Transzformáció, SSA, Optimalizálás

Simon Balázs

BME IIT, 2010.

forrás: http://www.info.uni-karlsruhe.de/lehre/2007WS/uebau1/

Tartalom

- SSA
- SSA példa
- Optimalizálás

Static Single Assignment

SSA

- Static Single Assignment
- Cél:
  - Függvény szintű optimalizációk elősegítése
  - Definiálás-felhasználás explicit jelzése
- Definíció: egy program akkor van SSA formában reprezentálva, ha minden változónak pontosan egy helyen adunk értéket.
  - A program ettől kezdve függvényt jelent
  - A változó ezentúl alias nélküli lokális változó
  - Vigyázat: az SSA nem azt jelenti, hogy minden érték csak egyszer kerül kiszámításra!

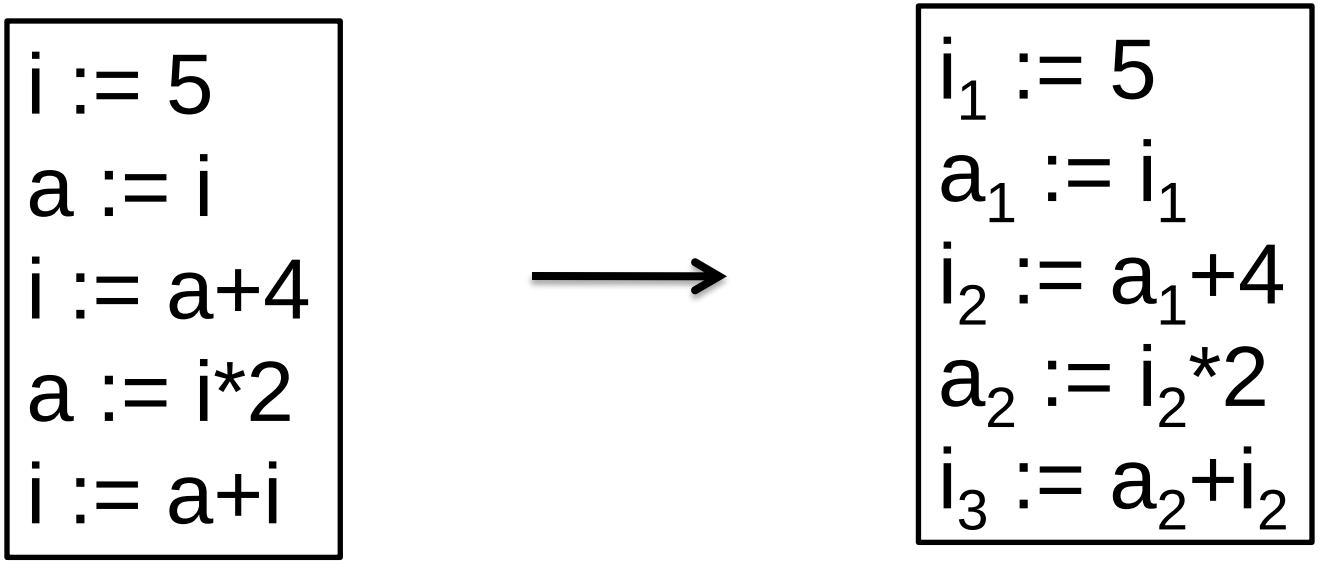
SSA indokoltsága

- A gépi utasítások memóriacellákon és regisztereken operálnak, függetlenül attól, hogy a forráskódban mi volt a változó neve.
- Ha egy operáció kétszer ugyanazokon az operandusokon értelmezett, akkor az egyik operáció elhagyható. (értékszámozás)
- Hogy két operandus ugyanolyan értékű-e, az az előzményekből kikövetkeztethető. De: figyelni kell a lefutási ágakra!

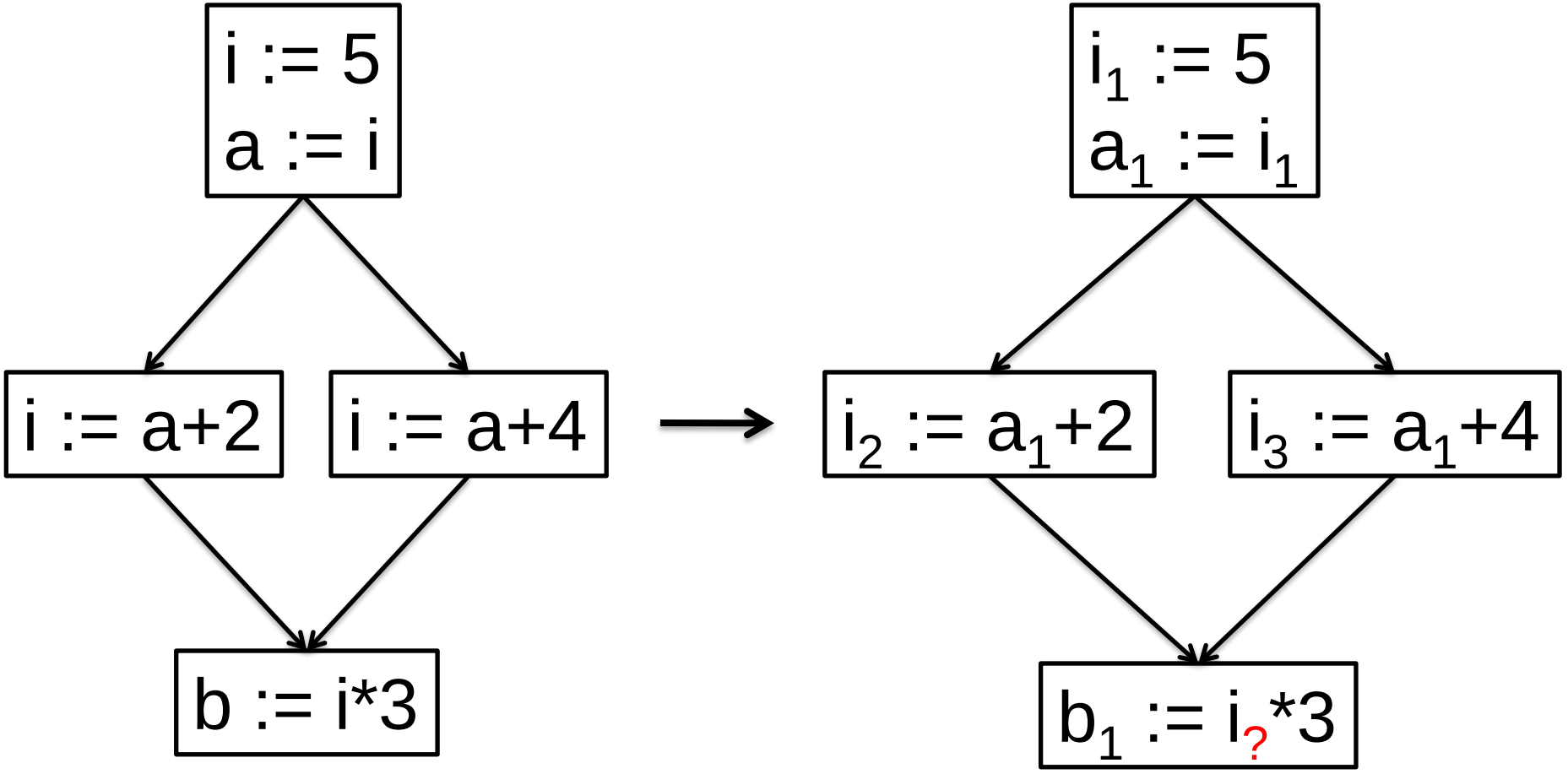
SSA konstrukció

- Értékadás helyett: változódefiníció
  - A változó csak itt kap értéket!
- A programnyelvben *a* nevű változó SSA formában számozódik: a_1, a_2, ...
- Az a minden felhasználási helyén a megfelelő a_i-t kell behelyettesíteni
- Probléma: elágazások
- Megoldás: Φ-függvény bevezetése, amely mindig a megfelelő lefutásiághoz tartozó értéket választja ki

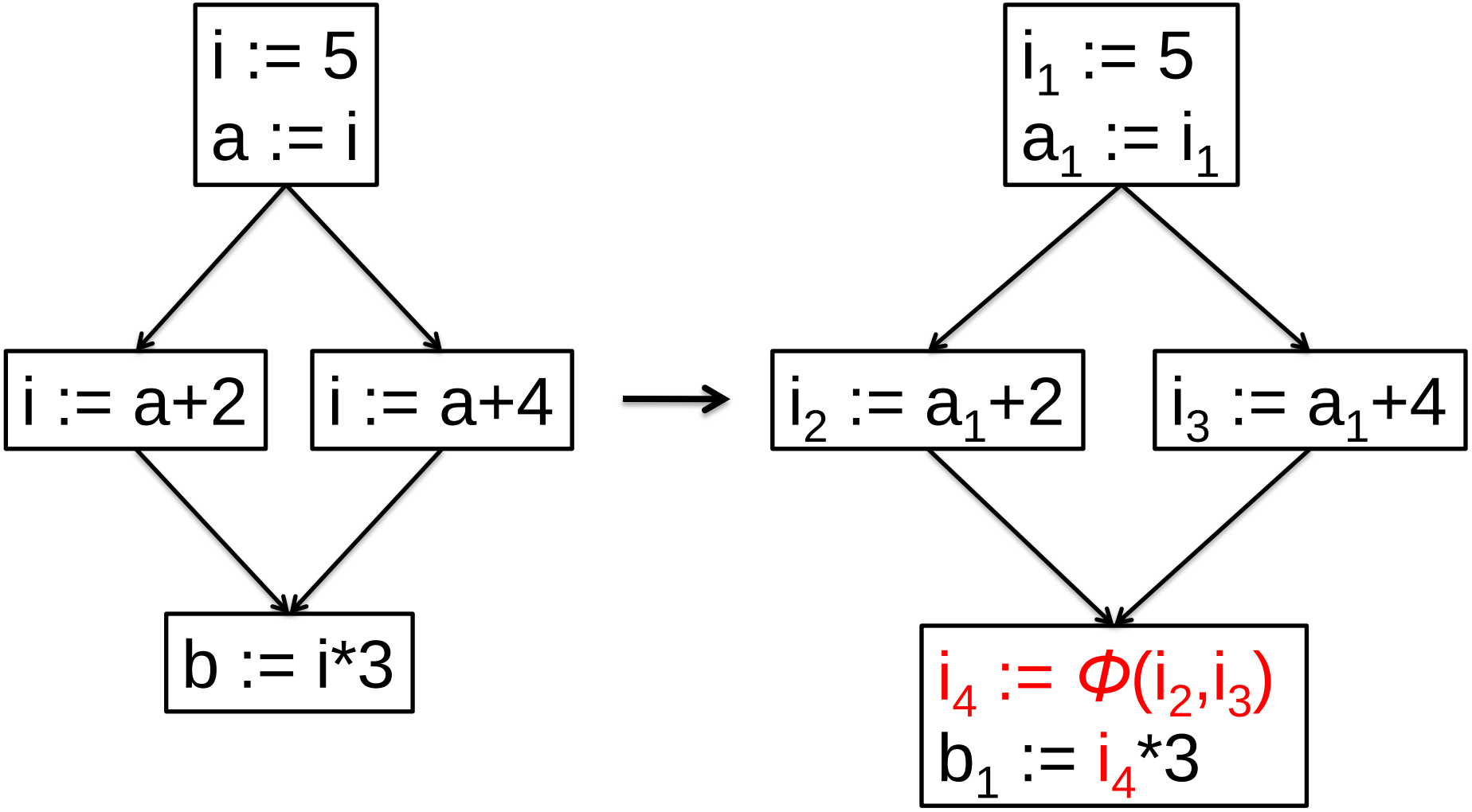
SSA példa



SSA példa: elágazás

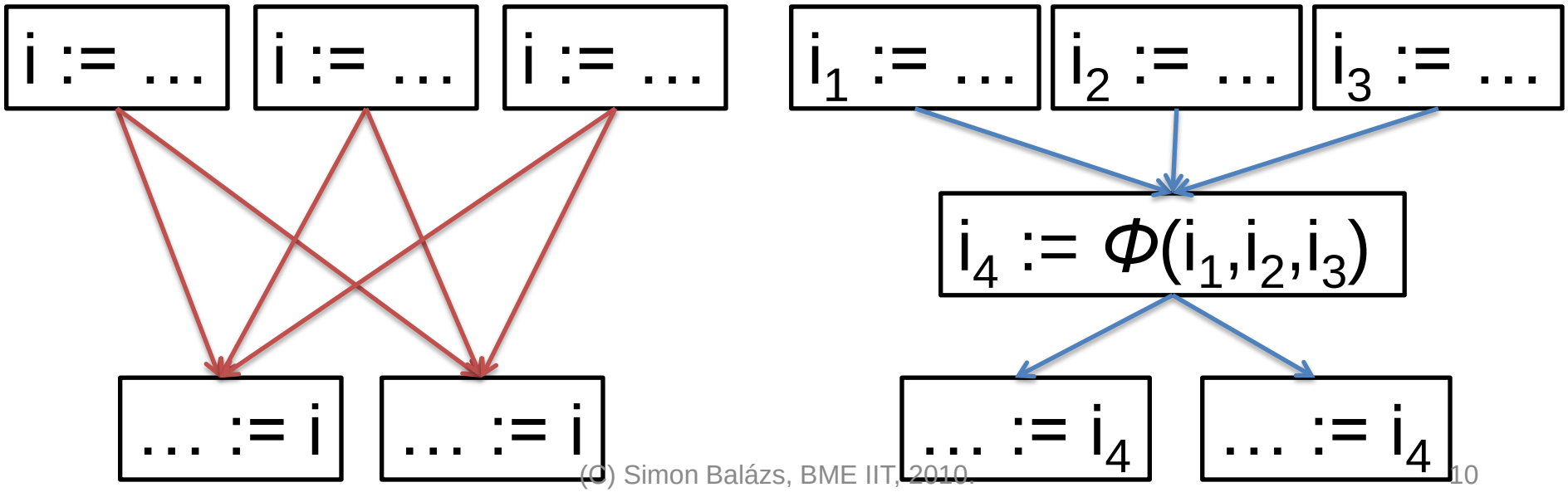


SSA példa: elágazás (Φ-függvény)



Definiálás-felhasználás

- Az SSA forma csökkenti a definiálás-felhasználás függőségek számát:
  - eddig: definiálás * felhasználás
  - most: definiálás + felhasználás



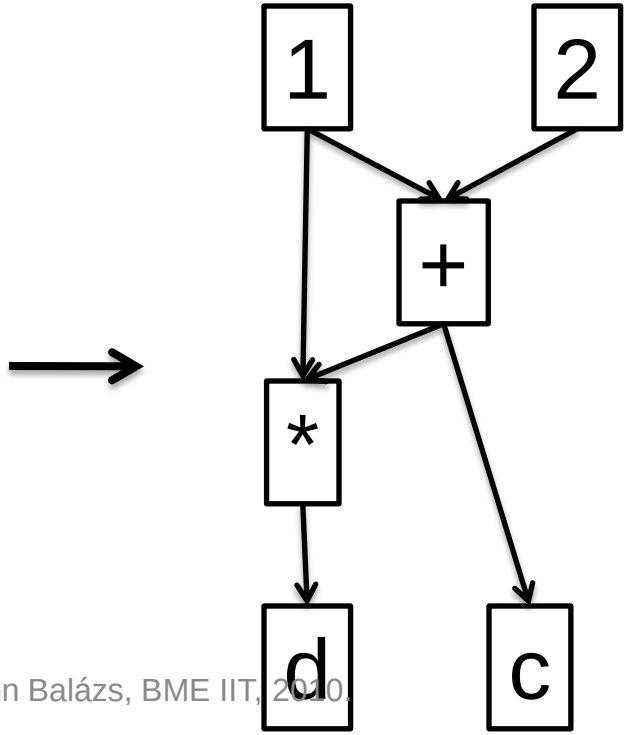
Φ-függvény

- Egy $i_3 := \Phi(i_1,i_2)$ függvény a lefutástól függően i_1 és i_2 közül kiválasztja a megfelelőt és ezt adja i_3 értékéül
- A Φ-függvény bemeneteinek száma a közvetlen megelőző kódblokkok száma
- A Φ-függvény k. bemenete egyértelműen a k. blokkhoz van rendelve
- A Φ-függvény kimenete a programfutásnak megfelelőághoz rendelt bemenet
- A Φ-függvény mindig blokkok elején jelenik meg
- Egy blokkban minden Φ-függvény egyszerre értékelődik ki

SSA gráfreprezentáció

- Csúcs: absztrakt érték (konstans, művelet)
- Adatfolyam él: Definiálási-felhasználási függőség
  - Az élek megfordítva adatfüggőségnek felelnek meg
- Vezérlési él: végrehajtási sorrend, elágazások, stb.
- Példa:

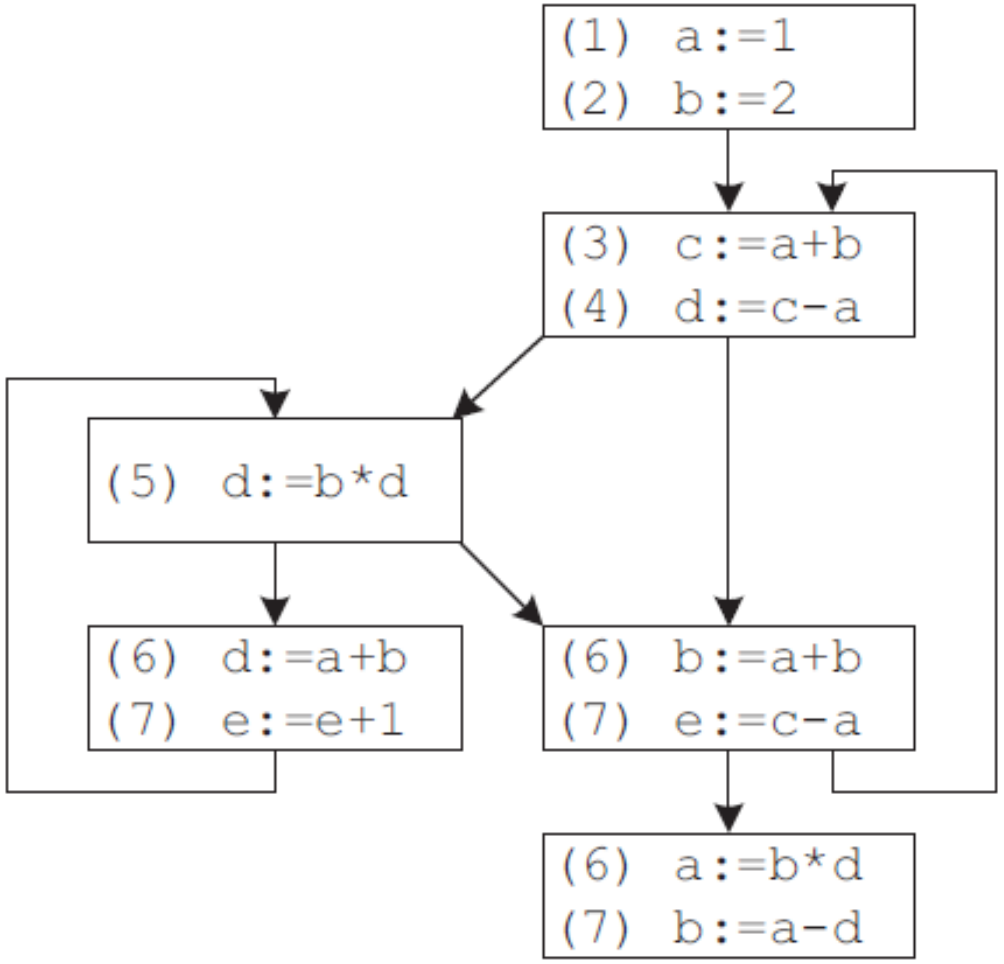
```
a := 1
b := 2
c := a+b
d := c*a
...
```



SSA példa

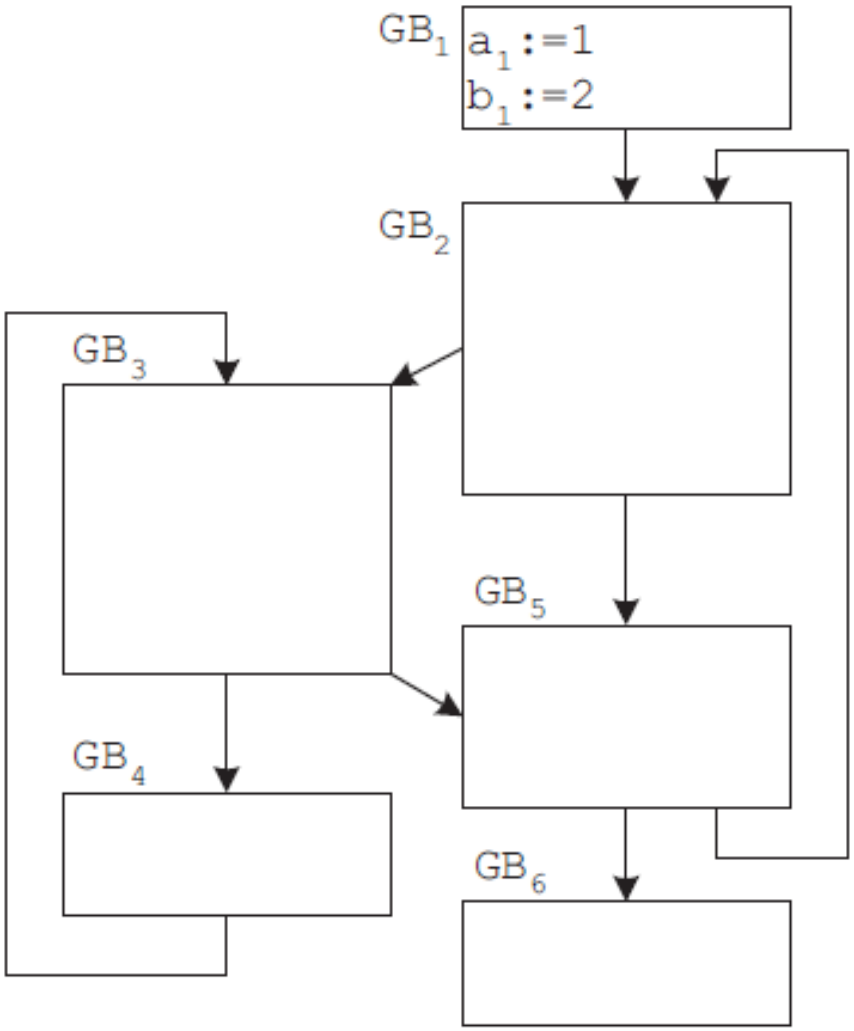
Példaprogram

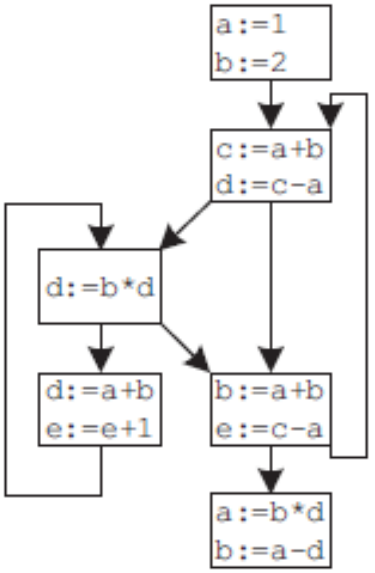
```
(1)  a:=1;
(2)  b:=2;
     while true {
(3)    c:=a+b;
(4)    if (d=c-a)
(5)      while (d=b*d) {
(6)        d:=a+b;
(7)        e:=e+1;
         }
(8)    b:=a+b;
(9)    if (e=c-a) break;
     }
(10) a:=b*d;
(11) b:=a-d;
```



1. blokk

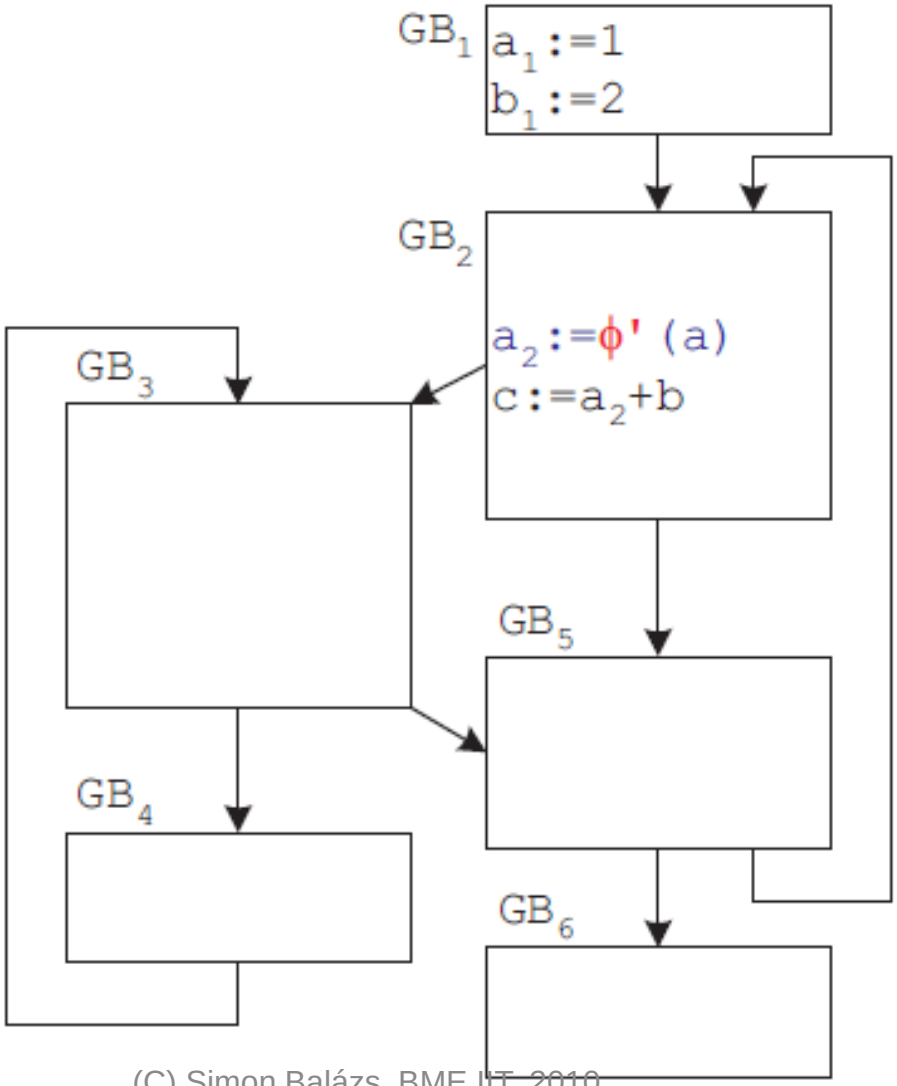
- Megszámozzuk *a*-t és *b*-t

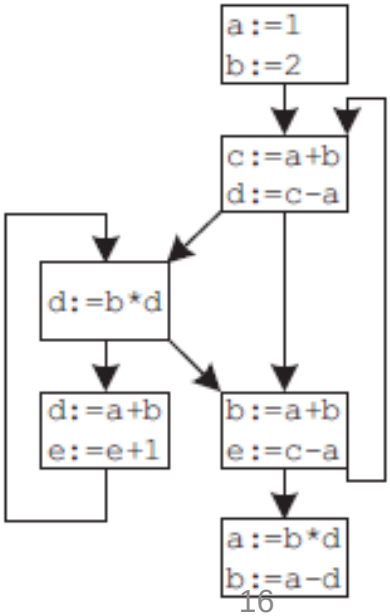




2. blokk

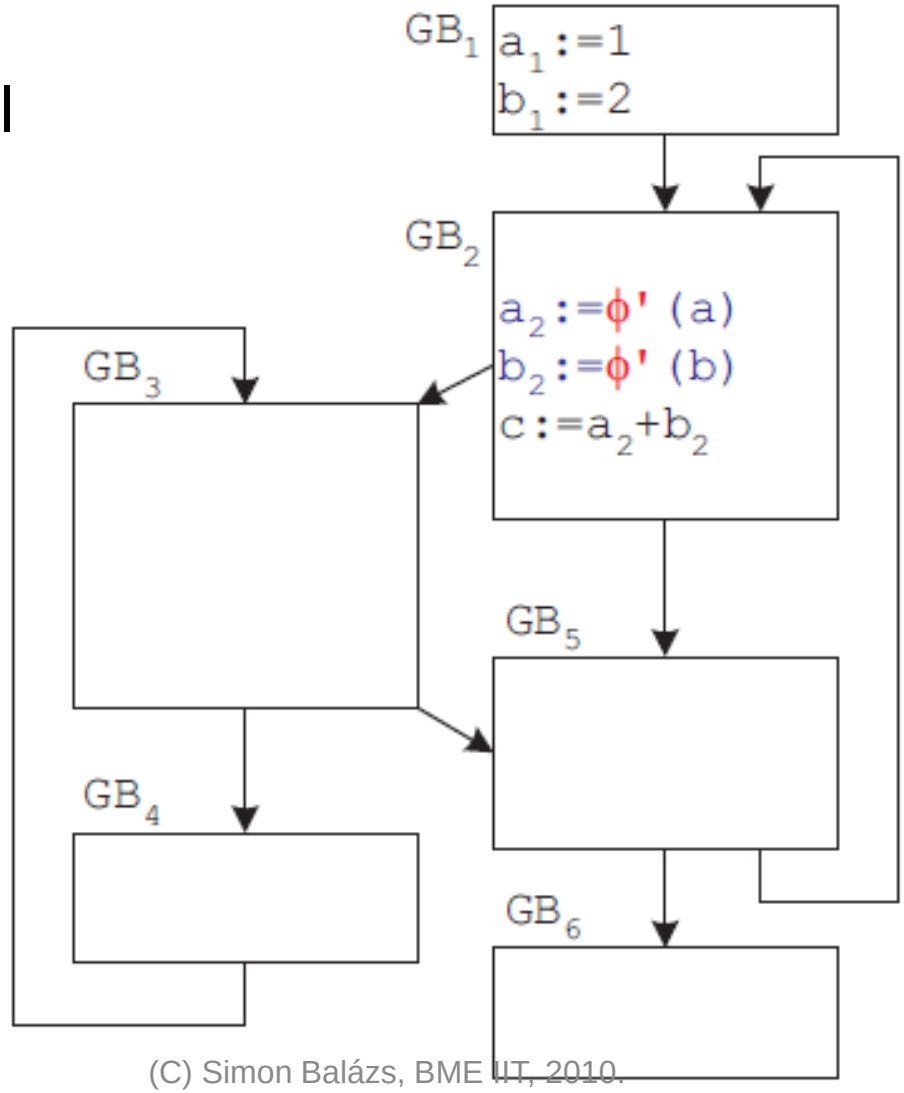
- Az *a* értéke függ a lefutástól
- Egyelőre csak megjegyezzük, mivel nem minden megelőző értéket ismerünk

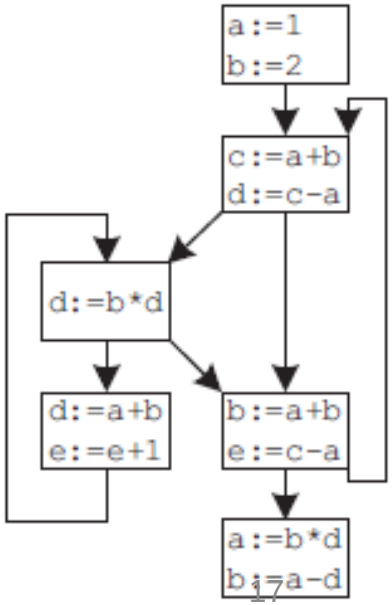




2. blokk

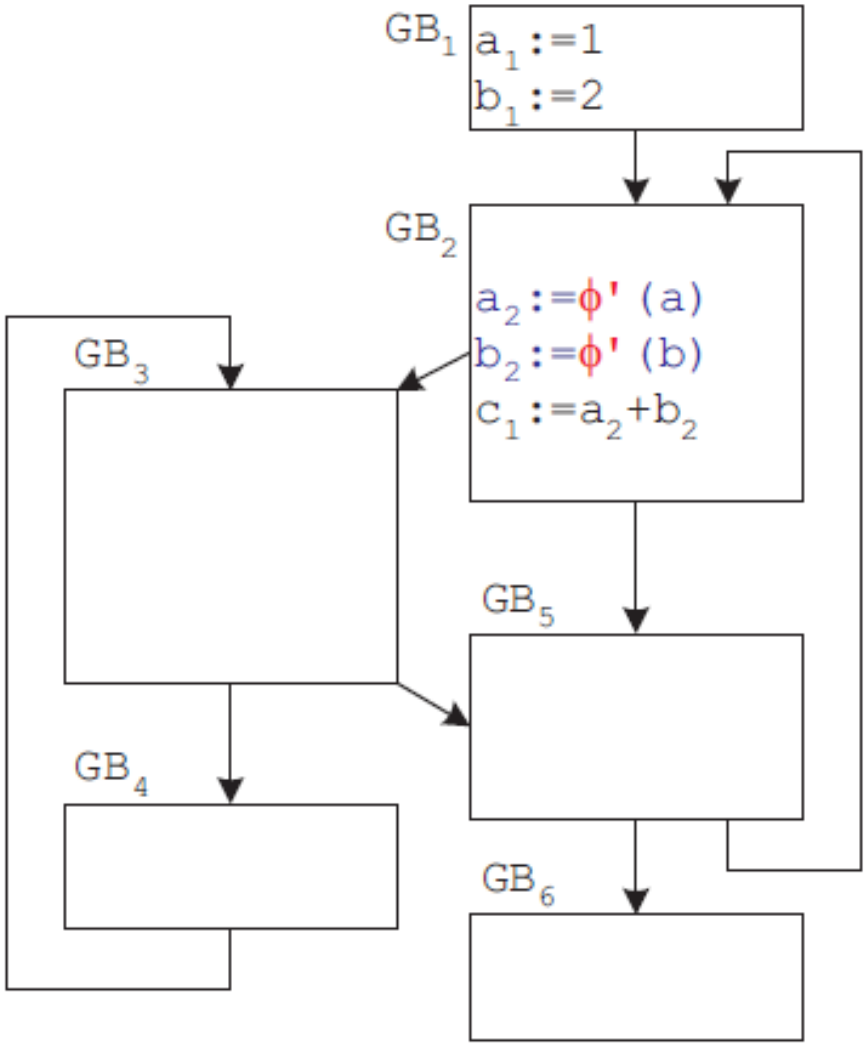
- A *b* értéke is függ a lefutástól

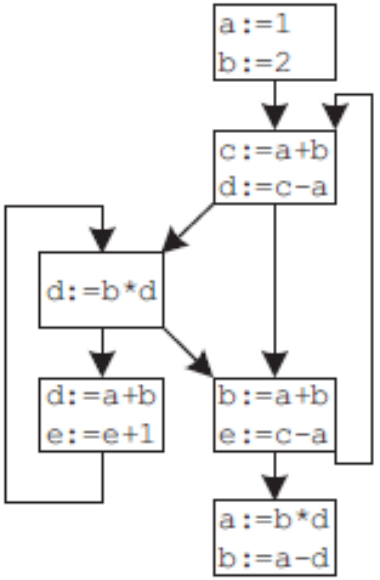




2. blokk

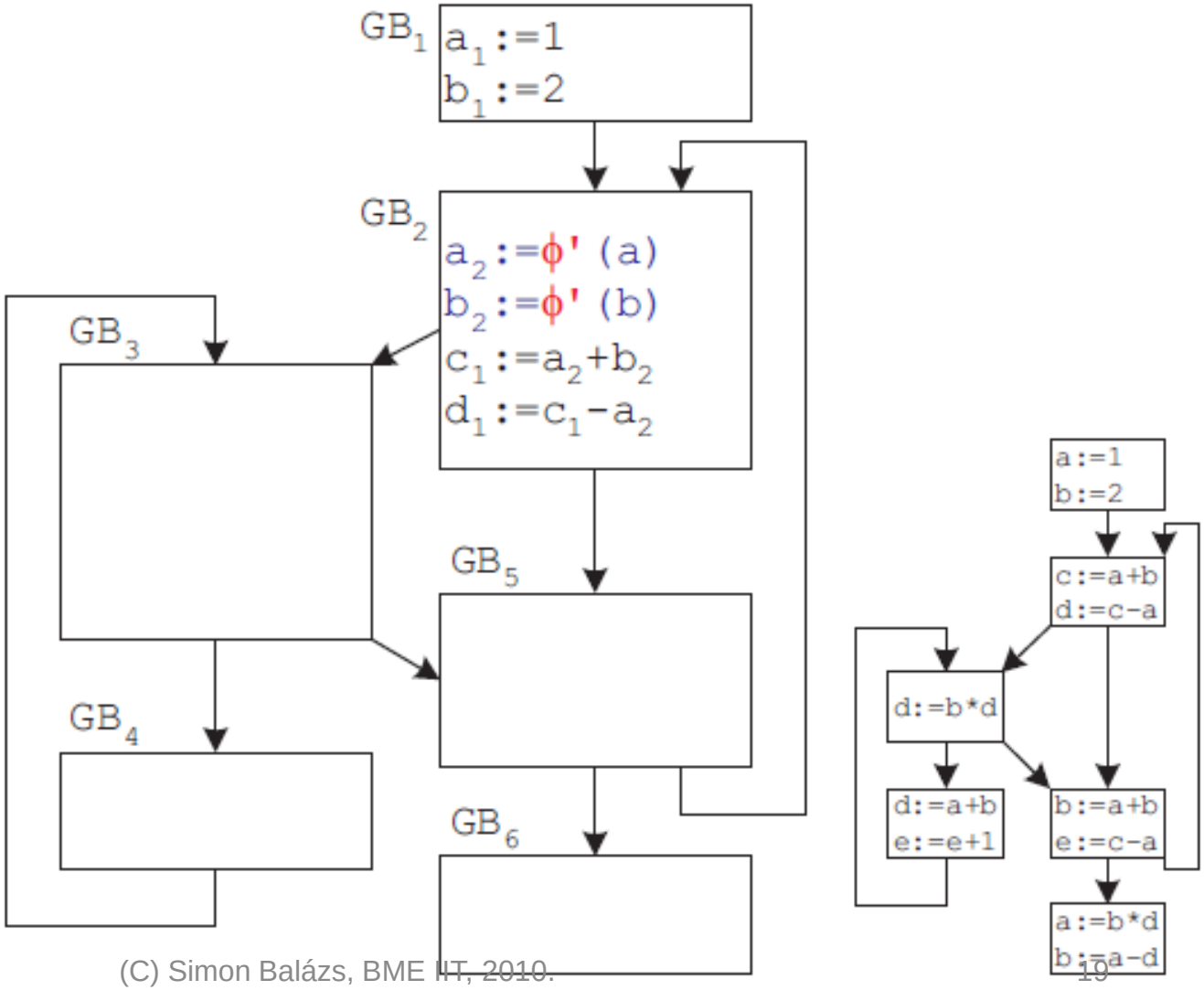
- Majd megszá-
  mozzuk a *c*-t





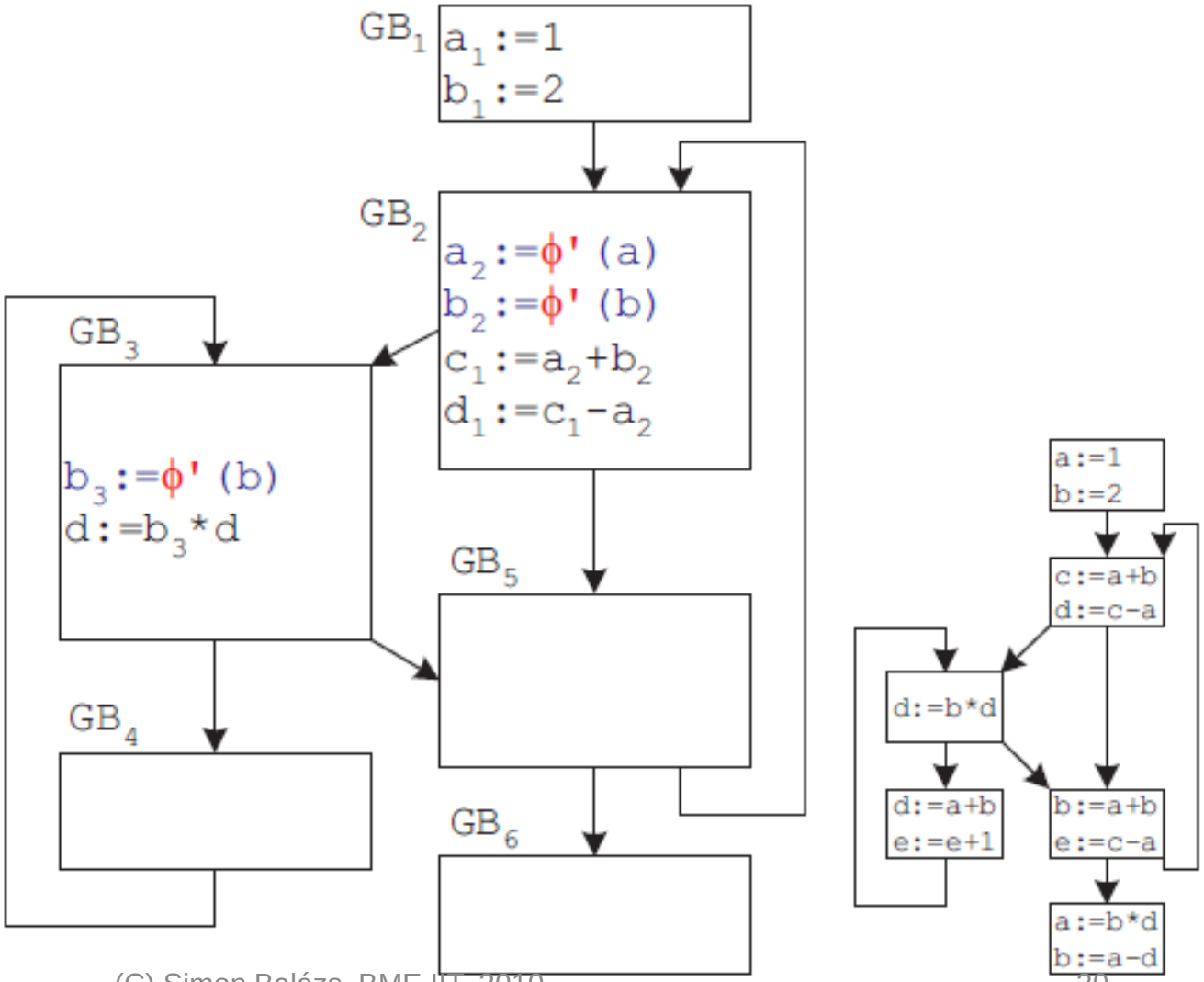
2. blokk

- Végül pedig *d*-t



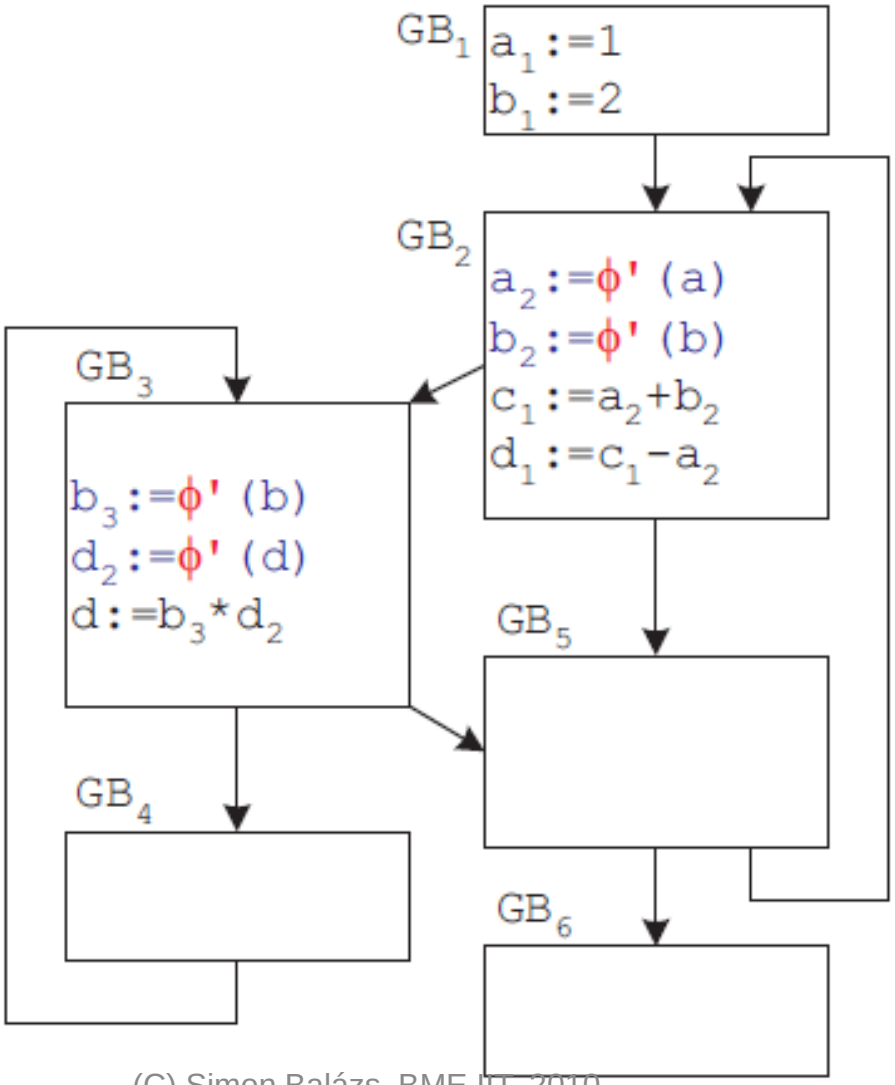
3. blokk

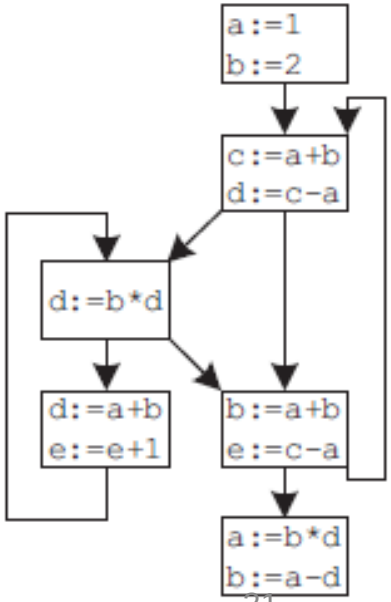
- A *b* értéke függ a lefutástól



3. blokk

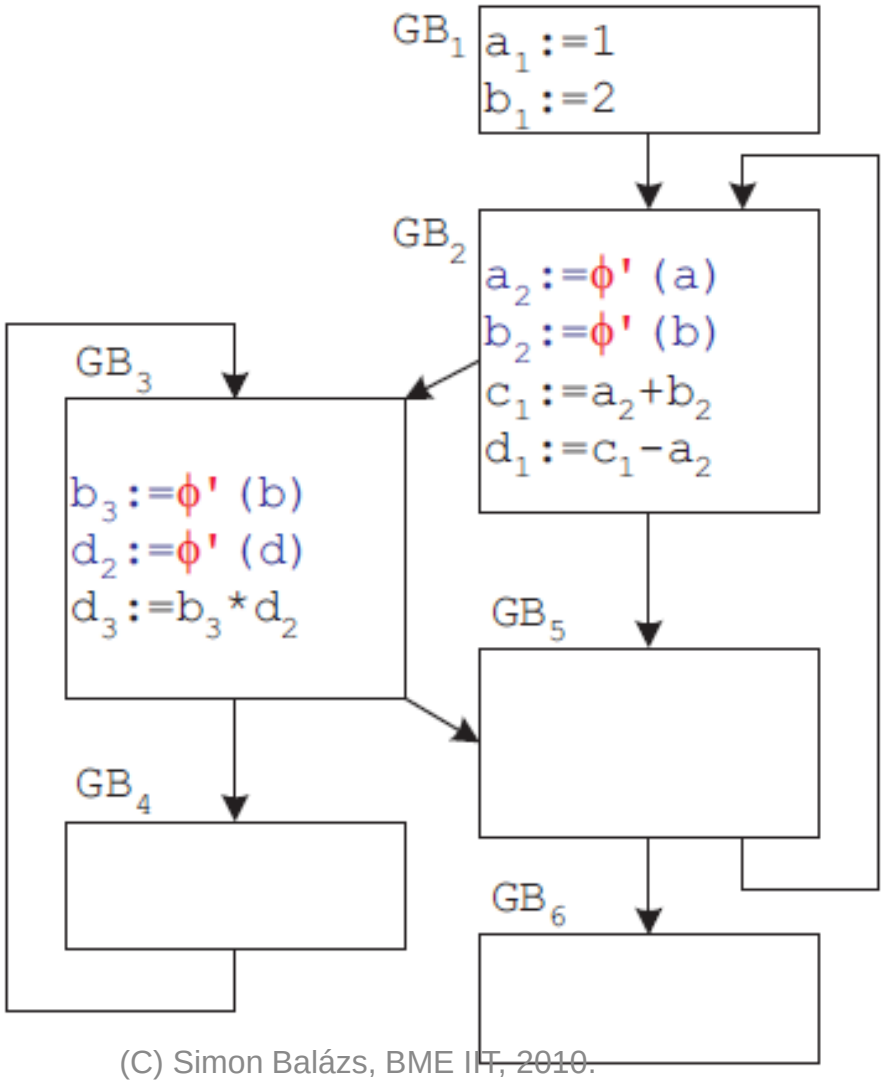
- A *d* értéke is függ a lefutástól

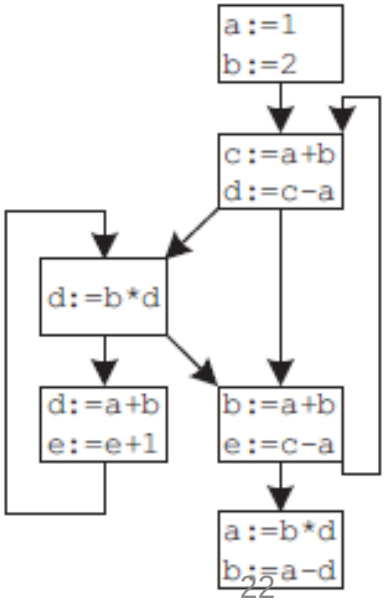




3. blokk

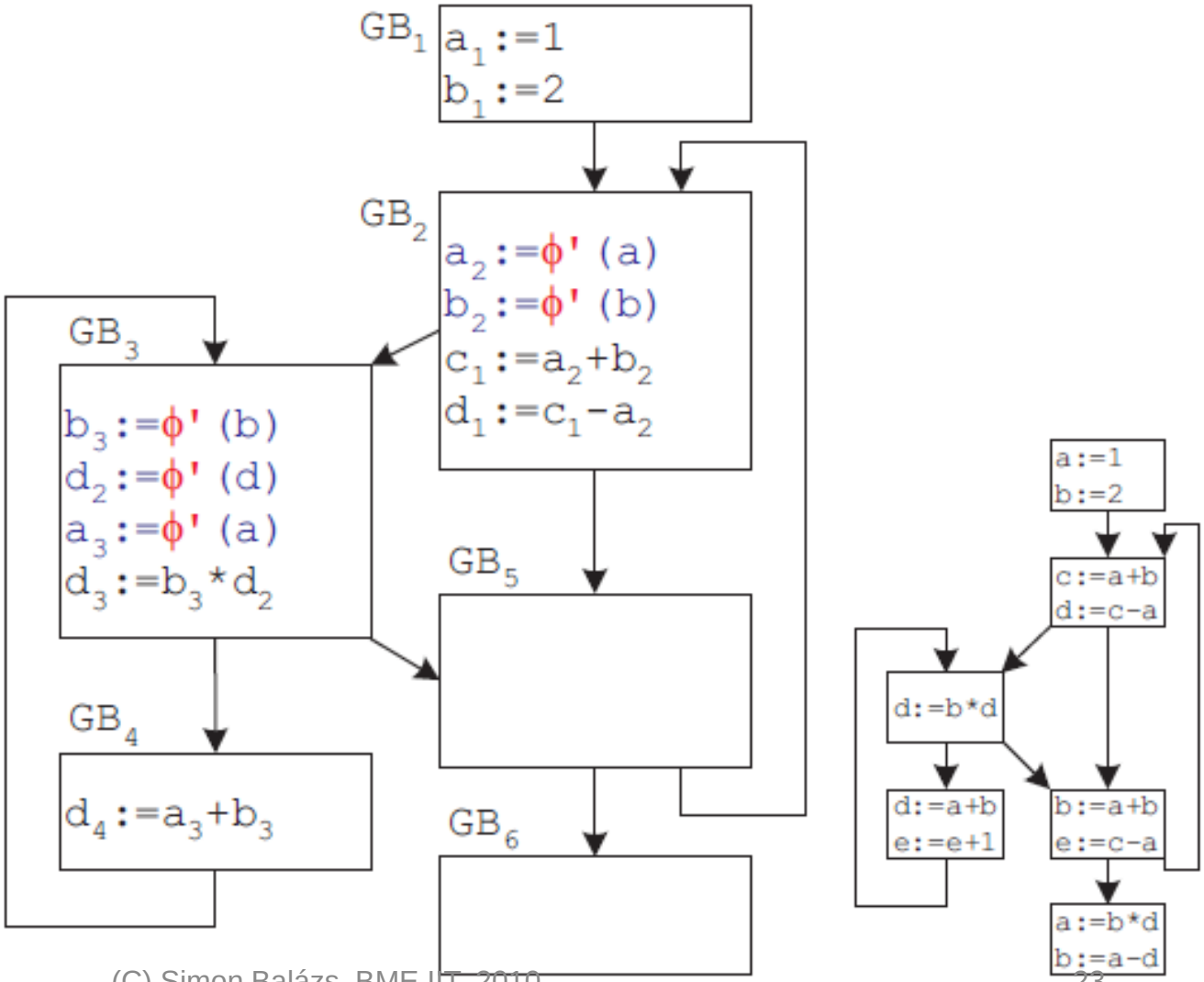
- Megszámozzuk a *d*-t





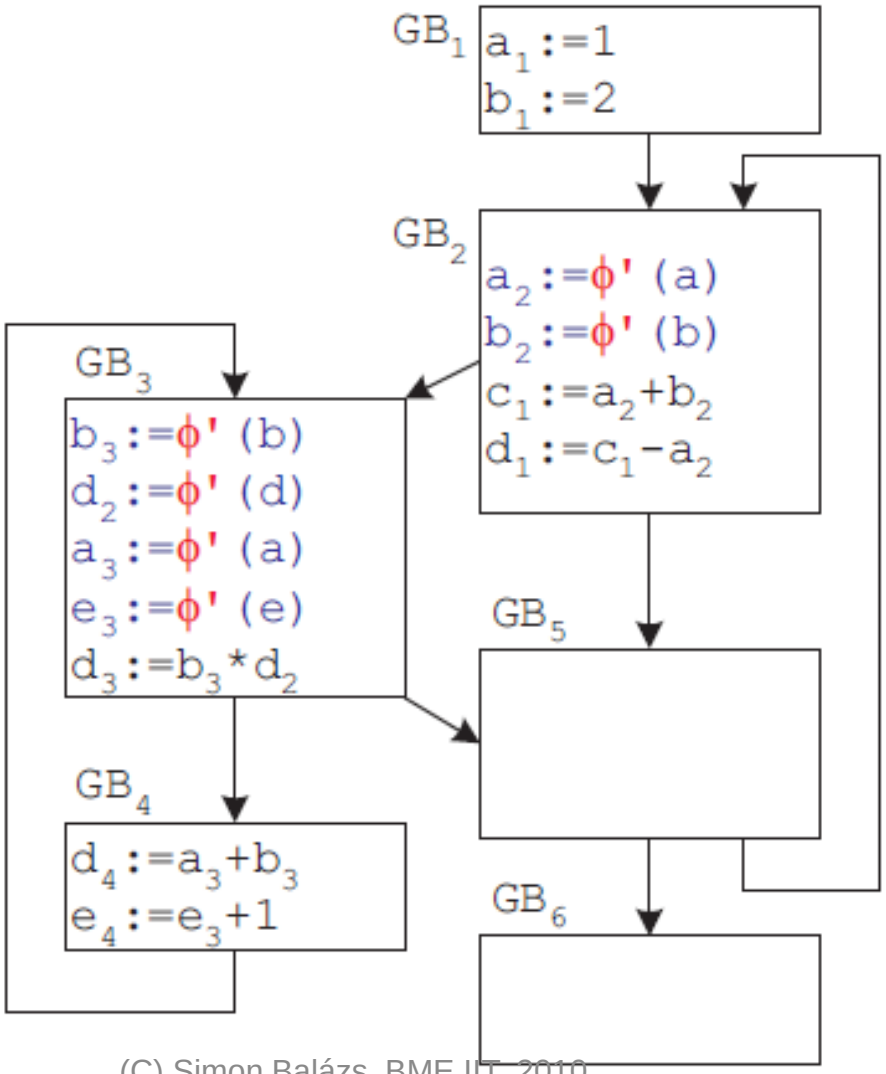
4. blokk

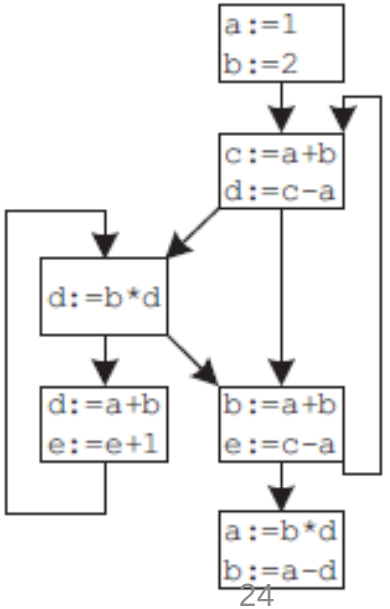
- Az *a* használata a 4. blokkban rekurzívan új *Φ* függvényt eredményez a 3. blokkban



4. blokk

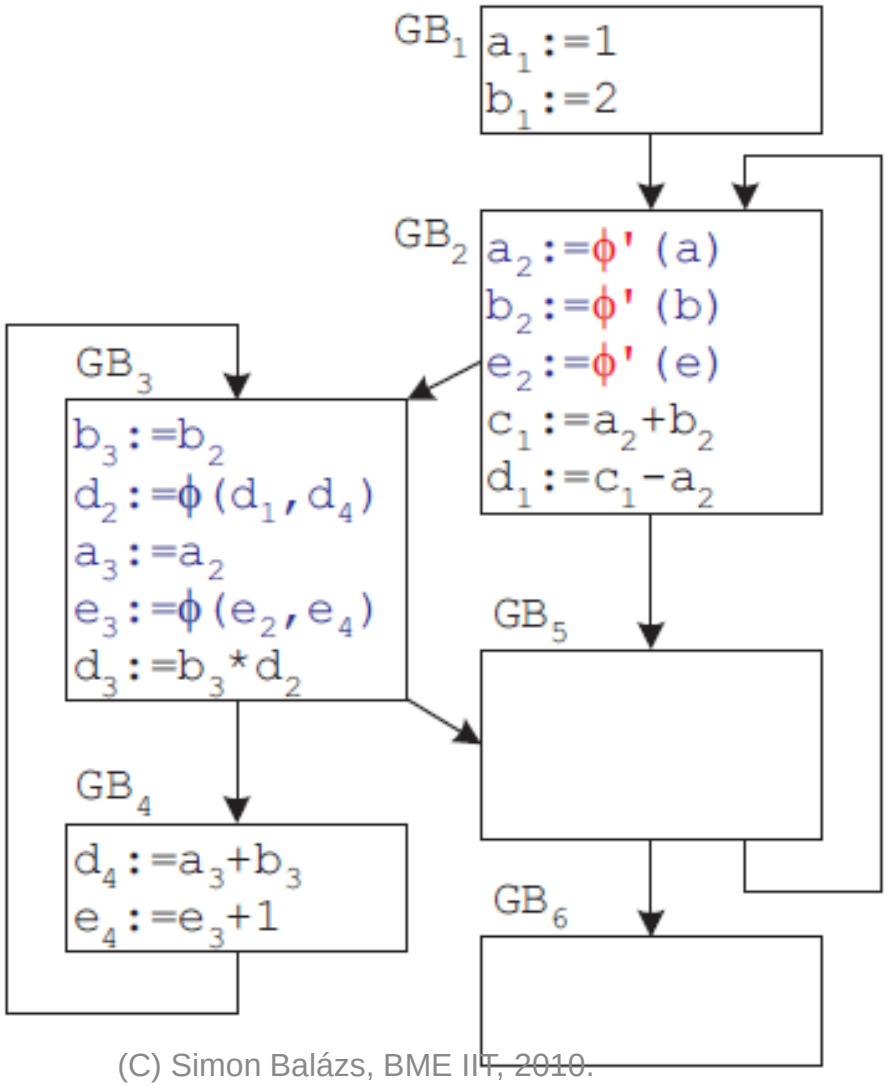
- Az *e* használata a 4. blokkban rekurzívan új *Φ* függvényt eredményez a 3. blokkban

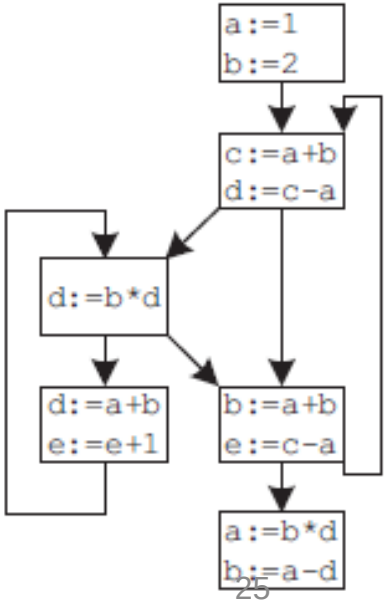




4. blokk

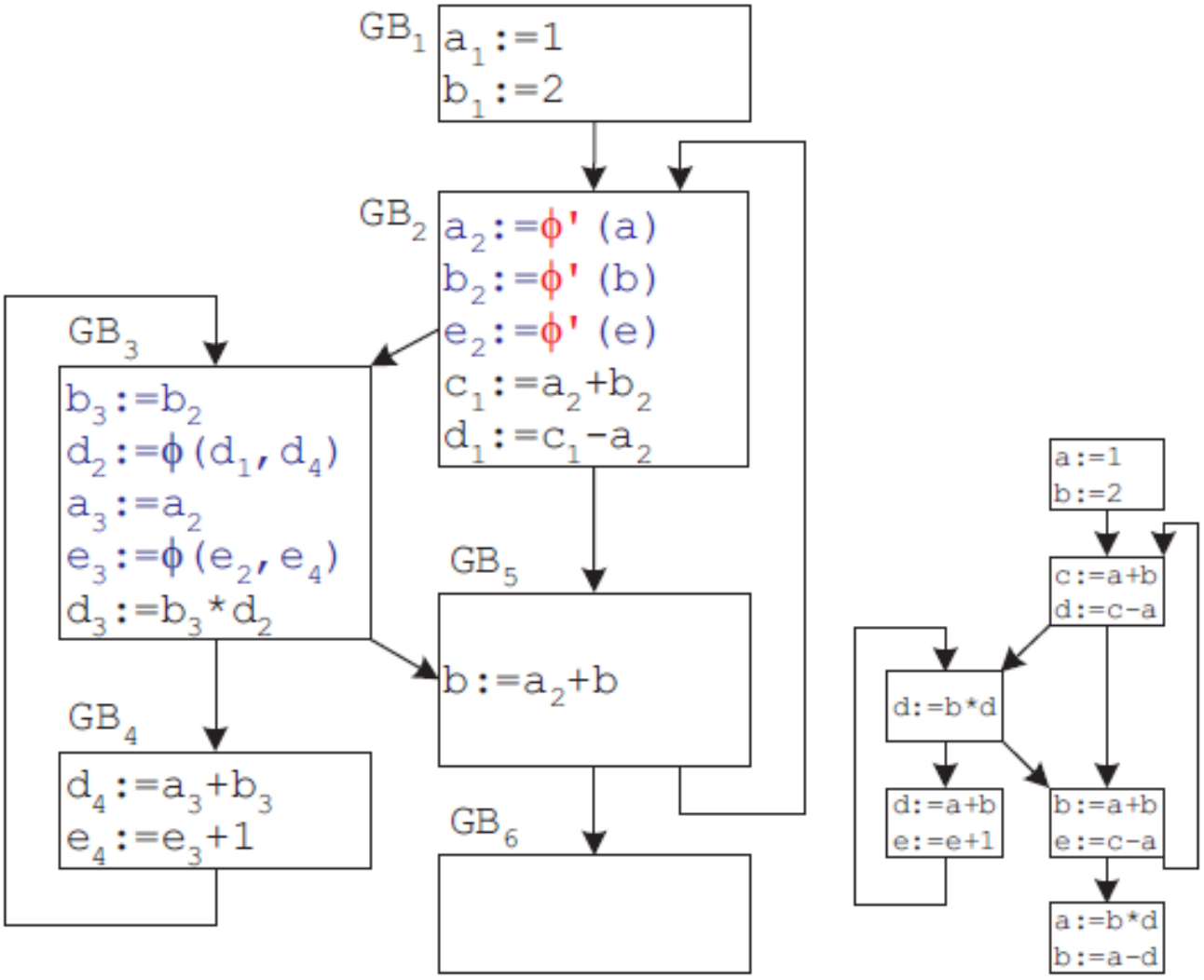
- A 3. blokk minden megelőzője SSA formában van
- A Φ függvények kiszámíthatók





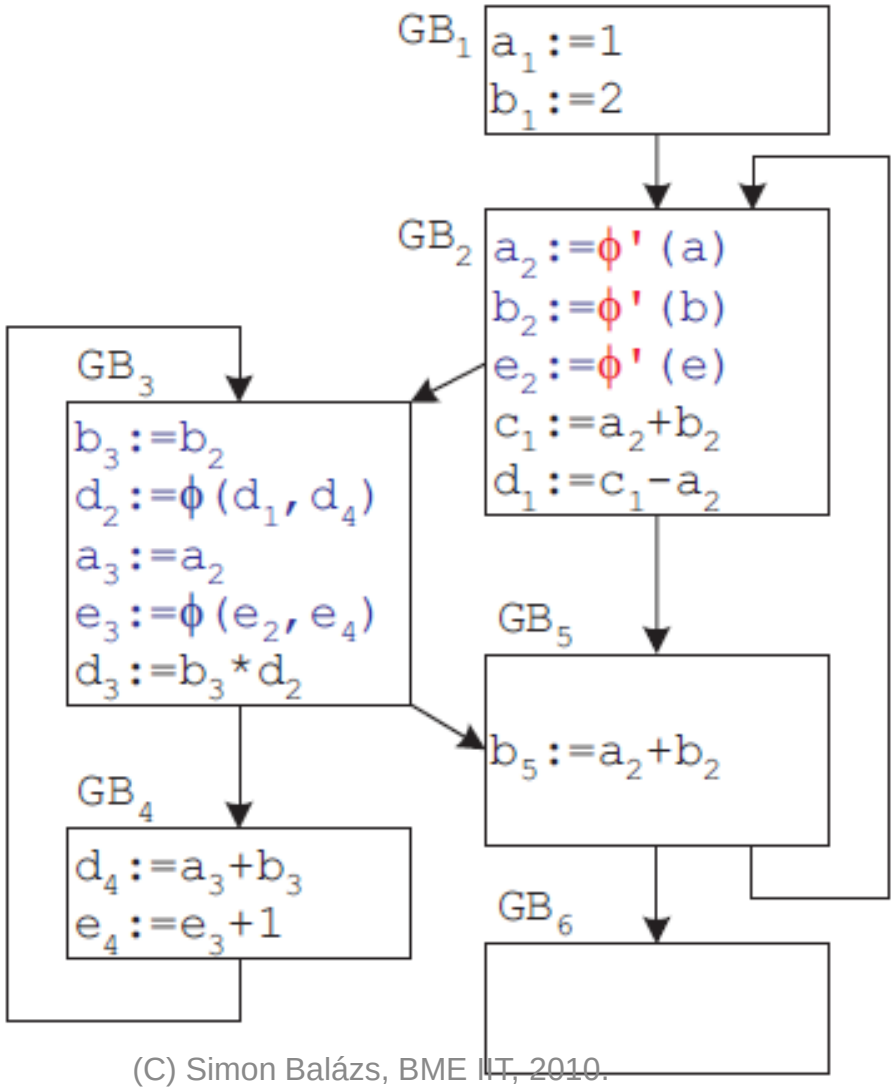
5. blokk

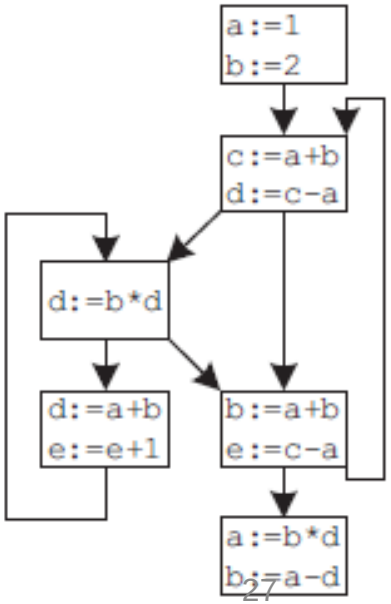
- Az *a* definíciója minden ágon ugyanaz, nincs szükség Φ függvényre



5. blokk

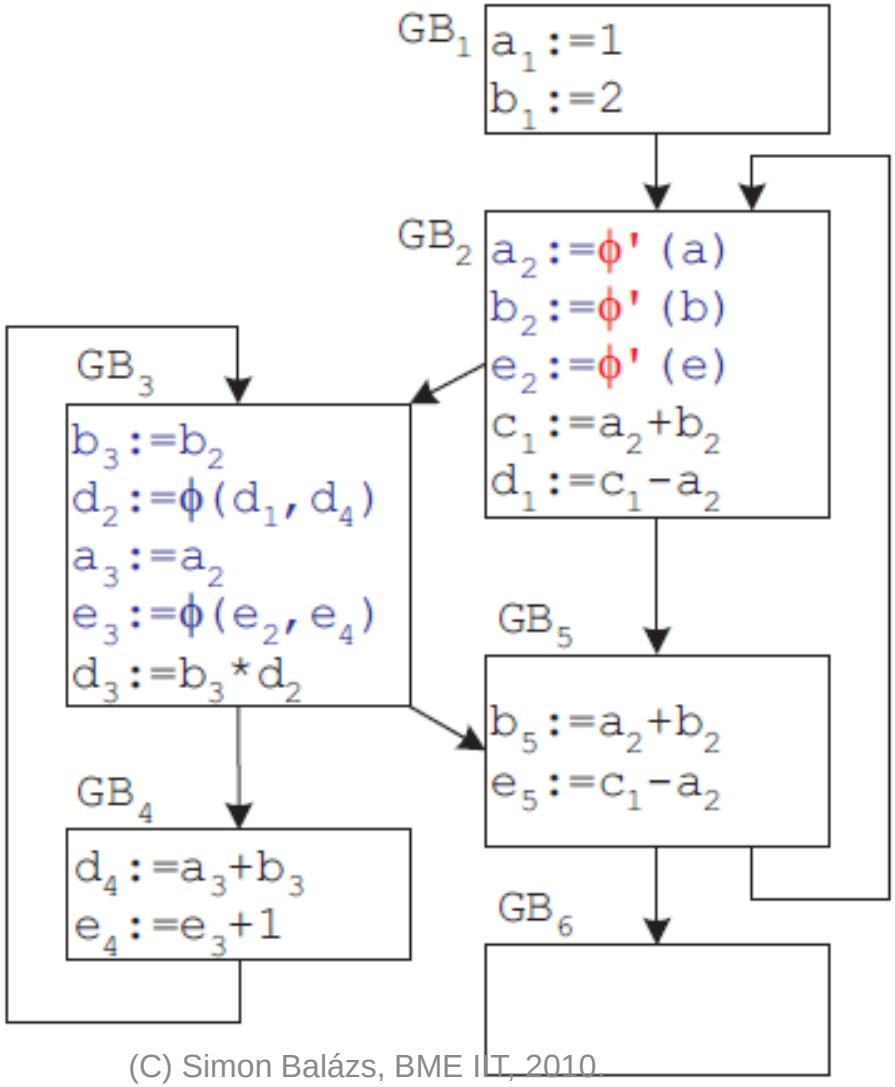
- A *b* definíciója minden ágon ugyanaz, nincs szükség *Φ* függvényre

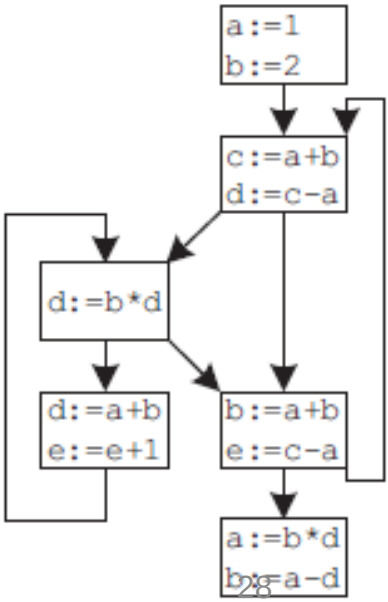




5. blokk

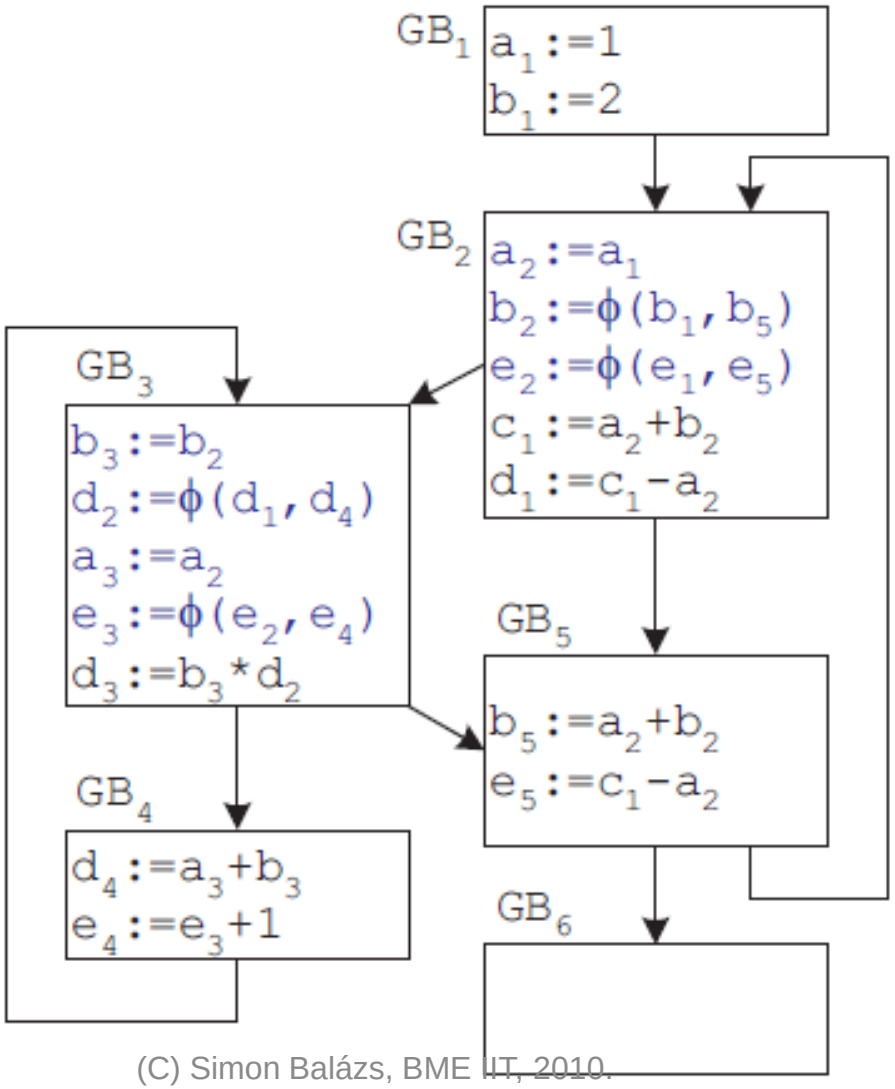
- A *c* definíciója minden ágon ugyanaz, nincs szükség *Φ* függvényre

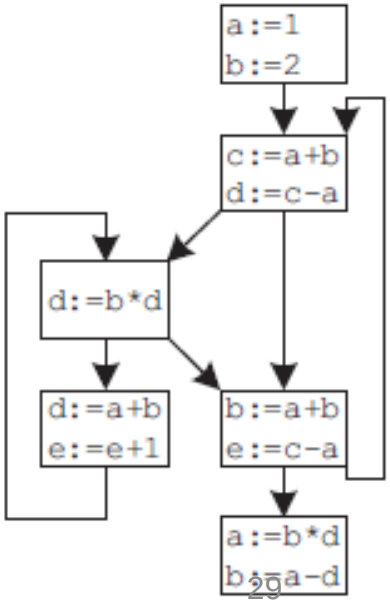




5. blokk

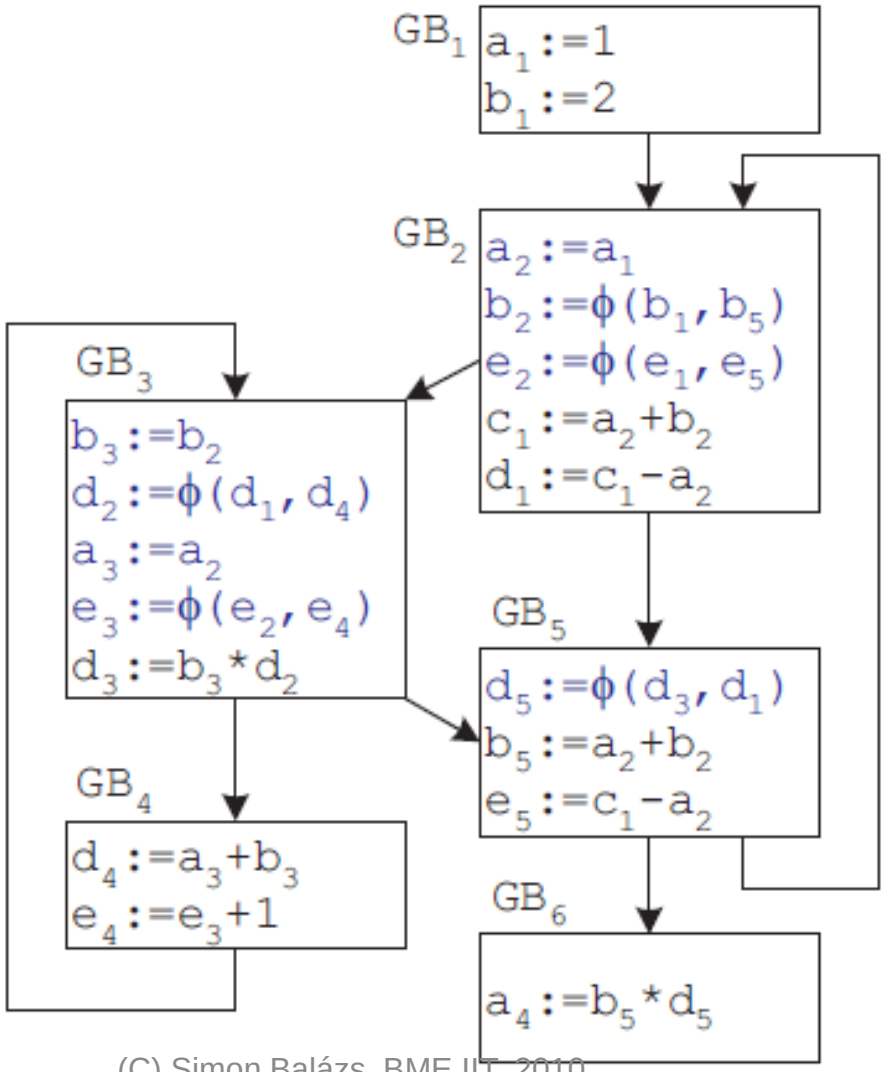
- A 2. blokk minden megelőzője SSA formában van, a Φ függvények számíthatók
- Az algoritmus felismeri, hogy *e* nincs inicializálva: feltesszük, hogy értéke e_1

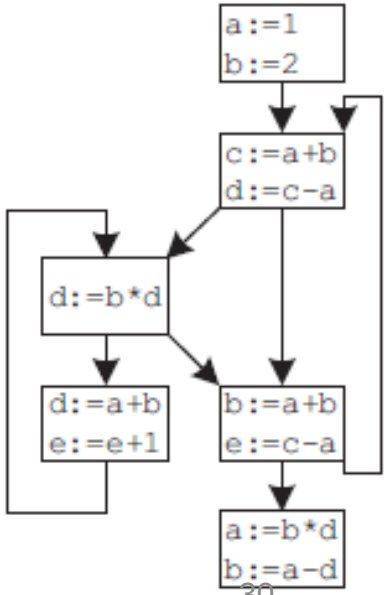




6. blokk

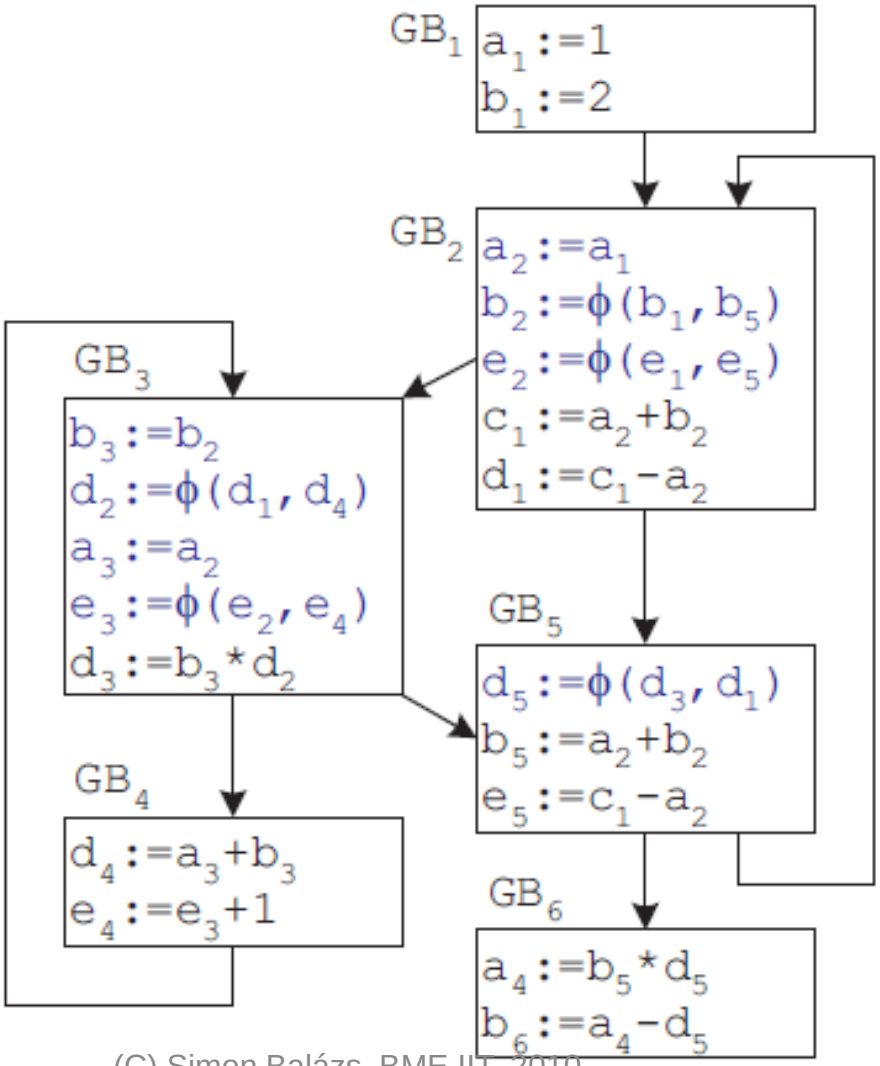
- A 6. blokkban a *d* keresése rekurzívan új Φ függvényt eredményez az 5. blokkban

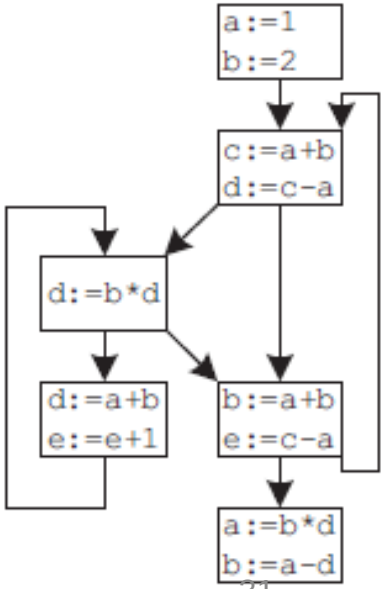




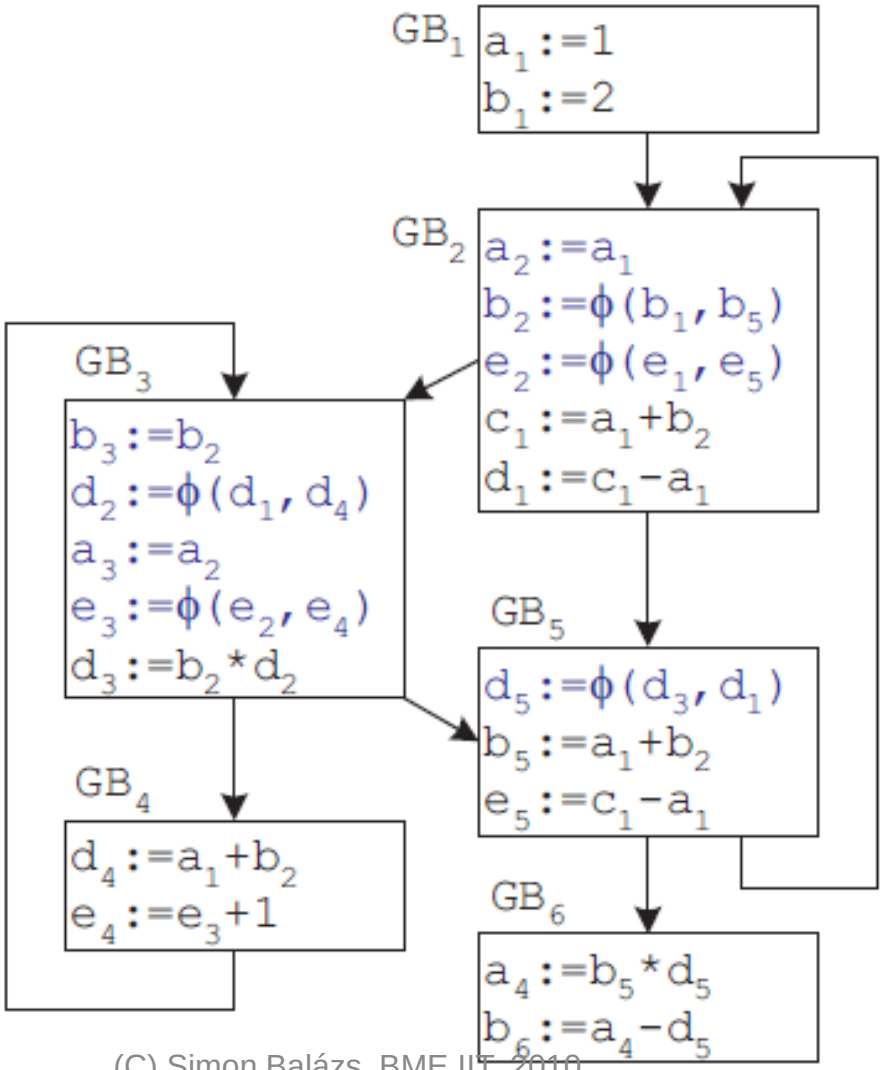
6. blokk

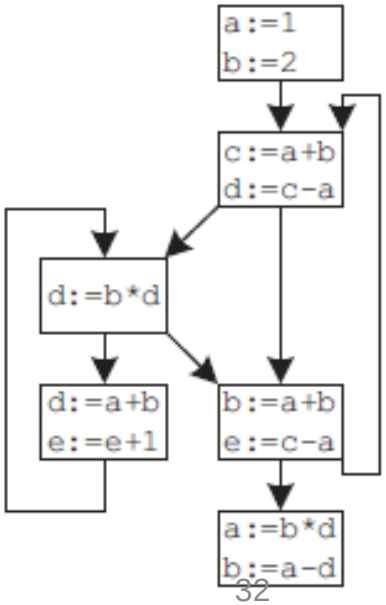
- A *b* azonnal számítható



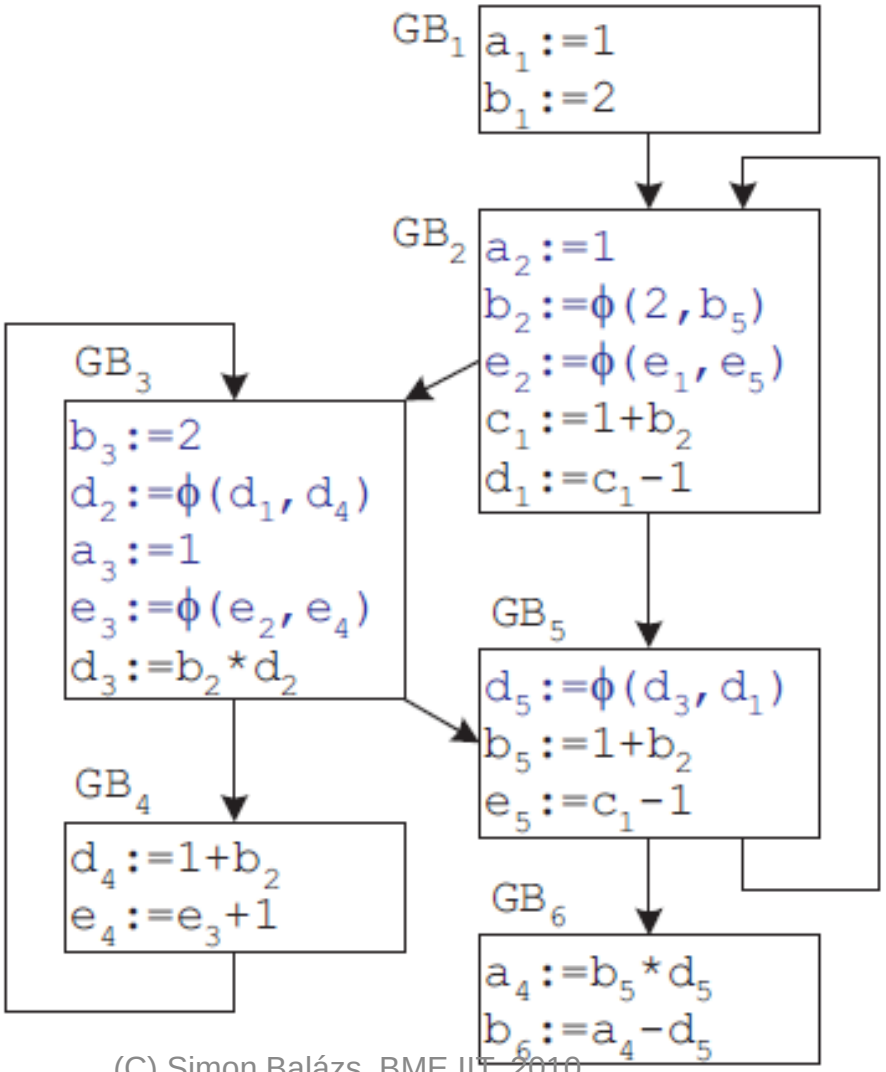


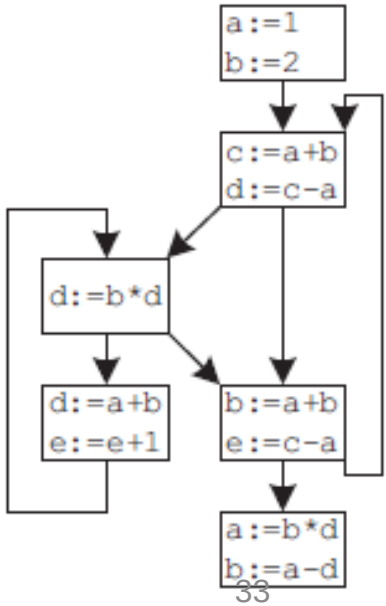
Egyszerűsítések: másolatok feloldása



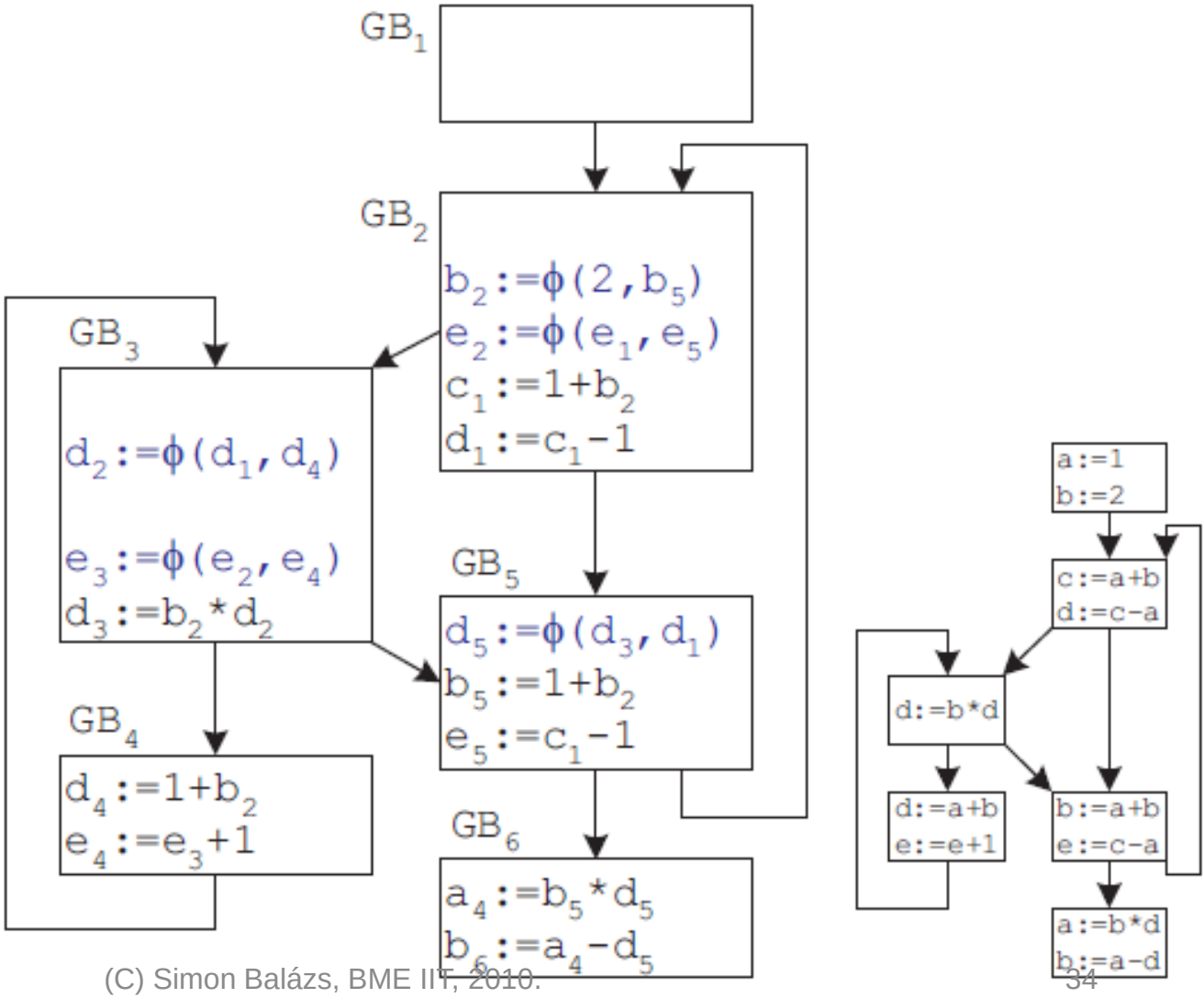


Egyszerűsítések: konstansok feloldása



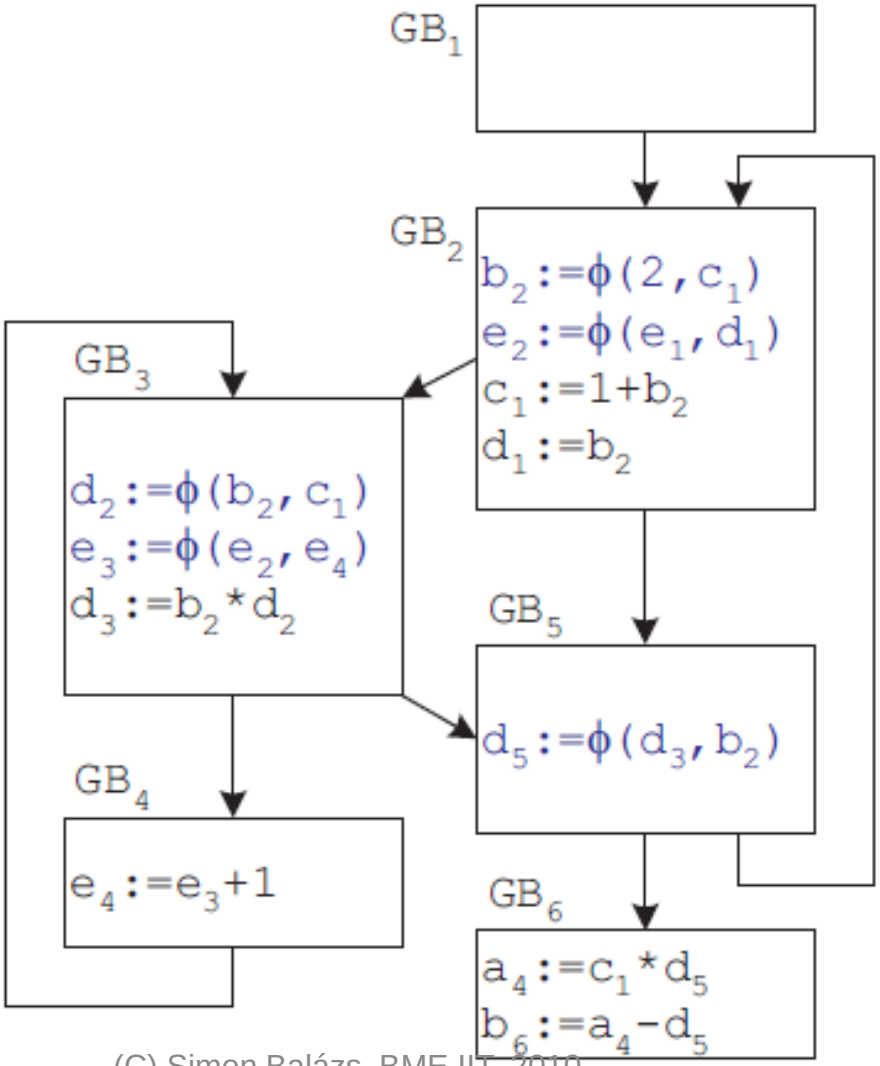


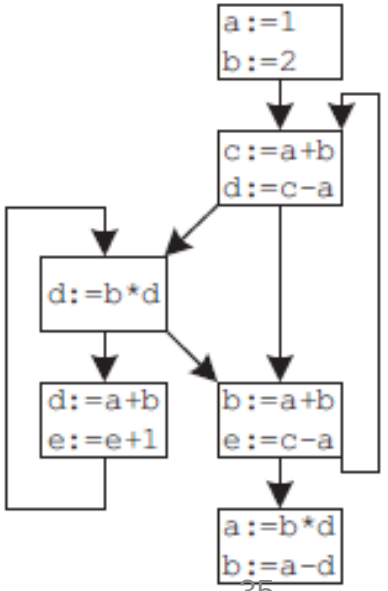
Egyszerűsítések: halott kód eliminálása



További egyszerűsítések

- Közös részkifejezések
- Konstans kifejezések kiértékelése
- Konstansok feloldása
- Halott kód eliminálása





SSA felépítése AST-ből

- Balról-jobbra történő bejárás:
  - Az aktuális blokk globális változóban
- Kifejezés: SSA forma generálása az aktuális blokkba
  - A köztes eredményeket közvetlenül újrahasznosítjuk, nincs szükség explicit számozásra
- Utasítások:
  - Blokk generálása
  - Blokkok kódjának generálása
  - A blokkok lefutásának összefűzése
  - A blokkok SSA előállításának lezárása
- A függvényhívások is kifejezésnek minősülnek
- Az ugrások lezárásához egy újabb bejárás szükséges

SSA összefoglalás

- SSA: változók helyett dinamikus konstansok
- Elágazások találkozásakor Φ függvény szükséges
- Az eredmény egy adatfolyamgráf
- Lehetővé teszi a függvényen belüli optimalizálást
- A Φ függvény felépítése: amikor az értéket felhasználjuk (rekurzívan a korábbi blokkokra is)
- A végtelen ciklusok elkerülésére lezáratlan Φ' függvények

Optimalizálás

Optimalizálási lehetőségek (nem csak SSA)

- 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
- Adatok betöltése előre
- Jobbrekurzió feloldása ciklussá
- Regiszterkiosztás

A kutatás állása

- Nyitott problémák:
  - Melyik optimalizálásokat válasszuk ki?
  - Milyen sorrendben futtassuk le őket?
- Tipikusan:
  - A cache-optimalizáláson és az operátoregyszerűsítésen kívül az első optimalizálás kb. 15%-ot javít, a többi max. 5%-ot.
  - Ezek az értékek függetlenek attól, melyik optimalizálásokat választjuk és milyen sorrendben futtatjuk le őket.
  - Sok optimalizálás hasonló ill. tartalmazza a másikat.
  - Numerikus programoknál az operátoregyszerűsítés 2x-es, a cache-optimalizálás 2-5x-ös gyorsulást is elérhet.

Tétel az optimalizálásról

- Rice, 1953: Minden algoritmikusan működő U fordító esetén létezik U' fordító, amely bizonyos programokra rövidebb kódot állít elő.
- Következmény: minden fordítónál van jobb fordító.
- Bizonyítás: egy nem végetérő programot a legjobb fordító így optimalizálna:

```
m: goto m;
```

Ezáltal a megállási probléma algoritmikusan eldönthető lenne, ami lehetetlen.

Az optimalizálás költségfüggvénye

- Futásidő
- Memóriahasználat
- Energiafelhasználás

- A pontos értékek nem határozhatók meg a processzor felépítése miatt:
  - utasítások átrendezése
  - cache-elés: adatfüggő
  - hyperthreading

SSA optimalizálás

- Könnyen végrehajtható az SSA formán
- Kétfajta transzformáció:
  - normalizálás
    - az SSA gráfban különbözőképpen leírt kifejezések összehasonlíthatóak legyenek
    - elősegíti az optimalizációt
    - algebrai azonosságok felhasználása
      - asszociativitás, disztributivitás
    - alkalmazhatóságát korlátozzák:
      - kiértékelési sorrend (Java)
      - kivételek (Eiffel, Java)
      - fixpont aritmetika (nem asszociatív és nem disztributív)
  - optimalizálás

Tanult optimalizálások I.

- 1. Operátoregyszerűsítés
  - ötlet: a drága operátorok helyettesítése olcsóbb, de szemantikusan ekvivalens operátorokkal
  - fő felhasználási terület: ciklusok indexváltozóival való szorzás helyett összeadás

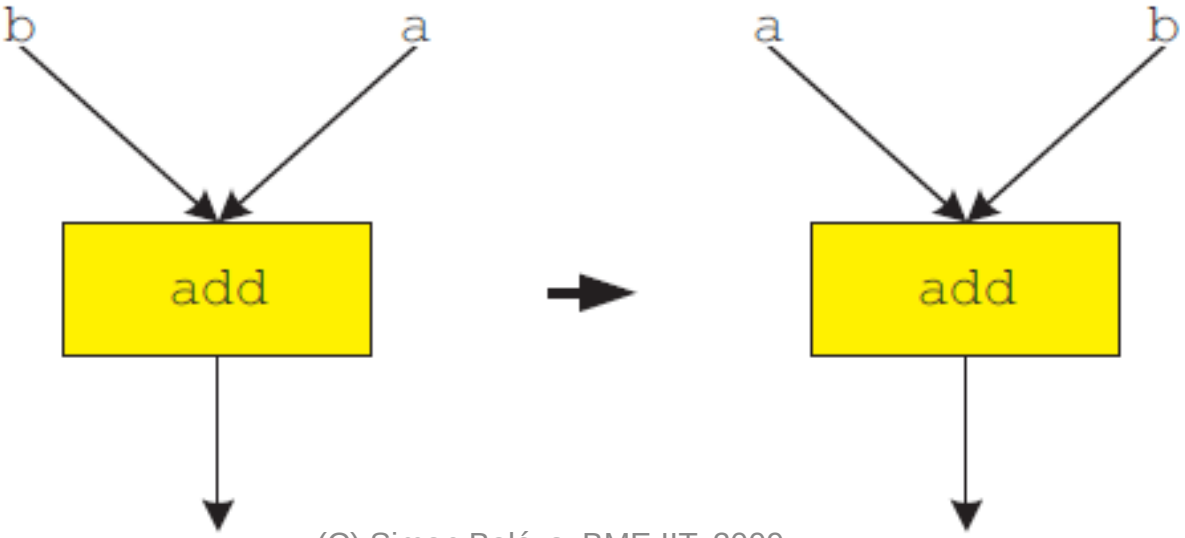
Tanult optimalizálások II.

- 2. Közös részkifejezések eliminálása
  - ötlet: ugyanazon értékek kiszámítása csak 1x
  - fő felhasználási terület: címszámítás
- 3. Részleges redundanciák eliminálása
  - ötlet: azon értékek kiszámításának elkerülése, amelyeket nem kell minden ágon számolni
  - felhasználás: ciklusinvariáns kód kivétele a ciklusból

Alapvető transzformációk

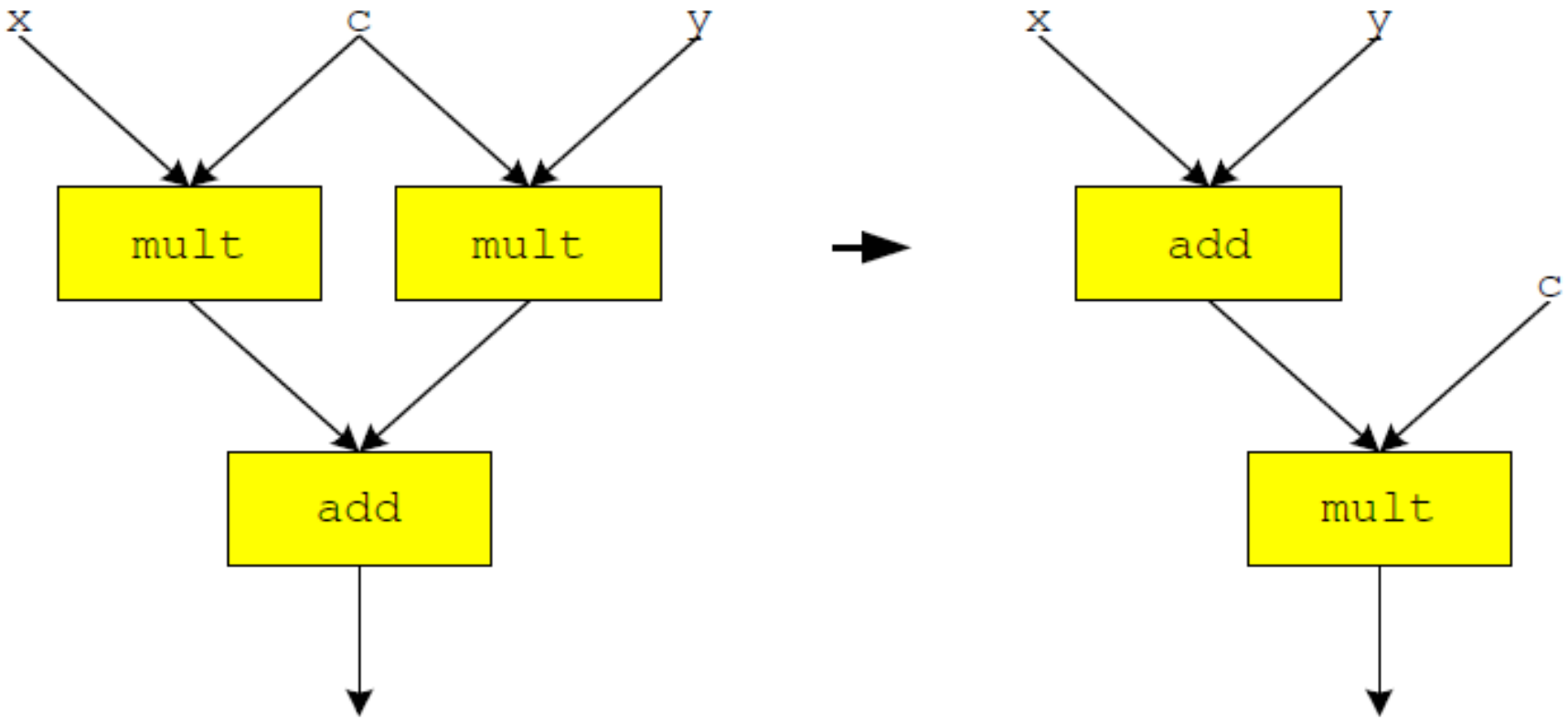
Normalizálás

- Cél: aritmetikai kiejezések összehasonlíthatóvá tétele közös részkifejezések felismeréséhez
- Pl. a csomópontok neve alapján: kommutativitás



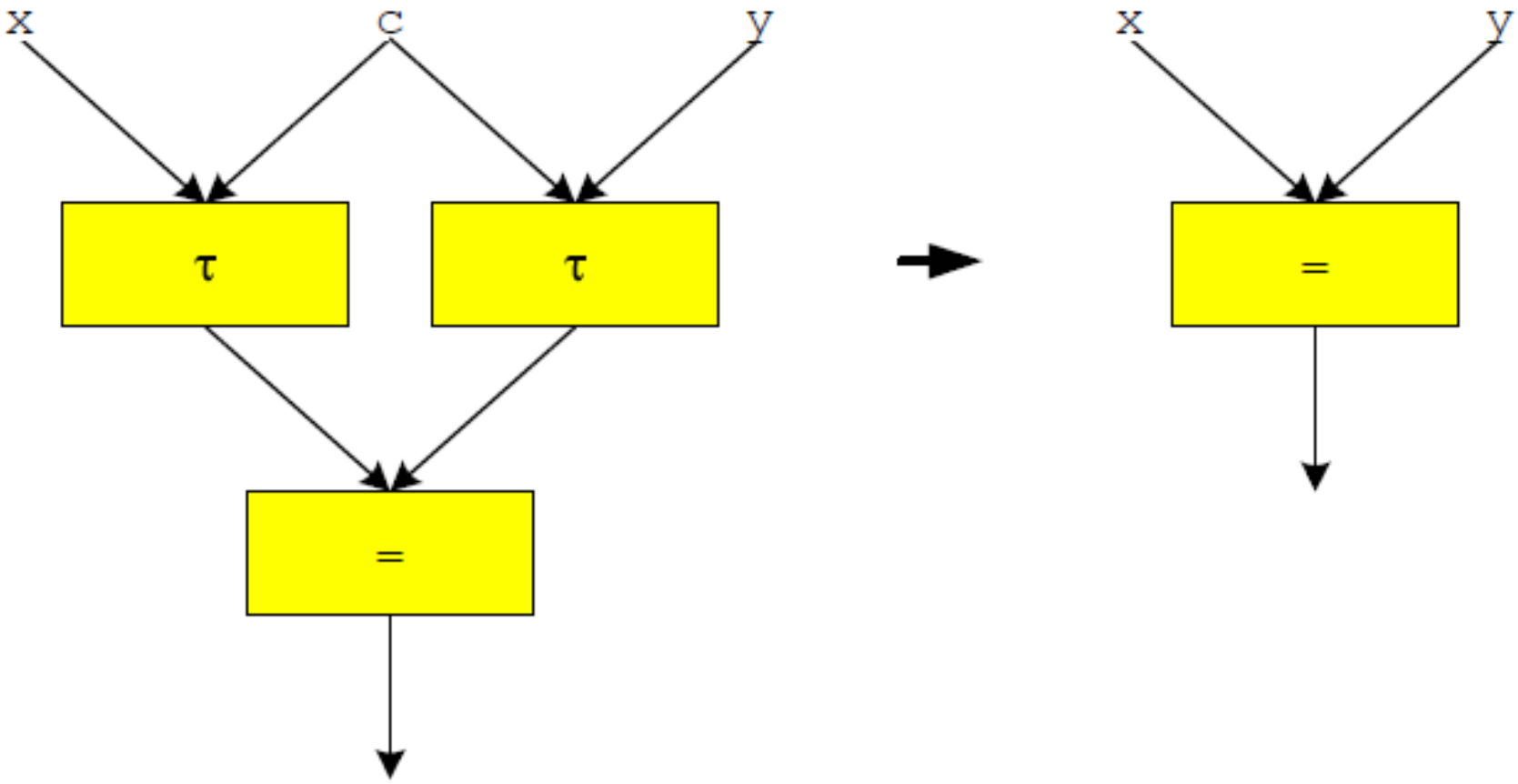
Normalizálás

- Disztributivitás:



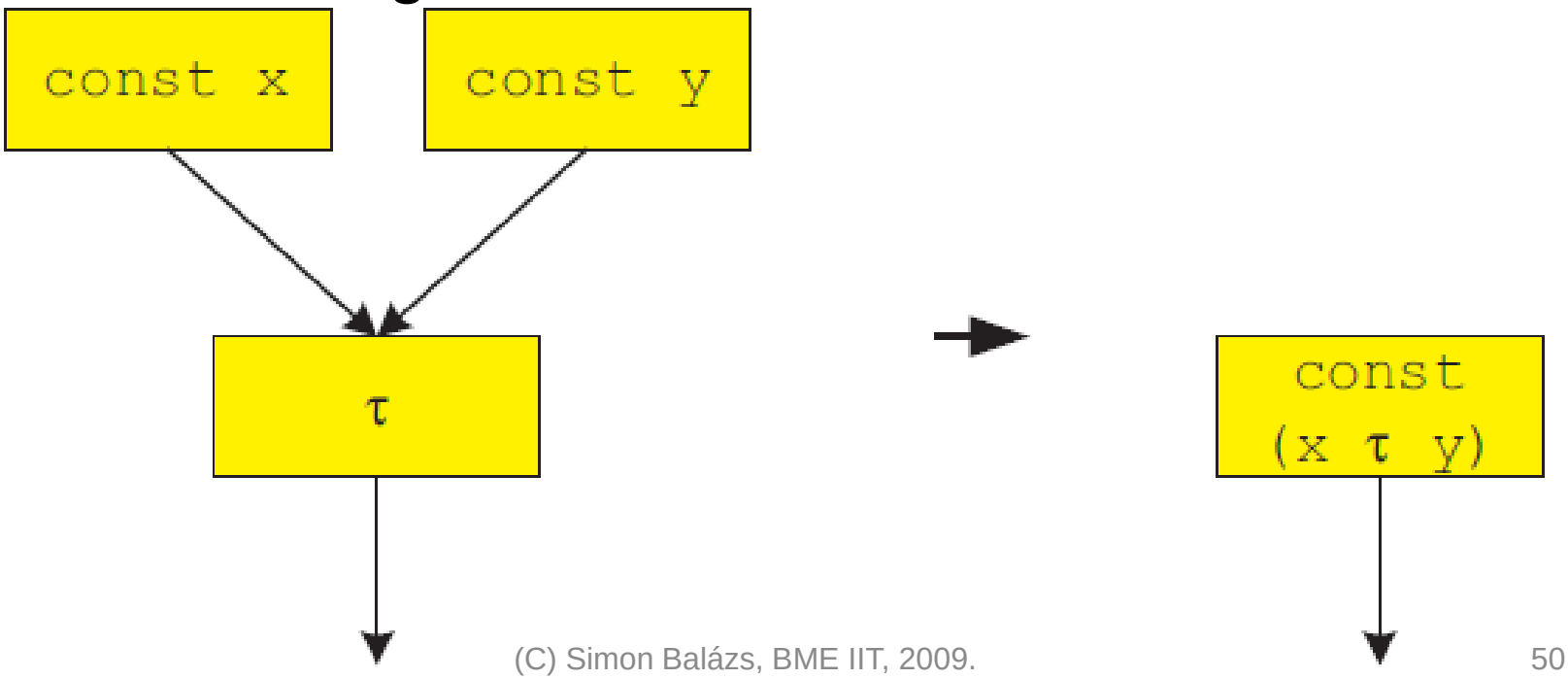
Normalizálás

- Összehasonlítás



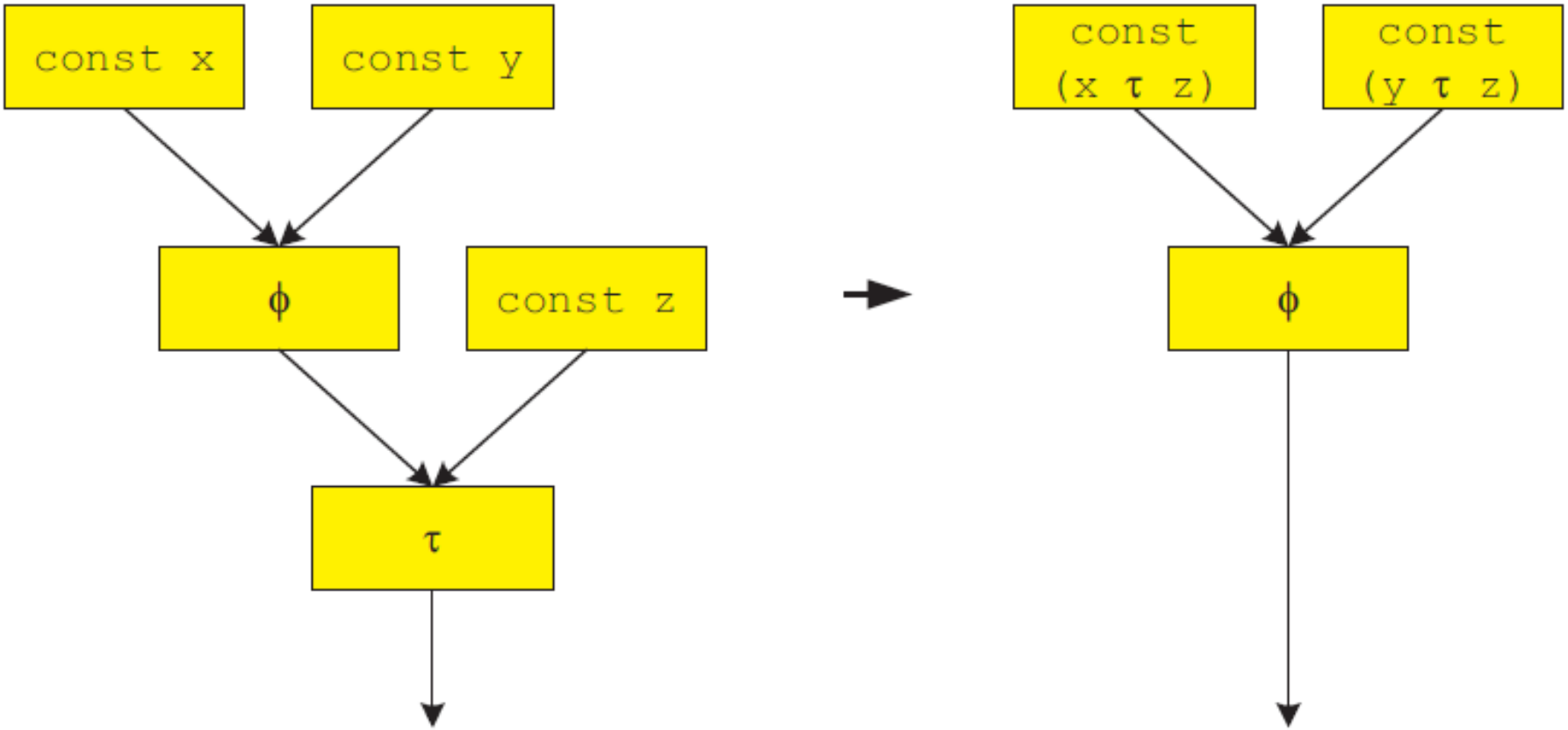
Optimalizálás

- Konstansok kiértékelése
  - a nyelv és a célgép figyelembe vételével
  - az esetek 80-90%-ában címszámításnál van rá szükség



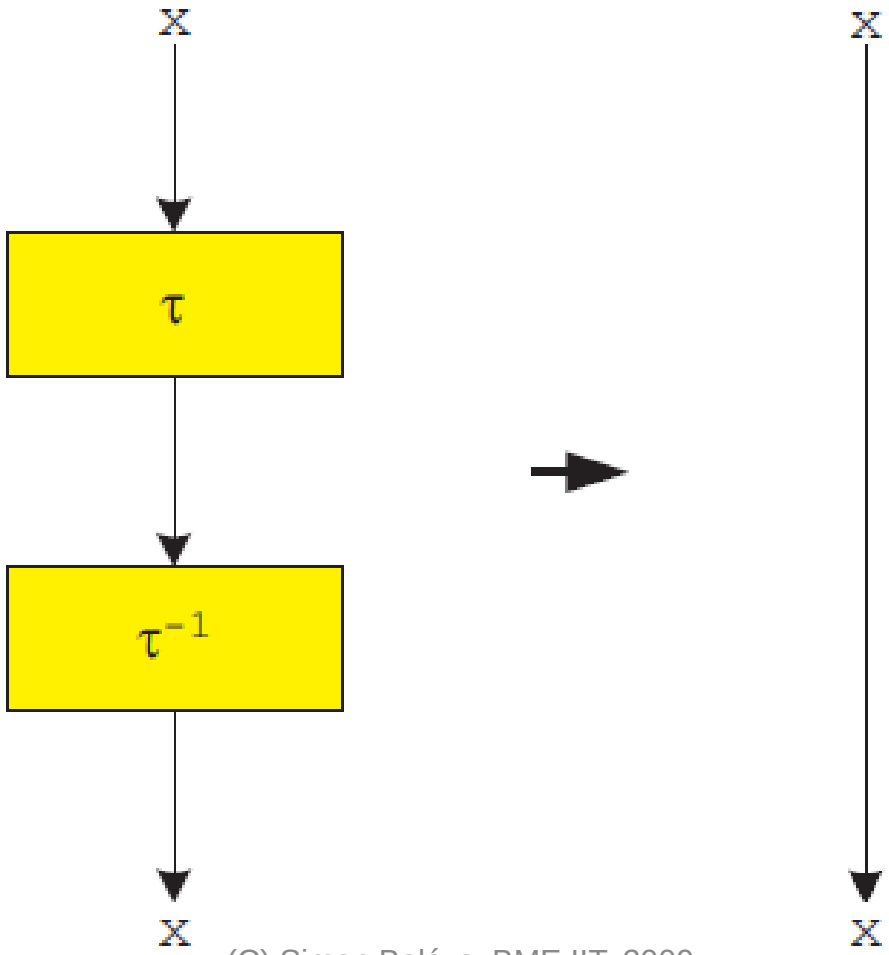
Optimalizálás

- Φ-függvényeken keresztül is működik



Optimalizálás

- Inverz operátor törlése



További optimalizálások

Operátoregyszerűsítés

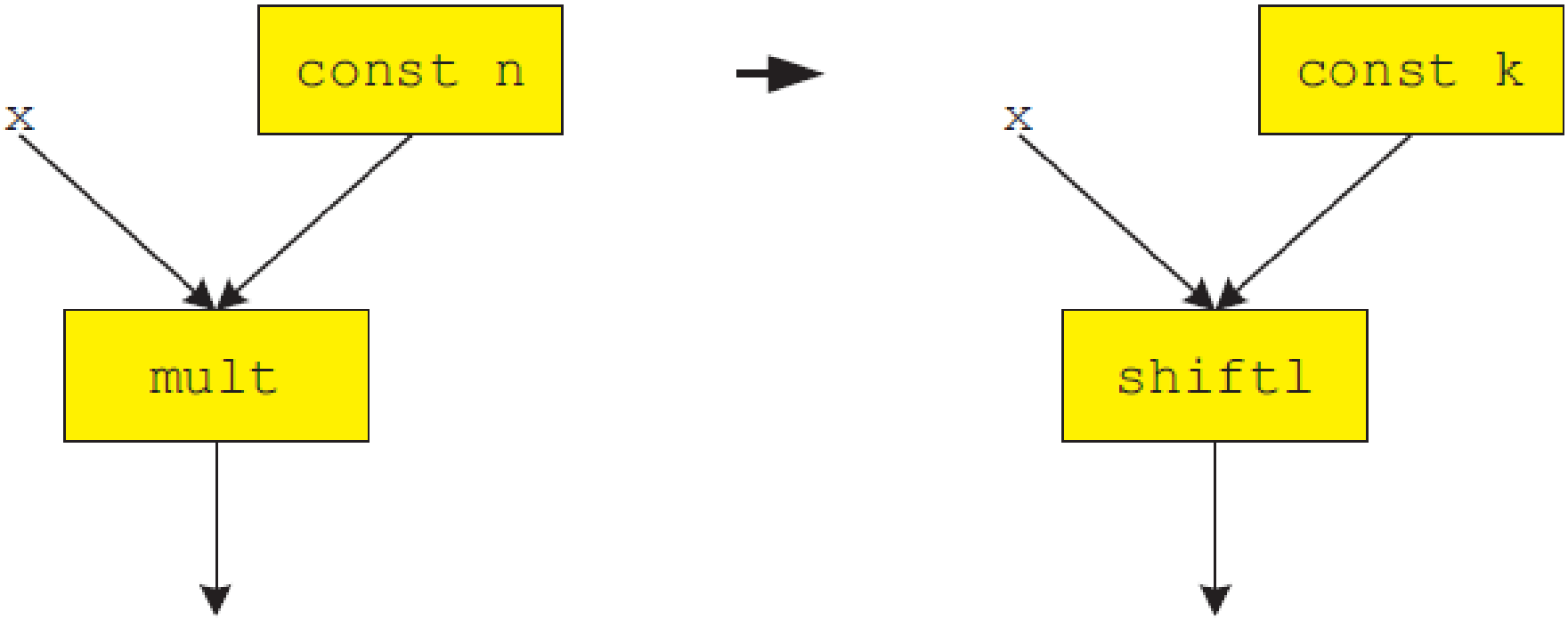
Optimalizálás

- Operátoregyszerűsítés
  - ötlet: a drága operátorok helyettesítése olcsóbb, de szemantikusan ekvivalens operátorokkal
  - fő felhasználási terület: ciklusok indexváltozóival való szorzás helyett összeadás
    - főleg ahol a szorzás sokkal költségesebb, mint az összeadás (akár 3x-os sebességnövekedés is elérhető)

Operátoregyszerűsítés

- Kettő-hatvánnyal való szorzás

$$x \times n, n = 2^k$$



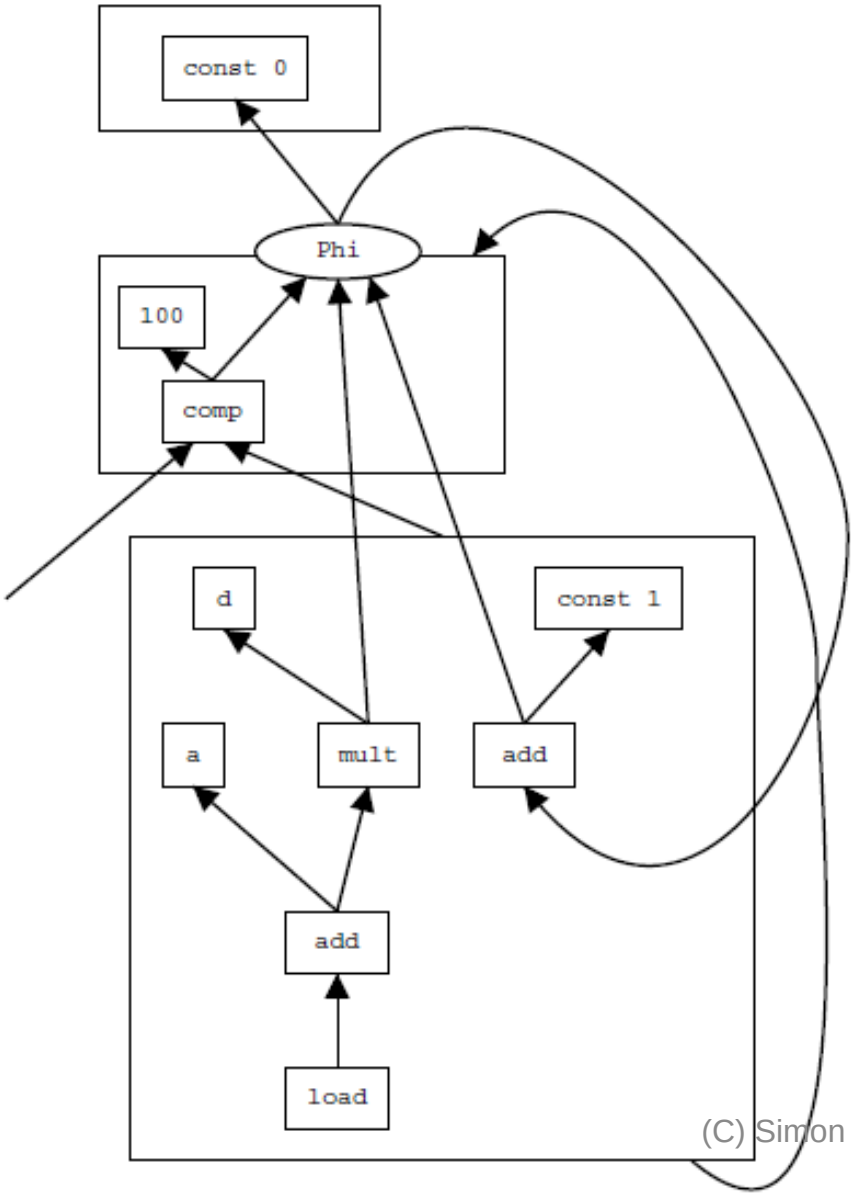
Operátoregyszerűsítés

- Gyakori a következő kód:

```
for (i=0; i<n; i++)
  for (j=0; j<m; j++)
    ... a[i,j] ...;
```

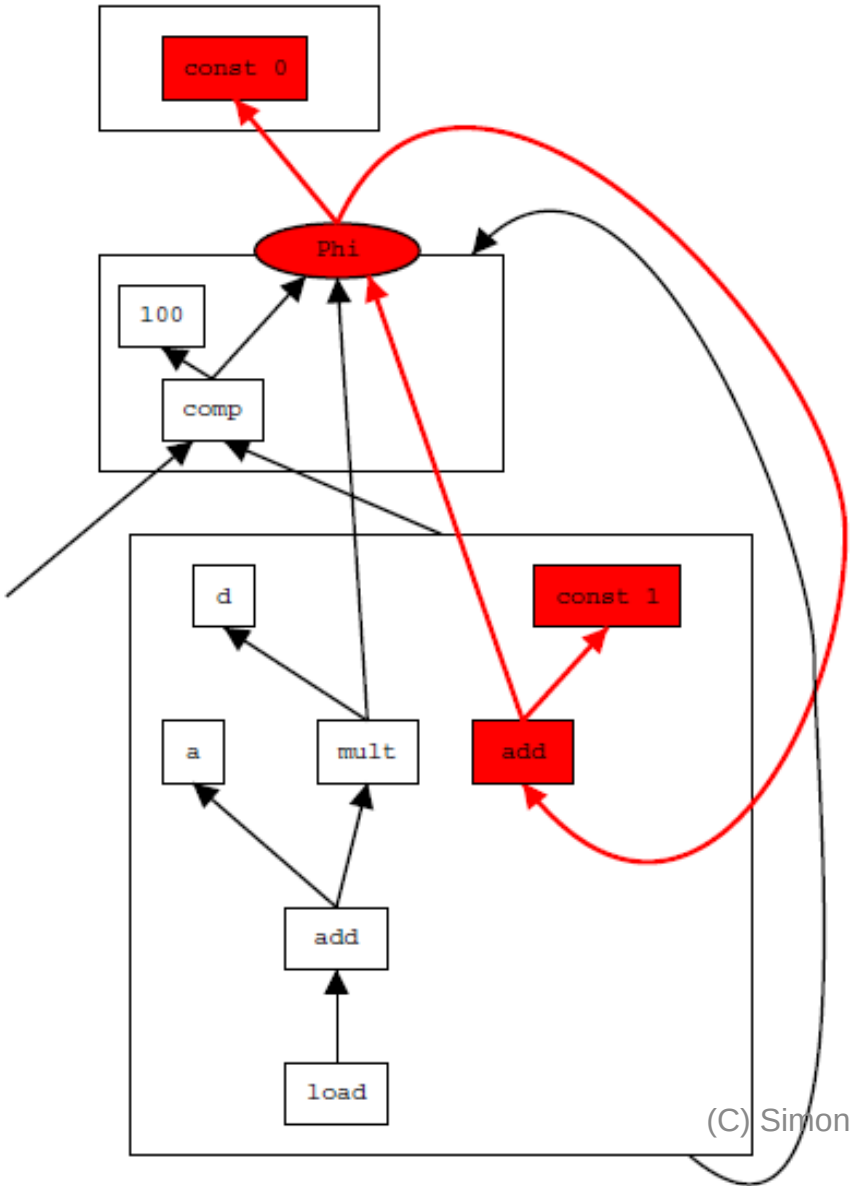
- Tipikus címszámítás:
  - **>a[i,j]< = >a[0,0]<+i*m*d+j*d**
  - ciklusonként 3 szorzás és 2 összeadás
- Ötlet:
  - ciklus elején: **>adr< = >a[0,0]<**
  - minden iterációban: **adr = adr+d**
  - így minden lefutáskor: **>adr< = >a[i,j]<**
  - ciklusonként 1 összeadás

Ciklusváltozók egyszerűsítése



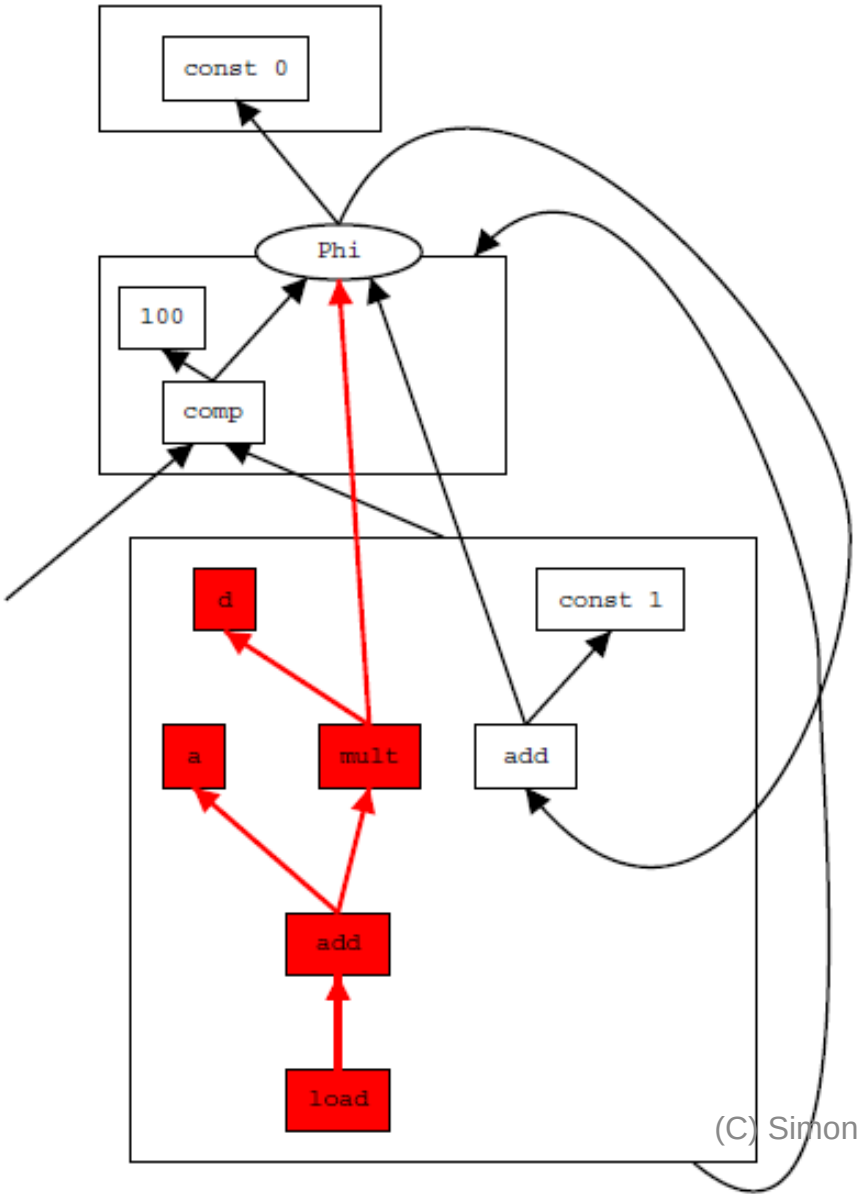
- Tipikus ciklus: címszámítással
- cím: **a+i*d**
- ciklusváltozó:
  - inicializálás: **i = 0**
  - léptetés: **i = i+1**
  - amíg: **i < 100**

Ciklusváltozók egyszerűsítése



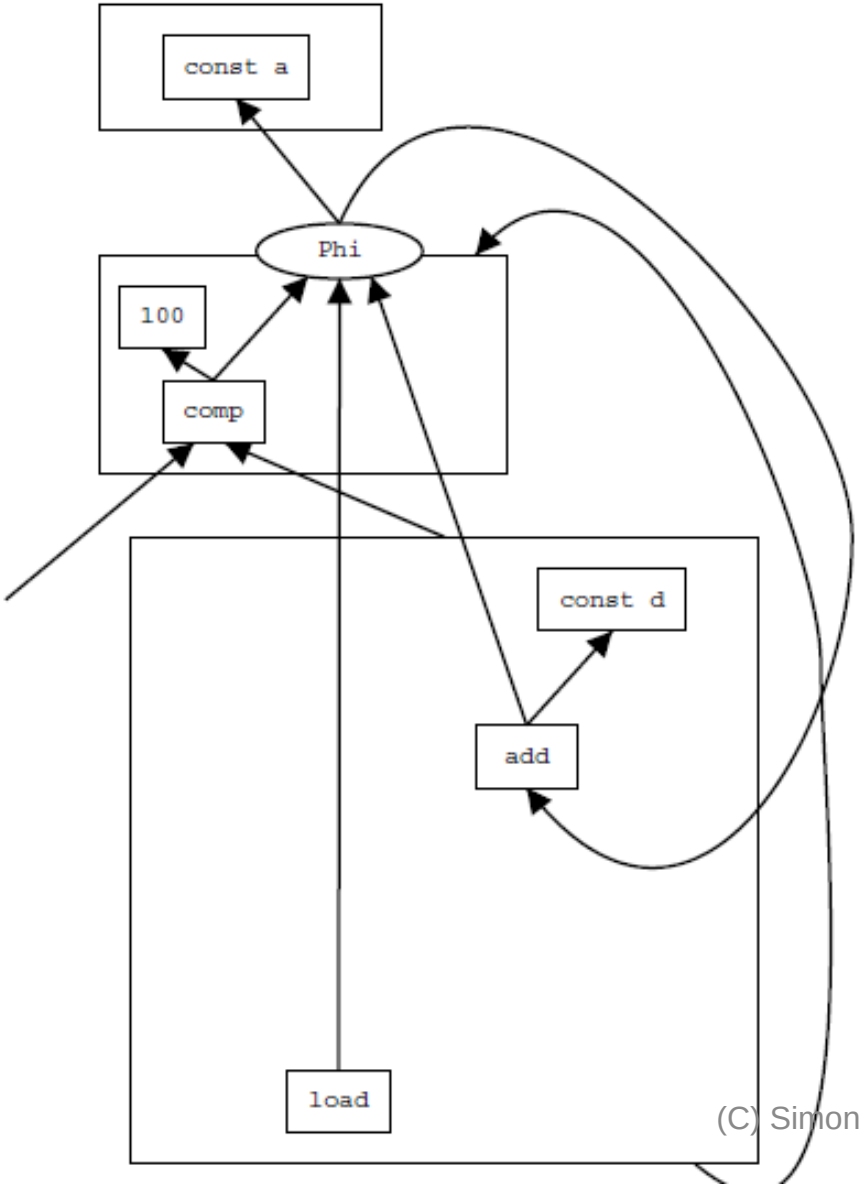
- Ciklusváltozó növelése: **i = i+1**

Ciklusváltozók egyszerűsítése



- Címszámítás:
  - ciklusváltozó szorzása az elemmérettel: **i*d**
  - hozzáadás a báziscímhez: **a**

Ciklusváltozók egyszerűsítése



- Báziscím:
  - ciklusinvariáns, kivihető
- Léptetés:
  - 1 helyett az elemmérettel
- Címszámítás:
  - a ciklusváltozó adja a címet
- Előny: ciklusonként mindössze 1 db összeadás

Összefoglalás

- Sokféle optimalizálás
- Csak néhány kiválasztásával is jó eredmények érhetők el
- Az SSA forma segítségével sok analízis és optimalizálás egyszerűsödik
- Operátoregyszerűsítés: drága operációk (pl. mul) olcsóbbra cserélése (pl. add)
- Az SSA forma fő előnye: nem csak kiindulópont, hanem egyben az optimalizálás tere is