Fordítóprogramok a gyakorlatban

5-7. Előadás – Szemantikai elemzés.

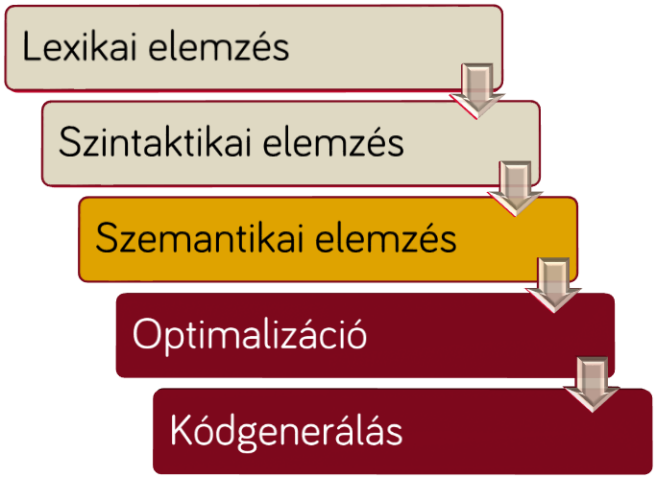
BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Miről volt szó legutóbb?

- Levezetési fa és AST (Abstract Syntax Tree)
  - Konstruálás
  - Szintaktikai elemzés

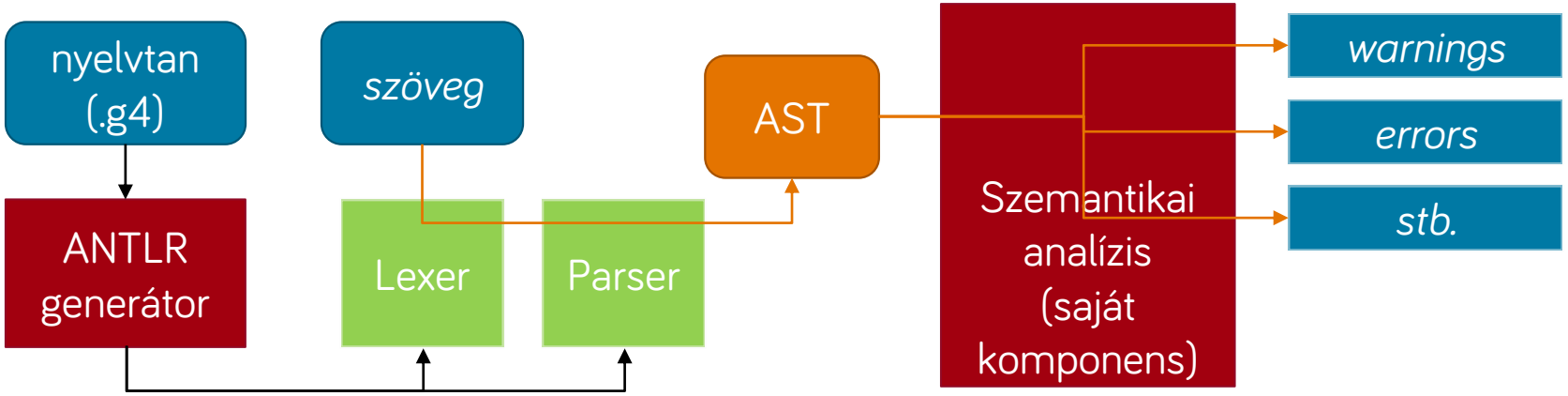
A mai napon…

- AST feldolgozás ANTLR API-val Visual Studioban
- Szemantikai elemzés:
  - Szimbólumtábla
  - Típusrendszerek, típusellenőrzés



ANTLR – feldolgozás

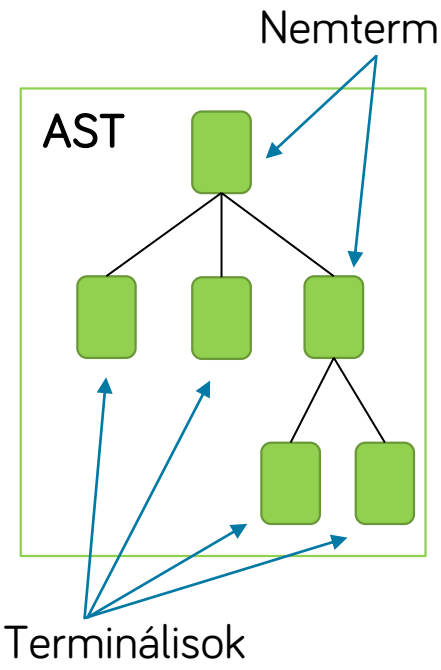
ANTRL API


nyelvtan
(.g4)
ANTLR
generátor
szöveg
Lexer
Parser
AST
Szemantikai
analízis
(saját
komponens)
warnings
errors
stb.

ANTRL API

- AST reprezentáció ANTLR-ben (típusok, függvények)
  - Lexer osztály //lexikai elemzés
  - Parser osztály //szintaktikai elemzés
- Mi a feladat?
  - Feldolgozni az AST-t!

ANTLR API – AST



ANTLR alaposztályok

API a fa bejárására:
- children
- parent
- GetChild(int index)
- GetText()

ParseTree

RuleNode

TerminalNode

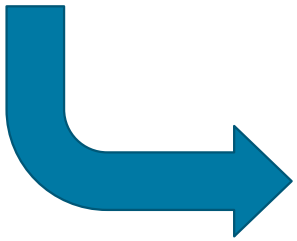
Nemterminálisok
- Típusos API a bejáráshoz

Terminálisok

ANTLR API – Típusos API

```
stmt
    :   assignStmt
    ;

assignStmt
    : ID '=' expr ';'
    ;
```



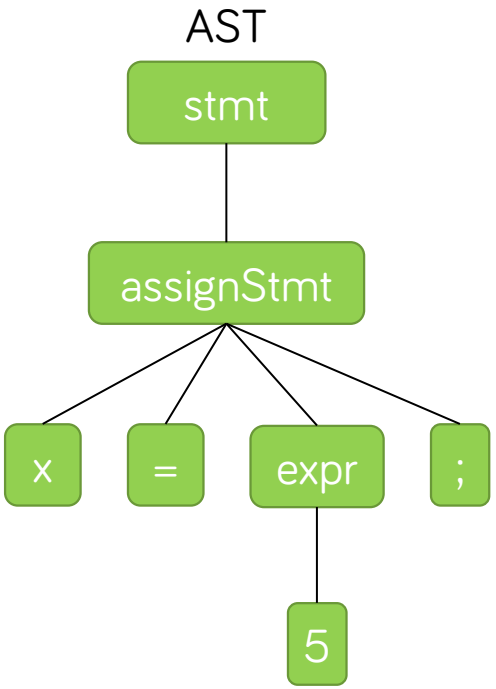
```
class StmtContext : IRuleContext {
  public AssignStmtContext assignStmt(){
     //…
    }
}
class AssignStmtContext : IRuleContext {
   public ExprContext expr(){
     //…
    }
}
class Expr : IRuleContext {

}
```

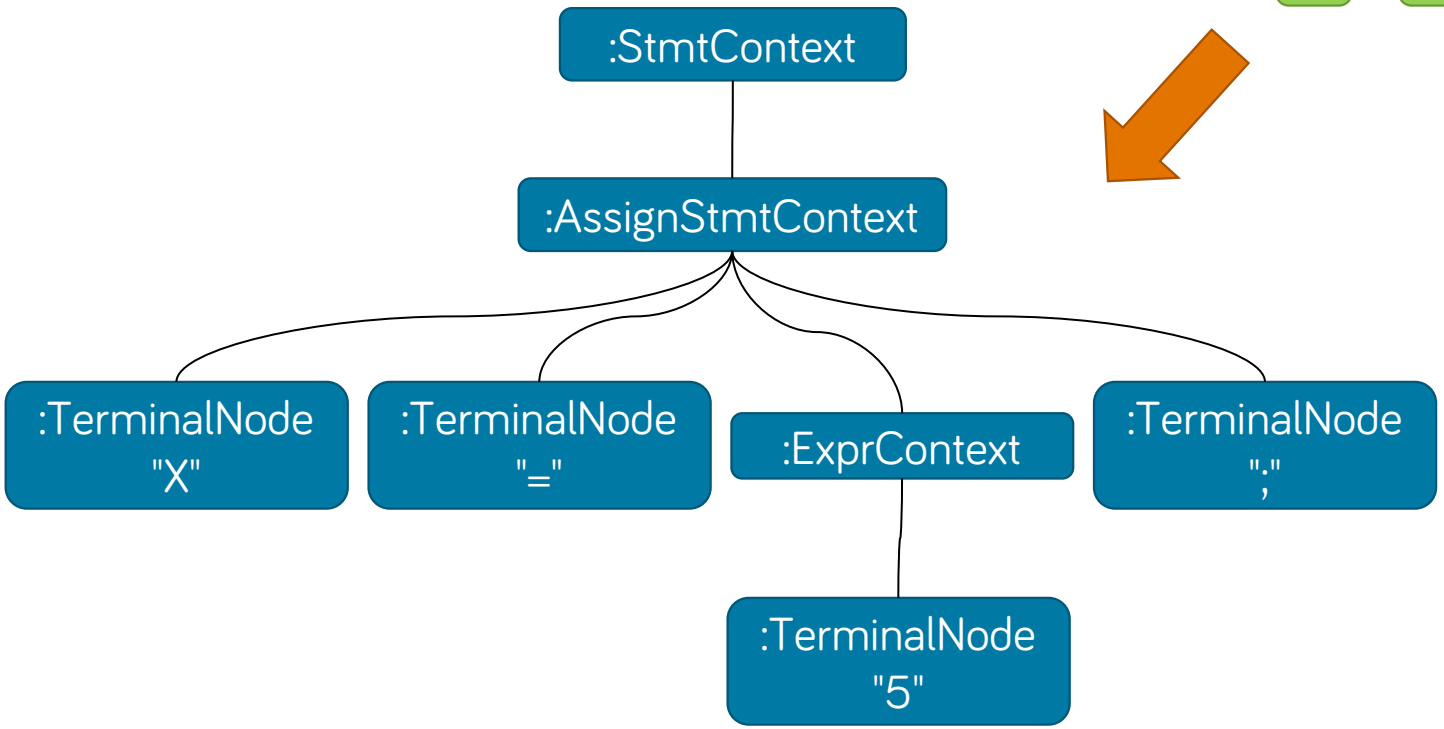
ANTLR – Típusos API

Forráskód

```
x = 5;
```

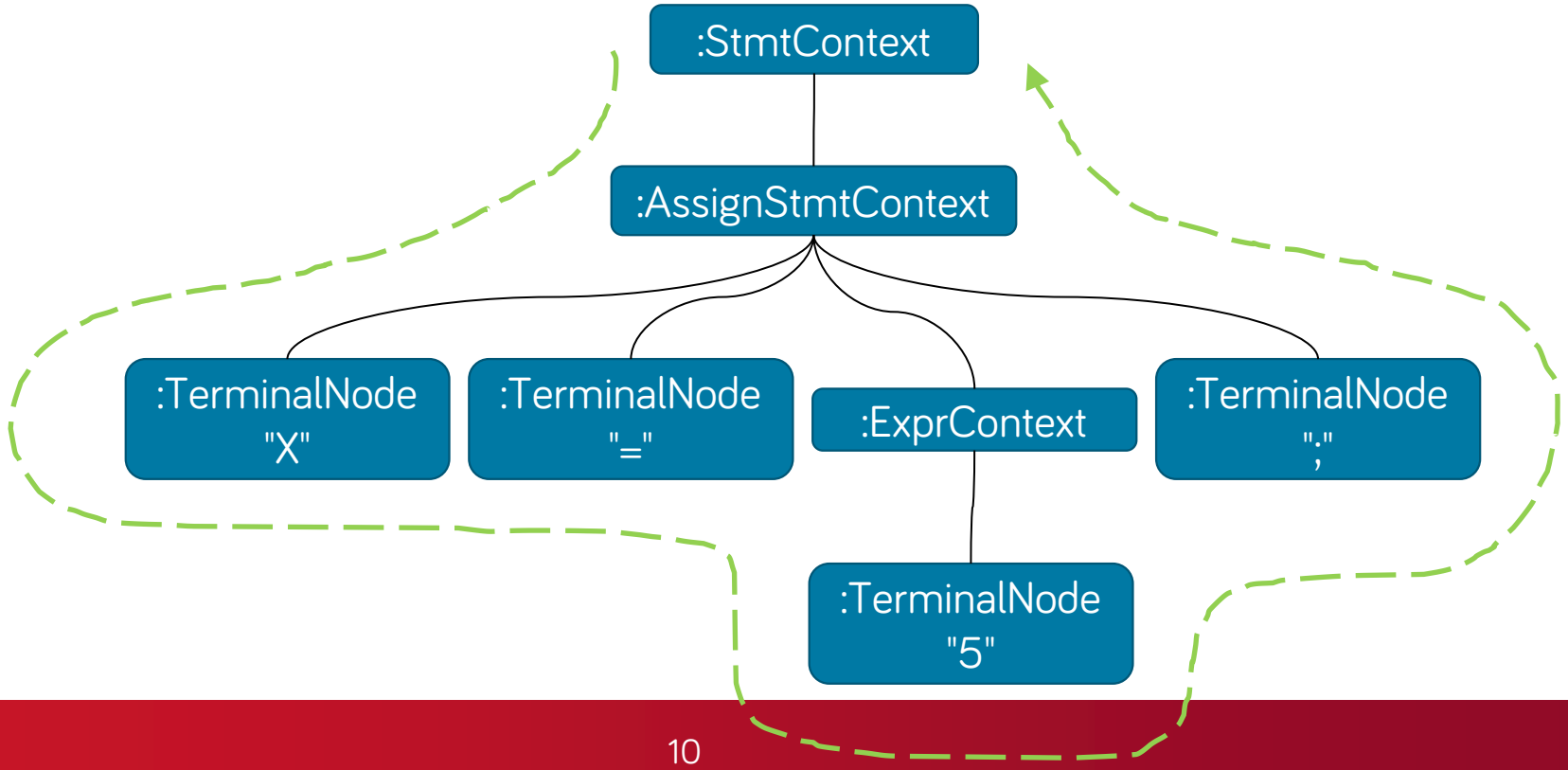


AST ANTLR objektumokkal



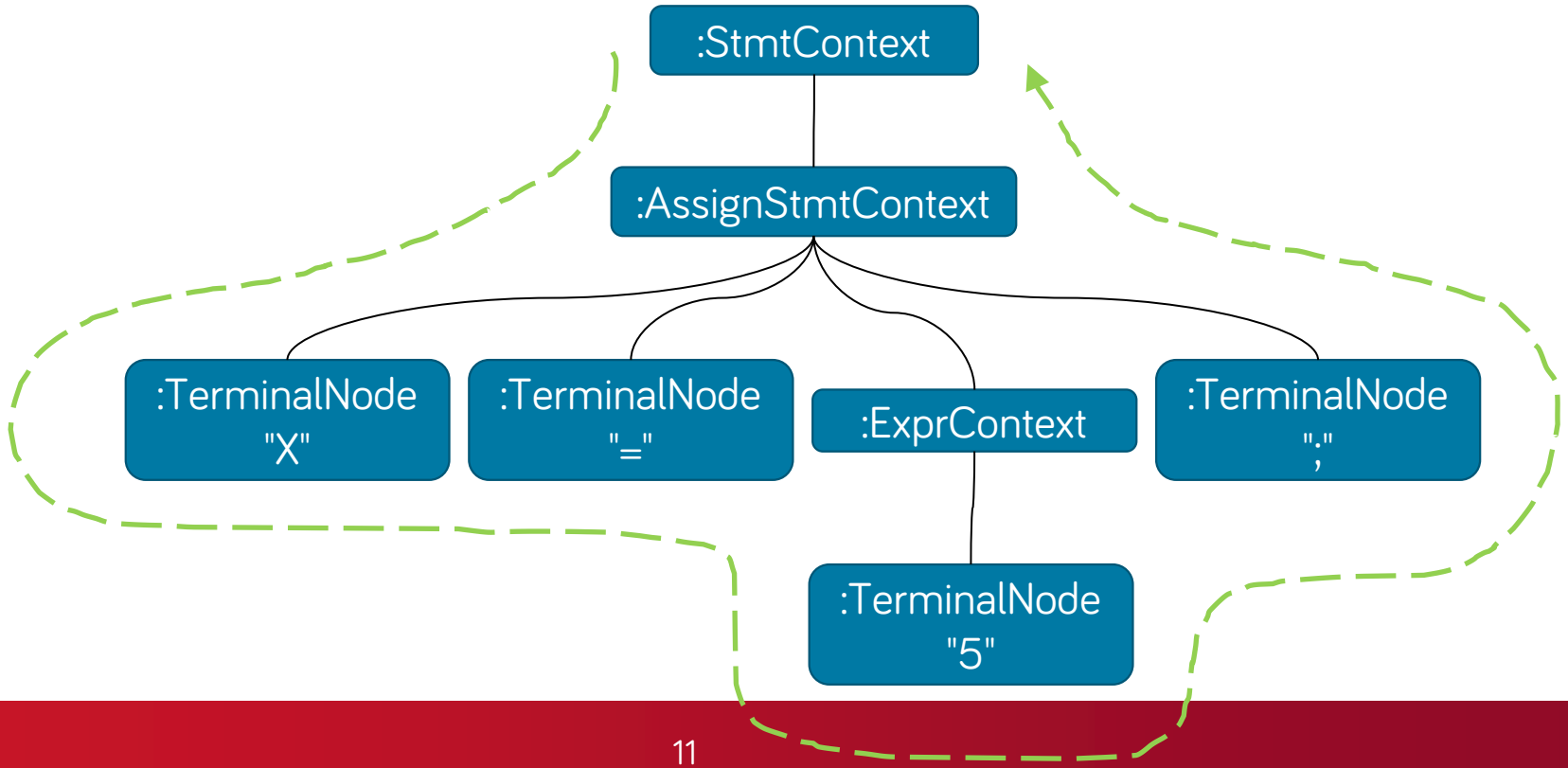
Visitor minta

- Az ANTLR generálhat egy *Visitor* osztályt a nyelvtanhoz
- Az osztály bejárja a fát a **Visit** metódus meghívásával
- Virtuális hook metódusok az egyes típusokhoz → leszármaztatás



Visitor minta

```
public override object VisitProgram(StmtContext context)
{
        //Meghívódik, minden egyes Stmt-re
}
```



Gyakorlat

ANTLR API + bejárás

Szemantikai elemzés

- Mit jelent? (szintaktika vs szemantika)
- Miért van rá szükség?
- Miért nem a szintaktikai elemzés része?

Szemantikai elemezés

```
namespace Demo {
    class Book : IBook {
        private int isbn;
        public string Title { get; set; }
    }
    class Program {
        public static void main(string[] args) {
            Book book = new Book();
            book.isbn = 5;
            Book book2 = new Book("Piszkos Fred közbelép");
            Book book3 = new Book {
                Title = book2.Title - " közbelép" +
                        ", a kapitány"
            };
        }
    }
}
```

Nem definiált interfész név

Privát tagváltozó hivatkozása

Nem létező konstruktor

Operátor típushiba

Szemantikai elemzés

- Szintaktikailag helyes programok mindig helyesek? **NEM**
  - Változó hivatkozása deklaráció előtt
  - Scope ellenőrzés
  - Típusnévellenőrzés
  - Kifejezések típusainak ellenőrzése
  - …
  - Helyes = érvényes ≠ a specifikációnak megfelelően működik
- Szemantikai ellenőrzés:
  - A fenti hibák kiküszöbölése
  - Szimbólumtábla készítés – a kódban hivatkozott szimbólumok (változók, osztályok stb.) és azok tulajdonságainak gyűjteménye

Szemantikai elemzés

- Miért nem a szintaktikai elemzőbe építjük?

```
class Computer : IMachine {
    int Price { get ;}
}

interface IMachine {
    int Weight { get; }
    int Price { get; }
}
```

Hiányzik egy property implementációja. Ezt még nem tudjuk akkor, amikor beolvassuk az első sort, csak amikor már beolvastuk az interfész definíciót és értelmeztük azt.

Szemantikai elemzés

- Témakörök
  - Scope ellenőrzés
  - Típusellenőrzés
  - …

Scope ellenőrzés

Scope

```
class Person
    {
        string name;

        public Person(string name)
        {
            this.name = name;
        }

        void Test()
        {
            {
                string name;
            }
        }
    }
```

A név nem egyedi azonosító egy programon belül.

Scope (hatókör): egy entitás hatóköre a program azon helyei, amelyekben az entitás neve az entitásra utal.

Szimbólumtábla

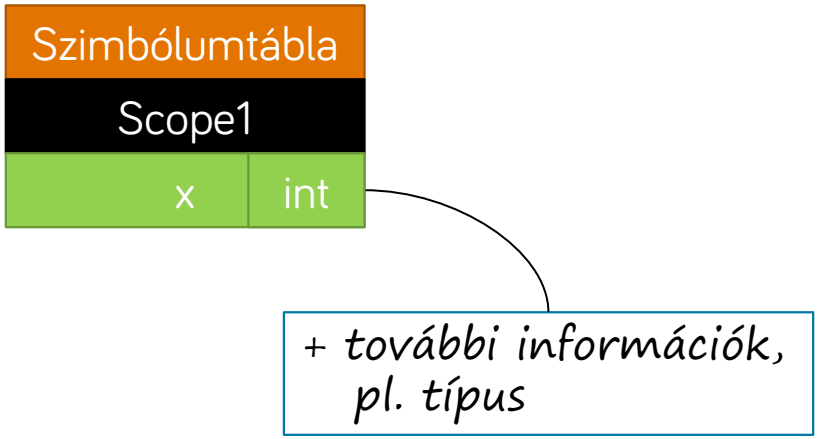
- A szemantikai analízis során az elejétől kezdve lépésenként feldolgozzuk a programot
  - **Szimbólumtábla**: egy leképezés, amelyben az egyes neveket leképezzük a hozzájuk tartozó entitásokra
  - A szemantikai analízis során folyamatosan karbantartott adatszerkezet
  - A program különböző pontjain más és más a tartalma

Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```

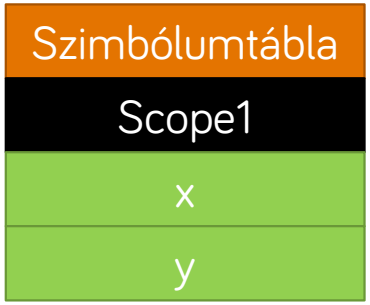
Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



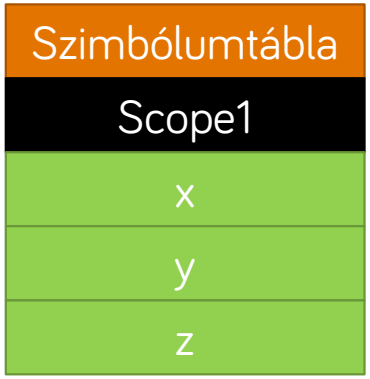
Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



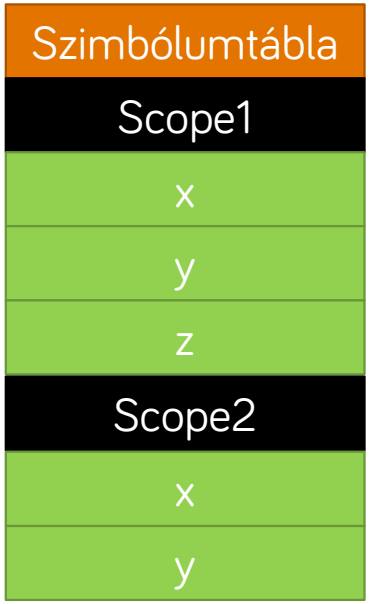
Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



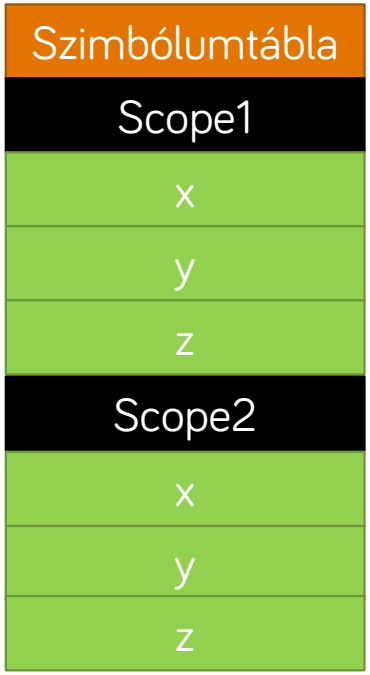
Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



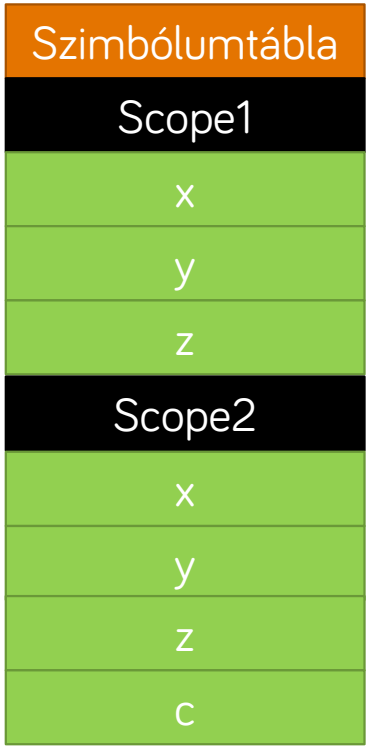
Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



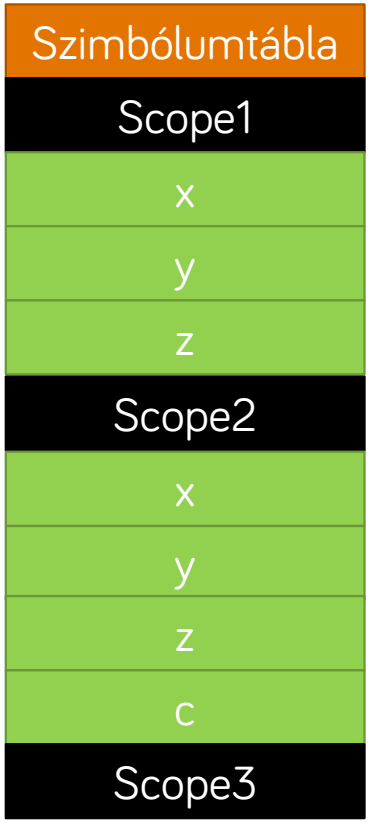
Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```

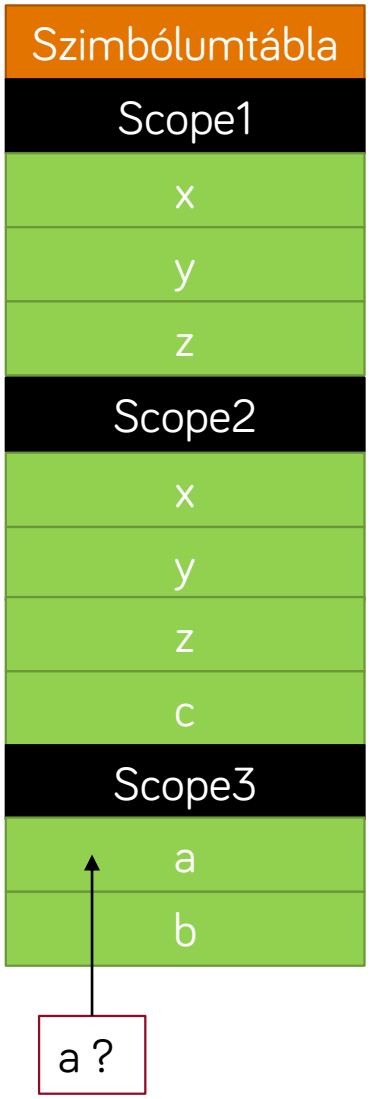


Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```

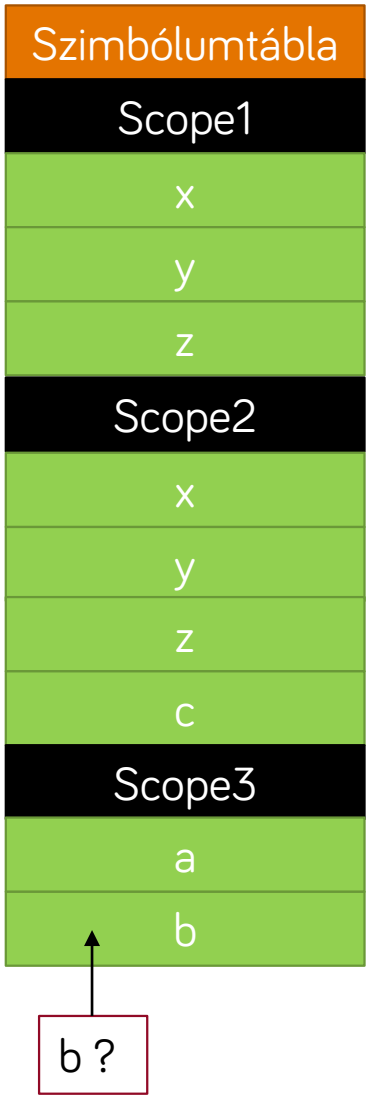
Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



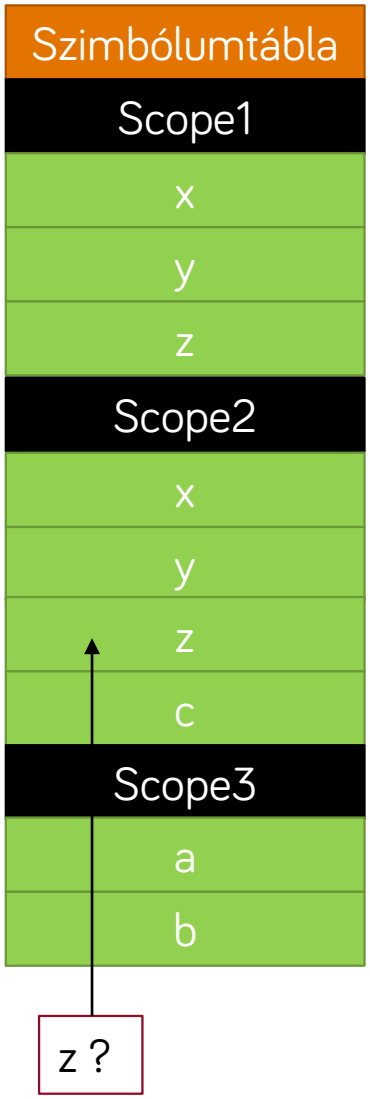
Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



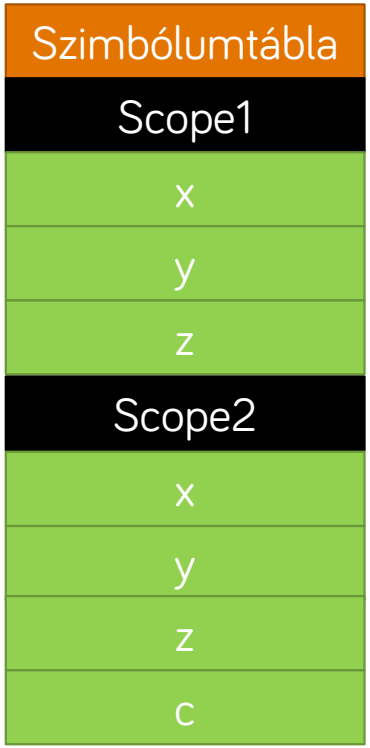
Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



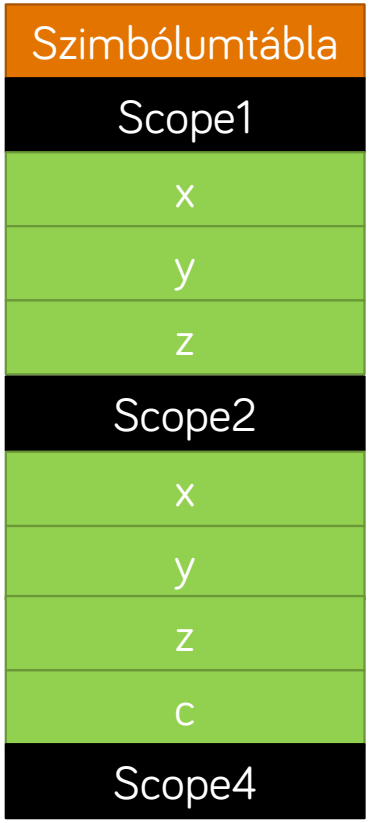
Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```

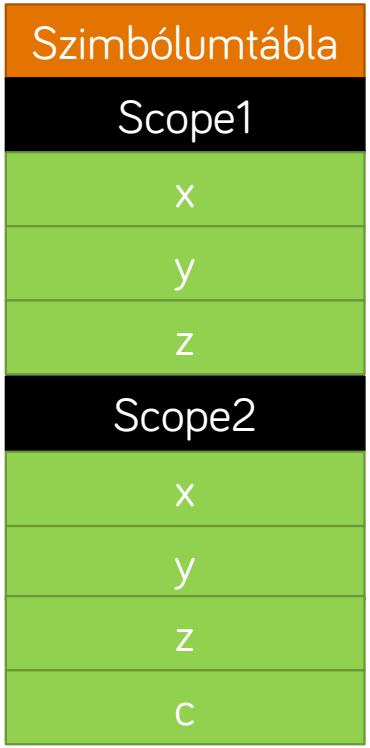


Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```

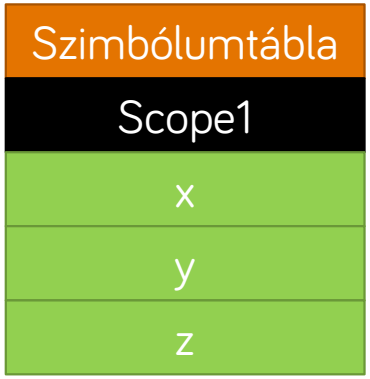
Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



Szimbólumtábla

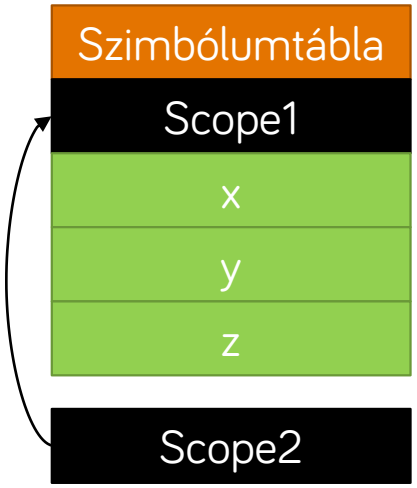
- **Symbol stack**:
  - Egy verem, amiben leképezés halmazokat tárolunk
  - Minden halmaz megfelel egy hatókörnek
- Műveletek a szimbólumtáblán:
  - Új hatókör hozzáadása (**push scope**)
  - Az utoljára hozzáadott hatókör törlése (**pop scope**)
  - Szimbólum hozzáadása az aktuális hatókörhöz (**insert symbol**)
  - Szimbólum keresése (**lookup symbol**)
- Mit tárol? Név + hatókör + típus + …

Szimbólumtábla

- 2. megközelítés: verem helyett láncolt lista
  - Statikus struktúra ← nem változik dinamikusan

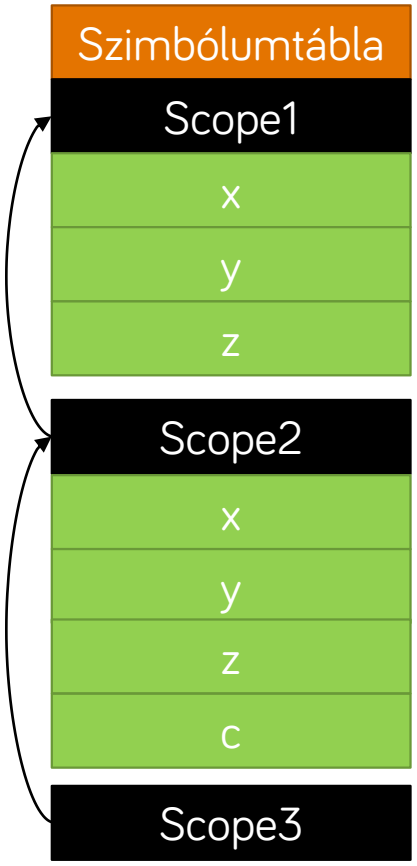
Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



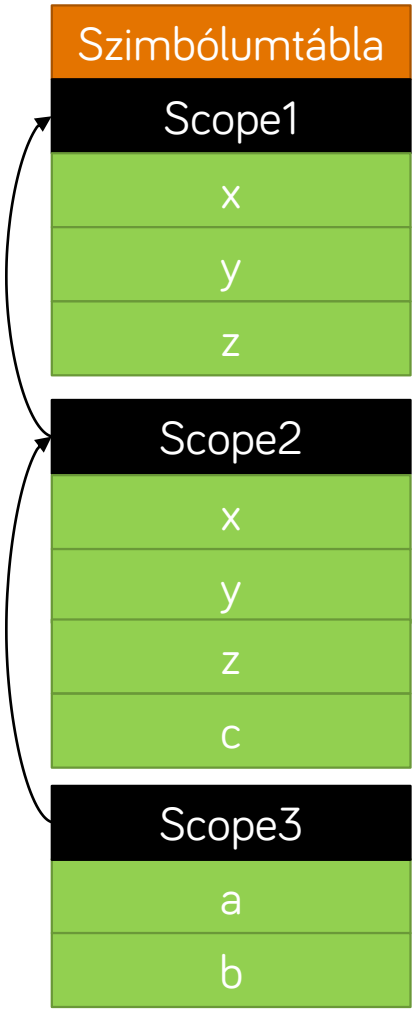
Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



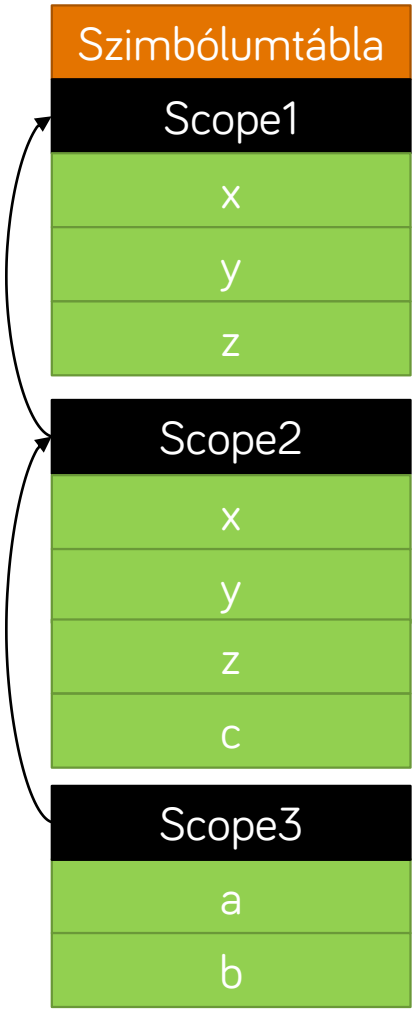
Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



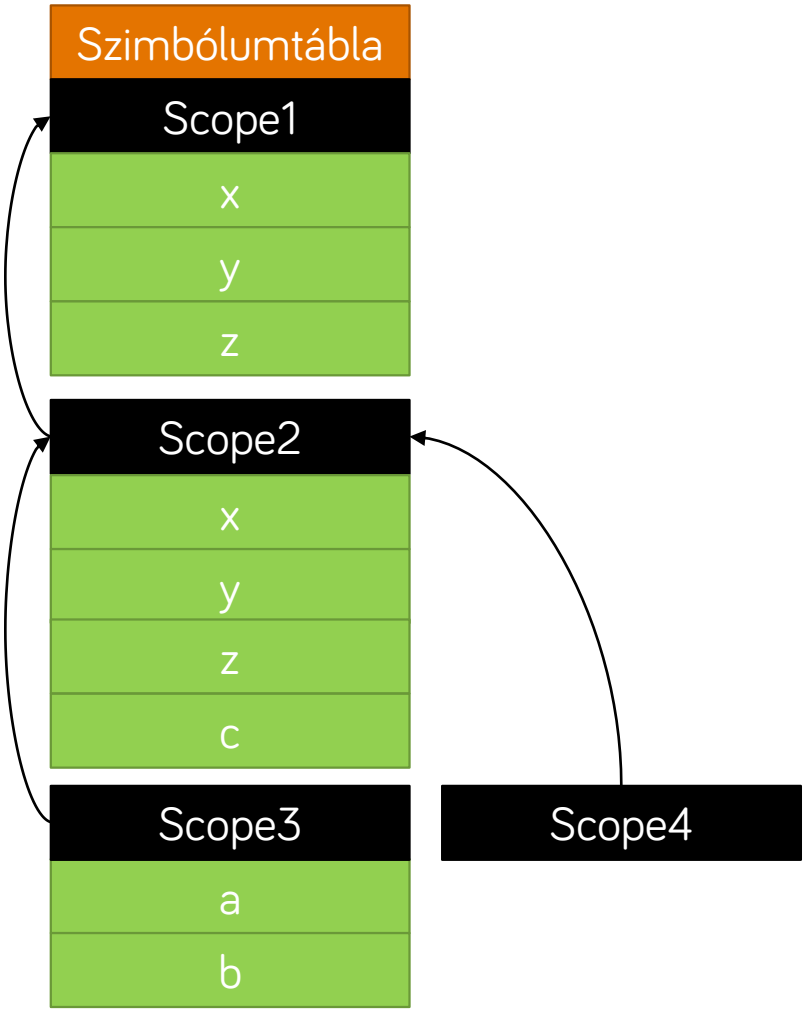
Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



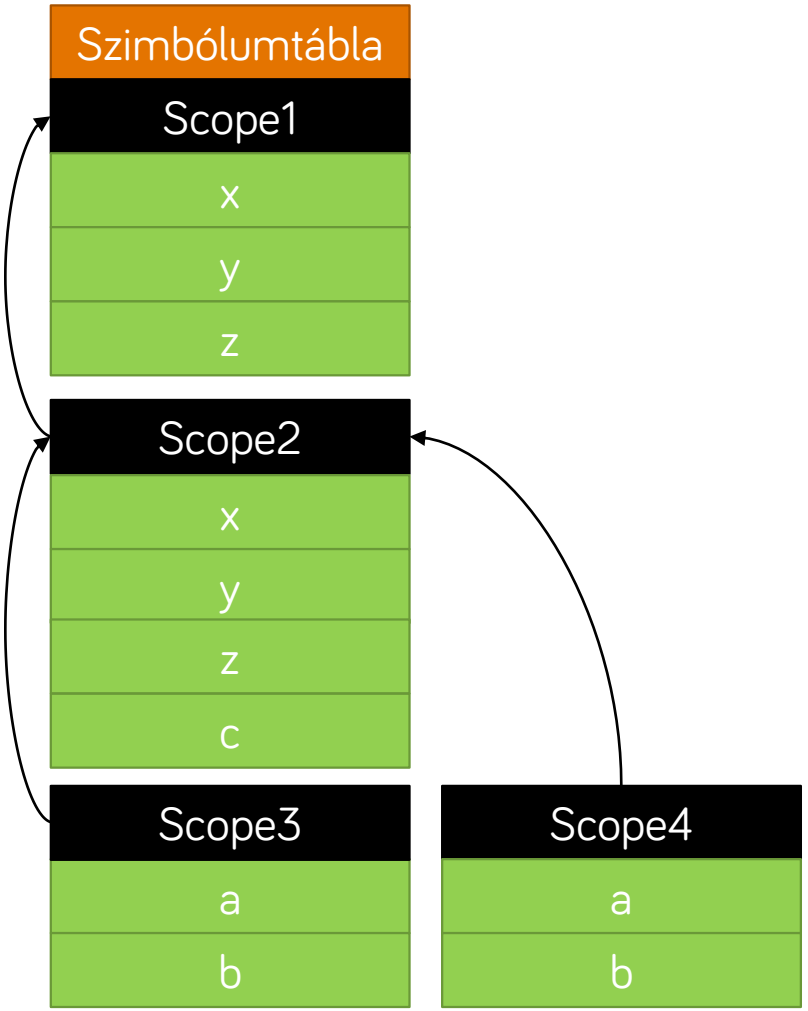
Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



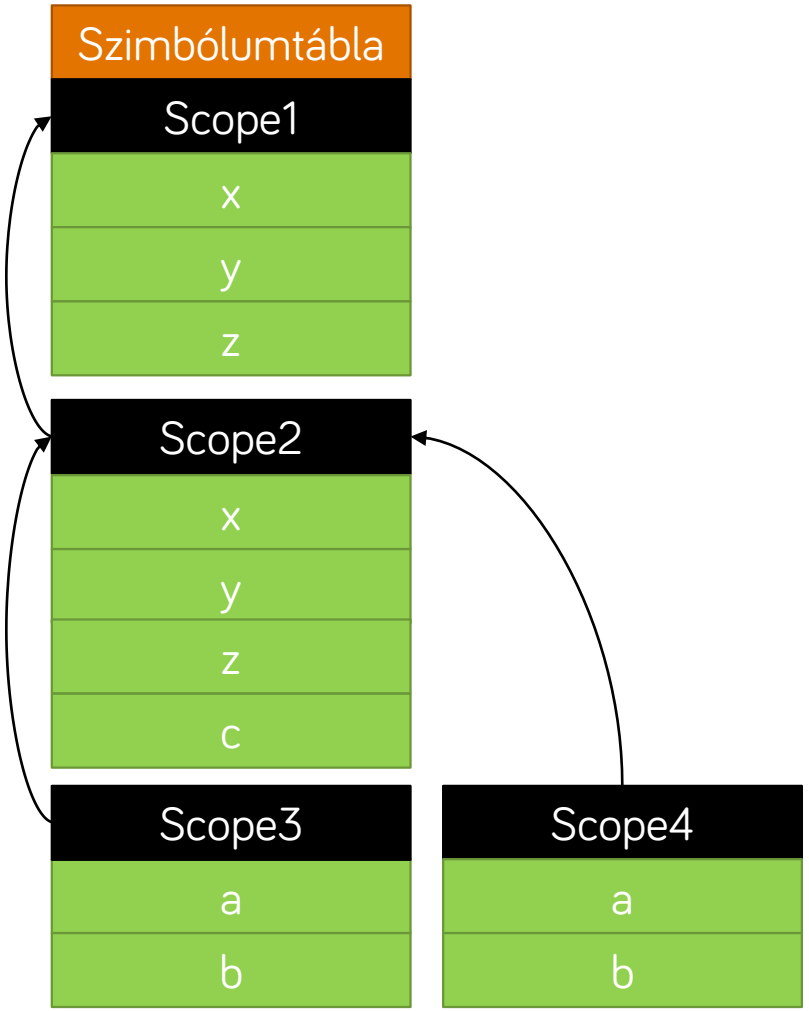
Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



Szimbólumtábla

```
int x = 1;
int y = 2;
int z;
int Add(int x, int y) {
    int z;
    int c;
    {
        int a = x;
        int b = y;
        z = a + b;
    }
    {
        int a = x;
        int b = y;
        c = a + b;
    }
    return c;
}
z = Add(x, y);
```



Szimbólumtábla

- Objektum-orientált környezetben
  - Osztályok, metódusok saját hatókörrel rendelkeznek
  - Öröklés esetén a leszármazott osztály hatóköre mutat a szülő osztályra mutat

Gyakorlat

Szimbólumtábla felépítése és karbantartása

Szemantikai ellenőrzés

- További feladatok:
  - Speciális ellenőzrések:
    - Pl. **break**, **continue** csak bizonyos blokkokon belül állhat
    - Adott egy szabályrendszer
    - Csomópontonként feldolgozzuk az AST-t
    - Minden csomópontra ellenőrizzük a szabályokat
  - Típusellenőrzés (következő órán)

Típusellenőrzés

Programozási nyelvek szemantikus analízise

Alapfogalmak

- **Típus**: *értékek és azokon értelmezett műveletek egy halmaza.*
  - Tipikus programozási nyelvekben:
    - Primitív típusok (int, float, double, char, bool stb.)
    - Összetett típusok (struct, class, array stb.)
      - Nyelvi szintű kiegészítések lehetnek hozzájuk
- **Típusos nyelv**: *az egyes entitások (szimbólumok) típussal rendelkeznek.*
- **Típusrendszer (type system)**: *a típusokkal kapcsolatos szabályok összessége*

Típushiba

```
int x = "abcd";
```

```
Book book;
book = new List<Book>();
```

```
if (1 + 2 + 3) {
   //...
}
```

- **Típushiba** (**type error**): amikor *egy objektumon olyan műveletet akarunk végrehajtani, amelyet a típusa nem támogat*

Alapfogalmak

- **Típusellenőrzés:** a típushibák elkerülésére az egyes entitások típusát a műveletek végrehajtása előtt ellenőrizzük
  - fordítási időben,
  - futási időben,
  - vegyesen
- **Erősen típusos nyelv**: minden típushiba felderíthető fordítási időben
- **Gyengén típusos nyelv**: futás idejű típushiba előfordulhat

Alapfogalmak

- **Statikus típusellenőrzés**: az egyes entitások deklarálásánál megadjuk azok típusát, csak ezen fordításidejű információk alapján történik a fordításidejű típusellenőrzés
  - Gyors,
  - Nehéz minden speciális esetet kezelni
- **Dinamikus típusellenőrzés**: futási időben az aktuális adatok alapján történik az ellenőrzés
  - Lassabb
  - Precízebb

→Erősen típusos nyelv statikus típusellenőrzéssel nagyon nehéz

Alapfogalmak

- **Implicit típuskonverzió (*coercion*)**: (≠ cast)
  automatikus típus átalakítás a legegyszerűbb típushibák elkerülésére
  - Pl. int → double
- **Speciális esetek összetett típusrendszereknél**: (pl. C#, Java)
  - Erősen típusos nyelvek
  - Statikus típusellenőrzés
  - Futási időben előfordulhat **null** hivatkozás
  - Explicit castolásnál történhet futásidejű hiba

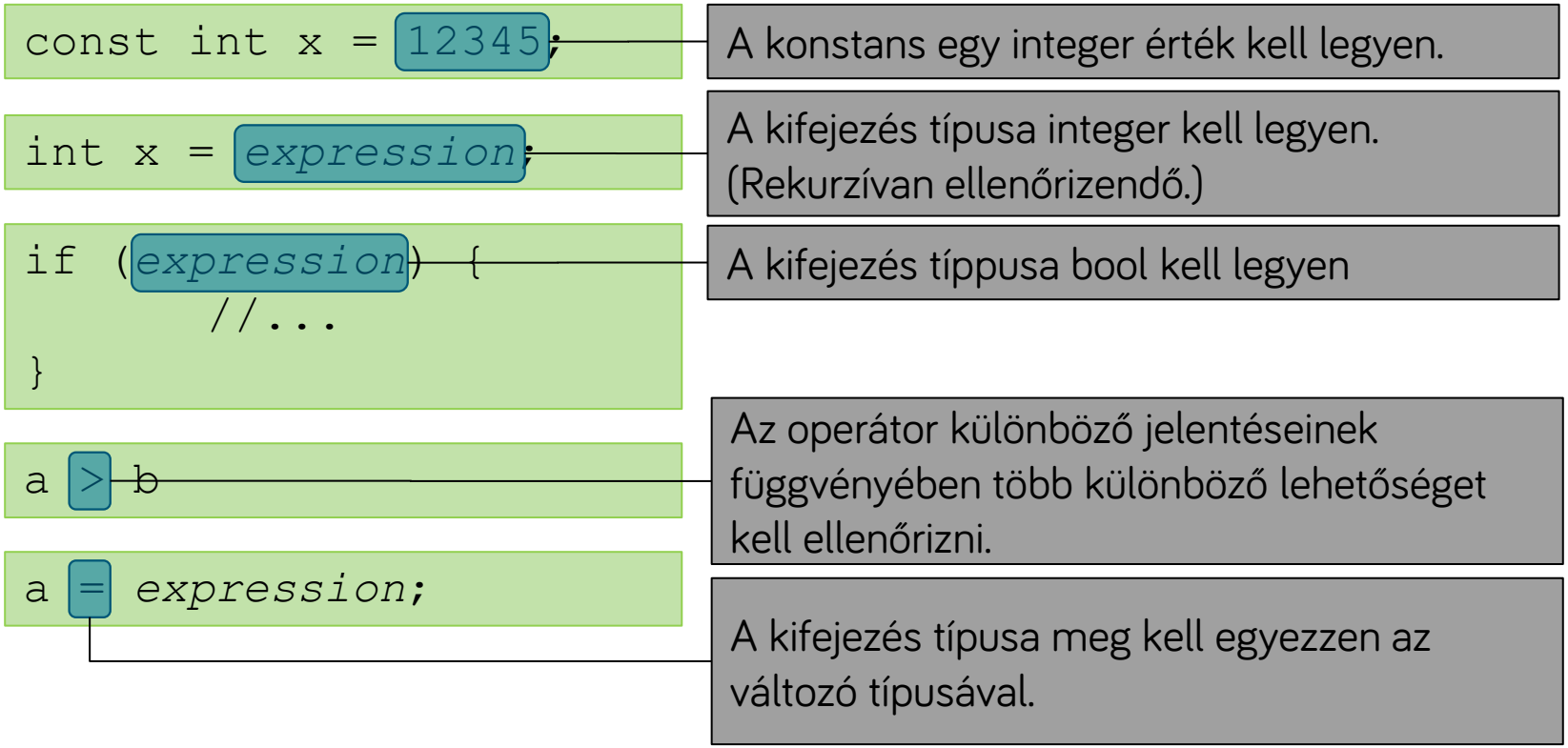
Típusellenőrzés

- Feladatok:
  1. El kell tudni dönteni, hogy a program egy adott pontján milyen típusú kifejezés állhat
  2. El kell tudni dönteni, hogy egy kifejezésnek mi a típusa
- Módszer:
  - **A típusrendszer szabályai** (1)
    - Pl. „Az if (expr) statement” utasításban a kifejezés típusa boolean kell legyen.
      - Amennyiben a szemantikai ellenőrzés során egy if csomópontot találunk az AST-ben, ellenőrizzük a kifejezés típusát
    - Pl. értékadás esetén a bal és a jobb oldal típusa meg kell egyezzen.
      - Amennyiben a szemantikai ellenőrzés során egy értékadás csomópontot találunk az AST-ben, ellenőrizzük a bal és jobb oldal típusát

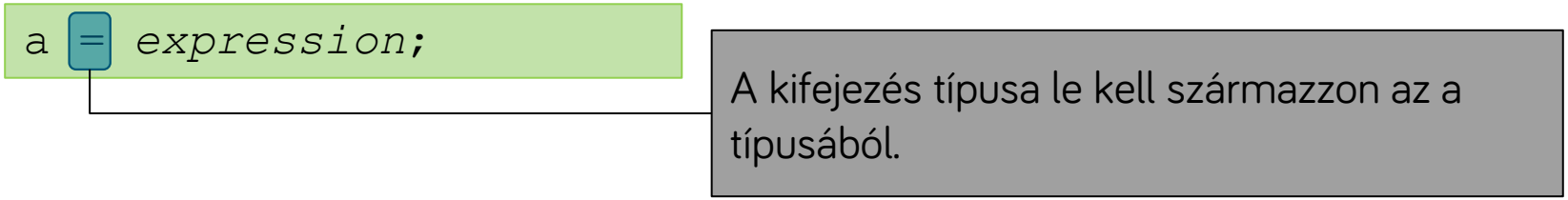
Típusellenőrzés

- Módszer:
  - Kifejezések típusainak megállapítása (2) – következtetőrendszer
    - Konstansok típusa ismert (bool, string, int stb.)
    - Változók típusa → ld. szimbólumtábla
    - Összetett kifejezések – az egyes operátorokra szabályaink vannak
      - Pl. „Ha a kifejezés exp1 + exp2 és exp1 és exp2 string típusú, akkor a kifejezés eredménye string típusú”
      - Pl. „Ha a kifejezés exp1 + exp2 és exp1 és exp2 int típusúak, akkor a kifejezés eredménye int típusú”

Típusellenőrzés példák



- Öröklés esetén:



Típushierarchia

- A típusok rendezése (<), hogy az ellenőrzés egyszerűbb legyen
- Pl: `a = b` kifejezés akkor helyes, ha

$$type(b) \leq type(a)$$

- Mi a **`null`** kifejezés típusa?
  - Vagy speciálisan kell kezelni minden egyes ellenőrzés során, vagy:
  - Bevezetjük a `NullType` típust
  - Minden `A` class típusra: `NullType < A`

Típushierarchia

```
bool a = (true && "false") && true;
if (a) {
        //...
}
```

```
Type error: && operátor string operandussal
```

Mi lesz a részkifejezés típusa?

- Nem tudjuk
- Új típust vezetünk be: `ErrorType`
- Egyetlen példánya van: **`error`**

Típushierarchia

```
bool a = (true && "false") && true;
if (a) {
        //...
}
```

```
Type error: && operátor string operandussal
```

```
Type error: && operátor ErrorType operandussal
```

Mi lesz a kifejezés típusa? `ErrorType`

Típushierarchia

```
bool a = (true && "false") && true;
if (a) {
        //...
}
```

```
Type error: && operátor string operandussal
```

```
Type error: && operátor ErrorType operandussal
```

```
Type error: = operátor, bool-nak nem adhatunk
ErrorType típusú értéket
```

Mi lesz **a** típusa? `ErrorType`

Típushierarchia

```
bool a = (true && "false") && true;
if (a) {
        //...
}
```

```
Type error: && operátor string operandussal
```

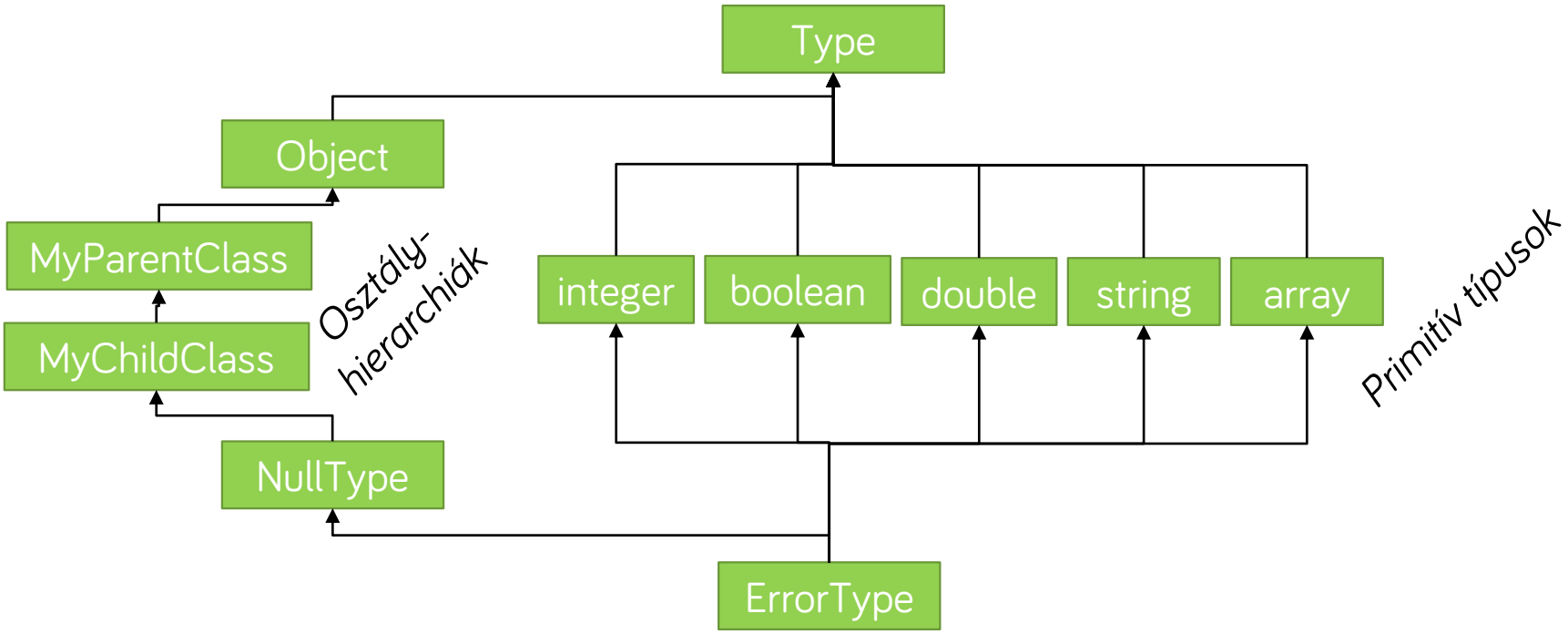
```
Type error: && operátor ErrorType operandussal
```

```
Type error: = operátor, bool-nak nem adhatunk
ErrorType típusú értéket
```

```
Type error: if utasítás, a feltétel csak bool
lehet
```

Típushierarchia

- Probléma: a típushiba eszkalálódik, pedig elég lenne csak az első helyen jelezni
- Megoldás: minden T típusra ErrorType < T
- ErrorType-ot szokták bottom típusnak nevezni



Szemantikai ellenőrzés

- Témák, amik kimaradtak:
  - Típusellenőrzéssel kapcsolatban
    - Függvények típusai
    - Függvénynév túlterhelés (overload)
    - `this` típusa?
  - Láthatósággal kapcsolatban
    - `private/public/protected` módosítók
    - Öröklés (többszörös öröklés, virtuális függvények stb.)
  - Speciális nyelvi elemek és a kapcsolódó szemantikai ellenőrzés
    - `break/continue`
    - `return`
    - Tömbök kezelése (indexelés, speciális inicializálások)
    - `new` kulcsszó
  - Dinamikus típusellenőrzés

Gyakorlat

Típusellenőrzés – TinyScript

Köszönöm a figyelmet!