Fordítóprogram (compiler), értelmezőprogram (interpreter)

magasszintű programozási nyelvek fordításával foglalkozunk,

imperatív programozási nyelvek fordítási algoritmusait tanulmányozzuk

Ha a forrásnyelv magasszintű nyelv, akkor a forrásnyelv és a számítógép által végrehajtható gépi kód nagyon különböznek egymástól.

Ezt a különbséget kétféleképpen lehet áthidalni: fordítóprogrammal vagy értelmezőprogrammal.

• 1. módszer: magasszintű nyelven írt programból egy alacsonyabb szintű (assembly nyelvű vagy gépi kódú) programot készítünk

Az ilyen átalakítást végző program: fordítóprogram (angolul compiler, románul compilator).

bemenete: forrásnyelvű program vagy röviden forrásprogram
eredménye: tárgynyelvű program vagy vagy röviden tárgyprogram

fordítási idő, futtatási idő vagy futási idő egymástól jól elkülöníthető időintervallumok

• 2. módszer: készítünk egy olyan gépet, amelyik a magasszintű nyelvet értelmezi, azaz amelyiknek a gépi kódja ez a magasszintű nyelv.

Ha ezt a gépet hardver úton valósítjuk meg, akkor ezt a gépet formulavezérlésű számítógépnek, ha programmal valósítjuk meg, akkor ezt a programot interpreternek (értelmezőprogramnak) nevezzük.

Itt a fordítási és a futási idő egybeesik, és az interpreter nem készít tárgyprogramot.

A fordítás szempontjából a compilereket és az interpretereket nem kell megkülönböztetnünk.

A kezdetek

Néhány példa magasszintű programozási nyelvekre:

FORTRAN

```
C AREA OF A TRIANGLE - HERON'S FORMULA
C INPUT - CARD READER UNIT 5, INTEGER INPUT
C OUTPUT -
C INTEGER VARIABLES START WITH I,J,K,L,M OR N
      READ(5,501) IA,IB,IC
  501 FORMAT(3I5)
      IF(IA.EQ.0 .OR. IB.EQ.0 .OR. IC.EQ.0) STOP 1
      S = (IA + IB + IC) / 2.0
      AREA = SQRT( S * (S - IA) * (S - IB) * (S - IC) )
      WRITE(6,601) IA,IB,IC,AREA
  601 FORMAT(4H A= ,I5,5H  B= ,I5,5H  C= ,I5,8H  AREA= ,F10.2,
     $13H SQUARE UNITS)
      STOP
      END
```

COBOL

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  Formula-Heron.
AUTHOR.  Ion IVAN.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  A          PIC 9(5)  VALUE ZEROS.
77  B          PIC 9(5)  VALUE ZEROS.
77  V          PIC 9(5)  VALUE ZEROS.
77  SUM     PIC 9(5)  VALUE ZEROS.
77  P          PIC 9(5)V99  VALUE ZEROS.
77  TMP1   PIC 9(5)V99  VALUE ZEROS.
77  TMP2   PIC 9(5)V99  VALUE ZEROS.
77  TMP3   PIC 9(5)V99  VALUE ZEROS.
77  S          PIC 9(8)V99  VALUE ZEROS.
PROCEDURE DIVISION.
CITIRE-DATE.
      DISPLAY "Introduceti prima latura a triunghiului : " WITH NO ADVANCING.
      ACCEPT A.
      DISPLAY "Introduceti a doua latura a triunghiului : " WITH NO ADVANCING.
      ACCEPT B.
      DISPLAY "Introduceti a treia latura a triunghiului : " WITH NO ADVANCING.
      ACCEPT C
CALCULE..
      ADD A TO SUM.
      ADD B TO SUM.
      ADD C TO SUM.
      DIVIDE 2 INTO SUM GIVING P.
      SUBTRACT A FROM P GIVING TMP1.
      SUBTRACT B FROM P GIVING TMP2.
      SUBTRACT C FROM P GIVING TMP3.
      COMPUTE S=P*TMP1*TPM2*TMP3.
      COMPUTE S=S**.5.
      DISPLAY "Aria triunghiului este = ", S.
   SFARSIT-PROGRAM.
   STOP RUN.
```

BASIC

```
10 PRINT "Area of a Regular Triangle"
20 INPUT "Enter base: "; b
30 INPUT "Enter height: "; h
40 LET A = (0.5) * b * h
50 PRINT "Area = " ; A
```

Sample Output (with test values):

```
Area of a Regular Triangle
Enter base: 10
Enter height: 8
Area = 40
```

C

```
#include<stdio.h>
#include<math.h>
void main()
{
    float a,b,c,p=0,s=0;
    printf("Introduceti lungimile in metri ale laturilor triunghiului \n");
    scanf("%f %f %f",&a,&b,&c)
    if(a>0 && b>0 && c>0)
            {
               p = (a+b+c)/2.0; /* s este semiparametrul*/
               s = (sqrt)(p*(p-a)*(p-b)*(p-c));
               printf("\n Area of triangle =\t %f",s);
             }
             else printf("\n Laturile nu formează un triunghi");
    getch();
}
```

C++

```
int main()
{
    float side1, side2, side3, area, s;
    cout << "\n\n Find the area of any triangle using Heron's Formula :\n";
    cout << "------------------------------------------------------------\n";
    cout<<" Input the length of 1st side  of the triangle : ";
    cin>>side1;
    cout<<" Input the length of 2nd side  of the triangle : ";
    cin>>side2;
    cout<<" Input the length of 3rd side  of the triangle : ";
    cin>>side3;
    s = (side1+side2+side3)/2;
    area = sqrt(s*(s-side1)*(s-side2)*(s-side3));
    cout<<" The area of the triangle is : "<< area << endl;
    cout << endl;
    return 0;
}
```

Java

```
public class Demo {
   public static void main(String[] args) {
      // sides of a triangle
      double s1, s2, s3;
      double area, resArea;
      // three sides of a triangle
      s1 = 15191235.0;
      s2 = 15191235.0;
      s3 = 1.01235479;
      area = (s1+s2+s3)/2.0d;
      resArea = Math.sqrt(area* (area - s1) * (area - s2) * (area - s3));
      System.out.println("Area of Triangle = " + resArea);
   }
}
```

Python

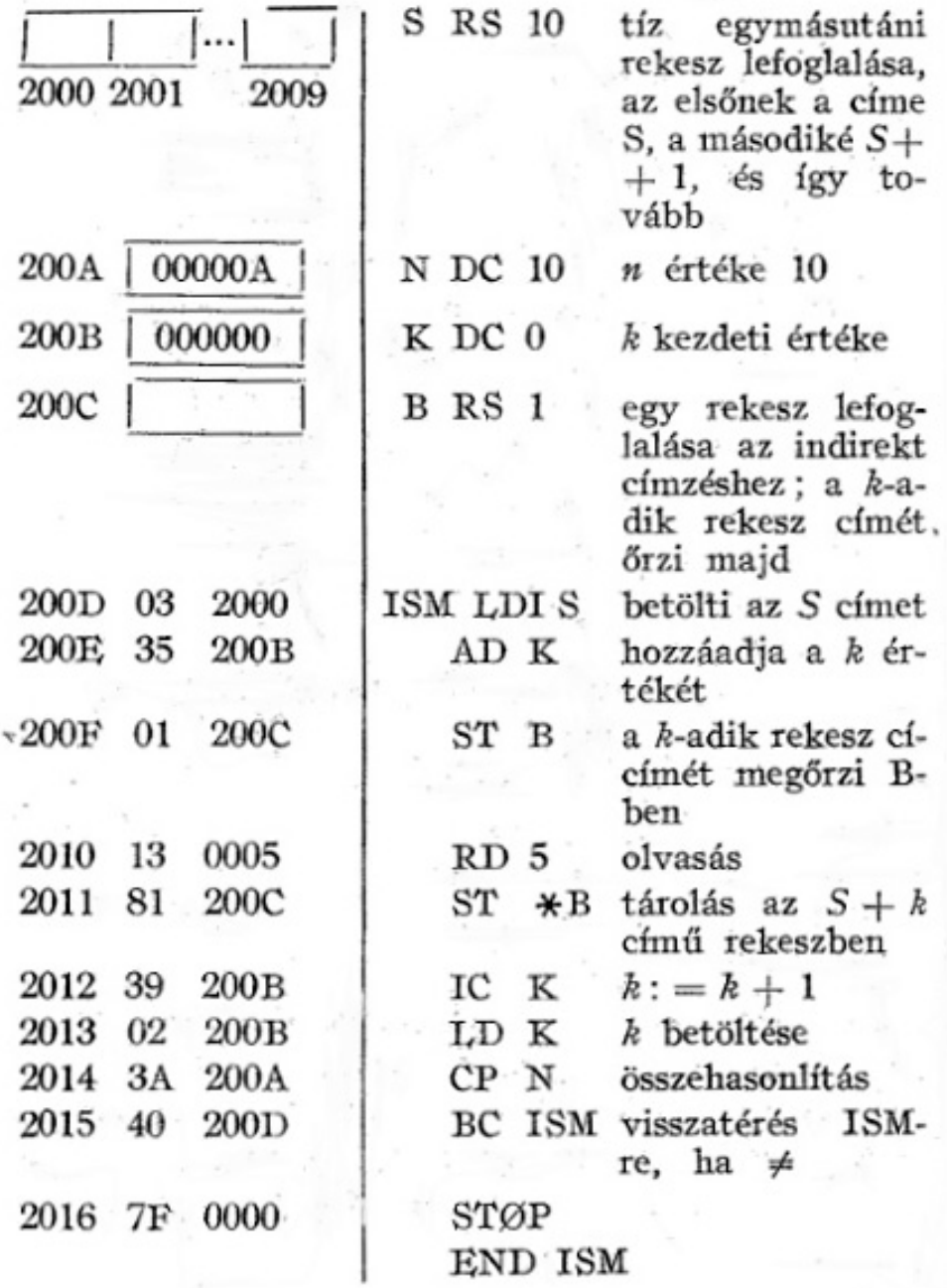
```
def calculateTriangleArea(a, b, c):
  perimeter = a + b + c
  s = perimeter / 2
  area = (s*(s-a)*(s-b)*(s-c))**0.5
  return area

side1 = float(input("Enter the length of the first side:"))
side2 = float(input("Enter the length of the second side:"))
side3 = float(input("Enter the length of the third side:"))

area = calculateTriangleArea(side1, side2, side3)

print("The area of your triangle is: " + str(area))
```

Programozás gépi kódban és assembly nyelven



2000 2001 ... 2009			S	RS 10	tíz egymásutáni rekesz lefoglalása, az elsőnek a címe S, a másodiké $S+1$, és így tovább
200A	00000A		N	DC 10	n értéke 10
200B	000000		K	DC 0	k kezdeti értéke
200C			B	RS 1	egy rekesz lefoglalása az indirekt címzéshez; a k-adik rekesz címét őrzi majd
200D	03	2000	ISM	LDI S	betölti az S címet
200E	35	200B		AD K	hozzáadja a k értékét
200F	01	200C		ST B	a k-adik rekesz címét megőrzi B-ben
2010	13	0005		RD 5	olvasás
2011	81	200C		ST *B	tárolás az $S+k$ című rekeszben
2012	39	200B		IC K	$k := k+1$
2013	02	200B		LD K	k betöltése
2014	3A	200A		CP N	összehasonlítás
2015	40	200D		BC ISM	visszatérés ISM-re, ha $\neq$
2016	7F	0000		STØP	
				END ISM	

KZ: Ismerkedés az informatikával. 64–75. old.

A fordítóprogram szerkezete

A fordítóprogram a forrásnyelvű programot egy tárgyprogramra fordítja le. A fordítás folyamatáról, a fordítás eredményéről egy listát is készít, amely a programozó számára visszaigazolja a forrásnyelvű szöveget, tartalmazza a felfedezett hibákat, és információt ad a tárgyprogramról is, például közli az egyes utasításoknak a program elejéhez viszonyított relatív kezdőcímét.

A fordítóprogram struktúrája a következő:

jelölés: program(bemenet)(kimenet)

fordítóprogram(forrásnyelvű program)(tárgyprogram, lista),

Ezt a jelölésmódot alkalmazva, a továbbiakban a fordítóprogram struktúráját finomítjuk.

A forrásnyelvű program egy adathordozón, általában egy állományban, az operációs rendszertől függő formátumban helyezkedik el.

A fordítóprogram első lépése ezért egy olyan program végrehajtása lesz, amelyik ezt a forrásnyelvű programot a fordítás számára könnyen hozzáférhető karaktersorozattá alakítja át. Ezt a műveletet végzi el az bemenetkezelő (*input-handler*):

bemenetkezelő(forrásnyelvű program)(karaktersorozat)

Az input-handler a forrásnyelvű program egy vagy több rekordját egy pufferben helyezi el, meghívásakor innen olvassa a forrásnyelvű program soron következő sorát, levágva például már a további feldolgozás szempontjából közömbös kocsivissza és soremelés karaktereket is.

Ezekből a forrásnyelvű sorokból készül a lista. A lista összeállítását a kimenetkezelő (*output-handler*) végzi, amely a listát szintén

a háttértárolón, egy állományban, az operációs rendszertől függő formátumban helyezi el:

kimenetkezelő(forrásnyelvű program, hibák)(lista).

A fordítóprogramok szinte mindig hibás programokat fordítanak, ezért a fordítóprogramok íróinak rendkívül nagy figyelmet kell fordítaniuk a lista formájára, a hibajelzés módjára.

Az output-handlerhez hasonlóan, a compiler által készített tárgykódot is háttértárolón, egy fájlban, például relokálható bináris formátumban kell elhelyezni, a formátum természetesen ismét az operációs rendszertől függ. Ezt a műveletet a kódkezelő *code-handler* végzi el:

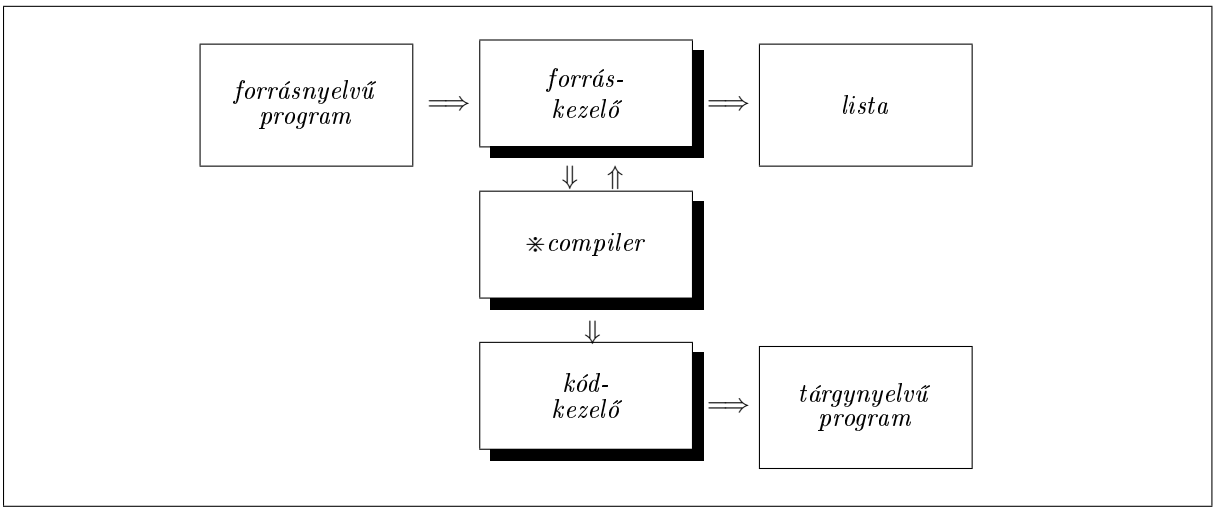
kódkezelő(tárgykód)(tárgyprogram).

Mivel mind az input-handler, mind az output-handler bemenete a forrásnyelvű program, célszerű ezeket egy programba összefogni. Nevezzük ezt a programot forráskezelőnek (*source-handlernek*):

forráskezelő(forrásnyelvű program, hibák)(karaktersorozat, lista).

Így tehát a fordítóprogram struktúrája a következő lesz:

forráskezelő(forrásnyelvű program, hibák)(karaktersorozat, lista),
∗compiler(karaktersorozat)(tárgykód, hibák),
kódkezelő(tárgykód)(tárgyprogram).



1.. ábra. A fordítóprogram felépítése

Ez a felbontás nem szekvenciát jelöl, a három programelem nem szekvenciálisan hajtódik végre. A fenti felbontással a fordítóprogramot három egymástól jól elkülöníthető funkcionális, működési egységre bontottuk fel. Az egyes működési egységek kapcsolatát az egységek bemenete és kimenete jelzi.

A *source-handler* és a *code-handler* végzi el az összes perifériafüggő és az összes, operációs rendszertől függő műveletet, a ∗*compiler* most már ténylegesen csak a fordítással foglalkozik. A két handlerrel, elsősorban a számítógéptől, a perifériáktól és az operációs rendszerektől való függőségük miatt, a továbbiakban nem foglalkozunk, bár a fordítóprogramok külső jellemzőit, kezelhetőségét, a felhasználóval való kapcsolatát alapvetően ezek határozzák meg.

A középső programelemnek, a ∗*compiler*-nek két nagy feladatot kell megoldania:

– elemeznie kell a bemenetén kapott karaktersorozatot, és
– szintetizálnia kell a tárgykódot.

Az analízis a forrásnyelvű program karaktersorozatát részekre bontja és ezt vizsgálja, míg a szintézis az egyes részeknek megfelelő tárgykódokból építi fel a program teljes tárgykódját.

Az *analízis* első feladata az, hogy a karaktersorozatban meghatározza az egyes szimbolikus egységeket, a konstansokat, változókat, kulcsszókat, operátorokat. Azt a programelemet, amelyik ezt a feladatot végzi, lexikális elemzőnek nevezzük.

A lexikális elemző a karaktersorozatból szimbólumsorozatot készít:

lexikális elemző(karaktersorozat)(szimbólumsorozat, lexikális hibák).

A létrehozott szimbólumsorozat ábrázolása hatékonysági okokból általában kódolt, egy szimbólumot egy típuskód és egy cím határoz meg, a típuskód a szimbólum típusára utal, a cím pedig a szimbólumtáblának az a címe, ahol az elemző a szimbólumhoz tartozó szöveget elhelyezte.

A lexikális elemzőnek kell kiszűrnie a szóköz karaktereket, a forrásnyelvű programba írt megjegyzéseket. A magasszintű programnyelvek egy utasítása több sorba is írható, a lexikális elemző feladata egy több sorba írt utasítás összeállítása is.

A lexikális elemző által készített szimbólumsorozat a bemenete a szintaktikai elemzőnek.

A szintaktikai elemzőnek a feladata a program struktúrájának a felismerése és ellenőrzése. A szintaktikus elemző azt vizsgálja, hogy a szimbólumsorozatban az egyes szimbólumok a megfelelő helyen vannak-e, a szimbólumok sorrendje megfelel-e a programnyelv szabályainak, nem hiányzik-e esetleg valahonnan egy szimbólum.

szintaktikai elemző(szimbólumsorozat)(szintaktikusan elemzett program, szintaktikai hibák).

Az analízis harmadik programja a szemantikai elemző, amelynek a feladata bizonyos szemantikai jellegű tulajdonságok vizsgálata. A szemantikai elemző feladata például az $\langle$*azonosító*$\rangle$ + $\langle$*konstans*$\rangle$ kifejezés elemzésekor az, hogy az összeadás műveletének felismerése után megvizsgálja, hogy az $\langle$*azonosító* $\rangle$-val megadott szimbólum deklarálva van-e, a $\langle$*konstans*$\rangle$-nak van-e értéke, és típusuk azonos-e.

szemantikai elemző(szintaktikusan elemzett program)(elemzett program, szemantikai hibák).

A szemantikai elemző kimenete lesz a szintetizálást végző programok bemenő adata. A szintaxisfa alakjában, vagy például lengyelformában ábrázolt elemzett programból készül el a tárgyprogram. A szintézis első lépése a kódgenerálás, amelyet a kódgenerátor program végez el:

kódgenerátor(elemzett program)(tárgykód)

A tárgykód a forrásnyelvű program egy közbülső programformájának tekinthető. Ez a kód lehet a számítógéptől és az operációs rendszertől független speciális kód, de a legtöbb fordítóprogram tárgykódként az adott számítógép assembly nyelvű vagy gépi kódú programját állítja elő.

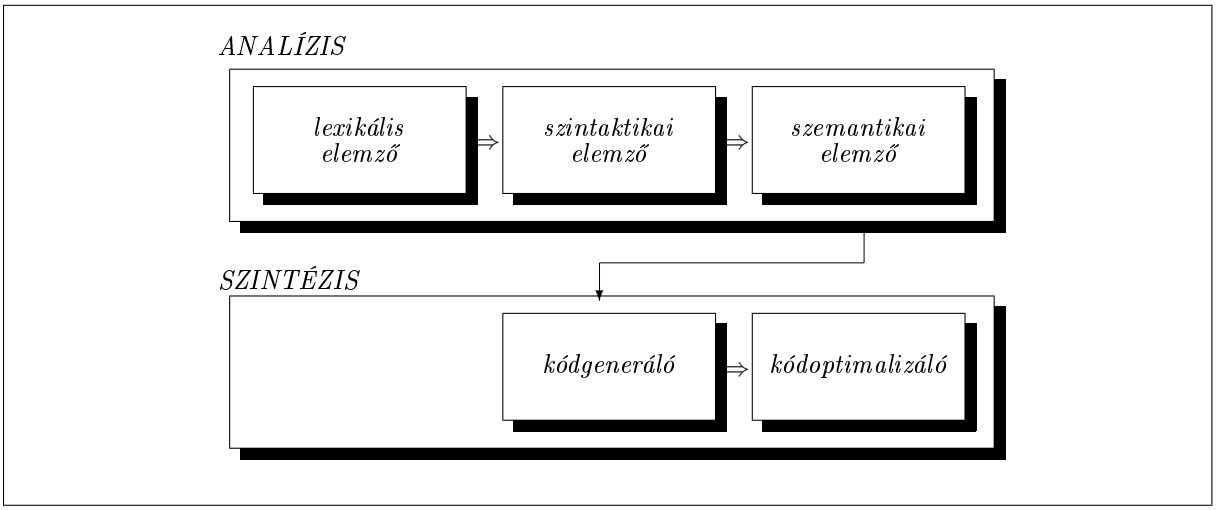
A szintetizálás következő lépése a kódoptimalizálás:

kódoptimalizáló(tárgykód)(tárgykód)

A kódoptimalizálás a legegyszerűbb esetben a tárgykódban levő azonos programrészek felfedezését és egy alprogramba való helyezését, vagy a hurkok ciklusváltozótól független részeinek megkeresését és a hurkon kívül való elhelyezését jelenti. Bonyolultabbak

a gépfüggő kódoptimalizáló programok, amelyek például optimális regiszter-használatot biztosítanak.

A fordítóprogramok íróinak a célja, hogy a kódoptimalizáló jobb és hatékonyabb tárgyprogramot állítson elő, mint amit egy (az assembly nyelvű programozásban) gyakorlott programozó készíteni tud.



2.. ábra. Az analízis és szintézis programjai

A fordítóprogram a következő részekre bontható:

forráskezelő(forrásnyelvű program, hibák)(karaktersorozat, lista),

lexikális elemző(karaktersorozat)(szimbólumsorozat, lexikális hibák),

szintaktikai elemző(szimbólumsorozat)(szintaktikusan elemzett program, szintaktikai hibák),

szemantikai elemző(szintaktikusan elemzett program)(elemzett program, szemantikai hibák),

kódgenerátor(elemzett program)(tárgykód),

kódoptimalizáló(tárgykód)(tárgykód),

kódkezelő(tárgykód)(tárgyprogram).

A fordítóprogramok analízist és szintézist végző részének algoritmusa a következőképpen írható le:

⁎Fordítóprogram

1. határozzuk meg a forrásnyelvű program szövegében a lexikális szimbólumokat
2. ellenőrizzük a szimbólumsorozat szintaktikai helyességét
3. ellenőrizzük a szimbólumsorozat szemantikai helyességét
4. határozzuk meg a tárgyprogramba kerülő kódot
5. optimalizáljuk a tárgyprogram kódját

A következőkben ezekkel a programokkal foglalkozunk.

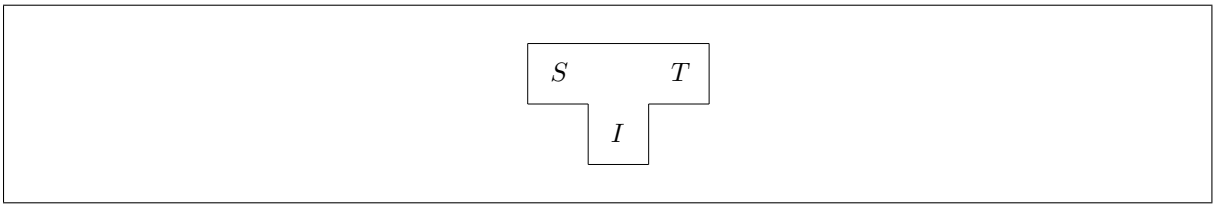
Kezdő lépések

A fordítóprogramok bonyolult programok, célszerű őket valamilyen magasszintű nyelven írni.

Egy fordítóprogram három nyelvvel jellemezhető: forrásnyelv, tárgynyelv és az a nyelv, amelyen a fordítóprogramot megírták, azaz implementálták.

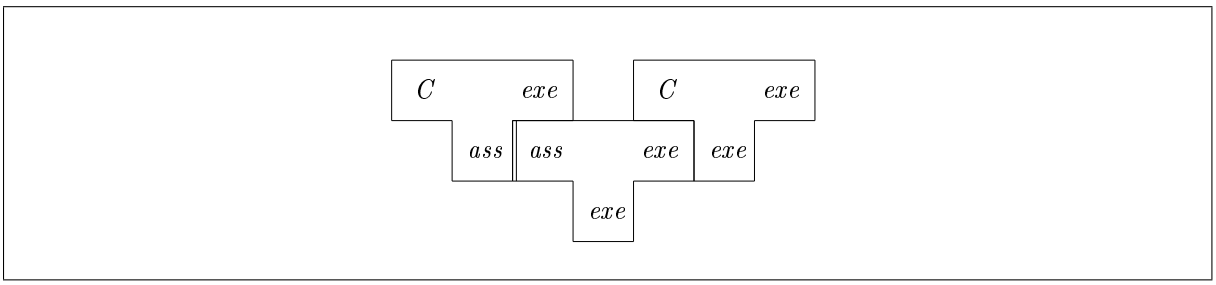
Ez a három nyelv természetesen lényegesen különbözhet egymástól. Ha a fordítóprogram nem annak a gépnek a kódjára fordít, mint amelyiken fut, akkor a fordítóprogramot kereszt-fordítóprogramnak nevezzük.

A fordítóprogram három nyelvét egy *T-diagrammal* ábrázolhatjuk, S-sel a forrásnyelvet, T-vel a tárgynyelvet, I-vel az implementáció nyelvét jelöltük. A programot az I nyelven implementált $S \to T$ fordítóprogramnak nevezzük.



3.. ábra. A T-diagram

Két T-diagram egymáshoz illeszthető egy olyan ponton, ahol a két diagramban azonos nyelv van. Az ábrákon az illesztési pontokat dupla vonallal jelöljük. Például, ha van egy *ass* assembly nyelven írt $C \to exe$ fordítóprogramunk, és van egy *exe* végrehajtható kódú $ass \to exe$ assemblerünk, akkor ezzel az assemblerrel elő tudunk állítani egy végrehajtható $C \to exe$ fordítóprogramot.



4.. ábra. A $C \to exe$ fordítóprogram

A *bootstrapping*, az indítás, azaz a „kezdő lépések" témakör foglalkozik azokkal a kérdésekkel, hogy

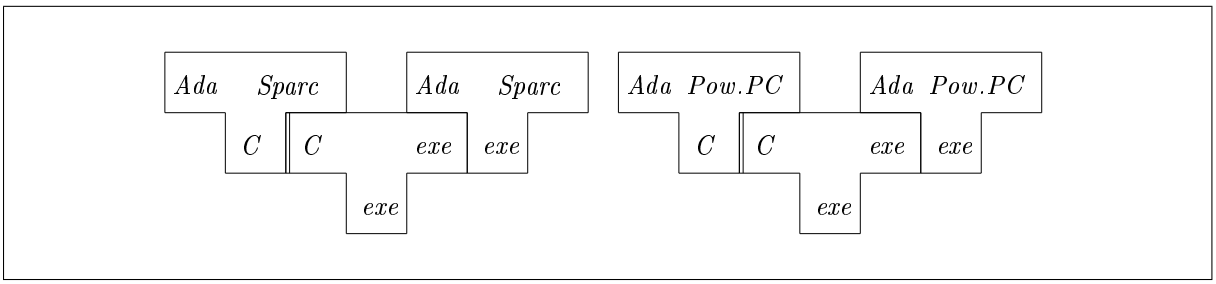
1. hogyan lehet az egyik gépen már működő fordítóprogramot a lehető legkönnyebben átírni úgy, hogy az egy másik gép kódjára fordítson,
2. hogyan fordították le az első magasszintű nyelven megírt fordítóprogramot akkor, amikor még nem volt erre a nyelvre fordítóprogram,
3. hogyan lehet egy olyan fordítóprogramot előállítani, amelyet a fordítóprogram forrásnyelvén írtak meg,
4. tudja-e egy optimalizált kódot előállító fordítóprogram optimalizálni saját végrehajtási kódját.

Ezek a problémák már az 1950-es években felmerültek.

Először vizsgáljuk meg az 1. pontban felvetett problémát, azaz a kereszt-fordítóprogramok előállítását. Például olyan *Ada* fordítóprogramokat szeretnénk készíteni, amelyek a *Sparc* és *PowerPC* gépekre készítenek tárgyprogramot. Tegyük fel, hogy van egy magasszintű nyelvet, például C-t fordító $C \rightarrow exe$ fordítóprogramunk. Ekkor a kereszt-fordítóprogramokat elég ezen a C nyelven megírni, hiszen a rendelkezésre álló C fordítóprogrammal a kívánt programokat elő tudjuk állítani.

A 2. és a 3. pontban leírt probléma nem független egymástól, hiszen az első magasszintű nyelven írt fordítóprogramnak nyilván ugyanazt a magasszintű nyelvet kellett fordítania, amelyen a fordítóprogramot megírták. Tehát elég azzal a problémával foglalkoznunk, hogy hogyan lehet egy P nyelven megírt $P \rightarrow exe$ fordítóprogramot készíteni.

A módszer a következő.



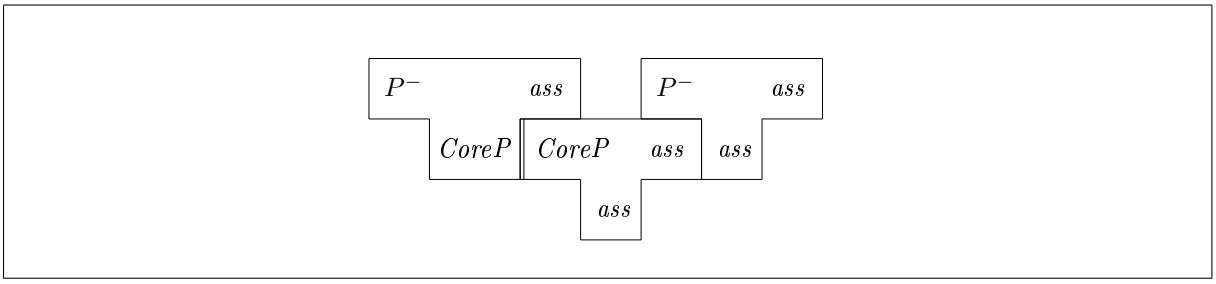
5.. ábra. Az $Ada \rightarrow Sparc$ és az $Ada \rightarrow PowerPC$ fordítóprogram

- Először írjunk egy assemblert, csak azért, hogy ne gépi kódban kelljen programozni, az assembly nyelven írt programokat ezzel az assemblerrel *exe* végrehajtható kódú programra le tudjuk fordítani.
- Ezután a P nyelvet egyszerűsítjük úgy, hogy elhagyunk belőle néhány bonyolultabb dolgot, például néhány bonyolult deklarációt, típust, utasítást. Így megkapjuk a P^- nyelvet. Ezután ezt a nyelvet is tovább (akár több lépésben is) egyszerűsítjük, úgy, hogy végül azért még egy, ha nehezen is, de használható *CoreP* nyelvet kapjunk.
- Most következik egy nehéz programozási feladat, és a fordítóprogram létrehozásában ez az egyedüli nehéz munka, meg kell írni assembly nyelven a $CoreP \rightarrow ass$ fordítóprogramot.
- Ezután már használhatjuk implementációs nyelvként a *CoreP* nyelvet, ezen a nyelven már könnyebben megírhatjuk a $P^- \rightarrow ass$ fordítóprogramot, amit az előző lépésben már megírt fordítóprogrammal le tudunk fordítani.
- Egyre bővülő, nagyobb hatásfokú programnyelvet, most már a P^--t használjuk implementációs nyelvként, most ezen a nyelven kell megírnunk a $P \rightarrow ass$ fordítóprogramot, amiből könnyen előállíthatjuk az assembly nyelvű $P \rightarrow ass$ programot

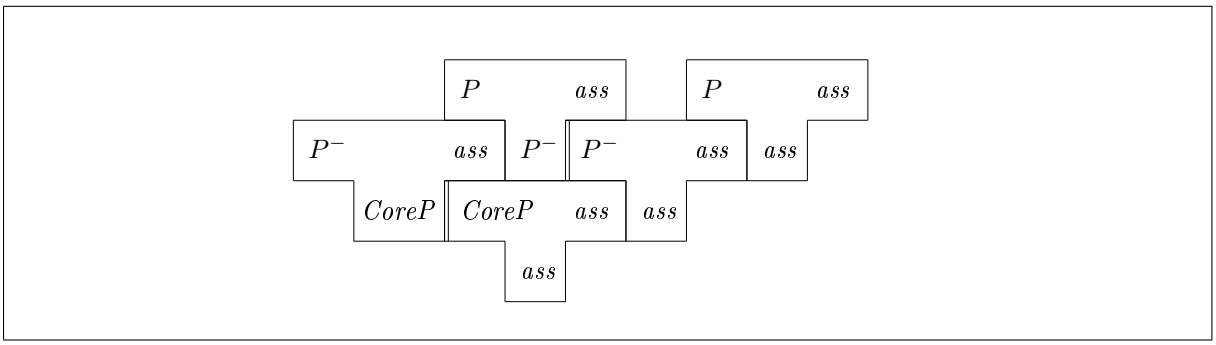
- Mivel a P nyelv jobb, mint a P^- nyelv, valószínű, hogy a P nyelven jobb fordítóprogramot tudunk írni, mint a P^- nyelven. Például, ez a fordítóprogram már optimalizálja a generált *ass* tárgyprogramot. Az optimalizálást jelöljük $\rightsquigarrow$ nyíllal, az optimalizált tárgykódot *ass**-gal, ezt a P nyelven implementált $P \rightsquigarrow$ *ass** fordítóprogramot az előbbi $P \rightarrow$ *ass* fordítóprogrammal már le tudjuk fordítani.

Nézzük meg a 4. pontban felvetett problémát, ha már van egy optimalizált kódot előállító fordítóprogramunk, hogyan tudjuk magának a fordítóprogramnak a kódját optimalizálni. Az implementáció kódjának az optimalizálását két lépésben tudjuk megvalósítani.

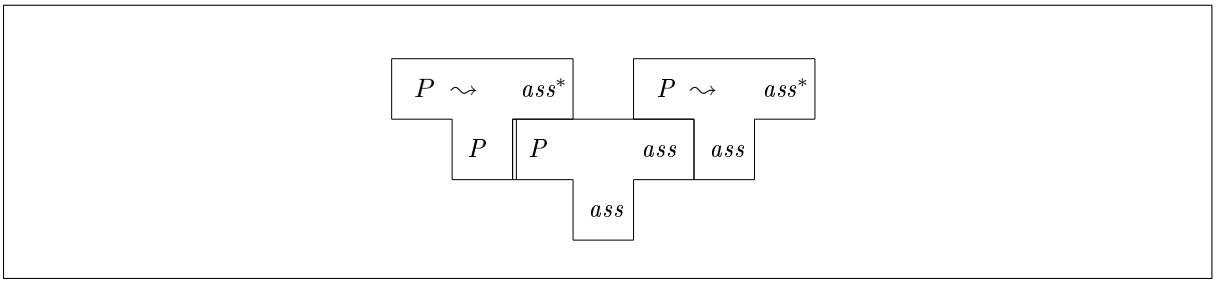
- Viszonylag nem nagy munkával készíthető egy nem feltétlenül optimális kódú, lassú fordítóprogram, ami ráadásul nagyon gyenge minőségű tárgykódot állít elő. Legyen ez az $\underline{ass}$ kódú $P \rightarrow \underline{ass}$ program. Ezzel fordítsuk le a $P \rightsquigarrow$ *ass** programot, eredményül egy optimalizált tárgykódot készítő, de lassú fordítóprogramot kapunk.
- Most ezzel a fordítóprogrammal fordítsuk az eredeti $P \rightsquigarrow$ *ass** fordítóprogramot, a fordítás eredménye egy optimalizált kódú és optimalizált tárgyprogramot előállító fordítóprogram.



6.. ábra. A $P^- \rightarrow ass$ fordítóprogram



7.. ábra. A P^- nyelven implementált $P \to ass$ fordítóprogram



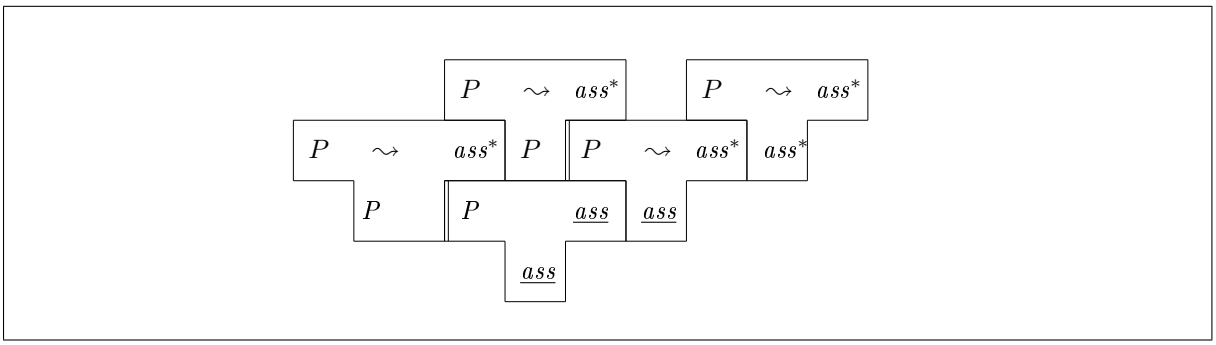
8.. ábra. A $P \rightsquigarrow ass^*$ fordítóprogram

A lexikális elemzés

A forrásnyelvű programból a *source-handler* egy karaktersorozatot készít. A lexikális elemző alapvető feladata az, hogy ebben a karaktersorozatban felismerje az összetartozó, a *szimbólumokat* alkotó karaktersorozatokat. A lexikális elemző a karaktersorozatot egy szimbólumsorozatra alakítja át.

Különböző programnyelvekben az azonos, vagy hasonló fogalmakat jelentő szimbolikus egységek lényegesen különbözhetnek, például van, ahol a *program*, és van, ahol a *main* szó jelenti a főprogramot, az egyik nyelvben a *case*, a másikban a *switch* a többirányú elágazás kulcsszava.

Különböző szimbólumhoz azonos karakterek vagy karaktersoroza-



9.. ábra. Az optimalizált kódú $P \rightsquigarrow ass^*$ fordítóprogram

tok is tartozhatnak, például a – karakter a legtöbb nyelvben a kivonás jele, de van olyan nyelv is, ahol ez a hosszú azonosító szimbólumok tagolására szolgál, vagy például a ; karakter az egyik nyelvben az utasítás végét, míg egy másik nyelvben két egymás-utáni utasítás elkülönítését jelzi.

A lexikális elemző működése

A lexikális elemző működésének alapelve az, hogy egy szimbólumot mindig a lehető leghosszabb karaktersorozatból kell felépíteni, például a *cicamica* szöveg nem nyolc egybetűs, hanem egy nyolcbetűs szimbólum lesz.

A szimbolikus egységek precíz megadása reguláris nyelvtannal vagy más néven *Chomsky 3-as típusú nyelvtannal*, reguláris kifejezésekkel vagy determinisztikus véges automatával írható le.

A lexikális elemző bemenetében benne van az összes szóköz és tabulátor jel, hiszen a source-handlerről csak annyit tételeztünk fel, hogy a kocsivissza és soremelés karaktereket hagyja el. A legtöbb programozási nyelv tetszőlegesen sok szóköz és tabulátor karaktert megenged az egyes szimbólumok között. Ezeknek a karaktereknek a fordítás szempontjából a szimbólumok felismerése után már

nincs szerepük, ezért ezeket fehér szóközöknek nevezzük. A fehér szóközöket a

$(\textit{space} \mid \textit{tab})^*$ $\qquad$ $\Big($ $(\textit{space} + \textit{tab})^*$ helyett$\Big)$

reguláris kifejezéssel adhatjuk meg, ahol a *space* a szóköz, a *tab* a tabulátor karaktert jelenti. A fehér szóközökkel a felismerés után a lexikális elemzőnek semmi teendője nincs, ezt a szimbólumot nem kell továbbadnia a szintaktikai elemzőnek.

A lexikális elemzők létrehozásának módszere az, hogy megadjuk a szimbolikus egységek leírását, például a reguláris kifejezések nyelvén, megkonstruáljuk az ekvivalens determinisztikus véges automatát, és elkészítjük a determinisztikus véges automata implementációját. Az elemző a szimbólumot az automata végállapotaival ismeri fel.

A lexikális elemző egy szimbólumhoz egy előre megadott kódot rendel, és ez a kód kerül bele a lexikális elemző outputjába. Egy szimbólumhoz kiegészítő információ is tartozhat, ilyen például a *konstans* szimbólum *típusa*, *értéke*. Ezekre az információkra a következő elemző programnak, a szintaktikus elemzőnek nincs szüksége, de például a szemantikus elemző már a típusokat ellenőrzi, a kódgeneráláshoz pedig ezek az információk feltétlenül szükségesek. Tehát a lexikális elemzőnek a szimbólumokhoz tartozó kiegészítő információkat tárolnia kell, erre a feladatra a lexikális elemzők általában egy *szimbólumtáblát* használnak, és a szimbólum kódja után egy pointert helyeznek el, ami a szimbólumhoz tartozó szimbólumtábla bejegyzésre mutat.

Példa. Tegyük fel, hogy a szimbólumok kódjai a következők:

azonosító	1
konstans	200
if	50
then	51
else	52
$=$	800
$:=$	700
;	999

0001	***cica12***
0002	***12***
0003	***alma***
0004	***55***
0005	***c1***
0006	***99***
0007	
0008	

Ekkor a lexikális elemző az

if cica12 = 12 then alma := 55 else c1 := 99;

programsorból az

50 1-0001 800 200-0002 51 1-0003 700 200-0004 52 1-0005 700 200-0006 999

sorozatot készíti, ahol 0001 $, \ldots,$ 0006 a szimbólumtábla bejegyzésekre mutató pointerek.

A lexikális elemző kimenetét, a szimbólumsorozatot arra is használhatjuk, hogy a kódokból visszaállítsuk az eredeti forrásnyelvű programot, rendezett, szépen tabulált, tagolt formában.

A következőkben néhány példát adunk reguláris kifejezésekre. Megjegyezzük, hogy a szimbólumok leírására azért használunk reguláris kifejezéseket, mert ezekkel a szimbólumok egyszerűbben és könnyebben írhatók le, mint a reguláris nyelvtanokkal.

Példa. Vezessük be a következő jelöléseket: jelöljön D egy tetszőleges számjegyet és L egy tetszőleges betűt, azaz $D \in \{0, 1, \ldots, 9\}$, és

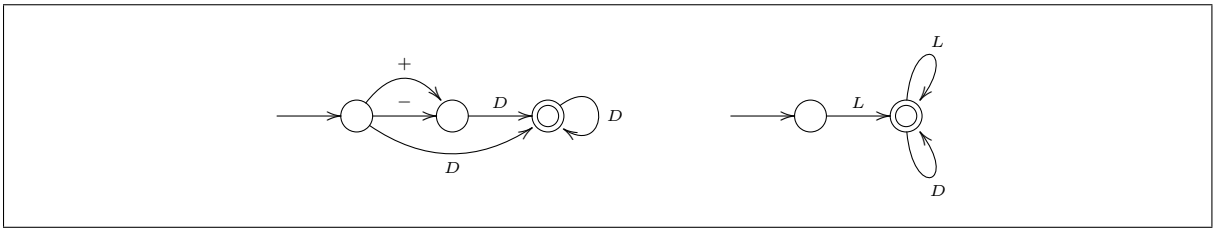
$L \in \{a, b, \ldots, z, A, B, \ldots, Z\}$, a nem látható karaktereket jelöljük a rövid nevükkel (például *space*, *Eol*, *Eof*), és legyen ε az üres

karaktersorozat jele. ***Not***(a) **jelentsen egy** a**-tól,** ***Not***$(a|b)$ **egy** a**-tól és** b**-től különböző karaktert. A reguláris kifejezések:**

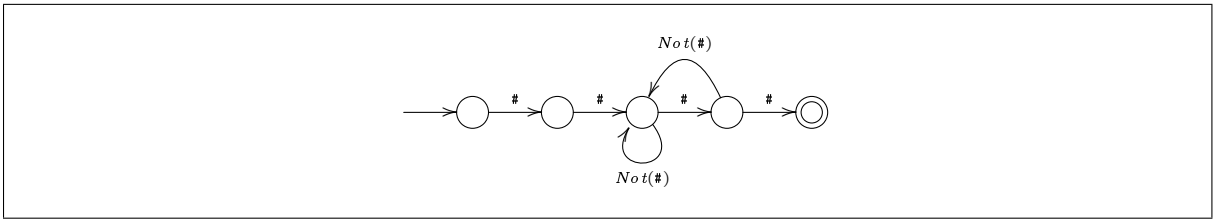
1. **egész szám:**
$(+ \mid - \mid \varepsilon)D^+$

2. **egyszerű azonosító szimbólum:**
$L(L \mid D)^*$

3. $\#\#$ **karakterpárokkal határolt megjegyzés:**
$\#\#((\# \mid \varepsilon)$***Not***$(\#))^*\#\#$



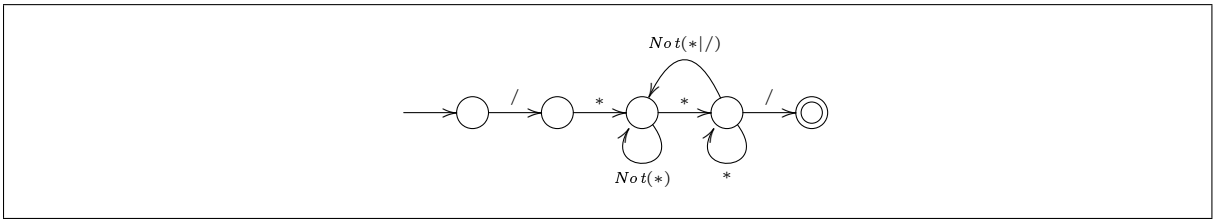
10.. ábra. 1. és 2. automatája



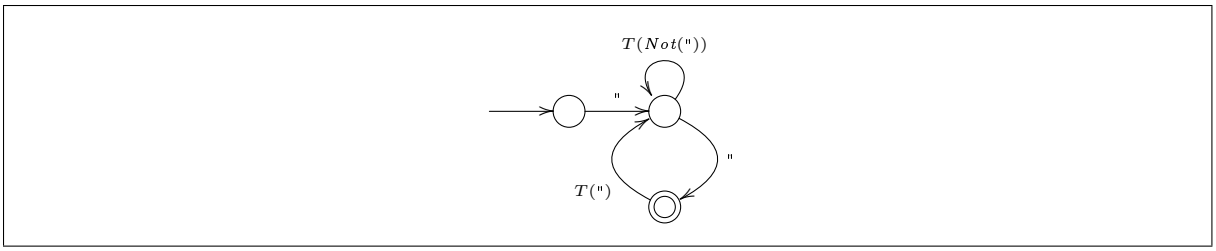
11.. ábra. 3. kifejezés felismerő automatája

4. $/*$ és $*/$ karakterpárokkal határolt megjegyzés:
$/*(\textbf{\textit{Not}}(*))^* *^+ (\textbf{\textit{Not}}(*|/)\textbf{\textit{Not}}(*)^**^+)^*/$

5. karakterfüzér:
$"(\textbf{\textit{Not}}(") \mid " ")^* "$.



12.. ábra. 4. kifejezés felismerő automatája



13.. ábra. 5. kifejezés felismerő automatája

A lexikális elemző feladata a szimbólum szövegének a meghatározása, azonban például a fenti 5. reguláris kifejezésben nem minden karakter tartozik a szimbólumhoz, a kezdő és a befejező " karakterek nem elemei a szimbólumnak. Ezért a lexikális elemzőhöz rendeljünk hozzá egy puffert, egy szimbólum felismerése után a szimbólumot alkotó karakterek ebben a pufferben lesznek. A determinisztikus véges automatát pedig egészítsük ki egy T átviteli függvénnyel, ahol $T(a)$ jelentse azt, hogy az a karakter a pufferbe kerül.

Példa. A 3. és 4. reguláris kifejezések automatái, mivel egy megjegyzést ismernek fel, egyáltalán nem tartalmaznak T függvényt.

Az 5. kifejezés automatája például az *"Ez egy ""idézet"""* szövegből az *Ez egy "idézet"* szöveget viszi a pufferbe.

Most adjuk meg egy determinisztikus véges automatával megadott lexikális elemzés algoritmusát (az egyszerűség kedvéért az egyelemű állapot halmazokat a halmazok egyetlen elemével jelöljük).

Az elemzést végző determinisztikus véges automata Σ ábécéjét egészítsük ki egy új jellel, jelöljük *egyéb*-bel a nem Σ-beli karaktereket. Ennek megfelelően a δ átmenetfüggvény a következőképpen módosul:

$$\delta'(q,a) = \begin{cases} \delta(q,a), & \text{ha } a \neq \textbf{\textit{egyéb}} \\ \emptyset, & \text{egyébként} \end{cases} .$$

Az így kapott A' kiegészített automatával az elemzés a következő lesz:

LEX-ELEMEZ$(x\#, A')$

```
 1  q ← q_0, a ← x első karaktere
 2  s' ← elemez
 3  while a ≠ # és s' = elemez
 4        do if δ'(q,a) ≠ ∅
 5              then q ← δ'(q,a)
 6                   a ← x következő karaktere
 7              else s' ← hiba
 8  if s' = elemez és q ∈ F
 9     then s' ← O.K.
10     else s' ← HIBA
11  return s', a
```

Az algoritmus bemenő paramétere a # jellel lezárt elemezendő karaktersorozat és az elemző automata. Az 1. sorban az elemző állapotát az automata q_0 állapotára állítjuk, és meghatározzuk az elemezendő szöveg első karakterét. Az s' változó az elemző működését jelzi, a 2. sorban a változóba az *elemez* szöveget töltjük. Az 5. sorban az automata állapotátmeneteit hajtjuk végre, és látható, hogy az eredeti automata fenti kiegészítésére azért volt szükség, hogy az automata működése az *egyéb* hibás karakterre is befejeződjék. A 8–10. sorokban *O.K.* azt jelenti, hogy az elemzett karaktersorozat helyes, *HIBA* a lexikális hiba megjelenését jelzi. Sikeres elemzés esetén a a # karaktert, hiba esetén éppen a hibás karaktert tartalmazza.

Megjegyezzük, hogy a LEX-ELEMEZ algoritmus csak egy szimbólumot ismer fel, és ezután működése befejeződik. Egy program nyilvánvalóan sok szimbólumból áll, és egy szimbólum meghatározása után a lexikális elemzést a következő szimbólum meghatározásával kell folytatni, azaz az elemzést az automata kezdőállapotával újra kell indítani.

Speciális problémák

Kulcsszók, standard szavak

Minden programozási nyelvben vannak olyan azonosítók, amelyeknek speciális célra fenntartott nevük, előre megadott jelentésük van, ezek a kulcsszók. A kulcsszók eredeti jelentésüktől eltérően nem használhatók.

Vannak azonban olyan azonosítók is, amelyekre szintén fennáll, hogy előre megadott jelentésük van, de ez a jelentés a programban megváltoztatható. Ezeket a szavakat standard szavaknak nevezzük.

A kulcsszók és a standard szavak száma programnyelvenként változik. A COBOL nyelvben több mint 300 ilyen szó van, a nulla érték jelölésére például három kulcsszó is van: *zero*, *zeros* és *zeroes*.

Foglalkozzunk azzal a problémával, hogy a lexikális elemző hogyan ismeri fel a kulcsszókat és a standard szavakat, és hogy hogyan különbözteti meg őket a felhasználó által használt azonosítóktól.

A kulcsszók és standard szavak felismerése akkor nagyon egyszerű, ha azokat speciális karakterekkel írják, vagy speciális elő- és utókarakterekkel jelölik meg.

A standard szavaknak az eredeti értelemtől eltérő felhasználása azonban nemcsak a lexikális elemző feladatát nehezíti meg, hanem a program olvashatóságát is erősen rontja, mint például a következő utasításban:

if if then else = then;

vagy, hogyha egy *begin* és egy *end* nevű eljárást deklarálunk:

```
begin
  begin; begin end; end; begin end;
end;
```

A kulcsszók kezelésére két módszert adunk.

1. Minden kulcsszót egy reguláris kifejezéssel írunk le, és megadjuk a reguláris kifejezéshez tartozó automata implementációját. Ennek a módszernek a hátránya az, hogy még akkor is nagyon nagyméretű programot kapunk, ha a kulcsszók leírását az azonos kezdőbetűk szerint összevonjuk.
2. A kulcsszókat egy külön táblázatban tároljuk. A karaktersorozatban a szavakat egy általános azonosító-felismerővel határozzuk meg, majd egy kereső algoritmust alkalmazva megnézzük, hogy az azonosító benne van-e a táblázatban. Ha igen, akkor a szimbólum egy kulcsszó, ellenkező esetben egy, a felhasználó által megadott azonosító. Ez a módszer nagyon egyszerű, de a keresés sebessége függ a táblázat felépítésétől, a keresési algoritmustól. Egy jól megválasztott leképező függvény és egy lineárisan szétszórt altáblákból felépített kulcsszó-táblázat nagyon hatékony lehet.

Ha a programnyelv lehetővé teszi standard szavak használatát, akkor a lexikális elemző a kulcsszókra alkalmazott módszerrel meghatározhatja, hogy a vizsgált szimbólum standard szó-e. Az, hogy a standard szó az eredeti jelentésben használt szimbólum, vagy hogy a szimbólumot a felhasználó újraértelmezte, a szimbólum környezetétől függ. Ennek eldöntése a szintaktikus elemző feladata lesz.

Az előreolvasás

Mivel a lexikális elemző a leghosszabb karaktersorozatból álló szimbólum felismerésére törekszik, a szimbólum jobb oldali végpontjá-

nak meghatározására gyakran egy vagy több karaktert is előre kell olvasnia. Erre a klasszikus példa a következő két FORTRAN utasítás:

DO 123 K = 9.92
DO 123 K = 9,92

ahol, mivel a FORTRAN nyelvben a szóköz karakterek semmilyen szerepet nem játszanak, a *9* és *92* közötti jel dönti el, hogy az utasítás egy *DO* kulcsszóval kezdődő ciklusutasítás, vagy a *DO123K* azonosítóra vonatkozó értékadás.

A reguláris kifejezések leírásában vezessük be a szimbólum jobb oldali végpontjának a jelölésére a / jelet, és nevezzük ezt *előreolvasási operátornak*. Így a fenti *DO* kulcsszó értelmezése a következő:

$DO\ /\ (betű \mid számjegy)^* = (betű \mid számjegy)^*$,

Ez az értelmezés tehát azt jelenti, hogy a lexikális elemző csak akkor tudja eldönteni, hogy az első két karakteren a *D* és az *O* betű a *DO* kulcsszó, ha előreolvasva, betűk vagy számjegyek, egy egyenlőségjel és ismét betűk vagy számjegyek után egy „ , ” karaktert talál. Az előreolvasási operátor azt is jelenti, hogy a következő szimbólum keresését a *DO* utáni karakterrel kell kezdeni. Megjegyezzük, hogy az elemző ezzel az előreolvasással a *DO* szimbólumot azonosítja akkor is, ha a *DO* után programhiba van, mint például a *DO2A=3B,* karaktersorozatban, de helyes értékadó utasításban sohasem fogja az azonosító első két *D* és *O* karakterét a *DO* kulcsszónak értelmezni.

A fenti példánál bonyolultabb esetek is előfordulhatnak, mint például a következő programsorban:

DO 123 I = (P12(3,"),)")),5

A lexikális elemzőnek fel kell ismernie, hogy az egyenlőség jel után a két paraméteres *P12* függvény *P12(3,"),)")* értéke van, az első paraméter *3*, a második egy idézőjelek közé tett karaktersorozat. Ezután következik egy vessző, és ez a vessző fogja azt jelenteni, hogy a *DO* egy kulcsszó, a függvényérték a ciklusváltozó kezdőértéke, és *5* a végérték.

Az előreolvasás karakterszámát a programozási nyelv leírásából lehet meghatározni. A modern programozási nyelveknél egy szimbólum felismeréséhez a szimbólum karakterei után egy, pesszimális esetben négy karakter előreolvasása szükséges.

Példa.

A C++ nyelv néhány kétkarakteres szimbóluma: $++$, $+=$, --, $-=$, $->$, $>=$, $>>$,
háromkarakteres szimbólumai: $<<=$, $>>=$.
A Java nyelv háromkarakteres szimbólumai: $>>=$, $>>>$, $<<=$,
és egy négykarakteres szimbólum: $>>>=$.

Direktívák

A forrásnyelvekben a direktívák a fordítóprogram működésének vezérlésére szolgálnak. A direktívákat és a direktívák operandusaiban szereplő szimbólumokat is a lexikális elemzőnek kell meghatároznia, de ezekkel az elemzőnek további teendői is vannak.

Ha a direktíva például egy feltételes fordítás *if* direktívája, akkor fel kell ismernie a direktíva összes szimbólumát, majd ki kell értékelnie az elágazás feltételét. Ha ez *false*, akkor a további sorokban szereplő szimbólumokat nem szabad elemeznie egészen addig, amíg egy *else*, vagy az *if* végét jelentő *endif* direktívát nem talál. Ez azt jelenti, hogy a lexikális elemzőnek már szintaktikus és szemantikus

ellenőrzéseket is kell végeznie, és kódjellegű információt kell előállítania. A feladatot különösen bonyolíthatja, ha a nyelv lehetőséget ad a feltételek skatulyázására.

Egy másik tipikus direktíva a makróhelyettesítés, vagy egy adott nevű állomány bemásolása a forrásnyelvű szövegbe, aminek a végrehajtása szintén távol áll a lexikális elemző eredeti alapfeladatától.

A legtöbb fordítóprogramban ezeket a problémákat úgy oldják meg, hogy a szintaktikus elemző előtt egy előfeldolgozó programot működtetnek, amelynek a feladata a direktívák által megadott feladatok végrehajtása.

Hibakezelés

Ha a lexikális elemző egy karaktersorozatnak nem tud egy szimbólumot sem megfeleltetni, akkor azt mondjuk, hogy a karaktersorozatban lexikális hiba van.

A lexikális hibákat legtöbbször az okozza, hogy illegális karakterek kerülnek a szövegbe, karakterek felcserélődnek vagy esetleg karakterek hiányoznak. Mivel nem célszerű egy lexikális hiba detektálásakor a fordítást megszakítani, valamilyen hibaelfedő algoritmust kell alkalmazni, amelynek az a célja, hogy az elemzés a lehető legkevesebb karakter kihagyásával folytatódjon.

Tegyük fel, hogy az elemző a defiξne karaktersorozatot vizsgálja, ahol a ξ egy nem megengedett karakter. Ekkor a ξ karakterre egy hibajelzést ad, és az egyik lehetőség az, hogy nem foglalkozik a már beolvasott karakterekkel, az elemzést a következő, még nem vizsgált karakterrel folytatja. Az elemző így az ne karakterekből például egy logikai reláció szimbólumot ismer fel. Egy másik lehetőség az, hogy egyszerűen kihagyja, vagy egy tetszőleges karakterrel, általában a space szóköz karakterrel helyettesíti az illegális

karaktert. Így a fenti szövegben vagy a define szimbólumot, vagy a defi és ne szimbólumokat határozza meg.

Hiányzó karaktereknek gyakran az a hatásuk, hogy a lexikális elemző nem tudja a szimbólumokat elkülöníteni egymástól. Például, ha az alma+112 karaktersorozatból a + jel kimarad, akkor az elemző minden hibajelzés nélkül az alma112 azonosítót ismeri fel. Ha a 112+alma-ból hiányzik a + jel, akkor a lexikális elemző ezt egy 112 konstansra és egy alma azonosítóra bontja, és azt, hogy közülük egy műveleti jel hiányzik, majd csak a szintaktikus elemző fogja felfedezni.

Hasonlóan, a következő C++ programsor is meglepő eredményt ad, a "sárga" és "narancs" szövegek közül kimaradt a vessző:

```
char *szín[] = { "piros" , "kék", "sárga" "narancs" } ;
```

a harmadik elem a sárganarancs lesz.

Vannak olyan lexikális hibák is, amelyeknek a kezelésére különösen nagy gondot kell fordítani. Ilyenek a karakterfüzérek és a megjegyzések befejező karaktereinek hiányai, ezek a hibák gyakran azt okozzák, hogy a program további része, egészen a program végéig, karakterfüzér vagy megjegyzés lesz.