



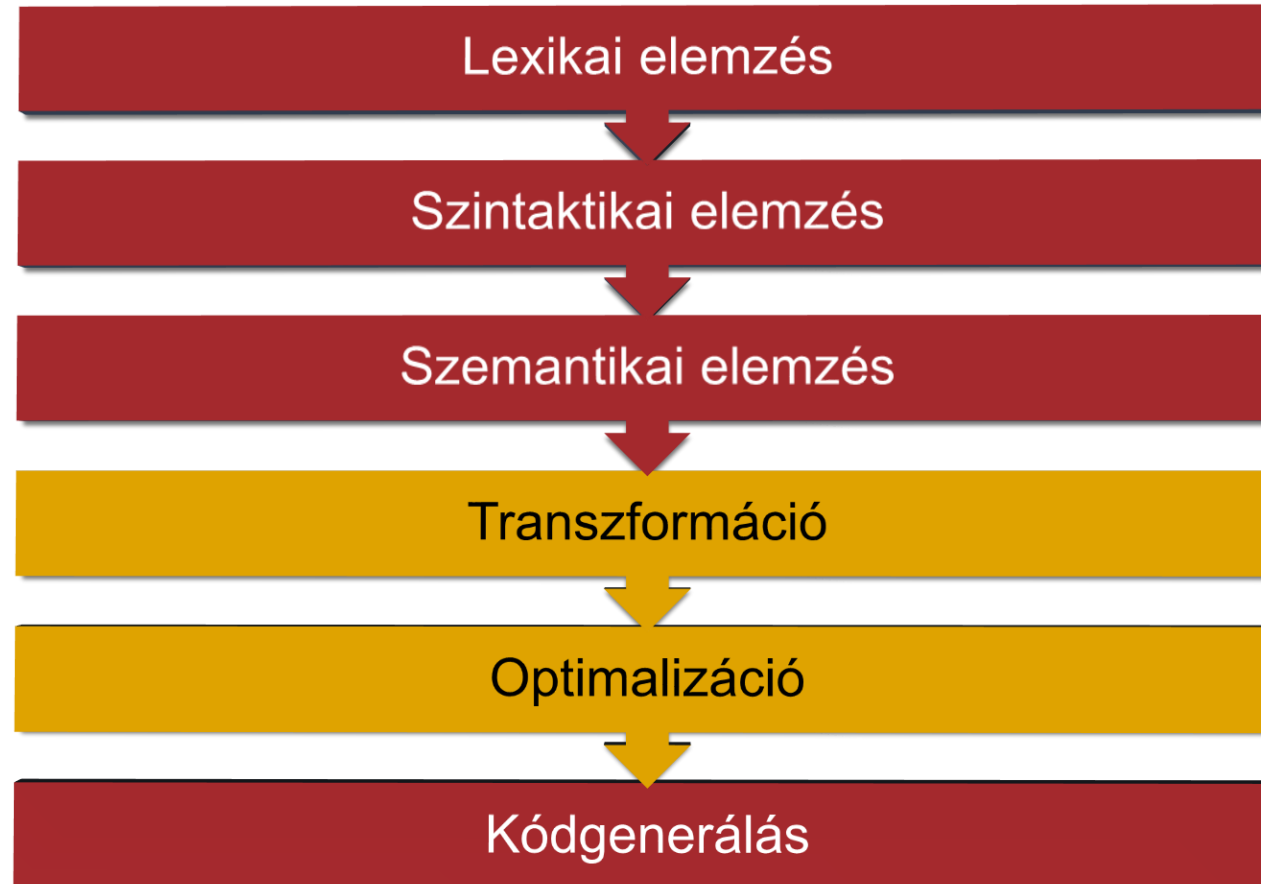
Modellalapú szoftverfejlesztés

V. előadás

Transzformáció,
optimalizálás

Dr. Simon Balázs

Fordítás fázisai



A mai előadás: Transzformáció, Optimalizálás

I. Transzformáció

II. Típusok leképezése

III. Utasítások leképezése: SSA

IV. Optimalizáció

V. Optimalizációs technikák



Transzformáció

- Cél: szintaxisfából gépközeli köztes kód (IR = Intermediate Representation)
 - > típusok, adatstruktúrák memóriakiosztása
 - > utasításokból elemi műveletek, amelyek könnyen gépi kóddá fordíthatók
 - > erőforráslimit nincs (pl. regiszterek száma nincs korlátozva)
- Gépközeli köztes kód lehetőségek:
 - > 1. Nincs explicit köztes kód: szintaxisfából közvetlen gépi kód
 - > 2. Three-Address Code (TAC v. 3AC): minden művelet max. 3 operandusú (pl. $t1 = t2 * t3$)
 - > 3. Static Single-Assignment (SSA): mint a TAC, de minden változó immutable

A mai előadás: Transzformáció, Optimalizálás

I. Transzformáció

II. Típusok leképezése

III. Utasítások leképezése: SSA

IV. Optimalizáció

V. Optimalizációs technikák



Egyszerű típusok leképezése

■ Primitív típusok:

- > pointer, bool, char, byte, short, int, long, float, double, ...
- > címezhető memóriaterületre kell illeszteni, minimális méret tipikusan 1 byte
- > kezelendő: bool értéke, char kódolása, big/little endian, fixed/floating point, hívási konvenció, ...

■ Enum/Flags:

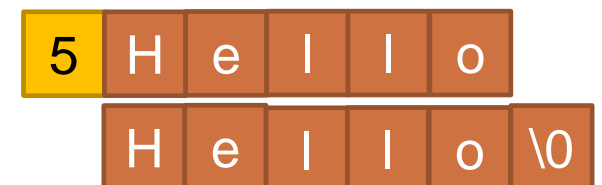
- > egészekre leképezve (byte, short, int, long, ...)
- > értékészleltől függ a méret, de van olyan nyelv (pl. C#), ahol explicit megadható

■ String típus:

- > programnyelvtől függ, hogy primitív vagy összetett típus
- > egyszerűként (pl. Pascal, C#): hossz + karakterek
- > összetettként (pl. C): karaktertömb '\0'-val lezárva

"Hello"

"Hello"



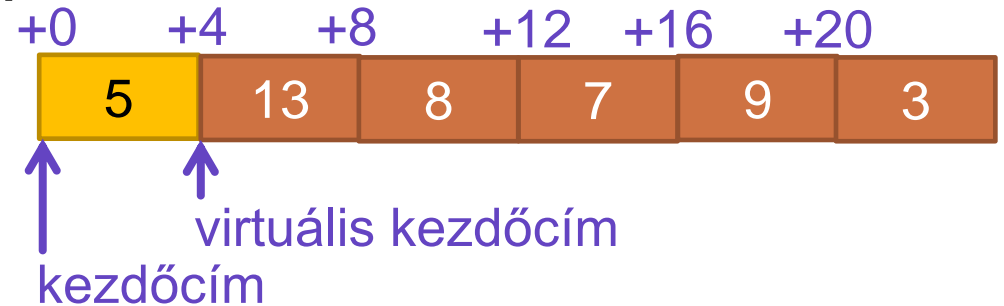
Összetett típusok leképezése: Tömb

■ Fajtái:

- > statikus: fix méret, fordítási időben foglalva a stack-en (pl. C)
- > dinamikus: fix méret, futási időben foglalva a heap-en (pl. C malloc, C#)
- > flexibilis: futásidőben változtatható (pl. Python)

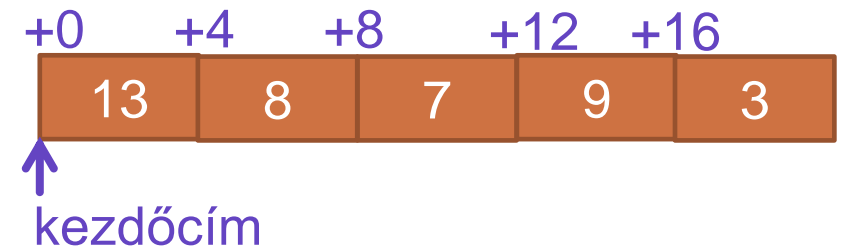
■ Ellenőrzött méret (pl. C#): hossz + elemek

```
int[] a = new int[] { 13, 8, 7, 9, 3 };
```



■ Nem ellenőrzött méret (pl. C): elemek

```
int a[] = { 13, 8, 7, 9, 3 };
```

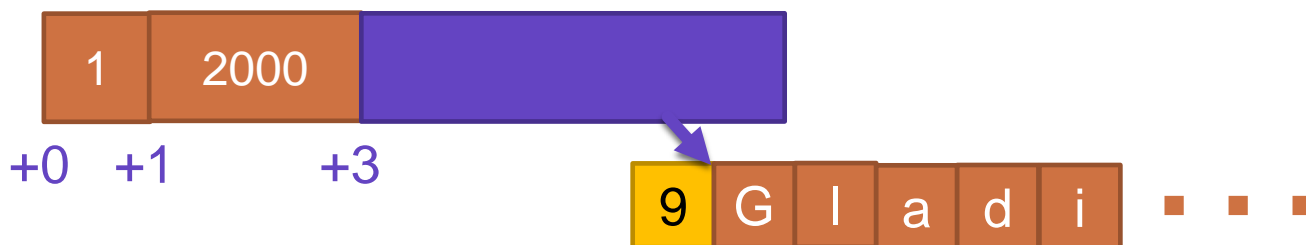


Összetett típusok leképzése: Struktúra (Unió esetén: mezők átlapolódva)

```
var m = new Movie() { Genre = Genre.Drama, Year = 2000, Title = "Gladiator" }
```

■ Tömörített (packed)

> struktúra mérete = mezők méreteinek összege

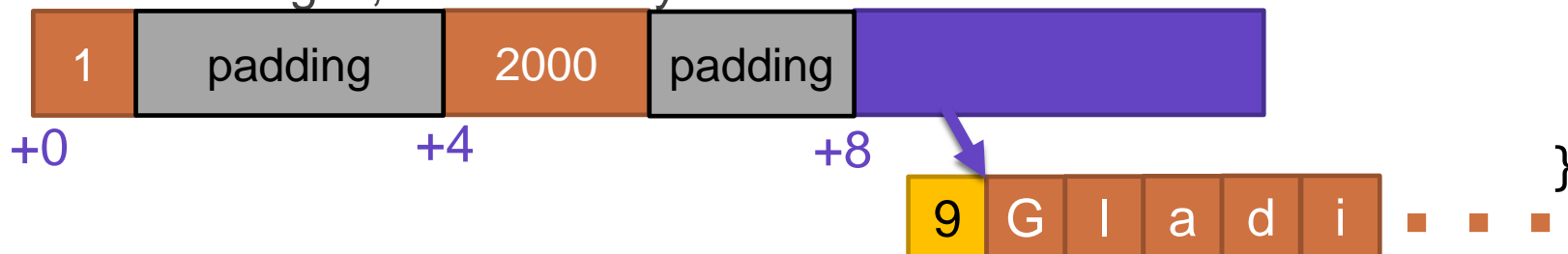


```
struct Movie
{
    Genre Genre;
    short Year;
    string Title;
}
```

■ Igazított (aligned):

> mezők szóhatárhoz igazítva

> több területet foglal, de hatékonyabb a címzés



```
enum Genre : byte
{
    Action = 0,
    Drama = 1,
    Romance = 2,
    Comedy = 3
}
```


Összetett típusok leképezése: Osztály és objektum

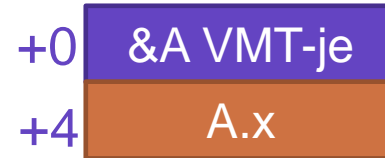
- Objektum: osztály példány szintű változói (mezői) egy struktúrába szervezve
 - > nulladik mező: mutató a típusleíróra
 - > ősosztályok tagváltozói is szerepelnek a struktúrában
- Osztály: statikus mezők + mutatók a statikus és példányszintű metódusokra
 - > statikus mezők: osztályra jellemző struktúrába szervezve
 - > tagfüggvények: implicit nulladik paraméter az objektumra mutató pointer (this)
 - > statikus függvények: nincs objektumra mutató pointer
 - > polimorfizmus: virtuális metódus tábla (VMT)
- Konstruktor:
 - > objektum lefoglalása a heap-en + inicializálás

Összetett típusok leképezése: Osztály és objektum

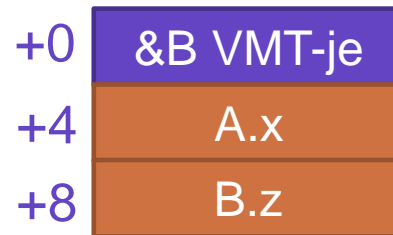
```
class A
{
    int x;
    static int y;
    void foo() { ... }
    virtual void bar() { ... }
    static void quux() { ... }
}

class B : A
{
    int z;
    static int w;
    new void foo() { ... }
    override void bar() { ... }
    virtual void garply() { ... }
}
```

Egy A objektum struktúrája:



Egy B objektum struktúrája:



A osztály struktúrája:



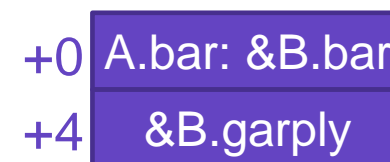
B osztály struktúrája:



A osztály VMT-je:



B osztály VMT-je:



A.foo kódja

A.bar kódja

A.quux kódja

B.foo kódja

B.bar kódja

B.garply kódja

A mai előadás: Transzformáció, Optimalizálás

I. Transzformáció

II. Típusok leképezése

III. Utasítások leképezése: SSA

IV. Optimalizáció

V. Optimalizációs technikák



Static Single Assignment (SSA)

- Minden változó pontosan egyszer kap értéket
 - > a definíciója helyén, még a felhasználása előtt
- Változók jelölése: eredeti név számozva (verziózva)
- Minden művelet max. 3 operandusú (pl. $t1 = t2 * t3$)
- Példa:

Programkód:

```
x = 3;  
y = 4;  
z = 5;  
x = x+y;  
z = (x+y)*z;
```



SSA kód:

```
x1: 3  
y1: 4  
z1: 5  
x2: x1+y1  
t1: x1+y1  
z2: t1*z1
```

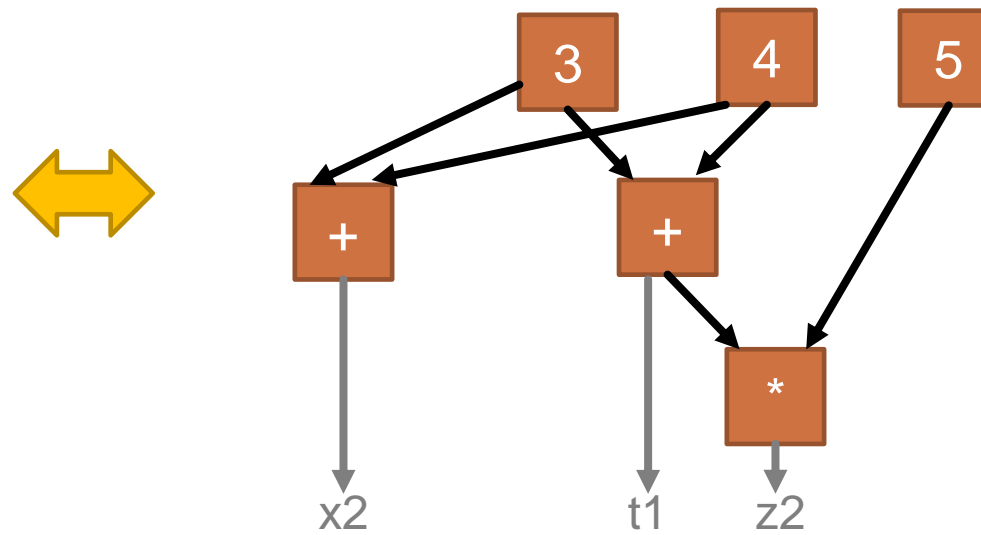
SSA gráf

- SSA kód szemléletesebb ábrázolása adatfolyam gráf (dataflow-graph, DFG) formájában
 - > Csúcsok: konstansok vagy műveletek
 - > Irányított élek: definíció-használat kapcsolatok (élek megfordítása: adatfüggőség)
- Egyes optimalizációk egyszerűbbek/szemléletesebbek gráf formában

SSA kód:

```
x1: 3  
y1: 4  
z1: 5  
x2: x1+y1  
t1: x1+y1  
z2: t1*z1
```

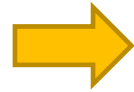
Gráf:



Φ -függvény

Kérdés: mi történik, ha két vagy több vezérlési ág találkozik?

```
if (x < y) z = x;  
else z = y;  
w = z;
```



```
if (x1 < y1) z1 = x1;  
else z2 = y1;  
w1 = ???
```

Be kell vezetni egy speciális jelölést: Φ -függvény

Értéke: a tényleges vezérlési ágnak megfelelő paraméter.

```
if (x < y) z = x;  
else z = y;  
w = z;
```



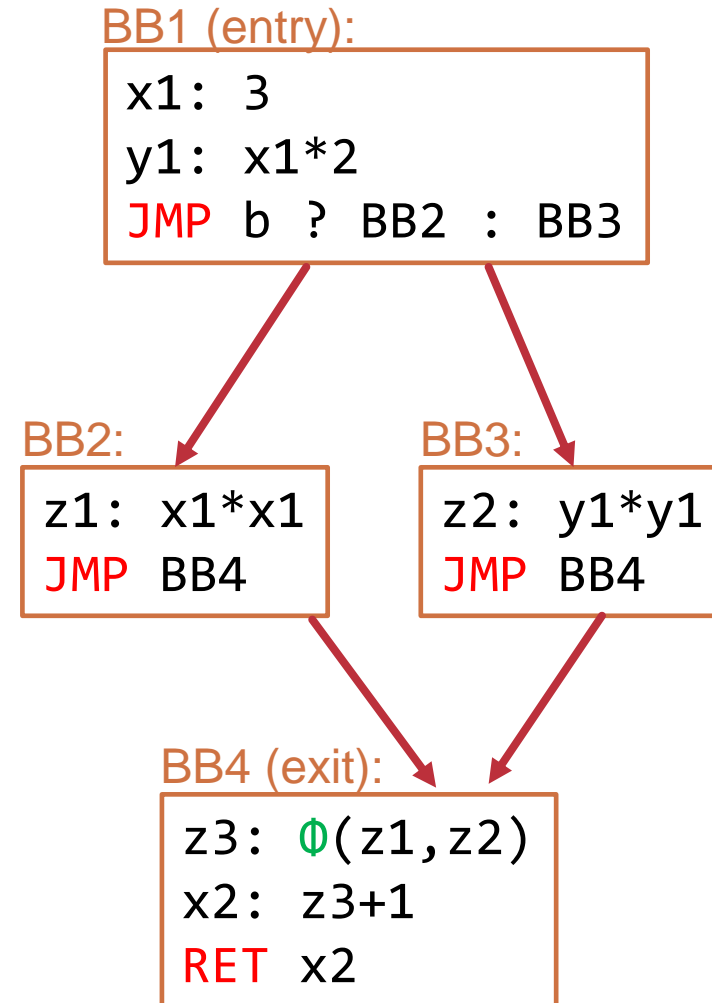
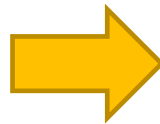
```
if (x1 < y1) z1 = x1;  
else z2 = y1;  
z3 =  $\Phi$ (z1, z2);  
w1 = z3;
```

Program leképzése

- Az egész program: függvényekre darabolva
- Függvény: vezérlési folyam gráf (control-flow graph, CFG)
 - > csúcs: alapblokk
 - > él: ugrás egyik blokkból a másikba
 - > két kitüntetett blokk: belépési blokk (entry block) és kilépési blokk (exit block)
- Alapblokk (Basic Block):
 - > maximális hosszúságú utasítássorozat (SSA utasítások)
 - > atomi: ha a blokk egy utasítása végrehajtódik, akkor az összes többi is
 - > címkével kezdődik, nem tartalmaz egyéb címkét
 - > címkére ugrással vagy feltételes címkére ugrással fejeződik be, nem tartalmaz egyéb ugrást (függvényhívás nem számít ugrásnak)

Vezérlési folyamat gráf (control-flow graph) példa

```
int foo(bool b)
{
    int x = 3;
    int y = x*2;
    int z;
    if (b) z = x*x;
    else z = y*y;
    x = z+1;
    return x;
}
```

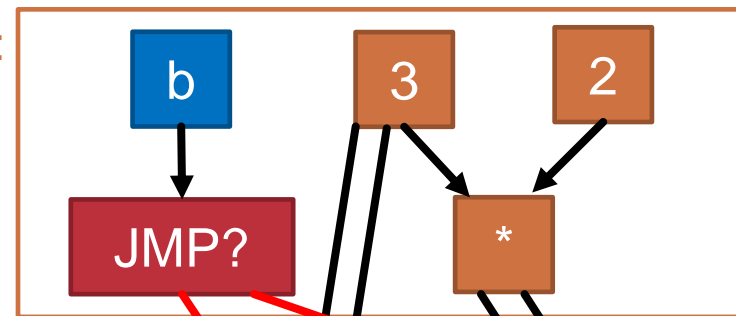


Vezérlési folyam- és adatfolyam gráf

```
int foo(bool b)
{
    int x = 3;
    int y = x*2;
    int z;
    if (b) z = x*x;
    else z = y*y;
    x = z+1;
    return x;
}
```



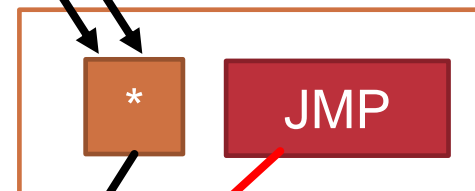
BB1 (entry):



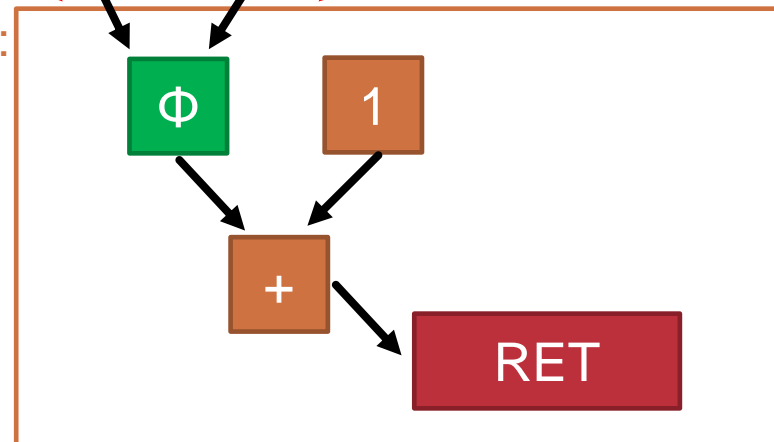
BB2:



BB3:



BB4 (exit):



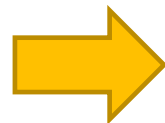
Utasítások leképezése

- Aritmetikai és logikai műveletek: egy-az-egyben leképezve
 - > egyes programnyelveknél túlcsordulás-ellenőrzés is kell (drága!)
- Elágazások, ciklusok, kivételek: újabb alapblokkok és vezérlési élek
- Memória-hozzáférés: címaritmetika (bázis cím + relatív cím)
 - > írás, olvasás
 - > indexelés ellenőrzés is kellhet (drága!)
- Típuskonverzió: megfelelő utasításokra leképezve
- Függvényhívás:
 - > állapotmentés, veremfoglalás, vezérlésátadás, verem-felszabadítás, állapot-visszaállítás

Aritmetikai és logikai műveletek

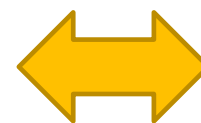
Programkód:

```
bool foo(int y)
{
    return 3 * y + 4 > 7;
}
```

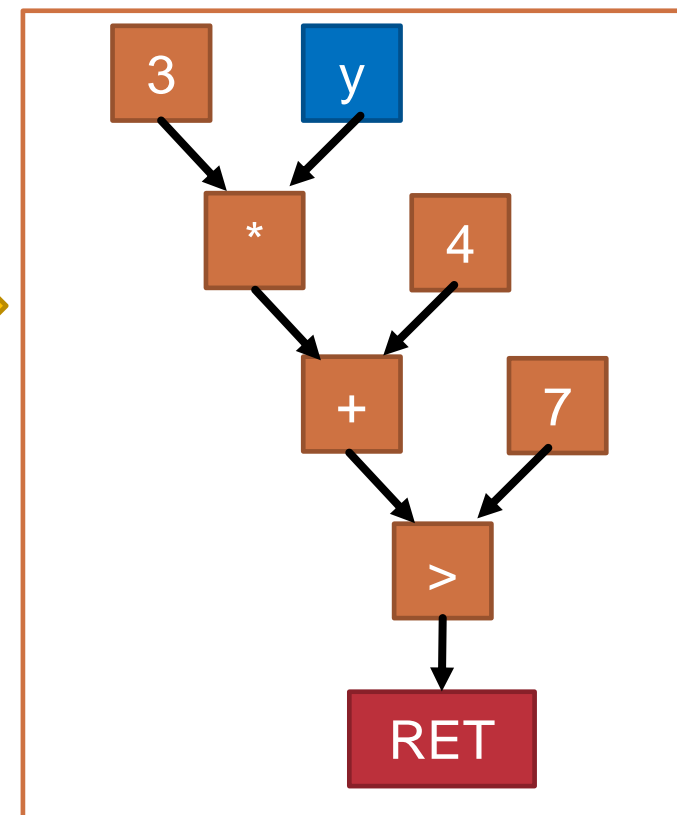


SSA:

```
t1: 3 * y
t2: t1 + 4
t3: t2 > 7
RET t3
```



Gráf:

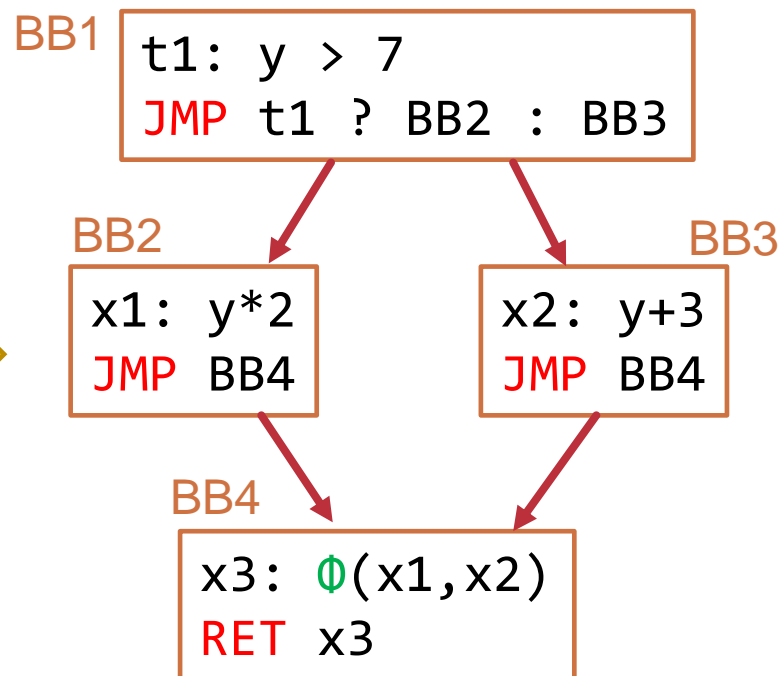


Elágazás: if

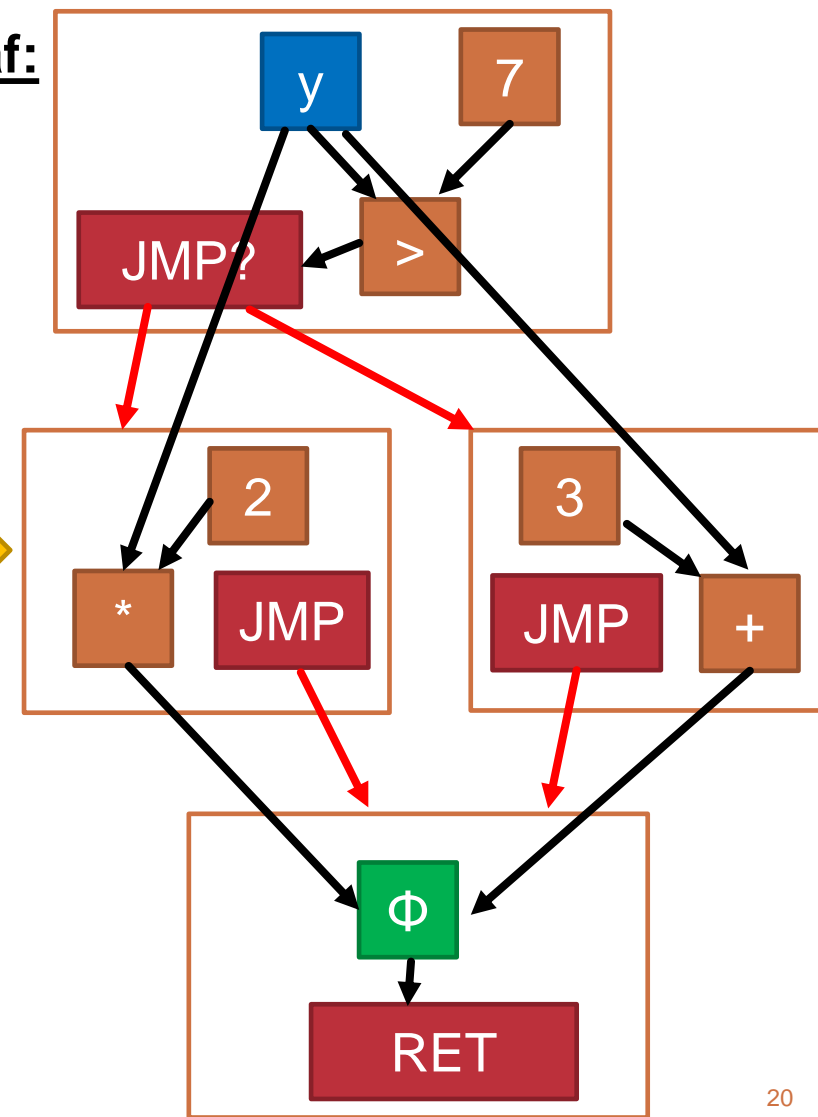
Programkód:

```
int foo(int y)
{
    int x;
    if (y > 7)
    {
        x = y*2;
    }
    else
    {
        x = y+3;
    }
    return x;
}
```

SSA:



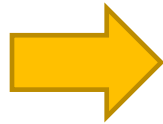
Gráf:



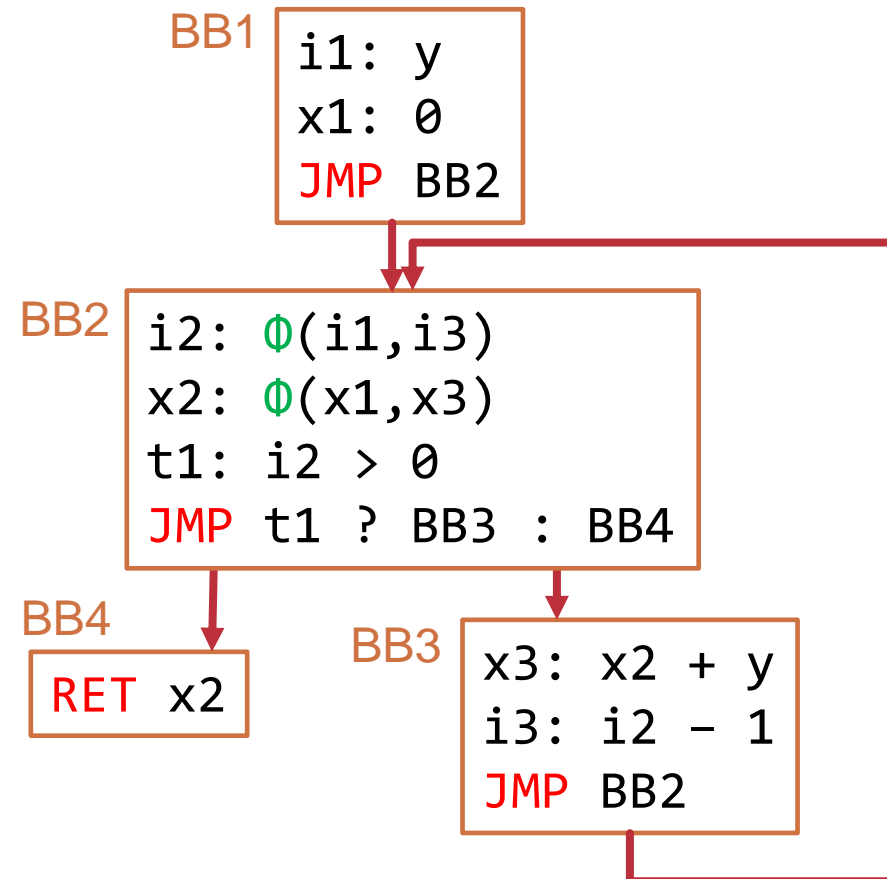
Ciklus: while

Programkód:

```
int foo(int y)
{
    int i = y;
    int x = 0;
    while (i > 0)
    {
        x += y;
        --i;
    }
    return x;
}
```



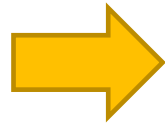
SSA:



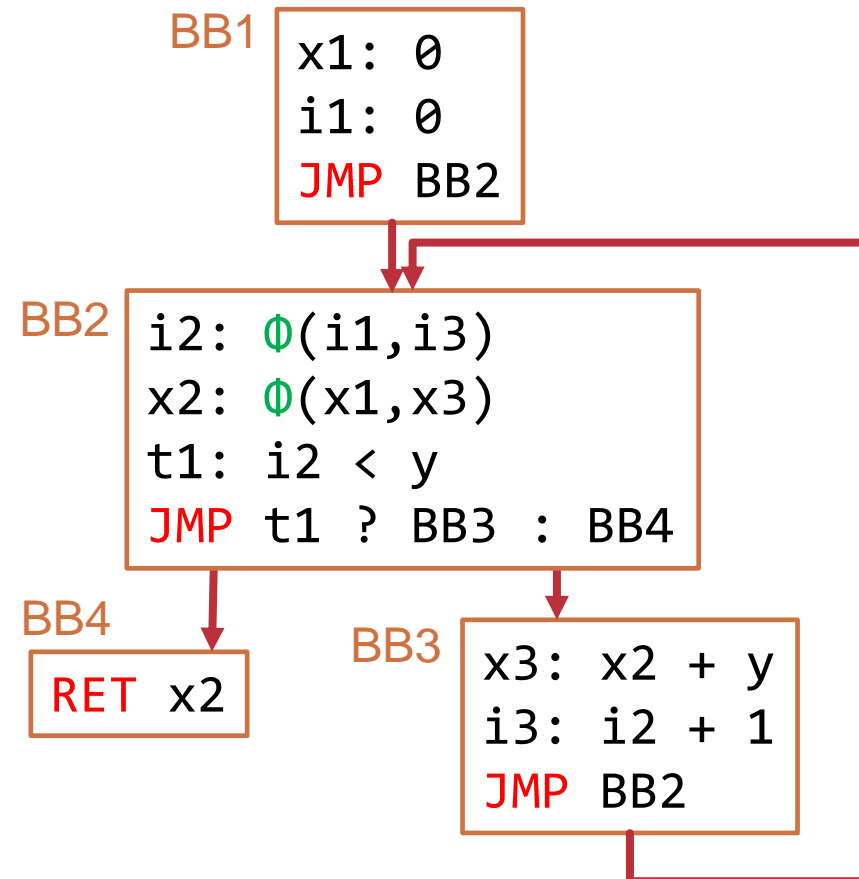
Ciklus: for

Programkód:

```
int foo(int y)
{
    int x = 0;
    for (int i = 0; i < y; ++i)
    {
        x += y;
    }
    return x;
}
```



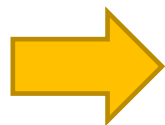
SSA:



Memória: olvasás (**LD** = load), írás (**ST** = store) Címszámítás: bázis + offset

Programkód:

```
class P { int Q; int R; }  
  
void foo(P p, int[] a)  
{  
    int q = p.Q;  
    int r = p.R;  
    int s = a[2];  
    p.R = s;  
    a[2] = q;  
}
```

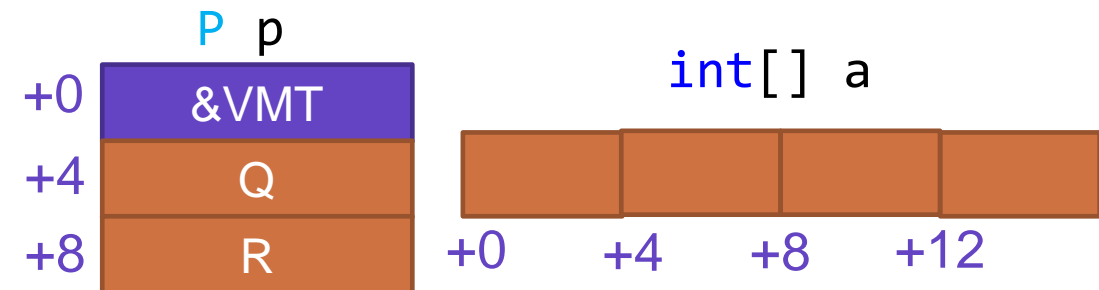
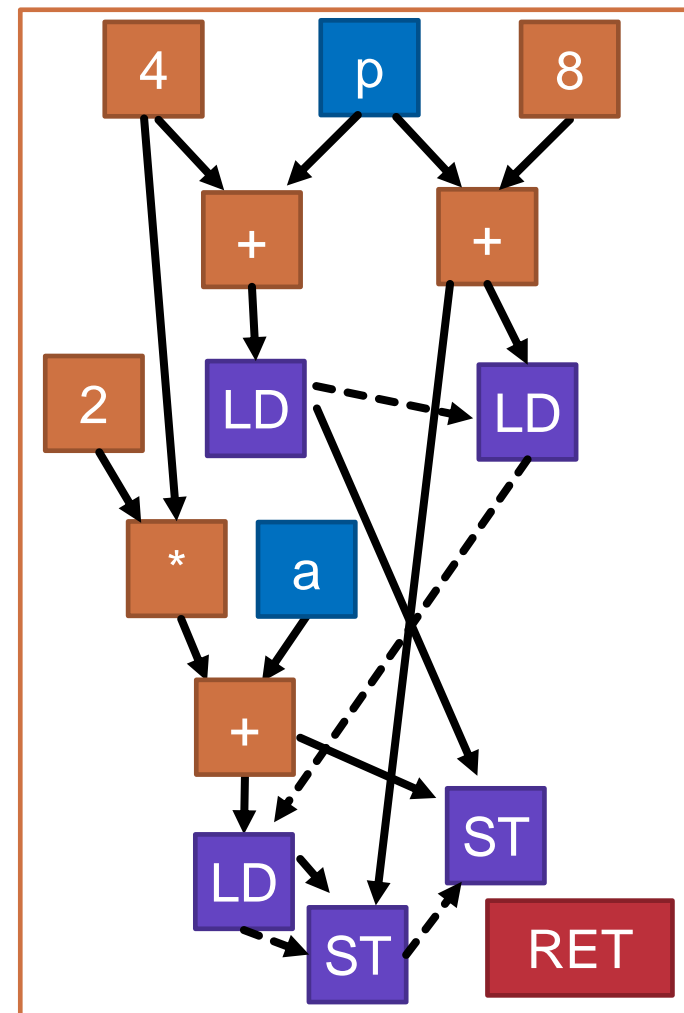


SSA:

```
t1: p + 4  
q1: LD t1  
t2: p + 8  
r1: LD t2  
t3: 4 * 2  
t4: a + t3  
s1: LD t4  
ST t2: s1  
ST t4: q1  
RET
```



Gráf:



Függvényhívás

Programkód:

```
bool foo(int y)
{
    return bar(y+1)-3;
}
```

```
int bar(int x)
{
    return x*2;
}
```

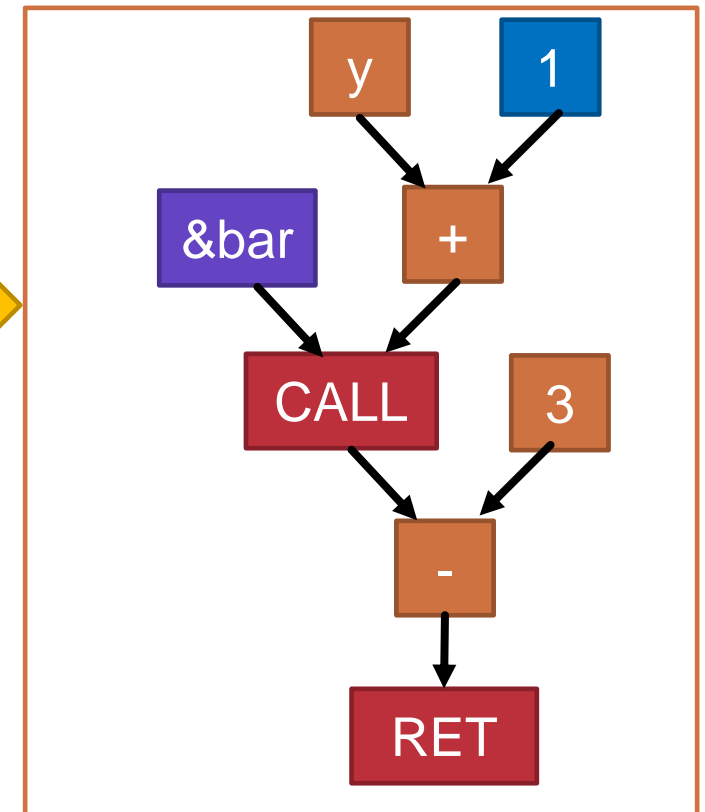
SSA:

→

```
t1: y + 1
t2: CALL bar(t1)
t3: t2 - 3
RET t3
```

↔

Gráf:



Függvényhívás

- 1. Állapot (regiszterek) mentése
- 2. Hívott függvénynek hely foglalása a veremben (Stack Frame)
 - > visszatérési cím, visszatérési érték, paraméterek, lokális változók
- 3. Visszatérési cím beírása a verembe (kivételkezelés esetén 2 visszatérési cím van!)
- 4. Argumentumok beírása a verembe
- 5. Programszámláló (Program Counter) átállítása a hívott függvény címére
- 6. Hívott függvény törzsének végrehajtása
- 7. Programszámláló (Program Counter) átállítása a visszatérési címre
- 8. Visszatérési érték kinyerése a veremből
- 9. Lefoglalt veremterület (Stack Frame) felszabadítása
- 10. Állapot (regiszterek) visszaállítása

SSA számítása

- Alapblokkok és a vezérlési gráf létrehozása
- Blokkonként sorban: blokkokon belül értékek sorszámozása
- Φ -függvények csak akkor számíthatók, ha minden megelőző blokk kész
 - > amíg nem számíthatók: ideiglenesen megjegyezzük egy Φ' függvénnyel
- Φ -függvény bekerülhet egy megelőző blokkba is!
- Előfordulhat, hogy a Φ' függvényből nem lesz Φ -függvény

Φ -függvény szabályok

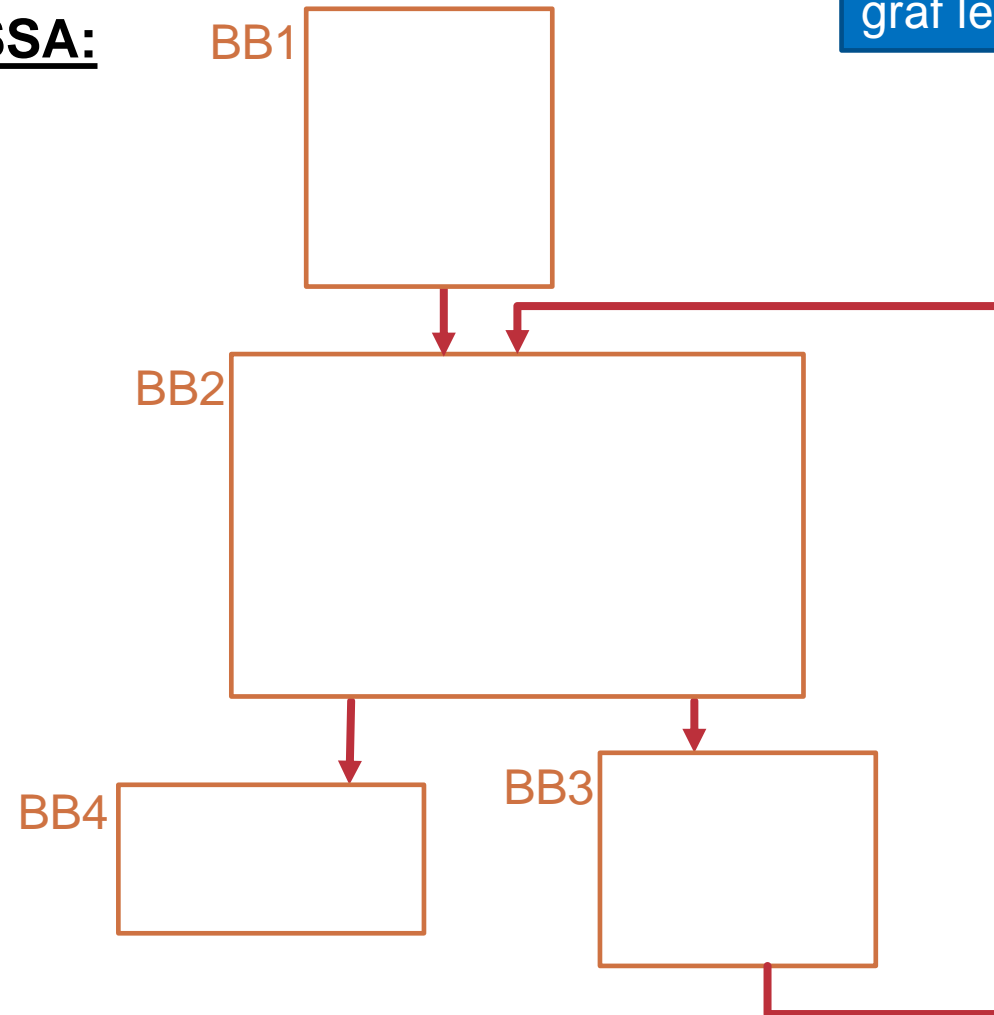
- A Φ -függvények mindig egy olyan blokk elején szerepelnek, amelynek több megelőző blokkja is van
- Egy Φ -függvénynek pontosan annyi operandusa van, ahány megelőző alapelőblokkból érkezhet a Φ -függvény blokkjába a vezérlés
- A Φ -függvény értéke annak az operandusnak az értéke, amelyik a végrehajtáskor ténylegesen megelőző alapelőblokkhoz tartozik
- Egy blokkon belül minden Φ -függvényt szimultán egyszerre kell kiértékelni

SSA számítás példa (1/6)

Programkód:

```
int foo(int y)
{
    int x = 0;
    int z = 2;
    for (int i = 0; i < y; ++i)
    {
        x += y;
    }
    return x + z;
}
```

SSA:



Alapblokkok és vezérlési gráf létrehozása.

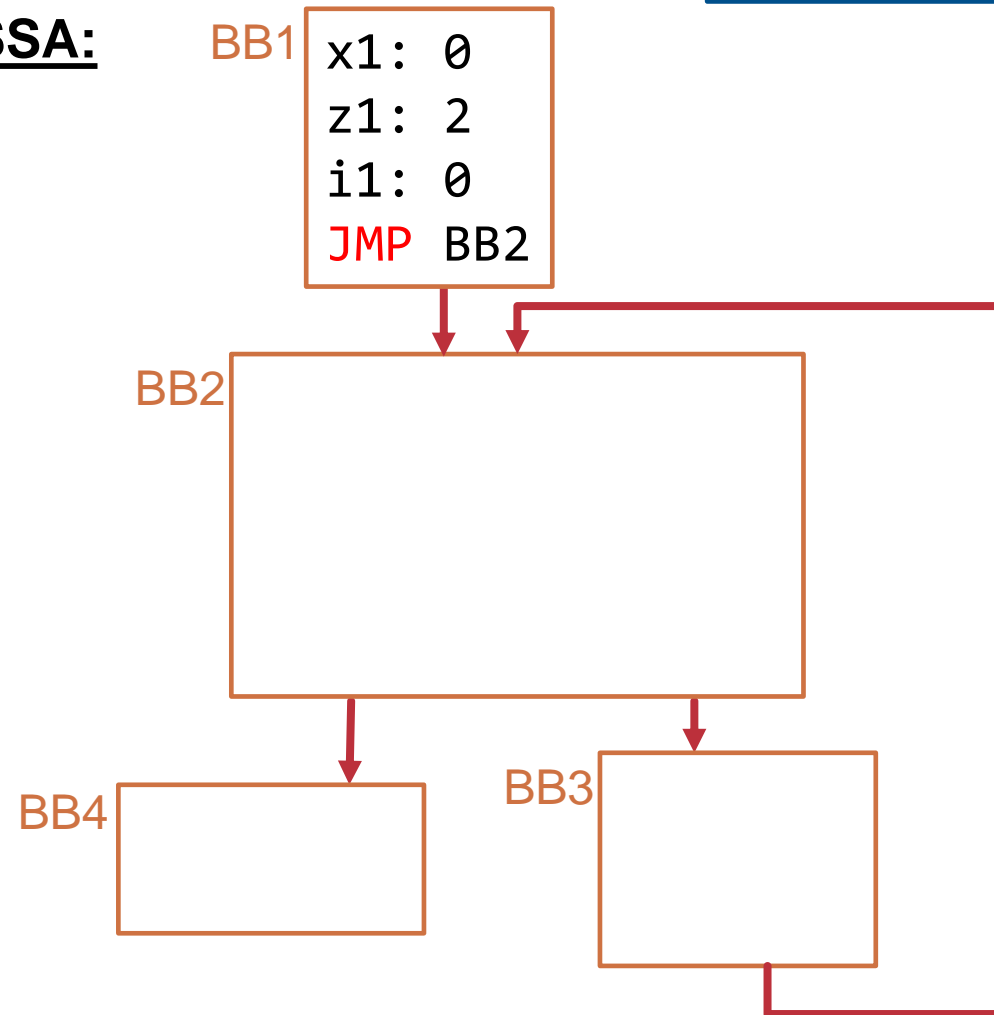
SSA számítás példa (2/6)

BB1-ben az értékek számozása.

Programkód:

```
int foo(int y)
{
    int x = 0;
    int z = 2;
    for (int i = 0; i < y; ++i)
    {
        x += y;
    }
    return x + z;
}
```

SSA:

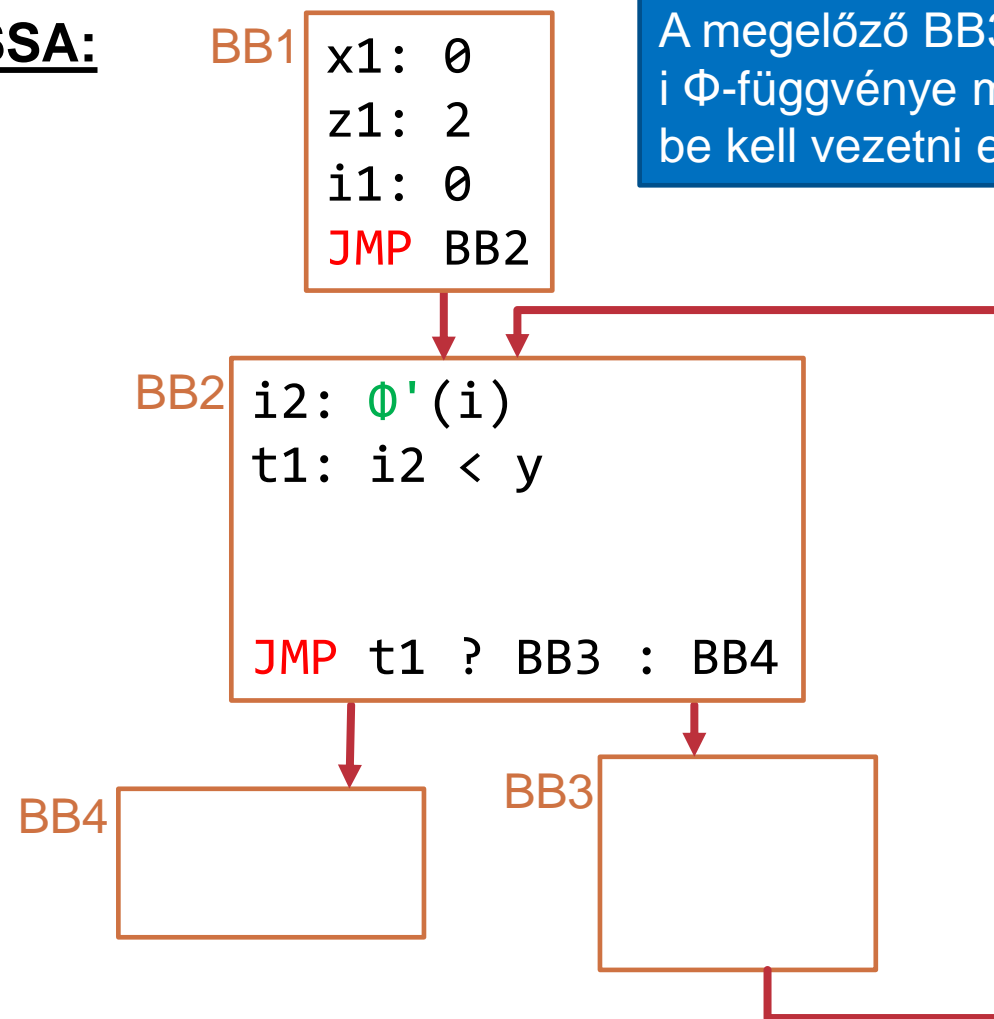


SSA számítás példa (3/6)

Programkód:

```
int foo(int y)
{
    int x = 0;
    int z = 2;
    for (int i = 0; i < y; ++i)
    {
        x += y;
    }
    return x + z;
}
```

SSA:



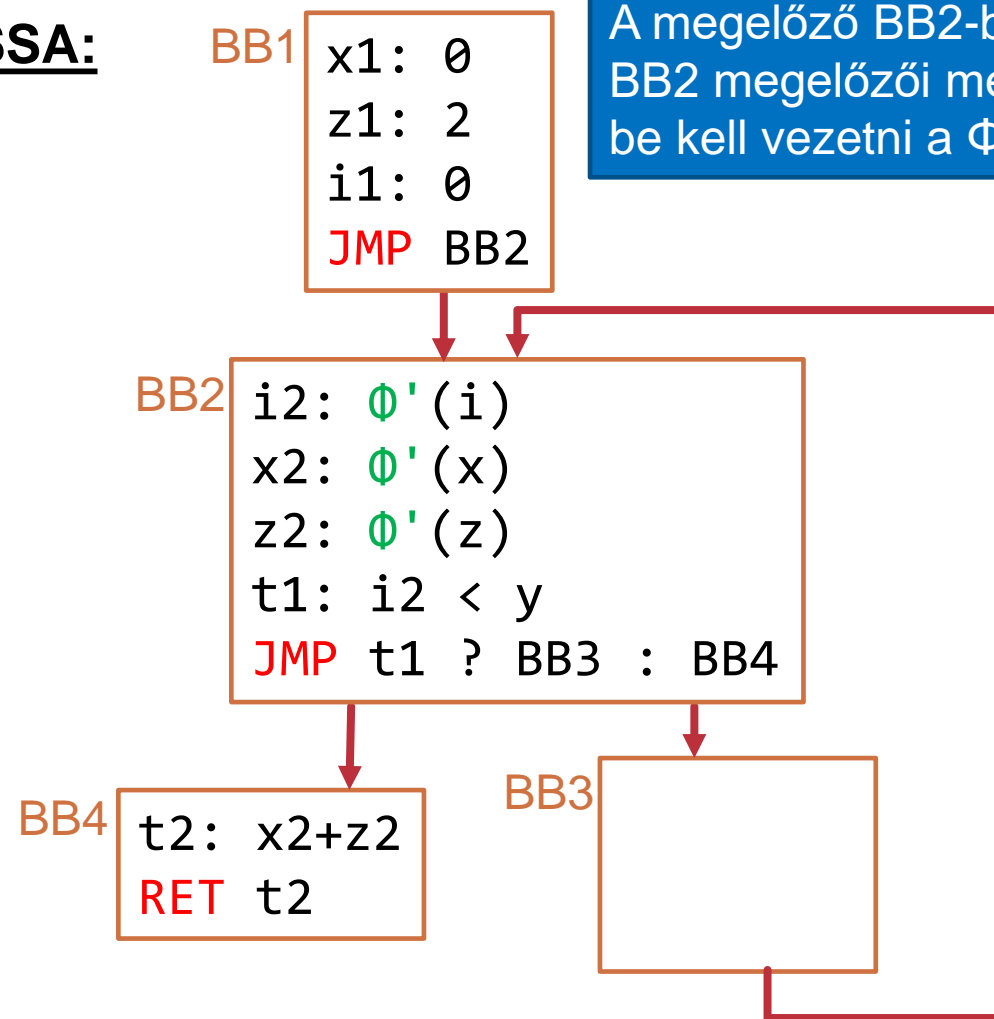
BB2-ben az értékek számozása.
A megelőző BB3 még nincs kész, így
i Φ -függvénye még nem számítható:
be kell vezetni egy $\Phi'(i)$ függvényt.

SSA számítás példa (4/6)

Programkód:

```
int foo(int y)
{
    int x = 0;
    int z = 2;
    for (int i = 0; i < y; ++i)
    {
        x += y;
    }
    return x + z;
}
```

SSA:



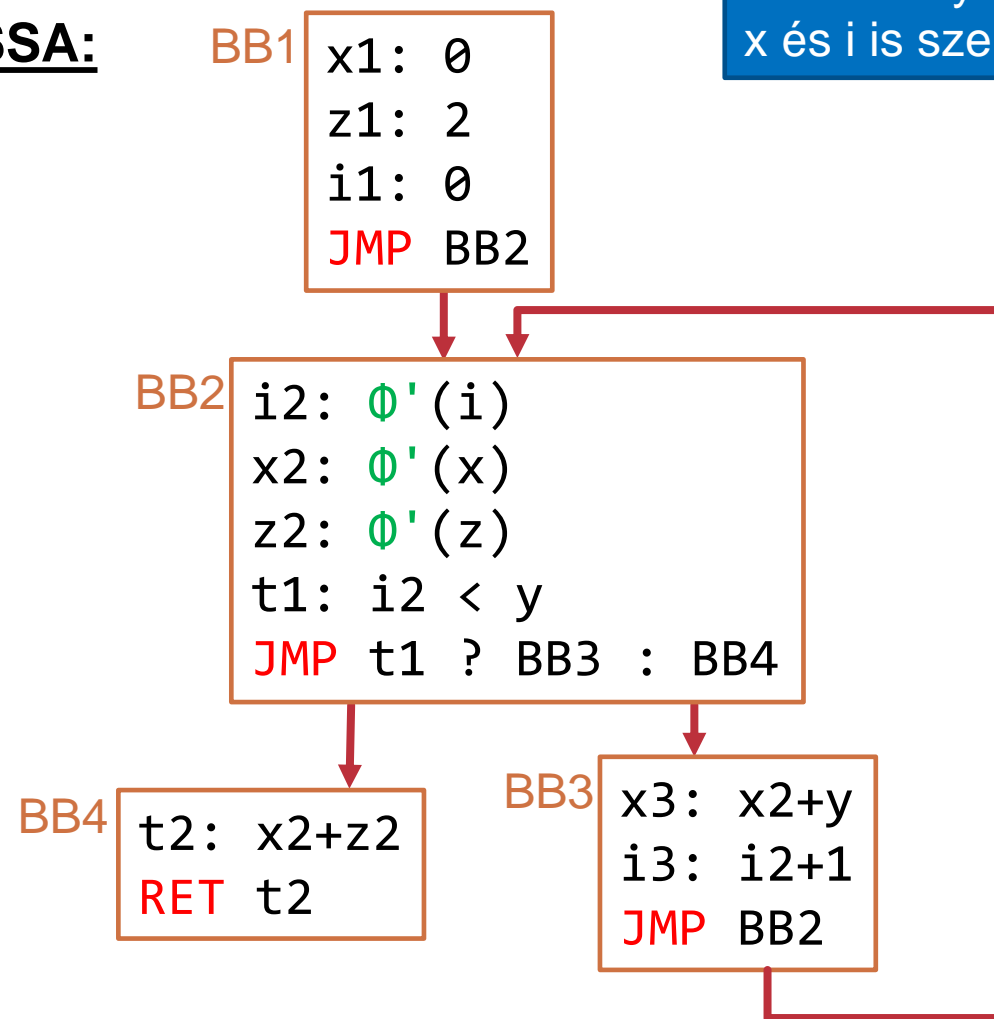
BB4-ben x és z is használva van.
A megelőző BB2-ben nem szerepelnek.
BB2 megelőzői még nincsenek kész:
be kell vezetni a $\Phi'(x)$ és $\Phi'(z)$ értékeket.

SSA számítás példa (5/6)

Programkód:

```
int foo(int y)
{
    int x = 0;
    int z = 2;
    for (int i = 0; i < y; ++i)
    {
        x += y;
    }
    return x + z;
}
```

SSA:



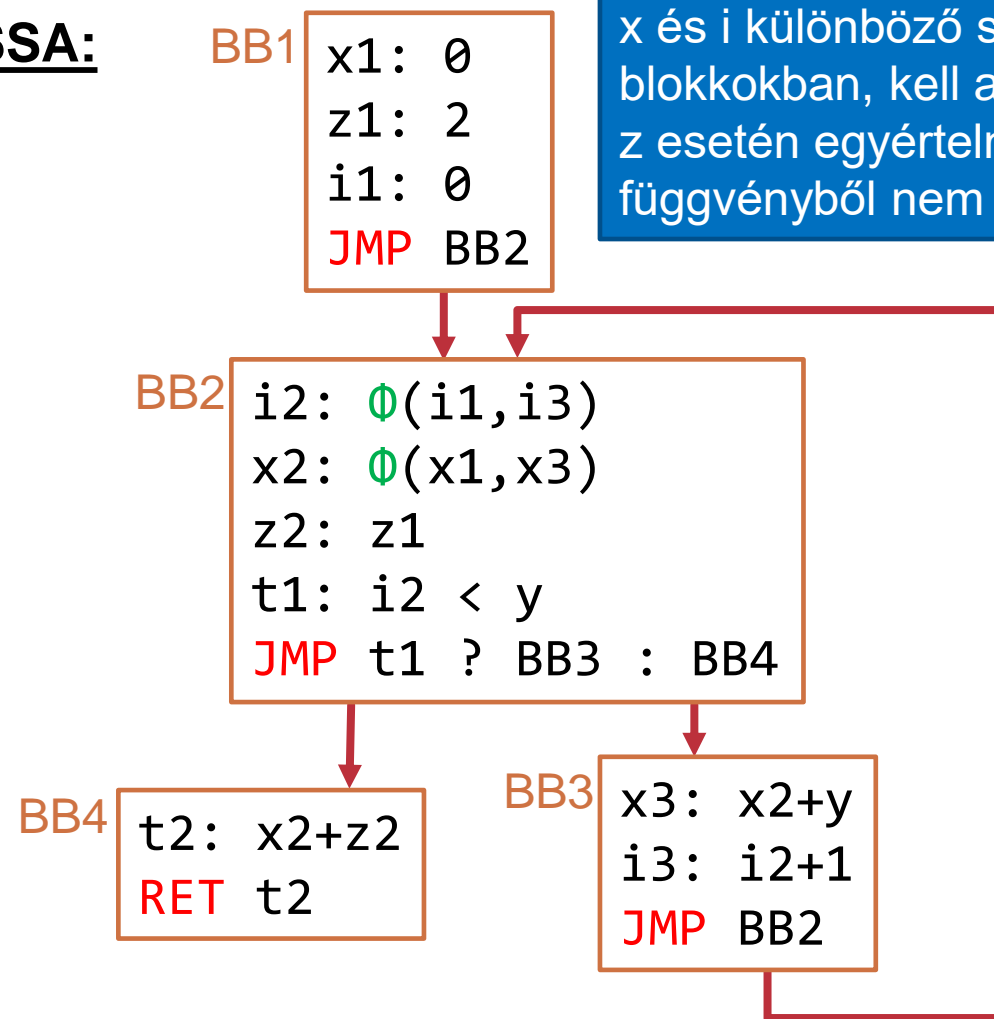
BB3 könnyen számítható, mert x és i is szerepel BB2-ben.

SSA számítás példa (6/6)

Programkód:

```
int foo(int y)
{
    int x = 0;
    int z = 2;
    for (int i = 0; i < y; ++i)
    {
        x += y;
    }
    return x + z;
}
```

SSA:

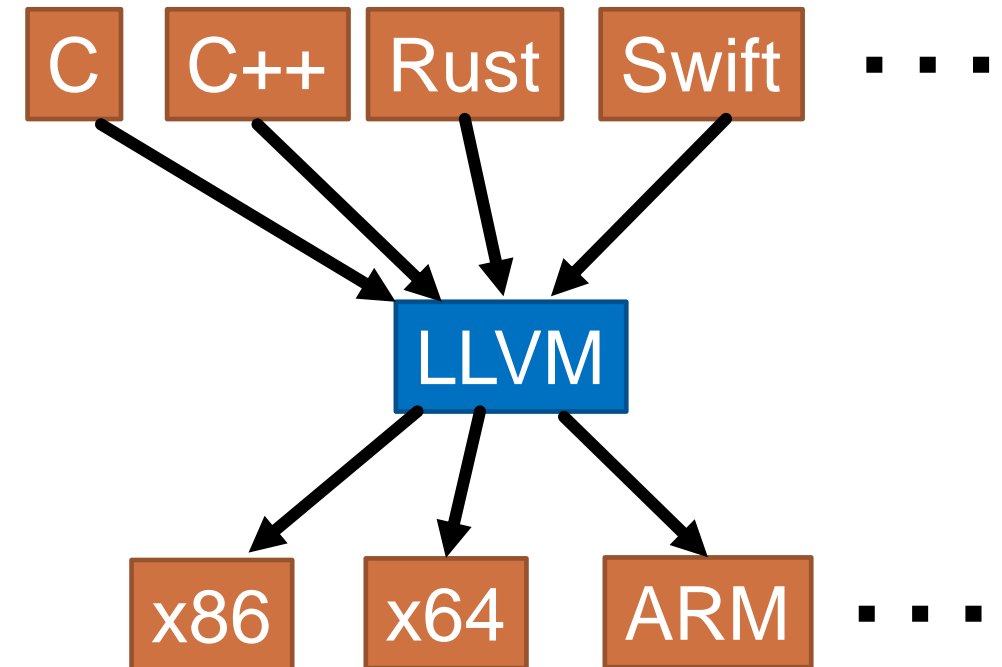


BB2-ben Φ' átalakítása:

x és i különböző sorszámú a megelőző blokkokban, kell a Φ függvény.
z esetén egyértelmű a sorszám, a Φ' függvényből nem lesz Φ függvény.

LLVM

- Fordító eszközrendszer
 - > front end (transzformáció SSA formára)
 - > optimalizáció
 - > back end (gépi kód generálása)
- Programnyelv-független köztes nyelv (Intermediate Representation – IR)
 - > típusrendszer, utasítások, kifejezések
- Saját programnyelvünkhöz is felhasználhatjuk!
- Alternatíva LLVM helyett:
 - > generálhatunk C, Java, C#, stb. kódot, és lefordíthatjuk a hagyományos fordítóval



A mai előadás: Transzformáció, Optimalizálás

I. Transzformáció

II. Típusok leképezése

III. Utasítások leképezése: SSA

IV. Optimalizáció

V. Optimalizációs technikák



Optimalizálás

- Cél: hatékonyabb végrehajtás a viselkedés megváltoztatása nélkül
 - > kisebb kódméret, gyorsabb futás, kevesebb memória, alacsonyabb fogyasztás, stb.
- Nincs optimális program, csak optimalizált!
 - > különben lenne algoritmusunk a megállási problémára, de az eldönthetetlen
- Néhány példa optimalizálási technikákra:
 - > Konstans kifejezések kiértékelése, Halott kód eliminálása, Operátorok egyszerűsítése, Ciklusinvariáns kódok átmozgatása, Részleges redundanciák eliminálása, Kódáthelyezés, Indexhatárok ellenőrzésének eliminálása, Függvények inline beépítése, Lefutási ágak egyszerűsítése, Ciklusok kibontása/feldarabolása , Regiszterkiosztás, Jobbrekurzió feloldása ciklussá

Optimalizálás sorrendje

- Kérdés: Melyik optimalizációkat alkalmazzuk? Milyen sorrendben?
 - > nincs rá pontos válasz
- Néhány optimalizáció hasonlít egymásra, vagy részhalmaza egymásnak
- Numerikus programoknál:
 - > operátoregyszerűsítés: általában legalább 2-szeres gyorsulást eredményez
 - > cache-optimalizálás: általában 2-5-szörös gyorsulást eredményez
- Egyéb optimalizálások:
 - > az elsőnek választott kb. 15%-os gyorsulást eredményez
 - > minden további kevesebb, mint 5%-ot hoz
- Forrás: http://www.info.uni-karlsruhe.de/lehre/2007WS/uebau1/folien/10-SSA_v2.pdf

Optimalizáció szintjei

- Lokális
 - > Egyetlen alapblokkon belül, izolálva a többitől
- Globális (intraprocedurális)
 - > Alapblokkok sorozatán (control-flow gráf) belül
- Interprocedurális
 - > A teljes program kódját figyelembe veszi
 - > Függvényhatárokon átívelő
 - > Ritkán támogatják a fordítók

Optimalizálás SSA segítségével

- Sok optimalizálás könnyen végrehajtható az SSA formán
- Két fő transzformáció: normalizálás és optimalizálás
- Normalizálás: különbözőképpen leírt kifejezések összehasonlíthatóvá tétele
 - > elősegíti az optimalizációt
 - > algebrai azonosságok felhasználása: kommutativitás, asszociativitás, disztributivitás
 - > alkalmazhatóságát korlátozzák a kivételek vagy az előírt kiértékelési sorrend (pl. Java)
- Optimalizálás: program hatékonyabbá tétele

Adatfolyam-elemzés SSA segítségével

- Egyes optimalizációk átrendezhetik az utasítások sorrendjét
- Adatfolyam-elemzés: adatfüggőségek felderítése
 - > változó definíció-használat, értékadás-olvasás műveletekből adódó függőségek
 - > fontos a kódgenerálási fázisban az utasítások megfelelő sorrendezéséhez
- Adatfolyam-elemzés SSA nélkül is végezhető, de SSA-val gyakran egyszerűbb
- Szemantikai elemzéshez is hasznos
 - > nem inicializált változók felismerése
 - > nem használt változók kijelzése
 - > élő (később potenciálisan használt) változók felderítése
 - > annak felismerése, hogy egy függvény nem minden ágon tér vissza
 - > stb.

A mai előadás: Transzformáció, Optimalizálás

I. Transzformáció

II. Típusok leképezése

III. Utasítások leképezése: SSA

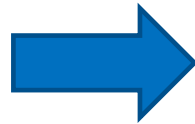
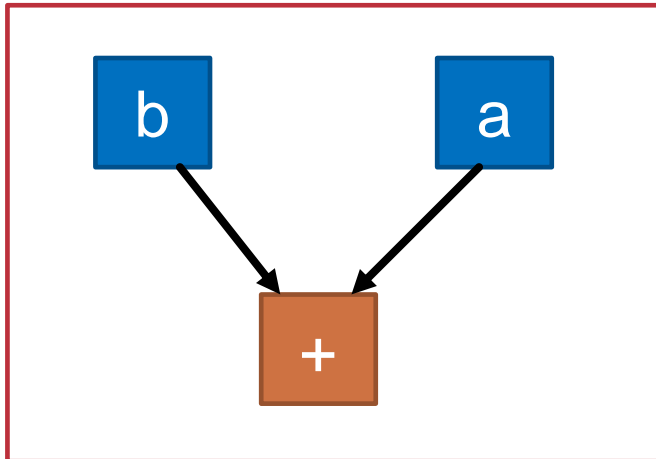
IV. Optimalizáció

V. Optimalizációs technikák

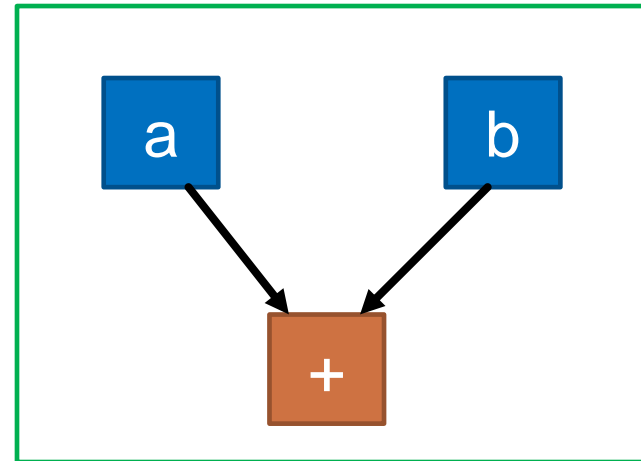


Normalizálás: kommutativitás kihasználása

t1: b+a



t1: a+b

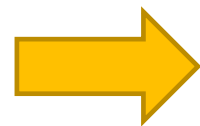


Így könnyebb felismerni a
közös részkifejezéseket

Optimalizálás: közös részkifejezések eliminálása

$x = b+a;$
 $y = (a+b)*2;$

SSA



$x1: b+a$
 $t2: a+b$
 $y1: t2*2$

Norm.

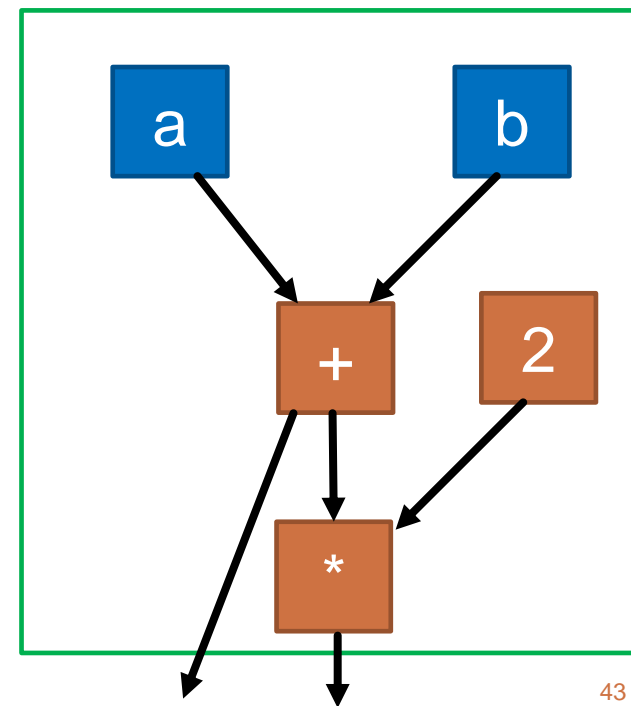
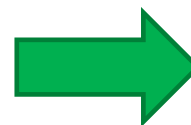
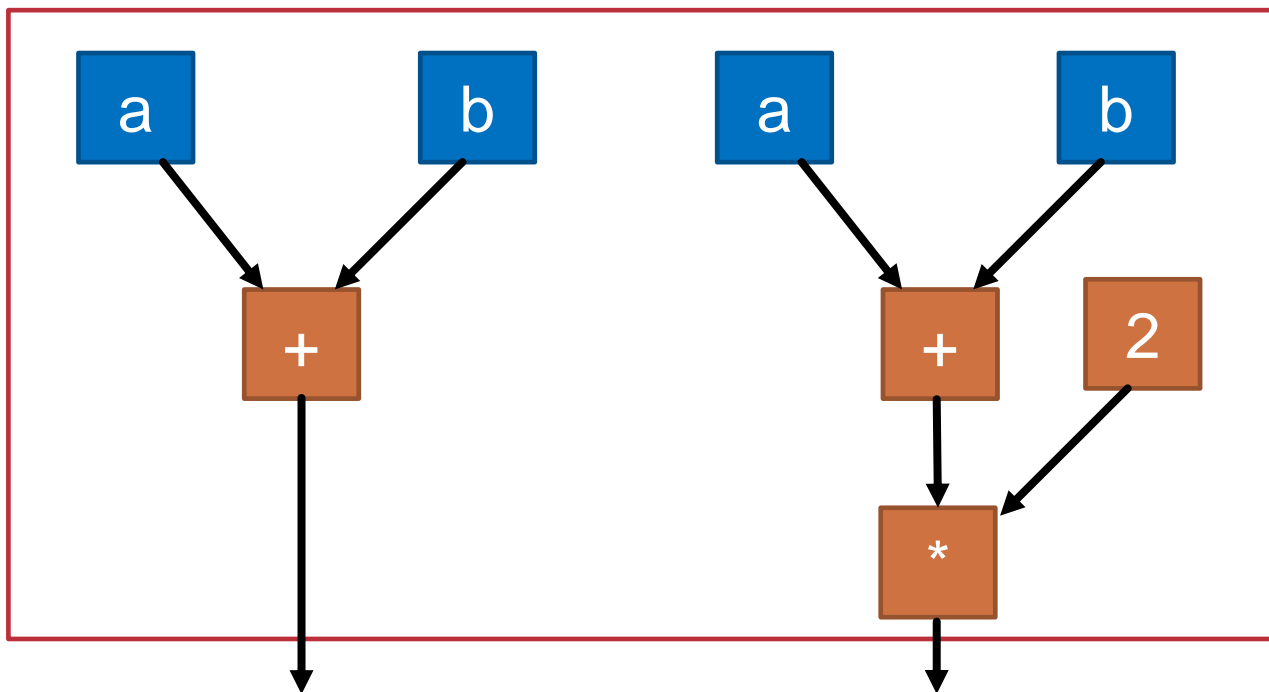


$x1: a+b$
 $t2: a+b$
 $y1: t2*2$

Opt.

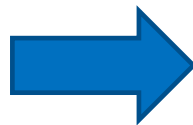
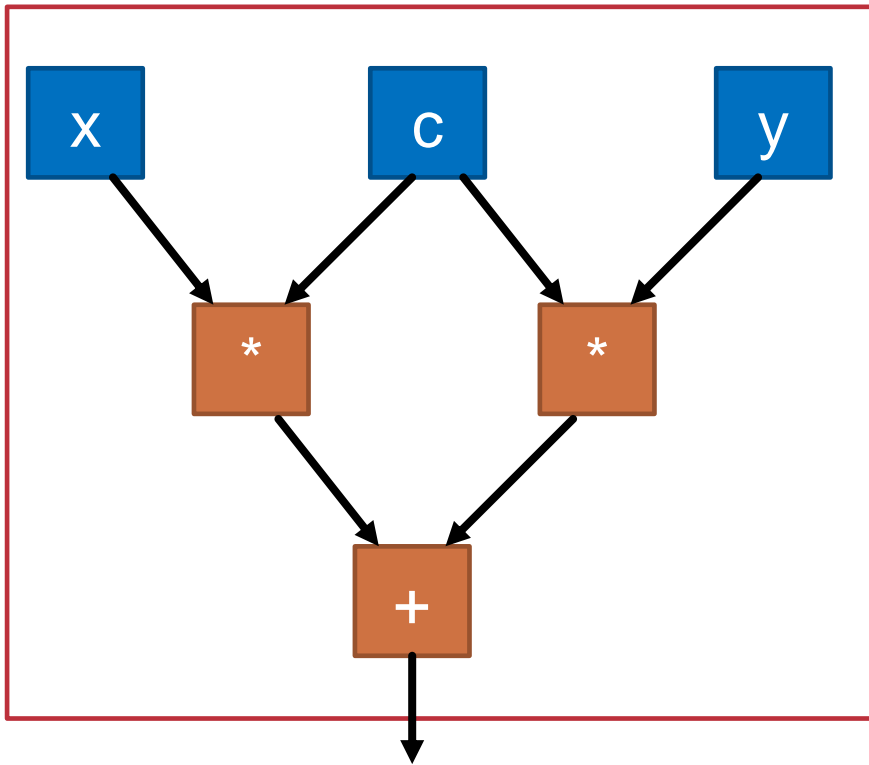


$x1: a+b$
 $y1: x1*2$

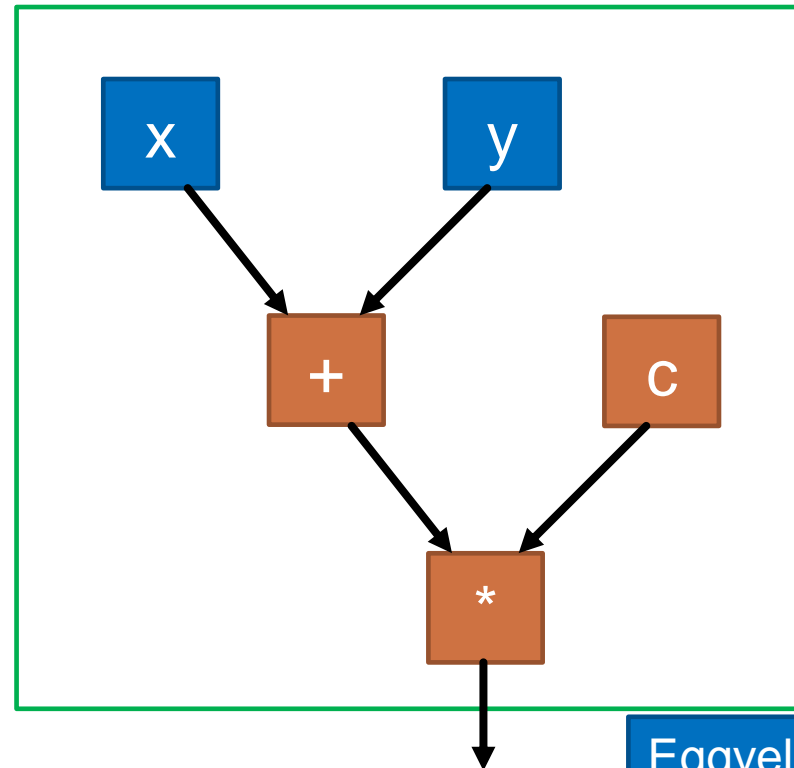


Normalizálás: disztributivitás kihasználása

t1: $x * c$
t2: $y * c$
t3: $t1 + t2$



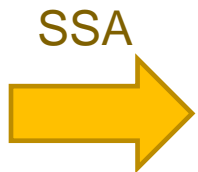
t1: $x + y$
t2: $t1 * c$



Eggyel kevesebb szorzás!

Optimalizálás: konstans kifejezések kiértékelése

$x = 2+4;$
 $y = x*3;$



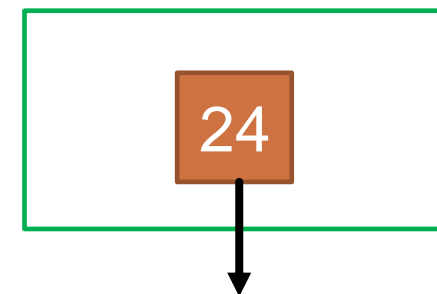
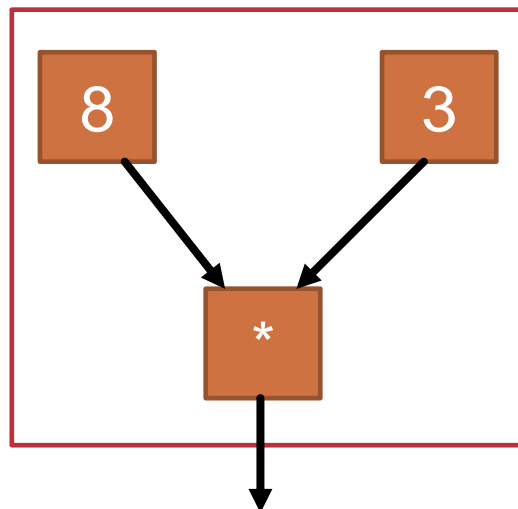
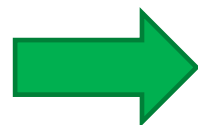
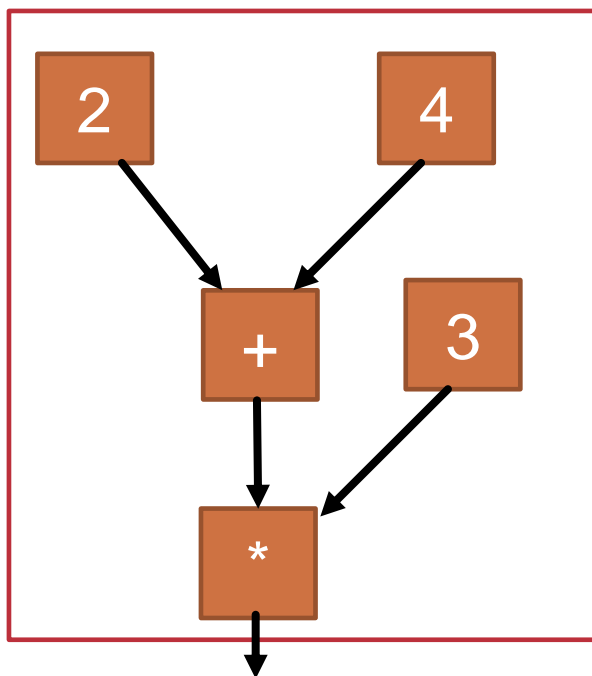
$x1: 2+4$
 $y1: x1*3$



$x1: 8$
 $y1: x1*3$



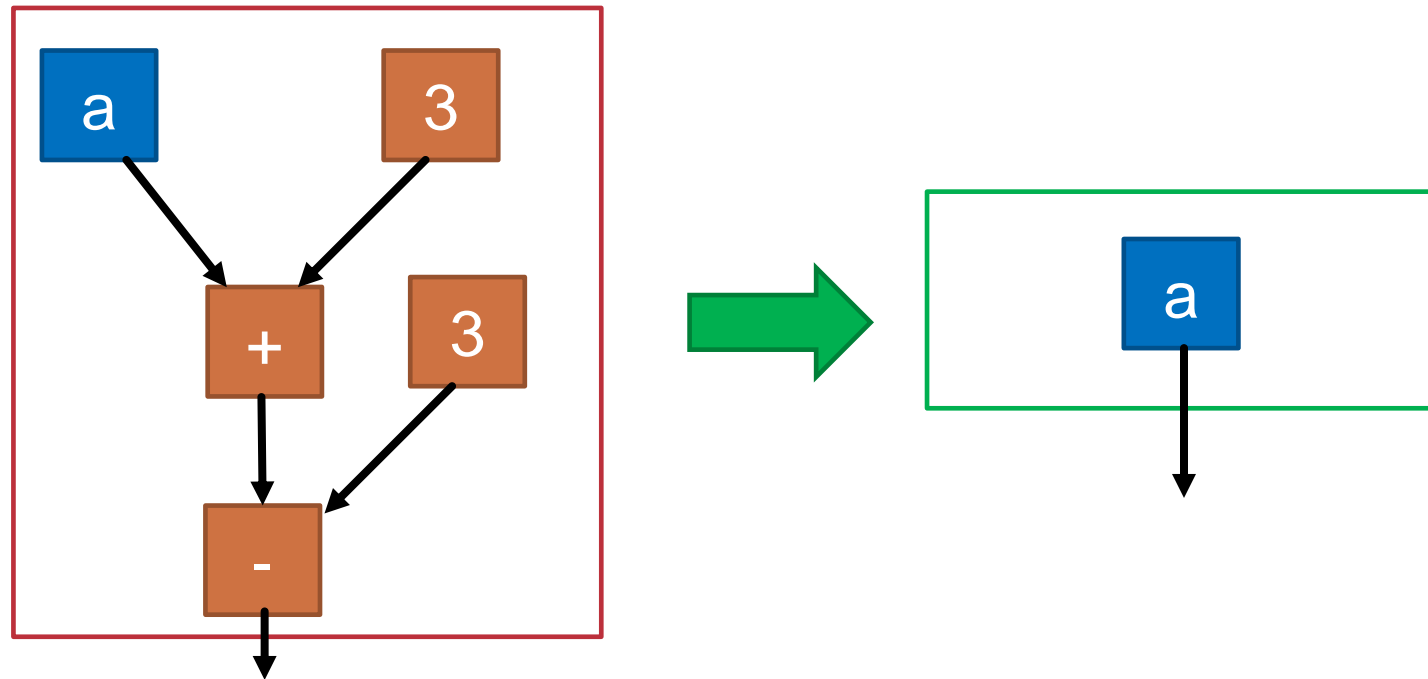
$y1: 24$



Legtöbbször címszámításkor fordul elő.

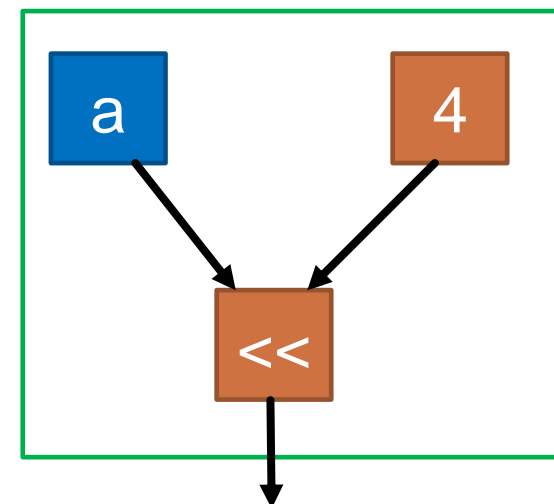
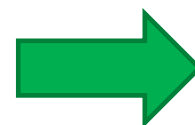
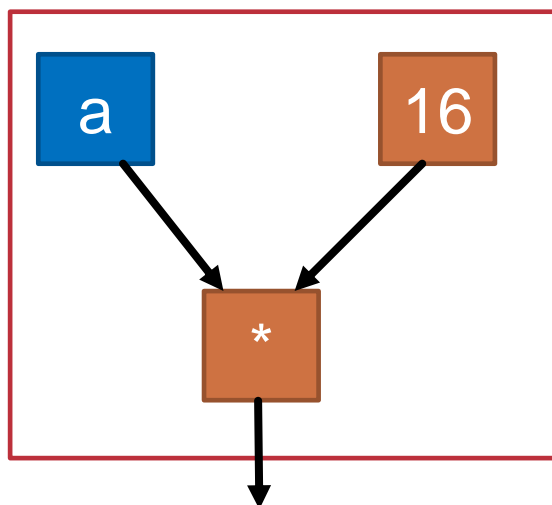
Optimalizálás: inverz operátor törlése

$x = a+3;$
 $y = x-3;$ $\xrightarrow{\text{SSA}}$ $x1: a+3$
 $y1: x1-3$ $\xrightarrow{\text{Opt.}}$ $y1: a$



Optimalizálás: operátoregyszerűsítés

$x = a * 16;$ $\xrightarrow{\text{SSA}}$ $x1: a * 16$ $\xrightarrow{\text{Opt.}}$ $x1: a \ll 4$

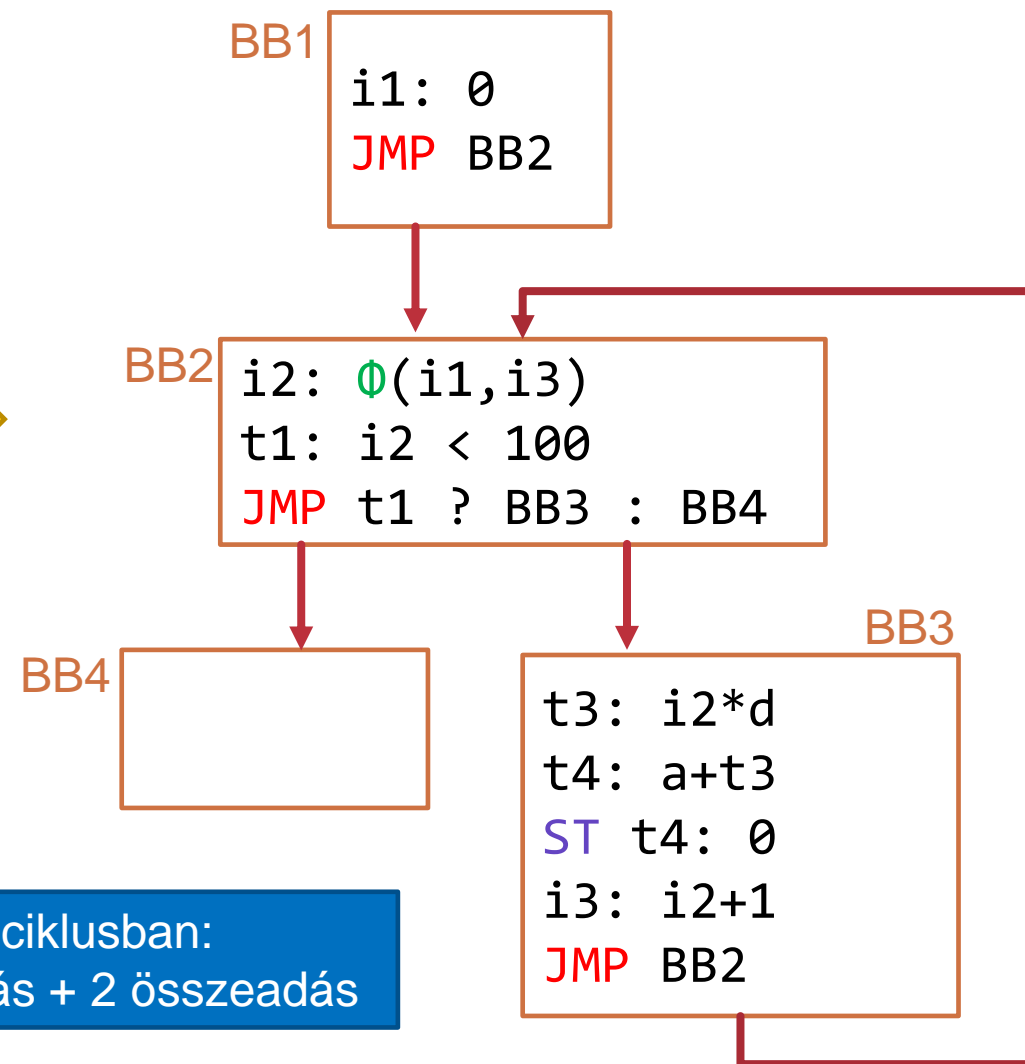


2-hatvánnyal szorzás/osztás helyett shift-elés.

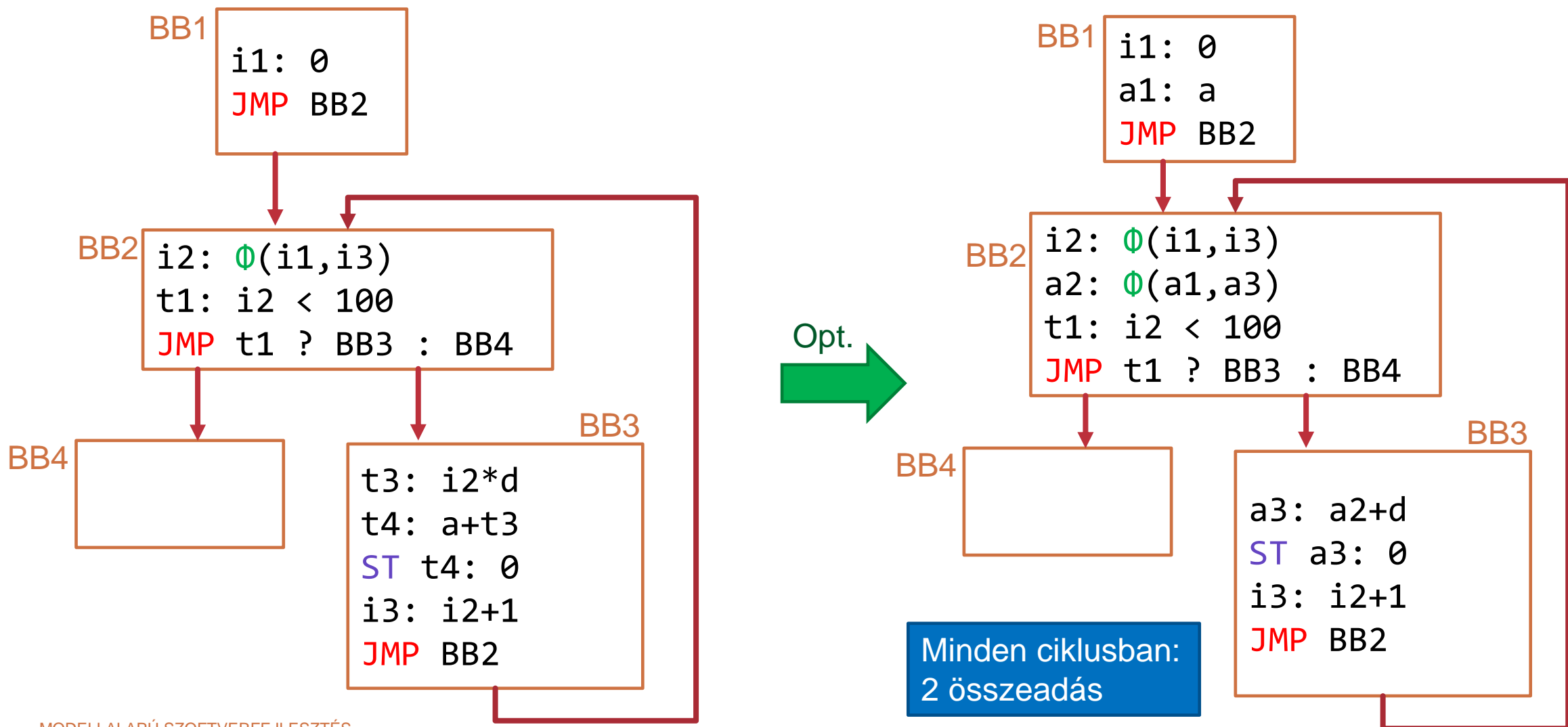
Optimalizálás: ciklusok egyszerűsítése (1/3)

```
for (int i = 0; i < 100; ++i)
{
    a[i] = 0;
}
```

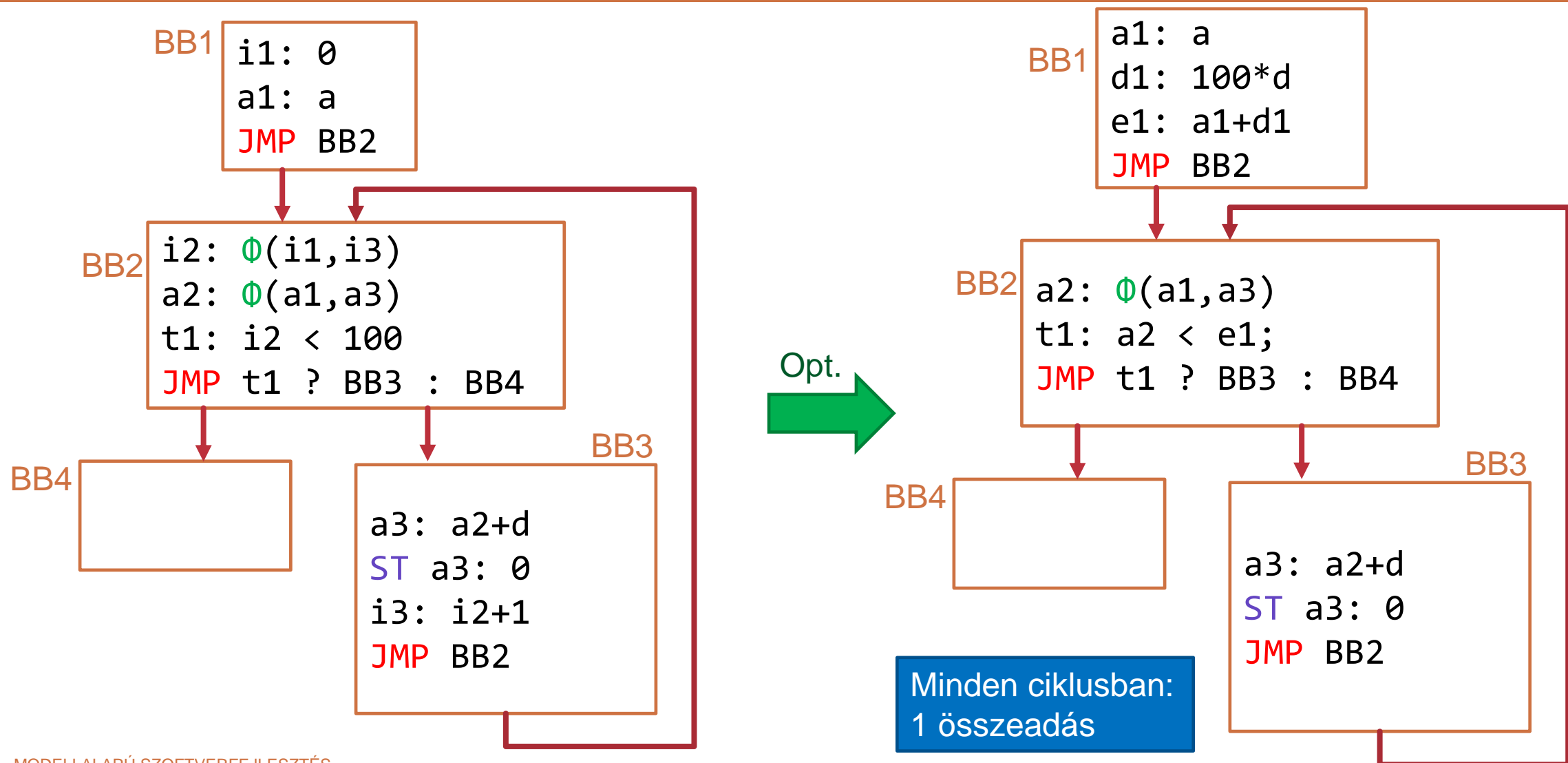
SSA



Optimalizálás: ciklusok egyszerűsítése (2/3)



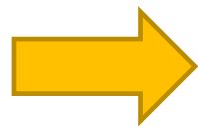
Optimalizálás: ciklusok egyszerűsítése (3/3)



Optimalizálás: store-load

`a[1] = x;`
`y = a[1];`

SSA

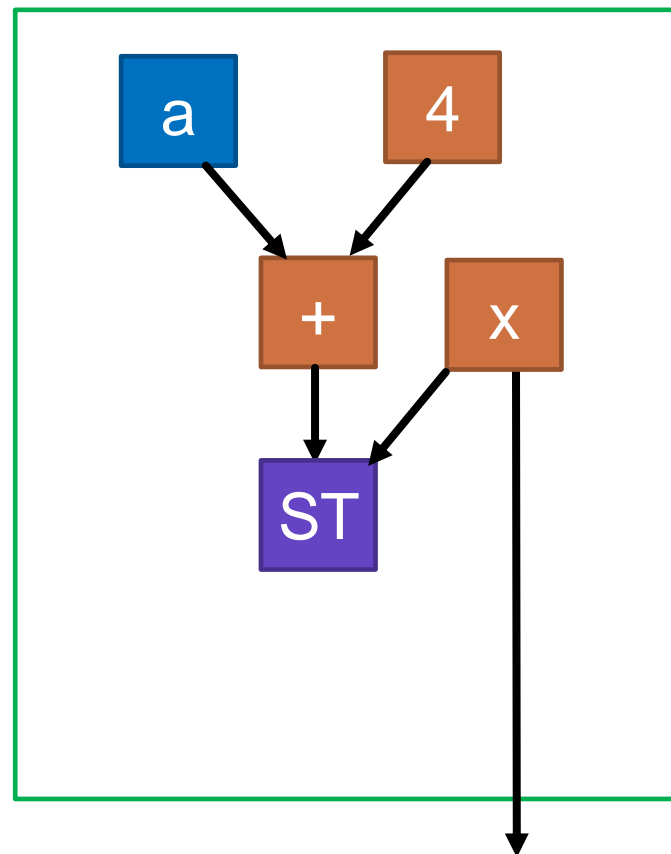
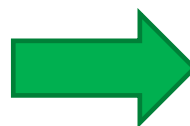
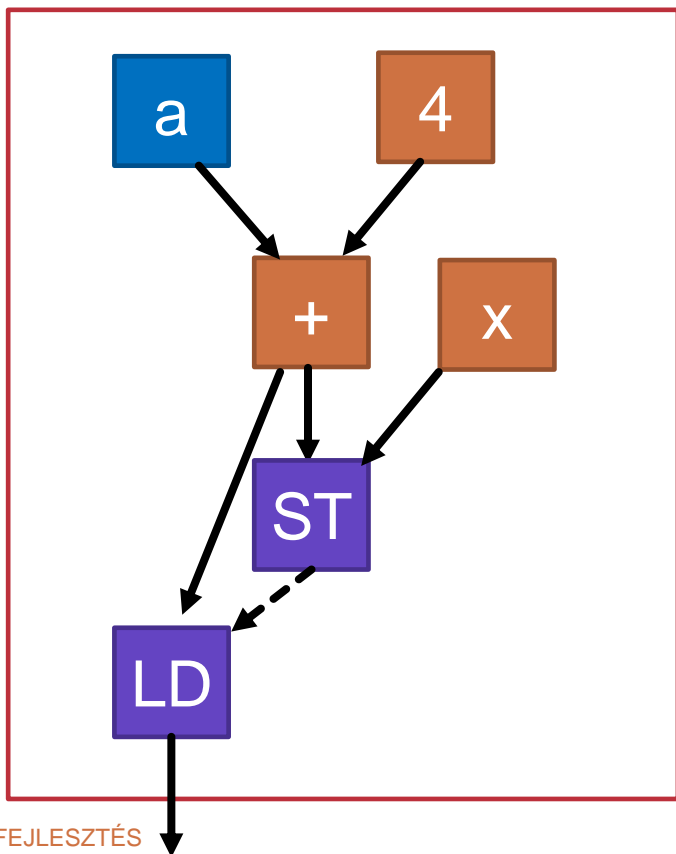


`t1: a+4`
`ST t1: x`
`y1: LD t1`

Opt.



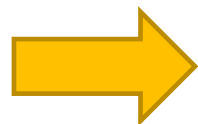
`t1: a+4`
`ST t1: x`
`y1: x`



Optimalizálás: load-load

x = a[1];
y = a[1];

SSA

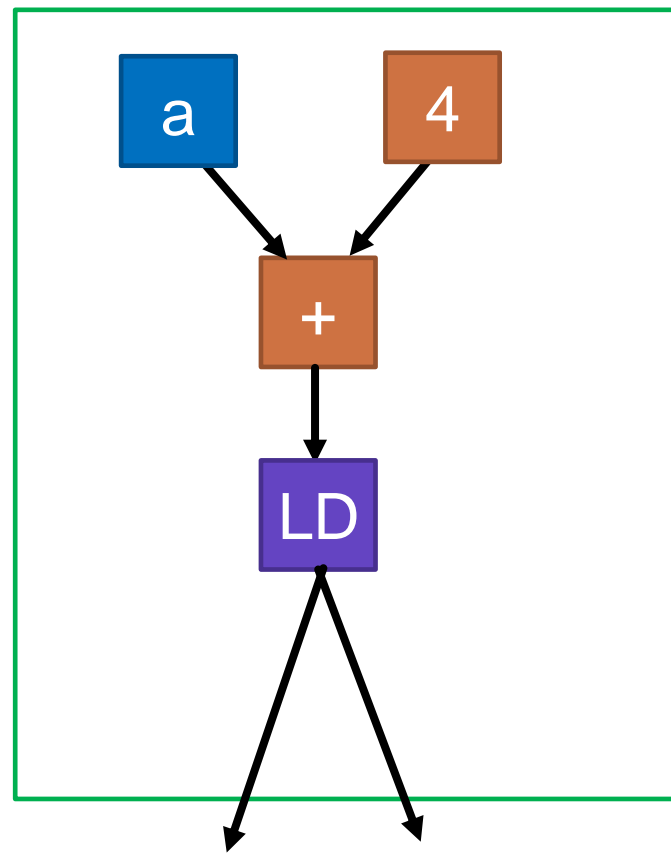
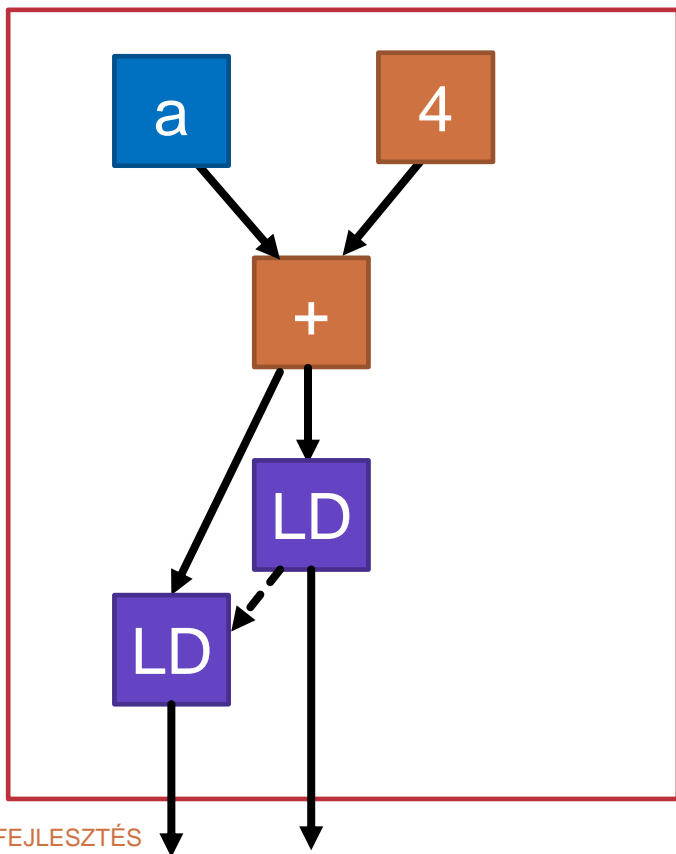


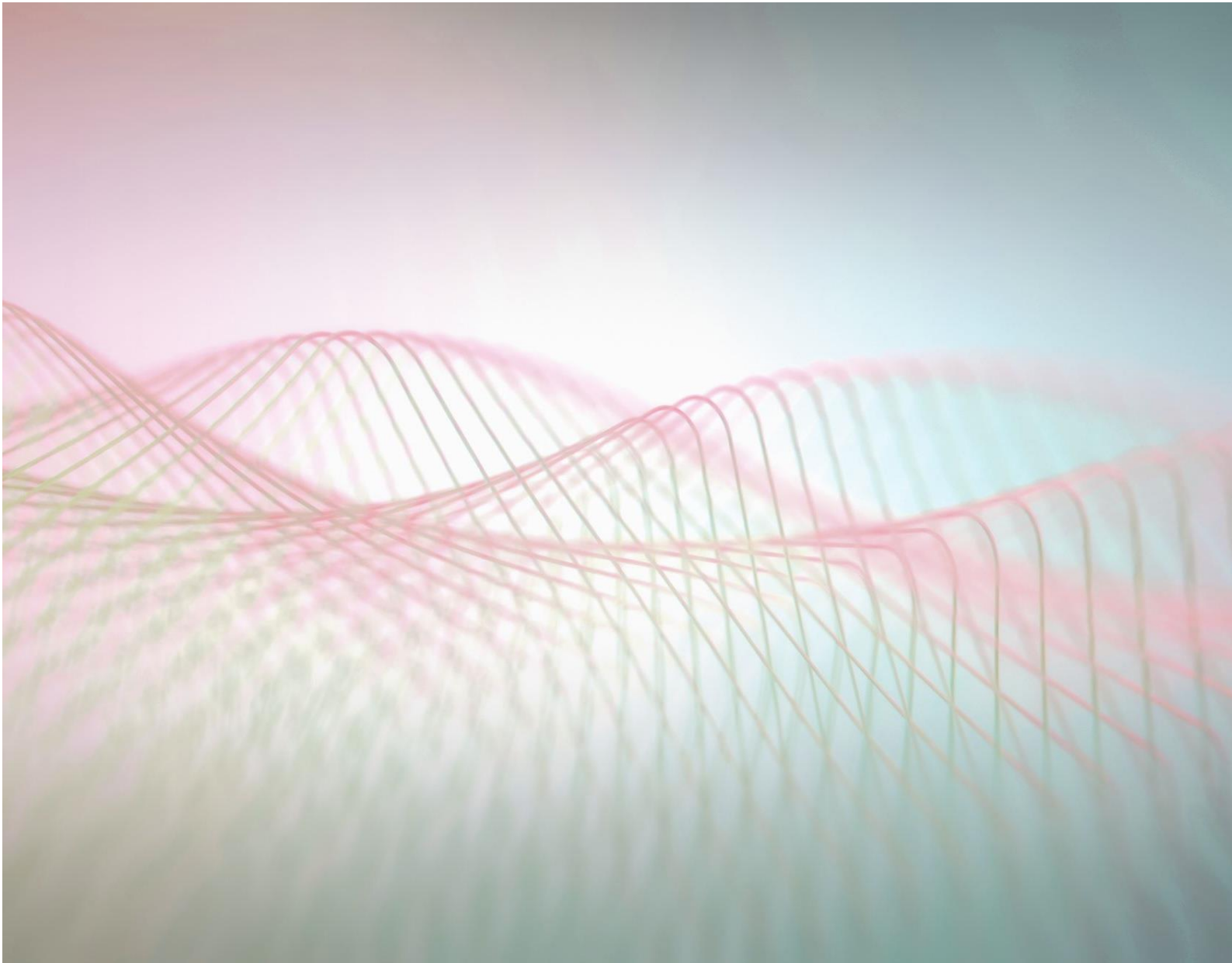
t1: a+4
x1: LD t1
y1: LD t1

Opt.



t1: a+4
x1: LD t1
y1: x1





Harmadik gyakorlat

A következő rész tartalmából...

- Témakör: Szöveges szakterületi nyelvek IDE támogatása
- Eclipse környezetben – Java nyelv
- Xtext – szöveges szakterületi nyelv leírása, feldolgozása és IDE támogatása
- Xcore – metamodellezés (egyben AST leírás)
- Xtend – sablon alapú kódgenerálás
- IDE funkciók:
 - syntax highlighting, validation, error markers, content assist, hyperlinking, outline, automatic code formatting, automatic build / code generation



Köszönöm a figyelmet!