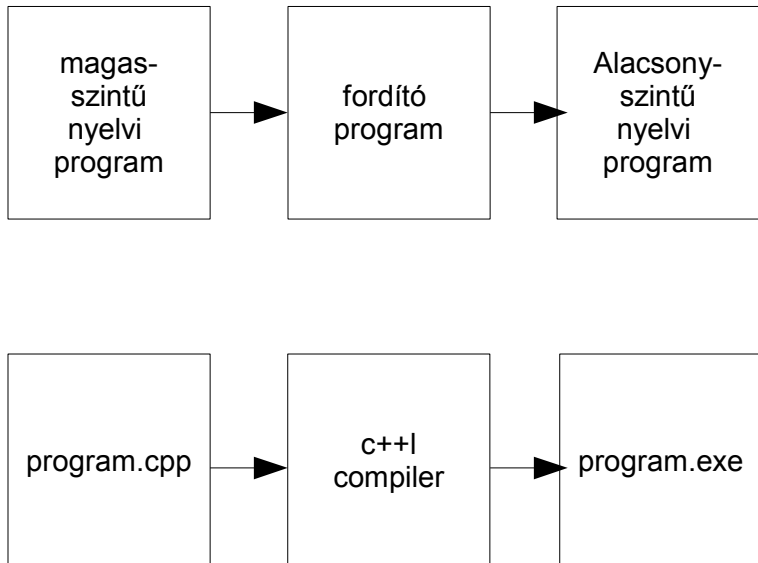


# Fordítóprogramok és formális nyelvek

Szerkesztette : Király Roland  
2007

## Fordítóprogram szerkezete

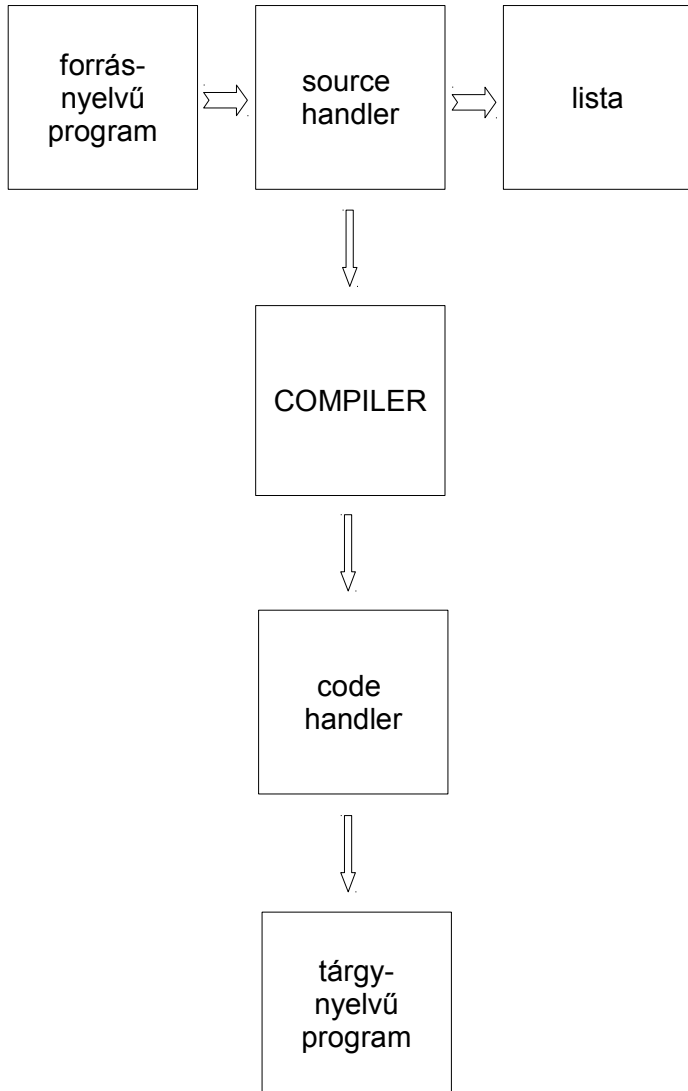


- A fordítóprogramok általánosan forrásnyelvi szövegből állítanak elő tárgykódot.
- A fordítóprogramok feladata, hogy nyelvek közti konverziót hajtsanak végre.
- A fordítóprogram a forrásprogram beolvasása után elvégzi a lexikális, szintaktikus és szemantikus elemzést, előállítja a szintaxis fát, generálja, majd optimalizálja a tárgykódot.

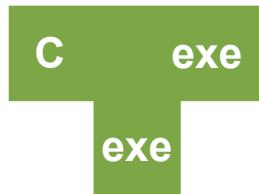
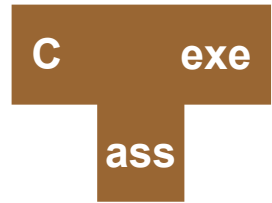
## Compiler és Interpreter

- A compiler magas szintű, forrásnyelvi szöveget, más néven forráskódot transzformál alacsonyszintű nyelvre, legtöbbször assembly, vagy gépi kódra.
  - **input-handler(forráskód) -> hibák, karaktersorozat**
  - **compiler(karaktersorozat) -> tárgykód**
  - **code-handler(tárgykód) -> tárgyprogram**
- Az interpreter hardveres, vagy virtuális gép, mely értelmezni képes a magasszintű nyelvet, vagyis melynek gépi kódja a magasszintű nyelv. Ez egy kétszintű gép, melynek az alsó szintje a hardver, a felső az értelmező és futtató rendszer programja.
- Fordítási idő: azt az időt mely alatt a compiler elvégzi a transzformációt, fordítási időnek nevezzük.
- Futási idő: azt az időt, mely alatt a lefordított programot végrehajtjuk, futási, vagy futtatási időnek nevezzük.
- Az interpreter esetében a fordítási és a futási idő egybe esik, amíg a compiler esetén a két időpont jól elkülöníthető.

## Fordító (compiler) felépítése

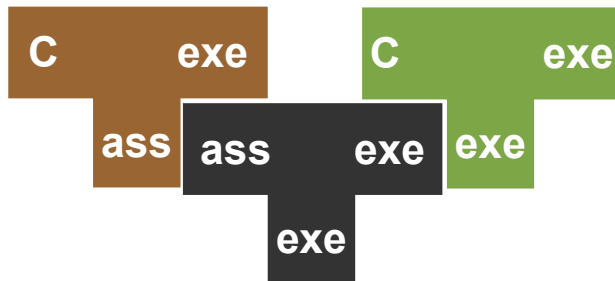


## T diagram



- A fordítóprogramokat három nyelvvel jellemezhetjük.
- Implementáció nyelve, **I** melyen a fordítóprogramot írták. Lehetőleg magasszintű nyelv.
- Forrásnyelv **S** amit a program lefordít.
- Tárgynyelv **T**, melyre a forrásnyelvből fordít .

## A C - exe fordítóprogram



- A T diagramok azon pontokon illeszthetők egymáshoz, ahol a diagramban azonos nyelvet találunk.
- Az első diagram egy assembly nyelven írt C-exe fordító.
- A második exe végrehajtható kódú ass – exe assembler.
- Ezekből előállítható egy végrehajtható C – exe fordítóprogram

## Input handler (source handler)

- Az input-handler a forrásnyelvi szöveget beolvassa, majd az újsor és a kocsivissza karaktereket levágja a sorok végéről.
- Sok esetben ez a program a lexikális elemző részeként van implementálva.
- Az input-handler nem végez szemantikai és szintaktikai ellenőrzést, ezt a feladatot a lexikális elemző végzi.
- Az input-handler kimenete a lexikális elemző bemenete.
- A szintaktikusan és szemantikusan ellenőrzött és helyes kódból a fordítóprogram tárgykódot állít elő. Ezt a feladatot az output-handler végzi.
- Ha az input és output handlert egy programban implementálják, akkor ezt a programot source-handlernek nevezzük.

## Input handler implementációja

```
class handler
{
    public string sourceHandler(string source)
    {
        string S="";
        System.IO.StreamReader Stream = new
        System.IO.StreamReader
        (
            System.IO.File.OpenRead(source)
        );

        while (Stream.Peek() > -1)
        {
            S += Stream.ReadLine();
        }
        return S;
    }
}
```



## Formális nyelvtanok

Nézzünk pár fontos alapfogalmat!

formális nyelvtan

: ahol a nemterminálisok, a terminálisok halmaza, a kezdőszimbólum, pedig az alakú szabályok halmaza.

környezet függetlenség

: a szabályok alakúak.

reguláris nyelvtanok

: minden szabály vagy vagy alakú.

derivációs fa

: a gyökérben a kezdőszimbólum van, a levelek a terminálisok, a csúcspontokban pedig nemterminálisok ülnek a levezetésnek megfelelően „összekötve”. Bemutatás egy példán keresztül:  $S \rightarrow AB$

$A \rightarrow aA$  |      input: aab

$B \rightarrow bB$  |

# Reguláris kifejezések

Jelöljön  $D$  egy tetszőleges számjegyet és  $L$  egy tetszőleges karaktert, a whitespace karaktereket a nevükkel helyettesítjük  $\{eol, eof, space\}$ .

$D \in \{0, 1, 2, \dots, 9\}$ ,  $L \in \{a, b, \dots, z, A, B, \dots, Z\}$ ,

akkor:

1) Egész számok:

$(+ | - | \epsilon)D^+$

2) Pozitív egész és valós számok:

$(D^+ (\epsilon | .)) | (D^* .D^+)$

3) Egyszerű azonosító szimbólum:

$L(L | D)^*$

4) Azonosító szimbólum:

$L((\_ | \epsilon)(D | L))^*$

5) `//` -el határolt komment:

$//( \text{Not}(eol))^* eol$

6) `##` -al határolt komment:

$##((\# | \epsilon) \text{Not}(\#))^* ##$

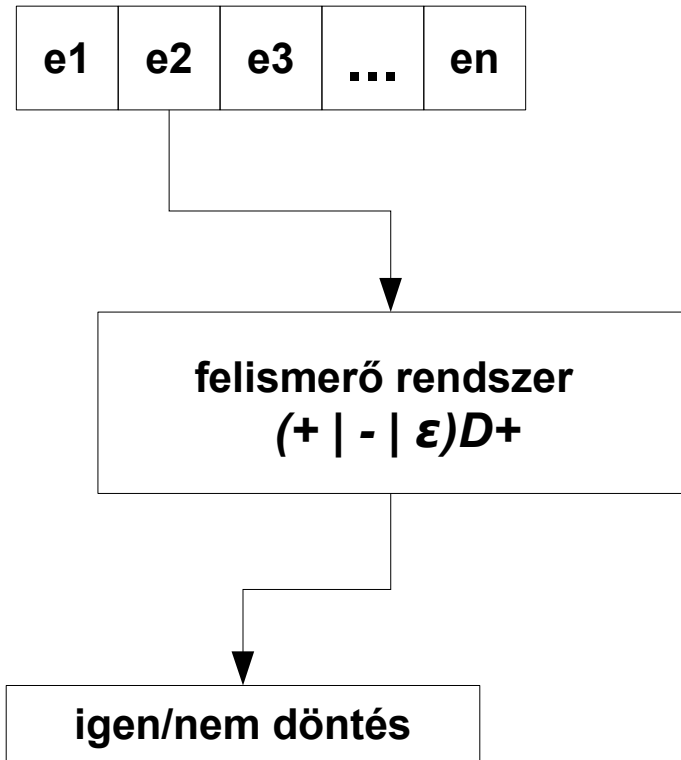
7) Karaktersztring:

$“( \text{Not}(\text{“}) | \text{“} )^* \text{“}$

8) Kitevős valós szám:

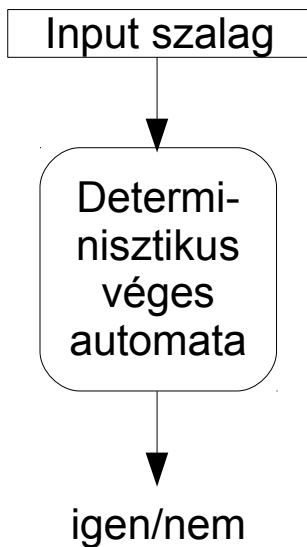
$(+ | - | \epsilon)D^+.D^+(E(+ | - | \epsilon)D^+ | \epsilon)$

## Reguláris kifejezések felismerése

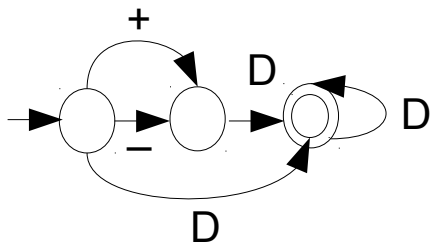


- A döntéshozó rendszer, vagyis az automata eldönti, hogy az input szalagon lévő sorozat a nyelvnek eleme, vagy sem.

## Determinisztikus-véges automaták

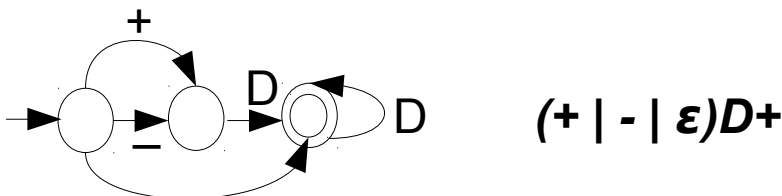


$(+ \mid - \mid \epsilon)D^+$



- Reguláris kifejezésekhez determinisztikus véges automata konstruálható.
- Az automata implementációja állapotai és az állapot átmenetek magasszintű programozási nyelveken jól implementálhatóak.

## Kifejezésekhez konstruált automaták I.



$$A = (Q, \Sigma, E, i, F)$$

véges automata, ahol

- Q** - a belső állapotok halmaza
- $\Sigma$**  - az input ABC
- E** - az átmenetek halmaza
- i** - a kezdőállapot (halmaz)
- F** - végállapot (halmaz)

$$\Sigma = \{+, -, D\}.$$

$$Q = \{q_0, q_1, q_2\},$$

$$E = (q_0, +, q_1), (q_0, -, q_1), (q_0, D, q_2), \\ (q_1, D, q_2), \\ (q_2, D, q_2),$$

$$F = \{q_2\},$$

$$i = \{q_0\}$$

## Kifejezésekhez konstruált automaták II.

$$A = (\mathbf{Q}, \Sigma, \mathbf{E}, \mathbf{I}, \mathbf{F})$$

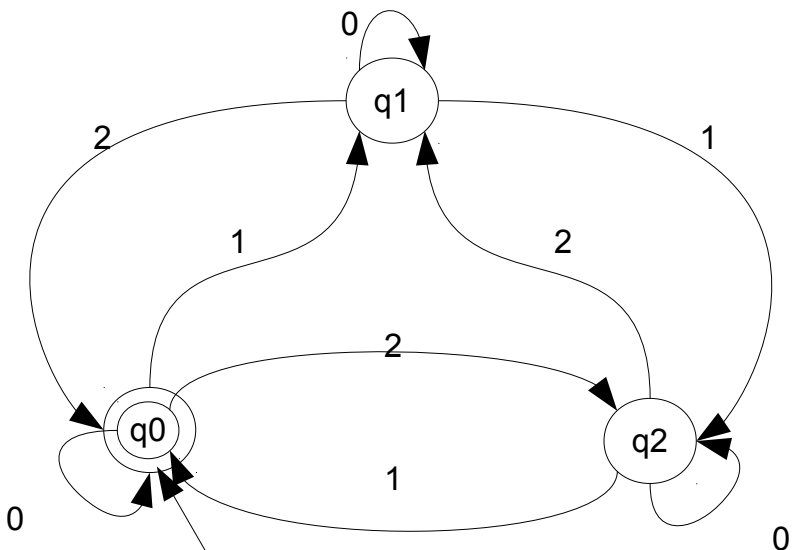
*A terminális jelek = {0, 1, 2}.*

*Az állapotok = {q0, q1, q2},*

*E = (q0, 0, q1), (q0, 1, q1), (q0, 2, q2),  
 (q1, 0, q1), (q1, 1, q2), (q1, 2, q0),  
 (q2, 0, q2), (q2, 1, q0), (q2, 2, q1),*

*Kezdő állapot = {q0},*

*Elfogadó végállapot = {q0}*



# Automata Implementációja

```
public string analyst(string grammar)
{
    string STATE="q0";
    string OK="OK";
    int i=0;
    while (i < grammar.Length &&
           STATE != "error")
    {
        STATE=transition(STATE,grammar[i]);
        ++i;
    }
    if (i<grammar.Length)
    {
        OK= "A hiba pozíciója"
        +i.ToString()+" "+grammar[i]
        +" nem eleme a nyelvnek";
    }
    return OK;
}
}
```

- Elemző program implementációjához érdemes magasszintű nyelveket használni.
- Az elemző programja determinisztikus véges automata, melynek állapotait switch ágaival, vagy állapotátmenet függvényekkel valósítjuk meg.

# Állapotátmenet függvény

```
string transition(string state, char char_)
{
    string nstate;
    switch (state+char_)
    {
        case "q00":nstate="q0";break;
        case "q01":nstate="q1";break;
        case "q02":nstate="q2";break;
        case "q10":nstate="q1";break;
        case "q11":nstate="q2";break;
        case "q12":nstate="q0";break;
        case "q20":nstate="q2";break;
        case "q21":nstate="q0";break;
        case "q22":nstate="q1";break;
        default:nstate="error";break;
    }
    return nstate;
}
```

- Az automata állapotait a switch ágaival valósítjuk meg. Minden egyes case ág az automata egy állapotátmenetét írja le.
- Az állapotátmenet függvény bemenő paramétere az aktuális állapot és az un. input szalag következő karaktere.



## Lexikális elemző

- A lexikális elemző inputja a source handler outputja, vagyis egy karaktersorozat, mely az újsor és a kocsivissza karakterek kivételével mindent tartalmaz a forráskódból.
  - A lexikális elemző reguláris, vagyis Chomsky 3-as nyelvet használ.
  - Feladata: felismerni az adott nyelv lexikális elemeit, majd helyettesíteni azokat a Szintaktikus elemző számára érthető jelekkel.
  - A lexikális elemző outputja a szintaktikus elemző inputja lesz és egy hibalista a lexikális hibákról.
- 
- Az egyes nyelvi elemeket leírjuk reguláris kifejezésekkel és megalkotjuk az ekvivalens determinisztikus – véges automatát.
  - Elkészítjük az automata implementációját.
  - (Az automata implementációja magasszintű nyelveken egyszerűen a switch, vagy case nyelvi elem felhasználásával kivitelezhető. Az állapotok a switch egyes ágai...)

# Lexikális elemző működése

If  $a > 0$  then  $x := a$  else  $x := 0$

- A fenti programszövegből a lexikális elemző az alábbi kimenetet készíti el, mely már teljesen olvashatatlan.

10	if
20	then
30	else
40	:=
50	>

10 001 50 002 20 003 40 001 30 003 40 0004

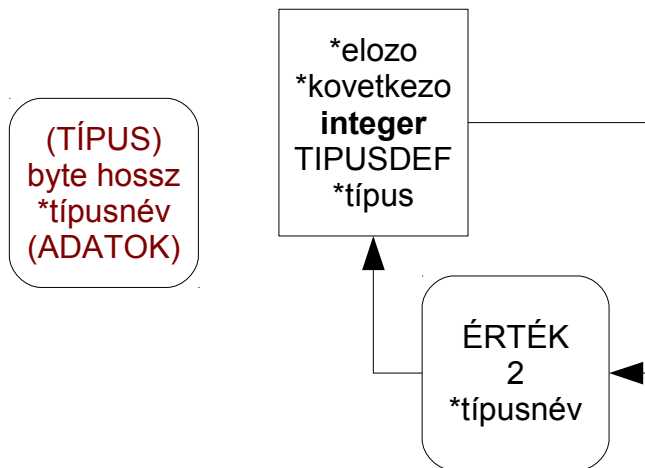
- A használt változókhoz, konstansokhoz, stb... az ún. **szimbólumtábla** bejegyzések tartoznak, melyben tárolható a változók azonosítójához tartozó érték, típus és egyéb attribútumok.
- A szimbólumtáblát legtöbbször láncolt listával valósítják meg.
- A szimbólumtábla bejegyzéseit a szemantikus elemző használja fel.
- A lexikális elemző felhasználható forrásnyelvi programok más nyelvekre való konvertálására is.

## Szimbólumtábla

- A szimbólumtábla olyan táblázat, mely tartalmazza a programszövegben előforduló szimbólumok nevét és jellemzőit, vagyis az attribútumokat.
- Ezek az információk a lexikális és szintaktikus elemző számára nem lényegesek, de a szemantikus elemzés és a kódgenerálás számára elengedhetetlenek.
- A szimbólum minden egyes előfordulása esetén a táblában keresést, vagy beszúrást kell végezni. Összesen ez a két művelete van.
- Szimbólumtábla egy sorának a tartalma:
  - Szimbólum neve
  - Attribútumok
    - definíciós adatok
    - típus
    - programbeli cím (tárgyprogram)
    - forrásnyelvi sorszám
    - hivatkozott sorszám
    - láncolási cím

## Szimbólumtábla

- **Definíciós adatok, típus:**
  - Mit azonosít a szimbólum, Pl.: változó, konstans, stb. Konstansnál az érték, típusnál a típusdescriptorra mutató pointer, eljárásnál annak paraméterei, azok száma.



- **Tárgyprogrambeli cím:**
  - ezt a címet kell a tárgyprogramba beépíteni a szimbólumra való hivatkozáskor.
- **Definíció sorszáma:**
  - explicit definíciónál a definíció sora, implicitnél az első előfordulás
- **Hivatkozott sorszám, láncolási cím**
  - a hivatkozási lista elkészítéséhez

# Szimbólumtábla implementációja

```
struct symrec
{
    char *name; /* szimbólum neve */
    struct symrec *next; /* mutató mező */
};

typedef struct symrec symrec;
symrec *sym_table = (symrec *)0;
symrec *put ();
symrec *get ();
```

- A fenti szimbólumtábla implementációja láncolt listával történik. Csak a szimbólum nevét tartalmazza, attribútumokat nem.
- A tábla két művelete:
  - beszúrás – put(), mely nemlétező szimbólum esetén beszúrja azt a táblába, majd visszatér a címével. Létező szimbólum esetén is a címével tér vissza.
  - kivét – get(), mely visszatér a szimbólum címével, vagy a 0 értékkel nemlétező szimbólum esetén.

# Szimbólumtábla műveletei

- put a szimbólumok elhelyezésére a listában

```
symrec *put ( char *sym_name )
{
    symrec *ptr;
    ptr = (symrec *) malloc (sizeof(symrec));
    ptr->name =
        (char *) malloc (strlen(sym_name)+1);
    strcpy (ptr->name,sym_name);
    ptr->next = (struct symrec *)sym_table;
    sym_table = ptr;
    return ptr;
}
```

- get a szimbólum ellenőrzéséhez a hivatkozások esetén

```
symrec * get ( char *sym_name )
{
    symrec *ptr;
    for (ptr = sym_table; ptr != (symrec *) 0;
        ptr = (symrec *)ptr->next)
        if (strcmp (ptr->name,sym_name) == 0)
            return ptr;
    return 0;
}
```

## Programnyelvek konvertálása

# Szintaxis

program :

változó **<deklaráció>** kezd **<parancsok>** vége;

deklaráció: típus **<azonosító\_lista>** azonosító;

azonosító\_lista:

azonosító | azonosító\_lista;

parancslista:

| parancslista **<parancs>**;

parancs: üres\_utasítás

| beolvas azonosító

| kiír kifejezés

| azonosító := kifejezés

| ha kifejezés akkor parancsok különben  
parancsok vége

| ciklus **<kifejezés>** elvégez parancsok vége;

kifejezés: szám | azonosító

| kifejezés < kifejezés

| kifejezés = kifejezés

| kifejezés > kifejezés

| kifejezés + kifejezés

| kifejezés – kifejezés

| kifejezés \* kifejezés

| kifejezés / kifejezés

| kifejezés ^ kifejezés | (kifejezés)

Az azonosító és a szám, leírható meta  
karakterekkel...



## BNF

program  
 ::= LET [ declarations ] IN command sequence  
 END

declarations ::= INTEGER [ id seq ] IDENTIFIER .  
 id seq ::= id seq... IDENTIFIER ,  
 command sequence ::= command... command  
 command ::= SKIP ;  
           | IDENTIFIER := expression ;  
           | IF exp THEN command sequence  
               ELSE command sequence FI  
           | WHILE exp  
               DO command sequence END ;  
           | READ IDENTIFIER ;  
           | WRITE expression ;

expression ::=  
           NUMBER | IDENTIFIER  
           | '(' expression ')'  
           | expression + expression  
           | expression - expression  
           | expression \* expression  
           | expression / expression  
           | expression ^ expression  
           | expression = expression  
           | expression < expression  
           | expression > expression

## Lex és Yacc

A **LEX** reguláris kifejezések formális leírásából hozza létre egy determinisztikus véges automata implementációját. Valójában lexikális elemző programot generál (C kódot állít elő).

Mintákat használ, hogy tokeneket készítsen az input stringből.

A token a string számszerű reprezentációja, mely megkönnyíti a feldolgozást.

A **YACC** C kódot generál a szintaktikus elemzőhöz. Nyelvtani szabályokat használ a lex tokenjeinek elemzéséhez és a szintaxisfa felépítéséhez.

A felépített szintaxisfa segítségével kódot generálhatunk, mely virtuális gépen, vagy a számítógép hardverén futtatható.

**lexer**  
lexikális elemző

**Parser**  
szintaktikus elemző

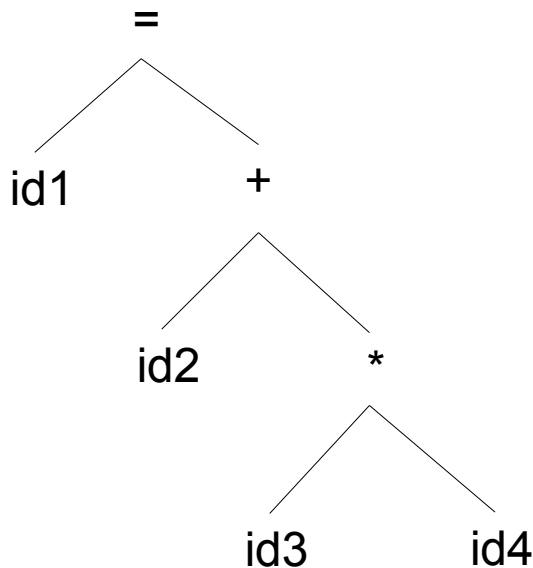
## Szintaxis fa

$a = b + c * d$

A szintaxis-fa a forrásszövegből, több lépésben, a nyelvtani szabályok felhasználásával építhető fel. A **LEX LALR(1)** nyelvtant használ és alulról felfelé elemez.

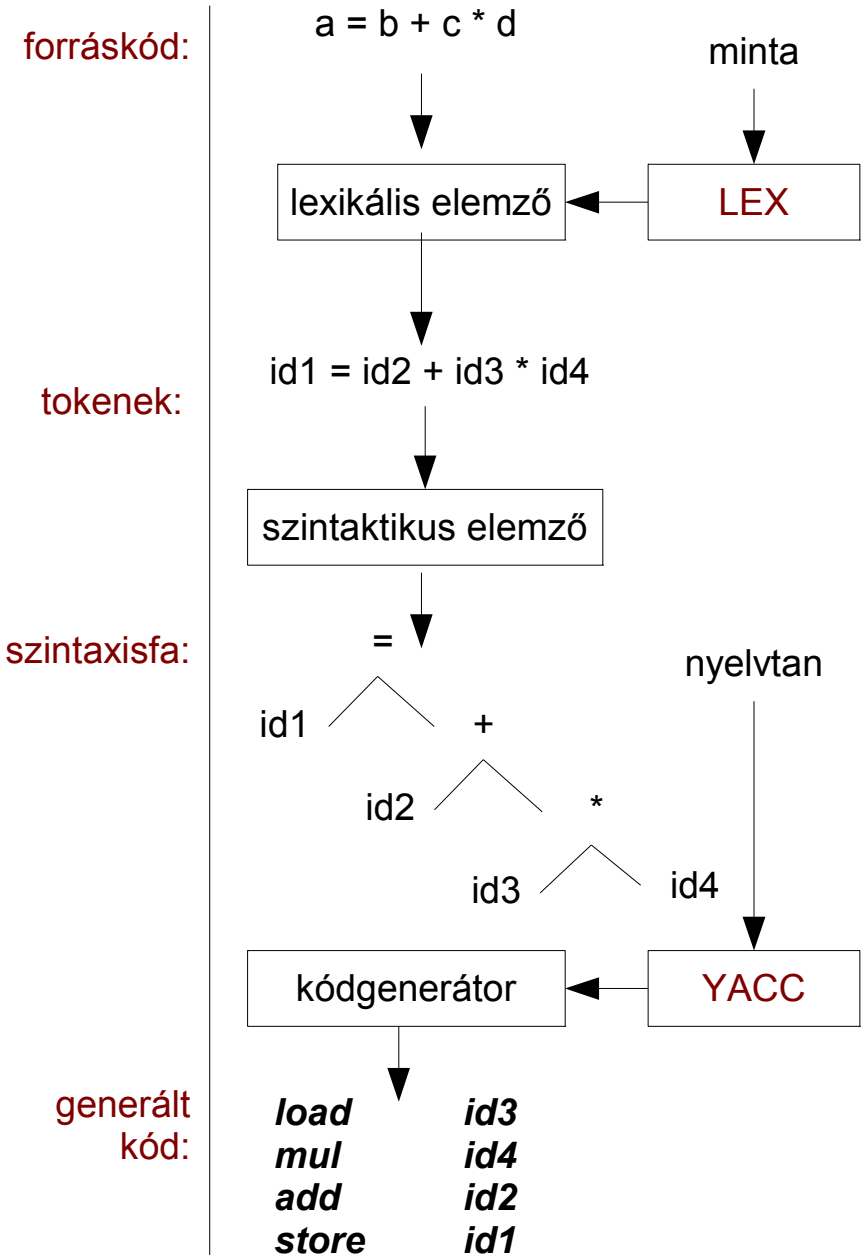
Szintaxis fa:

---

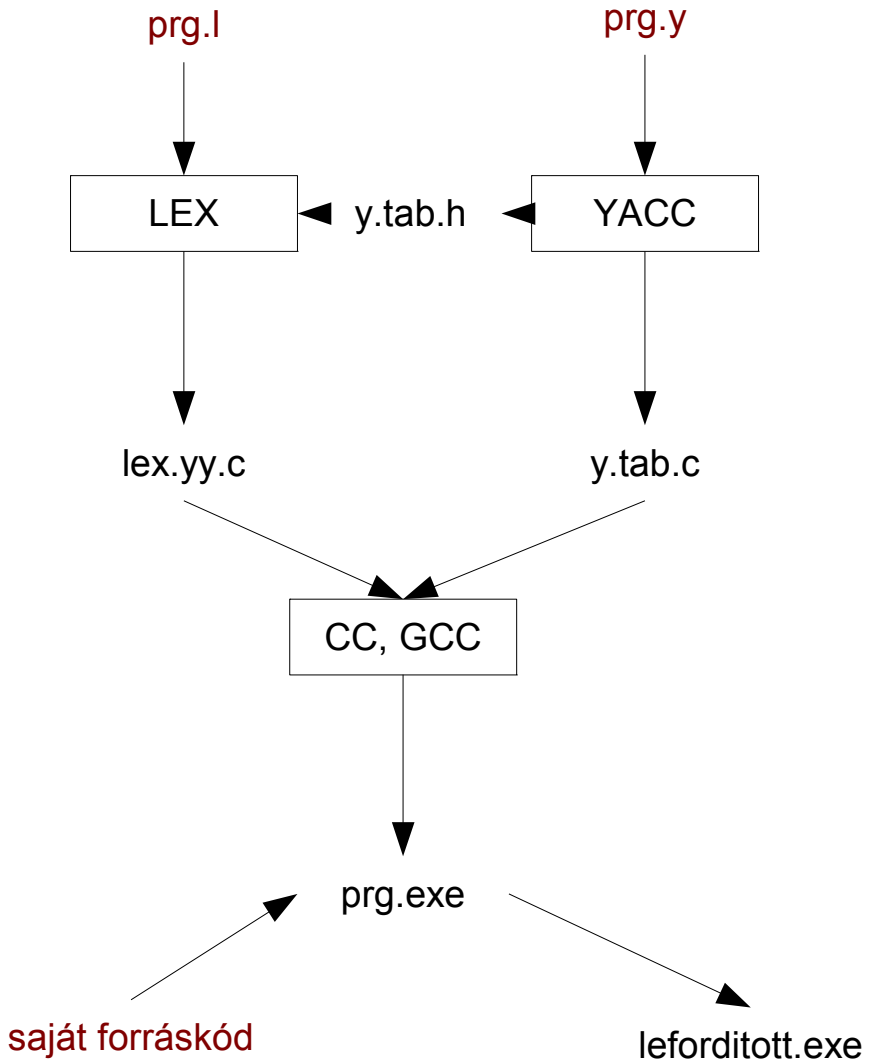


# Fordítóprogramok

## Lex és Yacc



## Lex és Yacc



## Compiler fordítás lépései

```
yacc -d prg.y    -> y.tab.h, és az y.tab.c  
lex prg.l        -> létrehozza a lex.yy.c fájlt  
cc lex.yy.c y.tab.c -o prg.exe    -> fordítás
```

A YACC kiolvassa a nyelvtant a **prg.l** fájlból, majd generálja ebből a szintaktikus elemzőt (parser), és elhelyezi az **yyparse** függvényt az **y.tab.c** fájlba.

A **prg.y** fájl tartalmazza a tokeneket. A **-d** opció hatására a YACC generálja a token definíciókat az **y.tab.h** fájlba.

A LEX kiolvassa a mintákat a **prg.l** fájlból és elhelyezi az **y.tab.h** fájlban, majd generálja a lexikális elemzőt úgy, hogy elhelyezi az **yylex** függvényt a **lex.y.c** fájlba.

Végül a lexer és a parser lefordítására és linkelésére kerül sor a **prg.exe** fájlba.

Az **yyparse** függvényt a main függvényben kell elhelyezni, s ez meghívja a **yylex**, hogy hozzáférjen a tokenekhez.

## Lex - prg.l

LEX input file szerkezete:

definíciók

%%

fordítási szabályok

%%

felhasználói programok

---

```
%{
    #include <stdlib.h>
    void yyerror(char *);
    #include "y.tab.h"
}%
%%
[a-z] {
    yylval = *yytext - 'a';
    return VARIABLE;
}
[0-9]+ {
    yylval = atoi(yytext);
    return INTEGER;
}
[-+()=/*\n] { return *yytext; }
[ \t\n] ;
.      yyerror("invalid character");
%%
int yywrap(void) {
    return 1;
}
```

## Lex - metakarakterek

<b>\n</b>	új sor karakter
<b>.</b>	bármilyen karakter, kivéve az új sor
<b>*</b>	0,1,2... szerez ismétlése a kifejezésnek
<b>+</b>	1,2... szerez ismétlése a kifejezésnek
<b>?</b>	0, 1-szeres ismétlése a kifejezésnek
<b>^</b>	sor kezdete
<b>\$</b>	sor vége
<b>a b</b>	a vagy b
<b>"sss"</b>	string literál
<b>[ ]</b>	karakter osztály
<b>a{1,5}</b>	az 'a' 1-5 közti előfordulása
<b>[^ab]</b>	bármi, kivéve a, vagy b,
<b>[a^b]</b>	a vagy ^ vagy b!
<b>[a-z]</b>	a tól z-ig
<b>[-az]</b>	a, z, '-' karakterek
<b>szoveg&amp;</b>	szöveg a sor végén

### Példák:

digit	<b>[0-9]</b>
delim	<b>[ \t\n]</b>
whitespace	<b>{delim}*</b>
number	<b>{digit}+(.{digit}+)?(E[+-]?{digit}+)?</b>



## Lex- előre definiált elemek

**yytext**

új sor karakter

**yylval**

beolvasott elem értéke

**atoi**

sd

**ECHO**

kiírja az elemet az outputra

stb

/home/roland/Desktop/lex/fourth\_lexer\_test

lang.lex

lang.y

## Fordítás GCC compilerrel

### Példafájl:

```
#include <stdio.h>
int main()
{
    printf("compilertest\n");
    return (0);
}
```

---

### Normál fordítás:

```
gcc test.c -o test.out
```

### Előfordított file:

```
cpp test.c test.i
```

### Assembly file:

```
gcc -S test.c -o test.ass
```

### GNU linker – linkelt file - ld.:

```
nm test.out
```

## Szintaktikus elemző

### Fentről lefelé elemzés

[1]. A lexikális elemző a forrásszöveget terminális jelek sorozatává alakítja. Ez a szintaktikus elemző inputja.

A szintaktikus elemzőnek meg kell határoznia a szintaxisfát, majd ismerve a gyökerét és az elemeit, elő kell állítania az éleket és a többi elemet, vagyis meg kell határoznia a program egy levezetését.

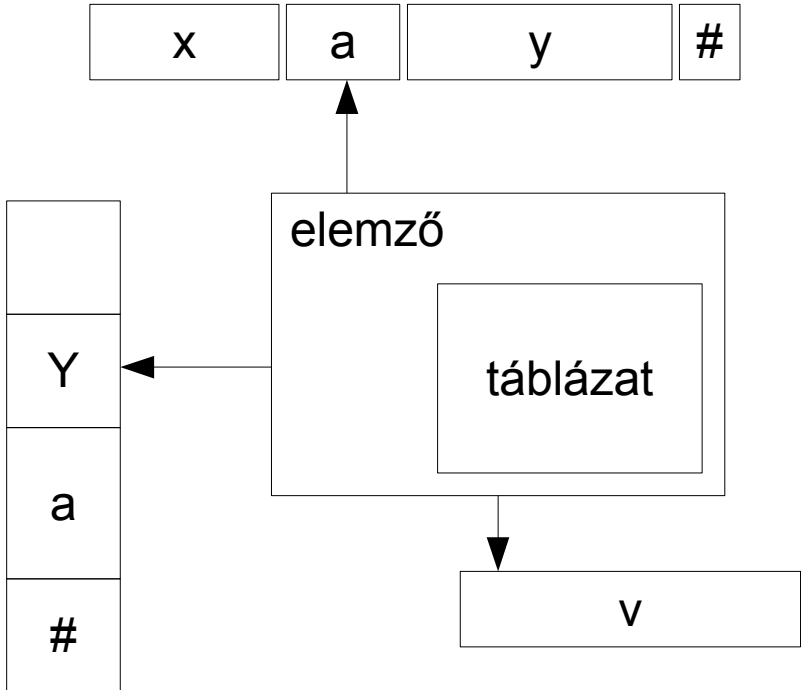
Ha sikerül, a program eleme a nyelvnek, vagyis **szintaktikusan helyes**.

Balról jobbra elemzést használva két módszer létezik:

- **alulról felfelé elemzés:**
  - a levél elemekből indulva haladunk az S szimbólum felé.
- **felülről-lefelé elemzés:**
  - az S szimbólumtól indulva építjük a fát. A cél, hogy a szintaxisfa levélelemeire az elemzett szöveg terminális szimbólumai kerüljenek

## Elemző modellje

[1], [2]. A szintaktikus elemző szerkezete:



[1].

- # az elemzendő szöveg vége és a verem alja.
  - x,a,y Az elemzendő szöveg eleje, az aktuális szimbóluma és a még nem elemzett szöveg.
- Y a verem tetején lévő szimbólum.
- Táblázat az elemzéshez
  - v a szintaxisfa építéséhez szüksége lista

# Rekurzív leszállás módszere

A visszalépés nélküli, felülről lefelé elemzések egyik gyakran alkalmazott módszere, melynek lényege, hogy:

**A grammatika szimbólumaihoz eljárásokat rendelünk**

Majd az elemzés közben a rekurzív eljárásokon keresztül

**A programnyelv implementációja valósítja meg az elemző vermét és a veremkezelést.**

[1]. Legyen egy  $G$  grammatika a következő:

**$G = (\{S, E, E', T, T', F\}, \{+, *, (, ), i, \#\}, P, S)$**

ahol a helyettesítési szabályok:

**$S \rightarrow E\#$**

**$E \rightarrow TE'$**

**$E' \rightarrow +TE' \mid e$**

**$T \rightarrow FT'$**

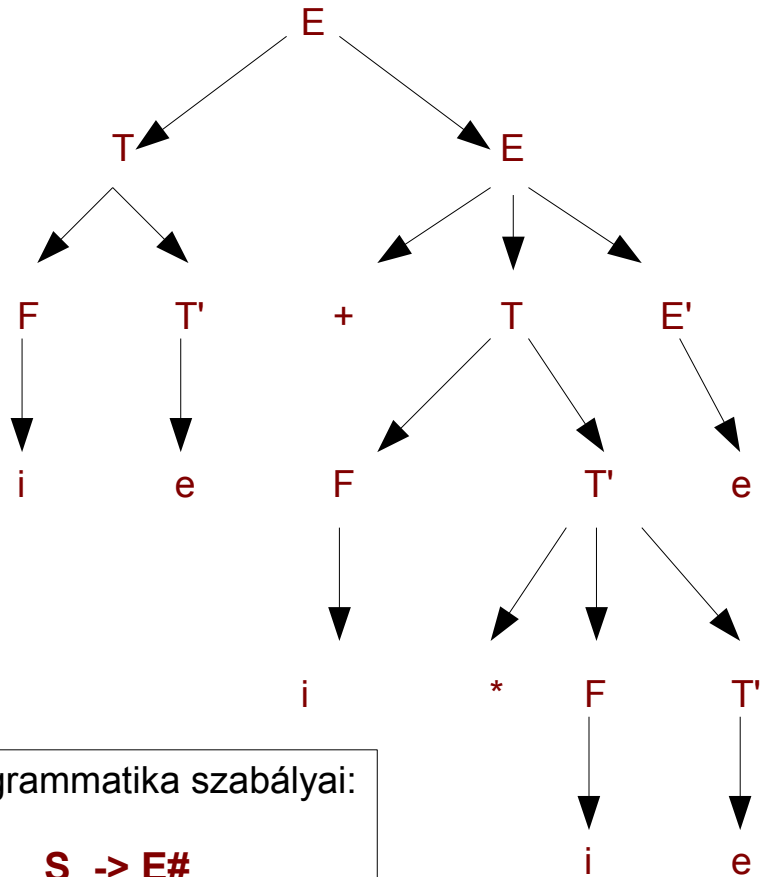
**$T' \rightarrow *FT' \mid e$**

**$F \rightarrow (E) \mid i$**

Mely alapján generálhat Pl.: a következő szintaktikusan helyes mondatot

**$i + i * i$**

## Az $i + i * i$ mondat szintaxisfája



A grammatika szabályai:

$S \rightarrow E\#$   
 $E \rightarrow TE'$   
 $E' \rightarrow +TE \mid e$   
 $T \rightarrow FT'$   
 $T' \rightarrow *FT' \mid e$   
 $F \rightarrow (E) \mid i$

## Elemző eljárásai

Az elkészíthető eljárások a következők:

A terminális szimbólumok vizsgálatának eljárása:

**elfogad(szimbólum)**

```
{  
  if (aktualis_szimbolum == szimbolum)  
    kovetkezo_szimbolum();  
  else error();  
}
```

eljárást, ahol az aktualis\_szimbolum globális változó és a terminális soron következő elemével.

A kovetkezo\_szimbolum() az az eljárás, mely meghívja a lexikális elemzőt, vagyis a következő elemet az aktualis\_szimbolum változóba tölti.

A visszatérési érték lehet a következő szimbólum, vagy hibajelzés.

A grammatika minden szabályához rendelünk egy eljárást. Pl.: az A szabályhoz tartozó eljárás a következő:

```
A()  
{  
  T(A);  
}
```

A T(A) meghatározása mindig az A-ra vonatkozó szabályoktól függ.

# Fordítóprogramok

$S \rightarrow E\#$   
 $E \rightarrow TE'$   
 $E' \rightarrow +TE \mid e$   
 $T \rightarrow FT'$   
 $T' \rightarrow *FT' \mid e$   
 $F \rightarrow (E) \mid i$

$T(A) \rightarrow$

1.  $A \rightarrow a$  szabályhoz az **elfogad(a)** eljárást
2.  $A \rightarrow B$  szabályhoz a **B()** eljárást
3.  $A \rightarrow X_1, X_2, \dots, X_n$  szabályhoz a következő blokkot:

```
{  
    T(X1);  
    T(X2);  
    ...  
    T(Xn);  
}
```

Egyébként

```
választ aktuális_szimbolum  
{  
    elso(X1): T(X1);  
    elso(X2): T(X2);  
    ...  
    elso(Xn): T(Xn);  
    koveto(X): skip;  
}
```

$\{Els\}$  es  $\{koveto\}$  halmazok lsd.: később.



# Fordítóprogramok

Az elemző programja tehát:

```
S()
{
    E();
    elfogad('#');
}
```

---

```
E()
{
    T();
    E'();
}
```

---

```
E'()
{
    valaszt aktualis szimbolum
        {'+'}:{
            elfogad('+');
            T();
            E'();
        }
        {')', '#'}: skip;
}
```

---

```
T()
{
    F();
    T();
}
```

Kov.:

# Fordítóprogramok

```
T'()
{
    választ aktualis_szimbolum
        {'*'}:{
            elfogad('*');
            F();
            T();
        }
    {'+', ')', '#'}: skip;
}
```

---

```
F()
{
    választ aktualis_szimbolum
        {'('}:{
            elfogad('(');
            E();
            elfogad(')');
        }
    {'i'}: accept('i');
}
```

Az elemző főprogramja a grammatika kezdő szimbólumához tartozó eljárás, vagyis az

```
S();
```

# LL(k) elemzők

[1] Ha  $A \rightarrow a \in P$ , akkor az  $xA\beta$  mondatforma **legbaloldalibb helyettesítése**  $x\alpha\beta$ , azaz

$$xA\beta \Rightarrow x\alpha\beta \text{ (legbal).}$$

[2] Ha az  $S \Rightarrow^* x$  ( $X \in T^*$ ) levezetésben minden helyettesítés **legbaloldalibb helyettesítés**, akkor ezt a levezetést **legbaloldalibb levezetésnek** nevezzük.

$$S \Rightarrow^* x \text{ (legbal).}$$

[1] Ha  $S \Rightarrow^* x$  és  $x \Rightarrow^* yz$  ( $a \in (N \cup T)^*$ ,  $x, y, z \in T^*$ ) és  $|x| = |y|$ ,  $x = y$

**magyarázat:**

A mondatforma bal oldalán a terminálisokból álló  $x$  sorozatot a környezetfüggetlen nyelvtan helyettesítési szabályai nem változtatják meg.

Ezt **kezdőszelet egyeztetésnek** hívják. Ha a szintaxisfa építéskor a bal oldali terminálisok nem egyeznek meg az elemzendő szöveg bal oldalán állókkal, akkor a fa építése rossz irányba halad. Ekkor vissza kell lépni.

Az ilyen visszalépéses algoritmusok alkalmazása rendkívül költséges.

## Alapfogalmak

[1], [2] LL(k) nyelvtanok alaptulajdonsága:

Az  $S \Rightarrow^* wx$  legbaloldalibb levezetés építése során eljutunk a  $S \Rightarrow^* wA\beta$  mondatformáig és az  $A\beta \Rightarrow^* x$ -t szeretnénk elérni, akkor az A-ra alkalmazható  $A \rightarrow a$  helyettesítést egyértelműen meghatározhatjuk az  $x$  első  $k$  db szimbólumának előre olvasásával.

(Ekkor és csak ekkor LL(k) nyelvtanról beszélhetünk.)

Erre megadható az *Első  $k$*  függvény.

*Első  $k(a)$*  legyen az  $a$ -ból ( $k \geq 0$ ,  $a \in (N \cup T)^*$ ) levezethető szimbólumsorozatok  $k$  hosszúságú kezdő terminálisainak halmaza. (*Első $k(x)$*  az  $x$  első  $k$  db szimbólumát tartalmazza)

Pl.:

LL(1) nyelvtan:

$G = (\{A, S\}, \{a, b\}, P, S)$ , ahol  $a$

$P = S \rightarrow AS \mid \varepsilon$

$A \rightarrow aA \mid b$

Az  $S$  szimbólumra az  $S \rightarrow AS$  szabályt kell alkalmazni, ha az elemzendő szövegben  $a$ , vagy  $b$  a következő elem, és  $S \rightarrow \varepsilon$ , ha a következő szimbólum a  $\#$  (végjel).

# Alapfogalmak

Az LL(k) elemzéseknél nem elég csak a szabályokat vizsgálni. Ha pl.: van az A-ra egy

$A \rightarrow \varepsilon$

szabály, akkor az Elsők halmazban a B-ből származó terminális sorozatok k hosszúságú kezdő szimbólumai is szerepelnek. ezért a nem mindig véges számú levezetéseket is figyelembe kell venni.

Gyakorlati módszer csak az LL(1) nyelvtanokra létezik.

Szükség van egy Követők függvény definiálására, mely megadja a szimbólumsorozatot követő k hosszúságú terminális sorozatok halmazát.:

**Követő(A)** ( $A \in N$ ) tehát azokat a terminális szimbólumokat tartalmazza, melyek a levezetésben közvetlenül az A után állhatnak.

Mind az Elsők, mind a Követők halmaz meghatározására létezik gyakorlatban használható algoritmus. [2]

# Táblázatos elemzés

[1] Az elemzendő terminális sorozat  $xay$ , ahol az  $x$  szöveget már elemeztük.

Felülről lefelé legbaloldalibb helyettesítéseket alkalmazunk, szintaktikus hibát nem találtunk, így az elemzendő mondatformánk  **$xYa$** , azaz vagy:

(1.)  **$xBa$**  vagy,

(2.)  **$xba$**  alakú.

(1.) a szintaxisfa építéskor a  $B$ -t kell helyettesítenünk egy  $B \rightarrow \beta$  szabállyal amelyre igaz, hogy ( **$a \in \text{Első}(\beta \text{ Követő}(B))$** ).

Ha van ilyen szabály, akkor pontosan egy van (mivel  $LL(1)$  a nyelvtan), különben szintaktikai hiba helyét detektáltuk.

(2.) a második esetben a mondatforma következő szimbóluma a  $b$  terminális jel, tehát az elemzendő szövegben is  $b$ -nek kell szerepelnie.

Ha így van, továbblépünk, ha nem, akkor szintaktikai hiba helyét detektáltuk.

# Elemző működése

[1],[2] Az elemző állapotait egy

(ay#, Xa#, v) hármassal írjuk le, ahol a

# jel az elemzendő szöveg vége és a verem alja,

ay# a még nem elemzett szöveg,

Xa# az elemzendő szöveg mondatformájának még nem elemzett része ez van a veremben,

v a szabályok sorszámát tartalmazó lista.

A helyettesítési szabályokat megszámozzuk. Az elemzés során alkalmazott szabályok sorszámát a v listában tároljuk.

Az elemzés során mindig a verem tetején lévő X szimbólumot hasonlítja össze a még nem elemzett szöveg következő szimbólumával (a-val). Az a-t aktuális szimbólumnak nevezzük.

Kezdőállapot: (xay#, S#, ε)

Az elemzés egy T elemző táblázattal végezhető el. A T táblázat kitöltésére megfelelően gyors és jól használható algoritmusok léteznek.

# Az elemző táblázata

$T[X, a] = \{ (B, i)$	ha $X \rightarrow B$ a $E$ Első( $B$ ) vagy ( $e \in E$ Első( $B$ ) és $a \in E$ Követő( $X$ )).
pop	ha $X = a$ ,
elfogad	ha $X = \#$ és $a = \#$ ,
hiba	egyébként.

## Az elemző algoritmus

Az elemző működése állapotátmenetekkel adható meg. A kezdeti állapot ( $x\#, S\#, \epsilon$ ). Sikeres befejezésnél a végállapot ( $\#, \#, w$ ).

Ha a még nem elemzett szöveg az  $ay\#$ , és a verem tetején az  $X$  szimbólum áll, az állapotátmenetek a következők:

$(ay\#, Xa\#, v) \rightarrow \{$	$(ay\#, Ba\#, vi)$	ha $T[X, a] = (B, i)$ ,
$(y\#, a\#, v)$	OK.	ha $T[X, a] = \text{pop}$ .
	HIBA	ha $T[X, a] = \text{elfogad}$
		ha $T[X, a] = \text{hiba}$

LL(1) elemzés:  
algoritmus...



## Példa táblázatos elemzésre

# Rekurzív-leszállás módszere

A Rekurzív-leszállás módszere balról jobbra elemzi a szöveget, úgy, hogy a nyelvtan szabályait eljárásokkal implementálja és a verem kezelését, valamint a rekurzív hívásokat a programnyelv implementációjára bízza.

Minden szabályhoz egy eljárást rendel:

$S \rightarrow T$

$T \rightarrow \dots$

Procedute S

```
{  
  T()  
}
```

Procedure T{...}

...

A nyelvtan kezdő szimbólumához rendelt eljárás mindig az elemző főprogramja, mely elindítja a rekurzív hívásokat.

A módszert a fentről lefelé elemzés és a rekurzív eljáráshívások miatt nevezik Rekurzív-Leszállás módszernek.

## Kalkulátor példa (lexer)

```
%{
    #include <stdlib.h>
    void yyerror(char *);
    #include "y.tab.h"
}%
%%
[0-9]+    {
    yylval = atoi(yytext);return INTEGER;
}
[-+\n]    return *yytext;
[ \t]     ; /* skip whitespace */
.         yyerror("invalid character");
%%
int yywrap(void)
{
    return 1;
}
```

## Kalkulátor példa (parser)

```
%{
    int yylex(void);
    void yyerror(char *);
}%
%token INTEGER
%%
program:
    program expr '\n'      { printf("%d\n", $2); }
    |;
expr:
    INTEGER                { $$ = $1; }
    | expr '+' expr        { $$ = $1 + $3; }
    | expr '-' expr        { $$ = $1 - $3; }
    ;
%%
void yyerror(char *s)
{
    printf("%s\n", s);
}
int main(void)
{
    yyparse();
    return 0;
}
```

## Kibővített példa (lexer)

```
%{
    #include <stdlib.h>
    void yyerror(char *);
    #include "y.tab.h"
}%
%%
[a-z]      {
            yylval = *yytext - 'a';
            return VARIABLE;
        }
[0-9]+     {
            yylval = atoi(yytext);
            return INTEGER;
        }

/* operators */
[-+()=/*\n] { return *yytext; }
/* skip whitespace */
[ \t]      ;
/* anything else is an error */
.          yyerror("invalid character");
%%
int yywrap(void) {
    return 1;
}
```

## Kibővített (parser)

```

%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'
%{
    void yyerror(char *);
    int yylex(void);
    int sym[26];
}%
%%
program:
    program statement '\n' |;
statement:
    expr                                { printf("%d\n", $1); }
    | VARIABLE '=' expr                { sym[$1] = $3; }
    ;
expr:
    INTEGER
    | VARIABLE                        { $$ = sym[$1]; }
    | expr '+' expr                    { $$ = $1 + $3; }
    | expr '-' expr                    { $$ = $1 - $3; }
    | expr '*' expr                    { $$ = $1 * $3; }
    | expr '/' expr                    { $$ = $1 / $3; }
    | '(' expr ')'                     { $$ = $2; }
    ;
%%
void yyerror(char *s) { printf("%s\n", s); }
int main(void) { yyparse(); return 0; }

```

## Fordítás és output

```
yacc -d bas.y
```

```
#létrehozza az y.tab.hfájlt y.tab.c
```

```
lex bas.l
```

```
#létrehozza a lex.yy.c
```

```
cc lex.yy.c y.tab.c -obas.exe
```

```
# fordítás és linkelés
```

```
A kész exe: bas.exe
```

```
Futtatás: bas.exe
```

```
User: 3+2
```

```
Prg : 5
```

```
User: 3 * (1 + 2)
```

```
Prg : 9
```

# Parser kibővítése (fájlból olvasás)

```
%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'
%{
    #include <stdio.h> //filekezeléshez
    #include <stdlib.h>
    int yyerror(char *s);
    int yylex(void);
    int sym[26];
}%
%%
```

A köztes rész marad az eredeti

```
int main ( int argc, char *argv[] )
{
    extern FILE *yyin;
    ++argv;--argc;
    yyin=fopen( argv[0], "r");
    yyparse();
    printf("Parse Completed\n");
    return 0;
}
```



## BNF – Implementációja

program : LET declarations IN commands END;

declarations: type id\_seq;

type: INT | FLOAT;

id\_seq:  
     IDENTIFIER { setType(\$1, \$0); }  
     | id\_seq ',' IDENTIFIER { setType(\$3, \$0); };

id\_seq:  
     | id\_seq IDENTIFIER ',' ;

commands:  
     | commands command ',' ;

command: SKIP  
     | READ IDENTIFIER  
     | WRITE exp  
     | IDENTIFIER ASSGNOP exp  
     | IF exp THEN commands ELSE commands FI  
     | WHILE exp DO commands END;

exp:   NUMBER  
       | IDENTIFIER  
       | exp '<' exp       { \$\$ = \$1 < \$3; }  
       | exp '=' exp       | exp '>' exp  
       | exp '+' exp       | exp '-' exp  
       | exp '\*' exp       | exp '/' exp  
       | exp '^' exp       | '(' exp ')';

## Gyakorlatok anyaga

Ismerkedés a programozási környezettel. Linux, C parancssorból, linux shell programozás. reguláris kifejezések programozása C#-nyelv reguláris kifejezéseinek (regex) segítségével.

Input handler programjának elkészítése C#-nyelv reguláris kifejezéseinek (regex) felhasználásával. Determinisztikus véges automata programja jegyzet alapján C# nyelven, regex segítségével.

Rekurzív leszállás módszere Program szintaktikus elemzőre a példa alapján: Yacc/Lex bemutatása.

Saját teszt programnyelv megtervezése, szintaxis definiálása BNF segítségével. Lexer és parser készítése Lex/Yacc programmal.

Lexer és Parser összekapcsolása, kódgenerálás. Szimbólum tábla modul implementációja. Virtuális gép implementációja C nyelven.

Saját virtuális gép, szimbólum tábla, lexer és parser összekapcsolása egy programban, majd a fordítóprogram elkészítése.

Szintaktikus elemző LL(1), LR(1) gyakorlás. Szemantikus elemzők működésének vizsgálata, gyakorlás. Kód generálás, optimalizálás.

## Felhasznált irodalom:

- I. Csörnyei Zoltán, *Fordítóprogramok*, ELTE Budapest, 2005
- II. Kása Zoltán, Csörnyei Zoltán, *Formális nyelvek és fordítóprogramok*, Babes-Boyai, Kolozsvár, 2007
- III. Tom Nieman, *A compact Lex & Yacc* Portland, Oregon, epaperpress.com
- IV. Anthony A., *Compiler Construction using Flex and Bison*, AabWalla Walla College April 22, 2005
- V. Hernyák Zoltán, *Formális nyelvek és automaták*, EKF, Eger, 2001