# Formális nyelvek és fordítóprogramok

## 8. Fordítóprogramok. A lexikális és szintaktikai elemzés

Bodó Zalán

Babeș–Bolyai Tudományegyetem
Matematika és Informatika Kar
Magyar Matematika és Informatika Intézet



# Fordítóprogramok

- magasszintű programozási nyelv ⇒ **fordítóprogram** ⇒ alacsonyszintű programozási nyelv
  - magasszintű nyelvek: C, C++, Pascal, ...
  - alacsonyszintű nyelvek: assembly, gépi kód

```
#include <stdio.h>

int main() {
for (int i=10; i>0; i--)
printf("%d\n", i);
return 0;
}
```

```
segment .data
;;;;;;;;;;;;;
dformat db "%d", 13, 10, 0
i dd 0
segment .text
;;;;;;;;;;;;;
global _main
extern _printf
_main:
enter 0,0
mov dword [i], 10
mov ecx, [i]
loop_start:
pusha
pushf
push dword [i]
push dformat
call _printf
add esp, 8
popf
popa
dec dword [i]
cmp dword [i], 0
jnz loop_start

leave
ret
```

- **forrásprogram, forrásnyelvű program**: a magasszintű nyelven írt program
- **tárgyprogram, tárgynyelvű program**: az alacsonyszintű nyelvre (assemblyre vagy gépi kódra) lefordított, futtatható program
- **fordítási idő**: fordítás időtartama
- **futtatási idő**: a lefordított tárgyprogram végrehajtási ideje
- **egymenetes/többmenetes fordítási folyamat**: Legyen $P$ a forrásnyelvű, $Q$ a tárgynyelvű program, $T$ a fordítás transzformációja; ekkor a fordítási folyamat a

$$Q = T(P)$$

képlettel írható le. Ha $T = T_n \circ T_{n-1} \circ \ldots \circ T_1$, akkor

$$\begin{aligned} P_{n-1} &= T_n(P) \\ P_{n-2} &= T_{n-1}(P_{n-1}) \\ &\ldots \\ P_1 &= T_2(P_2) \\ Q &= T_1(P_1) \end{aligned}$$

## Egymenetes (Pascal)

```
var x:Integer;
procedure inc;
begin
  x:=x+1;
end;
{var x:Integer;}
BEGIN
x:=0;
inc;
END.
```

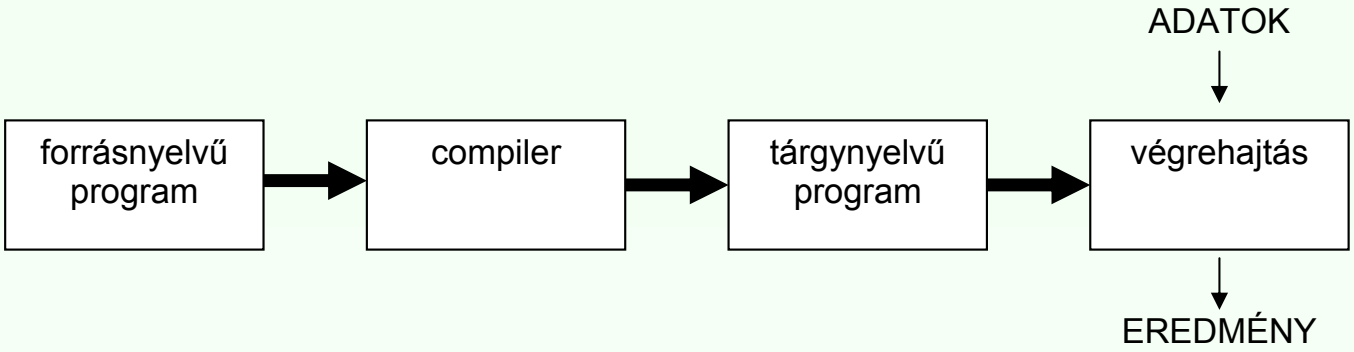
## Többmenetes (Java)

```
public class Example {
  public static void main(String [] args) {
    assert(x==0);
    x++;
    assert(x==1);
  }
  static int x=0;
}
```

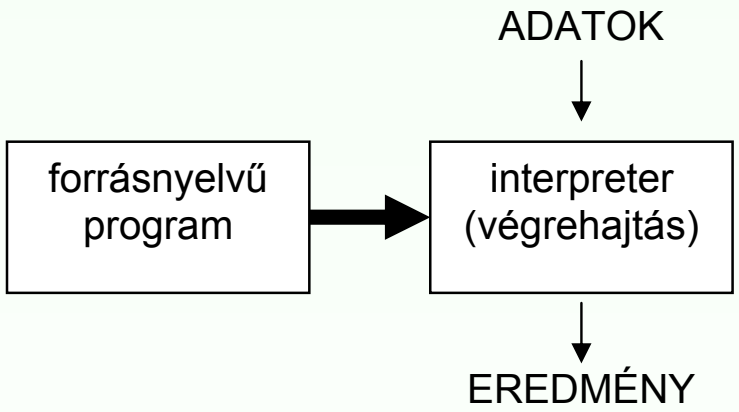
- az előbbi **többmenetes/$n$-menetes** esetben $P_i, i = 1, 2, \ldots, n-1$ a **program közbülső programformája**
- kódoptimalizálás $\Rightarrow$ több menet
- fordítóprogram típusai:
  - **compiler** – ez az amivel foglalkozni fogunk(!)
  - **interpreter** – olyan „gép” (program) készítése, amelyik a magasszintű nyelvet ismeri fel = a gép gépi kódja a magasszintű nyelv
  - **formulavezérlésű számítógép** – ha az interpretálást hardver szinten valósítjuk meg
- interpreter esetén a fordítási és futási idő egybeesik

Compilerek: C, C++, Pascal, ...
Interpreterek: Java (interpreter = Java Virtual Machine; + JIT), Perl (interpreter = perl), PHP (interpreter = PHP/FI [Personal Home Page/Forms Interpreter], Zend Engine, Zend Engine 2 és 3), ...

- első compilerek: 50-es évek eleje
- elsők között volt a FORTRAN compiler is (1957)




ábra: Compiler: fordítási folyamat



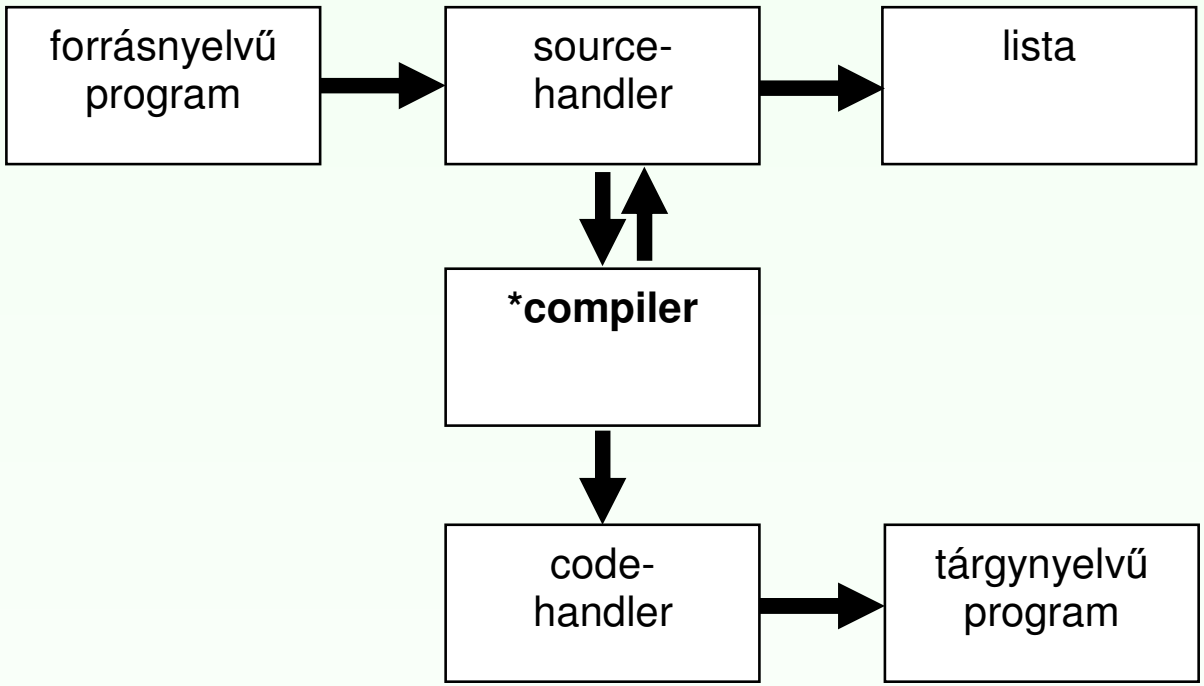

ábra: Interpreter: interpretálási folyamat

## A fordítóprogram szerkezete

`compiler(forrásprogram)(tárgyprogram, lista)`

- `source-handler(forrásprogram, hibák)(karaktersorozat, lista)`
  - `input-handler(forrásprogram)(karaktersorozat)`
    a forrásprogramot alakítja át, és küldi a compilernek (általában egy pufferben helyezi el ezeket); levágja az újsor karaktereket
  - `output-handler(forrásprogram, hibák)(lista)`
    listát készít a forrásprogramból és a hibákból, melyet a compiler készít; a listát, amely a hibák (pontos) helyét tartalmazza a háttértárolón helyezi el.
    **Fontos**: a fordítóprogramok szinte mindig hibás programot fordítanak, hibátlan programokkal csak elhanyagolhatóan kevés esetben találkoznak, ezért nagyon fontos a pontos hibajelzés
- `code-handler(tárgykód)(tárgyprogram)`
  a compiler által készített tárgykód elhelyezése a háttértárolón

- `*compiler(karaktersorozat)(tárgykód, hibák)`
  (ténylegesen csak a fordítóprogram feladatával foglalkozik)




ábra: A compiler felépítése

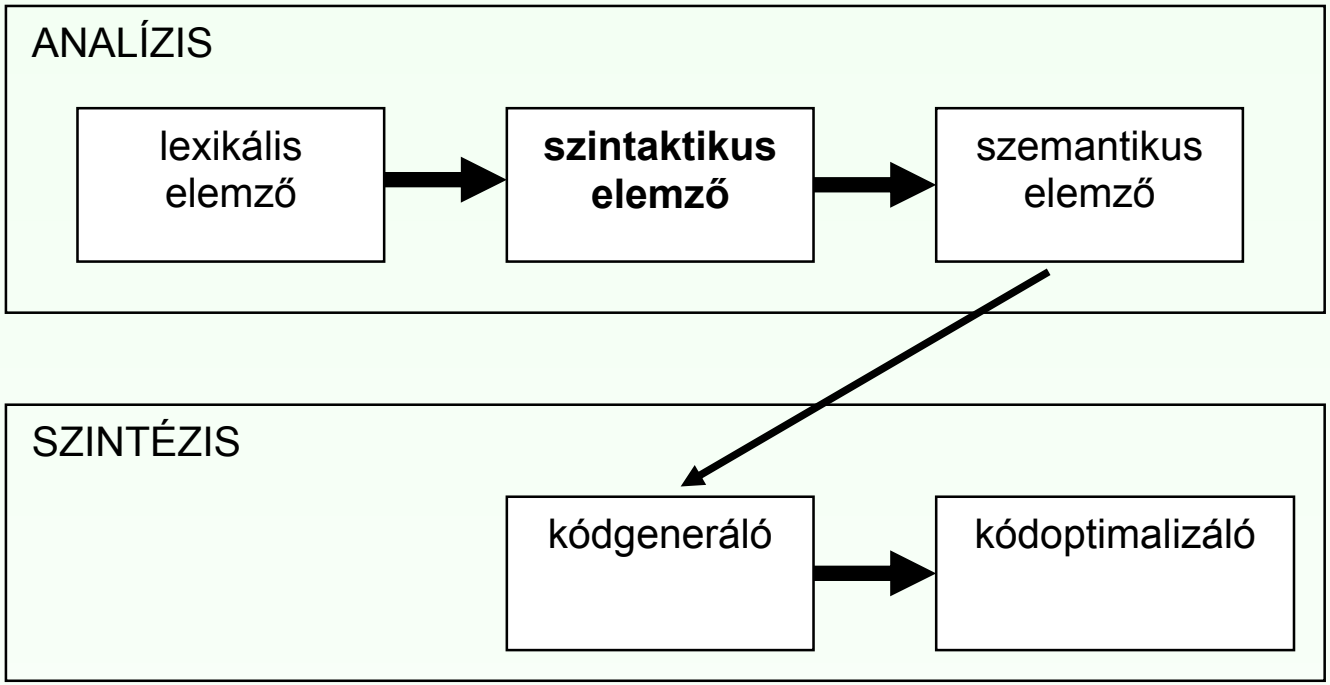
## A *compiler

- **analízis**: a forrásprogram karaktersorozatát részekre bontja és vizsgálja
  - **lexikális elemző**: `lexikális elemző(karaktersorozat)(szimbólumsorozat, lexikális hibák)`
    - szimbólumtábla: általában a szimbólumra a hozzá rendelt kóddal hivatkozunk
    - kiszűri: szóköz vagy más fehér karaktereket, kommentek
  - **szintaktikai elemző**: `szintaktikai elemző(szimbólumsorozat)(szintaktikailag elemzett program, szintaktikai hibák)`
    - a program struktúrájának felismerése: az egyes szimbólumok a megfelelő helyen vannak-e, megfelelnek-e a programozási nyelv szabályainak, nem hiányzik-e valahonnan valamilyen szimbólum
    - kimenet: szintaktikailag elemzett program (pl. szintaxisfa)

- **(analízis)**:
  - **szemantikai elemző**:
    `szemantikai elemző(szintaktikailag elemzett program)(elemzett program, szemantikai hibák)`
    - szemantikai tulajdonságok vizsgálata
    - pl.: $\langle azonosito \rangle + \langle konstans \rangle$ vizsgálatakor megnézi, hogy az $\langle azonosito \rangle$ szimbólum deklarálva van-e, a $\langle konstans \rangle$-nak van-e értéke, és típusuk azonos-e

- **szintézis**: az egyes részeknek megfelelő tárgykódokból felépíti a program teljes tárgykódját
  - **kódgeneráló**:
    `kódgenerátor(elemzett program)(tárgykód)`
    - assembly vagy gépi kód előállítása
  - **kódoptimalizáló**:
    `kódoptimalizáló(tárgykód)(tárgykód)`
    - cél: hatékonyabb kód elkészítése, mint amilyet egy jó assembly programozó írni tud

- **többmenetes fordítóprogram**: bizonyos lépéseket több menetben tud csak elvégezni
- a menetek számát a köv. tényezők befolyásolhatják:
  - a compiler rendelkezésére álló memória
  - a compiler mérete és sebessége
  - a tárgyprogram mérete és sebessége
  - a hibajavítási lehetőségek
  - hibafelismerési és hibajavítási stratégiák
  - (a compiler megírására rendelkezésre álló idő és szellemi kapacitás)
- az egyszerű fordítóprogramok egymenetesek
- segédeszközök:
  - lexikális elemző: `lex` vagy `flex`
  - szintaktikai elemző: `yacc` vagy `bison` (a szemantikai elemző és kódgenerátor is beépíthető)




ábra: Az analízis és szintézis fázisai

# Lexikális elemzés

- meghatározza a lexikális elemeket/egységeket
- kiszűri az információt nem hordozó részeket (fehér karakterek, újsor, kommentek)
- továbbadja a lexikális elemeket a szintaktikai elemzőnek
- a feladat megoldása: **reguláris grammatikával** (Chomsky 3-as típusú grammatikával), **reguláris kifejezésekkel**, **véges automatákkal**
- *általában*: a szimbolikus egységeket reguláris kifejezésekkel adjuk meg
- építhet szimbólumtáblát, ahol a szimbólumok és a hozzájuk tartozó kódok szerepelnek

# 1. Példa

Tekintsük a következő kódokat:

| *azonosító* | 01 |
|---|---|
| `if` | 25 |
| `else` | 26 |
| `(` | 43 |
| `)` | 44 |
| `<` | 29 |
| `++` | 98 |
| `;` | 45 |

Ekkor a lexikális elemző a

```
if (_ == __) _++; else __++;
```

kódból a

```
25 43 01 29 01 44 01 98 45 26 01 98 45
```

sorozatot készíti, amely már kevésbé „olvasható”.

## Kiterjesztett reguláris kifejezések (`flex`)

| | |
|---|---|
| `x` | `x` karakter |
| `.` | minden karaktek, kivéve az újsort |
| `[xyz]` | karakterosztály, az `x`, `y` vagy `z` karakterre illeszkedik |
| `[abj-oZ]` | karakterosztály, `a` vagy `b` vagy `j`-től `o`-g vagy `Z` |
| `[^A-Z]` | karakterosztály negáltja (komplementere) |
| `r*` | zéró vagy több `r` |
| `r+` | egy vagy több `r` |
| `r?` | opcionalitás: zéró vagy egy db. `r` |
| `r{2,5}` | kettő, három, négy vagy öt `r` |
| `r{2,}` | legalább két `r` |
| `r{4}` | pontosan négy `r` |
| `{nev}` | a `nev` kiterjesztése |
| `(r)` | zárójelezés a precedencia/prioritás megváltoztatása érdekében |
| `rs` | konkatenáció |

| | |
|---|---|
| `r\|s` | vagy |
| `r/s` | `r` ha követi őt egy `s`; az `s`-et beszámítja az illeszkedés hosszába, de csak az `r`-re illeszkedő sztringet téríti vissza (előreolvasási szimbólum, követő kontextus, *trailing context*) |
| `^r` | `r` a sor elején |
| `r$` | `r` a sor végén |
| `<s>r` | `r` ha `s` startfeltételben (*start condition*) vagyunk; `s` lehet `s1,s2,s3` alakú is |
| `<*>r` | bármilyen startfeltételben |
| `<<EOF>>` | fájlvége szimbólum |

## 2. Példa

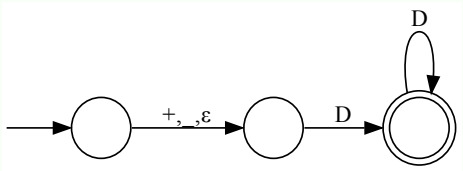
Írjuk le reguláris kifejezésekkel a következő lexikális elemeket: egész szám, valós szám, egysoros komment, azonosító szimbólum, fehér karakterek. Jelöljük a következő reguláris kifejezéseket:

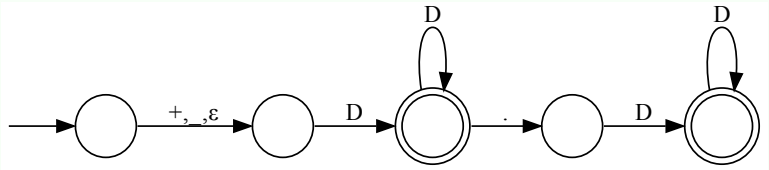
- `D = [0-9]`
- `A = [a-zA-Z]`

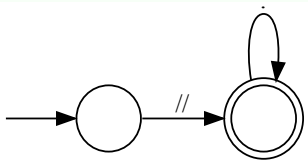
Ekkor:

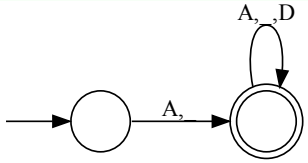
1. egész szám: `("+"|"-")?{D}+`
2. valós szám: `("+"|"-")?{D}+(.{D}+)`
3. egysoros komment: `//.*`
4. azonosító: `({A}|_)({A}|_|{D})*`
5. `[ \t\n]+`

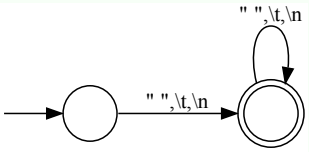
# Az ezekhez tartozó automaták sorban:



## Speciális problémák

- **kulcsszavak** és **standard szavak**
  - vannak programozási nyelvek, melyek különbséget tesznek kulcsszavak között
  - kulcsszónak nem változtathatjuk meg a jelentését, standard szavaknak megváltoztathatjuk
  - C/C++-ban pl. csak kulcsszavak vannak, de Pascal-ban vannak standard szavak
  - standard szavak használata nagyon rontja a program olvashatóságát, a programozónak sokkal nehezebb megtalálni a hibát anomália esetén
  - például: (ez nem fordítható le Pascal-ban, de létezhet olyan nyelv, amiben igen):
    `if if then else = then;`
  - tanács: „hanyagoljuk” standard szavak bevezetését

- **előreolvasás** (lásd az előreolvasási szimbólumot a kiterjesztett reguláris kifejezéseknél)
  - bizonyos esetekben előfordul, hogy a lexikális elemzőnek előre kell olvasni néhány karaktert, hogy meg tudja állapítani a helyes lexikális elemet
  - pl.1: meg akarjuk különböztetni a változókat a függvényektől, de mindkettő nevét ugyanúgy építhetjük fel, vagyis mindkettőt az `({A}|_)({A}|_|{D})*` reguláris kifejezéssel írjuk le; a következő kifejezés megoldja a problémát (`flex`): `({A}|_)({A}|_|{D})*/"("`

- **direktívák**
  - a fordítóprogram működésének vezérlésére szolgának
  - pl. egy `if` direktíva esetén (C/C++: `#if`, `#ifdef`, `ifndef`, `#elif`, `#else`, `#endif`) ki kell értékelnie a feltételt, és csak a megfelelő ágat bennhagynia a kódban
  - másik típusú ilyen probléma a makróhelyettesítés
  - a lexikális elemzőnek tehát szintaktikai és szemantikai ellenőrzéseket is kell végeznie
  - általában a lexikális elemző utáni előfeldolgozóval oldják meg ezeket

- **hibakezelés**
  - **lexikális hiba**: nem tud megfeleltetni egyetlen szimbólumot sem
  - hibaelfedő módszereket használunk; nem jó ha a fordítóprogram már ebben a fázisban „kiakad”
  - módszerek:
    1. figyelmen kívül hagyjuk a „rossz” karaktereket, és külön lexikális elemként továbbítjuk a szintaktikai elemzőnek a két oldalán álló szimbólumokat
    2. figyelmen kívül hagyjuk a „rossz” karaktereket, és összeolvasztjuk a két oldalán levő szimbólumokat
    3. továbbítjuk a „rossz” karaktereket a szintaktikai elemzőnek egy `undef` szimbólumként
  - más problémák: a sztringek végét jelző " hiánya, a többsoros kommentek zárószimbólumának hiánya

# Szintaktikai elemzés

- a szimbólumsorozat a nyelv egy mondata?
- **környezetfüggetlen grammatikákkal** oldjuk meg
- pontosabban **kiterjesztett környezetfüggetlen grammatikákkal**:

$$A \to \alpha, A \in N, \alpha \in (N \cup T)*$$

## Def. Mondatforma

Legyen $G = (N, T, P, S)$ egy grammatika. Ha $S \overset{*}{\Rightarrow} \alpha$, $\alpha \in (N \cup T)*$, akkor $\alpha$ **mondatforma**. Ha $S \overset{*}{\Rightarrow} x$, $x \in T*$, akkor $x$ az $L(G)$ egy **mondata**.

## Def. Részmondat

Legyen a $G = (N, T, P, S)$ grammatikának $\alpha = \alpha_1 \beta \alpha_2$ egy mondatformája. A $\beta$ **részmondat**, ha $\exists A \in N$ úgy, hogy $S \overset{*}{\Rightarrow} \alpha_1 A \alpha_2$ és $A \overset{+}{\Rightarrow} \beta$. A $\beta$ **egyszerű részmondata** $\alpha$-nak, ha a fentiekben $A \Rightarrow \beta$ teljesül, vagyis $(A \to \beta) \in P$.

## 3. Példa

Tekintsük a következő szabályok által definiált grammatikát:

$$\begin{aligned} E &\rightarrow T \mid E + T \\ T &\rightarrow F \mid T * F \\ F &\rightarrow i \mid (E) \end{aligned}$$

és az $E + T * i + T * F$ mondatformát.
Ennek $i + T$ és $i + T * F$ nem részmondata;
$E + T * i$, $T * F$ és $T * i$ részmondata;
$T * F$ egyszerű részmondata.

- csak **egyértelmű grammatikákkal** foglalkozunk
- nem egyértelmű = több szintaxisfa tartozik hozzá $\Rightarrow$ többféle elemzés tartozik hozzá $\Rightarrow$ többféle tárgykód (!)
- csak olyan grammatikákkal foglalkozunk, melyek a köv. feltételeket is teljesítik:
  1. a grammatika **ciklusmentes**: nem tartalmaz $A \overset{+}{\Rightarrow} A$ levezetéseket
  2. a grammatika **redukált**: nem tartalmaz felesleges nemterminális szimbólumokat (minden szimbólum szerepel legalább egy $S$-ből induló levezetésben)

## Def. Nyél

Egy mondatforma legbaloldalibb egyszerű részmondatát a mondatforma **nyelének** nevezzük.

## 4. Példa

A 3. Példában szereplő mondatforma nyele az $i$.

- balról jobbra haladó elemzésekkel foglalkozunk
- kétféle elemzés:
  1. **felülről-lefelé** haladó elemzés: az $S$ szimbólumból kiindulva próbáljuk meg felépíteni a szintaxisfát, azaz levezetni a programot
  2. **alulról-felfelé** haladó elemzés: a terminálisokból (a programból) kiindulva próbáljuk felépíteni a szintaxisfát, azaz eljutni az $S$ szimbólumhoz