



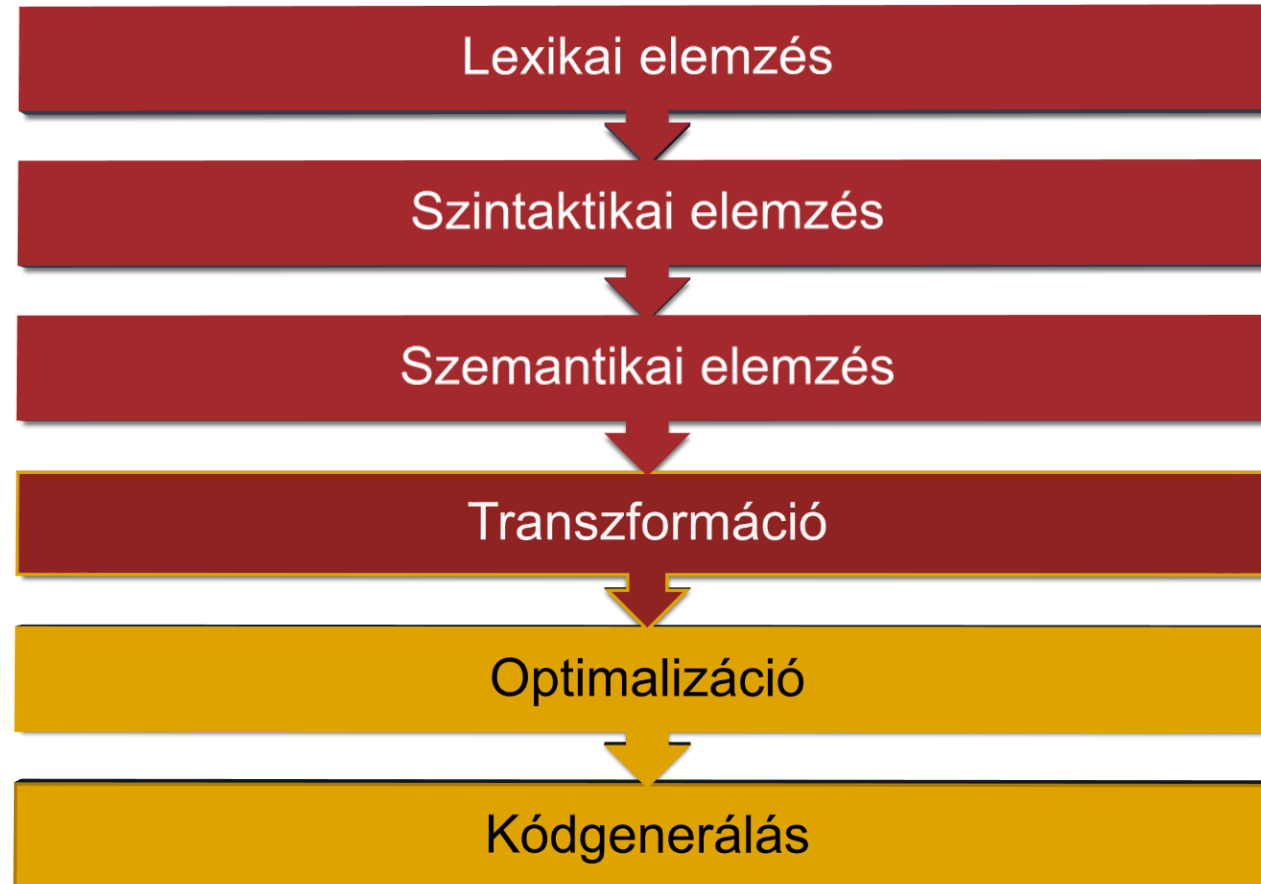
# Modellalapú szoftverfejlesztés

## VI. előadás

Optimalizáció,  
Obfuscáció,  
Kódgenerálás

Dr. Mezei Gergely,  
Dr. Somogyi Ferenc

# Fordítás fázisai



# Optimalizálás, obfuscálás, kódgenerálás

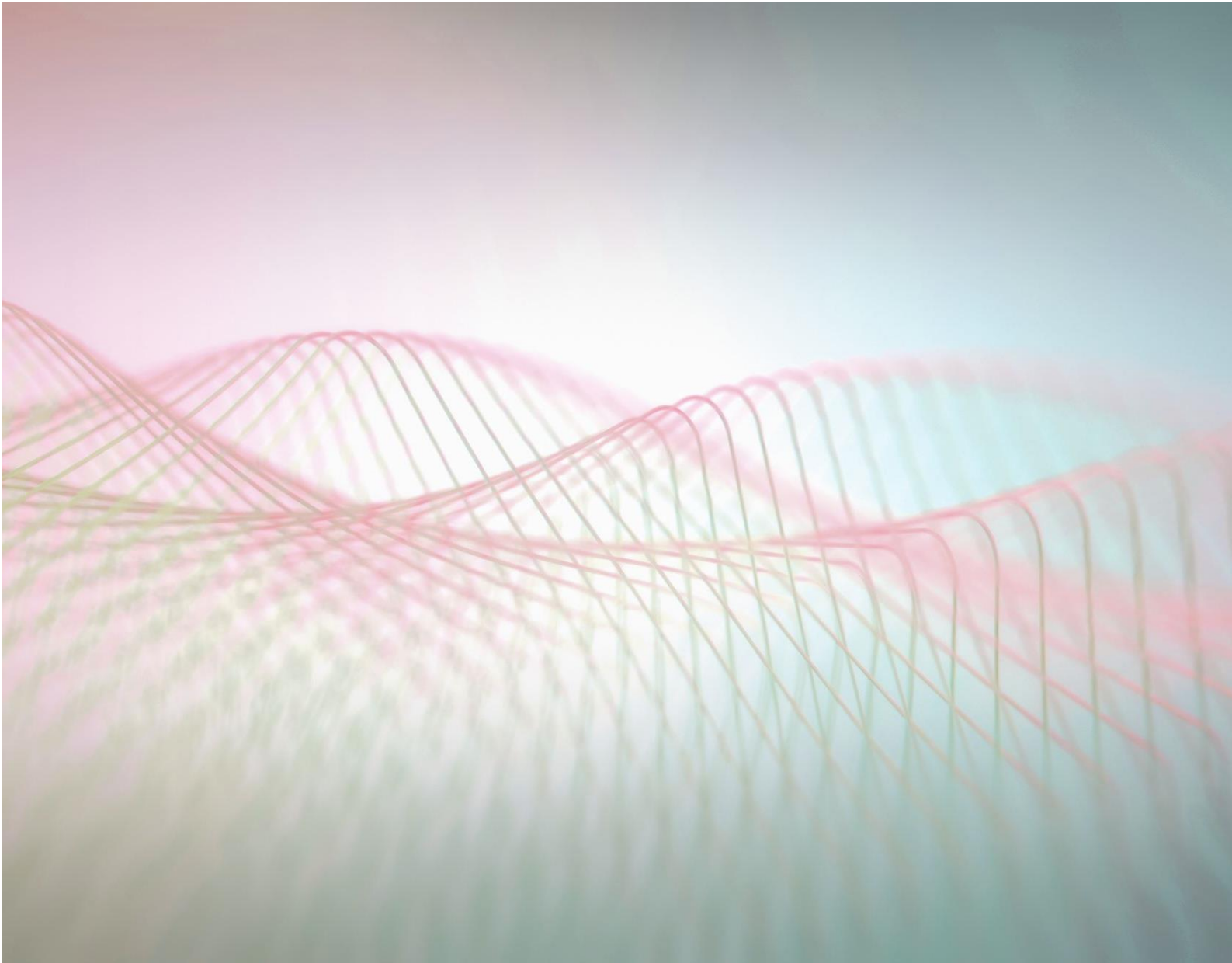
**I. Optimalizáció (folytatás)**

**II. Obfuscálás**

**III. Kódgenerálás**

**IV. Editor támogatás**





# Adatfolyam optimalizáció

# Adatfolyam optimalizáció

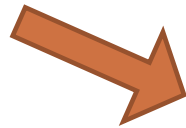
- Közös kifejezés (common subexpression)
- Konstans/változó propagáció (constant/copy propagation)
- Elérhető kifejezések
- Halott kód (dead code)
- Élő változók felderítése

**Csak alapblokkon belül (lokális)!**

# Common subexpression

- Ne számoljuk ki kétszer ugyanazt!
  - Ha az operandusok nem változtak és a művelet mellékhatás mentes

```
...  
a = b ⊙ c + 5  
e = (b ⊙ c) * 2  
...
```



```
...  
tmp = b ⊙ c  
a = tmp + 5  
e = tmp * 2  
...
```

# Constant propagation

- Ha egy változóba konstans érték kerül, akkor a változót lecserélhetjük az értékre

```
...  
a = 7  
b = a ⊗ 3  
...  
if (a > 4) { write("."); }  
...
```



```
...  
a = 7  
b = 7 ⊗ 3  
...  
write(".");  
...
```

# Copy propagation

- Ha két változó értéke megegyezik, kicserélhetőek

```
...  
a = b ⊙ c  
d = a  
e = d * 5 + 8  
...
```



```
...  
a = b ⊙ c  
d = a  
e = a * 5 + 8  
...
```



# Optimalizáció - példa

```
...  
b = a * a;  
c = a * a;  
d = b + c;  
e = 1;  
f = b + b;  
g = f * e;  
...
```

# Elérhető kifejezések

- *Elérhető* kifejezés (available expression): ha van aktuálisan olyan változó, ami a kifejezés értékét tartalmazza, akkor a kifejezés lecserélhető a változóra
- Elérhető kifejezések felderítése
  - > Kezdetben üres halmaz
  - >  $a = b \odot c$  kifejezésnél
    - Az  $a$ -t tartalmazó kifejezéseket kivesszük
    - $a = b \odot c$  kifejezést betesszük

# Elérhető kifejezések alkalmazása

```
{ }
```

```
a = b;
```

```
{ a = b }
```

```
c = a;
```

```
{ a = b, c = b }
```

```
d = a + b;
```

```
{ a = b, c = b, d = a + b }
```

```
e = d;
```

```
{ a = b, c = b, d = a + b, e = a + b }
```

```
d = a;
```

```
{ a = b, c = b, d = b, e = a + b }
```

```
f = e;
```

```
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

# Dead code

- Ha egy értékadás bal oldalán szereplő változó értékét sehol nem olvassuk ki, akkor az értékadás törölhető

```
...  
a = b * c  
d = a  
e = a * 5 + 8  
...
```



```
...  
a = b * c  
d = a  
e = a * 5 + 8  
...
```

# Élő kód felderítése

- Dead code felderítés: **liveness analysis**
- Egy változó élő (live) a program egy pontján, ha a később következő kódban előbb olvassák ki az értékét legalább egyszer, minthogy felülírnák azt
- Dead code szűrés ennek megfelelően:
  - > Minden változóra liveness számítás
  - > Minden olyan értékadás törlése, ami nem élő változónak ad értéket
- Fordított sorrendben dolgozzuk fel az alablokk utasításait
- Néhány változó alapesetben élőknek számít (pl. kimenetet befolyásolja)

# Élő kifejezések

$\{ b \}$

$a = b;$

$\{ a, b \}$

$\{ a, b \}$

$d = a + b;$

$\{ a, b, d \}$

$e = d;$

$\{ a, b, e \}$

$d = a;$

$\{ b, d, e \}$

$\{ b, d \}$

# Élő kifejezések

{ *b* }

a = b;

{ *a, b* }

d = a + b;

{ *a, b, d* }

{ *a, b* }

d = a;

{ *b, d* }

# Élő kifejezések

$\{ b \}$

$a = b;$

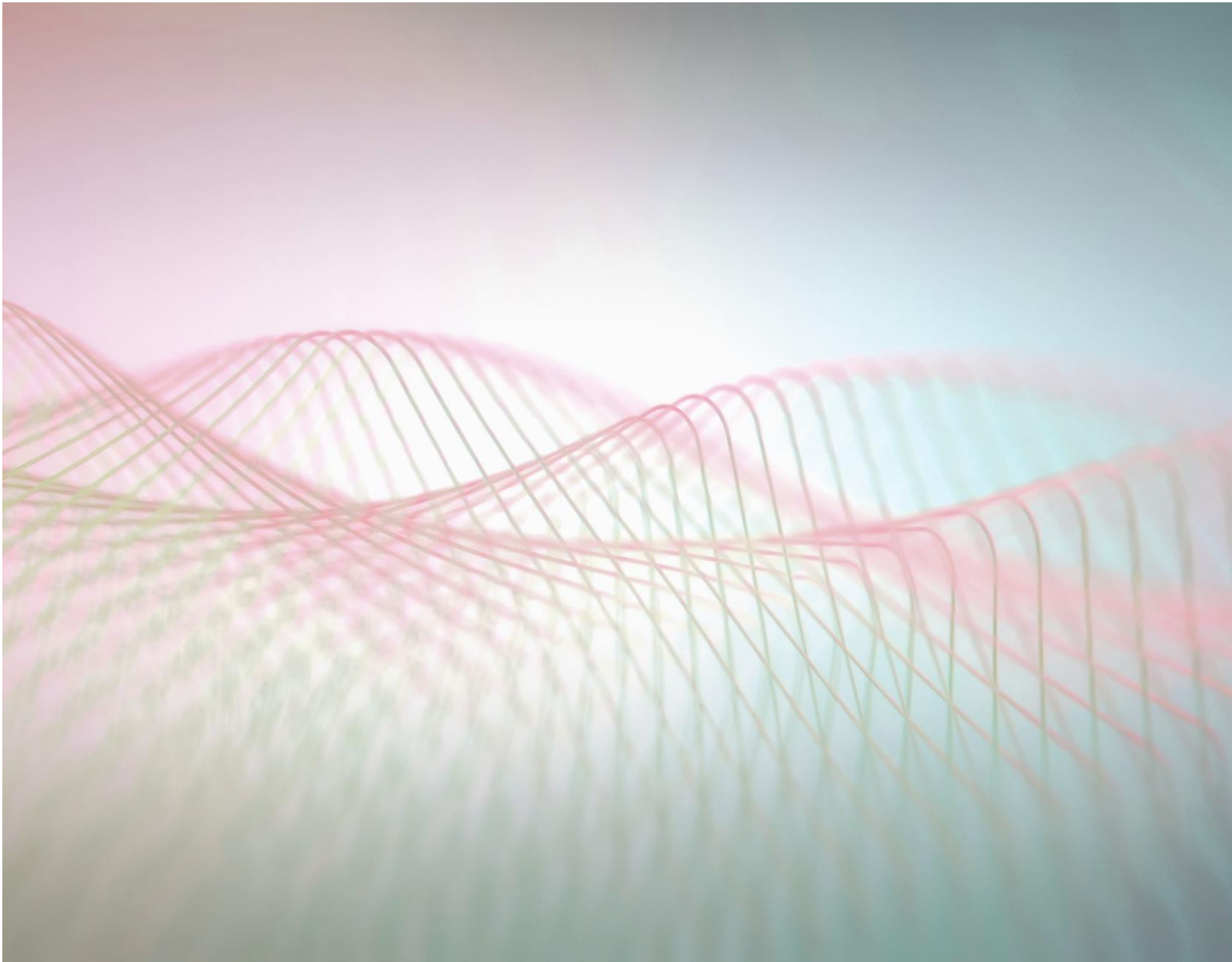
$\{ a, b \}$

$\{ a, b \}$

$d = a;$

$\{ b, d \}$





# Kód áthelyezés

# Kódáthelyezés (code motion)

- Kódsüllyesztés (code sinking)
- Kódfaktorizálás (code factoring)
- Kódütemezés (code scheduling)
- Invariáns ciklusrészlet mozgatás (loop invariant code motion)

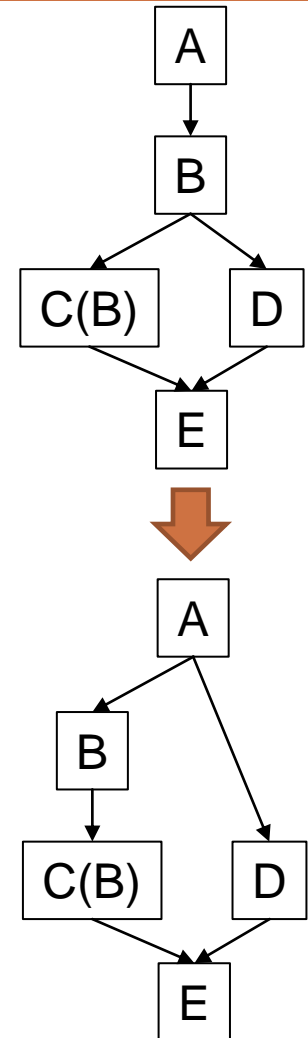
# Kódsüllyesztés

- Az utasításokat csak ott hajtjuk végre, ahol szükség van rájuk
  - > Hasonlít a dead code-ra, de mellékhatással járó utasításoknál is használható

```
...  
  b=5;  
  if (a>5) { write(b); } else { write("none"); }  
...
```



```
...  
  if (a>5) { b=5; write(b); } else { write("not ok"); }  
...
```



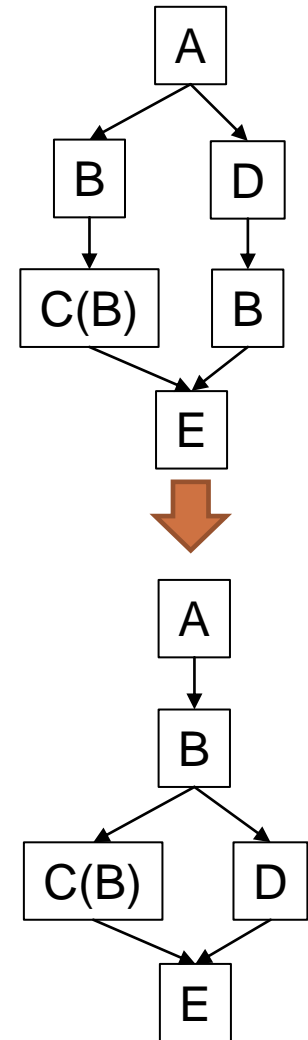
# Kódfaktorizálás

- Közös utasításokat a közös ágban hajtjuk végre
  - > A kódsüllyesztés “ellentéte”
  - > Gyakran kisebb részekre bontjuk a kódot (faktorizáljuk)

```
...  
if (a>b) { c=5; write(c); } else { write("none"); c=5; }  
...
```



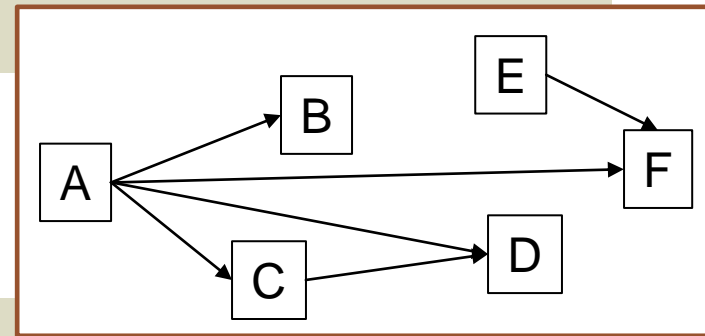
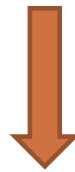
```
...  
c=5;  
if (a>b) { write(c); } else { write("none"); }  
...
```



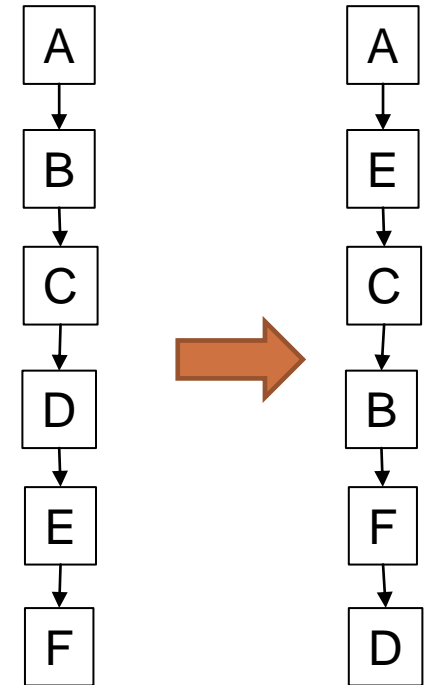
# Kódütemezés

- Kerüljük el a hosszú várakozási sorokat!
  - > Párhuzamos/hatékony végrehajtás gyorsabb más sorrendben
  - > Függőségi gráf optimalizálása

```
...  
a=f(); b=a+1; c=f2(a); d=a+c; e=f3(); f=e+a;  
...
```



```
...  
a=f(); e=f3(); c=f2(a); b=a+1; f=e+a; d=a+c;  
...
```



# Invariáns ciklusrészlet mozgatás

- Emeljük ki az invariáns kódokat a ciklus elé!
  - > Ne számoljuk ki minden körben, feleslegesen

```
...  
int i = 0;  
while (i < j) {  
    a = b + c;  
    d[i] = i + a * a;  
    ++i;  
}  
...
```



```
...  
int i = 0;  
a = b + c;  
while (i < j) {  
    d[i] = i + a * a;  
    ++i;  
}  
...
```

# Invariáns ciklusrészlet mozgatás

- Emeljük ki az invariáns kódokat a ciklus elé!
  - > Kiemelhetünk részben invariáns kódokat is

```
...  
int i = 0;  
while (i < j) {  
    a = b + c;  
    d[i] = i + a * a;  
    ++i;  
}  
...
```



```
...  
int i = 0;  
a = b + c;  
int const a' = a * a;  
while (i < j) {  
    d[i] = i + a';  
    ++i;  
}  
...
```

# Invariáns ciklusrészlet mozgatás

- Emeljük ki az invariáns kódokat a ciklus elé!
  - > Kezelni kell, hogy nem biztos, hogy egyáltalán belelép a ciklusba a program

```
...  
int i = 0;  
while (i < j) {  
    a = b + c;  
    d[i] = i + a * a;  
    ++i;  
}  
...
```



```
...  
int i = 0;  
if (i < j) {  
    a = b + c;  
    int const a' = a * a;  
    while (i < j) {  
        d[i] = i + a';  
        ++i;  
    }  
}  
}...
```



# Invariáns ciklusrészlet mozgatás

- Emeljük ki az invariáns kódokat a ciklus elé!
  - > Mi történik, ha az ciklus feltételnek van mellékhatása?

```
...  
int i = 0;  
while (fn(i)) {  
    a = b + c;  
    d[i] = i + a * a;  
    ++i;  
}  
...
```



```
...  
int i = 0;  
if (fn(i)) {  
    a = b + c;  
    int const a' = a * a;  
    while (fn(i)) {  
        d[i] = i + a';  
        ++i;  
    }  
}...
```

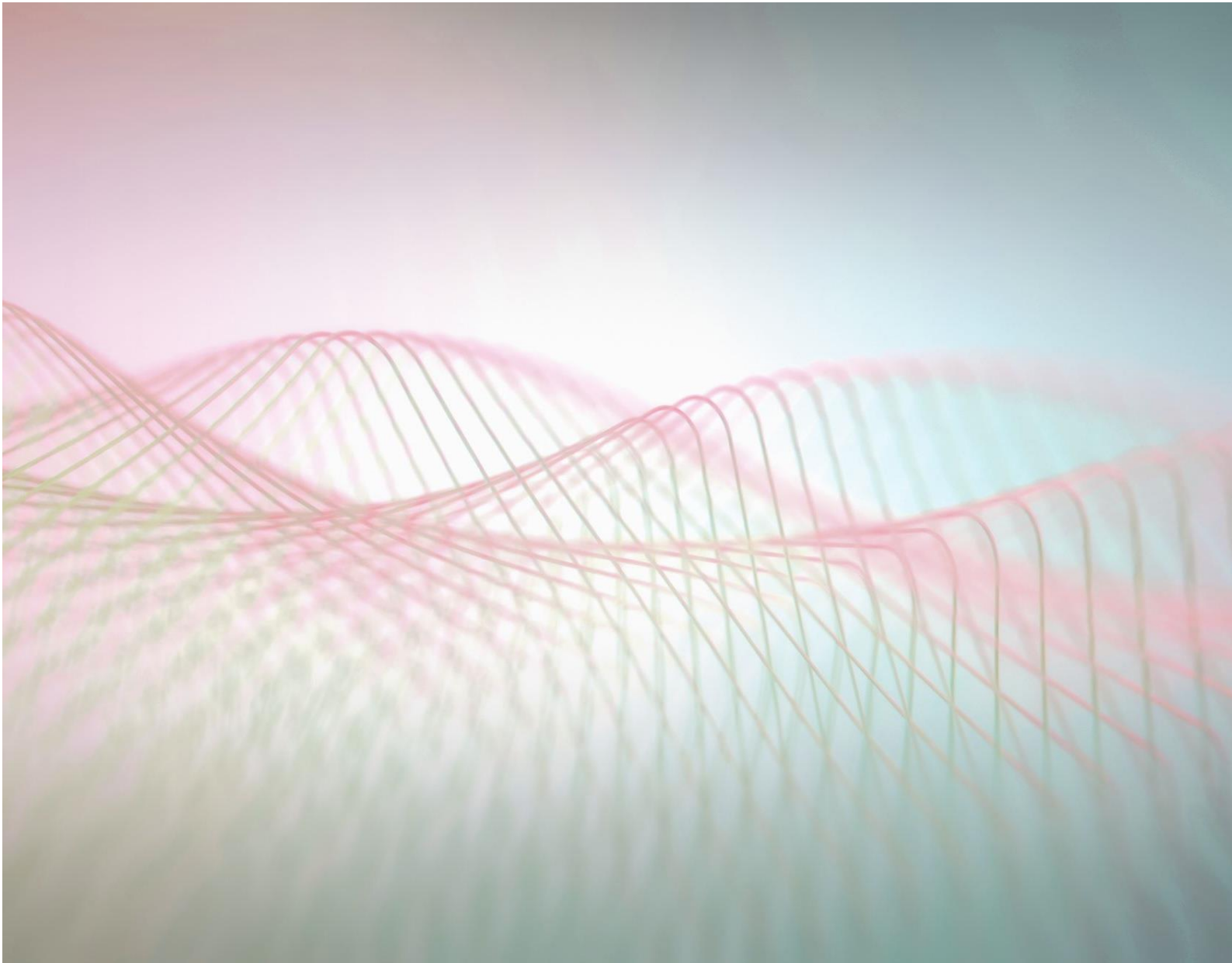
# Invariáns ciklusrészlet mozgatás

- Emeljük ki az invariáns kódokat a ciklus elé!
  - > Mi történik, ha a ciklus feltételnek van mellékhatása?

```
...  
int i = 0;  
while (fn(i)) {  
    a = b + c;  
    d[i] = i + a * a;  
    ++i;  
}  
...
```



```
...  
int i = 0;  
if (fn(i)) {  
    a = b + c;  
    int const a' = a * a;  
    do {  
        d[i] = i + a';  
        ++i;  
    } while (fn(i));  
}...
```



# További optimalizációs technikák

# Inline függvények használata

- Inline függvény – hívás helyett bemásoljuk a függvény kódját
  - > Manuális v. automatikus művelet
  - > Nem azonos a makrókkal (ott a forráskód a fordítás előtt változik)
- Előny
  - > Megspórolja a függvényhívás költségét (fv. pointerok, stack, regiszterek állítása)
  - > Lokális/globális optimalizáció scope-ja nagyobbá válik
    - Invariáns kódrészletek kiszervezése (a ciklusokhoz hasonlóan)
    - Regiszter optimalizáció
- Hátrány
  - > Nő a memóriaméret (duplikált adatszerkezet)
  - > Műveleti cache túllépés lehetséges

# Regiszterkiosztás

- Regiszter – gyors, könnyen elérhető, kis tároló
  - > Korlátozott mennyiség ( $< 32$ )
  - > Minden élő változót be kellene tölteni
  - > Ami nem fér be  $\rightarrow$  RAM
  - > Regiszterek néha nem függetlenek (pl. 32 bites használható 2x16 bitesként is)
- Mit vegyünk ki, mit tartsunk bent?
  - > NP teljes kérdés (gráf színezésre vezethető vissza, szín = regiszter)
  - > Újraszámolás (rematerialization) tárolás helyett (pl. konstans egész értékek)

# További technikák

- Elemi technikák
  - > Elérhetetlen kód eliminálása
  - > Ciklusok kibontása, összevonása
  - > Indexhatár ellenőrzések elhagyása
- Gép/op. rendszerfüggő optimalizációk (pl. utasítás választás)
- Domain-függő optimalizálás

# Optimalizálás, obfuscálás, kódgenerálás

**I. Optimalizáció**

**II. Obfuscálás**

**III. Kódgenerálás**

**IV. Editor támogatás**



# Obfuszkáció

- Művelet, ami megnehezíti a program
  - > Működésének megértését
  - > Visszafejtését (gépi/köztes kódból)
  - > Elemzését
- Főként védekezésként
  - > Biztonsági hibák ellen
  - > Lopás / másolás ellen
- Csak megnehezíteni tudjuk a visszafejtést, meggátolni nem!
- Hogyan működik?
  - > Hasonlóan az optimalizációhoz
  - > Erősen függ a lehetőségektől (idő, visszafejthetetlenség fontossága, stb.)



# Technikák

- Név obfuszkáció (lexikai átalakítás)
- Adat obfuszkáció (adatszerkezetek módosítása)
- Control flow obfuszkáció
- Debug információ obfuszkálás

# Név obfuszkálás

- Beszédes azonosítók (osztályok, metódusok, változók, függvények, stb.) cseréje értelmetlen szövegre
- Korlátozások
  - > Beépített osztályok, API-k neve fix
  - > Szerializálható osztályok neve fix
  - > Natív elérés és reflection esetén is nagyon trükkös

# Adat obfuszkálás

- Megváltoztatja az adatok tárolásának a módját a memóriában
- Módszerek
  - > Kódolás (encoding) módjának változtatása
  - > Adataggregálás (tömbök, kollekciók)
  - > Megváltoztatni az adat szerepét (pl. lokális vs globális)

## Példa: név és adat obfuszkálás

```
function foo( arg1)
{
    var myVar1 = "some string"; //first comment
    var intVar = 24 * 3600; //second comment
    /* here is
       a long multi-line comment . . . */
    document.write( "vars are:" + myVar1 + " " + intVar + " " + arg1) ;
} ;
```



```
function z001c775808( z3833986e2c) { var z0d8bd8ba25=
"\x73\x6f\x6d\x65\x20\x73\x74\x72\x69\x6e\x67"; var z0ed9bcbcc2=
(0x90b+785-0xc04)* (0x1136+6437-0x1c4b); document.write(
"\x76\x61\x72\x73\x20\x61\x72\x65\x3a"+ z0d8bd8ba25+ "\x20"+
z0ed9bcbcc2+ "\x20"+ z3833986e2c);};
```

# Példa: név és adat obfuszkálás

## ■ Változók, azonosítók:

- > foo → z001c775808
- > arg1 → z3833986e2c
- > myvar1 → z0d8bd8ba25
- > intvar → z0ed9bcbcc2

## ■ Integer számok ábrázolása:

- > 20 → (0x90b+785-0xc04)
- > 3600 → (0x1136+6437-0x1c4b)

## ■ Kiírás:

- > "vars are" → \x76\x61\x72\x73\x20\x61\x72\x65\x3a
- > Space → \x20

```
function foo( arg1)
{
    var myVar1 = "some string";
    //first comment
    var intVar = 24 * 3600;
    //second comment
    /* here is
       a long multi-line
    comment ... */
    document.write( "vars are:" +
myVar1 + " " + intVar + " " +
+ arg1) ;
} ;
```

```
function z001c775808( z3833986e2c) { var z0d8bd8ba25=
"\x73\x6f\x6d\x65\x20\x73\x74\x72\x69\x6e\x67"; var z0ed9bcbcc2=
(0x90b+785-0xc04)* (0x1136+6437-0x1c4b); document. write(
"\x76\x61\x72\x73\x20\x61\x72\x65\x3a"+ z0d8bd8ba25+ "\x20"+
z0ed9bcbcc2+ "\x20"+ z3833986e2c);};
```

# Control Flow obfuszkálás

- A vezérlési folyamat megváltoztatása
- Azonos eredmény, de nehezebben érthető kód
- Módszerek
  - > Módosított vezérlés, pl. inline metódusok hívása metódushívás helyett
  - > Utasítássorrend megváltoztatása
  - > Számítások átalakítása
    - Elérhetetlen kód beszúrása
    - Feltétel nélküli ugrások elhelyezése
    - Utasításblokkok kettévágása feltételes utasítássá
    - Mindig igaz/hamis elágazásblokkok

## Példa: Control Flow obfuszkálás

```
public int CompareTo(Object o)
{
    int n = occurrences - ((WordOccurrence)o).occurrences;
    if (n == 0)
    {
        n = String.Compare(word, ((WordOccurrence)o).word);
    }
    return(n);
}
```



```
public virtual int _a(Object A_0)
{
    int local0; int local1;
    local0 = this.a - (c) A_0.a;
    if (local0 != 0) goto i0;
    goto i1;
    while (true) {
        return local1;
        i0: local1 = local0;}
    i1: local0 = System.String.Compare(this.b, (c) A_0.b); goto i0;
}
```

# Debug információ obfuszkálása

- Debug információk eltávolítása
  - > Stack trace
  - > Line number
  - > Fájl nevek, etc.



# Optimalizálás, obfuscálás, kódgenerálás

**I. Optimalizáció**

**II. Obfuscálás**

**III. Kódgenerálás**

**IV. Editor támogatás**



# Kódgenerálás

- Kódgenerálás: a fordító belső reprezentációjából (optimalizált köztes kód) futtatható kód előállítása
  - > Nem feltétlenül bináris gépi kód a cél!
- Bemenet jellege szerint
  - > 3-című kód (Three-Address Code, 3AC)
  - > Szintaxisfa
  - > stb.
- Kimenet jellege szerint
  - > Gépi kód
  - > Köztes kód (pl. LLVM, IL)
  - > Magas szintű programozási nyelvű kód (transpiler)

# Kódgenerálás - gépi kód

## ■ Kódgenerálás

- > A cél architektúra által támogatott műveletekre alakítás
- > Regiszterfoglalások menedzselése – RegisterDescriptor + getReg
- > Címadminisztráció (változók címe, változhat a futás alatt) – AddressDescriptor

## ■ **x = y op z**

- > getReg hívás az eredmény tárolásának (L) címéhez
- > Y címének felderítése (AddressDescriptor), majd másolás: **MOV y L**
- > Z címének felderítése (AddressDescriptor), majd műveletvégzés: **OP z L**
- > L-ben elérhető az eredmény, ha L regiszter, akkor x leírójába beírni, hogy L tárolja

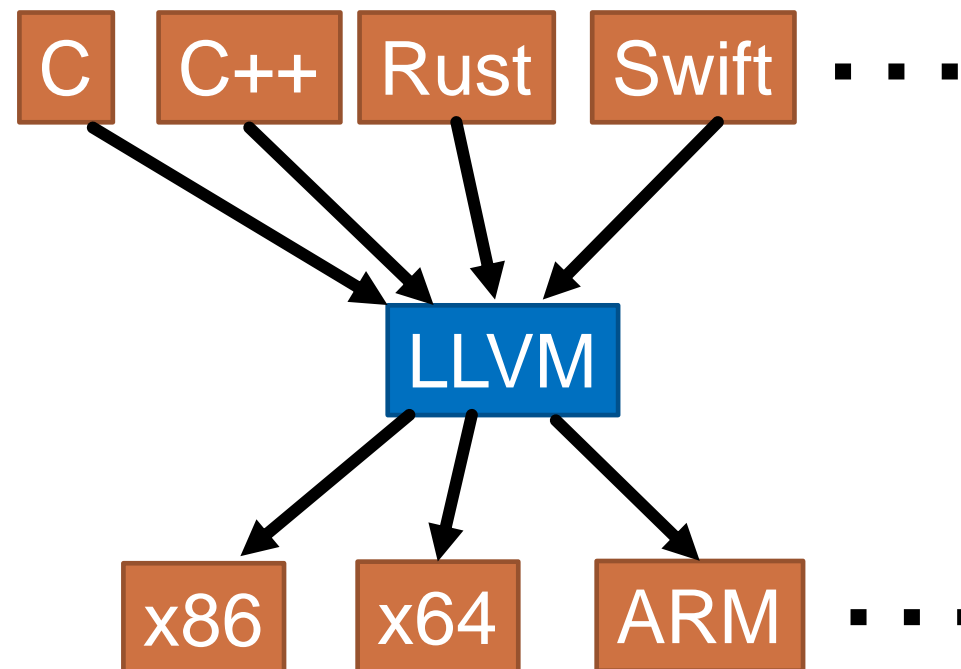
# Kódgenerálás - gépi kód

- Közvetlenül futtatható a generálás eredménye
- Előnyök
  - > Optimális gyorsaság és méret érhető el vele
  - > Gyors, teljesen ki tudja használni az architektúra sajátosságait
  - > Bármilyen leírható vele, nincsenek nyelvi korlátok
- Hátrányok
  - > Alacsonyszintű, sok munka megírni
  - > Minden ellenőrzést (pl. stack overflow) nekünk kell végezni
  - > A generált kódban a hibakeresés nehéz

# Kódgenerálás - köztes kód

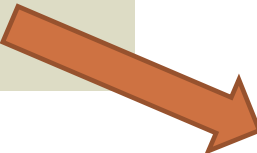
## ■ LLVM

- > Frontend: platformfüggetlen SSA utasítás lista
- > Backend: platformspecifikus bináris kód
- > ~ “olvasható assembly”
- > végtelen virtuális regiszter
- > saját platformfüggetlen típusrendszer
- > beépített optimalizálók (pl. dead code, common subexpression)



# LLVM - példa

```
function int factorial(int n){  
    if (n==0) {  
        1  
    } else {  
        n * factorial(n - 1)  
    }  
}
```



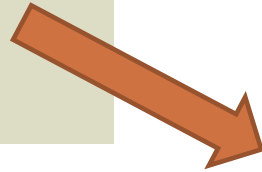
```
define i32 @factorial(i32) {  
  
entry:  
    %eq = icmp eq i32 %0, 0    // n == 0  
    br i1 %eq, label %then, label %else  
  
then:                                ; preds = %entry  
    br label %ifcont  
  
else:                                ; preds = %entry  
    %sub = sub i32 %0, 1        // n - 1  
    %2 = call i32 @factorial(i32 %sub)    // factorial(n-1)  
  
    %mult = mul i32 %0, %2      // n * factorial(n-1)  
    br label %ifcont  
  
ifcont:                              ; preds = %else, %then  
    %iftmp = phi i32 [ 1, %then ], [ %mult, %else ]  
    ret i32 %iftmp  
}
```

# Kódgenerálás - köztes kód

- Common Intermediate Language (CIL)
  - > Objektorientált (objektumok, tagfüggvények, tagváltozók)
  - > Stack-alapú (regiszterek helyett)
  - > Hardver és op. rendszer független
  - > Meta-adat tárolására képes
- Just-in-Time (JIT) compiler futtatja
  - > Ahead-of-time fordítás is elérhető
  - > Biztonsági ellenőrzések
- Disassembler segítségével magas szintű programnyelvekre is alakítható

# IL - példa

```
function int factorial(int n){  
    if (n==0) {  
        1  
    } else {  
        n * factorial(n - 1)  
    }  
}
```



```
.method public hidebysig  
instance int32 factorial (int32 n) cil managed {  
    // Method begins at RVA 0x2064  
    // Code size 31 (0x1f)  
    .maxstack 4  
    .locals init ([0] bool,[1] int32)  
  
    // {  
    IL_0000: nop  
  
    // if (n == 0)  
    IL_0001: ldarg.1  
    IL_0002: ldc.i4.0  
    IL_0003: ceq  
    IL_0005: stloc.0  
    IL_0006: ldloc.0  
    IL_0007: brfalse.s IL_000e
```

```
// return 1;  
IL_0009: nop  
IL_000a: ldc.i4.1  
IL_000b: stloc.1  
  
// (no C# code)  
IL_000c: br.s IL_001d  
  
// return n * factorial(n - 1);  
IL_000e: nop  
IL_000f: ldarg.1  
IL_0010: ldarg.0  
IL_0011: ldarg.1  
IL_0012: ldc.i4.1  
IL_0013: sub  
IL_0014: call instance int32  
                                Host::factorial(int32)  
  
IL_0019: mul  
IL_001a: stloc.1  
  
// (no C# code)  
IL_001b: br.s IL_001d  
IL_001d: ldloc.1  
IL_001e: ret  
} // end of method Host::factorial
```



# Kódgenerálás - köztes kód

## ■ Előnyök

- > Könnyebb generálni, mint gépi kódot, olvashatóbb emberi felhasználásra
- > Kevesebb technikai nehézség (pl. regiszter allokáció)
- > Hatékony tud lenni, mivel alacsonyszintű
- > Több platformra fordítható

## ■ Hátrányok

- > Kell hozzá egy speciális compiler
- > A célkód (köztes kód) előállítása nem egyszerű

# Kódgenerálás - transpiler

- Transpiler: más, jellemzően magas absztrakciós szintű célnyelvre fordítunk
  - > Bejárjuk az annotált szintaxisfát és tipikusan sablonnal kódot generálunk
  - > Tipikus megoldás: minden AST elem képes a saját kódját előállítani, hierarchikus dekompozíció
  - > Több fordító is “sorba köthető” egymás után

```
if (x > 5) then  
begin  
  while (y < z) do  
  begin  
    y := x;  
  end  
end
```



```
if (x > 5)  
  while (y < z) {  
    y = x;  
  }
```

# Kódgenerálás - transpiler

## ■ Előnyök

- > Könnyű megírni a kódgenerátort
- > A gépi kód a célnyelvű, generált forrás fordításával áll elő
  - Rábízhatjuk magunkat a célnyelv fordítójára
  - Külső kód optimalizálás (pl. Visual Studio, GCC)
  - Biztonsági ellenőrzések (pl. memória túlcímzés)
- > Könnyebb ellenőrizni a generált kód szemantikáját (C# vs. Assembly)
- > A generált kódot könnyű integrálni meglévő alkalmazásokba/eszközökbe

## ■ Hátrányok

- > Nem lehet annyira optimalizálni, mint a gépi kódot
- > A célnyelv képességei korlátozzák a lehetőségeket

## Kódgenerálás után: linkelés, buildelés

- Kódgenerálás kimenete nem mindig futtatható magában: több modul ('object' file)
- Linkelés: az object fájlok összefűzése futtatható állománnyá  
Buildelés: az összelinkelt fájlok futtatható állománnyá alakítása
- Linkelés - előnyök
  - > Egyetlen monolitikus állomány helyett több kisebb (kevésbé komplex)
  - > Egyszerűbb hibakezelés és inkrementális fordítás
  - > Lefordított object fájlok újrahasznosítása

# Optimalizálás, obfuscálás, kódgenerálás

**I. Optimalizáció**

**II. Obfuscálás**

**III. Kódgenerálás**

**IV. Editor támogatás**



# Szöveges szerkesztők (Editorok)

- Szöveges nyelvek – jó tool support
  - Írhatnánk Notepad-ben is, de nem ajánlott 😊
  - Vannak kiforrottabb szövegszerkesztők (Pl. Vim, Emacs, Atom)
    - De ezeket összekötni a nyelvünkkel nem túl egyszerű
  - Fejlesztői környezetek (IDE)  
(Pl. VSCode, Eclipse, Eclipse Theia, IntelliJ IDEA, NetBeans)
    - Ezek közül több kiterjeszthető (plugin, extension, stb.), ami megkönnyíti az összekötést
- Nyelv és editor összekötése
  - A fordítás fázisaival összhangban (elsősorban szemantikai elemzésnél)



# Szöveges szerkesztők (Editorok)

- Editor funkciók
  - Syntax highlighting
  - Code completion
  - Hibák jelzése
  - Refaktorálás támogatása
  - Folding
  - Quick fixek
  - Stb.

```
1 //app/database.ts
2 import { Tweet, TweetWithId } from "../models";
3 import { v4 as uuidv4 } from 'uuid';
4
5 function generateTweetId() : string {
6   return uuid();
7 }
8
9 > export class Database { ...
40 }
```

```
1 Device Window can be OPEN, SHUT
2 Device Heating can be ON, OFF
3
4 Rule 'Close Window, when heating turned on'
5 when Heating.ON
6 then Window.SHUT
7
8 Rule 'Switch off heating, when windows gets opened'
9 when Window.OPEN
10 then Heating.OFF
```

Heating.OFF	State
Heating.ON	State
Window.OPEN	State
Window.SHUT	State

Forrás:

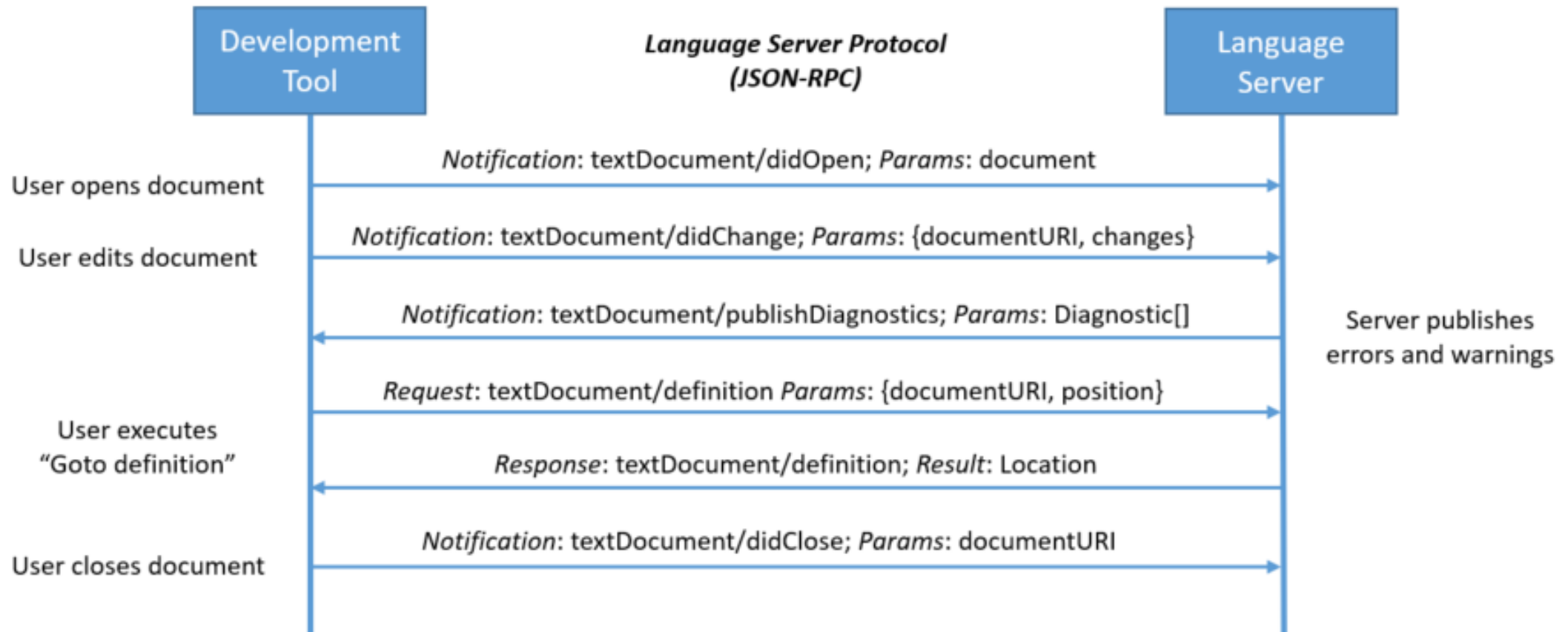
<https://www.eclipse.org/Xtext/>

# Language Server Protocol (LSP)

- JSON-RPC alapú protokoll
- Szöveges editorok és language serverek közötti kommunikáció
- Főbb editor funkciók támogatása
  - Syntax highlighting, hibák jelzése, code completion, refaktorálás, stb.
- Miért jó ez nekünk?
  - Egyszer kell megírni, több editorral is összeköthető
  - Sok népszerű editor támogatja (Pl. VSCode, Monaco, Eclipse IDE, Eclipse Theia)
  - Sok nyelvhez van létező Language Server implementáció
    - <https://langserver.org/>



# Language Server Protocol (LSP)



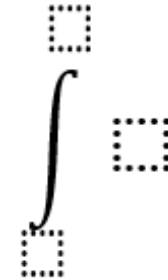
Forrás: <https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/>

# Projekciós Editorok

- Hagyományos editorok
  - Amikről eddig szó volt
  - Bevált, sok helyen használják
  - Bármilyen szövegszerkesztő használható (ajánlott editor funkciókkal)
- Projekciós editorok
  - A szintaxisfa elemei közvetlenül vizualizálva vannak
  - *Nem kell szintaxisfát építeni!*
  - Speciális tooling kell hozzá
  - Pl. JetBrains MPS, Gentleman

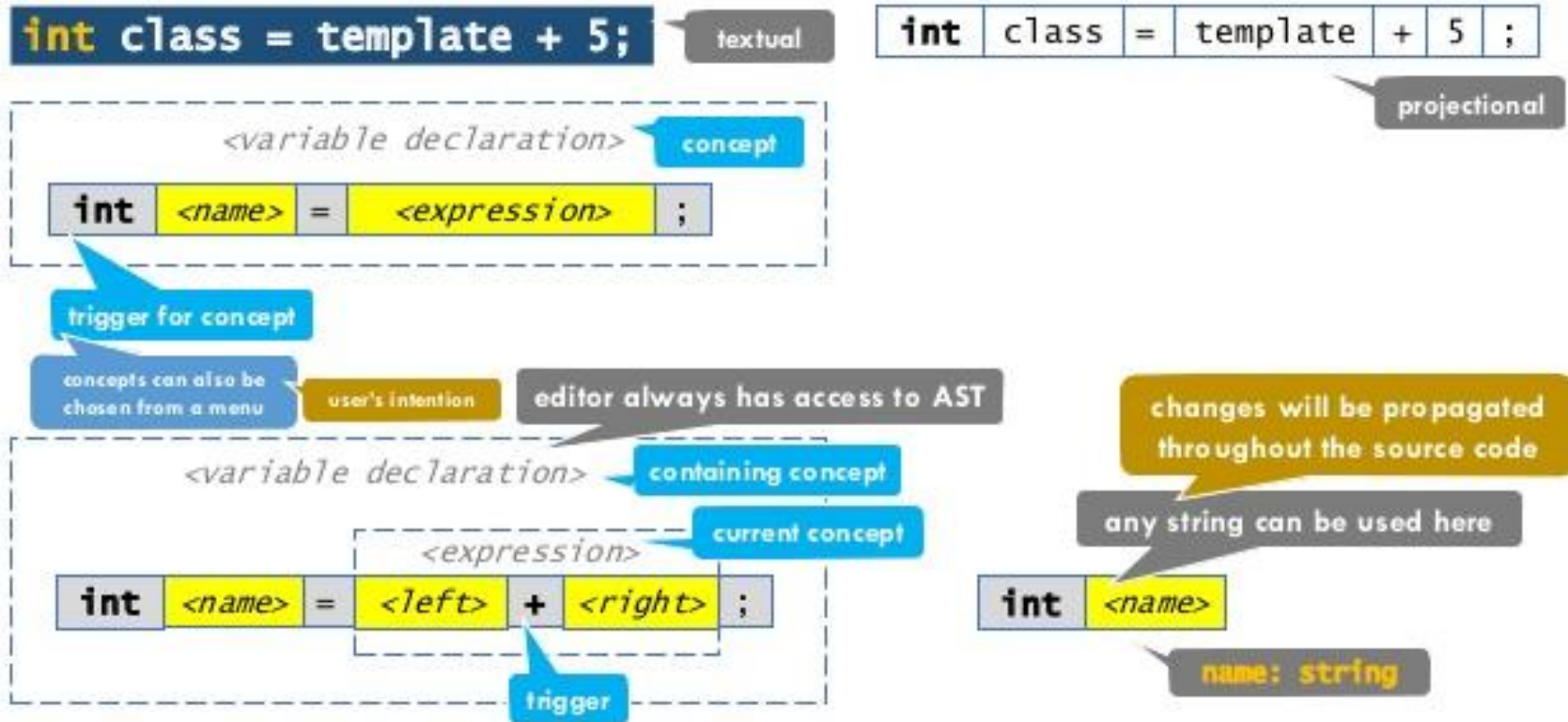
## Projekciós Editorok – példa

- Szemantikus modell
  - intervallum alsó, felső határa
  - integrálandó kifejezés
- Konkrét kifejezés:  $\int_{-\infty}^{\infty} e^{-x^2} dx$
- Közvetlenül leképezhető a szemantikus modellre



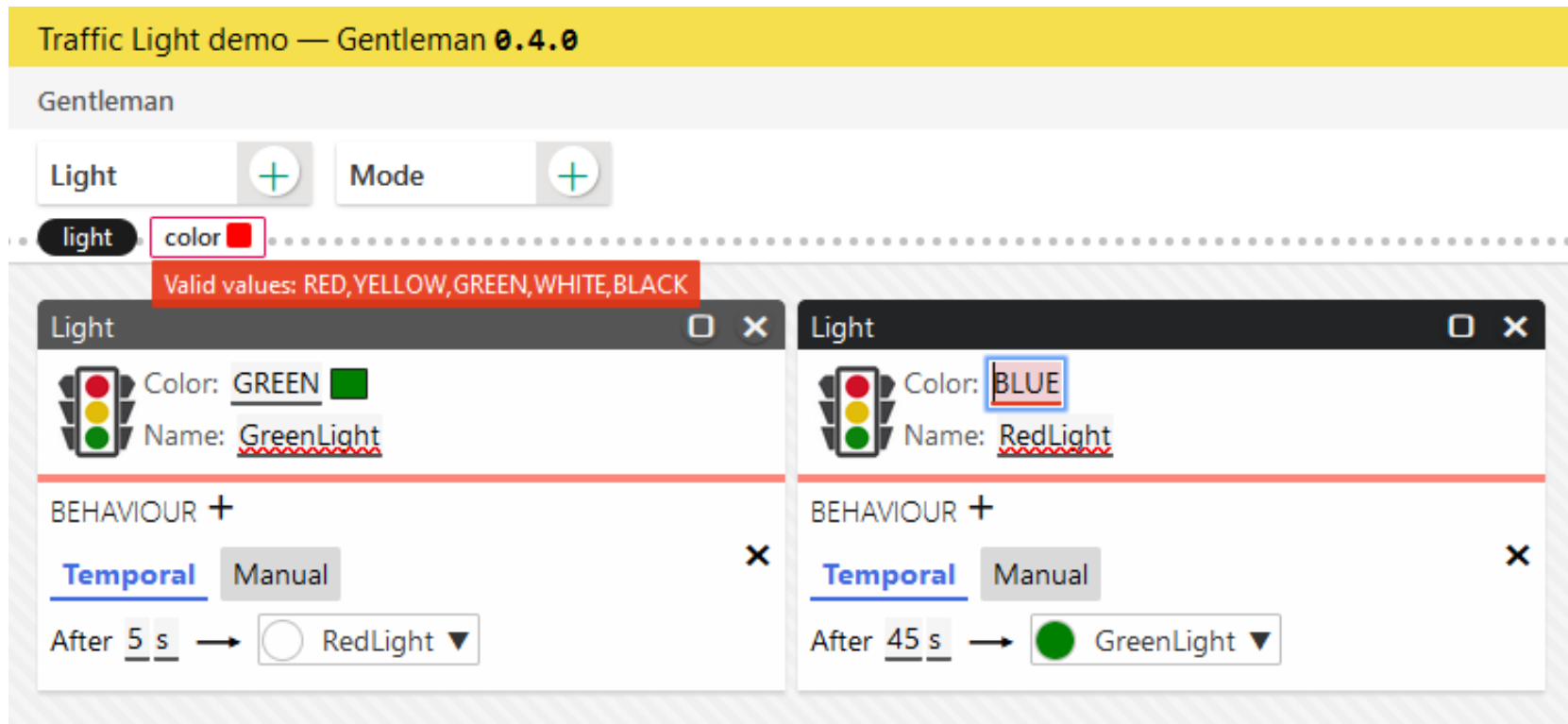
# Projekciós Editorok – JetBrains MPS

## Parsing vs. projectional editing



Forrás: Mikhail Barash: Reflections on teaching JetBrains MPS within a university course

# Projekciós Editorok – Gentleman



Forrás: <https://geodes.iro.umontreal.ca/gentleman/demo/traffic-light/index.html>



Köszönöm a figyelmet!