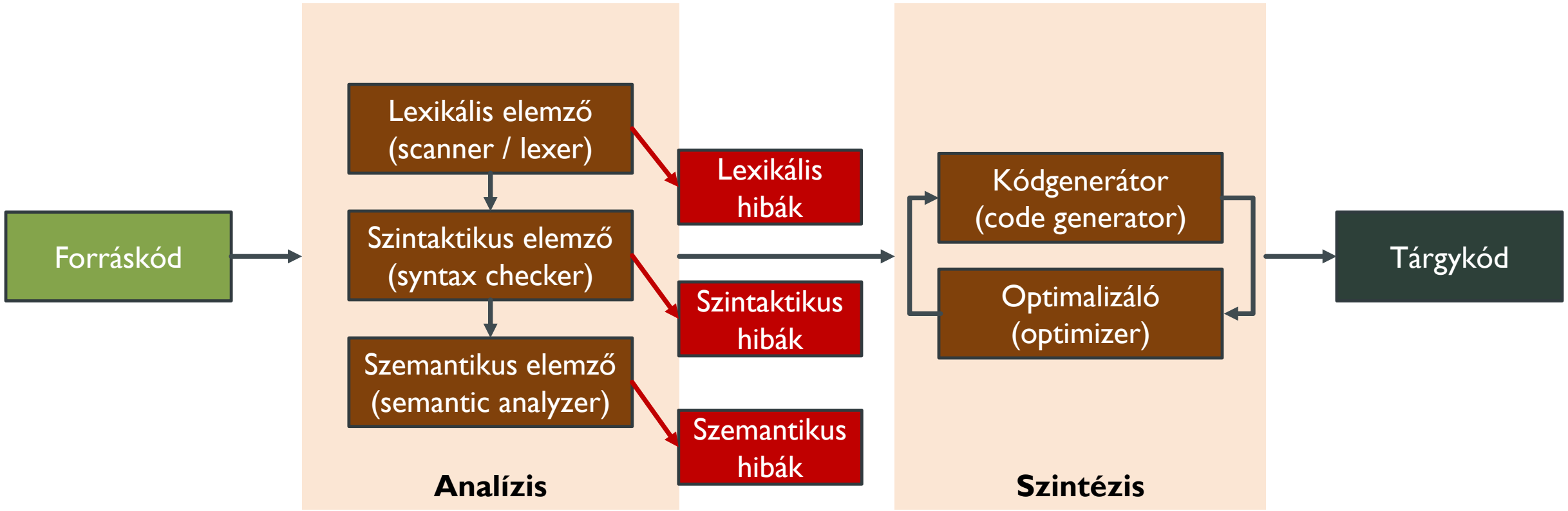
SZEMANTIKUS ELEMZÉS

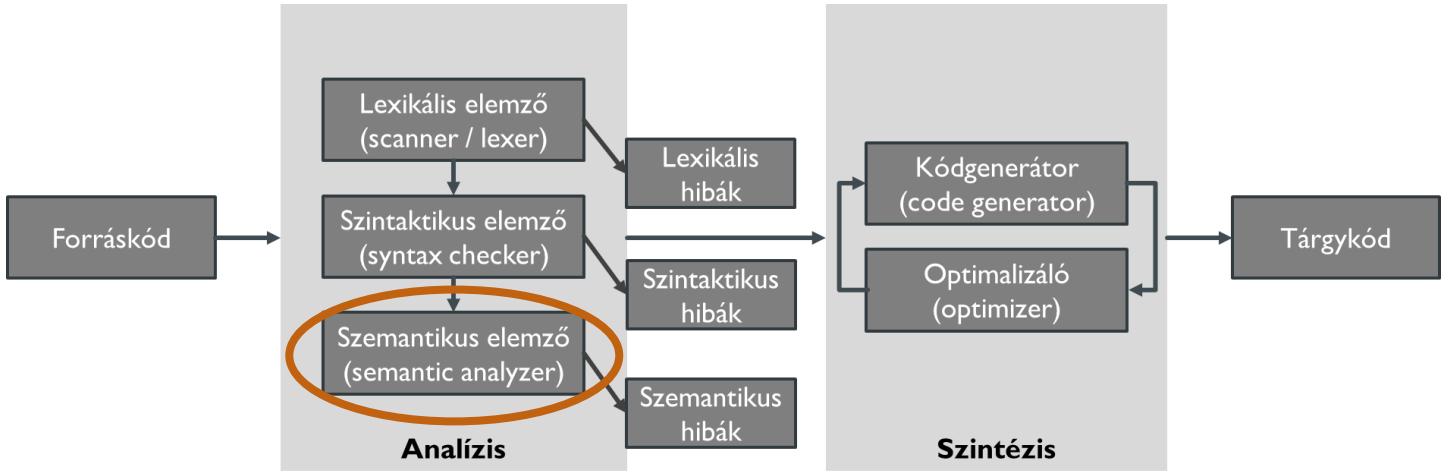
FORMÁLIS NYELVEK ÉS FORDÍTÓPROGRAMOK ALAPJAI

Dévai Gergely
ELTE

A FORDÍTÓPROGRAMOK LOGIKAI FELÉPÍTÉSE



SZEMANTIKUS ELEMZŐ



- *Feladat:*
  A statikus szemantika (pl. változók deklaráltsága, típushelyesség stb.) ellenőrzése
- *Bemenet:*
  Szintaxisfa
- *Kimenet:*
  Szintaxisfa attribútumokkal, szimbólumtábla + szemantikus hibák
- *Eszközök:*
  Attribútumnyelvtanok

A SZEMANTIKUS ELEMZÉS FELADATAI

- Deklarációk feldolgozása
- Azonosítószimbólumok deklarációhoz kötése
- Hatókörrel és láthatósággal kapcsolatos szabályok ellenőrzése

Szimbólumtábla

- Típusellenőrzés / Típuslevezetés
- Típuskonverziók

Attribútumnyelvtan

- Figyelmeztetés fordítási időben észrevehető hibákra

A szemantikus elemzés feladatai erősen nyelvfüggőek!

SZIMBÓLUMTÁBLA

DEKLARÁCIÓKKAL KAPCSOLATOS ELLENŐRZÉSEK

DEKLARÁCIÓKKAL, LÁTHATÓSÁGGAL KAPCSOLATOS HIBALEHETŐSÉGEK

Deklarálatlan változó:

```
int main() {
  cout << x;
}
```

Újradeklarált változó:

```
int main() {
  int x = 1;
  cout << x;
  char x = 'a';
  cout << x;
}
```

Hatókörön kívüli használat:

```
while(...) {
  int x = 5;
  ...
}
cout << x;
```

Típushiba elfedés miatt:

```
string x = "compiler";
while(...) {
  int x = 0;
  cout << x.length();
  ...
}
```

Privát adattag elérése:

```
class c {
  private:
    int x;
};
int main() {
  c c_obj;
  cout << c_obj.x;
}
```

TOKENEKHEZ CSATOLT INFORMÁCIÓK

Ez a tokensorozat elegendő információt hordoz ahhoz, hogy a szintaktikus elemző szintaxisfát építhessen,
de a szemantikus elemzőnek több információra van szüksége.

- Mi a deklarált változó neve? Mi a típusa?
- Az operátor két oldalán mely változók állnak?

A lexikális elemző kimenete valójában:

azonosító (”int”), azonosító (”x”), utasításvég, azonosító (”cout”), operátor (”<<”), azonosító (”x”), utasításvég

A tokenekhez a lexikális elemző által hozzárendelt kiegészítő információkat
kitüntetett szintetizált attribútumoknak
nevezzük. (További attribútumfajták hamarosan...)

SZIMBÓLUMTÁBLA

```
1  void f(int p) {
2    int x;
3    cin >> x;
4    cout << x+p+y;
5    int x;
6  }
```

Név	Fajta	Típus	Deklaráció	Használat

Megjegyzés: A következő példákban a C++ deklarációs szabályait használjuk. Más szabályokat használó nyelvek esetén a szimbólumtábla másképp működik.

SZIMBÓLUMTÁBLA

```
1  void f(int p) {
2    int x;
3    cin >> x;
4    cout << x+p+y;
5    int x;
6  }
```

Név	Fajta	Típus	Deklaráció	Használat
“f”	függvény	int → void	1. sor 6. oszlop	

A szimbólumtábla műveletei:

- **Beszúrás**:
  - Deklaráció esetén
  - Az új szimbólum és adatai bekerülnek a szimbólumtáblába

SZIMBÓLUMTÁBLA

```
1  void f(int p) {
2    int x;
3    cin >> x;
4    cout << x+p+y;
5    int x;
6  }
```

Név	Fajta	Típus	Deklaráció	Használat
“f”	függvény	int → void	1. sor 6. oszlop	
“p”	paraméter	int	1. sor 12. oszlop	

A szimbólumtábla műveletei:

- **Beszúrás**:
  - Deklaráció esetén
  - Az új szimbólum és adatai bekerülnek a szimbólumtáblába

SZIMBÓLUMTÁBLA

```
1  void f(int p) {
2    int x;
3    cin >> x;
4    cout << x+p+y;
5    int x;
6  }
```

Név	Fajta	Típus	Deklaráció	Használat
“f”	függvény	int → void	1. sor 6. oszlop	
“p”	paraméter	int	1. sor 12. oszlop	
“x”	lokális változó	int	2. sor 7. oszlop	

A szimbólumtábla műveletei:

- **Beszúrás**:
  - Deklaráció esetén
  - Az új szimbólum és adatai bekerülnek a szimbólumtáblába

SZIMBÓLUMTÁBLA

```
1  void f(int p) {
2    int x;
3    cin >> x;
4    cout << x+p+y;
5    int x;
6  }
```

Név	Fajta	Típus	Deklaráció	Használat
“f”	függvény	int → void	1. sor 6. oszlop	
“p”	paraméter	int	1. sor 12. oszlop	
“x”	lokális változó	int	2. sor 7. oszlop	3. sor 10. oszlop

A szimbólumtábla műveletei:

- **Beszúrás**:
  - Deklaráció esetén
  - Az új szimbólum és adatai bekerülnek a szimbólumtáblába

- **Keresés**:
  - Szimbólum használatakor
  - A szimbólum neve a kulcs a kereséshez
  - A szimbólum használatát érdemes feljegyezni (pl. refaktoráláshoz)

SZIMBÓLUMTÁBLA

```
1  void f(int p) {
2    int x;
3    cin >> x;
4    cout << x+p+y;
5    int x;
6  }
```

Név	Fajta	Típus	Deklaráció	Használat
“f”	függvény	int → void	1. sor 6. oszlop	
“p”	paraméter	int	1. sor 12. oszlop	
“x”	lokális változó	int	2. sor 7. oszlop	3. sor 10. oszlop 4. sor 11. oszlop

A szimbólumtábla műveletei:

- **Beszúrás**:
  - Deklaráció esetén
  - Az új szimbólum és adatai bekerülnek a szimbólumtáblába

- **Keresés**:
  - Szimbólum használatakor
  - A szimbólum neve a kulcs a kereséshez
  - A szimbólum használatát érdemes feljegyezni (pl. refaktoráláshoz)

SZIMBÓLUMTÁBLA

```
1  void f(int p) {
2    int x;
3    cin >> x;
4    cout << x+p+y;
5    int x;
6  }
```

Név	Fajta	Típus	Deklaráció	Használat
“f”	függvény	int → void	1. sor 6. oszlop	
“p”	paraméter	int	1. sor 12. oszlop	4. sor 13. oszlop
“x”	lokális változó	int	2. sor 7. oszlop	3. sor 10. oszlop 4. sor 11. oszlop

A szimbólumtábla műveletei:

- **Beszúrás**:
  - Deklaráció esetén
  - Az új szimbólum és adatai bekerülnek a szimbólumtáblába

- **Keresés**:
  - Szimbólum használatakor
  - A szimbólum neve a kulcs a kereséshez
  - A szimbólum használatát érdemes feljegyezni (pl. refaktoráláshoz)

SZIMBÓLUMTÁBLA

```
1  void f(int p) {
2    int x;
3    cin >> x;
4    cout << x+p+y;
5    int x;
6  }
```

Név	Fajta	Típus	Deklaráció	Használat
“f”	függvény	int → void	1. sor 6. oszlop	
“p”	paraméter	int	1. sor 12. oszlop	4. sor 13. oszlop
“x”	lokális változó	int	2. sor 7. oszlop	3. sor 10. oszlop 4. sor 11. oszlop

“y” nincs a szimbólumtáblában!

4. sor 15. oszlop:
szemantikus hiba:
“y” nincs deklarálva.

A szimbólumtábla műveletei:

- **Beszúrás**:
  - Deklaráció esetén
  - Az új szimbólum és adatai bekerülnek a szimbólumtáblába

- **Keresés**:
  - Szimbólum használatakor
  - A szimbólum neve a kulcs a kereséshez
  - A szimbólum használatát érdemes feljegyezni (pl. refaktoráláshoz)

SZIMBÓLUMTÁBLA

```
1  void f(int p) {
2    int x;
3    cin >> x;
4    cout << x+p+y;
5    int x;
6  }
```

Név	Fajta	Típus	Deklaráció	Használat
“f”	függvény	int → void	1. sor 6. oszlop	
“p”	paraméter	int	1. sor 12. oszlop	4. sor 13. oszlop
“x”	lokális változó	int	2. sor 7. oszlop	3. sor 10. oszlop 4. sor 11. oszlop

“x” már van a szimbólumtáblában!

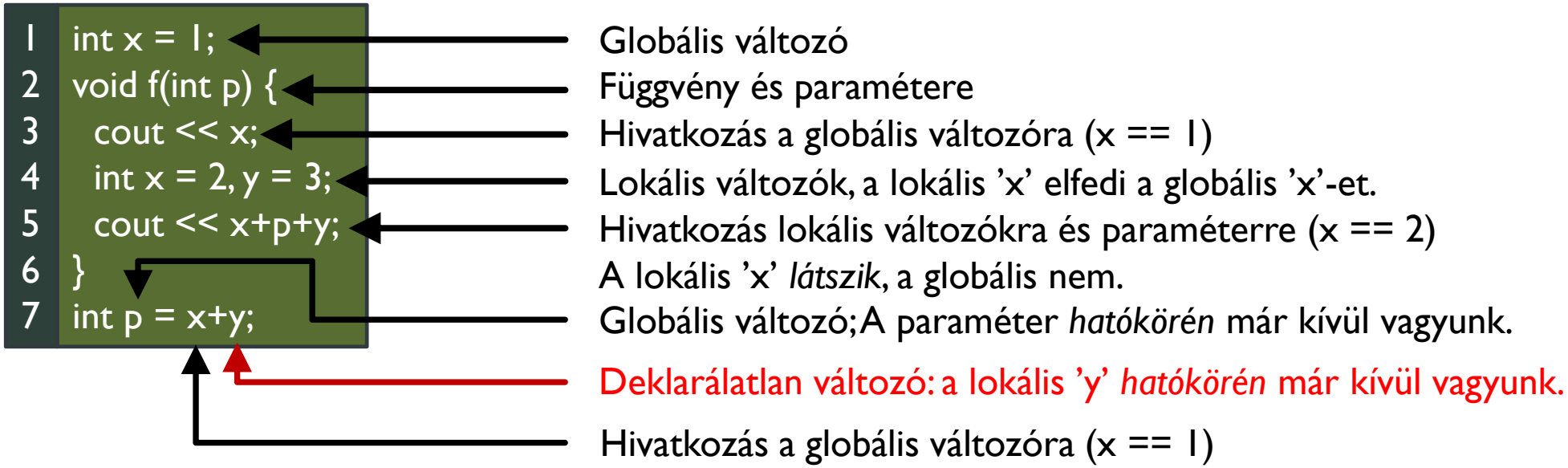
4. sor 15. oszlop:
szemantikus hiba:
“y” nincs deklarálva.

5. sor 7. oszlop:
szemantikus hiba:
“x” újradeklarálva

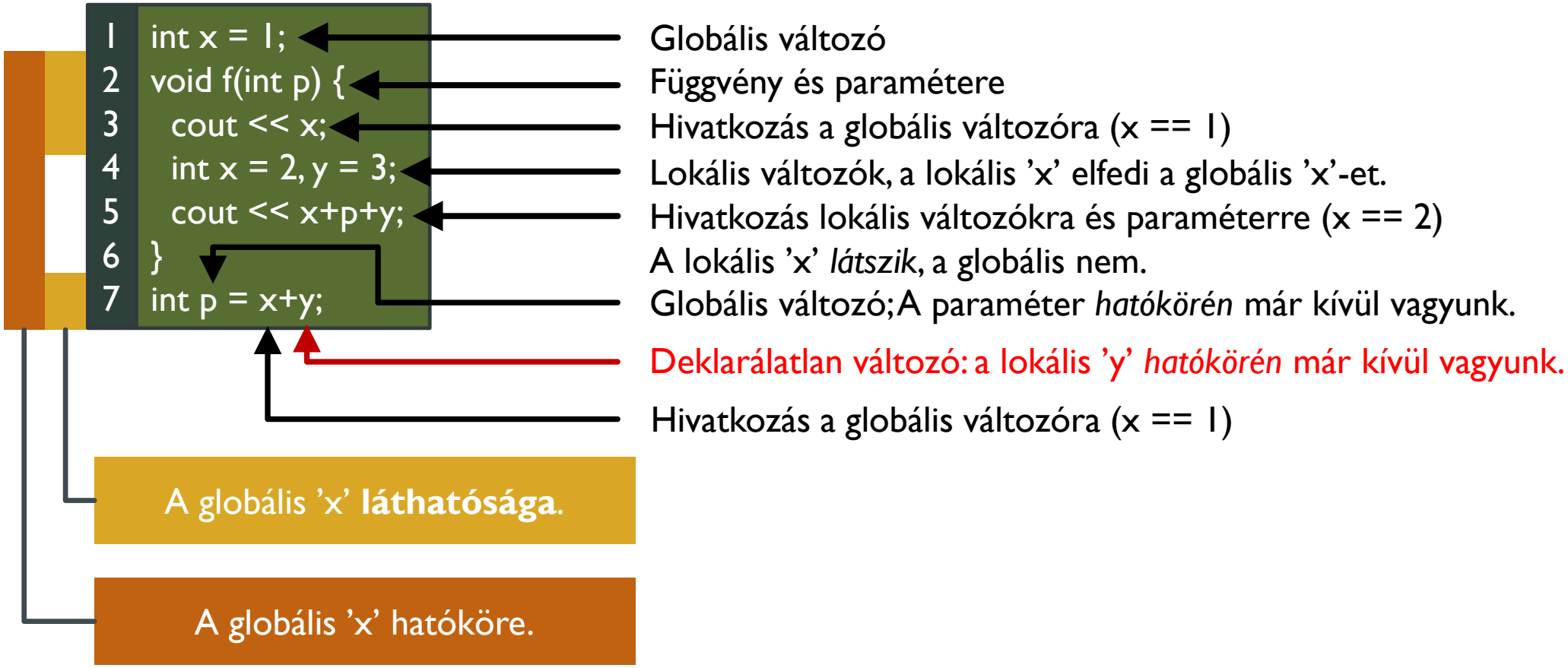
A szimbólumtábla műveletei:

- **Beszúrás**:
  - Deklaráció esetén
  - Az új szimbólum és adatai bekerülnek a szimbólumtáblába
  - A beszúrás mindig egy kereséssel kezdődik, hogy kiderüljön az újradeklarálás
- **Keresés**:
  - Szimbólum használatakor
  - A szimbólum neve a kulcs a kereséshez
  - A szimbólum használatát érdemes feljegyezni (pl. refaktoráláshoz)

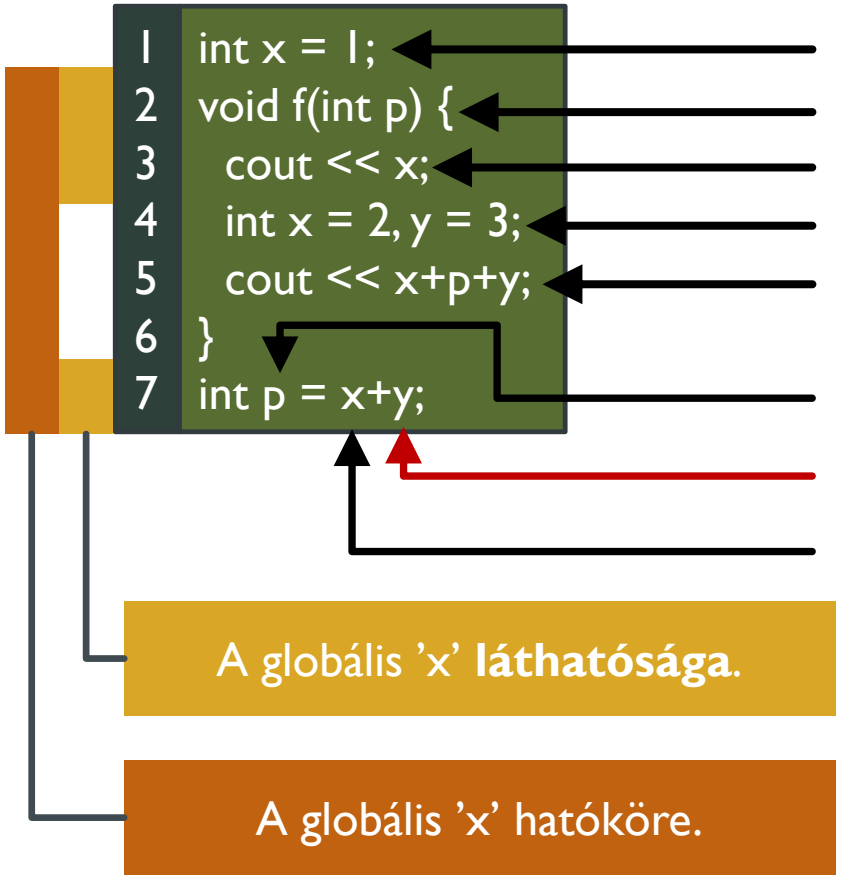
DEKLARÁCIÓK ÉS BLOKKSZERKEZET



DEKLARÁCIÓK ÉS BLOKKSZERKEZET: HATÓKÖR, LÁTHATÓSÁG



DEKLARÁCIÓK ÉS BLOKKSZERKEZET: HATÓKÖR, LÁTHATÓSÁG



Globális változó

Függvény és paramétere

Hivatkozás a globális változóra (x == 1)

Lokális változók, a lokális 'x' elfedi a globális 'x'-et.

Hivatkozás lokális változókra és paraméterre (x == 2)

A lokális 'x' *látszik*, a globális nem.

Globális változó; A paraméter *hatókörén* már kívül vagyunk.

Deklarálatlan változó: a lokális 'y' *hatókörén* már kívül vagyunk.

Hivatkozás a globális változóra (x == 1)

Hogyan valósítsuk meg a hatókör és láthatóság szabályait a szimbólumtábla segítségével?

VEREM SZERKEZETŰ SZIMBÓLUMTÁBLA

```
1  int x = 1;
2  void f(int p) {
3    cout << x;
4    int x = 2, y = 3;
5    cout << x+p+y;
6  }
7  int p = x+y;
```

Név	**Fajta**	**Típus**	**Deklaráció**

- **A szimbólumtábla egy verem.**

VEREM SZERKEZETŰ SZIMBÓLUMTÁBLA

```
1  int x = 1;
2  void f(int p) {
3    cout << x;
4    int x = 2, y = 3;
5    cout << x+p+y;
6  }
7  int p = x+y;
```

“x”	globális változó	int	1. sor 5. oszlop
Név	**Fajta**	**Típus**	**Deklaráció**

- A szimbólumtábla egy verem.
- **Az új szimbólumokat a verem tetejére helyezzük (*push*).**

VEREM SZERKEZETŰ SZIMBÓLUMTÁBLA

```
1  int x = 1;
2  void f(int p) {
3    cout << x;
4    int x = 2, y = 3;
5    cout << x+p+y;
6  }
7  int p = x+y;
```

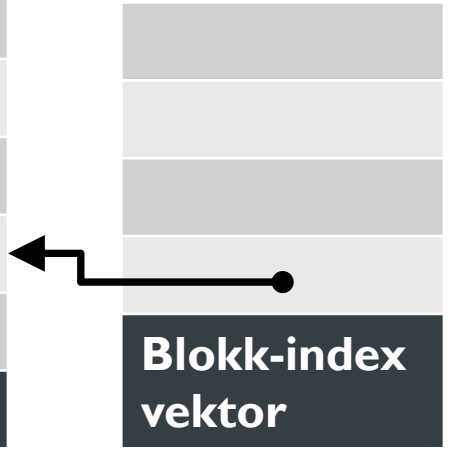
“f”	függvény	int → void	2. sor 6. oszlop
“x”	globális változó	int	1. sor 5. oszlop
Név	**Fajta**	**Típus**	**Deklaráció**

- A szimbólumtábla egy verem.
- **Az új szimbólumokat a verem tetejére helyezzük (*push*).**

VEREM SZERKEZETŰ SZIMBÓLUMTÁBLA

```
1  int x = 1;
2  void f(int p) {
3    cout << x;
4    int x = 2, y = 3;
5    cout << x+p+y;
6  }
7  int p = x+y;
```

Név	Fajta	Típus	Deklaráció
“x”	globális változó	int	1. sor 5. oszlop
“f”	függvény	int → void	2. sor 6. oszlop

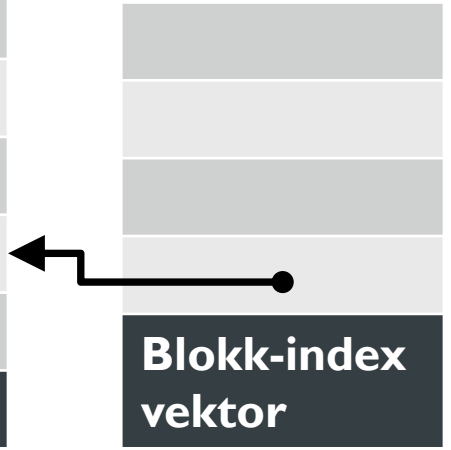


- A szimbólumtábla egy verem.
- Az új szimbólumokat a verem tetejére helyezzük (*push*).
- **A blokk-index vektor is egy verem.**
  Amikor új blokk kezdődik, a blokk-index vektorban megjelöljük a szimbólumtábla-verem tetejét.

VEREM SZERKEZETŰ SZIMBÓLUMTÁBLA

```
1  int x = 1;
2  void f(int p) {
3    cout << x;
4    int x = 2, y = 3;
5    cout << x+p+y;
6  }
7  int p = x+y;
```

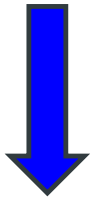
Név	Fajta	Típus	Deklaráció
“x”	globális változó	int	1. sor 5. oszlop
“f”	függvény	int → void	2. sor 6. oszlop
“p”	paraméter	int	2. sor 12. oszlop



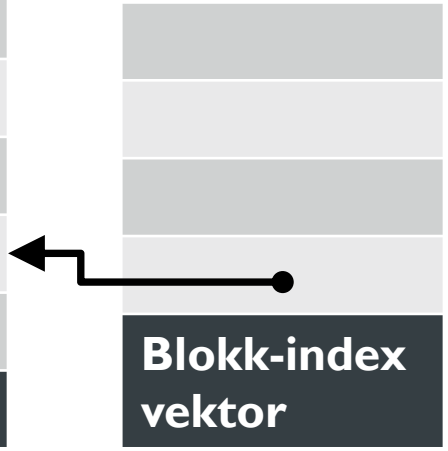
- A szimbólumtábla egy verem.
- **Az új szimbólumokat a verem tetejére helyezzük (*push*).**
- A blokk-index vektor is egy verem.
  Amikor új blokk kezdődik, a blokk-index vektorban megjelöljük a szimbólumtábla-verem tetejét.

VEREM SZERKEZETŰ SZIMBÓLUMTÁBLA

```
1  int x = 1;
2  void f(int p) {
3    cout << x;
4    int x = 2, y = 3;
5    cout << x+p+y;
6  }
7  int p = x+y;
```



Név	Fajta	Típus	Deklaráció
“x”	globális változó	int	1. sor 5. oszlop
“f”	függvény	int → void	2. sor 6. oszlop
“p”	paraméter	int	2. sor 12. oszlop

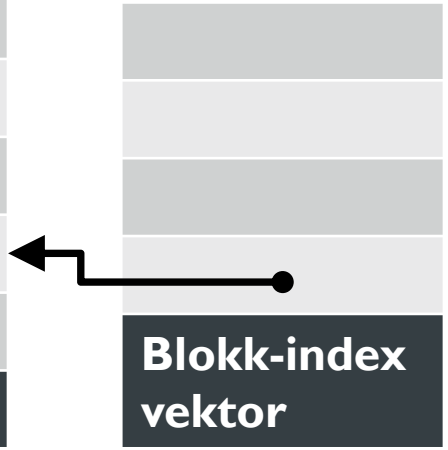


- A szimbólumtábla egy verem.
- Az új szimbólumokat a verem tetejére helyezzük (*push*).
- A blokk-index vektor is egy verem.
  Amikor új blokk kezdődik, a blokk-index vektorban megjelöljük a szimbólumtábla-verem tetejét.
- **Keresékor fentről lefelé keresünk a veremben, és az első találatnál megállunk.**

VEREM SZERKEZETŰ SZIMBÓLUMTÁBLA

```
1  int x = 1;
2  void f(int p) {
3    cout << x;
4    int x = 2, y = 3;
5    cout << x+p+y;
6  }
7  int p = x+y;
```

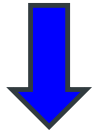
Név	Fajta	Típus	Deklaráció
“y”	lokális változó	int	4. sor 14. oszlop
“x”	lokális változó	int	4. sor 7. oszlop
“p”	paraméter	int	2. sor 12. oszlop
“f”	függvény	int → void	2. sor 6. oszlop
“x”	globális változó	int	1. sor 5. oszlop



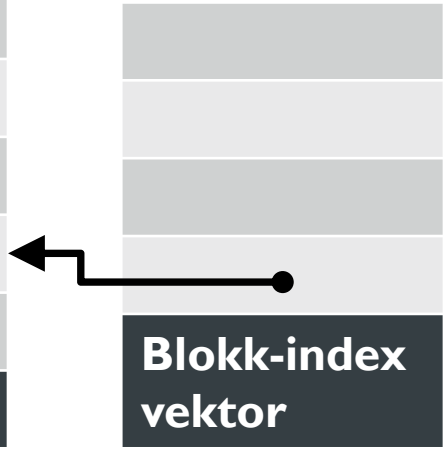
- A szimbólumtábla egy verem.
- **Az új szimbólumokat a verem tetejére helyezzük (*push*).**
- A blokk-index vektor is egy verem.
  Amikor új blokk kezdődik, a blokk-index vektorban megjelöljük a szimbólumtábla-verem tetejét.
- Keresékor fentről lefelé keresünk a veremben, és az első találatnál megállunk.

VEREM SZERKEZETŰ SZIMBÓLUMTÁBLA

```
1  int x = 1;
2  void f(int p) {
3    cout << x;
4    int x = 2, y = 3;
5    cout << x+p+y;
6  }
7  int p = x+y;
```



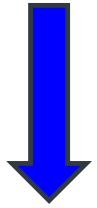
Név	Fajta	Típus	Deklaráció
“x”	globális változó	int	1. sor 5. oszlop
“f”	függvény	int → void	2. sor 6. oszlop
“p”	paraméter	int	2. sor 12. oszlop
“x”	lokális változó	int	4. sor 7. oszlop
“y”	lokális változó	int	4. sor 14. oszlop



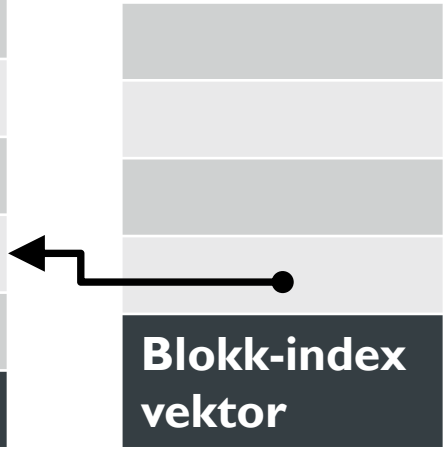
- A szimbólumtábla egy verem.
- Az új szimbólumokat a verem tetejére helyezzük (*push*).
- A blokk-index vektor is egy verem.
  Amikor új blokk kezdődik, a blokk-index vektorban megjelöljük a szimbólumtábla-verem tetejét.
- **Keresékor fentről lefelé keresünk a veremben, és az első találatnál megállunk.**

VEREM SZERKEZETŰ SZIMBÓLUMTÁBLA

```
1  int x = 1;
2  void f(int p) {
3    cout << x;
4    int x = 2, y = 3;
5    cout << x+p+y;
6  }
7  int p = x+y;
```



Név	Fajta	Típus	Deklaráció
“y”	lokális változó	int	4. sor 14. oszlop
“x”	lokális változó	int	4. sor 7. oszlop
“p”	paraméter	int	2. sor 12. oszlop
“f”	függvény	int → void	2. sor 6. oszlop
“x”	globális változó	int	1. sor 5. oszlop

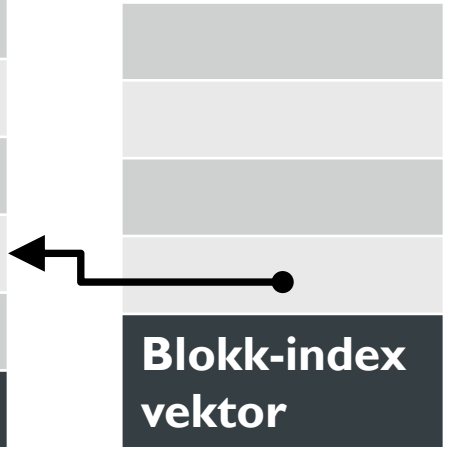


- A szimbólumtábla egy verem.
- Az új szimbólumokat a verem tetejére helyezzük (*push*).
- A blokk-index vektor is egy verem.
  Amikor új blokk kezdődik, a blokk-index vektorban megjelöljük a szimbólumtábla-verem tetejét.
- **Keresékor fentről lefelé keresünk a veremben, és az első találatnál megállunk.**

VEREM SZERKEZETŰ SZIMBÓLUMTÁBLA

```
1  int x = 1;
2  void f(int p) {
3    cout << x;
4    int x = 2, y = 3;
5    cout << x+p+y;
6  }
7  int p = x+y;
```

Név	Fajta	Típus	Deklaráció
“y”	lokális változó	int	4. sor 14. oszlop
“x”	lokális változó	int	4. sor 7. oszlop
“p”	paraméter	int	2. sor 12. oszlop
“f”	függvény	int → void	2. sor 6. oszlop
“x”	globális változó	int	1. sor 5. oszlop

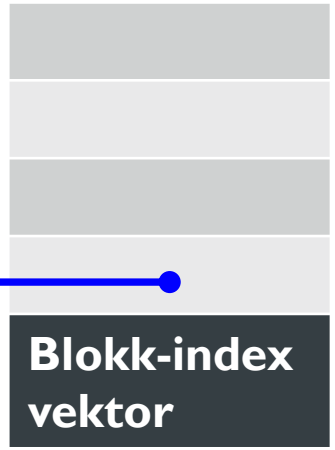


- A szimbólumtábla egy verem.
- Az új szimbólumokat a verem tetejére helyezzük (*push*).
- A blokk-index vektor is egy verem.
  Amikor új blokk kezdődik, a blokk-index vektorban megjelöljük a szimbólumtábla-verem tetejét.
- **Keresékor fentről lefelé keresünk a veremben, és az első találatnál megállunk.**

VEREM SZERKEZETŰ SZIMBÓLUMTÁBLA

```
1  int x = 1;
2  void f(int p) {
3    cout << x;
4    int x = 2, y = 3;
5    cout << x+p+y;
6  }
7  int p = x+y;
```

Név	Fajta	Típus	Deklaráció
“x”	globális változó	int	1. sor 5. oszlop
“f”	függvény	int → void	2. sor 6. oszlop
“p”	paraméter	int	2. sor 12. oszlop
“x”	lokális változó	int	4. sor 7. oszlop
“y”	lokális változó	int	4. sor 14. oszlop

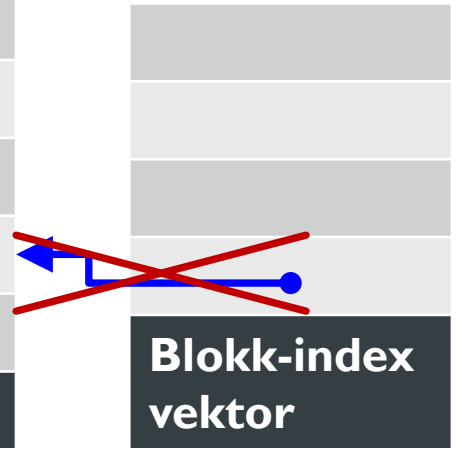


- A szimbólumtábla egy verem.
- Az új szimbólumokat a verem tetejére helyezzük (*push*).
- A blokk-index vektor is egy verem.
  Amikor új blokk kezdődik, a blokk-index vektorban megjelöljük a szimbólumtábla-verem tetejét.
- Keresékor fentről lefelé keresünk a veremben, és az első találatnál megállunk.
- **Blokk végén:**
  - **Eltávolítjuk a szimbólumokat a blokk-index vektor legfelső jelöléséig (néhány *pop*).**

VEREM SZERKEZETŰ SZIMBÓLUMTÁBLA

```
1  int x = 1;
2  void f(int p) {
3    cout << x;
4    int x = 2, y = 3;
5    cout << x+p+y;
6  }
7  int p = x+y;
```

Név	Fajta	Típus	Deklaráció
“x”	globális változó	int	1. sor 5. oszlop
“f”	függvény	int → void	2. sor 6. oszlop



- A szimbólumtábla egy verem.
- Az új szimbólumokat a verem tetejére helyezzük (*push*).
- A blokk-index vektor is egy verem.
  Amikor új blokk kezdődik, a blokk-index vektorban megjelöljük a szimbólumtábla-verem tetejét.
- Keresékor fentről lefelé keresünk a veremben, és az első találatnál megállunk.
- Blokk végén:
  - Eltávolítjuk a szimbólumokat a blokk-index vektor legfelső jelöléséig (néhány *pop*).
  - **Végül a jelölést is eltávolítjuk a blokk-index vektorból (*pop*).**

VEREM SZERKEZETŰ SZIMBÓLUMTÁBLA

```
1  int x = 1;
2  void f(int p) {
3    cout << x;
4    int x = 2, y = 3;
5    cout << x+p+y;
6  }
7  int p = x+y;
```

Név	Fajta	Típus	Deklaráció
“x”	globális változó	int	1. sor 5. oszlop
“f”	függvény	int → void	2. sor 6. oszlop

Blokk-index vektor

- A szimbólumtábla egy verem.
- Az új szimbólumokat a verem tetejére helyezzük (*push*).
- A blokk-index vektor is egy verem.
  Amikor új blokk kezdődik, a blokk-index vektorban megjelöljük a szimbólumtábla-verem tetejét.
- Keresékor fentről lefelé keresünk a veremben, és az első találatnál megállunk.
- Blokk végén:
  - Eltávolítjuk a szimbólumokat a blokk-index vektor legfelső jelöléséig (néhány *pop*).
  - Végül a jelölést is eltávolítjuk a blokk-index vektorból (*pop*).

VEREM SZERKEZETŰ SZIMBÓLUMTÁBLA

```
1  int x = 1;
2  void f(int p) {
3    cout << x;
4    int x = 2, y = 3;
5    cout << x+p+y;
6  }
7  int p = x+y;
```

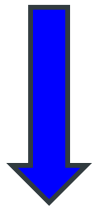
Név	Fajta	Típus	Deklaráció
“x”	globális változó	int	1. sor 5. oszlop
“f”	függvény	int → void	2. sor 6. oszlop
“p”	globális változó	int	7. sor 5. oszlop

Blokk-index vektor

- A szimbólumtábla egy verem.
- **Az új szimbólumokat a verem tetejére helyezzük (*push*).**
- A blokk-index vektor is egy verem.
  Amikor új blokk kezdődik, a blokk-index vektorban megjelöljük a szimbólumtábla-verem tetejét.
- Keresékor fentről lefelé keresünk a veremben, és az első találatnál megállunk.
- Blokk végén:
  - Eltávolítjuk a szimbólumokat a blokk-index vektor legfelső jelöléséig (néhány *pop*).
  - Végül a jelölést is eltávolítjuk a blokk-index vektorból (*pop*).

VEREM SZERKEZETŰ SZIMBÓLUMTÁBLA

```
1  int x = 1;
2  void f(int p) {
3    cout << x;
4    int x = 2, y = 3;
5    cout << x+p+y;
6  }
7  int p = x+y;
```



“p”	globális változó	int	7. sor 5. oszlop
“f”	függvény	int → void	2. sor 6. oszlop
“x”	globális változó	int	1. sor 5. oszlop
Név	**Fajta**	**Típus**	**Deklaráció**

Blokk-index vektor

- A szimbólumtábla egy verem.
- Az új szimbólumokat a verem tetejére helyezzük (*push*).
- A blokk-index vektor is egy verem.
  Amikor új blokk kezdődik, a blokk-index vektorban megjelöljük a szimbólumtábla-verem tetejét.
- **Keresékor fentről lefelé keresünk a veremben, és az első találatnál megállunk.**
- Blokk végén:
  - Eltávolítjuk a szimbólumokat a blokk-index vektor legfelső jelöléséig (néhány *pop*).
  - Végül a jelölést is eltávolítjuk a blokk-index vektorból (*pop*).

VEREM SZERKEZETŰ SZIMBÓLUMTÁBLA

```
1  int x = 1;
2  void f(int p) {
3    cout << x;
4    int x = 2, y = 3;
5    cout << x+p+y;
6  }
7  int p = x+y;
```

Név	Fajta	Típus	Deklaráció
“x”	globális változó	int	1. sor 5. oszlop
“f”	függvény	int → void	2. sor 6. oszlop
“p”	globális változó	int	7. sor 5. oszlop

Blokk-index vektor

“y” nem található
→ szemantikus hiba: nem deklarált változó

- A szimbó
- Az új ük (*push*).
- A blokk
  Amikor új blokk kezdődik, a blokk-index vektorban megjelöljük a szimbólumtábla-verem tetejét.
- **Keresékor fentről lefelé keresünk a veremben, és az első találatnál megállunk.**
- Blokk végén:
  - Eltávolítjuk a szimbólumokat a blokk-index vektor legfelső jelöléséig (néhány *pop*).
  - Végül a jelölést is eltávolítjuk a blokk-index vektorból (*pop*).

ÚJRADEKLARÁLÁS

- Deklaráció feldolgozásakor ellenőrizni kell, hogy nem újradeklarált változóról van-e szó.
- Csak ez után szabad beszúrni a szimbólumot és adatait a táblázatba.
- A verem szerkezetű szimbólumtáblában hogyan kell elvégezni ezt az ellenőrzést?

```
1  int x = 1;
2  void f(int p) {
3    int x = 2;
4    int p = 3;
5  }
6  int p;
```

Ez rendben. Elfedi a globális 'x' változót.

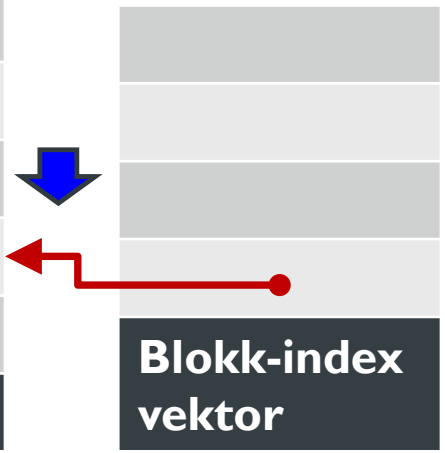
Ez újradeklarálás, szemantikus hiba.

Ez is rendben. Kívül vagyunk a 'p' paraméter hatókörén.

ÚJRADEKLARÁLÁS ELLENŐRZÉSE VEREM SZERKEZETŰ SZIMBÓLUMTÁBLÁBAN

```
1  int x = 1;
2  void f(int p) {
3    int x = 2;
4    int p = 3;
5  }
6  int p;
```

Név	Fajta	Típus	Deklaráció
“p”	paraméter	int	2. sor 12. oszlop
“f”	függvény	int → void	2. sor 6. oszlop
“x”	globális változó	int	1. sor 5. oszlop

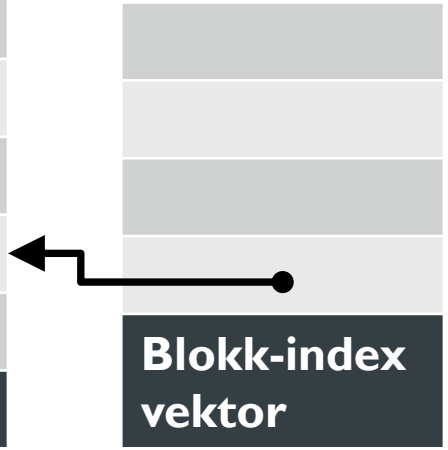


- **Deklaráció ellenőrzésekor csak a blokk-index vektor legfelső bejegyzése által mutatott rekord fölött keresünk, azaz az aktuális blokk szimbólumai között.**

ÚJRADEKLARÁLÁS ELLENŐRZÉSE VEREM SZERKEZETŰ SZIMBÓLUMTÁBLÁBAN

```
1  int x = 1;
2  void f(int p) {
3    int x = 2;
4    int p = 3;
5  }
6  int p;
```

Név	Fajta	Típus	Deklaráció
“x”	lokális változó	int	3. sor 7. oszlop
“p”	paraméter	int	2. sor 12. oszlop
“f”	függvény	int → void	2. sor 6. oszlop
“x”	globális változó	int	1. sor 5. oszlop

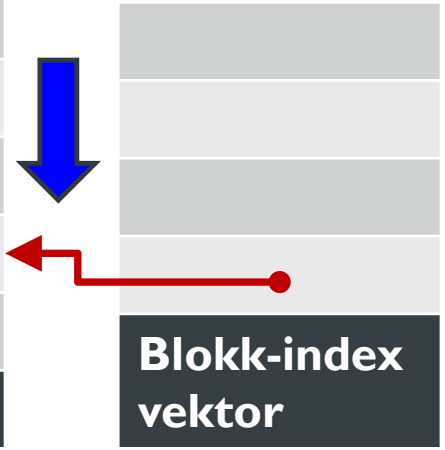


- Deklaráció ellenőrzésekor csak a blokk-index vektor legfelső bejegyzése által mutatott rekord fölött keresünk, azaz az aktuális blokk szimbólumai között.
- **Ha nincs hiba, a szimbólum beszúrható a táblába.**

ÚJRADEKLARÁLÁS ELLENŐRZÉSE VEREM SZERKEZETŰ SZIMBÓLUMTÁBLÁBAN

```
1  int x = 1;
2  void f(int p) {
3    int x = 2;
4    int p = 3;
5  }
6  int p;
```

Név	Fajta	Típus	Deklaráció
“x”	lokális változó	int	3. sor 7. oszlop
“p”	paraméter	int	2. sor 12. oszlop
“f”	függvény	int → void	2. sor 6. oszlop
“x”	globális változó	int	1. sor 5. oszlop



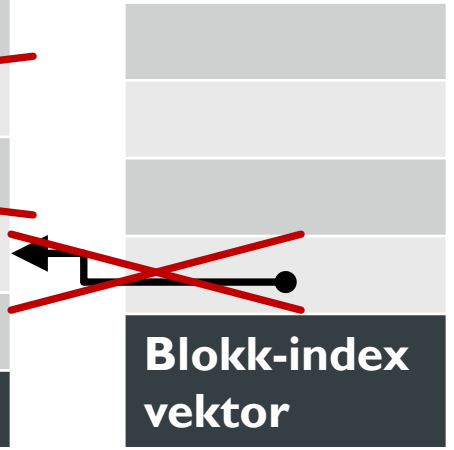
- **Deklaráció ellenőrzésekor csak a blokk-index vektor legfelső bejegyzése által mutatott rekord fölött keresünk, azaz az aktuális blokk szimbólumai között.**
- Ha nincs hiba, a szimbólum beszúrható a táblába.

“p” már van az aktuális blokk szimbólumai között: → szemantikus hiba: újradeklarált változó

ÚJRADEKLARÁLÁS ELLENŐRZÉSE VEREM SZERKEZETŰ SZIMBÓLUMTÁBLÁBAN

```
1  int x = 1;
2  void f(int p) {
3    int x = 2;
4    int p = 3;
5  }
6  int p;
```

Név	Fajta	Típus	Deklaráció
“x”	lokális változó	int	3. sor 7. oszlop
“p”	paraméter	int	2. sor 12. oszlop
“f”	függvény	int → void	2. sor 6. oszlop
“x”	globális változó	int	1. sor 5. oszlop



- Deklaráció ellenőrzésekor csak a blokk-index vektor legfelső bejegyzése által mutatott rekord fölött keresünk, azaz az aktuális blokk szimbólumai között.
- Ha nincs hiba, a szimbólum beszúrható a táblába.
- **Blokk végén:**
  - **Eltávolítjuk a szimbólumokat a blokk-index vektor legfelső jelöléséig (néhány *pop*).**
  - **Végül a jelölést is eltávolítjuk a blokk-index vektorból (*pop*).**

ÚJRADEKLARÁLÁS ELLENŐRZÉSE VEREM SZERKEZETŰ SZIMBÓLUMTÁBLÁBAN

```
1  int x = 1;
2  void f(int p) {
3    int x = 2;
4    int p = 3;
5  }
6  int p;
```

Név	Fajta	Típus	Deklaráció
“f”	függvény	int → void	2. sor 6. oszlop
“x”	globális változó	int	1. sor 5. oszlop

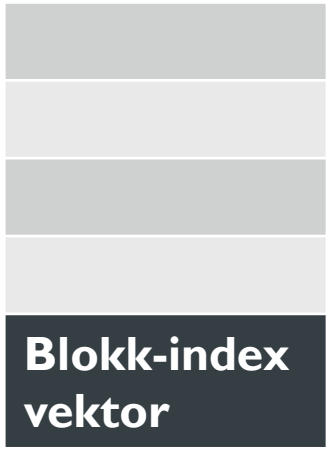
Blokk-index vektor

- Deklaráció ellenőrzésekor csak a blokk-index vektor legfelső bejegyzése által mutatott rekord fölött keresünk, azaz az aktuális blokk szimbólumai között.
- Ha nincs hiba, a szimbólum beszúrható a táblába.

ÚJRADEKLARÁLÁS ELLENŐRZÉSE VEREM SZERKEZETŰ SZIMBÓLUMTÁBLÁBAN

```
1  int x = 1;
2  void f(int p) {
3    int x = 2;
4    int p = 3;
5  }
6  int p;
```

“f”	függvény	int → void	2. sor 6. oszlop
“x”	globális változó	int	1. sor 5. oszlop
Név	**Fajta**	**Típus**	**Deklaráció**



- Deklaráció ellenőrzésekor csak a blokk-index vektor legfelső bejegyzése által mutatott rekord fölött keresünk, azaz az aktuális blokk szimbólumai között.
  - **Ha a blokk-index vektor üres, akkor az egész szimbólumtáblában keresünk.**
- Ha nincs hiba, a szimbólum beszúrható a táblába.

ÚJRADEKLARÁLÁS ELLENŐRZÉSE VEREM SZERKEZETŰ SZIMBÓLUMTÁBLÁBAN

```
1  int x = 1;
2  void f(int p) {
3    int x = 2;
4    int p = 3;
5  }
6  int p;
```

Név	Fajta	Típus	Deklaráció
“x”	globális változó	int	1. sor 5. oszlop
“f”	függvény	int → void	2. sor 6. oszlop
“p”	globális változó	int	6. sor 5. oszlop

Blokk-index vektor

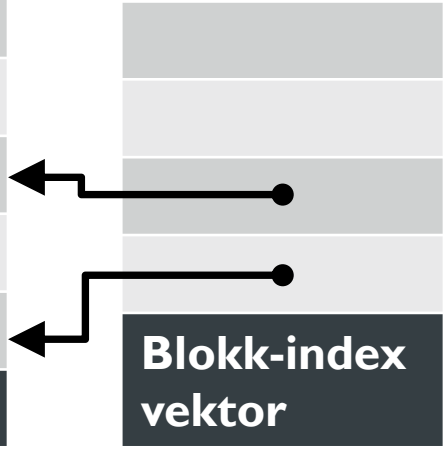
- Deklaráció ellenőrzésekor csak a blokk-index vektor legfelső bejegyzése által mutatott rekord fölött keresünk, azaz az aktuális blokk szimbólumai között.
  - Ha a blokk-index vektor üres, akkor az egész szimbólumtáblában keresünk.
- **Ha nincs hiba, a szimbólum beszúrható a táblába.**

TÖBB MÉLYSÉGŰ BLOKKSZERKEZET VEREM SZERKEZETŰ SZIMBÓLUMTÁBLÁBAN

- Blokk fajták: függvény, ciklus, elágazás, névtelen blokk...
- Minden blokk megkezdésekor új mutató kerül a blokk-index vektor tetejére.

```
1  int a;
2  {
3    bool b;
4    char c;
5    {
6      double d;
7    }
8  }
```

“d”	lokális változó	double	6. sor 12. oszlop
“c”	lokális változó	char	4. sor 8. oszlop
“b”	lokális változó	bool	3. sor 8. oszlop
“a”	globális változó	int	1. sor 5. oszlop
Név	**Fajta**	**Típus**	**Deklaráció**



ATTRIBÚTUM NYELVTAN

TÍPUSOKKAL KAPCSOLATOS ELLENŐRZÉSEK

TÍPUSOK SZEREPE

- Típusrendszerek a programhibák felderítésének legfontosabb eszközei
- A típusok jelölik ki, milyen műveletek végezhetők az adatokkal
- Típus néküli programozási nyelvek: pl. a legtöbb assembly nyelv
- Alaptípusok: bool, int, char, ...
- Összetett típusok: tömb, rekord, mutató és referencia, osztály, interfész, unió, algebrai adattípus, ...

MIKOR TÖRTÉNIK AZ ELLENŐRZÉS?

- Fordítási időben → *Statikus típusozás*
  - Futás közben már csak az értékeket kell tárolni, típusinformációt nem
  - Ha a program lefordul, típusokkal kapcsolatos hiba nem történhet futás közben: biztonságosabb megoldás
  - Ada, C++, Haskell ...
- Futási időben → *Dinamikus típusozás*
  - Futási időben az értékek mellett típusokat is kell tárolni
  - Az utasítások végrehajtása előtt kell ellenőrizni a típusokat
  - Futás közben derülnek ki a típushibák, cserébe hajlékonyabbak az ilyen nyelvek
  - Lisp, Erlang ...

- A statikusan típusos nyelvek is használnak dinamikus technikákat:
  - dinamikus kötés, Java instanceof operátora

KI ADJA MEG A TÍPUSOKAT?

- Programozó adja meg → *Típusellenőrzés*
  - A deklarációk típusozottak
  - A kifejezések egyszerű szabályok alapján típusozhatók
  - Egyszerűbb fordítóprogram, gyorsabb fordítás
- Fordítóprogram találja ki → *Típuslevezetés, típuskikövetkeztetés*
  - A deklarációkhoz (általában) nem kell típust megadni
  - A kifejezések típusát a fordítóprogram „találja ki” a műveletek alapján
  - Kényelmesebb a programozónak
    - Azonban ajánlott típusozni a deklarációkat, hogy olvashatóbb legyen a kód

C++

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

Haskell

```
factorial n =
  if n == 0
  then 1
  else n * factorial (n-1)
```

TÍPUSKONVERZIÓK

- Kifejezés típusának megváltoztatása
- Automatikus vagy explicit
- Osztályhierarchiához kapcsolódó típuskonverziók (Liskov-féle helyettesítési elv)
- A típuskonverziókkal a kódgenerátornak is törődnie kell: adatkonverzióra is szükség lehet

```
void f(double d);
void g(int i);
...
int x = 5;
double y = 3.14;
f(x);
g((int)y);
```

ATTRIBÚTUMNYELVTANOK

- A szintaxist leíró nyelvtan szimbólumaihoz *attribútumokat* rendelünk
  - A szemantikus elemzés vagy a kódgenerálás, kódoptimalizálás számára fontos, kiegészítő információk
- A szabályokhoz *akciókat* (programkód részleteket) rendelünk
  - Meglévő attribútumértékekből újabb attribútumok értékeit számolják ki
  - Ellenőrzéseket végeznek, szemantikus hibákat jeleznek

ATTRIBÚTUMNYELVTANOK: PÉLDA

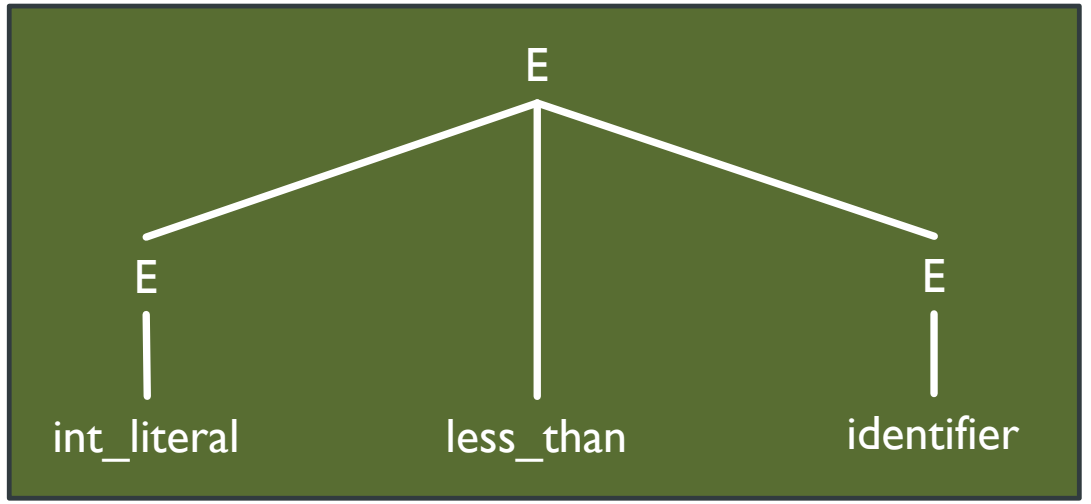
Nyelvtan

E → int_literal

E → identifier

E → E less_than E

Szintaxisfa



10 < x

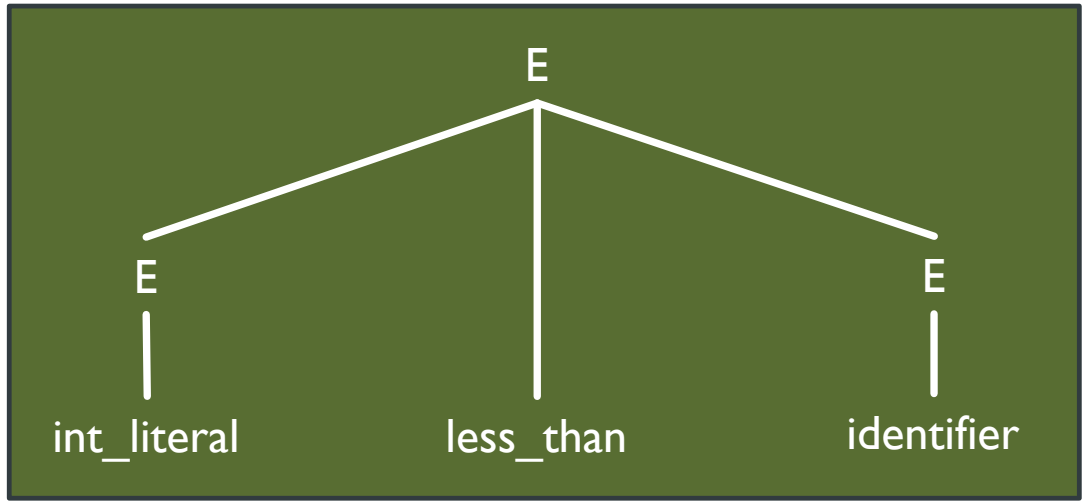
Forrásszöveg

ATTRIBÚTUMNYELVTANOK: PÉLDA

E → int_literal

E → identifier

E → E less_than E



10 < x

Attribútumok:

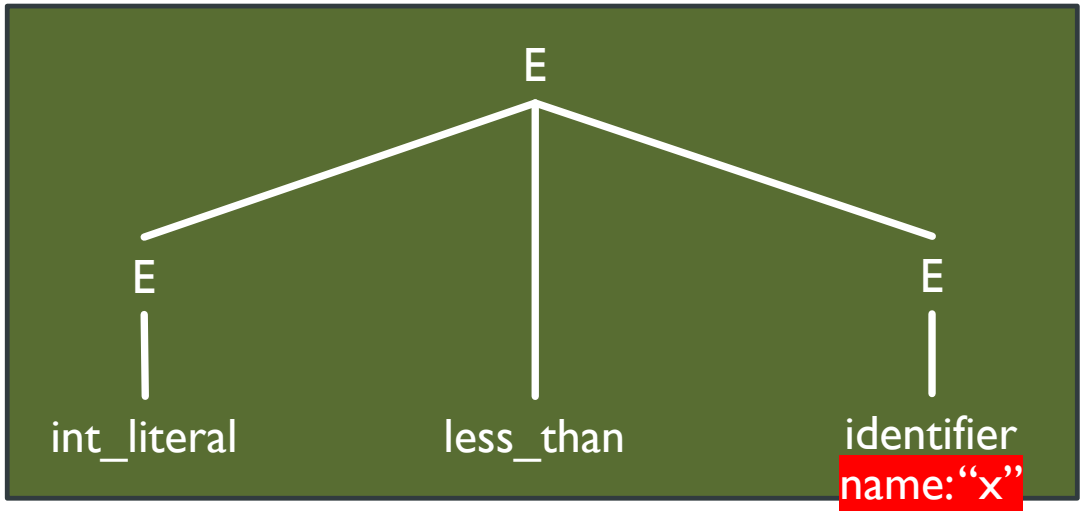
- *E.type*: A kifejezés típusa (pl. 'int', 'bool')
- *identifier.name*: Az azonosító neve (szöveg)

ATTRIBÚTUMNYELVTANOK: PÉLDA

E → int_literal

E → identifier

E → E less_than E



10 x

Az azonosítók nevét a lexikális elemző adja meg.

Attribútumok:

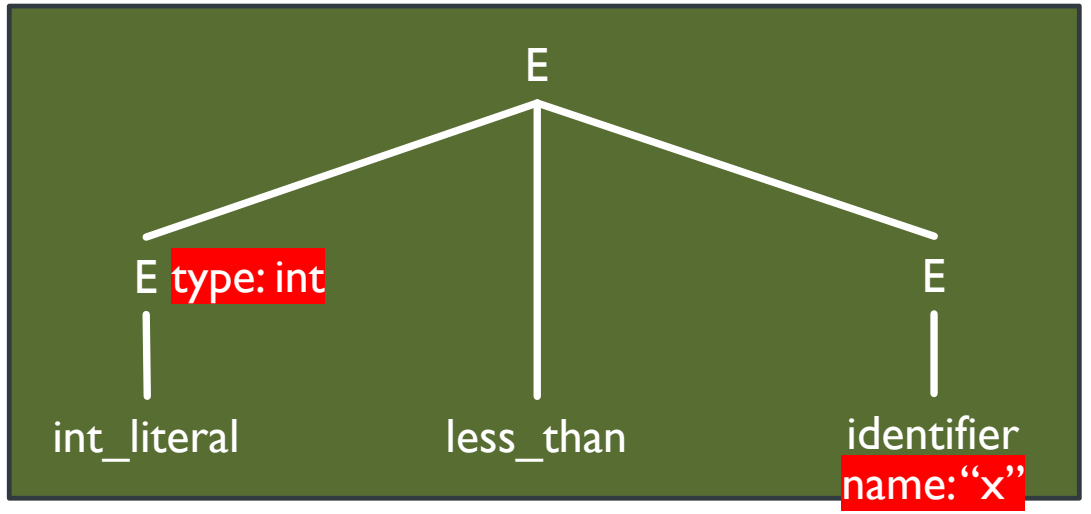
- *E.type*: A kifejezés típusa (pl. 'int', 'bool')
- *identifier.name*: Az azonosító neve (szöveg)

ATTRIBÚTUMNYELVTANOK: PÉLDA

E → int_literal {
  E.type = int
}
E → identifier

Akció, amely az egyetlen egészszám-literálból álló kifejezés típusát számolja ki.

E → E less_than E



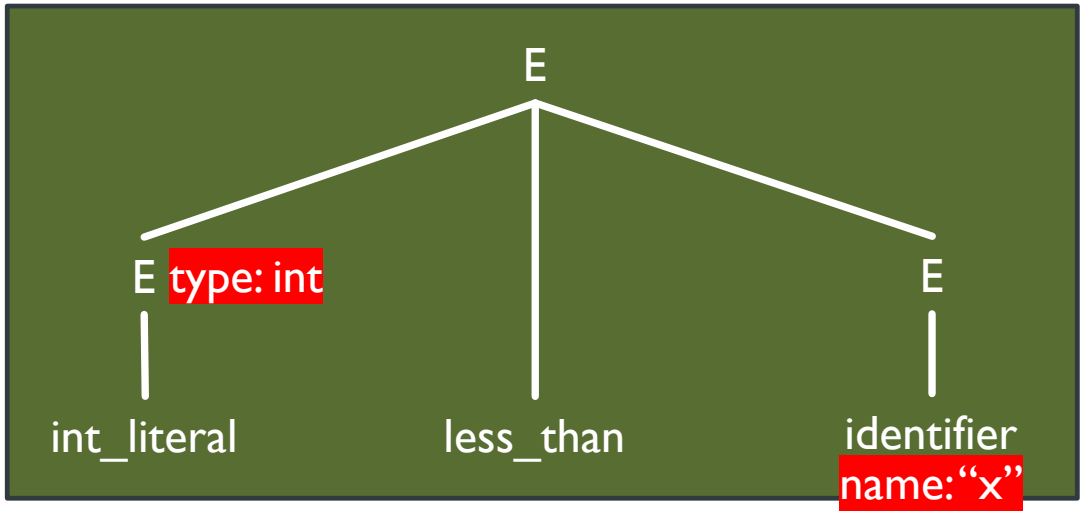
10 < x

Attribútumok:
- *E.type*: A kifejezés típusa (pl. 'int', 'bool')
- *identifier.name*: Az azonosító neve (szöveg)

ATTRIBÚTUMNYELVTANOK: PÉLDA

```
E → int_literal {
  E.type = int
}
E → identifier {
  if identifier.name not in symbol_table then
    error(...);
}
E → E less_than E
```

Ellenőrzés:
Deklarálva volt-e az azonosító?



10 < x

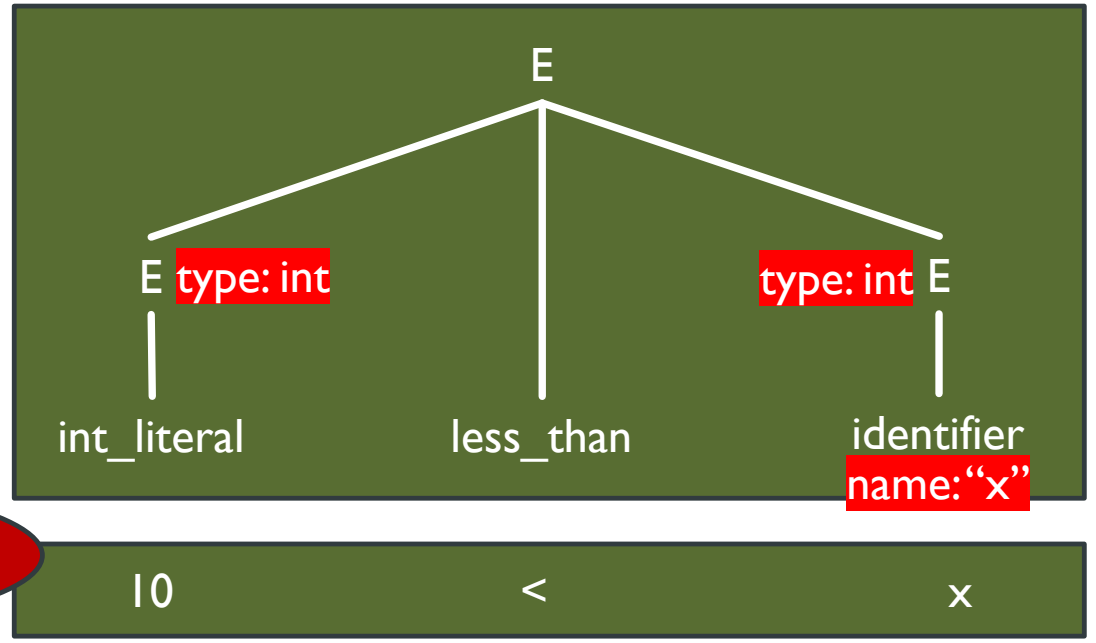
Attribútumok:

- *E.type*: A kifejezés típusa (pl. 'int', 'bool')
- *identifier.name*: Az azonosító neve (szöveg)

ATTRIBÚTUMNYELVTANOK: PÉLDA

```
E → int_literal {
  E.type = int
}
E → identifier {
  if identifier.name not in symbol_table then
    error(...);
  else
    E.type = symbol_table.get_type(identifier.name);
}
E → E less_than E
```

Azonosító kifejezés típusának beállítása.



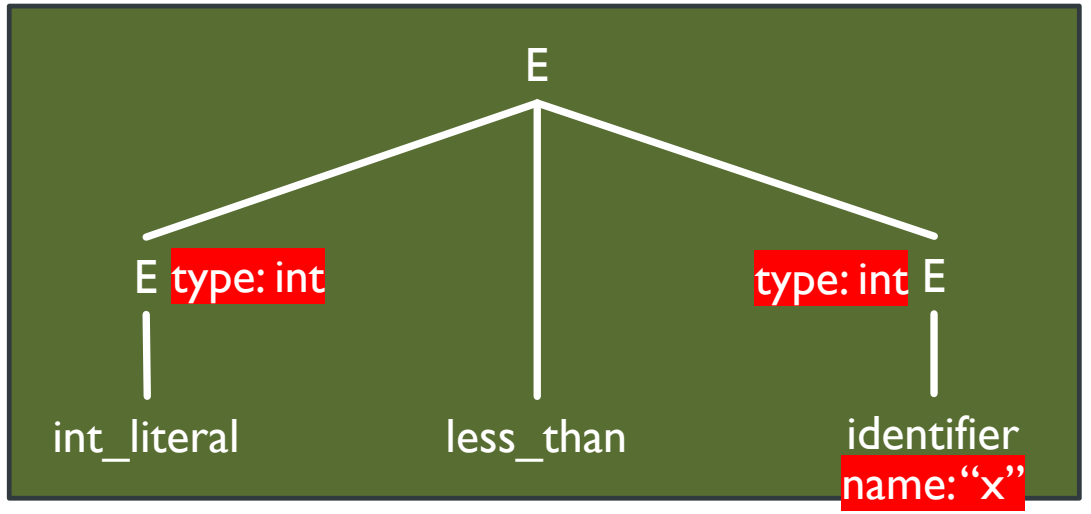
Attribútumok:

- *E.type*: A kifejezés típusa (pl. 'int', 'bool')
- *identifier.name*: Az azonosító neve (szöveg)

ATTRIBÚTUMNYELVTANOK: PÉLDA

```
E → int_literal {
  E.type = int
}
E → identifier {
  if identifier.name not in symbol_table then
    error(...);
  else
    E.type = symbol_table.get_type(identifier.name);
}
E1 → E2 less_than E3
```

Sorszámozzuk a szimbólumokat, hogy tudjuk, melyiknek az attribútumára hivatkozunk.



Attribútumok:

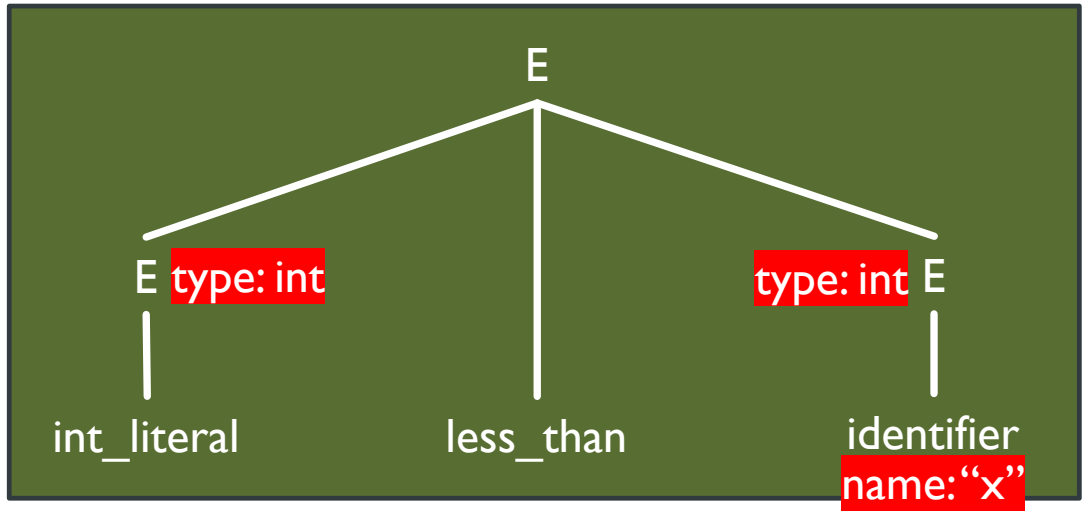
- *E.type*: A kifejezés típusa (pl. 'int', 'bool')
- *identifier.name*: Az azonosító neve (szöveg)

ATTRIBÚTUMNYELVTANOK: PÉLDA

```
E → int_literal {
  E.type = int
}
E → identifier {
  if identifier.name not in symbol_table then
    error(...);
  else
    E.type = symbol_table.get_type(identifier.name);
}
E1 → E2 less_than E3 {
  if E2.type != int || E3.type != int then
    error(...);

}
```

Ellenőrzés:
Megfelelő típusúak-e a '<' operátor paraméterei?

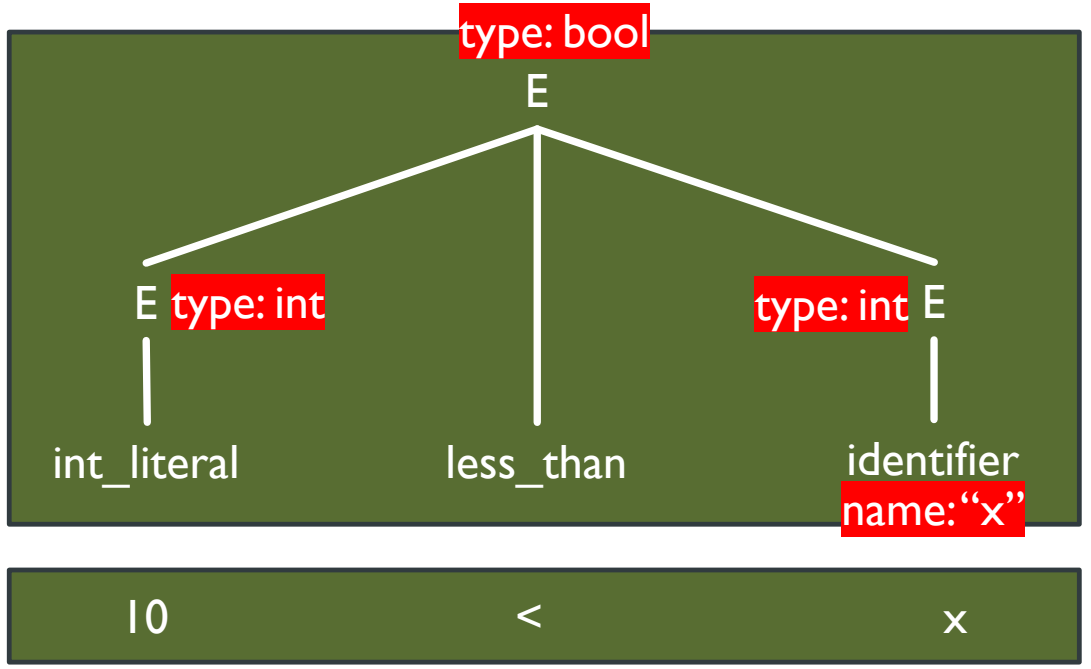


Attribútumok:
- *E.type*: A kifejezés típusa (pl. 'int', 'bool')
- *identifier.name*: Az azonosító neve (szöveg)

ATTRIBÚTUMNYELVTANOK: PÉLDA

```
E → int_literal {
  E.type = int
}
E → identifier {
  if identifier.name not in symbol_table then
    error(...);
  else
    E.type = symbol_table.get_type(identifier.name);
}
E1 → E2 less_than E3 {
  if E2.type != int || E3.type != int then
    error(...);
  else
    E1.type = bool;
}
```

Az összetett kifejezés típusának kiszámítása.



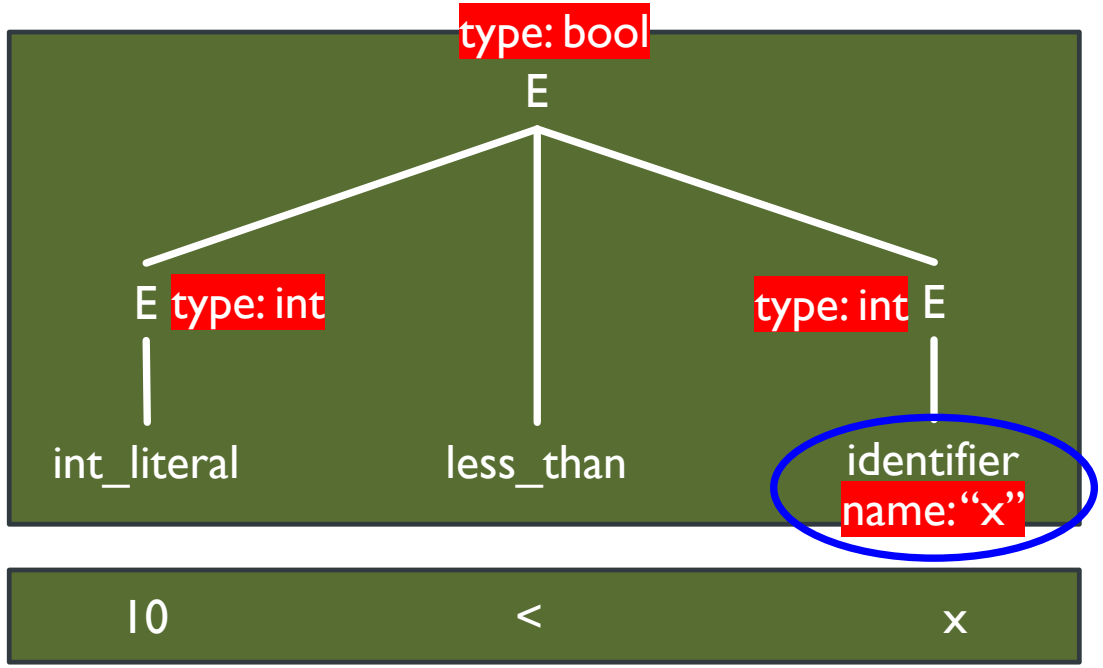
Attribútumok:

- *E.type*: A kifejezés típusa (pl. 'int', 'bool')
- *identifier.name*: Az azonosító neve (szöveg)

KITÜNTETETT SZINTETIZÁLT ATTRIBÚTUMOK

Kitüntetett szintetizált attribútum:
- Terminális szimbólum attribútuma.
- Kiszámításához nincs szükség más attribútumra.
- A lexikális elemző is meghatározhatja.

```
E → int_literal {
  E.type = int
}
E → identifier {
  if identifier.name not in symbol_table then
    error(...);
  else
    E.type = symbol_table.get_type(identifier.name);
}
E1 → E2 less_than E3 {
  if E2.type != int || E3.type != int then
    error(...);
  else
    E1.type = bool;
}
```



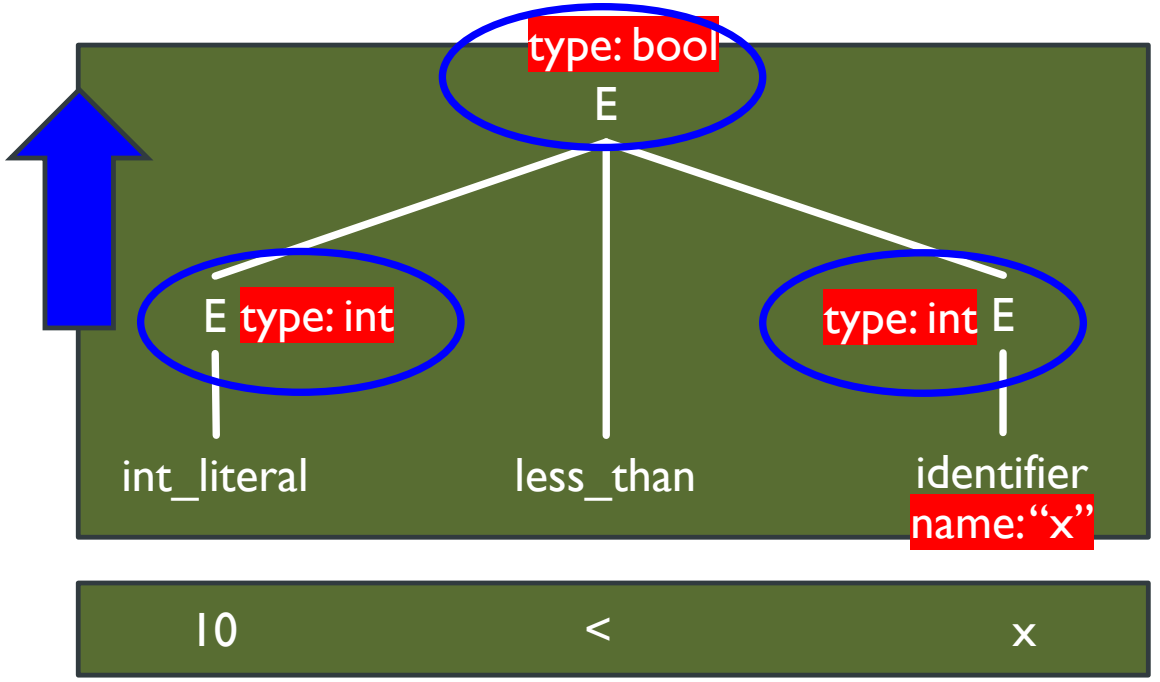
Attribútumok:
- *E.type*: A kifejezés típusa (pl. 'int', 'bool')
- *identifier.name*: Az azonosító neve (szöveg)

Kitüntetett szintetizált

SZINTETIZÁLT ATTRIBÚTUMOK

Szintetizált attribútum:
- A szabály bal oldalán áll, amikor kiszámítjuk.
- Alulról felfelé közvetít információt a szintaxisfában.

```
E → int_literal {
  E.type = int
}
E → identifier {
  if identifier.name not in symbol_table then
    error(...);
  else
    E.type = symbol_table.get_type(identifier.name);
}
E1 → E2 less_than E3 {
  if E2.type != int || E3.type != int then
    error(...);
  else
    E1.type = bool;
}
```



Attribútumok:
- *E.type*: A kifejezés típusa (pl. 'int', 'bool')
- *identifier.name*: Az azonosító neve (szöveg)

Szintetizált

Kitüntetett szintetizált

ATTRIBÚTUMNYELVTANOK: MÁSIK PÉLDA

Nyelvtan

$S \rightarrow L$

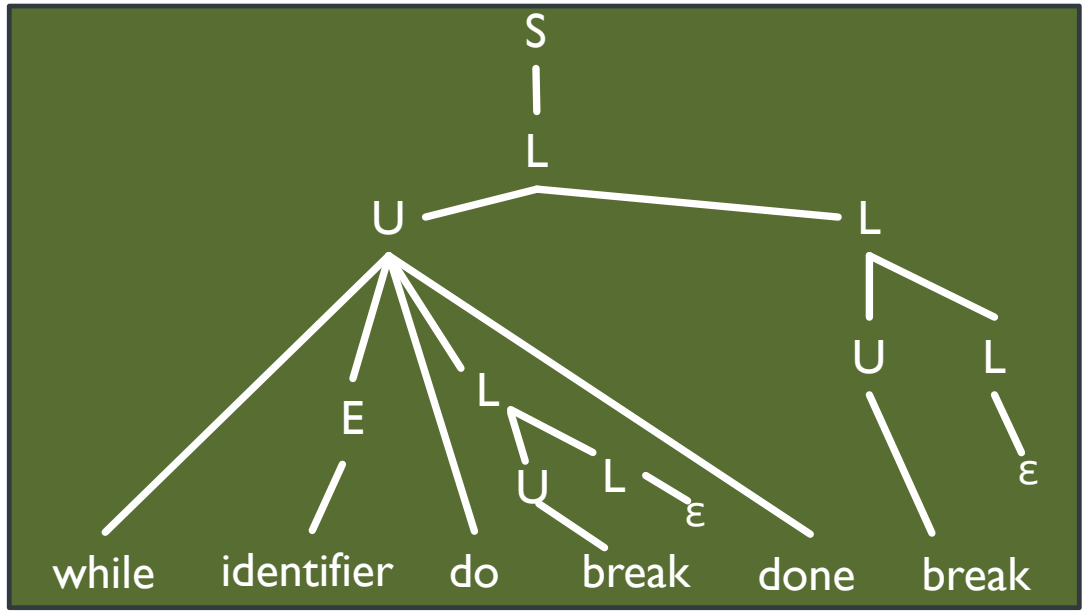
$L \rightarrow \varepsilon$
$L \rightarrow UL$

$U \rightarrow$ break

$U \rightarrow$ while E do L done

$E \rightarrow$ identifier

Szintaxisfa



Forrásszöveg

ATTRIBÚTUMNYELVTANOK: MÁSIK PÉLDA

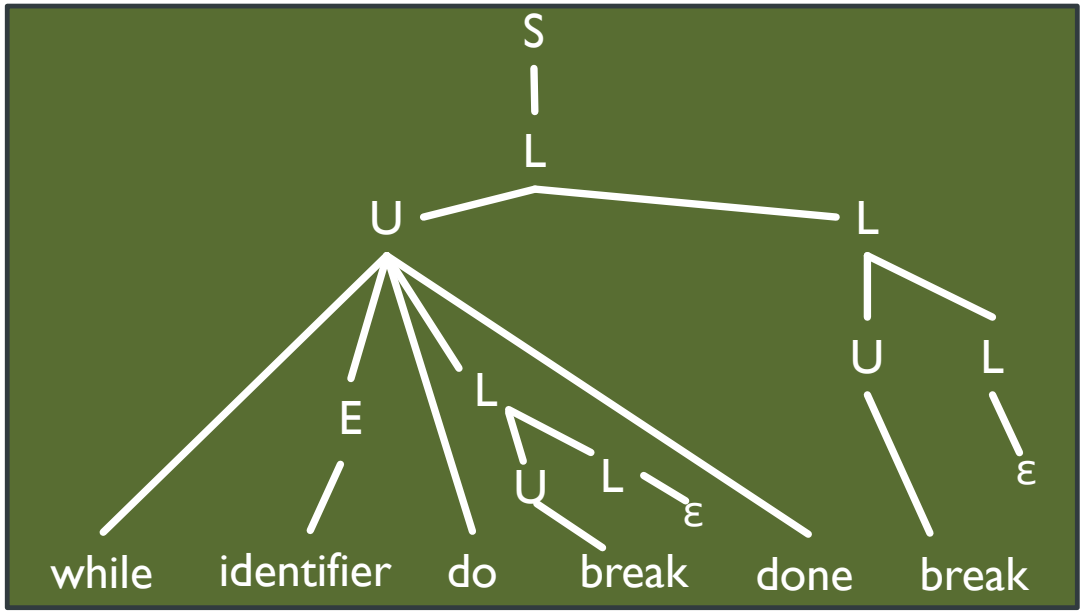
$S \rightarrow L$

$L \rightarrow \varepsilon$
$L \rightarrow UL$

$U \rightarrow$ break

$U \rightarrow$ while E do L done

$E \rightarrow$ identifier



while	b	do	break	done	break

Attribútumok:

- *L.in_loop*: Az utasításlista ciklus része-e (bool)
- *U.in_loop*: Az utasítás ciklus része-e (bool)

ATTRIBÚTUMNYELVTANOK: MÁSIK PÉLDA

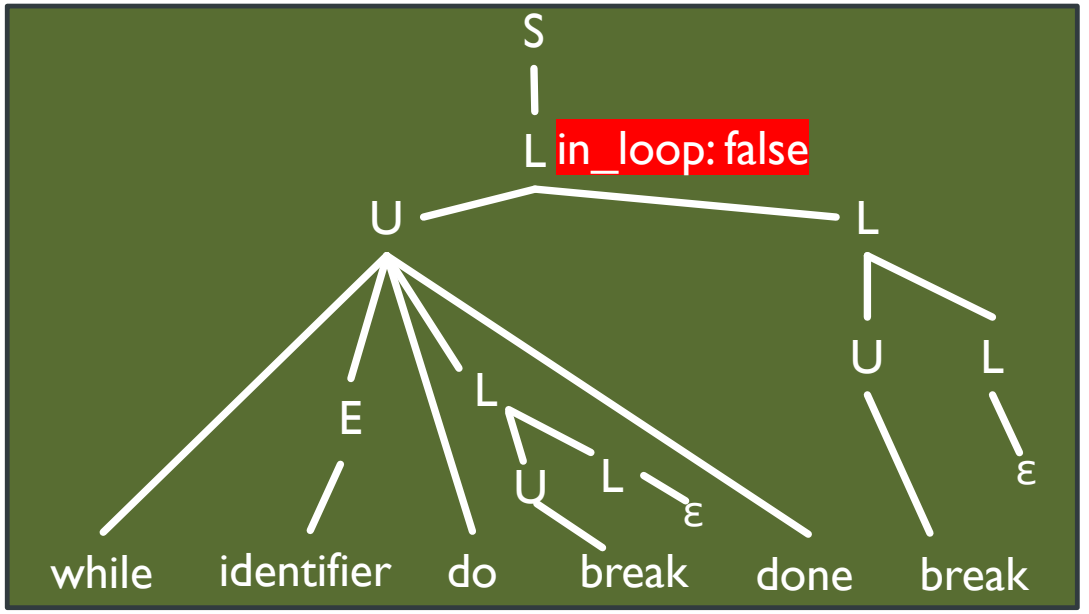
S → L {
  L.in_loop = false;
}
L → ε
L → UL

U → break

U → while E do L done

E → identifier

A legfelső szintű utasításlista nincs ciklus belsejében.



while	b	do	break	done	break

Attribútumok:

- *L.in_loop*: Az utasításlista ciklus része-e (bool)
- *U.in_loop*: Az utasítás ciklus része-e (bool)

ATTRIBÚTUMNYELVTANOK: MÁSIK PÉLDA

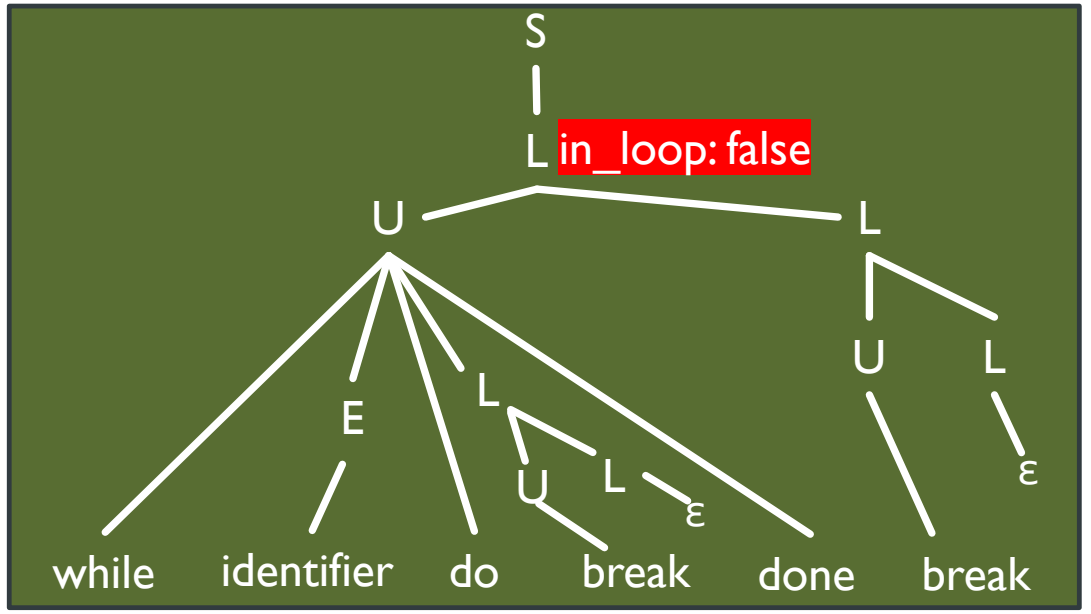
$S \rightarrow L$ {
  L.in_loop = false;
}
$L \rightarrow \varepsilon$
$L_1 \rightarrow UL_2$

Sorszámozás...

$U \rightarrow$ break

$U \rightarrow$ while E do L done

$E \rightarrow$ identifier



while	b	do	break	done	break

Attribútumok:

- *L.in_loop*: Az utasításlista ciklus része-e (bool)
- *U.in_loop*: Az utasítás ciklus része-e (bool)

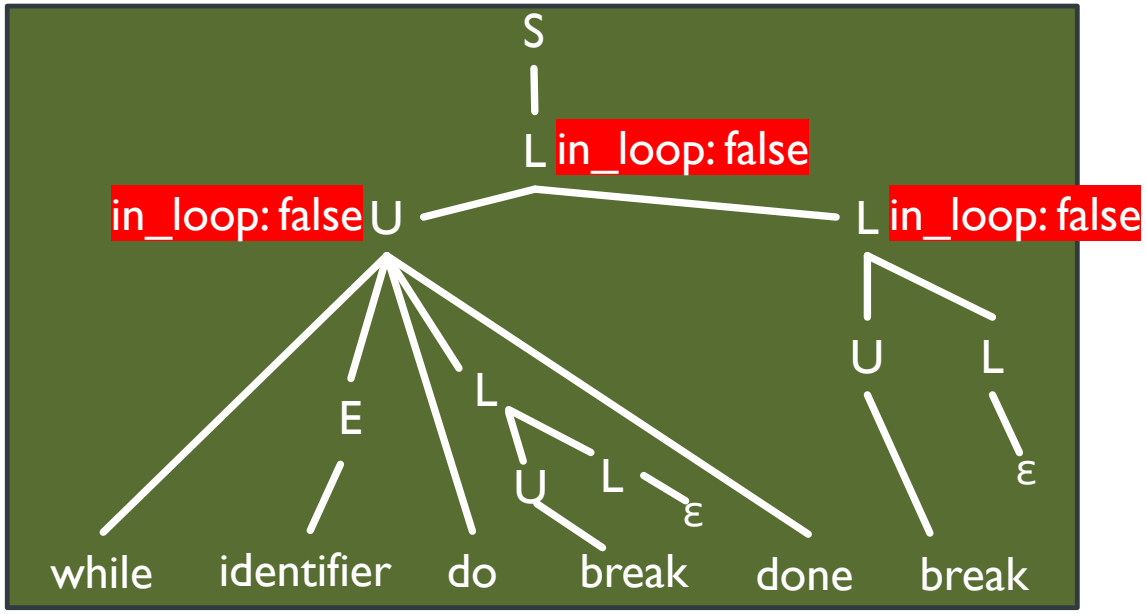
ATTRIBÚTUMNYELVTANOK: MÁSIK PÉLDA

$S \rightarrow L$ {
  L.in_loop = false;
}
$L \rightarrow \varepsilon$
$L_1 \rightarrow UL_2$ {
  U.in_loop = L_1.in_loop;
  L_2.in_loop = L_1.in_loop;
}
$U \rightarrow$ break

$U \rightarrow$ while E do L done

$E \rightarrow$ identifier

Az utasításlista tulajdonsága öröklődik a lista elemeire.



while	b	do	break	done	break

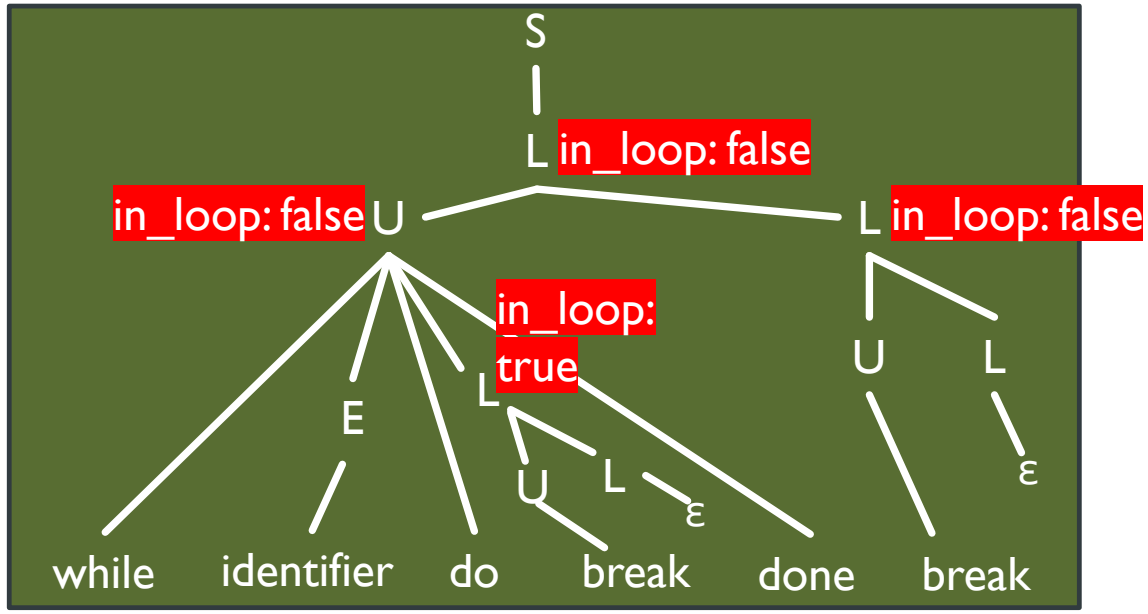
Attribútumok:
- *L.in_loop*: Az utasításlista ciklus része-e (bool)
- *U.in_loop*: Az utasítás ciklus része-e (bool)

ATTRIBÚTUMNYELVTANOK: MÁSIK PÉLDA

$S \rightarrow L$ {
  L.in_loop = false;
}
$L \rightarrow \varepsilon$
$L_1 \rightarrow UL_2$ {
  U.in_loop = L_1.in_loop;
  L_2.in_loop = L_1.in_loop;
}
$U \rightarrow$ break

$U \rightarrow$ while E do L done {
  L.in_loop = true;
}
$E \rightarrow$ identifier

A ciklus magja ciklus belsejében van.



while	b	do	break	done	break

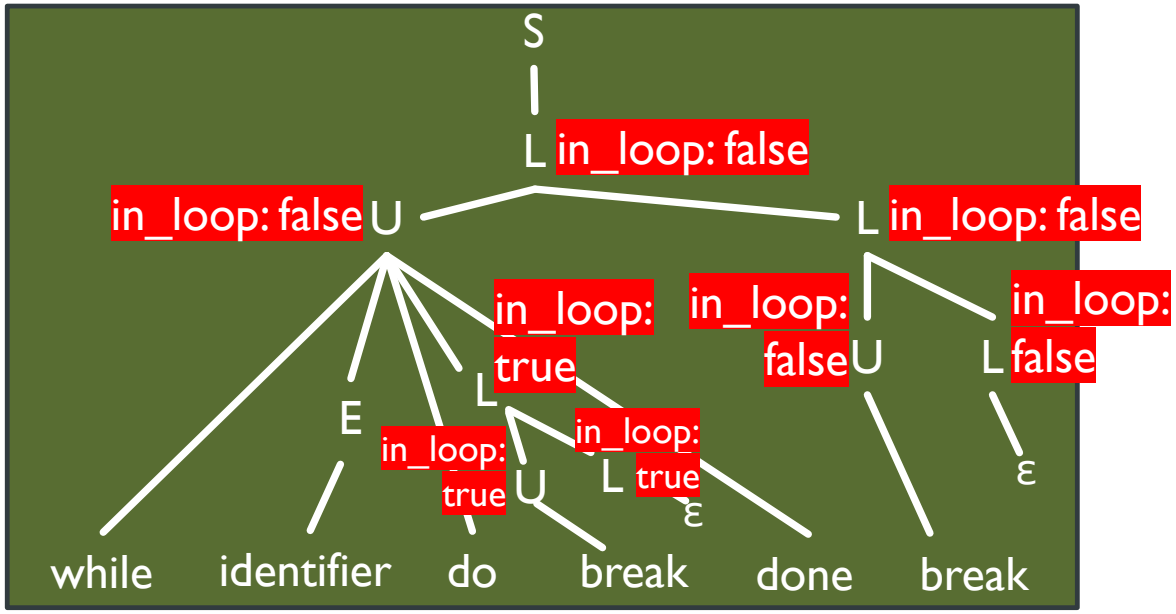
Attribútumok:
- *L.in_loop*: Az utasításlista ciklus része-e (bool)
- *U.in_loop*: Az utasítás ciklus része-e (bool)

ATTRIBÚTUMNYELVTANOK: MÁSIK PÉLDA

$S \rightarrow L$ {
  L.in_loop = false;
}
$L \rightarrow \varepsilon$
$L_1 \rightarrow UL_2$ {
  U.in_loop = L_1.in_loop;
  L_2.in_loop = L_1.in_loop;
}
$U \rightarrow$ break

$U \rightarrow$ while E do L done {
  L.in_loop = true;
}
$E \rightarrow$ identifier

Az utasításlista tulajdonsága öröklődik a lista elemeire.



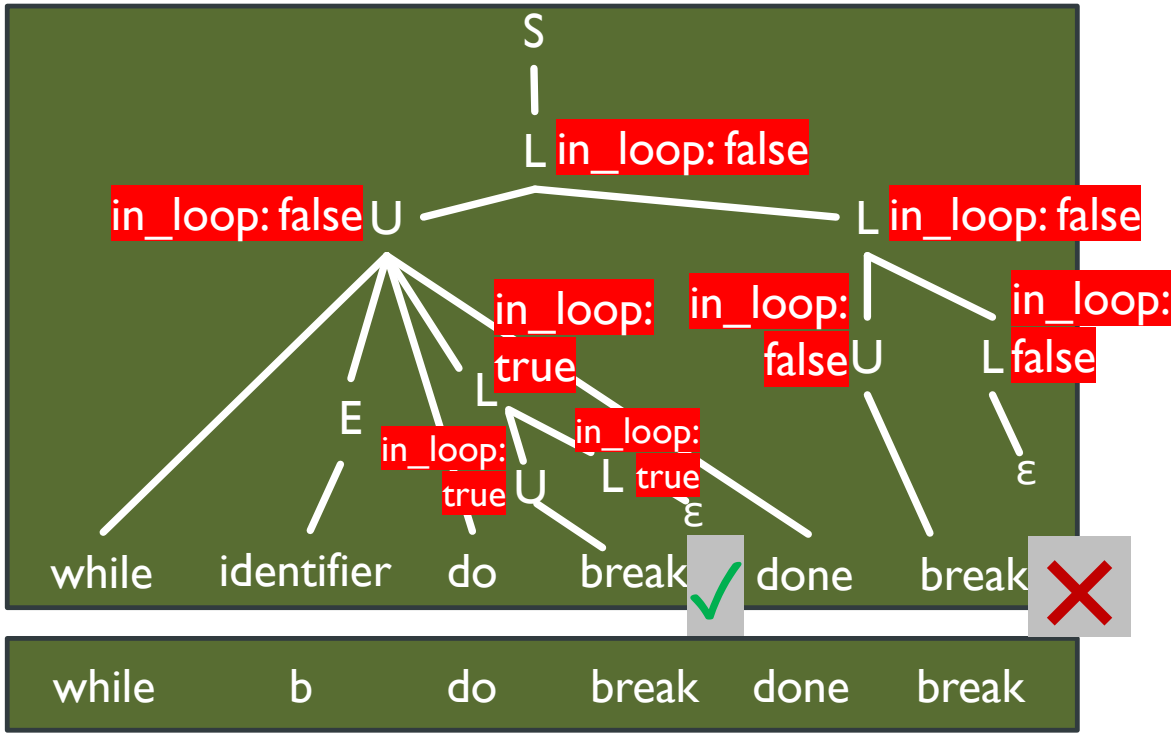
while b do break done break

Attribútumok:
- *L.in_loop*: Az utasításlista ciklus része-e (bool)
- *U.in_loop*: Az utasítás ciklus része-e (bool)

ATTRIBÚTUMNYELVTANOK: MÁSIK PÉLDA

$S \rightarrow L$ {
  L.in_loop = false;
}
$L \rightarrow \varepsilon$
$L_1 \rightarrow UL_2$ {
  U.in_loop = L_1.in_loop;
  L_2.in_loop = L_1.in_loop;
}
$U \rightarrow$ break {
  if not U.in_loop then
    error(...);
}
$U \rightarrow$ while E do L done {
  L.in_loop = true;
}
$E \rightarrow$ identifier

Ellenőrzés:
A ‘break’ utasítás ciklus belsejében van?

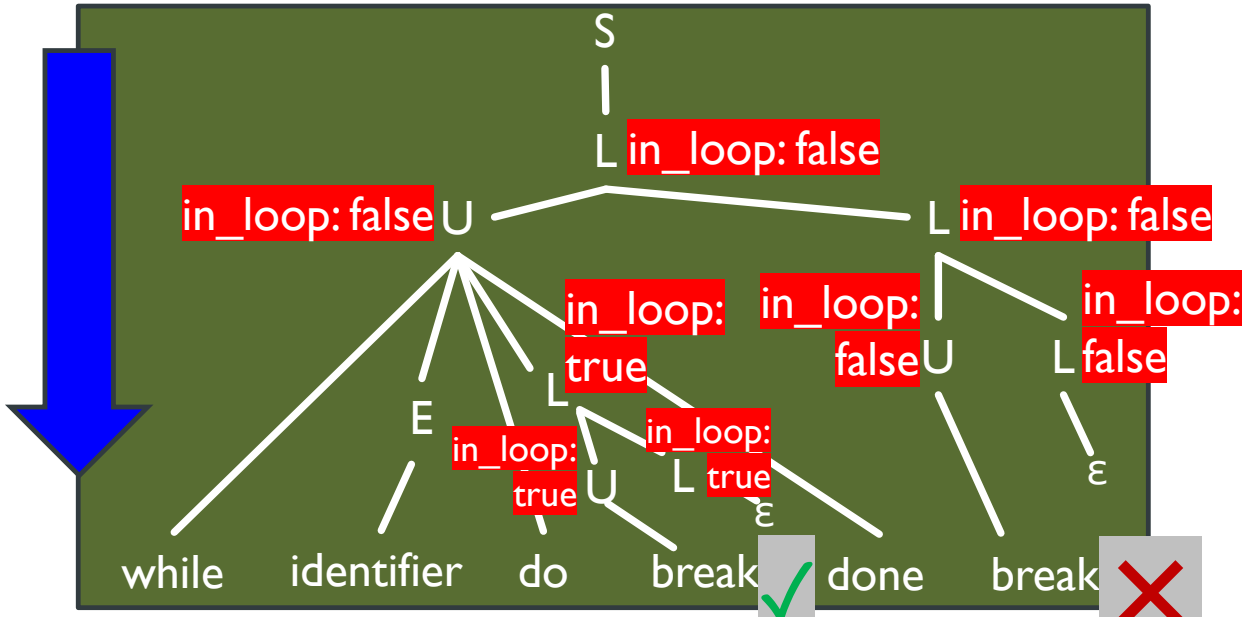


Attribútumok:
- *L.in_loop*: Az utasításlista ciklus része-e (bool)
- *U.in_loop*: Az utasítás ciklus része-e (bool)

ÖRÖKÖLT ATTRIBÚTUMOK

Örökölt attribútum:
- A szabály jobb oldalán áll, amikor kiszámítjuk.
- Felülről lefelé közvetít információt a szintaxisfában.

```
S → L {
  L.in_loop = false;
}
L → ε
L1 → UL2 {
  U.in_loop = L1.in_loop;
  L2.in_loop = L1.in_loop;
}
U → break {
  if not U.in_loop then
    error(...);
}
U → while E do L done {
  L.in_loop = true;
}
E → identifier
```



Attribútumok:
- *L.in_loop*: Az utasításlista ciklus része-e (bool) Örökölt
- *U.in_loop*: Az utasítás ciklus része-e (bool) Örökölt

ATTRIBÚTUMNYELVTANOK

- Az akciókban csak az adott szabályban szereplő szimbólumok attribútumai olvashatók és írhatók.
- Minden szintaxisfában minden attribútumértéket csak egy akció határozhat meg.

- Az attribútumok fajtái: kitüntetett szintetizált, szintetizált, örökölt
- Az attribútumnyelvtanok a szemantikus elemzésen túl kódgenerálásra is használhatók.
  - A generált tárgykód is lehet attribútum

A ‘BISON’ SZINTAKTIKUSELEMZŐ-GENERÁTOR

- Lásd a gyakorlatokon...
- A Bisonnak attribútumnyelvtant adhatunk meg
- Kitüntetett szintetizált és szintetizált attribútumokat támogat, örökölteket nem.
  - Ezt S-attribútumnyelvtannak nevezzük.
  - Jól illeszkedik az LR (alulról felfelé) elemzésekhez.