword** kulcsszavakkal lehet megadni. A legegyszerűbb címzési mód, ha ismerjük a konkrét memóriacímet, ahová írni szeretnénk, ekkor egyszerűen leírjuk szöges zárójelek között, elé pedig az adat hosszát: **word [11101b]** vagy **dword [0x1234ABCD]**. Ha egy utasítás egyik operandusról egyértelműen kiderül annak hossza a másik operandus alapján, a hosszinformáció elhagyható, *különben nem*.

Nem egyértelmű a következő utasítás.	mov [1101011b], 18	Hibás
A második operandusról nem derül ki, hogy byte, szó vagy duplaszó hosszan értelmezendő-e a 18 konstans.		
Meg kell adni az operandus hosszát.	mov [1101011b], dword 18	
Az operandus hosszát megadhatjuk így is.	mov dword [1101011b], 18	

Egyoperandusú utasításnál nincs lehetőség ilyen rövidítésre.	inc [794h]	mul [0B11h]	Hibás
Ekkor ki kell írni az operandus hosszát.	inc dword [794h]	mul byte [0B11h]	

Általában nem ismerjük számszerűleg a memóriacímet, mert kézzel kiszámolni nehézkes lenne. Ennek megkönyítésére lehetőség van a memória egyes pontjaihoz címkéket hozzárendelni. Címkét definiálni úgy lehet, hogy a forráskódban a sor elejére leírjuk a címke nevét. Ha a forráskód egy pontjain leírjuk a címke nevét, az ekvivalens azzal, mintha azt a memóriacímet írtuk volna le konstansként, ahol a címkét definiáltuk. Természetesen két különböző címke nem kaphatja ugyanazt a nevet.

A szögletes zárójelen belül az alább felsoroltak közül egy vagy több összetevő összege állhat.

- egy 32 bites regiszter
- egy skálázott (konstans szorzóval ellátott) 32 bites regiszter, a skálatényező 1, 2, 4 vagy 8 lehet
- egy 32 bites konstans<sup>7</sup>

Példa	<code>mov eax, [cimke]</code>	<code>mov al, [ebx+esi]</code>	<code>mov [1101011b], 18</code>	<code>inc [794h]</code>
	<code>mov eax, [ebx]</code>	<code>mov eax, [edx+4*edi]</code>	<code>mov [eax-ecx], edx</code>	<code>mul [0B11h]</code>
	<code>mov esi, [esi]</code>	<code>mov edx, [adat+8]</code>	<code>mov eax, [[esi+8]+4]</code>	<code>mov edi, [al]</code>
	<code>mov eax, [ebp-4]</code>			

Hilás

## Adatmozgatás

Alapvető igény, hogy adatokat mozgassunk. Ezt a **mov<sub>2</sub>** utasítással lehet elérni. Hatására a céloperandus felveszi a forrás értékét. Lehetőség van arra is, hogy kisebb méretű adat kiterjesztésével töltünk fel regisztert vagy memóriatartalmat. Ezt tehetjük előjel nélkül: **movzx<sub>2</sub>** (ekkor nullákkal töltődik fel a bővebb felső rész), vagy előjelesen: **movsx<sub>2</sub>** (a kisebb méretű adat legfelső bitjének értékével töltődik fel a maradék). Végrehajtásuk után a cél értéke megegyezik a forrás értékével előjel nélkül, illetve előjelesen. Amennyiben meg akarjuk cserélni a forrást és a célt, az **xchg<sub>2</sub>** utasítást használhatjuk. Ennek az utasításnak egyik operandusa sem lehet konstans.

## Logikai utasítások

A logikai utasítások: **not<sub>1</sub>**, **and<sub>2</sub>**, **or<sub>2</sub>**, **xor<sub>2</sub>** (negáció, és, vagy, kizáró vagy) bitenként valósítják meg a fent leírt logikai műveleteket az operandusokon. Az and segítségével törölni (nullára állítani) lehet kiválasztott biteket: olyan konstanst kell második operandusnak adni, amely pontosan a törlendő pozíciókon tartalmaz nullát, a többin egyest. Az or-ral éppen fordítva, beállítani lehet biteket, ha a konstans beállítandó pozícióin egyest, a többin nullát tartalmaz. Az xor pedig megfordítja azokat a biteket, amelyek egyesre vannak állítva.

**Regisztert úgy is törölhetünk, hogy xor-oljuk a regisztert önmagával.** Ez pontosan azokat a biteket fordítja meg a regiszterben, amelyek be vannak állítva benne, vagyis az eredmény valóban nulla.

A **setfeltétel<sub>1</sub>** utasítások egyetlen, byte méretű operandusukat 1-re állítják be, ha teljesül a megadott feltétel, 0-ra, ha nem. A feltételek lehetséges alakjait a feltételes ugrásoknál írjuk le részletesen; ezeknek az utasításoknak az alakja annyiban tér el a feltételes ugró utasításokétől, hogy 'j' helyett 'set' a mnemonik eleje.

## Bitléptető és bitforgató utasítások

Ezek az utasítások (**sar<sub>2</sub>** és **shl<sub>2</sub>**, **sal<sub>2</sub>**, **shr<sub>2</sub>**, **rol<sub>2</sub>**, **ror<sub>2</sub>**, **rcl<sub>2</sub>**) kétoperandusúak, a második paraméterük konstans vagy a cl regiszter lehet. A működésük szerint a biteket léptetik annyi pozícióval, amennyit a második operandus megszab. A léptetés iránya a mnemonik utolsó betűjétől függ: ha 'l', felfelé (left, azaz balra), ha 'r', lefelé (right, jobbra). Ha az első betű 'r' (rotate), akkor forgatásról van szó, vagyis a regiszter végén túlcorduló, kilépő bitek a másik oldalról jönnek be. Az sar utasítás esetén a legfelső bit értéke forgatódik be felülről, az összes többi léptető utasításnál 0 bitek lépnek be.

**Kettőhatvánnyal való szorzást elvégezhetünk léptető műveletek segítségével:** előjel nélküli szorzásra és osztásra az **shl** és **shr**, előjeles szorzásra és osztásra a **sal** és **sar** utasítások használhatóak. Az operandus azt adja meg, kettő hányadik hatványával szorzunk/osztunk. Könnyebb megértéshez lásd a munkafüzet végén szereplő ábrákat.

<sup>7</sup> Konstanst látszólag ki lehet vonni, ez azonban valójában egy modulo  $2^{32}$  összeadásként jelenik meg.

## Aritmetikai utasítások

Az **add<sub>2</sub>** és a **sub<sub>2</sub>** segítségével lehet összeadni és kivonni. Mindkettő működik előjeles és előjel nélküli számokra. Az eggyel való növelés és csökkentés annyira gyakori, hogy erre külön van egy-egy utasítás: **inc<sub>1</sub>** és **dec<sub>1</sub>**.

Szorozni a **mul<sub>1</sub>** *egyoperandusú* utasítással lehet előjel nélkül, az **imul<sub>1</sub>** segítségével pedig előjelesen. Ezek egy regisztert vagy memóriatartalmat kapnak; az operandusuk nem lehet konstans. Az operandus értékével megszorozzák eax-nak az operandus hosszával megegyező méretű részét (al, ax vagy eax) és a szorzatot elhelyezik ax-ben, dx:ax-ben vagy edx:eax-ben. Ez utóbbit kettő azt jelenti, hogy az eredmény felső fele kerül (e)dx-be, az alsó fele pedig (e)ax-be; a két regisztert átmenetileg egy nagyobb regiszter két felének képzelhetjük el. Erre azért van szükség, mert a szorzat közel kétszer akkora helyet is igényelhet, mint a tényezők.

Példa

mov eax, 4141659 mov ebx, 2356781 mul ebx	mov eax, 0xA7F04BFF ; ekvivalens eredményt ad ezzel mov ebx, 2356781 ; mert $414659 \cdot 2356781 =$ mov edx, 8E0h ; $8E0_{16} \cdot 2^{32} + A7F04BFF_{16}$
---	--

Osztani a **div<sub>1</sub>** és az **idiv<sub>1</sub>** *szintén egyoperandusú* utasításokkal lehet. Ezknél a forrás és a cél éppen fordítva van a szorzásokhoz képest (itt is a cél méretétől függően). Továbbá az osztás maradékát is megkapjuk, az operandus méretétől függően edx-ben, dx-ben illetve ah-ban.

Értéket negálni a **neg<sub>1</sub>** utasítással lehet. A negáció műveletének menetét lásd az előjeles számokat leíró részben.

## Vezérlésátadás

Előfordul, hogy a program végrehajtását egy másik címen szeretnénk folytatni. Egy ilyen eset, amikor végrehajtottuk egy elágazás igaz ágát, és a hamis ágat, aminek a kódja közvetlenül az igaz ág kódja után következik, már nem akarjuk végrehajtani. Ilyenkor az igaz ág kódjának a végére elhelyezünk egy feltétel nélküli ugrást, aminek hatására a vezérlés a hamis ág kódja után folytatódik, vagyis a teljes elágazás kódja után. A feltétel nélküli ugrás **jmp<sub>1</sub>**, operandusa pedig az a címke, ahová az ugrás után a vezérlést juttatni szeretnénk.

Vannak olyan esetek, amikor valamilyen feltételtől függően szeretnénk csak átadni a vezérlést máshová, ha pedig nem teljesül a feltétel, a következő utasítást akarjuk futtatni. Az előbbi példánál maradva, az elágazás feltételének vizsgálatakor pontosan ez a helyzet: a teljesül a feltétel, az igaz ág kódját hajtjuk végre, ami közvetlenül az ugrás után áll, ha nem teljesül a feltétel, akkor pedig el kell ugranunk a hamis ágra. A feltétel megvizsgálását a **cmp<sub>2</sub>**, a feltételes ugrásokat pedig a **jfeltétel<sub>1</sub>** utasításokkal tudjuk megvalósítani. Ezeknek az alakja: a 'j' (ugrás, jump) után opcionálisan egy 'n' (nem, not), aztán az 'a', 'b', 'c', 'g', 'l' közül valamelyik (felett, alatt, átvitel, nagyobb, kisebb: above, below, carry, greater, less), majd opcionálisan egy 'e' (egyenlő, equal); lehetséges alakok még önmagukban a je és jne. A cmp szerepe, hogy a jelzőbők regiszterét beállítja a feltételes ugrás végrehajtásához szükséges értékekre, de a paraméterek tartalmát nem változtatja meg. A feltételes ugrások megvizsgálják a jelzőbők állását, majd ennek eredménye szerint ugranak az operandusban kapott címre, vagy folytatják a végrehajtást. A feltételek a következők lehetnek.

- e ugrik, ha az őt megelőző cmp két paraméterének az értéke azonos
- a és b ugrik, ha a cmp első paraméterének az értéke előjel nélkül nagyobb (b: kisebb)
- g és l ugrik, ha a cmp első paraméterének az értéke előjelesen nagyobb (l: kisebb)

Az 'e' betű megengedi az egyenlőséget is, az 'n' betű a feltétel tagadása.

A feltétel nélküli ugrások tetszőlegesen távoli kódra át tudják adni a vezérlést. A feltételes ugrások alapesetben csak „közelre” tudnak ugrani, ezért célszerű a **near** kulcsszót használni a feltételes ugrás mnemonikja után.

Példa

cmp eax, ebx jnb near címke	; akkor ugrik, ha eax előjel nélkül összehasonlíta ; szigorúan nagyobb (nem kisebb vagy egyenlő), mint ebx
--------------------------------	---

Az alprogramhívás eszköze a **call<sub>1</sub>** és a **ret<sub>0</sub>**, a kivételkezelésé az **int<sub>1</sub>**, melyeket a megfelelő helyeken ismertetünk.

## Programozási konstrukciók megvalósítása

### Változók deklarációja, definíciója

Amikor egy változóval dolgozunk egy programozási nyelven, minden tudnunk kell, hogy mekkora területet foglal, illetve milyen műveleteket hajthatunk végre rajta. Magasszintű programnyelveken ezek az információk a deklarációkból derülnek ki, így a fordítóprogram akkor is képes a megfelelő kódot generálni az adat elérésére, ha az egy másik fordítási egységen helyezkedik el.

Az assembly szintjéről nézve az adatok pusztán byte-sorozatok. Itt a hosszra és az elérésre vonatkozó információkat közvetlenül a generált kódban kell helyesen felhasználni. Ehhez fordítóprogram írása során fel kell használni a rendelkezésre álló adatokat, forráskód közvetlen írása során pedig ügyelni kell, hogy ne helyesen írjuk le az adathozzáférés méretét, például duplaszavasan tárolt adatot ne próbálunk nyolcbites regiszterbe tölteni.

### Értékadás

A mov (ha szükséges: movsx, movzx) utasítás segítségével lehet értéket adni egy memóriatartalomnak.

Ha az adat hossza nagyobb egy gépi szónál, akkor több mozgatásra van szükség. Ehhez az adatot fel kell bontani legfeljebb duplaszó hosszúságú részekre, és külön kell értéket adni a részeknek. A mov nem tud két memóriatartalmat operandusként fogadni, hiszen az adatsínén nem tudunk egyszerre befelé és kifelé is adatot mozgatni, a két lépés közi tárolásra az egyik általános célú regisztert használhatjuk.

A v1 és v2 változók 8 byte hosszon tartalmaznak adatokat. Valósítsuk meg a v1 := v2 értékadást!

Példa

```
mov    eax,      [v1]      ; az első 4 byte felolvasása
mov    [v2],     eax       ; az első 4 byte kiírása
mov    eax,      [v1 + 4]   ; a második 4 byte felolvasása
mov    [v2 + 4],  eax      ; a második 4 byte kiírása
```

### Goto

Ez a konstrukció szinte minden programnyelven majdnem ugyanúgy jelenik meg, mint assemblyben, nem véletlen, hogy jóval megelőzte a strukturált programozást. Megvalósítani egy jmp utasítással lehet, amelynek operandusa a célcímke.

## Kifejezések

Magas szintű programozási nyelveken könnyen lehet egyetlen sorba hosszú kifejezéseket írni, ezeket assemblyre fordítva azonban akár több képernyőnyi hosszúságú kódot is kaphatunk. A kifejezések átírását a következő módszerrel tehetjük meg. Ennek a módszernek az alkalmazásához szükség van a futási idejű verem ismeretére is, lásd 25. oldal.

Alkossuk meg a kifejezés szintaxisfáját, majd járjuk be a fát posztorder sorrendben. Ha egy konstanssal vagy memóriatartalommal találkozunk, akkor azt tegyük be a verembe. Ha műveleti jelhez érünk, annak paramétereit sorban a verem tetején találhatóak; vegyük ki őket a regiszterekbe, hajtsuk végre a műveletet a megfelelő assembly utasítással, majd az eredményt tegyük ismét a verembe. Amikor a kifejezés végére értünk, a verem tetején a kifejezés értéke található.

Komolyabb nyelv esetén a közbülső lépések során az eredmény vermelését típusellenőrzés előzi meg, például történt-e túlcordulás. A tömbök indexelése, lista következő elemére hivatkozás, és bármilyen olyan lépés, ami mutató vagy hivatkozás feloldását vonja maga után, szintén műveleti jelnek minősül, amit megvalósítani indirekcióval lehet.

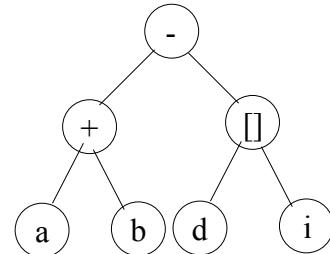
Természetesen sokkal hatékonyabb és kényelmesebb módon is ki lehet értékelni a kifejezéseket, amennyiben a verem helyett regisztereket használunk az átmeneti értékek tárolására. Amint az ember egy kis gyakorlatra tesz szert az assembly programozás terén, kézzel elég egyszerű lesz olvashatóbb, rövidebb módon leírni egy kifejezés kiértékelését. Ugyanerre a fordítóprogramok fejlett technikákat, például gráfszínezést alkalmaznak.

Az a, b, c, i és j változók 32 bitesek, d 32 bites értékeket tartalmazó tömb.

Valósítsuk meg a  $j := (a + b) * (c - d[i])$  értékadást!

Példa

		<u>a verem tartalma</u>	<u>eax</u>	<u>ebx</u>	<u>ugyanez a számítás rövidebb kódval</u>
	push dword [a]	; a			mov eax, [a]
	push dword [b]	; a b			add eax, [b] ; a + b
	pop eax	; a	b		mov ebx, [c]
	pop ebx	;	b	a	mov ecx, [i]
	add eax, ebx	;	a+b		mov edx, [d]
	push eax	;(a+b)			mov edx, [edx+4*ecx] ; d[i]
	push dword [c]	;(a+b) c			sub ebx, edx ; c - d[i]
	push dword [d]	;(a+b) c d			mul ebx
	push dword [i]	;(a+b) c d i			; a végeredmény eax-ben található
	pop eax	;(a+b) c d	i		
	pop ebx	;(a+b) c	i	d	
	mov eax, [ebx+4*eax]	;(a+b) c	d[i]		
	push eax	;(a+b) c d[i]			
	pop eax	;(a+b) c	d[i]		
	pop ebx	;(a+b)	d[i]	c	
	sub ebx, eax	;(a+b)		c-d[i]	
	push ebx	;(a+b) (c-d[i])			
	pop eax	;(a+b)	c-d[i]		
	pop ebx	;	c-d[i]	(a+b)	
	mul ebx	; feltételezzük, hogy a szorzat belefér eax-be			
	push eax	; a végeredmény a verem tetején van			



## Szekvencia

Ez a legegyszerűbb vezérlési szerkezet: csak le kell írni a programrészletek kódját egymás után.

## Elágazás

Az elágazás minden ága számára tartsunk fenn, még csak gondolatban, egy (lokális) címkét, valamint az elágazás vége számára is egyet. Az elágazások minden egy kifejezés kiértékelésével kezdődnek; igaz-hamis elágazások esetén a kifejezés egy logikai értéket ad. Ezután sorban következnek az ágak kódjai.

Az elágazás elején a feltétel vizsgálata a következőképpen történik. Kiszámítjuk a feltétel értékét (ami egy logikai érték) egy változóba. Ezután megvizsgáljuk, hogy az elágazás melyik ágába kell átadnunk a vezérlést. Az értéket egy cmp és egy feltételes ugrás segítségével tudjuk összehasonlítani; többágú elágazás esetén több ilyen utasításpárt írunk le egymás után. Kihasználható, hogy a vezérlés a következő utasításon folytatódik, ha nem teljesül a feltétel, ezért egy kiválasztott ág kódja közvetlenül az elágazás után következhet.

Gyakori eset, amikor azt kell megvizsgálnunk, hogy egy kifejezés értéke beleesik-e egy intervallumba. Ennek vizsgálatához, két cmp/jmp utasításpár szükséges: az első két utasítással vizsgáljuk meg, kisebb-e az alsó határnál, a második kettővel pedig azt, nagyobb-e a felsőnél. Mivel sem a cmp, sem a feltételes ugrások nem változtatják meg a vizsgálandó értéket, a konstrukció természetes módon konjunkcióként viselkedik.<sup>8</sup>

Írunk elágazást attól függően, hogy az a és a b változó értéke megegyezik-e.

```
    mov eax, [a]
    cmp eax, [b]
    jne .hamis.ag
```

Példa

```
.igaz.ag      ; erre a címkére nincsen feltétlenül szükség, a jobb olvashatóság kedvéért szerepel
; itt következik az igaz ág kódja
jmp .elagazas.vege

.hamis.ag
; itt következik a hamis ág kódja
; ide nem szükséges ugró utasítást elhelyezni, mert éppen az elágazás végénél vagyunk

.elagazas.vege
```

<sup>8</sup> nem túl gyakran előfordul, hogy ennél több összehasonlításra van szükség, például ha EBCDIC kódolás szerint vizsgáljuk meg, kisbetűről van-e szó. Ezekben az esetekben is fel lehet bontani a feltételeket intervallumok uniójára; sorban vizsgáljuk meg, hogy az egyes intervallumok közül beleesik-e valamelyikbe az érték.

## Rövidzásas operátorok

Kétfajta logikai vagy és logikai és szokott a programokban szerepelni. Az ún. **mohó operátorok** minden részkifejezést kiszámítják, majd az eredményekből meghatározzák a kifejezés eredményét a megfelelő logikai operátorral. A **lusta kiértékelésű** vagy más néven **rövidzásas operátorok** azonban, ha az első részkifejezésből már rögtön megadható az egész kifejezés értéke, a második részkifejezést nem számítják ki.

Egy lehetséges módszer mohó operátorok kiszámítására, hogy egyszerű kifejezéseknek tekintjük őket, a feltételeket pedig a setfeltétel utasítások közül a megfelelő segítségével állítjuk elő.

Rövidzásas operátoroknál a következő kódot írhatjuk. Ez több, azonos operátorból álló kifejezés-láncok kiszámítására alkalmas.

### konjunkciós lánc

; a két összehasonlítandó  
; meghatározása eax-be és ebx-be

cmp eax, ebx  
jfeltétel tagadása near .hamis.ag

; a fenti konstrukció  
; alkalmazása a konjunkciós lánc minden elemére

.igaz.ag ; itt szerepel az igaz ág kódja  
jmp .vege

.hamis.ag ; itt szerepel a hamis ág kódja

.vege

Valósítsuk meg az ( a < b ) && ( c - 1 < 2 \* d ) logikai feltétel kiértékelését rövidzásas operátorral!

mov eax, [a]  
mov ebx, [b]  
cmp eax, ebx  
jnl near .hamis.ag

mov eax, [c]  
dec eax  
mov ebx, [d]  
shl ebx, 1  
cmp eax, ebx  
jnl near .hamis.ag

.igaz.ag

...  
jmp .vege

.hamis.ag

...

.vege

### diszjunkciós lánc

; a két összehasonlítandó  
; meghatározása eax-be és ebx-be

cmp eax, ebx  
jfeltétel near .igaz.ag

; a fenti konstrukció alkalmazása  
; a diszjunkciós lánc minden elemére

.hamis.ag ; itt szerepel a hamis ág kódja  
jmp .vege

.igaz.ag ; itt szerepel az igaz ág kódja

.vege

Valósítsuk meg az ( a + b < 1 ) || ( c <= a & d ) logikai feltétel kiértékelését rövidzásas operátorral!

mov eax, [a]  
add eax, [b]  
cmp eax, 1  
jl near .igaz.ag

mov eax, [c]  
mov ebx, [a]  
and ebx, [d]  
cmp eax, ebx  
jnle near .hamis.ag ; az utolsó ág  
; összevonható a ;  
; jmp .hamis.ag utasítással ;  
; (a feltétel megfordul) ;

.igaz.ag

...  
jmp .vege

.hamis.ag

...

.vege

## Elöltesztelős ciklus

Elöltesztelős ciklus írásához két lokális címkére lesz szükségünk.

1. Az első címke a ciklus kezdetét jelzi, ide fogunk a ciklusmag végrehajtása után visszaugrani.
2. Ezután következik a belépési feltétel kiértékelése. Ha nem teljesül a belépési feltétel, akkor elugrunk a második lokális címkénkre.
3. Ekkor következik a ciklusmag kódja.
4. A ciklusmag kódja után egy ugró utasítással átadjuk a vezérlést az első címkénkre.
5. Végül leírjuk a második címkét, amelyen a program tovább folytatódik.

Tegyük fel, hogy van egy int i változónk, amelyet 32 biten, előjelesen ábrázolunk.

```
while ( 5 < i || i < 11 )
{
    ++i;
}
```

A fenti C programrészletet a következőképpen tudjuk megvalósítani.

```
Példa .ciklus.kezdete ; 1.
        cmp dword [i],      5 ; 2.
        jg near .ciklusmag ; 2., 5 < i teljesül
        cmp dword [i],      11 ; 2.
        jl near .ciklusmag ; 2., i < 11 teljesül
        jmp .ciklus.vege

.ciklusmag ; ez a címke a vagy rövidzárassága miatt keletkezett
        inc dword [i] ; 3.

        jmp .ciklus.kezdete ; 4.

.ciklus.vege ; 5.
```

## Hátultesztelős ciklus

A hátultesztelős ciklus nagyon hasonló szerkezetű az előtesztelős ciklushoz. Szintén két lokális címkére lesz szükségünk.

1. Az első címke a ciklus kezdetét jelzi, ide fogunk a ciklus végén visszaugrani.
2. Itt következik a ciklusmag kódja.
3. Ezután következik a folytatási feltétel kiértékelése. Ha teljesül a feltétel, akkor elugrunk a ciklus kezdetén elhelyezett lokális címkénkre, különben pedig a második címkére. A második címkére való ugrás egyszerű feltétel esetén el is maradhat, hiszen ekkor a feltételes ugráson túlhaladva a program futása ugyanott folytatódik.
4. A második címkével jelezzük a ciklus végét.

Példa: tegyük fel, hogy van két int típusú változónk, i és j, melyeket 32 biten, előjelesen ábrázolunk.

```
do
{
    --i;
} while ( 19 < i + j && i < 11 );
```

A fenti C programrészletet a következőképpen tudjuk megvalósítani.

```
.ciklus.kezdete          ; 1.
    dec dword [i]         ; 2.

    mov eax, [i]           ; 3.
    add eax, [j]           ; 3.
    cmp eax, 19            ; 3.
    jg .ciklus.kezdete    ; 3., 19 < i + j teljesül

    cmp dword [i], 11      ; 2.
    jnl .ciklus.vege       ; 2., i < 11 nem teljesül

    inc dword [i]          ; 3.

    jmp .ciklus.kezdete   ; 4.

.ciklus.vege              ; 5.
```

Példa

## Rögzített lépésszámú ciklus

Amennyiben a ciklussmagot korlátos sokszor kell végrehajtani, a következő kódot generálhatjuk. Két lokális címkét fogunk elhelyezni a kódban.

- Szükségünk lesz a ciklusszámláló eltárolására. Erre a célra egyszerű ciklusok esetén fenntarthatjuk valamelyik regisztert, összetettebb ciklusok esetén pedig lefoglalhatunk egy memóriaterületet az adatszegmensben, vagy tárolhatjuk a ciklusszámlálót a futási idejű veremben.
- Miután eldöntöttük, hol tároljuk a ciklusszámlálót, töltük fel a kezdeti értékét.
- Írjuk le az első lokális címkét, amely a ciklus kezdetét jelzi.
- Vizsgáljuk meg, hogy a ciklusszámláló túlhaladta-e már az értékhatarát. Ha igen, ugorjuk a második címkére. Amennyiben a ciklusszámláló még a határon belül van, a program futása folytatódik.
- Írjuk le a ciklusmag kódját. A ciklusmagon belül felhasználhatjuk a ciklusszámláló értékét, de ne változtassuk meg. A ciklusmagon belül elhelyezhetünk kiugrást a ciklus végére.
- Léptessük a ciklusszámlálót.
- Egy feltétel nélküli ugrással irányítsuk vissza a vezérlést az első címkére, a ciklus kezdetére.

Adott egy int n változó. Valósítsuk meg az alábbi programot assembly nyelven.

```
for ( int i = 0; i < 15; ++i )
{
    if (n % 2 == 0)      n = n / 2;
    else                  n = 3 * n + 1;
    if (1 == n)           break;
}

mov eax, 14          ; eax regiszterben tároljuk az n változót
mov ecx, 0           ; a ciklusszámláló inicializálása

.ciklus.kezdete
    cmp ecx, 15
    jnl .ciklus.vege ; a ciklusfeltétel vizsgálata

    mov edx, eax
    and edx, 1          ; a feltétel vizsgálatához csak az utolsó bitre van szükségünk
    cmp edx, 0
    jne .paratlan

.paros
    shr eax, 1          ; előjel nélküli, kettővel való osztás
    jmp .elagazas.vege

.paratlan
    mov edx, eax
    shl eax, 1          ; előjel nélküli, kettővel való szorzás
    add eax, edx         ; ... és még egyszer az eredeti
    inc eax

.elagazas.vege
    cmp eax, 1
    je .ciklus.vege    ; break

    inc ecx              ; a ciklusszámláló növelése
    jmp .ciklus.kezdete ; ugrás a ciklusmag elejére

.ciklus.vege
```

## Leszámláló ciklus

A leszámláló ciklus olyan rögzített lépésszámú ciklus, amelyben a ciklusszámláló csökken, és a ciklusfeltételt a ciklus végén ellenőrizzük. Ha nem végzünk külön ellenőrzést előtte, akkor a cikusmag legalább egyszer lefut. Ügyelni kell arra, hogy a ciklusszámláló kezdetben ne legyen nulla, különben alulcsordulás miatt a ciklus szándékainkkal ellentében nemhogy egyszer sem, de  $2^{32}$ -szer fut le. Amennyiben nem alkalmazunk egyéb számlálót, figyelembe kell venni a ciklusmagban, hogy a ciklusszámláló felülről lefele számol.

Szerkezete nagyon egyszerű. A ciklus kezdete előtt fel kell tölteni a ciklusszámlálót. Egyetlen címkére van szükségünk a ciklus elején. Utána közvetlenül a ciklus kódja következik, majd a ciklus végén alkalmazzuk a loop<sub>1</sub> utasítást<sup>9</sup>. Ennek egy címke az operandusa, és a következő három utasítással ekvivalens működésű. Kötött, hogy ecx regisztert tekinti ciklusszámlálónak.

```
dec ecx
cmp ecx, 0
je loop_utasitás_operandusa_címke
```

Számítsuk ki az első n szám összegét ( $n \geq 1$ ).

```
int i = n;
int osszeg = 0;
do
{
    osszeg += i;
} while (--i);

mov ecx, n          ; eax regiszterben tároljuk a ciklusszámlálót
mov eax, 0          ; osszeg inicializálása

.ciklus.kezdete
    add eax, ecx
    loop .ciklus.kezdete ; a ciklusfeltétel vizsgálata
```

Példa

<sup>9</sup> Természetesen a vele ekvivalens kódrészletet is leírhatjuk. Ennek további előnye lehet, hogy ekkor ecx helyett más regisztert is választhatunk ciklusszámlálónak.

## Feladatok

1. Add meg a regiszterek tartalmát minden utasítás után! A feladatot a jelölt pontokon is el lehet kezdeni.

	reg. vagy mem.	bináris (megfelelő bithosszon)	decimális		hexadecimális
			előjel nélküli	előjeles	
a.	mov ebx, 1023	ebx			
	mov al, -100	al			
	movsx eax, bl	eax			
b.	mov eax, 1C34F6E2h	ax			
		ah			
		al			
c.	mov byte [0FAh], 01Ch	[0xFA]			
	mov byte [0FBh], 0x22	[0xFB]			
	mov ax, [0FAh]	ax			
d.	mov eax, 89ABCDEh	ax			
	mov ah, al	ah			
	mov al, 42	al			
	mov bl, al	bl			
	mov bh, al	bx			
e.	mov ax, 0FF42h	ax			
	mov bx, 0DCBAh	bx			
	sub bh, al	bh			
	xor ax, bx	ax			
		bx			
f.	mov al, 01011010b	al			
	or ah, 10101010b	ah			
g.	mov al, 120	al			
	not al	al			
	neg al	al			
h.	mov bh, 254	bh			
	inc bh	bh			
	add bh, 1	bh			
i.	mov cx, 0x01F0b				
	mov eax, -128	(eax)			
	mul cx	(dx : ax)			
	imul cl	(ax)			

2. a. Az x, y és z címen byte-os adatok vannak. Számítsd ki z-be ( not x | y ) & x értékét!  
 b. Számítsd ki eax-be ( eax – 1 ) \* 2 értékét!

3. Adj meg minél több utasítást, amely

- a. nullára állítja eax értékét, és a memória és minden egyéb regiszter tartalmát változatlanul hagyja!
- b. nem változtatja meg a regiszterek értékét (legfeljebb a jelzőbitek regiszterének kivételével)!
- c. eax 2. bitjét egyesre állítja, a többöt pedig változatlanul hagyja!
- d. az ellentétére állítja ebx legalsó (0.) és legfelső (31.) bitjét!
- e. al tartalmát egy előre adott értékről egy másik adott értékre állítja! (Például 229-ről 33-ra.)

4. Az x és y két címke, melyeken 32 bites adatokat tárolunk.

- a. Valósítsd meg az  $x := y$  értékkadást! Közvetlenül egyik címről a másikra másoló utasítás nincsen.<sup>10</sup>
- b. Cserél meg x és y tartalmát!

5. Milyen byte-okat tartalmaz sorban az adatszegmens?

```
section .data
```

```
db    55h
db    55h, 56h
db    'a', 0Ah
db    "hello"
dw    1234h
dd    89ABCDEFh
```

6. Valósítsd meg az ismertetett utasítások segítségével az *adc eax, ebx* utasítást! Az adc az első operandushoz hozzáadja a másodikat (mint az add), és még ehhez az átviteli bitet (azaz hozzáad még egyet, ha az átviteli bit be van állítva, különben az eredmény a két operandus összege).

7. Mennyi a kódrészletek végrehajtása után eax értéke?

.a		.b		.c	
xor    ebx, ebx		cmp    eax, 0		cmp    eax, eax	
inc    ebx		je     near    .vege		je     near    .vege	
jmp    .vege		mov    eax, eax		dec    eax	
dec    ebx		.vege		jmp    .c	
mov    eax, ebx		inc    eax		.vege	
.vege					

8. Ír olyan kódrészletet, amely ecx-be meghatározza eax és ebx maximumát! A számok előjel nélkül vannak ábrázolva.

9. Lineáris keresés: az adat címkétől kiindulva keresd meg az első olyan byte-ot, amelyik nullát tartalmaz, és ennek a címét add meg eax-ben!

10. Valósítsd meg assembly nyelven az alábbi C programrészletet! Az a és b változók típusa int.

```
for ( int i = 0; i < 10; ++i ) a += b;
```

11. Az n paraméter a memóriában található egy duplaszavas változóban. Számítsd ki az eax regiszterbe

- a. az első n szám összegét!
- b. n! értékét!

10 Egy ilyen utasításnak egyszerre kellene a memóriabuszon beolvasnia és kiírnia az adatot, márpedig az csak egyirányú adatforgalomra képes.

12. Adott egy duplaszavas értékeket tartalmazó láncolt lista címke. A lista egy eleme egy duplaszavas mutatóból és a duplaszavas értékből áll. A mutató a következő elemre mutat; ha nincs következő elem, nullát tartalmaz. Add meg a lista hosszát!

13. Adott a szöveg címkén egy szöveg, amelynek ismert a hossza. Fordítsd meg helyben!

14. Adott három időpont: egy-egy óra és perc, amelyek duplaszavasan vannak eltárolva a memóriában. Az első két időpont között részleges napfogyatkozás következik be. A napfogyatkozás mértéke a félidőig lineárisan növekszik, félidőben pontosan 50%, onnantól lineárisan csökken. Add meg, hogy a harmadik időpontban (amelyről feltételezhető, hogy az előző kettő között helyezkedik el) mekkora volt egész százalékra kerekítve a napfogyatkozás mértéke!

15. A teniszben az a játékos nyer meg egy ún. rövidített játékot, aki először ér el legalább 7 pontot úgy, hogy legalább 2 ponttal vezet (azaz ha az állás 7-6, akkor még nem dölt el a rövidített játék, például a második játékos megszerezheti a következő három pontot, és akkor ő nyer 7-9-re). Egy tömbben adott, hogy melyik játékos nyerte meg a soron következő játékot. A tömb hossza előre nem ismert. Add meg, ki nyerte a rövidített játékot, és milyen arányban!

## A futási idejű verem szerepe: rekurzív alprogramhívások megvalósítása

Az **alprogramok** (routines, subroutines) paraméterezhető, távolról meghívható programkód-részek. Két fajtájuk a **függvények**, melyek ezekből visszatérési értéket számolnak ki, és az **eljárások**, melyek a paraméterként kapott mutatókon keresztül megváltoztathatják a memória tartalmát. Főleg a C alapú nyelvekben a két fogalom nem különül el, ezekben mindkettőt függvénynek nevezik.

A verem célja az, hogy egyszerű módon lehessen kezelní az egymásba ágyazott alprogramhívások adatait. Akkor kap igazi jelentőséget, ha a hívások rekurzívak, azaz az alprogram futása közben ismét meghívjuk az alprogramot (általában más paraméterezővel, mint első alkalommal). Íme egy példa arra C nyelven, miért is van erre szükség.

Hívjuk meg a függvényt **fakt( 2 )** paraméterezővel.

Ekkor a vezérlés átkerül a függvény kódjának az elejére. Mivel  $2 \neq 0$ , az eljárás hamis ágát hajtjuk végre. Itt ki kell számítanunk  $fakt( 1 ) * 2$ -t, ehhez pedig ismét meg kell hívnunk a függvényt, ezúttal **fakt( 1 )** paraméterezővel. A függvény most két példányban fut: két, eltérő paraméterezővel.

A „belső” faktoriális még egy példányt elindít a függvényből, **fakt( 0 )** hívással, ami visszatér 1-gyel, majd ezt felhasználva **fakt( 1 )** is visszatér 1-gyel, és csak ekkor tud **fakt( 2 )** is visszatérni,  $2 * 1$ -el.

```
int fakt( int n )
{
    int v;
    if ( n == 0 ) v = 1;
    else
        v = fakt( n - 1 ) * n;
    return v;
}
```

Azokat a kódrészleteket, amelyek több példányban is futhatnak egyszerre, **reentránsnak** nevezzük. Azokat a memóriacímeket, ahová átadva a vezérlést elkezdhetjük a kódrészlet végrehajtását, a kód **belépési pontjainak** nevezzük. A programozó, illetve a fordítóprogram feladata garantálni, hogy a kódot csak ezeken a pontokon kezdjük el végrehajtani. A belépési pontokat címkével fogjuk megjelölni.

## A futási idejű verem használata

A verem egy hardveresen támogatott ábrázolású adatszerkezet. Adatot betenni a **push<sub>1</sub>**, kivenni a **pop<sub>1</sub>** utasításokkal lehet. A paraméter 16 vagy 32 bites lehet, 8 bites *nem*. A pop operandusa nem lehet konstans.

Példa	push eax push dword [esi]	push word 5 pop ecx	push al	pop dh	Hiba
-------	------------------------------	------------------------	---------	--------	------

A veremműveletek – 386-ostól felfelé – a következő műveletekkel ekvivalensek hatásukban.

push adat	sub esp, adat hossza ; 2, ha szó, 4, ha duplaszó mov [esp], adat értéke	pop adat	mov cél, [esp] add esp, a mozgatott adat hossza
-----------	--	----------	--

Fontos, hogy rögzítsük, hogyan kerülnek át a paraméterek a hívás helyéről az alprogramba, illetve hogyan adja vissza az alprogram a visszatérési értékét, ha van neki; ezt **hívási konvenciónak** nevezzük. A **cdecl** konvenció a paramétereket jobbról balra haladva teszi a verembe, a visszatérési értéket pedig az eax regiszterbe teszi. Ha a visszatérési érték kisebb (pl. ASCII karakter), csak ax vagy al tartalmaz értékes adatot, a felső bitek értéke figyelmen kívül hagyható; ha nagyobb struktúra, például egy rekord, eax-ban a struktúra címét adjuk vissza.

a bevezető kód az alprogram elején	push ebp mov ebp, esp sub esp, lokális paraméterek összhossza (byte)	mov esp, ebp pop ebp ret	a kilépő kód az alprogram végén
--	--	--------------------------------	---------------------------------------

Az alprogramot a fenti kódrészletekkel kezdjük és fejezzük be. Az alprogramon belül az i-edik paramétert a **[ebp + 4 + i \* 4]** címen, a j-edik lokális változót az **[ebp - j \* 4]** címen érhetjük el.

Az alprogram egy futó példányához tartozó információk összességét **aktivációs rekordnak** nevezzük. Ez tartalmazza az összes paramétert és a lokális változót. A futási idejű veremben, **veremkeretekben** tároljuk az aktivációs rekordokat. Ezek megvalósításáról szól a következő oldal.

**Konkrét példa a futási idejű verem használatára: a faktorialis( 2 ) hívás megvalósítása**

	<p>Kezdetben a verem tartalma lényegtelen számunkra. Annyit tudunk csupán, hogy a verem tetejének a címe esp-ben található.</p>	<span style="border: 1px solid black; padding: 2px;">esp</span> <span style="border: 1px solid black; padding: 2px;">a verem alja &gt;&gt;&gt;&gt;&gt;</span>
<b>push dword 2</b>	<p>Amikor meghívjuk faktoriális függvényt, a paraméterét a verembe tesszük. Ezzel esp is csökken négygyel, és a verem új tetejére mutat.</p>	<span style="border: 1px solid black; padding: 2px;">esp</span> <span style="border: 1px solid black; padding: 2px;">par<sub>1</sub>: 2</span> <span style="border: 1px solid black; padding: 2px;">...</span>
<b>call fakt</b>	<p>Ezzel az utasítással hívjuk meg a függvény kódját. Hatására a verembe bekerül a következő utasítás címe; visszatéréskor innen tudjuk majd, honnan kell folytatni a végrehajtást. Ezután a vezérlés átkerül a függvény kezdőcímére (ezt tartalmazza a faktorialis címke).</p>	<span style="border: 1px solid black; padding: 2px;">esp</span> <span style="border: 1px solid black; padding: 2px;">régi eip</span> <span style="border: 1px solid black; padding: 2px;">par<sub>1</sub>: 2</span> <span style="border: 1px solid black; padding: 2px;">...</span>
<b>add esp, 4</b>	<p>Miután végrehajtottuk a függvény kódját, a veremből visszaáll eip regiszter értéke, és a végrehajtás ennél az utasításnál folytatódik. Itt még vissza kell állítanunk a vermet az eredeti állapotába, vagyis esp-hez annyit adnunk hozzá, ahány byte-ot a paraméterekkel elfoglaltunk a veremből.</p>	<span style="border: 1px solid black; padding: 2px;">esp</span> <span style="border: 1px solid black; padding: 2px;">par<sub>1</sub>: 2</span> <span style="border: 1px solid black; padding: 2px;">...</span>
	<p>Készen vagyunk, innentől folytatódhat a program futása. Az eax regiszter tartalmazza 2! értékét.</p>	<span style="border: 1px solid black; padding: 2px;">esp</span> <span style="border: 1px solid black; padding: 2px;">...</span>

### Konkrét példa a futási idejű verem használatára: a faktorialis függvény kódja

fakt	A függvény kezdetét címke jelöli, hogy könnyen meghívható legyen.	régi eip par <sub>1</sub> : 2 ... ebp+4 ebp+8 ebp+12
push ebp mov ebp, esp sub esp, 4	A szokásos bevezető kód. Az első két sor beállítja ebp-t arra a pozícióra, ahonnan a függvény végéig nem mozdul el. Ezért ebp-t a veremkeret <b>bázisának</b> nevezzük. A harmadik sor lefoglalja a helyet a lokális változónk, v számára.	esp v régi ebp régi eip par <sub>1</sub> : 2 ... ebp-4 ebp ebp+4 ebp+8 ebp+12
push ebx	Elmentjük a függvényben használt ebx-et. A cdecl alprogramhívási konvenció szerint az alprogramok eax, ecx és edx tartalmát változtathatják meg.	esp régi ebx v régi ebp régi eip par <sub>1</sub> : 2 ... ebp-8 ebp-4 ebp ebp+4 ebp+8 ebp+12
cmp dword [ebp+8], 0 jne .hamis.ag	A második sorban szereplő elágazás feltétele és igaz ága.	
mov dword [ebp-4], 1 jmp .elagazas.vege	Az elágazás igaz ága. A v változó értéket kap, majd elugrunk az elágazás végére, hogy ne fussunk bele a hamis ágba.	esp régi ebx v: 1 régi ebp régi eip par <sub>1</sub> : 2 ... ebp-8 ebp-4 ebp ebp+4 ebp+8 ebp+12
.hamis.ag		
mov eax, [ebp+8] dec eax push eax	Kiszámítjuk és a verembe helyezzük az egy szinttel mélyebben levő függvény paraméterét. Hasonló az alprogram külső példányának fent leírt meghívásához.	
call fakt add esp, 4	A függvény meghívása, majd a hívás paraméterének eltávolítása a veremből.	
mov ebx, [ebp+8] mul ebx mov [ebp-4], eax	Az előbb eax-be kiszámított fakt( n - 1 ) és az ebx-be betöltött n szorzatának kiszámítása v-be. Feltesszük, hogy a szorzat belefér eax-be.	esp régi ebx v: 2 régi ebp régi eip par <sub>1</sub> : 2 ... ebp-8 ebp-4 ebp ebp+4 ebp+8 ebp+12
.elagazas.vege	Akármelyik ágon is jutottunk el ide, a veremkeret szerkezete ugyanúgy néz ki. Eltér v, azaz [ebp-4] tartalma.	esp régi ebx v: ? régi ebp régi eip par <sub>1</sub> : 2 ... ebp-8 ebp-4 ebp ebp+4 ebp+8 ebp+12
mov eax, [ebp-4]	A visszatérési érték eax-be kerül.	
pop ebx	A használt regiszter visszaállítása. A hamis ágban a szorzás felülírta edx-et is, azonban ezt nem kell visszaállítani az alprogram kezdeti értékére.	esp v: ? régi ebp régi eip par <sub>1</sub> : 2 ... ebp-4 ebp ebp+4 ebp+8 ebp+12
mov esp, ebp pop ebp	A feleslegessé vált lokális változókat átugorjuk, majd visszaállítjuk ebp-t is. Ekkor ebp az előző szinten levő veremkeret bázisát mutatja.	esp régi eip par <sub>1</sub> : 2 ...
ret	Visszatérünk. A program futása a <i>call faktorialis</i> hívás után következő kódsornál folytatódik.	

## A veremkeret általános szerkezete

verem	...	lokVált <sub>2</sub>	lokVált <sub>1</sub>	régi ebp	régi eip	par <sub>1</sub>	par <sub>2</sub>	...	az előző aktivációs rekordok >>>>>
ebp....	ebp-8	ebp-4	ebp	ebp+4	ebp+8	ebp+12	ebp+...		

## Parancssori paraméterátadás

A parancssori paraméterek átadása rendszerenként különbözik, azon belül szerkesztőprogramok között is eltérő lehet. Linux alatt, gcc szerkesztővel nagyon könnyen elérhetők a paraméterek, mivel a főprogram is úgy viselkedik, mint egy **int main( int argc, char\*\* argv )** szignatúrájú C függvény. Ennek megfelelően az előzőekben leírt alprogramparaméter-elérési módszerek alkalmazhatóak.

```

section .text
global main
main
    push ebp
    mov ebp, esp
    ; alprogram bevezető kódja

    mov eax, [esp+4]
    ; parancssori paraméterek száma plusz egy, [ebp+12] filenév
    ; ami argv[0], a futtatott program neve
    cmp eax, 2
    jne .nem_egy
    ; ha nem pontosan egy parancssori paramétert kaptunk, kiléünk

    mov esi, [ebp+12] ; esi = argv
    mov esi, [esi+4] ; esi = argv[1], az első paraméter

.szamlal
    mov al, [esi]
    cmp al, 0
    je .tovabb

    inc esi
    jmp .szamlal
    ; byte-onként léptetjük a mutatót, mert a szöveg ASCII karakterekből áll
    ; megszámoljuk, milyen hosszú (ASCII string)

.tovabb
    mov edx, esi
    sub edx, ecx
    ; hossz: a nulla tartalmú végpozíció – a szöveg kezdete

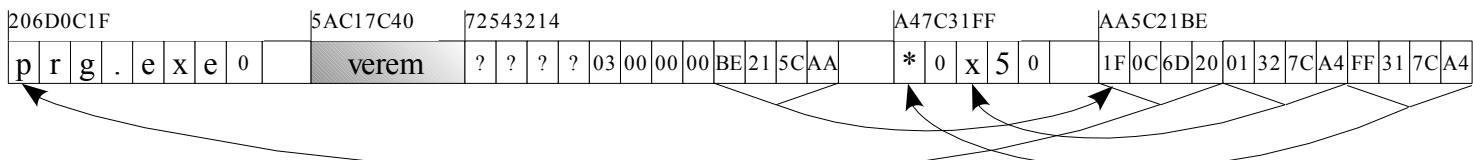
    mov eax, 4
    ; kiírás következik, bővebben lásd később
    mov ebx, 1
    mov ecx, [esp+8]
    mov ecx, [ecx+4]
    int 0x80

.nem_egy

```

### A memóriáh

verem 5AC17C40 és 72543214 hexa címek között található, azaz 98872693 duplaszó fér bele. Az argc paraméter címe 72543218 (értéke 3), argv címe 7254321B. A filenév címe, argv értéke AA5C21BE, az első paraméterre mutató argv[1] címe AA5C21C2, a második paraméterre mutató argv[2] címe AA5C21C6. A filenév szövegesen a 206D0C1F és 206D0C26 címek között helyezkedik el (a végén egy hexa nullával), az első paraméter A47C3201 és A47C3203 között, a második A47C31FF és A47C3200 között.



## A rendszerszolgáltatások elérése: fájlkezelés

művelet	név <sup>11</sup>	eax	ebx	ecx					edx				vissza eax					
fájlműveletek	létrehozás	creat	8	a fájlnév címe	elérési jogok					-				fájlleíró				
	megnyitás	open	5		0	olvas	1	ír	2	ír/olvas	elérési jogok				fájlleíró			
	olvasás	read	3		puffer címe					max. hossz (byte)				olvasott byte-ok				
	írás	write	4		puffer címe					hossz (byte)				írt byte-ok száma				
	pozicionálás	lseek	19		előjeles eltolás az edx-ben kapott pozícióhoz képest					0	elölről	2	hátulról	pozíció a fájlból				
	lezárás	close	6		-					1	az aktuális pozícióról				0	OK	1	hiba
	kilépés	exit	1	hibakód	-					-				-				

Az operációs rendszer szolgáltatásait **megszakítások** meghívásával lehet elérni. Ezek két dologban térnek el az alprogramhívástól: egyszerűt a paramétereket más konvenció szerint, a regisztereken keresztül adjuk át, másrészt nem a call, hanem az **int** utasítást használjuk, mert más védettségi szintű kódot – az operációs rendszerét – érjük el. Az int-nek paraméterül minden **80h**-t fogunk adni, ami a Linux rutingyűjteményének sorszáma a megszakítások között. Más operációs rendszerek is nyújtanak hasonló szolgáltatásokat, általában más sorszámú megszakítás alatt, és teljesen eltérő paraméterezővel.

A szolgáltatások megváltoztatják az eax, ecx és edx regisztereiket. Hiba esetén -1-et adnak vissza.

Egy fájl életciklusa a következőképpen alakul.

- létrehozás: ez a megnyitás egy speciális fajtája, a creat egy részben előre felparaméterezett open
- megnyitás
  - a megnyitandó fájlnév, amelynek a címét paraméterként kapja, egy nullára végződő ASCII string
  - az elérési jogok: egy-egy bit jelzi, hogy a felhasználó/csoporthoz köthető írhatja/olvashatja/futtatható-e a fájl<sup>12</sup>. Ezt a legegyszerűbb nyolcas számrendszerben leírni<sup>13</sup>.
  - sikeres esetén egy 32 bites leírót kapunk vissza, innentől ennek a segítségével hivatkozhatunk a fájlról. Előre megnyitott fájlok a **sztenderd bemenet**, **kimenet** és **hibafolyam**. Ezek leírói sorban 0, 1, és 2.
- írás, olvasás
  - meg kell adnunk egy mutatót a forrás-, illetve célterületre (ez a **puffer**), és azt, hány byte-ot szeretnénk átvinni.
  - a fájlhoz tartozik egy mutató, amely kezdetben a fájl elején áll. Ezt minden olvasási és írási művelet mozgatja, de közvetlenül is lehet állítani. Figyelni kell arra, hogy a fájl végéről negatív értékkel tudunk előrébb pozicionálni. A fájl végén, ha túlmutatunk rajta, az átlépett pozíciókon 0 byte-ok szerepelnek.
- lezárás: a program befejeződésekor a rendszer lezárja a nyitva maradt leírókat, de kézzel is megtehetjük.

<pre> mov eax, 4      ; kiírást fogunk igénybe venni mov ebx, 1      ; a kiírás a sztenderd kimenetre történik mov ecx, uzenet ; az üzenet címe mov edx, 12     ; a kiírandó üzenet hossza int 80h  mov eax, 1      ; kilépés következik xor ebx, ebx    ; nullás hibakód: minden rendben történt int 80h </pre>	<pre> section .data uzenet db "Hello"           db ' '           db "vilag"           db 0xA </pre>
--	---

<sup>11</sup> további információk találhatóak az /usr/include/asm/unistd.h fájlból, illetve a man 2 név parancs használatával

<sup>12</sup> például a „bárkinek bármit” 777q, a „felhasználónak bármit” 700q, a „felhasználó írhatja, bárki olvashatja” 644q

<sup>13</sup> egy nyolcas számrendszerbeli szám 0 és 7 közötti számjegyeket tartalmaz, és 'q' vagy 'o' áll a végén

## Feladatok

1. a. Írd ki a „Hello világ!” szöveget betűről betűre bővülve! Először csak egy „H” betűt írj ki és egy sorvégét, aztán a „He” szöveget és sorvégét stb.  
b. Hasonlóan bővülő módon írd ki az első parancssori paramétert!
2. a) Alkoss a saját (lefordított, futtatható) programkódját kiíró programot<sup>14</sup>! Feltételezheted, hogy a program neve ismert. Használj 1024 bites puffert a fájl tartalmának beolvasása során!  
ii) Nehezítés: a program nevét a parancssori paraméterek közül kell kinyerned.  
b) i) Alkoss a saját forrásszövegét kiíró programot! Feltételezheted, hogy a program neve ismert.  
ii) Nehezítés: a program neve a futtatott fájl neve .asm kiterjesztéssel.  
Feltételezheted, hogy a fájl neve kevesebb, mint 256 karakter hosszú.
3. Írj olyan eljárást, amely egy duplaszón elférő, előjel nélküli egész számot ír ki a képernyőre tízes számrendszerben! A számot paraméterként kapja meg az eljárás. A kiírandó string legyen lokális változó (fix hosszúságú ASCII karaktertömb); az eljárásnak ezt töltse fel, majd hívja meg a kiírást.
4. Írj programot, amely a következőképpen működik. A program két paramétert kap: egy filenevet és egy három karakteren ábrázolt, előjel nélküli számot, n-et, amely értéke ábrázolható egy byte-on. Olvasd fel a file első byte-ját, m-et. Vizsgáld meg, a file n-edik és m-edik byte-ja megegyezik-e.

Példa

A program meghívása:                   ./a.out 004 file.txt        akkor n = 4

A file.txt tartalma (hexa számok):

02 07 FD 04 ...	akkor m = 2, a második byte 7, a negyedik 4, nincs egyezés,
01 AB 7C 01 ...	akkor m = 1, az első és a negyedik byte egyaránt 1, egyeznek

5. Írj programot, amely a következőképpen működik. A program egy filenevet kap paraméterként. A file hossza hárommal osztható. A program vizsgálja meg, hogy minden harmadik byte egyenlő-e az előző kettő összegével (modulo 256, azaz 0xAB 0xCD 0x78 helyes).

6. Írj programot, amely a következőképpen működik. A program egy filenevet kap paraméterként. A program keresse meg, melyik az az első pozíció, ahol a file előlről és hátulról olvasva eltér. Feltételezhető, hogy van ilyen pozíció.

Példa

Példa: ha a file tartalma abcdecba, akkor a program eredménye 4.

7. Add meg, hogy a parancssori paraméterként kapott file a legvégén tartalmazza-e a saját nevét.
8. Add meg a következő függvények kódját:

$$\begin{aligned}f(0) &= -2 \\f(1) &= 1 \\f(n) &= f(n-1) - 2 * (f(n-2) \text{ or } f(n-3))\end{aligned}$$

$$\begin{aligned}g(0) &= 2 \\g(1) &= -1 \\g(n) &= \begin{cases} (f(n-1) - n) \text{ xor } f(n-2) & \text{ha } f(n-1) \geq f(n-2) \\ f(n-1) \text{ xor } (f(n-2) + n) & \text{különben} \end{cases}\end{aligned}$$

<sup>14</sup> A feladat jellegéből következően a kilistázott tartalom meglehetősen olvashatatlan lesz, érdemes a diff programot használni az eredmény vizsgálatára.

9. Adott egy inteket tartalmazó tárhely kezdőcíme és a hossza.

- a. Adott egy int kovetkezo() szignatúrájú függvény. Töltsd fel a tárhelyet sorban az 1, 2, 3 stb. számokkal úgy, hogy a függvény mondja meg, hány pozíciót kell az adott értékkel feltölteni!

Példa

Ha a visszatérési értékek sorban 1, 3, 2, 1, akkor a feltöltött tárhelynek így kell kinéznie: 1, 2, 2, 2, 3, 3, 4.

- b. Adott két index, melyekre teljesül, hogy  $0 < i \leq j \leq \text{hossz}$ , valamint egy int f( int ) szignatúrájú függvény. Töltsd fel a tömböt úgy, hogy tetszőleges k indexre  $f(k) + \text{tomb}[k-1]$  kerüljön a tömbbe, ha  $i \leq k \leq j$ , különben pedig k.

- c. Adott egy int permatal() szignatúrájú alprogram. Ennek segítségével permutáljuk a tömböt helyben a következőképpen. A permatal() első meghívásakor azzal a pozícióval tér vissza, ahová az eredeti első elemek kell kerülnie. A következő híváskor a visszatérési érték az előbb lecserélt elem új pozíciója lesz. Ez addig folytatódik, amíg az első pozícióhoz vissza nem jutunk.

Példa

Ha az eredeti tömb tartalma 6, 2, 9, -4, 3, a visszatérési értékek sorban 3, 5, 4, 1, 2, akkor a tömb új tartalma -4, 3, 6, 9, 2 lesz.

10. Valósítsd meg a következő függvényt!

```
int osszegzo( int* tomb, int hossz )
{
    int i, j;
    int osszeg = 0;

    for ( i = 0; i < hossz; ++i )
        for ( j = i; j < hossz; ++j )
        {
            if ( tomb[i] <= tomb[j] )      osszeg = osszeg + tomb[i];
            else      osszeg = osszeg + tomb[j];
        }
    return osszeg;
}
```

## Makrók

Az eddigiekben az assembly nyelvek fordításának utolsó fázisát tárgyaltuk, amikor az assemblér a forrásprogramból gépi kódot állít elő, amely **futási időben** működik: megváltoztatja a regiszterek értékét, alprogramokat hív meg stb. Ebben a fejezetben a **fordítási időben**, az **előfordítási fázisban** működő makrókkal foglalkozunk. Ezek a forráskód általunk megírt szövegét alakítják át; csak azután kezd el működni a gépi kódra fordító fázis, hogy a makrófordító fázis véget ért. Tartsuk végig szem előtt, hogy bármennyire is hasonló dolgokat lehet elvégezni a makrók segítségével, mint az utasításokkal, lényeges különbség, hogy a makrók nem férnek hozzá a regiszterekhez és a memoriához, és nem tudnak elugrani a kód más pontjaira.

A nasm assemblér -e kapcsolójával lehet az előfordítási fázis végeredményét kiíratni.

A **szimbólumok** a fordítóprogram által értelmezett, a forráskóban előforduló karakterek sorozatai. Már találkoztunk a szimbólumok egy fajtájával: a címek egy memóriacím szimbolikus megjelenési formái. A mnemonikok és a regiszterek nevei nem szimbólumok, hanem kulcsszavak; ezeket a neveket is fel lehet venni a szimbólumok közé is, de határozottan nem ajánlott. A **makrók** a szimbólumok paramétereitől, felüldefiniálható fajtája.

Az előfordítási fázis a következőképpen működik. Az assemblér lineárisan végighalad a kódon, és sorban azt vizsgálja, hogy talál-e makródefiníciókat vagy makróhívásokat. A **makródefiníció** tartalmazza a makró **nevét**, **paramétereit** és **törzsét**, amely függ a paramétereiktől. Innentől, ha az assemblér olyan szimbólumsorozattal találkozik, amely a makró nevével kezdődik, és megfelelő számú szimbólummal folytatódik (úgy, hogy azok értelmezhetőek legyenek a makró paramétereiként), azt makróhívásnak tekinti, és **kifejti**. Ez úgy történik, hogy a kifejtendő kódrészletet, vagyis a makró nevét és a paraméterezést, eltávolítja a forrásszövegből, majd a helyébe beszúrja a makró törzsének az aktuális paraméterek szerint meghatározott alakját. A fordítóprogram ezután folytatódik a kifejtett rész **kezdetétől**, ezért a kifejtés után rögtön annak a vizsgálata következik, hogy a beillesztett részben található-e újabb kifejtendő makróhívás.

Figyelem: az alprogram paraméterei, a makró paraméterei és a parancssorból átvett paraméterek három teljesen különböző fogalom! Adódhat feladat, mely során olyan makrót kell írni, amely paraméterül kapja, hogy egy alprogram melyik paraméterei jelentenek fájl-paramétert, és ennek megfelelően kezeli őket...

### Egysoros makrók

A makróknak egy egyszerű fajtája az **egysoros makró**. Egyszerűsége abból fakad, hogy egyetlen sorra tud csak kifejtődni, ami legfeljebb egy utasítást tartalmazhat, ezért bonyolultabb kódot nem lehet vele generálni. Az egysoros makrókat a **%define makrónév tartalom** vagy a **%xdefine makrónév tartalom** kulcsszó vezeti be, utána kell leírni a makró nevét, majd zárójelek között, ha vannak, a **formális paramétereiket**, majd a makró törzsét. A formális paraméter szintén szimbólum, egy névvel jelöli meg a paraméter pozícióját; kifejtéskor a törzsben minden előfordulása lecserélődik a makróhívás megfelelő pozícióján szereplő aktuális paramétere. A makródefiníciók **felüldefiniálhatóak**: az azonos nevű és paraméterszámú makródefiníciók közül az utoljára elemzett van érvényben.

A tartalomban érdemes megfelelően zárójelezni a paramétereiket.

Példa

```
%define helytelen(a, b)      a*b
%define helyes(a, b)        ((a) * (b))

        mov eax, helytelen(2, 3 + 4) ; erre fejtődik ki: mov eax, 2 * 3 + 4
        mov eax, helyes(2, 3 + 4)    ; erre fejtődik ki: mov eax, ((2) * (3 + 4))
```

Az első sor is érvényes utasítás, de a programozónak valószínűleg nem ez volt a szándéka.

A `%xdefine` annyiban tér el a `%define`-tól, hogy míg az utóbbit alkalmazva a makró értéke a szövegszerűen lemasolt törzs lesz, addig az előbbi a törzs kifejtett alakját veszi fel a szimbólum tartalmának, ezért az szövegszerűen nem tartalmaz már makrókat. Ennek akkor van szerepe, ha a definíció és a kifejtés helye között felüldefiniálunk egy olyan makrót, amely szerepel a törzs szövegében: a `%define` esetében ez hatással lehet az eredeti makrónak, a `%xdefine` esetében nem.

Példa

```
%define a 1
%define b a
%xdefine c a
%define a 2

        mov eax, b          ; erre fejtődik ki: mov eax, 2
        mov eax, c          ; erre fejtődik ki: mov eax, 1
```

A *nullától eltérő* paraméterszámú makrók **túlterhelhetőek** is: érvényben lehet egyszerre több olyan makródefiníció is, amelyeknek ugyanaz a neve, de eltérő a paraméterszáma. A makródefiníciók **nem rekurzívak**: ha a kifejtésen belül újra találkozik az assembler ugyanazzal a makróval, azt nem fejti ki még egyszer.

Az egysoros makrók további, speciális fajtája a fordítási időben történő **értékkadás**. A **%assign szimbólumnév érték** kulcsszó után egy szimbólumnév következik, majd a sor további része egy *fordítási időben kiértékelhető* kifejezés<sup>15</sup>, amely meghatározza a szimbólum felvett, új értékét. Ez hatásában nagyon közel áll ahhoz, mintha `%define` segítségével hoztunk volna létre egy szimbólumot, a különbség egyszerű abban áll, hogy a `%define` nem csak számokkal dolgozhat, másrészt pedig a `%assign` a definíció során **kiértékeli** a kifejezést, ezért tömörebben tudja az assembler ábrázolni. További előnye, hogy ránézésre egyből látszik róla, hogy szám jellegű szimbólummal dolgozunk.

Példa

```
%assign n 1
%assign n n+1
...      ; összesen húszszor
%assign n n+1

        %define n 1
        %xdefine n n+1
        ...
        ; összesen húszszor
        %xdefine n n+1

        mov eax, n
        ; erre fejtődik ki: mov eax, 21
        mov eax, n
        ; erre fejtődik ki: mov eax, 1+1+...+1
```

Ezzel rokon konstrukció a **konstans szimbólum**. Alakja: **szimbólumnév equ kifejezés**. Fontos, hogy a kifejezésnek *fordítási időben kiértékelhetőnek* kell lennie. A szimbólum felveszi a kifejezés értékét, mintha `%assign` segítségével definiáltuk volna, azonban innentől nem változtatható meg az értéke.

Példa

```
negyvenketto equ 4 * 10 + 2
                    mov eax, negyvenketto ; ekvivalens ezzel: mov eax, 42
```

<sup>15</sup> konstansok, makrókkal definiált szimbólumok és azokkal végzett műveletek; ha ide egy regiszter, pl. `eax` nevét írjuk, az nem hiba, de a fordítóprogram érdeklődni fog, hogy mi az `eax` szimbólum tartalma, ha nem definiáltuk

## Többsoros makrók

Bonyolultabb működés leírására alkalmasak a **többsoros makrók**. Ezek törzsét a **%macro** és a **%endmacro** (rövidíthető: **%endm**) kulcsszavak közé lehet leírni. A kulcsszavakat külön sorba kell írni; a %macro kulcsszó után a sorban következik a makró neve, a paraméterek száma, és ha vannak, az alapértelmezett paraméterek.

**Makródefiníciókat törlni** – akár egysorosakat, akár többsorosakat – **%undef** makrónév leírásával lehet.

A paraméterek számának alakjai a következők lehetnek.

- egyetlen szám: pontosan ennyi paraméterrel kell meghívni a makrót
  - két szám kötőjellel elválasztva: a paraméterszámnak bele kell esnie ebbe a (zárt) intervallumba
    - a paraméterek száma valójában a második szám; meg kell adni vesszőkkel elválasztva, milyen értékeket vegyenek fel azok a paraméterek, amelyeket a makróhívás nem adott meg explicit módon
  - egy szám, kötőjel és csillag: a paraméterszám legalább annyi, mint az első szám, felső korlát nincs
    - a makró törzsének írásakor nem ismert, hány paramétert kapott
    - az **aktuális paraméterek számát %0** tartalmazza.

A makrókat lehetséges specializálni is, azaz egyes paraméter-kombinációkhoz külön kódot is meg lehet adni.

A **formális paraméterekre**, mivel most nincsenek névvel ellátva, **%1**, **%2** stb. leírásával lehet hivatkozni a törzsben. Főleg többsoros makrók használatánál fordul elő, hogy **vesszőt is tartalmazó paramétereket** is átszeretnénk adni. Ezeket *kapcsos zárójelek* közé kell tenni híváskor, mert különben az assembler külön makróparamétereknek nézne öket. Amikor a törzsben felhasználjuk öket, már nincs körülöttük a kapcsos zárójel, ezért ha egy másik makrónak szeretnénk átadni őket, akkor ismét be kell kapcsos zárójelezni őket.

```
%macro egypt 2           %macro ketop 3           %macro hanyop 1-*  
    %1 %2  
%endm  
  
    egypt inc, eax      ketop mov, eax, ebx      hanyop 1, 2, 3, 4, 5  
;  
; erre fejtdik ki: inc eax ; erre fejtdik ki: mov eax, ebx ; erre fejtdik ki: mov eax, 5 - 1
```

**Feltételes ugró utasítások feltétel-részét** is átadhatjuk makróparaméterként, a makró törzsében pedig a j betű után fűzve megkapjuk a megfelelő feltételes ugró utasítást. A j betű után %-1 fűzésével az ellentétes hatású ugró utasítást kapjuk; természetesen 1 helyett tetszőleges sorszám szerepelhet.

```
%macro felteteles 5
    cmp %1, %2
    j%+3 %4
    j%-3 %5
%endm

        felteteles eax, ebx, ne, .nem_egyenlo, .egyenlo

; erre fejtődik ki:    cmp eax, ebx
;                      jne .nem_egyenlo
;                      je .egyenlo
```

A makróra lokális szimbólumokat **%%szimbólumnév** alakban készíthetünk. Ezek a makró kifejtése során egyedi nevekre fordulnak le, vagyis a makró két kifejtésével különböző neveket kapunk ugyanaból a lokális szimbólumból. Ennek akkor van szerepe, ha címkékre vagy számítások végzéséhez szimbólumokra van szükségünk, azonban azt szeretnénk, hogy a makró kifejtése előtt definiált szimbólumokat a kifejtés után is fel lehessen használni: a makró ne változtassa meg őket. Ezért nem egy véletlenszerűen választott szimbólumnevet használunk a makrón belül, amelyről nem tudjuk, létezik-e már, és kockáztatjuk, hogy felülírjuk, hanem egy olyan lokális nevet vezetünk be, amelyről az assembler garantálja, hogy a makrón belül jelenik meg először.

Százas prefixsel csak konkrét makróparaméter-pozíciókra lehet hivatkozni. Nem konstansként megjelenő (pl. paraméterként kapott) indexű makróparamétert a **makróparaméterek forgatásával** lehet elérni. Tipikusan ilyen eset áll elő, amikor egy nemkorlátos paraméterszámú makró paramétereit szeretnénk elérni. Ennek eszköze a **%rotate** kulcsszó: utána egy *fordítási időben kiértékelhető* kifejezést írva megváltoztathatjuk a makróparaméterek sorrendjét. Például `%rotate 3` hatására `%1` az eddigi `%4` értékét veszi fel, `%2` `%5`-ét stb., illetve a régi `%1`, `%2` és `%3` innentől `%9`, `%10` és `%11`-ként érhetőek el, ha összesen tizenegy paraméterünk volt. A `%rotate` után negatív szám is állhat, ekkor a másik irányba forognak a makróparaméterek.

Példa	<pre>mov eax, %1 + %2   db %3, %2, %4 push %5</pre>	<pre>xor eax, %xyz sub ecx, %%%a add edx, %{%%b}</pre>	inc %%0 mov eax, %{%%0}
-------	---	--	-------------------------

Ha a forráskód egy részét szeretnénk konstans sokszor megismételni, a **%rep** és **%endrep** kulcsszavak között írhatjuk le. A `%rep` után *fordítási időben kiértékelhető* kifejezésként kell megadni, hányszor akarjuk kifejteni az ismétlendő kódrészletet. Ez a konstrukció makrókon kívül is használható, azonban a leggyakrabban nemkonstans paraméterszámú makrók paramétereinek bejárására alkalmazzák úgy, hogy a mag utolsó sora egy `%rotate`-tel forgatja a makróparamétereket.

Példa	Írunk olyan makrót, amely a kapott paramétereinek összegét számolja ki eax regiszterbe!
Példa	<pre>%macro osszeg 0-*     %define %%masikmakro %1     %define %%kezdet %2     %define %%darab %3     %rotate %%kezdet + 2     %rep %%darab         %%masikmakro %1         %rotate 1     %endrep %endm      forgat mar_definált_makro, 2, 3, "a", "b", "c", "d", "e", "f", "g"  ; ennek hatására a következő makróhívások keletkeznek a kódban: ;     mar_definált_makro "b" ;     mar_definált_makro "c" ;     mar_definált_makro "d" ; azok pedig továbbfejtődnek annak a makrónak a kódja szerint</pre>

## Feltételes fordítás makrókkal

Gyakran előfordul, hogy egy kódrészletet csak akkor akarunk belefordítani a kódba, ha valamilyen *fordítási időben kiértékelhető* kifejezés teljesül. Természetesen ezt kézzel is meg tudnánk tenni, azonban nagyon kényelmes lehet, hiszen így a forráskód egyetlen pontját megváltoztatva több helyen is megváltoztathatjuk a program működését. Jellemző alkalmazás a hibakeresés, amikor egy hibakereső szimbólumot átállítva, annak tartalmától függően, a kódba különböző hibakezelési, naplózási stb. részletek is belekerülnek a megfelelő pontokon.

Az általános feltételes fordítást a **%if feltétel** konstrukcióval végezhetjük. Ennek hatására az ettől az **%endif** kulcsszóig terjedő kódrészlet csak akkor jelenik meg a kódban, ha a feltétel fordítási időben igazra értékelődik ki. A feltételes fordításhoz további ágakat írhatunk **%elif feltétel** alakban, illetve lezáró ágat **%else** alakban.

Példa

```
%macro eliranyit 3-5
    %if %0 = 3
        %1 %2, %3
    %elif %0 = 4
        %1 %2, %3, %4
    %else
        %1 %2, %3, %4, %5
    %endif
%endm

eliranyit hivott_makro, 1, 2
eliranyit hivott_makro, 1, 2, 3
eliranyit hivott_makro, 1, 2, 3, 4
```

; ennek hatására az első paraméterben megadott makrónak minden a megfelelő paraméterszámú változata ; hívódik meg

Hasonló konstrukciókkal ellenőrizhetjük makrók létféről. Egysoros makrók létféről vizsgálatára alkalmas az **%ifdef makrónév**, amelyet előszeretettel alkalmaznak a fent említett hibakeresésre; ennek a további ágait **%elifdef makrónév** alakban adhatjuk meg. Többsorosakat az **%ifmacro makrónév paraméterszám** segítségével vizsgálhatunk, ahol a paraméterszám lehet egy konkrét szám, illetve egy intervallum.

A fentiekben hasznos lehet a **%error üzenet** direktíva, amely fordítási időben kiírja az utána megadott üzenetet.

Példa

```
%macro ellenor 2
    %ifdef %1
        %1 %2
    %else
        %error Nem talalhato a makro: %1
    %endif
%endm

eliranyit nem_letezo_makro, 12456
```

; ennek hatására a fordítóprogram ezt írja ki:

```
a.asm: warning (ellenor:4) Nem talalhato a makro: nem_letezo_makro
```

Még egy hasznos direktíva: fájlt beilleszteni a forráskód adott pontjára így lehet: **%include "fájlnév"**.

Írunk olyan makrót, amely tetszőlegesen sok paramétert kaphat. A paraméterek alprogramok címei, amelyek egy duplaszavas paramétert várnak. A makró generáljon olyan főprogramot, amely sorban meghívja az alprogramokat. A főprogram paraméterei 2 karakteren ábrázolt decimális számok; ha van elég parancssori paraméter, akkor az n-ediknek meghívott alprogram kapja paraméterként az n-edik parancssori paraméter értékét, ha nincs, akkor nullát.

```
%macro alprogramhivo 0-*
main
    %assign %%i 0

    %rep %0
        mov edx, [ebp+12]
        mov edx, [edx+4+4*%%i]
        mov al, [edx]
        mov cl, 10
        mul cl
        add al, [edx+1]
        cmp dword [esp+4], %%i + 1
        setbe dl
        dec dl
        and al, dl
        movzx eax, al
        push eax
        call %1
        add esp, 4

        %rotate 1
        %assign %%i %%i + 1
    %endrep
%endm
```

## Feladatok

1. Írd meg az adc utasítást általánosan, kétparaméteres makróként. Az utasítás leírása egy korábbi feladatban szerepel.

2. Mit kapunk, ha kiíratjuk az adat címke tartalmát?

```
%macro rajzol 4
    db    %1
%rep %2
    db    ' '
%endrep
    db    %3
%rep hossz - ( 2 * %2 )
    db    %4
%endrep
    db    %3
%rep %2
    db    ' '
%endrep
    db    %1
    db    0ah
%endmacro

#define hossz    7
```

**adat.a**

```
rajzol ' ', 1, 'v', 'v'
rajzol ' ', 0, '=' , '='
rajzol '| ', 2, 'X', 'X'
rajzol '| ', 0, ' ', ' '
rajzol '| ', 2, '~', '-'
rajzol '| ', 0, ' ', ' '
rajzol '| ', 0, 'V', '-'
```

```
db    0
```

```
%macro rajzol 5
    db    %1
%rep %2 - 1
    db    ' '
%endrep
    db    %3
%rep %4 - %2 - 1
    db    ' '
%endrep
    db    %5
%rep hossz - %4 - 1
    db    ' '
%endrep
    db    %1
    db    0ah
%endmacro
```

```
%assign hossz 8
```

**adat.b**

```
%assign i 0
%rep 3
    rajzol ' ', 4 - i, '/', 5 + i, '\'
    %assign i i + 1
%endrep

    db    '+'
    db    '-----'
    db    '+', 0ah

    rajzol '| ', 5, 'o', 6, 'o'
    rajzol '| ', 5, 'o', 6, 'o'
    rajzol '| ', 1, ' ', 2, ' '
    rajzol '| ', 4, '| ', 5, '| '

    db    '====='
    db    0
```

3. a. Készíts deriváló makrót. A makró tetszőleges számú paramétert kaphat, amelyek egy polinom együtthatóit adják. A makró tegye a verembe sorban a polinom deriváltjának együtthatóit szavakként! A deriválás szabálya:

$$(c \cdot x^n)' = c \cdot n \cdot x^{n-1}$$

b. Készíts integráló makrót a deriváló makróhoz hasonlóan. A konstans tagot nem kell figyelembe venni. Feltehető, hogy minden kapott együttható egész szám.

Példa

A derivál 1, 5, -4, 6 makróhívás esetén a polinom  $1 \cdot x^3 + 5 \cdot x^2 - 4 \cdot x^1 + 6 \cdot x^0$ , a derivált  $3 \cdot x^2 + 10 \cdot x^1 - 4 \cdot x^0$ , a vermelendő duplaszavak tehát 3, 10, -4. Az integral 6, -2, 3 hívás esetén az integrál  $2 \cdot x^3 - 1 \cdot x^2 + 3 \cdot x^1 + c$ , a vermelendő duplaszavak: 2, -1, 3.

#### 4. Készíts torpedó programot.

- a. Az adatok generálása makróval: egy olyan byte-tömböt kell létrehozni, amely a paraméterként kapott pozíciókon 1-et (van hajó), különben 0-t (nincs hajó) tartalmaz. A makró tetszőleges számú paraméterrel hívható, a pozíciók sorrendben egymás után jönnek.

A hajo 2, 4, 7, 8, 9, 11 makróhívás szerint a következő tömböt kell kapnunk:

db 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1

### b. A program menete:

- mi lövünk
    - beolvassuk a bemenetről, melyik pozícióra lövünk
    - meghívjuk az int mi\_lovunk( int pozíció ) szignatúrájú C függvényt vele
      - ha az egyet ad vissza, találatot értünk el
      - ha kettőt, egyúttal az ellenfél minden hajója elsüllyedt. Ekkor hívjuk meg a void nyert( void ) szignatúrájú C függvényt, majd lépjünk ki.
      - különben nincs találat
  - ők lönek
    - az int ok\_lonek( void ) szignatúrájú C függvény adja meg, melyik pozícióra löttek
      - ha volt ott hajónk, elsüllyed (nullázódik a pozíció)
      - ha minden hajónk elsüllyedt, hívjuk meg a void vesztett( int általunk\_kilött\_hajók\_száma ) szignatúrájú C függvényt, és lépjünk ki.

5. Írj olyan makrót, amely 1 + legalább még 2 paramétert vár, és olyan kódot generál, amely eldönti, hogy a paraméterek (amelyek regiszterek és memóriatartalmak lehetnek) szigorúan növekvő sorrendben vannak-e. Amennyiben nem, akkor ugorunk az első paraméterben megadott címkére.

Példa Ha így hívjuk meg: vizsgal címke, eax, {dword [x]}, esi akkor azt kell megvizsgálnia a kódnak, hogy eax < dword [x] < esi teljesül-e.

6. Adott egy `exp` nevű, háromparaméteres makró: `exp x, 3, 4` beállítja az `x` szimbólumot 34 értékére. Készítsd el az ellenor `n, m` makrót, ami ellenőrzi, hogy minden `k`-ra egytől `m`-ig helyesen számítja-e ki az `exp` makró `nk`-t. Azt kell megvizsgálni, hogy a makrót két szomszédos `k`-ra meghívva, a nagyobb érték megegyezik-e a kisebb érték `n`-szerével. Ha jól működik a makró, állítsuk be az `ok` szimbólumot egyre, ha pedig nem, akkor szüntessük meg az `ok` szimbólumot.

7. Adott egy Fibonacci nevű, kétparaméteres makró: Fibonacci x, 4 beállítja az x szimbólumot a negyedik Fibonacci-szám értékére. Készíts olyan egyparaméteres makrót, ami ellenőrzi, hogy a kapott paraméteréig jól működik-e a Fibonacci makró. Azt kell megvizsgálni, hogy a kezdőértékek jók-e, és hogy három szomszédos számra meghívva a makrót, a két nagyobbik kapott érték különbsége megegyezik-e a legkisebbel. Ha jól működik a makró, állítsuk be az ok szimbólumot egyre, ha pedig nem, akkor szüntessük meg az ok szimbólumot.

8. Írj olyan makrót, amely tetszőleges számú paramétert vár, és olyan kódot generál, ami megfordítja azokat a biteket eax-ben, amelyek sorszámát paraméterként kapta. A generált kód több utasításból is állhat, de minden egyéb regiszter értékét meg kell tartania.

Példa Ha eax tartalma kezdetben 01110101011110101011110110101010,  
akkor fordít 1, 16, 3, 31 után **111101010111101101111011010000** lesz eaz új tartalma.

## A program fordításának menete

Amikor elkészült a program forráskódja, át kell alakítanunk futtatható állománnyá. Ennek első lépcsőjében hívjuk meg az assemblert, amely még csak egy közbülső formátumot, **tárgykódot** állít elő. A tárgykód már gépi kódot tartalmaz, de még nincsen készen: más tárgykódokban levő memóriacímekre való utalások lehetnek benne, és ezek csak a következő fázisban, az **összeszerkesztés** során válnak ismertté. A szerkesztőprogram (linker) feladata az, hogy minden információk segítségével összeállítsa a végső, futtatható fájlt.

A mi esetünkben az assembler neve **nasm**. A következő paramétereit használjuk a könyvben.

- meg kell adni az elkészítendő tárgykód-formátumot a **-f formátum** kapcsolóval: számunkra ez mindig **elf** lesz
- meg kell adni a **forrásfájlt**, amelynek a kiterjesztését általában **.asm**-nak választjuk
- ha szükségünk van rá, készíthetünk listafájlt a **-l listafile-neve** kapcsolóval
- megadhatjuk a kimeneti fájl nevét a **-o fájlnév** kapcsolóval, alapértelmezés szerint ez a fájl neve **.asm** helyett **.o** kiterjesztéssel

Szerkesztőprogramnak a **gcc**-t használjuk<sup>16</sup>. Paraméterei közül csak a tárgykód-fájl nevét kell megadnunk. Ez létrehozza a futtatható fájlt, amely az **a.out** nevet kapja, ha felül nem bíráljuk a **-o fájlnév** kapcsolóval. Ha minden jól csináltunk, már csak futtatni kell az elkészült fájlt.

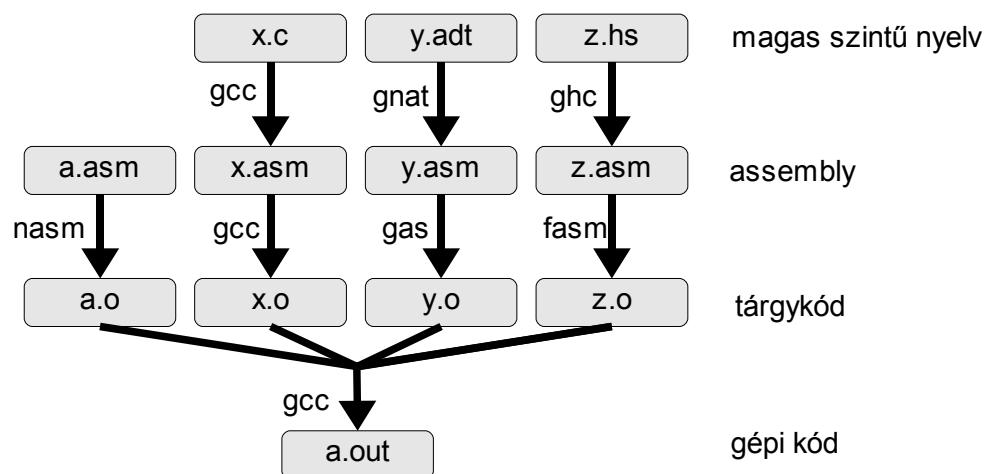
Összefoglalva: a következő parancsokra lesz szükségünk egy assembly fájl fordításához.

```
nasm -felf fájlnév.asm  
gcc fájlnév.o -o fájlnév  
./fájlnév
```

Amennyiben a program több tárgykódból áll, akkor a második lépés a következő alakú.

```
gcc fájl1.o fájl2.o fájl3.o -o futtatható_állomány_neve
```

Ha egy másik (akár más programnyelven megírt) fájlban akarjuk felhasználni valamelyik címkénket, akkor azt a **global címkénév** direktívával kell megjelölni.<sup>17</sup> Mivel a címkénév nem tartalmaz információt a címkén tárolt adat vagy kódrészlet típusáról, ezeket a programnyelvben az importálás során leírt deklarációval kell elérhetővé tenni. Fordítva, ha egy másik tárgykódból használunk fel egy címkét, akkor azt az **extern címkénév** direktívával kell jelezni.



16 A gcc többek között C nyelvről assemblyre fordító fordítóprogramként is működik. Fordítsunk le egy tetszőleges, egyszerű C programot a -S kapcsolóval! Ez egy más szintaxisú assemblyre fordít, mint a könyvben tárgyalta nasm.

17 Mivel a főprogramnak minden láthatónak kell lennie kívülről, a főprogram kódjában szerepelnie kell a **global main** sornak.

## A gépi kód szerkezetéről

Egy utasítás gépi kódja (ami egy bitsorozat) több mezőre bontható. Általában vagy egy byte osztódik több mezőre, vagy egy mező tartalmaz több byte-ot. Az utasítás következő bitjeinek értelmezése (milyen mezőkből áll) függhet az előző mezők értékétől.

Az utasítások gépi kódjának legfontosabb mezője az operációs kód (röviden opkód). Ez dönti el, hogy milyen utasításról van szó. Közeli kapcsolatban áll a mnemonikkal, de mégis különböznek: a mov jelenthet 8, 16 vagy 32 bites adatmozgatást, az opkód azonban ezt az információt is tartalmazza. Az opkód maga is tartalmazhat mezőket, például a duplaszavas adatot megnövelő inc egy byte-os gépi kódjának alsó három bitje azt kódolja, hogy melyik regisztert növeljük meg. A byte operandusú inc gépi kódja ezzel szemben két byte hosszú.

Az utasítások dekódolását tovább nehezíti, ha egyes mezők vegyesen értelmezettek. Egyes utasításoknál ugyanis előfordul, hogy egy mező lehetséges értékei közül egy adott bitminta speciális jelentést kap, a mező egyéb értékei pedig valamilyen más logika szerint szerveződnek. Például: az általános célú regisztereket el lehet kódolni három bit segítségével. Ilyen mezők szerepelnek majdnem minden utasítás gépi kódjában, mert a legtöbb utasításnak van regiszter operandusa, vagy tartalmazhat regiszteren keresztüli memóriahozzáférést. Azonban az [ebp] memóriacímzés nem kódolható el közvetlenül: legfeljebb egy regisztert használó memóriacímzés esetén az az érték, amely más esetekben ebp regisztert jelképezi, fenn van tartva a direkt (csak konstanssal való) memóriacímzés számára. Ezt a címzést más, ekvivalens módon kell elkódolni, [ebp+0] alakban.

0100 0000	inc eax
0100 0001	inc ecx
0100 0010	inc edx
0100 0011	inc ebx
0100 0100	inc esp
0100 0101	inc ebp
0100 0110	inc esi
0100 0111	inc edi

Az utasítás most csak az opkóból áll. Az alsó három bit jelöli ki, hogy melyik regiszterről van szó.

1111 1111 0000 0000	inc dword [eax]
1111 1111 0000 0110	inc dword [esi]

A második byte alsó három bitje a fentihez hasonló szerepet tölt be.

1111 1111 0000 0101	
0111 1000 0101 0110	0011 0100 0001 0010
	inc dword [0x12345678]

A kivételes eset az, ami az előbb ebp regisztert kódolta. Memóriahozzáférés esetén az 101 bitsorozat a direkt címzés számára van fenntartva. Ekkor az utasítás kódjához tartozik még egy négy byte hosszú mező is, ahol maga a konstans jelenik meg.

1111 1111 0100 0101	0000 0000	inc dword [ebp]
1111 1111 0100 0110	1010 1011	inc dword [esi+0xAB]

Az assembler mégis le tudja fordítani az utasítást, ha csak ebp regiszterrel címzünk. Úgy tekinti, mintha egy nulla konstanssal eltoltuk volna a címet. A második byte 6. bitje jelzi azt, hogy a címzésben konstans is szerepel. Maga a konstans egy egy byte-os mezőn kap helyet. Ebben a konstrukcióban ebp ismét a többi regiszterhez hasonlóan van kódolva.

## CISC-elvű architektúrák

A korai számítógépek számítási kapacitása alacsony volt, lassú a memória-hozzáférésük és kevés a memóriájuk, ezért tömör és hatékony kódot kellett rájuk készíteni. Ennek támogatására az utasításkészleteket úgy terveztek, hogy az egyes utasítások összetett számításokat legyenek képesek végrehajtani, amelyek gyakran megvalósíthatóak voltak más utasításokból alkotott rövid kódrészletekkel is. Innen ered a megközelítés neve: Complex Instruction Set Computer, röviden **CISC**. Egy példa erre az IBM 360-as nagygép, amelynek egyik utasításával egy bináris fa adatszerkezet lehetett kiegyensúlyozni.

A kódot az is tömörebbé tette, hogy bonyolult címzések segítségével egy utasítással lehetett adatelemeket lehetett egy lépésben elérni.

Példa

Az x86 architektúrán, ha nem használhatnánk az összetett címzéseket, a következő kifejezést sokkal hosszabb kóddal tudnánk csak megvalósítani.

```
mov eax, [ecx+4*esi+28]          mov eax, ecx
                                  mov ebx, esi
                                  shl ebx, 2
                                  add eax, ebx
                                  add eax, 28
                                  mov eax, [eax]
```

Természetesen a CISC architektúrájú gépek által végrehajtott gépi kód összetettsége is tükrözi a fentieket. Az egyszerű utasítások gépi kódú alakjában sokkal kevesebb információnak kell megjelennie, mint az összetettekében. Az architektúrák tervezőinek logikus döntése ezért a gépi kódjuk is rövidebb. Így viszont a gépi kód végrehajtás előtti dekódolása bonyolultabbá válik, hiszen még az az információ sem áll rendelkezésünkre előzetesen, hogy hány byte alkotja a következő utasítást. Ebből következően a processzorok belső szerkeze is bonyolultabb, mivel ott találhatóak az utasítások dekódolását végző áramkörök is. Az x86 architektúrán 1 és 15 byte közötti hosszúságú utasítások fordulhatnak elő.

## RISC architektúrák

Az 1980-as évekre nyilvánvalóvá vált, hogy a programok túlnyomó többsége fordítóprogram segítségével készül. Kiderült, hogy a fordítóprogramok kódgenerálás során általában nem használják ki a CISC architektúrák teljes kínálatát. A hardver sebessége és tárolókapacitása is rohamosan növekedett.

A **RISC** (Reduced Instruction Set Computer) elvű architektúrákat az új hardveres kritériumoknak megfelelően terveztek. Az utasítások többsége gyorsan – tipikusan egy órajel alatt – hajtható végre. Az utasítások egyszerű szerkezetűek, jellemzően minden utasítás gépi kódja azonos hosszúságú. Az utasítások szerkeze egyszerű, kevés mezőt tartalmaznak, az utánuk következő mezők szerkezetét lehetőleg nem módosítják, ebből következően könnyen dekódolhatóak. Az egyszerű szerkezet csak egyszerű címzéseket tesz lehetővé.

Az x86 architektúra modern gépeinek processzora RISC elveken alapuló magra épül, amely értelmezi és végrehajtja azt a CISC kódot, amely a külvilág számára látható módon hajtódi végre a processzoron. Ezt a kétszintű felépítést mikroarchitektúrának nevezzük.

## Listafájl

Az assembler által generált kódot a **listafájlból** tudjuk megtekinteni. Ezt a nasm **-l listafájl** kapcsolójával tudjuk létrehozni. A listafájl a tárgykód speciális nézete, amely a következőket tartalmazza.

- minden sor elején a sorszámát,
- a kódsor eltolását az utoljára megkezdett szegmens kezdetétől,
- a kódsorból generált byte-okat,
- a forrásprogramszöveget eredeti alakjában, soronként,
- a makrókifejtések szintjét.

A listafájlból megjelenő számok hexadecimális számrendszerben vannak kódolva (a sorszámot kivéve). A listafájlból nem tudjuk közvetlenül kiolvasni az utasítások mezőinek szerkezetét, mivel csak a generált byte-ok hexadecimális alakja jelenik meg benne.

Példa	sorszám	cím a szegmens kezdetétől	lefordított kód (byte-ok)	az eredeti assembly kód változtatás nélkül
	1			
	2			section .text
	3			global main
	4			
	5			main
	6	00000000	B804000000	mov eax, 4
	7	00000005	BB01000000	mov ebx, 1
	8	0000000A	B9[00000000]	mov ecx, szoveg
	9	0000000F	BA04000000	mov edx, 4
	10	00000014	CD80	int 0x80
	11			
	12	00000016	B801000000	mov eax, 1
	13	0000001B	BB00000000	mov ebx, 0
	14	00000020	CD80	int 0x80
	15			
	16			section .data
	17			
	18	00000000	48656C6C6F20	szoveg db "Hello "
	19	00000006	56696C61670A	db "Világ", 0xA

## A program betöltése

A listafájlból és a tárgykódban nem a kód végleges alakja szerepel. Amikor a kódban egy memóriahivatkozás található, a fordítóprogram több okból sem tudja a kódot végső alakban generálni.

Egyrészt előfordulhat, hogy a memóriahivatkozás egy másik tárgykódban szereplő címkére vonatkozik, amelyet az extern kulcsszóval jelöltünk meg. A címke pontos címét addig nem tudjuk eldönthetni, amíg a másik tárgykódot nem generáltuk. Ezért olyan forrásfájlból, amely külső hivatkozásokat tartalmaz, nem lehet listafájlt készíteni.

Másrészt még akkor is, ha minden címkénk ismert, a generált kód nem a végleges címeket tartalmazza. A program betöltésekor a kód- és adatszegmenseket a futtató környezet beolvassa és elhelyezi a memóriában. A szegmensek általában nem a 0 kezdőcímre kerülnek, ezért az összes rájuk vonatkozó memóriahivatkozás hozzá kell adni a szegmens kezdőcímét. Ezért a kódszegmensben található mindegyik, az adatszegmensben levő adatokat elérő memóriahivatkozásból generált kódot módosítani kell. Ezt **relokációknak** nevezzük.

## Hibakeresés a programban

A programok elkészítése során általában elsőre nem sikerül tökéletes munkát végezni. Előfordulhatnak egyszerűbb hibák, például lemarad egy záró zárójel; ezeket a fordítóprogram képes jelezni még azelőtt, hogy futtatható állományt állítana elő a kódóból. Nagyobb fejtörést okoz azonban az, ha a program lefordul, elindul, viszont a végrehajtás során mást csinál, mint amit a programozó elvár tőle. Ekkor meg kell keresni a hiba forrását, majd javítani kell.

A hiba megkeresésére egy kézenfekvő módszer, ha a program egyes pontjain jól látható módon szövegeket íratunk ki, amelyek jelzik számunkra, hogy a futás elérte az adott kódrészt. Ez a megközelítés azonban nehézes, és kevés információt ad a hiba felmerülése időpontjában a regiszterek és a memória tartalmáról. Erre a célra külön **hibakereső** szoftvereket, angolul **debugger** programokat fejlesztettek ki. A továbbiakban a **gdb** hibakereső néhány funkcióját tekintjük át. Ha grafikus hibakeresőre van szükségünk, a **ddd** programot használhatjuk.

Tegyük fel, hogy a következő programban szeretnénk hibát keresni. A program eredetileg az első nyolc természetes számot írná ki egy-egy sorban, de elrontottuk.

Fordítsuk le a programot a szokásos módon, majd indítsuk el a hibakeresőt a

`gdb`

parancsal. Ekkor egy megkapjuk a hibakereső parancssorát. minden parancs az egyértelműségig rövidíthető, ezt aláhúzással jelöljük a továbbiakban. Segítséget a `help` parancssal tudunk kérni.

```
section .text
global main

main
    mov ecx, 0

.ciklus
    cmp ecx, 8
    je .vege

    push ecx
    call kiir
    add esp, 4

    dec ecx
    jmp .ciklus

.vege
    mov eax, 1
    xor ebx, ebx
    int 0x80

    kiir
        push ebp
        mov ebp, esp

        push ecx

        mov eax, 4
        mov ebx, 1
        mov ecx, adat
        mov edx, 2
        int 0x80

        pop ecx

        mov esp, ebp
        pop ebp
        ret

section .data
adat db "0"
db 0xA
```

Először is töltük be a programot.	<a href="#">file kiiro</a>
Ahhoz, hogy a nasm-hoz hasonló formátumban írja ki a hibakereső az adatokat, adjuk ki a következő parancsot.	<a href="#">set disassembly-flavor intel</a>
Először is nézzük meg a betöltött programot. Ehhez a következő két parancsot adjuk ki.	<a href="#">disassemble main</a> <a href="#">disassemble kiir</a>
Az első parancs a main címkétől a kiir címkéig írja ki az utasításokat, a második pedig a megadott címkét (jelen esetben a kiir-t) körülvevő függvényt írja ki, ami most pont a kiir függvény lesz. Pontosan ugyanezt a kimenetet kapnánk például a <code>disas kiir+3</code> parancs eredményéül. Ha a main.ciklus-t szeretnénk kiíratni, egyszeres idézőjelek közé kell tennünk. Kényelmesen használható a tab gombbal az automatikus kiegészítés.	<a href="#">disassemble kiir+3</a> <a href="#">disassemble 'main.ciklus'</a>
Definiálunk egy megszakítási pontot a program kezdetén. Lokális címkére így lehet megszakítási pontot definiálni: <code>break *'main.vege'</code>	<a href="#">break main</a>
Ezután indítsuk el a programot, amely rögtön fenn is akad a megszakítási ponton. A programnak átadandó paramétereket a parancs után kell leírni.	<a href="#">run</a>
Hajtsuk végre lépésenként a programot. Ehhez a <u>ni</u> (következő utasítás, next instruction) parancsot alkalmazzuk. Csak egyszer szükséges begépelni, mivel üres parancssorban egy enter leütésével meg lehet ismételni az előző kiadott parancsot. Az <u>ni</u> -nek megadható, hány utasítást futtasson le egyszerre.	<a href="#">ni</a> <a href="#">ni 50</a>
A sok kiadott <u>ni</u> hatására látszik, hogy a programunk ciklizálni kezd: ugyanazok a címek tűnnek fel sorban. Tegyük fel, hogy az a gyanunk támadt, hogy a kiir függvényben van a gond; ekkor a <u>stepi</u> segítségével úgy hajtjuk végre a programot lépésenként, hogy a meghívott alprogramokba belépünk.	<a href="#">stepi</a>
Sajnos, ez sem segített, de újabb ötletünk támadt: mivel a kiir-t megnézve az jónak tűnik, megvizsgáljuk, megfelelő paraméterrel hívtuk-e meg. A hibakeresőben felvesszük az ecx regisztert figyelt értéknek. A hibakereső sajátossága, hogy a regiszterek neveit a dollár prefixsel kell ellátni.	<a href="#">display \$ecx</a>
A hibakövetést tovább könnyítendő, beállítjuk, hogy a hibakereső minden lépés után jelezze ki a következő utasítást. A /i formátumkód azt jelzi, hogy utasítást íratunk ki, az eip pedig az a számunkra el nem érhető regiszter, amely az aktuális utasítás címét tartalmazza.	<a href="#">display/i \$eip</a>
Innen, folytatva a léptetést, megállapítjuk, hogy a paraméter megfelelő, azonban a kiírás mégsem jó. Kilépünk a hibakeresőből, és ecx elvermelése után betoldjuk a következő sort.	<a href="#">quit</a>
mov [adat], cl	
A hibakeresés kezdete után látszik, hogy most már valóban különböző karakterek jelennek meg, de ezek nem a 0, 1, ... számok. Kapcsoljuk folytonos megjelenítésre az adat címkét, hiszen ennek a tartalmát írjuk ki. A /c kapcsoló azt jelzi, hogy karakteresen jelenítünk meg.	<a href="#">display/c adat</a>
Egy idő után észrevesszük, hogy a karakterek nem növekednek, hanem csökkennek, ezért át kell írnunk a main.ciklus-ban a dec ecx utasítást inc-re. Ezután még az is kiderül, hogy elfelejtettük, hogy a karakterkódolás szerint a számjegyek nem a nulla pozícióról indulnak, hanem 30h-ról, ezért a kiir függvényben ecx vermelése után még meg kell növelnünk cl-t a következő utasítással.	
add cl, 30h	
A program készen van.	

## Gyakran előforduló hibák

Az alábbiakban következik néhány, zárhelyi dolgozatokban gyakran előforduló hiba leírása. Ha adható rövid példa, dőlt betűvel szedve egy-egy konkrét előfordulást mutatunk be, majd pedig egy konkrét javítást. Csak az assemblyhez közvetlenül kapcsolódó hibák szerepelnek, így pl. a kiolvashatatlan kézírás, a feladat félreértése, csak speciális esetekre megoldott feladat stb. nem.

1. Nem elsősorban assembly vonatkozású, de a **kiolvashatatlan kézírás** következtében sokszor nem egyértelmű, hogy melyik regiszterről van szó, mennyi a leírt konstans értéke stb.
2. **Szintaktikai hibák.**
  - a) **Számábrázolás.**
    - i. **Betűvel kezdődő hexadecimális szám elejéről lemarad a nulla.**
    - ii. **Hexadecimális szám végéről lemarad a »h« betű** (és az elején sincsen 0x).
  - b) Utasítások operandusai közül lemarad a **vessző**.
3. **Az adatok mérete nem megfelelő.**
  - a) **A regiszterek méretének eltévesztése.**

*Az al regiszter 4 bitesként kezelése.*

- b) **Az utasítás operandusainak mérete nem egyezik meg** (kivéve movsx és movzx, ahol éppen eltérőnek kell lenniük).

*mov eax, bl → mov eax, ebx vagy mov al, bl*

- c) **Nem megfelelő adathosszat** használunk adatok definiálásához.

*Duplaszavas adatot szeretnénk definiálni, de csak szavas adatterületet foglalunk le.*

*adat resw 1 → adat resd 1*

- d) **A memoriához való hozzáférés** nem a megfelelő adathosszúságban történik. Ritka esetekben szándékosan lehetünk ilyet, de általában ez hiba.

*Olyan címre, amelyen duplaszavas méretű adat van eltárolva, byte hosszan írunk.*

*adat dd 0  
mov [adat], al → mov [adat], eax*

*Természetesen az is helytelen, ha ugyanarról a címről például szó hosszan próbálunk olvasni.*

*mov ax, [adat] → mov eax, [adat]*

4. **Adatábrázolás.**
  - a) Szám ábrázolásánál az (előjel nélküli ábrázolt) szám konstans és a számjegy karakter összekeverése. Pl. szövegesen leírt szám (számjegyeket tartalmazó string) konverziójánál minden karaktert előbb előjel nélkül ábrázolt számmá kell alakítani. Ezt legegyszerűbben a 0 számjegy kódjának levonásával lehet elérni. Ez a módszer kihasználja, hogy a számjegyek karakterkódjai sorrendben találhatóak.
5. **Szegmensek.**
  - a) **A szegmenshatárok jelölése lemarad.**
  - b) **Kódszegmensben adat, adatszegmensben kód szerepeltetése.** Az előbbi ekvivalens azzal, mintha gépi kódot írnánk kézzel.

- c) Az **inicializált** és **inicializálatlan adatszegmens összekeverése** (.data és .bss), a másikba való adatok szerepettetése.

## 6. Utasítások operandusai.

- a) **Adathossz explicit megadása hiányzik**, amikor ez nem derül ki máshonnan, pl. kétoperandusú utasítás esetén a másik operandusból.

`mov [eax], 2` → ***mov dword [eax], 2 vagy mov word [eax], 2 vagy mov byte [eax], 2***

- b) **Nem megfelelő számú vagy fajtájú operandus megadása** utasításhoz.

*Az alábbiakban elkerültük a 3. fajtájú hibát, mivel megadtuk az adat hosszát (ami máshonnan nem derülne ki), azonban a div-nek nincsen konstans operandust fogadó változata. A javításhoz felhasználjuk bl regisztert.*

`div byte 2` → ***mov bl, 2  
div bl***

- c) **Túlságosan összetett számítási műveletek egy utasításon belül.** Ezeket csak több utasítással lehet kiváltani, amihez szükség lehet további regiszterek felhasználására. Ebben az esetben a regiszter eredeti tartalmát a verembe menthetjük, vagy lefoglalhatunk egy tárterületet erre a célra.

*A címke címke tartalma plusz egyet szeretnénk eax-be tölteni.*

`mov eax, [címke] + 1` → ***mov eax, [címke]  
inc eax***

*Az ebx és ecx regiszterek összegét szeretnénk eax-be tölteni.*

`mov eax, ebx + ecx` → ***mov eax, ebx  
add eax, ecx***

*Egy komplex indexelési műveletet szeretnénk végrehajtani. Az utasítás az eredeti formájában nem érvényes, a helyettesítő kódban az egyszerűség kedvéért felhasználjuk eax és edx regisztereit.  
Megoldható a feladat úgy is, hogy átmeneti tárolókat veszünk fel az inicializálatlan adatszegmensbe.*

`cmp [esi-(ecx-4)*4], [esi-(ebx-1)*4]` → ***mov eax, ebx ; eax = ebx  
dec eax ; eax = ebx - 1  
shr eax, 2 ; eax = ( ebx - 1 ) \* 4  
sub eax, esi ; eax = ( ebx - 1 ) \* 4 - esi  
neg eax ; eax = esi - ( ebx - 1 ) \* 4  
mov eax, [eax] ; eax = [esi - ( ebx - 1 ) \* 4]  
mov edx, ecx ; edx = ecx  
sub edx, 4 ; edx = edx - 4  
shr edx, 2 ; edx = ( edx - 4 ) \* 4  
sub edx, esi ; edx = ( edx - 4 ) \* 4 - esi  
neg edx ; edx = esi - ( edx - 4 ) \* 4  
mov edx, [eax] ; edx = [esi - ( edx - 4 ) \* 4]  
cmp edx, eax ; edx = [esi - ( edx - 4 ) \* 4]***

**d) Adat és cím fogalmának összekeverése.**

- i. Adat címe helyett csak az adat szerepel: hiányzik a szögletes zárójel.

*A címke címke tartalmát szeretnénk eax-be tölteni.*

*mov eax, cimke → mov eax, [cimke]*

- ii. Adat helyett adat címe szerepel: feleslegesen megjelenő szögletes zárójel. Előfordul, hogy ez a hiba párosul 3.c)-vel.

- e) A cél és a forrás összekeverése (az operandusok fordított sorrendben vannak leírva). Különösen gyakran fordul elő, hogy konstans vagy címke az első operandus, amelyeknek futási időben nem lehet értéket adni.

**f) Indexelési hibák.**

- i. **Indexregiszter negatív előjellel.** Ezt 3.c)-hez hasonlóan csak hosszabb kódrészlettel lehet kiváltani.
- ii. **Adat hosszának figyelmen kívül hagyása** indexeléskor.

Duplaszó hosszú adatok tömbjének indexeléskor csak byte-onkénti indexelés.

*mov [bazis+ecx], eax → mov [bazis+4\*ecx], eax*

- iii. **A little endian tárolási mód** figyelmen kívül hagyása.

**7. Programkonstrukciók.**

- a) **Elágazás ágaiban lemarad a kiugrás az elágazás végére.** Ez többirányú elágazásra is vonatkozik.

*Az eax regisztert akarjuk nullára vagy egyre állítani annak függvényében, hogy az adat címén található duplaszavas adat értéke egy-e. A kiugrás hiánya miatt az igaz ág végrehajtása eax nullára állítása után ráfut a másik ág kódjára is, ami elállítja eax-ot egyre.*

<i>cmp dword [adat], 1 jne .else_ag</i>	<i>cmp dword [adat], 1 jne .else_ag</i>
<i>mov eax, 0</i>	<i>mov eax, 0</i>
→	<i>jmp .elagazas_vege</i>
<i>.else_ag</i>	<i>.else_ag</i>
<i>mov eax, 1</i>	<i>mov eax, 1</i>
<i>.elagazas_vege</i>	

**b) Ciklusok.**

- i. **A ciklus inicializáló lépése bekerül a ciklusmagba.**

- ii. **A ciklus inicializáló lépése lemarad.**

- iii. **C stílusú for ciklusok.**

- 1) **A feltételvizsgálatnak** a ciklus elején (közvetlenül a ciklus címkéje után) kell megtörténnie.
- 2) **A ciklusváltozó léptetésének** a ciklus végén (közvetlenül a ciklusmag elejére való visszaugrás előtt) kell megtörténnie.

- c) **Elérhetetlen programkód.** Mivel sosem fut le, felesleges létrehozni.

- d) **Alprogramok.** Az alprogramra lokális, a futási idejű veremben elhelyezkedő változó helyett globális változó létrehozása az adatszegmensben. Ez nem feltétlenül hiba...

**8. Egyes utasításokhoz köthető hibák.**

- a) **A not és a neg utasítás keverése.** A neg számítása során lemarad a +1.

- b) **Szorzás és osztás során a forrás- és célregiszterek eltévesztése.** Helyes alkalmazáshoz lásd a mul, imul, div, idiv leírását és a könyv végén a táblázatot.

## 9. Megszakítások, alprogramok.

- a) A **megszakítás meghívása lemarad**. A paraméterek beállításán kívül szükség van az „int 80h” sorra is.
- b) **Alprogram címkéje lemarad**. Nélküle nem lehet (kényelmesen) meghívni a belépési pontot a kód más részeiről.
- c) **Alprogram címkéje nem lehet lokális**. Az *alprogramra* lokális címkéket érdemes létrehozni.
- d) **Az alprogram bevezető és/vagy kilépő kódrészlete lemarad**. Kellő tapasztalattal a kódrészletek egyszerűsíthetők, de az alprogram végén legalább egy visszatérő „ret” utasításnak szerepelnie kell.
- e) Az alprogramok elején és végén a **használt regiszterek nincsenek elmentve és visszatöltve**.

## 10. Fájlok.

**Lemarad a fájlleíró eltárolása.** Ha megnyitás után nem tároljuk el a fájlleírót a memóriában (lokális vagy globális változóként), akkor a továbbiakban nem tudjuk elérni a fájlt.

## 11. Verem.

- a) **Nem megfelelően rendezett a verem**. Általában: a push-ok és pop-ok nincsenek párbán.
- b) A verembe tett és abból kivett adatok mérete nem egyezik.

*Az eax regiszter alsó szavát szeretnénk áttölteni bx-be, és ehhez a vermet próbáljuk felhasználni. Az eredeti kódrészlet ezt elvégzi, mivel a byte-sorrend fordul, azonban két byte-tal többet hagy a veremben, mint amennyi kezdetben benne volt. Ennek elkerülése érdekében a teljes ebx-be pop-olunk. Ha el akarjuk kerülni, hogy ebx felső felét is felülírjuk, akkor a pop után vissza is léptethetjük esp-t a kiinduló helyzetbe.*

push pop	eax bx	→	push eax <b>pop ebx</b>	vagy	push eax pop bx add esp, 2
-------------	-----------	---	----------------------------	------	----------------------------------

- c) **Nemszenderd hozzáférés a veremhez**. A verem push-on, pop-on és alprogramokon belül ebp-n való kezelésén kívül más módszerrel nagyon oda kell figyelni ahhoz, hogy a verem szerkezete el ne romoljon.

## 12. Makrók.

- a) **A futási és fordítási idő fogalmának összekeverése**. A makrók fordítási időben, az előfordítási fázis során alakítják át a forráskód szövegét, a regiszterek csak futási időben léteznek, az utasítások futási időben hatnak.

mov eax, 1 mov ebx, 2 %assign ecx eax + ebx	→	mov eax, 1 mov ebx, 2 <b>mov ecx, eax</b> <b>add ecx, ebx</b>
---	---	--

- b) **Makróra lokális címkék alkalmazásának hiánya**. Ha globális címkéket alkalmazunk makrókban, akkor nem lehet őket több helyen kifejteni, mert a fordító hibát ad többször definiált név miatt. Makróra lokális címke esetén minden kifejtéskor új, egyedi név keletkezik.

%macro faktorialis 1 %assign fakt 1 %assign i 1	→	%macro faktorialis 1 %assign fakt 1 %assign %%i 1
%rep %%0 - 1 %assign fakt i * fakt %endrep	→	%rep %%0 - 1 %assign fakt %%i * fakt %endrep
%endm	→	%endm

## A feladatok megoldásai

### Értékek ábrázolása

1. Mekkora értékek tárolhatóak egy bitvektorban? Add meg általánosan is!

bitek száma	számrendszer	előjel nélkül		előjelesen	
		legkisebb érték	legnagyobb érték	legkisebb érték	legnagyobb érték
8	hexadecimális	00	FF	80	7F
	decimális	0	255	-128	+127
16	hexadecimális	00 00	FF FF	80 00	7F FF FF FF
	decimális	0	65 535	-32 768	+32 767
32	hexadecimális	00 00 00 00	FF FF FF FF	80 00 00 00	7F FF FF FF
	decimális	0	4 294 967 295	-2 147 483 648	+2 147 483 647
4 · n	hexadecimális	00 00 ... 00 n számjegy	FF ... FF n számjegy	80 00 ... 00 n számjegy	7F FF ... FF n számjegy
	decimális	0	$16^n - 1$	$-2^{4n-1}$	$+2^{4n-1} - 1$

2. Töltsd ki az üres cellákat a táblázatban!

bitek száma	bináris	decimális		hexadecimális
		előjel nélküli	előjeles	
8	1100 1111	207	-49	C9
8	0110 1011	107	+107	6B
8	1110 1000	232	-24	E8
16	1001 0110 0001 1110	38 430	-27 106	96 1D
16	0010 0000 1010 0101	8357	+8357	20 A5
16	1101 0010 0011 0111	53 815	-11 721	D2 37

3. Végezd el az alábbi műveleteket!

$$\begin{array}{r}
 \begin{array}{r}
 1001 \ 1101 \\
 \text{and} \ 1100 \ 0111 \\
 \hline
 \textbf{1000} \ \textbf{0101}
 \end{array}
 \quad
 \begin{array}{r}
 0110 \ 1110 \\
 \text{or} \ 1000 \ 0000 \\
 \hline
 \textbf{1110} \ \textbf{1110}
 \end{array}
 \quad
 \begin{array}{r}
 0101 \ 1010 \\
 \text{xor} \ 1111 \ 0001 \\
 \hline
 \textbf{1010} \ \textbf{1011}
 \end{array}
 \end{array}$$

$$0101 \ 0110 \ 1101 \ 1101_2 + 1C \ E3_{16} = 0111 \ 0011 \ 1100 \ 0000_2 = 29 \ 632_{10} = 73 \ C0_{16}$$

$$16 \ 537_{10} - 0000 \ 0011 \ 1010 \ 1100_2 = 0011 \ 1100 \ 1110 \ 1101_2 = 15 \ 597_{10} = 3C \ ED_{16}$$

$$54 \ 32_{16} \text{ or } 38 \ 529_{10} = 0101 \ 0110 \ 1011 \ 0011_2 = 22 \ 195_{10} = 56 \ B3_{16}$$

4. Írd le a saját nevedet ASCII kódolással, ékezetek nélkül!

K	i	t	l	e	i	szóköz	R	o	b	e	r	t	sorvége
4B	69	74	6C	65	69		20	52	6F	62	65	72	74

5. Írd le a „Hello Világ” szöveget ASCII kódolással!

H	e	l	l	o	szóköz	V	i	l	a	g	sorvége
48	65	6C	6C	6F		20	56	69	6C	61	67

## Utasítások

1. Add meg a regiszterek tartalmát minden utasítás után! A feladatot a jelölt pontokon is el lehet kezdeni.

	reg. vagy mem.	bináris (megfelelő bithosszon)	decimális		hexadecimális
			előjel nélküli	előjeles	
a.	mov ebx, 1023	ebx 0000 0000 0000 0000 0000 0011 1111 1111	1023	+1023	00 00 02 FF
	mov al, -100	al 1001 1100	156	-100	9C
	movsx eax, bl	eax 1111 1111 1111 1111 1111 1111 1111 1111	65 535	-1	FF FF FF FF
b.	mov eax, 1C34F6E2h	ax 1111 0110 1110 0010	63 202	-2334	F6E2
		ah 1111 0110	246	-10	F6
		al 1110 0010	226	-30	E2
c.	mov byte [0FAh], 01Ch	[0xFA] 0001 1100	28	+28	1C
	mov byte [0FBh], 0x22	[0xFB] 0010 0010	34	+34	22
	mov ax, [0FAh]	ax 0010 0010 0001 1100	8732	+8732	221C
d.	mov eax, 89ABCDEFh	ax 1000 1001 1010 1011 1100 1101 1110 1111	2 309 737 967	-1 985 229 329	89 AB CD EF
	mov ah, al	ah 1110 1111	239	-17	EF
	mov al, 42	al 0010 1010	42	+42	2A
	mov bl, al	bl 0010 1010	42	+42	2A
	mov bh, al	bx 0010 1010 0010 1010	10794	+10 794	2A 2A
e.	mov ax, 0FF42h	ax 1111 1111 0100 0010	65 346	-190	FF 42
	mov bx, 0DCBAh	bx 1101 1100 1011 1010	56 506	-9 030	DC BA
	sub bh, al	bh 1001 1010	154	-102	9A
	xor ax, bx	ax 0110 0101 1111 1000	26 104	26 104	65 F8
		bx 1001 1010 1011 1010	39 610	-25 926	9A BA
f.	mov al, 01011010b	al 0101 1010	90	+90	5A
	or ah, 10101010b	ah 1111 1010	250	-6	FA
g.	mov al, 120	al 0111 1000	120	+120	78
	not al	al 1000 0111	135	-121	87
	neg al	al 0111 1001	121	121	79
h.	mov bh, 254	bh 1111 1110	254	-2	FE
	inc bh	bh 1111 1111	255	-1	FF
	add bh, 1	bh 0000 0000	0	0	00
i.	mov cx, 0x01F0b	496	+496	01F0	
	mov eax, -128	(eax) 1111 1111 1111 1111 1111 1111 1000 0000	4 294 967 168	-128	FF FF FF F0
	mul cx	(dx : ax) 0000 0000 0000 0000:1111 1000 0000 0000	0 : 63 488	0 : -2048	00 00:F8 00
	imul cl	ax 0	0	0	0

2. a. Az x, y és z címen byte-os adatok vannak.  
Számítsd ki z-be  
( not x | y ) & x értékét!

```
mov al, [x]
not al
or al, [y]
and al, [x]
```

2. b. Számítsd ki eax-be  
( eax - 1 ) \* 2 értékét!

```
dec eax
shr eax, 1
```

3. Adj meg minél több utasítást, amely  
 a. nullára állítja eax értékét!

<b>mov eax, 0</b>	<b>xor eax, eax</b>	<b>shr eax, 32</b>
<b>xor eax, eax</b>	<b>sub eax, eax</b>	<b>shl eax, 32</b>
<b>and eax, 0</b>		<b>sal eax, 32</b>

- b. nem változtatja meg a regiszterek értékét (legfeljebb a jelzőbitek regiszterének kivételével)!

<b>nop</b>	<b>bt eax, 0</b>	<b>ror eax, 0</b>
<b>mov eax, eax</b>	<b>and eax, 0xFFFFFFFF</b>	<b>rol eax, 0</b>
<b>xchg al, al</b>	<b>and eax, eax</b>	<b>cmp eax, ebx</b>
<b>add eax, 0</b>	<b>or eax, 0</b>	
<b>sub eax, 0</b>	<b>or eax, eax</b>	<b>jmp .kovetkezo</b>
	<b>xor eax, 0</b>	<b>.kovetkezo</b>

- c. eax 2. bitjét egyesre állítja, a többöt pedig változatlanul hagyja!

<b>or eax, 100b</b>	<b>bts eax, 2</b>
---------------------	-------------------

- d. az ellentétre állítja ebx legalsó (0.) és legfelső (31.) bitjét!

<b>xor ebx, 0x8001</b>
------------------------

- e. al tartalmát egy előre adott értékről egy másik adott értékre állítja! (Például 229-ről 33-ra.)

<b>mov al, 33</b>	<b>add al, 33-229</b>	<b>sub al, 229-33</b>
-------------------	-----------------------	-----------------------

A feladat megoldható még az **xor al, eltérés** utasítással, ahol az *eltérést* a két értékből az xor műveettel kaphatjuk meg, jelen esetben ez **1100 0100b**. Speciális esetekben további utasításokat is lehet alkalmazni, például bitforgatást, regiszter nullázását stb.

4. Az x és y két címke, melyeken 32 bites adatokat tárolunk.

- a. Valósítsd meg az  $x := y$  értékadást! Közvetlenül egyik címről a másikra másoló utasítás nincsen.

<b>mov eax, [x]</b>		<b>push dword [x]</b>
<b>mov [y], eax</b>	<i>vagy</i>	<b>pop dword [y]</b>

- b. Cserél meg x és y tartalmát!

<b>mov eax, [x]</b>		<b>push dword [x]</b>
<b>mov edx, [y]</b>		<b>push dword [y]</b>
<b>mov [x], edx</b>	<i>vagy</i>	<b>pop dword [x]</b>
<b>mov [y], eax</b>		<b>pop dword [y]</b>

5. Milyen byte-okat tartalmaz sorban az adatszegmens?

```
section .data  
db    55h  
db    55h, 56h  
db    'a', 0Ah  
db    "hello"  
dw    1234h  
dd    89ABCDEFh
```

A byte-ok sorban, hexadecimális alakban:

```
55 55 56 61 0A 68 65 6C 6C 6F 34 12 00 EF CD AB 89.
```

6. Valósítsd meg az ismertetett utasításokkal az *adc eax, ebx* utasítást! Az adc az első operandushoz hozzáadja a másodikat (mint az add), és még ehhez az átviteli bitet (azaz hozzáad még egyet, ha az átviteli bit be van állítva, különben az eredmény a két operandus összege).

```
jnc .nincs_carry  
inc eax  
add eax, ebx  
jmp .vege  
.nincs_carry  
    add eax, ebx  
.vege
```

A feladat rövidebben is megoldható. Ehhez felhasználjuk, hogy az egyik ág kódja posztfixe a másik ágának: a carry ág végén nem ugrunk el, hanem hagyjuk, hogy a vezérlés ráfusson a másik ág kódjára. Hasonló kód keletkezik, ha egy C programban a switch szerkezet egy ágát nem zárjuk le break utasítással.

```
jnc .nincs_carry  
inc eax  
.nincs_carry  
    add eax, ebx
```

7. Mennyi a kódrészletek végrehajtása után eax értéke?

```
.a  
xor ebx, ebx  
inc ebx  
jmp .vege  
  
dec ebx  
mov eax, ebx  
.vege
```

```
.b  
cmp eax, 0  
je near .vege  
  
mov eax, eax  
.vege  
inc eax
```

```
.c  
cmp eax, eax  
je near .vege  
dec eax  
  
jmp .c  
.vege
```

- Ugyanannyi, mint a program elején, mert nem hajtunk végre olyan utasítást, amely megváltoztatná eax értékét. Az utolsó két utasítás elérhetetlen, mert előtte feltétel nélkül elugrunk a .vege címkére; az ugrás előtti két utasítás ebx értékét állítja egyre.**
- A feltételvizsgálat szerint két eset lehetséges: vagy nulla volt kezdetben eax értéke, vagy sem. Amennyiben nulla volt, elugrunk a .vege címkére. Amennyiben nem nulla volt, folytatódik a végrehajtás, és a következő utasítás kinullázza eax-ot. Tehát a .vege címkére érve eax értéke nulla, bármelyik úton is érkezett ide a vezérlés. Innen még egyetlen utasítás van hátra, amely megnöveli eax értékét egyre.**
- A kezdeti feltételvizsgálat akkor ugrik a .vege címkére, ha eax regiszter értéke megegyezik eax regiszterével. Ez minden teljesül, ezért a következő két kódsor elérhetetlen, mert rögtön elugrunk. A regiszter értéke nem változik meg.**

8. Írj olyan kódrészletet, amely ecx-be meghatározza eax és ebx maximumát! A számok előjel nélkül vannak ábrázolva.

```
    cmp      eax, ebx
    ja near .eax_nagyobb ; előjel nélküli összehasonlítás

    mov      ecx, ebx      ; itt biztosan ebx a nagyobb (vagy egyenlők)
    jmp      .vege         ; kiugrás az elágazásból

.eax_nagyobb
    mov      ecx, eax

.vege
```

9. Lineáris keresés: az adat címkétől kiindulva keresd meg az első olyan byte-ot, amely nullát tartalmaz, és ennek a címét add meg eax-ben!

```
    mov      eax, adat      ; az adat címke címét betöljük eax-ra

.ciklus
    cmp byte [eax], 0
    je near .vege          ; ha az eax címen található byte értéke nulla,
    ; kiugrunk a ciklusból
    inc      eax            ; különben a következő byte-ra lépünk
    jmp      .ciklus         ; ... és folytatjuk a keresést

.vege
```

10. Valósítsd meg assembly nyelven az alábbi C++ programrészletet! Az a és b változók típusa int.

```
for ( int i = 0; i < 10; ++i ) a += b;

    mov      ecx, 0      ; i-t ecx-ben tároljuk

.ciklus
    cmp      ecx, 10
    jnl near .vege      ; előjeles vizsgálat, ha i >= 10, ciklus vége

    mov      eax, [a]
    add      [b], eax    ; a += b;

    dec      ecx        ; ++
    jmp      .ciklus

.vege
```

11. Az n paraméter a memóriában található egy duplaszavas változóban. Számítsd ki az eax regiszterbe

a. az első n szám összegét!

b. n! értékét!

Az adatszegmensben tárolt n minden esetben a következőképpen néz ki.

```
section .data  
n dd 9
```

Amennyiben egy másik kód részlet határozza meg n értékét számunkra, az inicializálatlan adatszegmensben is elhelyezhetjük.

```
section .bss  
n resd 1
```

a

```
mov eax, [n]  
mov ebx, 0  
mov ecx, 0  
; ebx: részletösszeg  
; ecx: számláló  
.ciklus  
add ebx, ecx  
inc ecx  
  
cmp ecx, eax  
ja .vege  
jmp .ciklus  
.vege
```

b

```
mov eax, 1  
mov ebx, 1  
.ciklus  
cmp ebx, [n]  
jg .vege  
  
mul ebx  
inc ebx  
jmp .ciklus
```

12. Adott egy duplaszavas értéket tartalmazó láncolt lista címke. Add meg a hosszát!

```
mov ebx, fejelem  
mov eax, 1  
  
.ciklus  
cmp dword [ebx], 0  
je .vege  
  
mov ebx, [ebx]  
inc eax  
jmp .ciklus  
.vege
```

Ez a kód részlet négyel tölti fel eax-et, ha az alábbi adatokra hívjuk meg.

elem2	dd	elem3
	dd	3
fejelem	dd	elem2
	dd	150
elem4	dd	0
	dd	365
elem3	dd	elem4
	dd	8163

13. Adott a szöveg címén egy szöveg, amelynek ismert a hossza. Fordítsd meg helyben!

```
mov esi, szöveg  
mov edi, szöveg + hossz - 1 ; szöveg + hossz már a szöveg utáni pozíció
```

```
.ciklus  
cmp esi, edi  
jnb .vege  
  
mov al, [esi]  
mov ah, [edi]  
mov [esi], ah  
mov [edi], al  
  
inc esi  
dec edi  
  
jmp .ciklus  
.vege
```

14. Adott három időpont: egy-egy óra és perc, amelyek duplaszavasan vannak eltárolva a memóriában. Az első két időpont között részleges napfogyatkozás következik be. A napfogyatkozás mértéke a félidőig lineárisan növekszik, félidőben pontosan 50%, onnantól lineárisan csökken. Add meg, hogy a harmadik időpontban (amelyről feltételezhető, hogy az előző kettő között helyezkedik el) mekkora volt egész százalékra kerekítve a napfogyatkozás mértéke!

**Tegyük fel, hogy az adatok a következőképpen vannak eltárolva.**

```
section .data
```

**kezdet**

.ora	dd	10
.perc	dd	00
<b>veg</b>		
.ora	dd	14
.perc	dd	00
<b>idopont</b>		
.ora	dd	10
.perc	dd	30

```

mov ebx, 60
mov eax, [veg.ora]
sub eax, [kezdet.ora]
mul ebx
add eax, [veg.perc]      ; eax = a teljes napfogyatkozás
sub eax, [kezdet.perc]    ;      időtartama
shr eax, 1

mov ecx, eax              ; ecx = a teljes időtartam fele

mov eax, [idopont.ora]
sub eax, [kezdet.ora]
mul ebx
add eax, [idopont.perc]
sub eax, [kezdet.perc]    ; eax = a kezdettől eltelt idő

mov ebx, ecx

cmp ebx, eax              ; ugrás, ha a félidő előtt vagyunk
ja .tovabb

sub eax, ecx
sub eax, ecx
neg eax                   ; eax = idő a napfogy. végéig

.tovabb
mov ebx, 50
mul ebx
div ecx                  ; eax * 50 / (a teljes idő fele)
                           ; pontosan a kért eredményt adja

```

15. A tenisben az a játékos nyer meg egy ún. rövidített játékot, aki először ér el legalább 7 pontot úgy, hogy legalább 2 ponttal vezet (azaz ha az állás 7-6, akkor még nem dölt el a rövidített játék, például a második játékos megszerezheti a következő három pontot, és akkor ő nyer 7-9-re). Egy tömbben adott, hogy melyik játékos nyerte meg a soron következő játékot. A tömb hossza előre nem ismert. Add meg, ki nyerte a rövidített játékot, és milyen arányban!

Tegyük fel, hogy a következőképpen adott a nyert játékok tömbje.

```
section .data
jatek db 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0
```

A következő kódrészlet kiszámítja al-be és ah-ba, melyik játékos hány játékot nyert. A program futása az .Anyert illetve a .Bnyert címkén folytatódik attól függően, melyik játékos nyerte a rövidített játékot.

<b>xor eax, eax</b>	; ah és al (a játékosok által nyert pontok) kinullázása
<b>xor ebx, ebx</b>	; bl (a játékosok által nyert pontok különbsége) kinullázása
<b>mov esi, jatek</b>	; esi mutatja az aktuális pozíciónkat a tömbben

```
.ciklus
    cmp bl, 2
    jnge .Anemvezet ; annak ellenőrzése, vezet-e a játékos legalább 2 ponttal

    cmp al, 7
    jae .Anyert ; ha igen, akkor nyert, ha elért legalább 7 pontot
```

; itt következhetne egy **jmp nincsnyertes** utasítás, mert ha az egyik játékos vezet, akkor a másik biztosan nem, de az sem okoz problémát, ha a vezérlés ráfut a másik ellenőrzésre

```
.Anemvezet
    cmp bl, -2
    jnle .nincsnyertes

    cmp ah, 7
    jae .Bnyert ; az előzőhez hasonló ellenőrzés a másik játékosra
```

```
.nincsnyertes
    add al, [esi]
    inc ah
    sub ah, [esi] ; ah és al (a játékosok által nyert pontok) állítása
    mov bh, [esi]
    shl bh, 1
    dec bh ; bh = +1 vagy -1 attól függően, melyik játékos nyerte a pontot
    add bl, bh ; bl (a játékosok által nyert pontok különbsége) állítása
    inc esi ; a mutató léptetése
    jmp .ciklus
```

## Parancssori paraméterátadás, rendszerszolgáltatások

1. a. Írd ki a „Hello világ!” szöveget betűről betűre bővülve! Először csak egy „H” betűt írj ki és egy sorvégét, aztán a „He” szöveget és sorvégét stb.

```

section .text
global main

main
    push ebp
    mov ebp, esp
    sub esp, 4      ; [ebp-4] : ciklusszámláló

    mov [ebp-4], dword 0
    ; kezdetben a szöveg nulladik pozícióján állunk

.ciklus
    cmp dword [ebp-4], 11
    je near .vege

    mov eax, [ebp-4]
    mov dl, [szoveg + eax]
    mov [kiir + eax], dl
    mov [kiir + eax + 1], byte 0xA
    ; a következő karakter és egy sorvége

    mov edx, eax      ; a kiírás hossza
    add edx, 2        ; +1 az indexelés miatt
                      ; +1 a sorvége miatt
    mov eax, 4
    mov ebx, 1
    mov ecx, kiir
    int 0x80

    inc dword [ebp-4] ; a következő karakter
    jmp .ciklus

.vege
    mov eax, 1
    xor ebx, ebx
    int 0x80

section .data

szoveg db      "Hello Vilag"

section .bss

kiir    resb 12

```

1. b. Hasonlóan bővülő módon írd ki az első parancssori paramétert!

```

section .text
global main

main
    push ebp
    mov ebp, esp
    sub esp, 4      ; [ebp-4] : ciklusszámláló

    mov [ebp-4], dword 0
    ; kezdetben a szöveg nulladik pozícióján állunk

.ciklus
    mov ecx, [ebp+12]
    mov ecx, [ecx+4]      ; a szöveg (argv[1])
    mov eax, [ebp-4]
    mov dl, [ecx+eax]    ; a következő karakter

    cmp dl, 0
    je near .vege

    mov [kiir + eax], dl
    mov [kiir + eax + 1], byte 0xA
    ; a következő karakter és egy sorvége

    mov edx, eax      ; a kiírás hossza
    add edx, 2
    mov eax, 4
    mov ebx, 1
    mov ecx, kiir
    int 0x80

    inc dword [ebp-4] ; a következő karakter
    jmp .ciklus

.vege
    mov eax, 1
    xor ebx, ebx
    int 0x80

section .data

szoveg db      "Hello Vilag"

section .bss

kiir    resb 12

```

2. a. i) Alkoss a saját (lefordított, futtatható) programkódját kiíró programot! Feltételezheted, hogy a program neve ismert. Használj 1024 bites puffert a fájl tartalmának beolvasása során!
- ii) Nehezítés: a program nevét a parancssori paraméterek közül kell kinyerned.
- b. i) Alkoss a saját forrásszöveget kiíró programot! Feltételezheted, hogy a program neve ismert.
- ii) Nehezítés: a program neve a futtatott fájl neve .asm kiterjesztéssel.

Feltételezheted, hogy a fájl neve kevesebb, mint 256 karakter hosszú.

section .text global main	section .text global main	section .text global main	section .text global main
<b>main</b>	<b>main</b>	<b>main</b>	<b>main</b>
push ebp mov ebp, esp			
mov eax, 5 mov ebx, fajlnev	mov eax, 5 mov ebx, [ebp+12]	mov eax, 5 mov ebx, [ebx]	mov esi, [ebp+12] ; argv mov edi, [esi]; argv[0] mov edi, fajlnev
mov ecx, 644q int 0x80	mov ecx, 644q int 0x80	mov ecx, 644q int 0x80	.masol cmp byte [esi], 0 je near.masolas.vege
mov [leiro], eax	mov [leiro], eax	mov [leiro], eax	mov al, [esi] mov [edi], al
<b>.beolvas</b>	<b>.beolvas</b>	<b>.beolvas</b>	inc esi inc edi
mov eax, 3 mov ebx, [leiro] mov ecx, adat mov edx, 1024 int 0x80	mov eax, 3 mov ebx, [leiro] mov ecx, adat mov edx, 1024 int 0x80	mov eax, 3 mov ebx, [leiro] mov ecx, adat mov edx, 1024 int 0x80	jmp .masol
cmp eax, 1024 jne near .vege	cmp eax, 1024 jne near .vege	cmp eax, 1024 jne near .vege	.masolas.vege mov byte [edi], '.' mov byte [edi+1], 'a' mov byte [edi+2], 's' mov byte [edi+3], 'm' mov byte [edi+4], 0
mov eax, 4 mov ebx, 1 mov ecx, adat mov edx, 1024 int 0x80	mov eax, 4 mov ebx, 1 mov ecx, adat mov edx, 1024 int 0x80	mov eax, 4 mov ebx, 1 mov ecx, adat mov edx, 1024 int 0x80	mov eax, 5 mov ebx, fajlnev mov ecx, 644q int 0x80 mov [leiro], eax
jmp .beolvas	jmp .beolvas	jmp .beolvas	.beolvas mov eax, 3 mov ebx, [leiro] mov ecx, adat mov edx, 1024 int 0x80
<b>.vege</b>	<b>.vege</b>	<b>.vege</b>	cmp eax, 1024 jne near .vege
mov edx, eax mov eax, 4 mov ebx, 1 mov ecx, adat int 0x80	mov edx, eax mov eax, 4 mov ebx, 1 mov ecx, adat int 0x80	mov edx, eax mov eax, 4 mov ebx, 1 mov ecx, adat int 0x80	mov eax, 4 mov ebx, 1 mov ecx, adat int 0x80
mov eax, 1 mov ebx, 0 int 0x80	mov eax, 1 mov ebx, 0 int 0x80	mov eax, 1 mov ebx, 0 int 0x80	jmp .beolvas
<b>section .data</b> fajlnev db "a.out", 0		<b>section .data</b> fajlnev db "listazo2.asm", 0	<b>.vege</b> mov edx, eax mov eax, 4 mov ebx, 1 mov ecx, adat int 0x80
<b>section .bss</b> adat resb 1024 leiro resd 1	<b>section .bss</b> adat resb 1024 leiro resd 1	<b>section .bss</b> adat resb 1024 leiro resd 1	mov eax, 1 mov ebx, 0 int 0x80
			<b>section .bss</b> adat resb 1024 leiro resd 1

3. Írj olyan eljárást, amely egy duplaszón elférő, előjel nélküli egész számot ír ki a képernyőre tízes számrendszerben! A számot paraméterként kapja meg az eljárás. A kiírandó string legyen lokális változó (fix hosszúságú ASCII karaktertömb); az eljárásnak ezt töltse fel, majd hívja meg a kiírást.

```
kiir
    push    ebp
    mov     ebp, esp
    sub     esp, 12          ; ezen a 12 byte-on alkotjuk meg a számot

    mov     byte [ebp-1], 0xA
    mov     byte [ebp-2], '0'      ; kezdetben a szám egy nulla karakterből és egy sorvégéből áll

    mov     edi, ebp
    sub     edi, 2          ; edi a nulla karakteren áll
    mov     eax, [ebp+8]      ; eax az első paraméter

    cmp     eax, 0          ; ha a kapott paraméter nulla, akkor nullát kell kiírni
    je     near .kiir

.ciklus
    cmp     eax, 0          ; ha a számból fennmaradó kiírandó rész nulla, akkor készen
    je     near .vege        ; vagyunk az átalakítással

    xor     edx, edx        ; 32 bitesen osztunk, ehhez kinullázzuk edx:eax felső felét

    mov     ecx, 10         ; osztás tízzel, a maradék (a következő karakter) edx-ben
    div     ecx             ; jelenik meg; mivel értéke 0..9 közötti, ezért dl-ben elfér

    add     dl, 30h         ; dl eddig szám volt, most számjeggyé alakítjuk
    mov     [edi], dl        ; ... és beírjuk az aktuális pozícióra

    dec     edi             ; a mutató léptetése
    jmp     .ciklus

.vege
    inc     edi             ; a ciklusmag utolsó végrehajtása végén túlléptettük edi-t,
    ; ezt itt korrigáljuk

.kiir
    mov     eax, 4
    mov     ebx, 1
    mov     ecx, edi        ; kiírni az aktuális pozícióról fogunk
    mov     edx, ebp
    sub     edx, ecx        ; ki kell írni az összes karaktert edi-től ebp-ig
    int     0x80

    mov     esp, ebp
    pop     ebp
    ret             ; visszatérés az alprogramból
```

## Makrók

1. Írd meg az adc utasítást általánosan, kétparaméteres makróként!

```
%macro    adc  2
    jnc %%nincs_carry          ; ha nincs beállítva az átviteli bit, az utasítás úgy működik,
    inc %1                     ; mint az összeadás
                                ; különben egygyel meg kell növelni a regiszter értékét

%%nincs_carry
    add %1, %2
%endm
```

2.

				/	\				
				/	\				
			/	\					
+	-	-	-	-	-	-	-	-	+
					o	o			
					o	o			
=	=	=	=	=	=	=	=	=	=

	v	v	v	v	v	v	v	v	
=	=	=	=	=	=	=	=	=	=
		X			X				
			~	-	-	-	~		
V	-	-	-	-	-	-	-	-	V

3. a. Készíts deriváló makrót. A makró tetszőleges számú paramétert kaphat, amelyek egy polinom együtthatóit adják. A makró tegye a verembe sorban a polinom deriváltjának együtthatóit szavakként! A deriválás szabálya:

$$(c \cdot x^n)' = c \cdot n \cdot x^{n-1}$$

b. Készíts integráló makrót a deriváló makróhoz hasonlóan. A konstans tagot nem kell figyelembe venni. Feltehető, hogy minden kapott együttható egész szám.

```
%macro      derival   1-*
%assign     %%i      %0
%rep %0 - 1
    push    dword ( %1 * %%i )
    %assign  %%i      ( %%i - 1 )
    %rotate 1
%endrep
%endm
```

```
%macro      integral  1-*
%assign     %%i      %0 + 1
%rep %0 - 1
    push    dword ( %1 / %%i )
    %assign  %%i      ( %%i - 1 )
    %rotate 1
%endrep
%endm
```

4. Add meg, hogy a parancssori paraméterként kapott file a legvégén tartalmazza-e a saját nevét. A fájl egy sorvége karakterrel zárul, ezt ne vedd figyelembe.

```
section .text
global main

main
    push ebp
    mov ebp, esp

    mov esi, [ebp+12]
    mov esi, [esi+4]          ; esi a fájlnév kezdetére mutat
    mov eax, -1

.hossz
    inc eax
    inc esi
    cmp byte [esi-1], 0
    jne .hossz              ; megszámoljuk, hány karakter hosszú a fájlnév

    mov [hossz], eax

    mov eax, 5
    mov ebx, [ebp+12]
    mov ebx, [ebx+4]
    mov ecx, 0
    mov edx, 777q
    int 0x80                 ; megnyitjuk a fájlt

    mov [leiro], eax          ; eltároljuk a fájlleírót

    mov eax, 19
    mov ebx, [leiro]
    mov ecx, [hossz]
    neg ecx
    dec ecx
    mov edx, 2
    int 0x80                 ; negatív irányban kell eltolni, a fájl végéről visszafelé
                                ; a fájl végén levő sorvége figyelen kívül hagyása
                                ; pozicionálás a szöveg kezdetére a fájl végén

    mov eax, 3
    mov ebx, [leiro]
    mov ecx, filevege
    mov edx, [hossz]
    int 0x80                 ; beolvasás

    mov ecx, [hossz]
    mov esi, [ebp+12]
    mov esi, [esi+4]
    mov edi, fajlvege
    mov edi, [esi+4]          ; ecx a ciklusszámláló
                                ; esi a fájlnév kezdetére mutat
                                ; edi a fájl végéről származó, vizsgálandó szövegre
                                ; mutat

.vizsgal
    cmp ecx, 0
    je .egyezik              ; ha minden olvasott karakter egyezett, és már nincs
                                ; több, akkor a szöveg megfelelő

    mov al, [esi]
    cmp al, [edi]
    jne .nemegyezik          ; a következő karakter vizsgálata; ha eltér,
                                ; a szöveg nem megfelelő

    inc esi
    inc edi
    dec ecx
    jmp .vizsgal              ; ha egyezett a karakter, a következő vizsgálatával
                                ; folytatjuk
```

```
section .bss
hossz      resd 1           ; a fájl kiszámított hosszát
                            ; itt tároljuk átmenetileg

leiro      resd 1           ; a fájl leírójának helye

fajlvege   resb 16          ; ide olvassuk fel a fájl

                            ; feltételezzük, hogy 16
                            ; karakternél nem hosszabb
```

## Irodalom

### (1) [http://www.intel.com/design/intarch/intel386/docs\\_386.htm](http://www.intel.com/design/intarch/intel386/docs_386.htm)

Az Intel, az x86 architektúra megalkotónak honlapja, azon belül a 80386-os processzor dokumentációs oldala. Innen elérhetőek a Software Developer's Manual (Szoftverfejlesztői kézikönyv) kötetei. Nagyon részletesen, több száz oldalon keresztül mutatja be az architektúra minden részletét.

### (2) <http://developer.amd.com/documentation.aspx>

Az AMD, az x86 architektúrájú processzorok másik népszerű gyártójának honlapja. Az Architecture Programmer's Manual (Architektúra-programozói kézikönyv) írja le a processzorok felépítését és programozását. Az általunk tárgyalt pontokon nem tér el az Intel architektúrától.

### (3) <http://nasm.sourceforge.net/>

A Netwide Assembler honlapja.

- <http://nasm.sourceforge.net/doc/nasmdoc0.html>

A Netwide Assembler on-line dokumentációjának tartalomjegyzéke.

- <http://nasm.sourceforge.net/doc/nasmdoc4.html>

A dokumentáció előfordítási fázisról (makrókról) szóló fejezete.

- <http://developer.apple.com/documentation/DeveloperTools/nasm/nasmdocb.html>

Az utasításokat bemutató függelék a dokumentációban. A munkafüzet írásakor a nasm hivatalos honlapján nem volt elérhető, pedig gyakorló assembly programozó számára a dokumentáció talán leghasznosabb része.

### (4) <http://asm.sourceforge.net/>

Linux alatti assembly programozáshoz hasznos oldal. Régebben <http://linuxassembly.org/> címen működött.

utasítás mnemonikja		operandusok		működés				
nop		<i>nincs operandusa</i>		semmit sem csinál				
mov		*		cél := forrás				
adatmozgatás	movsx	reg,	reg/mem	cél := forrás előjeles/előjel nélküli kiterjesztéssel				
	xchg	reg, mem,	reg/mem reg	cél := forrás, forrás := cél				
not		reg/mem		cél := cél bitenként negálva				
logikai	bt	bts	btc	btr	r/m <sup>32/16</sup> , r <sup>32/16</sup> /k <sup>8</sup>	átvitel := cél forrás-adik bitje, bit marad/0/1/fordul		
	<i>setfeltétel</i>			reg <sup>8</sup> /mem <sup>8</sup>		cél := 1, ha teljesül a feltétel, cél := 0, ha nem		
aritmetikai	and	or	xor		*	cél := cél és/vagy/kizáró vagy forrás bitenként		
	add	sub			*	cél := cél + forrás <i>illetve</i> cél := cél - forrás		
	inc	dec				cél := cél + 1 <i>illetve</i> cél := cél - 1		
	neg					cél := 0 - cél		
	mul				reg/mem	előjel nélküli	ax := al · forrás dx:ax := ax · forrás edx:eax := eax · forrás,	vagy vagy az op. méretétől függően
	imul					előjeles	al := ax / forrás, maradék: ah ax := dx:ax / forrás, maradék: dx eax := edx:eax / forrás, maradék: edx	vagy vagy
	div					előjel nélküli		
	idiv					előjeles		
	<i>léptetések</i>		r/m, konst/cl		külön táblázatban (a 2. operandus lehet a cl regiszter is)			
	cmp		*		összehasonlítás, a jelzőbitek beállítása			
vezérlésátadás	jmp	<i>jfeltétel</i>			címke	ugrás feltétel nélkül vagy feltétellel; <i>külön táblázatban</i>		
	call					alprogramhívás		
	ret		<i>nincs operandusa</i>			visszatérés alprogramhívásból		
	int		konst <sup>8</sup>			szoftveres kivétel kiváltása		
	push		r <sup>32/16</sup> /m <sup>32/16</sup> /k <sup>32/16/8</sup>			érték verembe helyezése		
véremkezelés	pop		reg <sup>32/16</sup> /mem <sup>32/16</sup>			érték visszatöltése a veremből		
	pushad		<i>nincs operandusa</i>			eax, ecx, edx, ebx, esp, ebp, esi, edi verembe mentése		
	popad		<i>nincs operandusa</i>			edi, esi, ebp, esp, ebx, edx, ecx, eax töltése a veremből		

\* Ezeknél az utasításoknál a reg/mem, reg/mem/konst operandusok megengedettek, a mem/mem kivételével.

feltételes ugrások	cél < forrás	cél ≤ forrás	cél ≥ forrás	cél > forrás	igaz	hamis
nem előjeles	jb	jnae	jbe	jna	je	jne
előjeles	jl	jnge	jle	jng	jc	jnc

előjel nélküli léptetés	előjeles léptetés	forgatás	31.....16	15...8	7....0
shr 0 → felső bit 0. bit → átvitel	sar felső bit 0. bit → átvitel	ror felső bit 0. bit → átvitel			
shl átvitel ← felső bit 0. bit ← 0	sal átvitel ← felső bit 0. bit ← 0	rol átvitel ← felső bit 0. bit ← 0			

A fentiekben r = reg = regiszter, m = mem = memóriatartalom, k = konst = konstans.

A felső indexben szereplő 8, 16 és 32 a méretet jelenti bitben.

