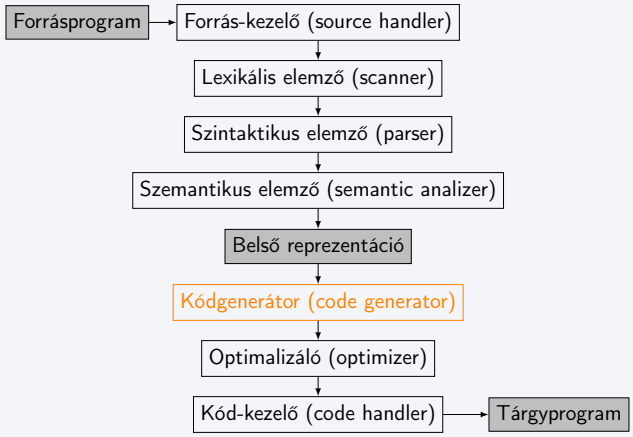
Kódgenerálás I.
(kifejezések és vezérlési szerkezetek)

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 megvalósító 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 konstruktor*)

Összetett típusokra vontakozó értékadások

- bizonyos típusokra (pl. rekordok, tömbök) könnyen lehet alapértelmezett másoló eljárást készíteni:
  - mezőkénti / elemenkénti másolás
- láncolt adatszerkezetek másolási stratégiái:
  - rekurzívan mindent 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

1. a feltételt az `al` regiszterbe kiértékelő kód
2. `cmp al,1`
3. `jne near Vége`
4. a then-ág programjának kódja
5. `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 hoszabb 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

1. a feltételt az `al` regiszterbe kiértékelő kód
2. `cmp al,1`
3. `je Then`
4. `jmp Vége`
5. `Then:` a then-ág programjának kódja
6. `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 konkatenálni:

```
stringstream ss;
ss << ''Lab'' << szamlalo++;
string cimkenev = ss.str();
```


Több ágú elágazás alakja

statement →
if $condition_1$ *then* $program_1$
elseif $condition_2$ *then* $program_2$
...
elseif $condition_n$ *then* $program_n$
else $program_{n+1}$ *end*

Több ágú elágazás kódja

1. az 1. feltétel kiértékelése az `al` regiszterbe
2. `cmp al,1`
3. `jne near Feltétel_2`
4. az 1. ág programjának kódja
5. `jmp Vége`
6. ...
7. `Feltétel_n:` az n-edik feltétel kiértékelése az `al` regiszterbe
8. `cmp al,1`
9. `jne near Else`
10. az n-edik ág programjának kódja
11. `jmp Vége`
12. `Else:` az else ág programjának kódja
13. `Vége:`

A switch-case utasítás fordítása

- alakja:
  statement → *switch variable*
  case $value_1$: $program_1$
  ...
  case $value_n$: $program_n$
- 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 hajtódik végre
  - C stílus: az első teljesülő ágtól kezdve az összes végrehajtódik (hacsak nem használunk `break` utasítást az ágak végén)

A switch-case utasítás fordítása (Ada stílus)

1. `cmp [Változó],Érték_1`
2. `jne near Feltétel_2`
3. első ág programjának kódja
4. `jmp Vége`
5. `Feltétel_2: cmp [Változó],Érték_2`
6. `...`
7. `Feltétel_n: cmp [Változó],Érték_n`
8. `jne near Vége`
9. n-edik ág programjának kódja
10. `Vége:`

A switch-case utasítás fordítása (C stílus)

1. `cmp [Változó],Érték_1`
2. `je near Program_1`
3. `cmp [Változó],Érték_2`
4. `je near Program_2`
5. `...`
6. `cmp [Változó],Érték_n`
7. `je near Program_n`
8. `jmp Vége`
9. `Program_1:` az 1. ág programjának kódja
10. `...`
11. `Program_n:` az n-edik ág programjának kódja
12. `Vége:`

Elöl tesztelő ciklus fordítása

- alakja: *statement* → *while condition program end*
- generálandó kód:

Elöl tesztelő ciklus kódja

1. `Eleje:` a ciklusfeltétel kiértékelése az al regiszterbe
2. `cmp al,1`
3. `jne near Vége`
4. a ciklusmag programjának kódja
5. `jmp Eleje`
6. `Vége:`

Hátul tesztelő ciklus fordítása

- alakja: *statement* → *loop program while condition*
- generálandó kód:

Hátul tesztelő ciklus kódja

1. `Eleje:` a ciklusmag programjának kódja
2. a ciklusfeltétel kiértékelése az al regiszterbe
3. `cmp al,1`
4. `je near Eleje`

For ciklus fordítása

- alakja:
  statement → *for variable from* $value_1$ *to* $value_2$ *program end*
- hasonlít a while ciklusok fordításához
  - hiszen minden for ciklus átalakítható while ciklussá

For ciklus kódja

1. a „from” érték kiszámítása a [Változó] memóriahelyre
2. `Eleje:` a „to” érték kiszámítása az eax regiszterbe
3. `cmp [Változó],eax`
4. `ja near Vége`
5. a ciklusmag kódja
6. `inc [Változó]`
7. `jmp Eleje`
8. `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
```

Ciklusváltozó tárolása regiszterben

- 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 elmentése

```
push ecx
; ciklusmag
pop ecx
```


Változódefiníciók fordítása

- sokféle változó van: statikus, lokális, dinamikusan allokált...
- most a statikus változókkal foglalkozunk
  - globális változók
  - kifejezetten statikusnak deklarált változók (pl. *C++* `static` kulcsszó)
- a statikus változók a `.data` vagy `.bss` szakaszban kapnak helyet
- a többi változóval a következő előadáson foglalkozunk

A statikus változók definíciójának fordítása

- kezdőérték nélkül: `int x;`

Kezdőérték nélküli változódefiníció fordítása

```
section .bss
; a korábban definiált változók...
Lab12: resd 1   ; 1 x 4 bájtnyi terület
```

- kezdőértékkel: `int x=5;`

Kezdőértékkel adott változódefiníció 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!

Kifejezéskiértékelés fordítása

- 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 címkéje
```

- 3. eset: összetett kifejezés (pl. `fibonacci(25) + factorial(x)`)
  - általános megoldás: függvényhívásokat teszünk a kódba
  - a beépített függvényekhez (+,-,*,/,&&,||,!,...) lehet hatékonyabban is

Beépített függvényekből álló kifejezések

1. próbálkozás ($kifejezés_1$+$kifejezés_2$)

```
; 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 ($kifejezés_1$+$kifejezés_2$)

```
; a 2. kifejezés kiértékelése eax-be
push eax
; az 1. kifejezés kiértékelése eax-be
pop ebx
add eax,ebx
```


Logikai kifejezések fordítása

- cél: az `al` regiszterbe kiértékelni a logikai kifejezés eredményét
- 1. eset: $<$, $>$, $=$, ... operátorok

$kifejezés_1$ < $kifejezés_2$ 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, kizáróvagy, ...* műveletek (hasonlóan a $+$, $-$, ... kiértékeléséhez)

$kifejezés_1$ *és* $kifejezés_2$ 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...

Rövidzáras logikai operátorok

- Ha *kifejezés*$_1$ `&&` *kifejezés*$_2$ kifejezésben az első hamis, akkor a másodikat garantáltan nem értékeli ki.
  - fontos lehet: `(a != 0) && (c == b / a)`

$kifejezés_1$ *és* $kifejezés_2$ 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:
```


Kódgenerálás és $S-ATG$

- Emlékeztető: S-attribútum fordítási grammatika
  - csak szintetizált („alulról felfelé" terjedő) attribútumok
  - jól illeszkedik az alulról felfelé elemzésekhez (*bisonc++*)

Kódgenerálás és $S-ATG$

- Emlékeztető: S-attribútum fordítási grammatika
  - csak szintetizált („alulról felfelé" terjedő) attribútumok
  - jól illeszkedik az alulról felfelé elemzésekhez (*bisonc++*)
- 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

Kódgenerálás és $S-ATG$

- Emlékeztető: S-attribútum fordítási grammatika
  - csak szintetizált („alulról felfelé" terjedő) attribútumok
  - jól illeszkedik az alulról felfelé elemzésekhez (*bisonc++*)
- 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

A `goto` utasítás fordítása

Forrásprogram

```
Lab: x++;
     ...
     goto Lab;
```

Generálandó kód

```
Lab: inc [X]
     ...
     jmp Lab
```

Vigyázni kell a következőkre:

- A felhasználó címkéje nehogy egybe essen egy generált címkével.
  - vagy generálni kell a felhasználó címkéje 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 alprogramokból is ki lehet ugrani...

A break utasítás fordítása

- A break utasítás a legbelső 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 kitöltjük a hiányzó címkét

A break és az $L - ATG$-k

- Emlékeztető: L-attribútum fordítási grammatika
  - 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 szintaxisfában elkészíthető 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 utasássorozat, ami utasításokból áll és egy utasítás lehet break is...)