9. Programok fordítása és végrehajtása

Bevezetés

Amikor programot írunk, azt valamilyen programozási nyelven tesszük. Ezt a programozási nyelvtől függvényében vagy lefordítjuk a gép által értelmezhető kódra, vagy interpreterrel futtatjuk azt.

Fordítás és Interpretálás

Fordítás

A fordítás során általában egy magas szintű programozási nyelvből gépi kód keletkezik, amelyet a processzor már képes értelmezni és futtatni.

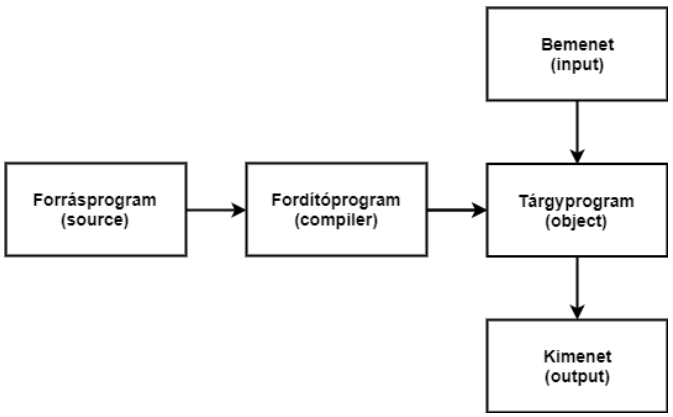
A következő elönyei vannak:

- **Gyors**, mivel a lexikális, szintaktikus és szemantikus elemzés fordítási időben, egyszer fut le, valamint ekkor optimalizáljuk a kódot.
- **Fordítási időben sok hibát ki lehet szűrni**, ezáltal megkönnyítve a hibakeresést (debuggolást).
- A gépi kód **nehezen visszafejthető**. (Reverse-engineering nehézkes)

Általában nagyobb programokhoz használjuk, ahol fontos a hatékonyság. A lefordított kódon később már nem (vagy csak nagyon nehezen) tudunk változtatni.

Hátránya, hogy a keletkezett kód nem platformfüggetlen, minden architektúrára különkülön le kell fordítani.

Például: C, C++, Ada, Haskell



ábra 1: A fordítás folyamata

Interpretálás

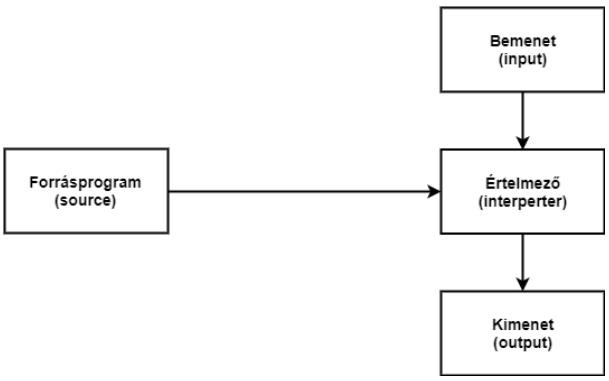
Az interpretálás során az értelmező a programkódot futás közben hajtja végre.

A következő elönyei vannak:

- Az interpretert mindössze **egyszer kell csak megírni** az adott rendszerre.
- **Platformfüggetlen**.

Hátránya, hogy **nehéz** benne **a hibakeresés**. Sok olyan hiba maradhat a kódban, amit egy fordító kiszűrt volna (pl. típus egyezőség).

Például: PHP, JavaScript, ShellScript



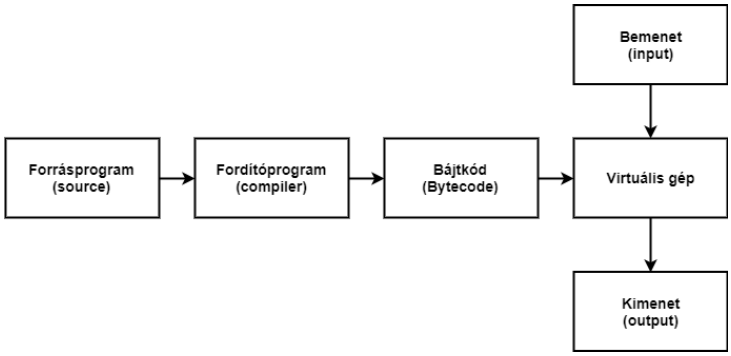
ábra 2: Az interpretálás folyamata

Fordítás és Interpretálás együtt

Egyes nyelvek (pl. Java) előfordítást használnak, melynek eredménye a *bájtkód*, amely gépi kód egy virtuális gép számára.

A következő elönyei vannak:

- Elérhető a **fordítási idejű hibaellenőrzés és optimalizálás**.
- **Platformfüggetlenség**.



ábra 3: Az interpretálás folyamata

Fordítási egység és a szerkesztés

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

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

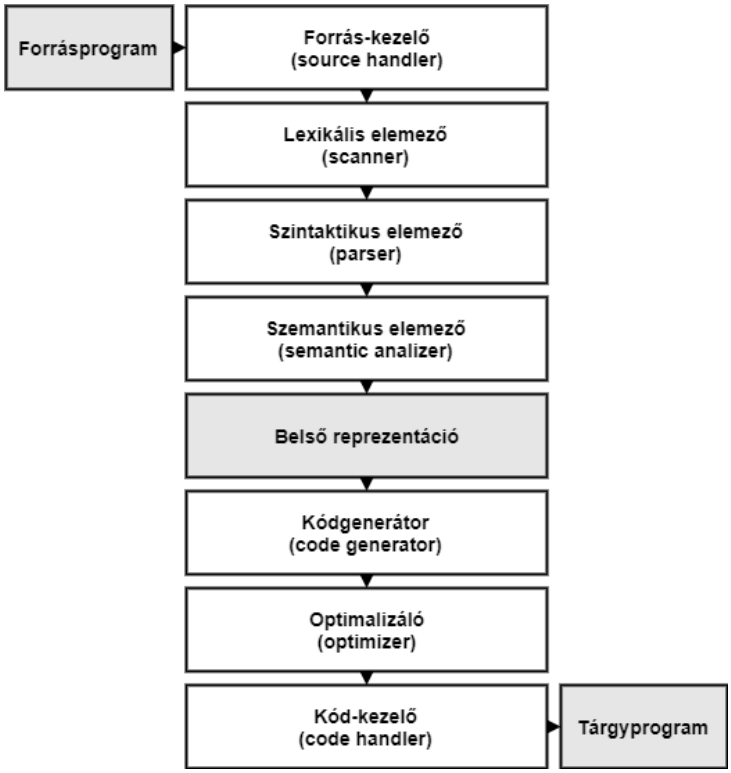
A tárgykód létrehozása két fázisban történik.

- Először a forrásfájlokat *lefordítjuk*, ebből keletkezik az un. *objektumkód* (pl.: .obj, .class). Ebben a gépi utasítások már megvannak, de hiányznak belőle a hivatkozások (pl változók, függvények), melyek más fájlokban vannak megvalósítva. *Fordítási egységnek* nevezzük azt, amiből egy objektumkód keletkezik.
- Második lépésben a *linker* (szerkesztő) feladata, hogy a hiányzó referenciákat kitöltse, hogy egyetlen fájlt generálva futtatható kódot kapjunk.

  A linkelés lehet:

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

A fordítóprogram komponensei



ábra 4: A fordítás lépései

Lexikális elemző

A lexikális elemző feladata, hogy tokenekre bontsa a forráskódot.

- Bemenete maga a forráskód.
- Adott egy a nyelvre jellemző reguláris (hármas típusú) nyelvtan. Ez adja meg, hogy milyen típusú tokenek szerepelhetnek a forrásban. A tokenekhez tulajdonságokat rendelhet (pl. változó neve, literál értéke).
- Kimenete ez a tokensorozat. Amennyiben az elemző olyan karaktersorozatot talál, amelynek nem feleltethető meg token, akkor az lexikális hibát vált ki.

  Megjegyzés: Lexikális hibánál nem feltétlen szakad meg a fordítás folyamata, megpróbálhatjuk átugrani az adott részt és folytatni az elemzést, így ha több hiba is van, akkor azokat egyszerre jelezhetjük.

A reguláris kifejezéseket *véges determinisztikus automatákkal* ismerjük fel.

Amennyiben egy lexikális elemre az egyik automata elfogadó állapotba kerül, úgy felismertünk egy tokent. Egy karaktersorozatot egyszerre több automata is felismerhet.

Ha a felismert tokenek *azonos hosszúak*, akkor *a nyelv konfliktusos*. Ennek nem szabad előfordulnia. Az viszont lehetséges, hogy egy szót, és az ő prefixét is felismerte egy automata. Ekkor mindig a hosszabbat választjuk.

Szintaktikus elemző

A szintaxis elemző bemenete a lexikális elemző kimenete.

Feladata, hogy

- *szintaxisfát* építsen a tokenekből, a nyelvez tartozó egy környezetfüggetlen (kettes típusú) grammatika alapján, vagy ha ez lehetetlen, akkor
- jelezze ezt *szintaktikus hibaként.*

LR0 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

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.
Az automata bizonyos nyelvek esetén konfliktusos lehet: nem tudjuk eldönteni, hogy léptessünk vagy redukáljunk.

LR1 elemzés

Az előző problémára kínál megoldást, kibővítve a lehetséges nyelvek halmazát.
Az ötlet, hogy *olvassunk előre* egy szimbólumot.
Ha az aktuális állapot i, és az előreolvasás eredménye az a szimbólum:

- ha $[A \to \alpha.a\beta, b] \in I_i$ és $read(I_i, a) = I_j$ akkor léptetni kell, és átlépni a j állapotba.
- ha $[A \to \alpha., a] \in I_i (A \neq S')$, akkor redukálni kell az $A \to \alpha$ szabály szerint.
- ha $[S' \to S., \#] \in I_i$ és $a = \#$, akkor el kell fogadni a szöveget, minden más esetben hibát kell jelezni.

  Ha az i állapotban A kerül a verem tetejére: ha $read(I_i, A) = I_j$, akkor át kell lépni a j állapotba, egyébként hibát kell jelezni.

Jelmagyarázat/Kanonikus halmazok

Closure/lezárás

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

$I \subseteq closure(I)$ ha $[A \to \alpha.B\gamma, a] \in closure(I)$, és $B \to \beta$ a grammatika egy szabálya, akkor $\forall b \in FIRST1(\gamma a)$ esetén $[B \to .\beta, b] \in closure(I)$

Read/olvasás

Ha I a grammatika egy $LR(1)$ elemhalmaza, X pedig terminális vagy nemterminális szimbóluma, akkor $read(I, X)$ a legszűkebb olyan halmaz, amely az alábbi tulajdonsággal rendelkezik: Ha $[A \to \alpha.X\beta, a] \in I$, akkor $closure([A \to \alpha X.\beta, a]) \subseteq read(I, X)$.

LR(1) kanonikus halmazok (I_n)

- $closure([S' \to .S, \#])$ a grammatika egy kanonikus halmaza.
- Ha I a grammatika egy kanonikus elemhalmaza, X egy terminális vagy nemterminális szimbóluma, és $read(I, X)$ nem üres, akkor $read(I, X)$ is a grammatika egy kanonikus halmaza.
- Az első két szabállyal az összes kanonikus halmaz előáll.

Szemantikus elemző

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 elemzéshez ki kell egészítenünk a grammatikát. Rendeljünk a szimbólumokhoz attribútumokat és a szabályokhoz akciókat! Egy adott szabályhoz tartozó feltételek csak a szabályban előforduló attribútumoktól függhetnek. (Ha egy feltétel nem teljesül, akkor szemantikus hibát kell jelezni!). A szemantikus rutinok csak annak a szabálynak az attribútumait használhatják és számíthatják ki, amelyikhez az őket reprezentáló akciószimbólum tartozik. Minden szintaxisfában minden attribútumértéket pontosan egy szemantikus rutin határozhat meg. Az így létrejövő nyelvtant ***attribútum fordítási grammatikának*** (ATG) hívjuk.
A *jól definiált attribútum fordítási grammatika*, olyan attribútum fordítási grammatika, amelyre igaz, hogy a grammatika által definiált nyelv mondataihoz tartozó minden szintaxisfában minden attribútum értéke egyértelműen kiszámítható.

Egy attribútumot kétféleképpen lehet meghatározni:

- **Szintézissel**: a szintaxisfában alulról felfelé terjed az információ, egy szülő attribútumát a gyerekekből számoljuk. Kitüntetettnek hívjuk azokat az attribútumokat, melyeket a lexikális elemző szolgáltat.
- **Öröklődéssel**: a szintaxisfában felülről lefelé terjed az információ. A gyerekek attribútumait a szülőé határozza meg.

Az *L-ATG* olyan attribútum fordítási grammatika, amelyben minden $A \to X_1X_2...X_n$szabályban az attribútumértékek az alábbi sorrendben meghatározhatók:

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

Amennyiben a nyelvtanunk ennek eleget tesz, úgy hatékonyan meghatározható minden attribútum.
A szemantikus elemzéshez jellemzően szimbólumtáblát használunk, verem szerkezettel és keresőfával vagy hash-táblával. Minden blokk egy új szint a veremben, egy szimbólum keresése a verem tetejéről indul.

Kódgenerálás alapvető vezérlési szerkezetekhez

A kódgenerálás feladata, hogy a szintaktikusan és szemantikusan elemzett programot tárgykóddá alakítsa. Általában szorosan összekapcsolódik a szemantikus elemzéssel.

Értékadás

assignment → *variable* **assignmentOperator** *expression*

```
// a kifejezést az eax regiszterbe kiértékelő kód
mov [variable],eax
```

Egy ágú elágazás

statement → **if** condition **then** program **end**

```
// a feltételt az al regiszterbe kiértékelő kód
cmp al,1
je Then
jmp End
Then:
// a then-ág programjának kódja
End:
```

Megjegyzés: a dupla ugrásra azért van szükség, mert a feltételes ugrás hatóköre limitált.

Több ágú elágazás

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

```
// az 1. feltétel kiértékelése az al regiszterbe
cmp al,1
jne near Condition_2
// az 1. ág programjának kódja
jmp End
...
//az n-edik feltétel kiértékelése az al regiszterbe
Condition_n:
cmp al,1
jne near Else
// az n-edik ág programjának kódja
jmp End
Else:
// az else ág programjának kódja
End:
```

Switch-case

statement → **switch** *variable*
case $value_1$: $program_1$
...
case $value_n$: $program_n$

```
cmp [variable], value_1
je near Program_1
cmp [variable], value_2
je near Program_2
.  ..
cmp [variable], value_n
je near Program_n
jmp End
Program_1:
// az 1. ág programjának kódja
.  ..
Program_n:
// az n-edik ág programjának kódja
End:
```

Ciklus

Elől tesztelő

statement → **while** *condition* statements **end**

```
Begin:
//a ciklusfeltétel kiértékelése az al regiszterbe
cmp al,1
jne near End
// a ciklusmag programjának kódja
jmp Begin
End:
```

Hátul tesztelő

statement → loop statements **while** *condition*

```
Begin:
//a ciklusmag programjának kódja
. ..
//a ciklusfeltétel kiértékelése az al regiszterbe
cmp al,1
je near Begin
```

For ciklus

statement → **for** variable **from** $value_1$ to $value_2$ statements **end**

```
// a "from" érték kiszámítása a [Változó] memóriahelyre
Begin:
// a "to" érték kiszámítása az eax regiszterbe
cmp [variable],eax
ja near End
// a ciklusmag kódja
inc [variable]
jmp Begin
End:
```

Statikus változók

Kezdőérték nélküli változódefiníció fordítása:

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

Kezdőértékkel adott változódefiníció fordítása:

```
section .data
// a korábban definiált változók...
Lab12:  dd 5 ; 4 bájton tárolva az 5-ös érték
```

Logikai kifejezések

kifejezés1 < kifejezés2

```
// a 2. kifejezés kiértékelése az eax regiszterbe
push eax
// az 1. kifejezés kiértékelése az eax regiszterbe
pop ebx
cmp eax,ebx
jb Smaller
mov al,0  // hamis
jmp End
Smaller:
mov al,1 // igaz
End:
```

kifejezés1 { és, vagy, nem, kizáróvagy } kifejezés2

```
// 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
```

lusta "és" kiértékelés

```
// az 1. kifejezés kiértékelése az al regiszterbe
cmp al,0
je End
push ax
// a 2. kifejezés kiértékelése az al regiszterbe
mov bl,al
pop ax
and al,bl
End:
```

Alprogramok megvalósítása

```
// az 1. kifejezés kiértékelése az al regiszterbe
cmp al,0
je End
push ax
// a 2. kifejezés kiértékelése az al regiszterbe
mov bl,al
pop ax
and al,bl
End:
```

Alprogramok hívása

```
// Alprogramok sémája
// utolsó paraméter kiértékelése eax-be
push eax
// ...
// 1. paraméter kiértékelése eax-be
push eax
call alprogram
add esp,'a paraméterek összhossza'
```

Kódoptimalizáló

Az optimalizálás feladata, hogy a keletkezett kód kisebb és gyorsabb legyen, úgy hogy a futás eredménye nem változik. A gyorsaság és a tömörség gyakran ellentmondanak egymásnak, és az egyik csak a másik rovására javítható.

Általában három lépésben szokás elvégezni:

- Optimalizálási lépések végrehajtása az eredeti programon, vagy egyszerűsített változatán
- Kódgenerálás
- Gépfüggő optimalizálás végrehajtása a generált kódon

Lokális optimalizáció

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 és ciklusok eleje*, *elágazások ágainak első utasítása*, *goto utasítások célpontjai*.
- az utolsó utasítás kivételével nincs benne vezérlés-átadó utasítás
  - assembly programokban: ahová a **jmp**, **call**, **ret** utasítások "ugranak"
  - magas szintű nyelvekben: *elágazások és ciklusok vége*, *eljárás vége*, *goto utasítások célpontjai*.

Az utasítás-sorozat nem bővíthető a fenti két szabály megsértése nélkül.

Jelöljük meg:

- a program első utasítását
- azokat az utasításokat, amelyekre távolról át lehet adni avezérlést
- a vezérlés-átadó utasításokat követő utasításokat

Minden megjelölt utasításhoz tartozik egy alapblokk, ami a következő megjelölt utasításig (vagy az utolsó utasításig) tart.

main: mov eax,[**Label1**]
`cmp eax,[Label2]`
`jz True`
dec dword [**Label1**]
`inc dword [Label2]`
`jmp vege`
True: inc dword [**Label1**]
`dec dword [Label2]`
End: ret

Ha az optimalizálás az alapblokkok keretein belül történik, akkor garantált, hogy az átalakításnak nincs mellékhatása. Ez a *lokális optimalizálás*.

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;`

Eredeti kód:
```
a := 6;
b := a / 2;
c := b + 5;
```

Optimalizált kód:
```
a := 6;
b := 3;
c := 8;
```

Eredeti kód:
```
x := 20 - (a * b);
y := (a * b) ^ 2;
```

Optimalizált kód:
```
t := a * b;
x := 20 - t;
y := t ^ 2;
```

Ablakoptimalizálás

Ez egy módszer a lokális optimalizálás egyes fajtáihoz. Egyszerre csak néhány utasításnyi részt vizsgálunk a kódból. 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 → helyettesítés } szabályhalmazzal
(a mintában lehet paramétereket is használni)

Példák:

- felesleges műveletek törlése: nulla hozzáadása vagy kivonása
- egyszerűsítések: nullával szorzás helyett a regiszter törlése
- regiszterbe töltés és ugyanoda visszaírás esetén a visszaírás elhagyható
- utasításismétlések törlése: ha lehetséges, az ismétlések törlése

Példa:

Ablak mérete: 1 utasítás
Szabályhalmaz:
{`mov reg,0` → `xor reg,reg`,
`add reg,0` → `elhagyható`}

Eredeti kód:
```
add eax,0
mov ebx,eax
mov ecx,0
```

Optimalizált kód:
```
; elhagyott utasítás
mov ebx,eax
xor ecx,ecx
```

Globális optimalizáció

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?

Ez lehetővé teszi az

- azonos kifejezések többszöri kiszámításának kiküszöbölését akkor is, ha különböző alapblokkokban szerepelnek
- a konstansok és változók továbbterjesztését alapblokkok között
- elágazások, ciklusok optimalizálását

Kódkiemelés

Eredeti kód:

```
if( x < 10 )
{
    b++;
    a = 0;
}
else
{
    b--;
    a = 0;
}
```

Optimalizált kód:
a = 0;

```
if( x < 10 )
{
    b++;
}
else
{
    b--;
}
```

A szekvenciális és párhuzamos/elosztott végrehajtás összehasonlítása

Szekvenciális végrehajtás:

Ilyenkor a végrehajtás egy processzoron történik. Minden művelet atomi. Egy inputhoz egy output tartozik. Két szekvenciális program ekvivalens, ha ezek a párosok megegyeznek. Nem használja fel az összes rendelkezésre álló erőforrást.

Párhuzamos végrehajtás:

Több processzoron hajtódik végre a program. A párhuzamos folyamatok egymással kommunikálva, szinkronban oldják meg az adott problémát. A konkurens program szétbontható elemi szekvenciális programokra, ezek a folyamatok. A folyamatok használhatnak közös erőforrásokat: pl. változók, adattípus objektumok, kommunikációs csatornák.
A kommunikációt általában kétféleképpen szokták megvalósítani.

Osztott memóriával.

Ekkor szinkronizálni kell, hogy ki mikor fér hozzá, hogy ne legyen ütközés.

Kommunikációs csatornával.

Garantálni kell, hogy ha egy folyamat üzenetet küld egy másiknak, akkor az meg is kapja azt, és jelezzen is vissza. Ügyelni kell, nehogy deadlock alakuljon ki.