Mesterséges intelligencia 2.

gyakorlat

1. feladat

(formula szintaktikai ellenőrzése)

Készítette:

Szathmáry László
III. PTM.

1. feladat:

Egy olyan program megírása, mely egy adott elsőrendű matematikai logikai nyelvben megadott formuláról eldönti, hogy szintaktikailag helyes-e; továbbá tároljuk le a formulát alkalmas módon (formulafa).

Elsőrendű matematikai logikai nyelv fogalma:

Négy halmazból álló rendszerrel adjuk meg: Ω = <Srt, Const, Fv, Pred>

I. Típus (Srt)

Egy *nem üres* halmaz, melynek elemeit típusoknak nevezzük. Az adott világ azonos tulajdonsággal rendelkező objektumait egy típusba sorolhatjuk. Adott típusú objektumok ábrázolására szolgáló jelek a változók.

II. Konstans (Const)

A nyelv kitüntetett objektumai, melyek típussal rendelkeznek. (Ez a halmaz *lehet üres* is).

III. Függvényszimbólumok (Fv)

(*Lehet üres is* ez a halmaz). Az adott világ objektumai között végrehajtott művelek(et) jelöli, ami egy újabb objektumot eredményez.
A szimbólum alakja: ez a művelet hány és milyen típusú objektumon végezhető el, ill. az eredmény milyen típusú lesz:

$f \in Fv,\ f(\pi_1, \dots, \pi_n) \rightarrow \pi_{n+1}$,ahol n>0 és $(\pi_1, \dots, \pi_{n+1})$ a nyelv típusai

IV. Predikátumszimbólumok (Pred)

Nem üres halmaz. Az objektumok közötti viszonyok (relációk) kifejezésére szolgál. Ennek is lesz alakja, vagyis: hány és milyen típusú objektumok közötti viszonyt fejez ki.

$p \in Pred,\ p(\pi_1, \dots, \pi_n) \rightarrow$ logikai igaz v. logikai hamis

szintaktikusan helyes jelsorozatok:

term: változó, konstans
bonyolult term:

$f \in Fv,\ f(\pi_1, \dots, \pi_n) \rightarrow \pi_{n+1}$

ahol $\pi_1, \dots, \pi_n$ megfelelő típusú termek

Minden term ezen lépések véges sok alkalmazásával készül.

atomi formula:

$p \in Pred,\ p(\pi_1, \dots, \pi_n)$

ahol $\pi_1, \dots, \pi_n$ megfelelő típusú termek

bonyolult formula: atomi formulákból logikai összekötő jelek segítségével állítjuk elő. Összetettebb állítások kifejezésére szolgál.

A feladat megoldásának terve:

- elsőrendű matematikai logikai nyelv megadása:
  - típusok és a hozzátartozó változó és konstans szimbólumok felvétele
  - függvény- és predikátumszimbólumok felvétele
  (ezeknél a *prefix* alakot alkalmaztam)
  Az ellenőrző rész az operandusok megadott sorában kéri vissza a formulában az operandus helyén megadott term típusát.
- formula ellenőrzése
  ez két lépésben történik:
  - A program *teljesen bezárójelezett* formulát kér be, s ellenőrzi, hogy ennek a kritérium-nak megfelel-e a beírt formula, továbbá hogy formailag helyes-e (pl.: függvény operanduslistájában van-e vessző az operandusok között).
  - Függvények és predikátumok ellenőrzése: az operandusok száma és típusa rendben van-e

Megvalósítás:

```
#include <iostream.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>
#define  MAX_SORHOSSZ       200    //file-ból olvasáskor egy sor max. hossza
#define  FORM_HOSSZ         100    //a beolvasandó formula max. hossza
#define  OPERAND_SZAMA      20     //max. hány operandusa lehet
#define  TIPUS         1
#define  VALTOZO       2
#define  KONSTANS      3
#define  OPERANDUS     4
#define  ERTEK         5
#define  LISTA         6                 //listát készítsen-e belőle
```

Itt adjuk meg a szükséges "include" állományokat, ill. készítünk néhány konstanst is.

A programot úgy készítettem el, hogy lehetősége legyen a felhasználónak egy *saját* nyelv megadására. Ennek kezelésére szükség lesz néhány osztályra, ill. rekordstruktúrákra:

```
struct faelem {
  char *nev;
  struct faelem *bal, *jobb;       //mivel fa, két leágazása lehet max.
  struct faelem *le;               //fv.-ek és pred.-ok esetén az operanduslistára mutat
  struct faelem *kov;              //az operanduslisták elemei így vannak öszekötve
};
```

A feladat második része az volt, hogy a formulát tároljuk le valamilyen alkalmas módon a tárban. Én a formulafát alkalmaztam, ahol a fa egyes csomópontjaiban ilyen rekordok áll-nak.

```
struct listaelem {
  char *nev;
  struct listaelem *kov;
};
```

Ez egy egyszerű láncolt listaelem. Később lesz rá szükség.

```
struct tipuselem {
  char *nev;                  //a típus neve
  struct listaelem *val;      //az ehhez a típushoz tartozó változók listája
  struct listaelem *kons;     //az ehhez a típushoz tartozó konstansok listája
  struct tipuselem *kov;      //ha több típus is van, akkor összekötjük őket
};

class Tipus {
  public:
    Tipus() : fej(NULL) {};
    int eleme(char *mit);
    struct tipuselem * berak(int mit);
    void berak(struct tipuselem *hova, int mit, int MIEZ);
    struct tipuselem * getFej() { return fej; }
    char miez(char *mit);
    char getTipusa(char *mit);
  private:
    struct tipuselem *fej;
    void hiba() { cout<<"Mem.-foglalasi hiba (Tipus osztaly).\n"; exit(-1); }
};
```

A típusokról, változókról, konstansokról való információinkat egy ilyen objektum segítségével fogjuk megvalósítani. Milyen műveleteket tud ez elvégezni?

Meg tudja állapítani, hogy egy szimbólum (ami lehet egy típusnév, egy változó vagy egy konstans neve) szerepel-e benne. Ez alapján visszaad egy logikai értéket (C-ben, C++ -ban ezt az integrális helyettesíti):

```
int Tipus::eleme(char *mit)
{
   if (!fej) return 0;
   struct tipuselem *akt = fej;
   struct listaelem *valtozo  = akt->val, *konstans = akt->kons;
   while (akt) {
      if (strcmp(akt->nev, mit)==0) return 1;
      while (valtozo)  {
            if (strcmp(valtozo->nev, mit)==0) return 1;
            valtozo = valtozo->kov; }            //end of while(valtozo)
      while (konstans) {
            if (strcmp(konstans->nev, mit)==0) return 1;
            konstans = konstans->kov; }          //end of while(konstans)
      akt=akt->kov;
   } //end of while(akt)
   return 0;
}
```

Új típus esetén azt fel kell tudni venni a listába: mivel ez még új, így nem is lehetnek ilyen típusú változók, ill. konstansok, így azokat a mutatókat NULL-ra kell állítani:

```
struct tipuselem * Tipus::berak(int mit)
{
  struct tipuselem *uj = (struct tipuselem *) malloc (sizeof (struct tipuselem));
  if (!uj) hiba();
  char str[2] = {mit, '\0'};
  uj->nev = strdup(str); if (!(uj->nev)) hiba();
  uj->val = NULL; uj->kons = NULL; uj->kov = fej; fej=uj;
  return uj;
}
```

Az előző fv. vége: **return uj;** , amely az új típuselemre mutat. Az ezt hívó fv. megjegyzi ezt a címet, s amikor ehhez a típushoz változókat, ill. konstansokat adunk meg, akkor biztos, hogy ezzel a típuselemmel kell dolgozni, s mivel ennek már tudjuk is a címét, át is tudjuk adni. Plusz információ meg: változót vagy konstanst akarunk felfűzni (**MIEZ**), ill. maga a szimbólum (**mit**)

```
void Tipus::berak(struct tipuselem *hova, int mit, int MIEZ)
{
  struct listaelem *uj = (struct listaelem *) malloc (sizeof (struct listaelem));
  if (!uj) hiba();
  char str[2] = {mit, '\0'};
  uj->nev = strdup(str); if (!(uj->nev)) hiba();
  switch (MIEZ)
  {
    case VALTOZO  : uj->kov=hova->val;  hova->val=uj; break;
    case KONSTANS : uj->kov=hova->kons; hova->kons=uj; break;
  }
}
```

Tehát ezzel a fv.-nyel tudunk egy már meglévő típushoz változókat, ill. konstansokat hozzáadni.

A Tpus osztályban tehát a típusok vannak összeláncolva. A legelső elem (fej) címét le tudjuk kérdezni a **getFej()** fv.-nyel, amelyet rövidsége miatt (**return fej;**) inline módon adtam meg.

Amikor majd beolvassuk a formulát, akkor az egyes karakterekről (szimbólumokról) jó lenne eldönteni, hogy mik is valójában: változók, konstansok, fv.-szimbólumok, predikátumszimbólumok, stb. ? A **miez()** nevezetű fv. ezt végzi el: kap egy szimbólumot, s végigszalad a listán. Minden típuselemben van 2 másik lista is: egy a változószimbólumokról, egy pedig a konstansszimbólumokról. Ha valahol megtalálja a keresett szimbólumot, akkor '**v**'-vel vagy '**c**'-vel tér vissza, aszerint hogy ez változó-e vagy konstans.

Ha nincs találat: '**h**', azaz hiba:

```
char Tipus::miez(char *mit)
{
  if (!fej) return 'h';                              //nincs benne
  struct tipuselem *akt = fej;
  struct listaelem *valtozo, *konstans;
  char str[2] = {*mit, '\0'};
  while (akt)
  {
    valtozo  = akt->val;
    konstans = akt->kons;
    while (valtozo)
    {
        if (strcmp(valtozo->nev, str)==0) return 'v';    //ez egy valtozo neve
        valtozo = valtozo->kov;
    }
    while (konstans)
    {
        if (strcmp(konstans->nev, str)==0) return 'c';   //ez egy konstans
        konstans = konstans->kov;
    }
    akt=akt->kov;
  }
  return 'h';                                        //ha nincs sehol
}
```

Később, amikor a függvények és predikátumok operanduslistáját ellenőrizzük aszerint, hogy egyeznek-e a típusok, akkor ha pl. találunk egy változót vagy konstanst (**miez()** fv. itt segít majd), akkor meg kellene tudni állapítani annak a típusát:

```
char Tipus::getTipusa(char *mit)
{
  struct tipuselem *akt_tip = fej;
  struct listaelem *futo;
  while (akt_tip)
  {
    futo = akt_tip->val;
    while (futo) {
        if (strcmp(futo->nev,mit)==0) return (akt_tip->nev)[0];
        futo = futo->kov;
    }
    futo = akt_tip->kons;
    while (futo)
    {
        if (strcmp(futo->nev,mit)==0) return (akt_tip->nev)[0];
        futo = futo->kov;
    }
    akt_tip = akt_tip->kov;
  }
  return '\0';          //a fordito miatt, kulonben mar hamarabb
                        //vissza kell ternie !!!
}
```

Tehát ez a fv. kap egy konstans- vagy változónevet, s visszatér annak típusával.

/***/
A Tipus osztály ezzel kész, de még számon kell tartani a fv.-ekről és predikátumokról szóló információkat is: **Fuggveny** és **Predikatum** osztály. Ezek felépítése nagyon hasonló a Tipus osztályhoz, csupán más a láncolt lista szerkezete, s így módosítani kell a megfelelő metódusokat is
/***/

```
struct fuggvenyelem {
  char *nev;                    //fv. neve
  char *ertek;                  //visszatérési értékének típusa
  struct listaelem *op_tipusa;  //operandusainak típusa (lényeges a sorrend!!!)
  struct fuggvenyelem *kov;     //a köv. fv.-elemre mutat
};

class Fuggveny {
  public:
    Fuggveny() : fej(NULL) {};
    int eleme(char *mit);
    struct fuggvenyelem * berak(int mit);
    void berak(struct fuggvenyelem *hova, int mit, int MIEZ);
    struct fuggvenyelem * getFej() { return fej; }
    char miez(char *mit);
    void opTipusa(char *keres, char str[]);
    char Fuggveny::getTipusa(char *mit);
  private:
    struct fuggvenyelem *fej;
    void hiba() { cout<<"Mem.-foglalasi hiba (Fuggveny osztaly).\n"; exit(-1); }
};
```

Mint látható a metódusok nevei teljesen megegyeznek a Tipus osztály fv.-neveivel, csupán a hozzájuk tartozó kódban van eltérés:

```
int Fuggveny::eleme(char *mit)
{
   struct fuggvenyelem *akt = fej;
   while (akt)
   {
      if (strcmp(akt->nev, mit)==0) return 1;
      akt=akt->kov;
   }
   return 0;
}
```

Kap egy fv.-nevet (szimbólumot), s megállapítja, hogy van-e már ilyen nevű fv.-ünk.
A végén egy logikai értékkel tér vissza.

A **berak()** metódusból kettő is van, kihasználva a C++ azon lehetőségét, hogy a fv.-neveket túl lehet terhelni: az egyik egy új fv.-szimbólumot rak be, a másik pedig eme függvényelem mutatóit fogja beállítani aszerint, hogy mi ennek a fv.-nek a visszatérési típusa, ill. mik a fv. operandusainak típusai:

```
int Fuggveny::eleme(char *mit)                  //új fv.-szimbólum berakása
{
  struct fuggvenyelem *akt = fej;
  while (akt)
  {
    if (strcmp(akt->nev, mit)==0) return 1;
    akt=akt->kov;
  }
  return 0;
}

void Fuggveny::berak(struct fuggvenyelem *hova, int mit, int MIEZ)
{
  char str[2] = {mit, '\0'};
  switch (MIEZ)
  {
    case ERTEK :
            hova->ertek = strdup(str); if (!(hova->nev)) hiba();
            break;
    case OPERANDUS :
            struct listaelem *uj = (struct listaelem *) malloc (sizeof (struct listaelem));
            if (!uj) hiba();
            uj->nev = strdup(str); if (!(uj->nev)) hiba(); uj->kov = NULL;
            struct listaelem *vege = hova->op_tipusa;
            if (vege==NULL) hova->op_tipusa=uj;
            else
            {
              while (vege->kov) vege = vege->kov;
              vege->kov = uj;
            }
            break;
  } //end of switch
}
```

A köv. fv. megállapítja egy szimbólumról, hogy az fv.-szimbólum-e?
Ha igen, akkor a visszatérési érték '**f**', különben '**h**'.

```
char Fuggveny::miez(char *mit)
{
  char str[2] = {*mit, '\0'};
  struct fuggvenyelem *akt = fej;
  while (akt)
  {
    if (strcmp(akt->nev, str)==0) return 'f';
    akt=akt->kov;
  }
  return 'h';
}
```

A köv. fv. megállapítja, hogy egy fv.-nek az operandusai milyen típusúak:

```
void Fuggveny::opTipusa(char *keres, char str[])
{
  int hova=0;
  struct fuggvenyelem *akt_fv = fej;
  struct listaelem *akt_op;
  while (!( strcmp(akt_fv->nev,keres)==0  )) akt_fv = akt_fv->kov;
  akt_op = akt_fv->op_tipusa;
  while (akt_op)
  {
    str[hova++] = (akt_op->nev)[0];
    akt_op = akt_op->kov;
  }
  str[hova] = '\0';
}
```

Vagyis: kap egy fv.-szimbólumot, ill. egy karaktertömböt cím szerint (ezt itt így tudjuk módosítani). Megkeressük a fv.-szimbólumot, majd az operandusok típusait tartalmazó listát bejárjuk, s ezeket a típusokat letároljuk a karaktertömbben.
Erre majd akkor lesz szükség, amikor a függvény operandusainak számát és típusát ellenőrizzük. Megnézzük, hogy a felhasználó hány és milyen típusú operandusokat adott meg. Ezután eme fv. segítségével lekérdezzük, hogy *minek kellene* szerepelnie, s ha eltérés van, akkor hibaüzenet.

Ez a fv. pedig megállapítja, hogy mi egy fv. *visszatérési értékének típusa.*

```
char Fuggveny::getTipusa(char *mit)
{
  struct fuggvenyelem *akt_fv = fej;
  struct listaelem *futo;
  while (!( strcmp(akt_fv->nev,mit)==0 )) akt_fv = akt_fv->kov;
  return (akt_fv->ertek)[0];
}
```

/***/
A Fuggveny osztály ezzel kész, de még számon kell tartani a predikátumokról szóló információkat is: **Predikatum** osztály. Ennek a felépítése majdnem azonos a Fuggveny osztállyal, de van egy kis eltérés: predikátum nem ad vissza olyan értéket, amelynek típusa lenne.
/***/

```
struct predikatumelem {
 char *nev;
 struct listaelem *op_tipusa;
 struct predikatumelem *kov;
};
```

Látható, hogy itt hiányzik az **ertek** mutató. Egy predikátum esetén csak az operandusok típusait kell letárolni, ill. természetesen a predikátumszimbólum nevét.

A **Predikatum** osztály tehát a következőképpen néz ki:

```
class Predikatum {
  public:
    Predikatum() : fej(NULL) {};
    int eleme(char *mit);
    struct predikatumelem * berak(int mit);
    void berak(struct predikatumelem * hova, int mit);
    struct predikatumelem * getFej() { return fej; }
    char miez(char *mit);
    void opTipusa(char *keres, char str[]);
  private:
    struct predikatumelem *fej;
    void hiba() { cout<<"Mem.-foglalasi hiba (Predikatum osztaly).\n"; exit(-1); }
};
```

A metódusok leírása teljesen megegyezik a **Fuggveny** osztálynál leírtakkal, csupán a második **berak()** tér el egy kicsit: itt ui. nem kell megadni, hogy amit berakunk az a visszatérési érték, vagy egy operandus típusa. Predikatum esetén csak ez utóbbi jöhet szóba. Az egyes metódusok felsorolása:

```
int Predikatum::eleme(char *mit)
{
   struct predikatumelem *akt = fej;
   while (akt)
   {
      if (strcmp(akt->nev, mit)==0) return 1;
      akt=akt->kov;
   }
   return 0;
}

struct predikatumelem * Predikatum::berak(int mit)
{
   struct predikatumelem *uj;
   uj = (struct predikatumelem *) malloc (sizeof (struct predikatumelem));
   if (!uj) hiba();
   char str[2] = {mit, '\0'};
   uj->nev = strdup(str); if (!(uj->nev)) hiba();
   uj->op_tipusa = NULL; uj->kov = fej; fej=uj;
   return uj;
}

char Predikatum::miez(char *mit)
{
   char str[2] = {*mit, '\0'};
   struct predikatumelem *akt = fej;
   while (akt) {
      if (strcmp(akt->nev, str)==0) return 'p';
      akt=akt->kov;
   }
   return 'h';
}
```

```
void Predikatum::berak(struct predikatumelem * hova, int mit)
{
  char str[2] = {mit, '\0'};
  struct listaelem *uj = (struct listaelem *) malloc (sizeof (struct listaelem));
  if (!uj) hiba();
  uj->nev = strdup(str); if (!(uj->nev)) hiba(); uj->kov = NULL;
  struct listaelem *vege = hova->op_tipusa;
  if (vege==NULL) hova->op_tipusa = uj;
  else
  {
    while (vege->kov) vege = vege->kov;
    vege->kov = uj;
  }
}

void Predikatum::opTipusa(char *keres, char str[])
{
  int hova=0;
  struct predikatumelem *akt_pred = fej;
  struct listaelem *akt_op;
  while (!( strcmp(akt_pred->nev,keres)==0 )) akt_pred = akt_pred->kov;
  akt_op = akt_pred->op_tipusa;
  while (akt_op)
  {
    str[hova++] = (akt_op->nev)[0];
    akt_op = akt_op->kov;
  }
  str[hova] = '\0';
}

/*******************************************************************************************/
                                   A Predikatum osztály ezzel kész.
/*******************************************************************************************/
```

Most már csupán egyetlen dolgot kellene még összefogni egy osztályba: a fenntartott nyelvi elemeket, mint pl. a zárójelek, pont, vessző, kvantorok, logikai összekötő jelek:

```
struct szimbolumok {
 char *azonosito;              //info. arról, hogy az mi, pl. 'k' -> kvantor
 char *jel;                    //maga a jel mint szimbólum
 int  prec;                    //a szimbólum precedenciája (ha van neki)
 struct szimbolumok *kov;      //köv. elemre mutat
};
```

Megjegyzés:
Mivel a felhasználótól TELJESEN bezárójelezett formulát kérünk majd, ezért a precedenciát nem kell figyelni, de a későbbi fejlesztés miatt bennhagytam.

A **Lista** osztály felépítése tehát:

```
class Lista {
  public:
    Lista();
    void berak(char *miez, char *jel, int prec=0);
    char miez(char *mit);
    int getPrec(char *mit);
    int eleme(char *mit);
  private:
    struct szimbolumok *fej;
    void hiba() { printf("Mem.-foglalasi hiba (Lista osztaly).\n"); exit(-1); }
};

Lista::Lista()
{
   fej=NULL;
   berak("s","("); berak("s",")"); berak("s",".");   berak("s",",");
   berak("k","V",5); berak("k","E",5); berak("l","&",3);
   berak("l","|",3);   berak("l",">",2);   berak("l"," ¬ ",4);
}
```

Az osztály konstruktora elvégzi a feltöltést a **berak()** fv. hívásaival. Ezek a karakterek (szimbólumok) lesznek a nyelv fenntartott karakterei, s így pl. ilyen nevű típus nem lesz megadható.
Ez a fv. tárterületet allokál, s beállítja a mezőértékeket.

```
void Lista::berak(char *miez, char *jel, int prec)
{
   struct szimbolumok *uj = (struct szimbolumok *)malloc(sizeof(struct szimbolumok));
   if (!uj) hiba();
   uj->azonosito = strdup(miez); uj->jel = strdup(jel);
   uj->prec = prec; uj->kov=NULL;
   if (!fej) fej=uj;
   else uj->kov=fej, fej=uj;
}
```

A köv. fv. arról ad tájékoztatást, hogy egy adott szimbólum mi is.

```
char Lista::miez(char *mit)
{
   char str[2] = {*mit, '\0'};
   struct szimbolumok *akt = fej;
   while (akt)
   {
      if (strcmp(str,akt->jel)==0) return ((akt->azonosito)[0]);
      akt = akt->kov;
   }
   return 'h';                          //hiba, nincs itt ilyen szimbólum
}
```

Következzék itt egy összefoglaló táblázat arról, hogy mik lehetnek a **miez()** fv.-ek visszatérési értékei (azért fv.-ek, mert több osztálynak is van ilyen nevű metódusa):

Visszatérési érték	jelentése
v	változó
c	konstans
f	fv.-szimbólum
p	predikátumszimbólum
k	kvantor
l	logikai összekötő jel
s	szintaktikai elem [**( ) , .**]
h	hiba, nincs ilyen elem

```
int Lista::eleme(char *mit)                    //adott szimbólum eleme-e az osztálynak
{
  struct szimbolumok *akt = fej;
  while (akt)
  {
    if (strcmp(akt->jel, mit)==0) return 1;
    akt=akt->kov;
  }
  return 0;
}

int Lista::getPrec(char *mit)                  //adott szimbólum precedenciájának lekérdezése
{
  char str[2] = {*mit, '\0'};
  struct szimbolumok *akt = fej;
  while (akt)
  {
    if (strcmp(str,akt->jel)==0) return (akt->prec);
    akt = akt->kov;
  }
  return 0;
}

/*******************************************************************************************/
```

A Lista osztály ezzel kész.

```
/*******************************************************************************************/
```

Az eddigi osztályokat azért hoztuk létre. hogy a segítségünkre legyenek. Származtatni fogjuk őket, s a metódusaik segítségével használjuk majd a lehetőségeiket.

A program hátralevő részét egyetlen osztályba fogjuk összefogni, ez lesz a **Formula** osztály. Eme osztály felépítése: van egy nyilvános metódusa, a **vezerlo()**, ezt meghívva fog "beindulni" az egész program.
Adattagjai: a fentebb definiált 4 osztály egy-egy példánya, a beolvasott formula, s egy mutató, amely ama fa gyökerére mutat, amelyet ebből a formulából előállítottunk.

```
class Formula {
  public:
    Formula() {};
    void vezerlo();
  private:
    Tipus tipus;
    Fuggveny fuggveny;
    Predikatum predikatum;
    Lista lista;
    char *formula;                    //a formulára mint "sztringre" mutató pointer
    struct faelem *gyoker;            //a felépülő formulafa gyökerére mutat

    void nyelv_megadasa();
    int  ellenoriz(int mit);
    void hiba() { cout<<"Mem.-foglalasi hiba.\n"; exit(-1); }
    void hiba(int kod, char betu='\0');
    int beker(int ebbe, int MAX=1);
    void filebol_epit();
    void fileba_ment();
    void formula_beolvasasa();
    char miez(char *mit);
    void eloellenorzes();
    void eloallit(char *&i, char atadando[]);
    struct faelem * epit(char *str, int lista_lesz=0);
    void bejar(struct faelem *gyoker);
    void utoellenorzes(struct faelem *gyoker);
    void operandus_lista(char *fv_pred_nev, char miez, char str[]);
    void itteni_lista(struct faelem *gyoker, char str[]);
};
```

A **vezerlo()** fv. így néz ki:

```
void Formula::vezerlo()
{
  clrscr();
  nyelv_megadasa();
  formula_beolvasasa();
  eloellenorzes();
  gyoker = epit(formula);
  printf("fa felepitese OK\n");
  getch();
  utoellenorzes(gyoker);
  printf("a fv.-ek es/vagy pred.-ok operandusainak tipusa(i) OK\n");
  bejar(gyoker);
}
```

Vagyis: lehetőségünk van egy saját nyelv megadására. Ezután beolvassuk a formulát, mint "sztringet". Végrehajtunk egy előellenőrzést, mely megvizsgálja, hogy formálisan helyes-e (zárójelek rendben vannak-e pl). Ebből még lehet fát építeni akkor is, ha a fv.-ek és predikátumok operandusainál valami hiba van. Ezek felderítésére szolgál az

utoellenorzes() fv., mely már a fát járja be, s ha találkozik egy fv.- vagy pred.-szimbólummal, akkor ez már csak az operandusok számát és típusát ellenőrzi.

Nyelv megadása:

A **nyelv_megadasa()** fv. használ egy **beker()** nevezetű fv.-t, amely a billentyűzetről való beolvasásért felel:

```
int Formula::beker(int ebbe, int MAX)
{
  int  c,i=0;

  while (!( (c=getch())==13 && i>0 ))
  {
    switch (c)
    {
         case '\b'   :  if (i>0) {
                             printf("\b \b"); --i;
                           } break;
         default     :  if ( c!=13 && i<MAX ) {
                             putch(c);
                             ebbe=c;
                             ++i;
                           } break;
    }
  }
  putchar('\n');
  return ebbe;
}
```

Az itt következő fv. felelős a nyelv beolvasásáért. Rossz inputot nem fogad el; figyeli a szimbólumokat, hogy már szerepeltek-e, s ha igen, akkor nem fogadja el, stb.
Mivel a változók, típusok, stb. megadása hosszadalmas, ezért lehetőség van arra, hogy a beírt adatokat file-ba mentsük, s a prg. következő indításakor betöltsük!

```
void Formula::nyelv_megadasa()
{
  int valasz;
  printf("Adjon meg egy elsorendu mat.log. nyelvet!\n");
  do
  {
    printf("Szeretne file-bol betolteni egy nyelvet? ");
    valasz = beker(valasz); valasz = toupper(valasz);
  } while (!(valasz=='I' || valasz=='N'));
  if (valasz=='I') filebol_epit();
  else
  {
    do
    {
         int tipusnev;
         printf("Tipus neve: "); tipusnev = beker(tipusnev);
```

```
        while (ellenoriz(tipusnev)==0)  //ha mar van ilyen nevu bejegyzes
        {
          printf("Hiba! mar van ilyen nevu bejegyzes.\n");
          printf("Tipus neve: "); tipusnev = beker(tipusnev);
        }
        struct tipuselem *hova = tipus.berak(tipusnev);
        do
        {
          int valtozonev;
          printf("Valtozo neve: "); valtozonev = beker(valtozonev);
valtozonev=tolower(valtozonev);
          while (ellenoriz(valtozonev)==0)                    //kisbetűsek a változónevek
          {
            printf("Hiba! mar van ilyen nevu bejegyzes.\n");
            printf("Valtozo neve: "); valtozonev = beker(valtozonev);
valtozonev=tolower(valtozonev);
          }
          tipus.berak(hova,valtozonev, VALTOZO);
          printf("Megad egy ujabb valtozot? (i/n): "); valasz = beker(valasz);
          valasz = toupper(valasz);
        } while (!(valasz=='N'));
        printf("Megad konstansokat ehhez a tipushoz? (i/n): "); valasz = beker(valasz);
       valasz = toupper(valasz);
        if (valasz=='I')
        {
          do
          {
            int konstansnev;
            printf("Konstans neve: "); konstansnev = beker(konstansnev);
            while (ellenoriz(konstansnev)==0)
            {
                 printf("Hiba! mar van ilyen nevu bejegyzes.\n");
                 printf("Konstans neve: "); konstansnev = beker(konstansnev);
            }
            tipus.berak(hova,konstansnev, KONSTANS);
            printf("Megad egy ujabb konstanst? (i/n): "); valasz = beker(valasz);
            valasz = toupper(valasz);
          } while (!(valasz=='N'));
        }
        printf("Megad egy ujabb tipust? (i/n): "); valasz = beker(valasz);
        valasz = toupper(valasz);
    }  while (!(valasz=='N'));

    printf("Megad fuggvenyeket? (i/n): "); valasz = beker(valasz);
    valasz = toupper(valasz);
    if (valasz=='I')
    {
        do
        {
          int fuggvenynev;  // fv. neve kisbetus lehet
          printf("A fv. neve: "); fuggvenynev = beker(fuggvenynev);
```

```
        fuggvenynev = tolower(fuggvenynev);

        while (ellenoriz(fuggvenynev)==0)
        {
          printf("A fv. neve: "); fuggvenynev = beker(fuggvenynev);
          fuggvenynev = tolower(fuggvenynev);
        }
        struct fuggvenyelem *hova = fuggveny.berak(fuggvenynev);
        int darab;
        do
        {
          printf("Hany darab operandusa van?: "); darab = beker(darab)-'0';
          if (darab<1)
              printf("Hiba! Fv.-nek kell lennie legalabb 1 operandusanak!\n");
        }  while (!(darab>0));
        for (int i=1; i<=darab; ++i)
        {
          int optipus;
          printf("A(z) %2d. op. tipusa: ",i); optipus = beker(optipus);
          while (ellenoriz(optipus))  //igazat ad vissza, ha nincs ilyen bejegyzes
          {
                printf("Hiba! Nincs ilyen tipus!\n");
                printf("A(z) %2d. op. tipusa: ",i); optipus = beker(optipus);
          }
          fuggveny.berak(hova, optipus, OPERANDUS);
        }
        int ertek;
        printf("A fv. visszateresi ertekenek tipusa: "); ertek = beker(ertek);
        while (ellenoriz(ertek))
        {
          printf("Hiba! Nincs ilyen tipus!\n");
          printf("A fv. visszateresi ertekenek tipusa: "); ertek = beker(ertek);
        }
        fuggveny.berak(hova, ertek, ERTEK);
        printf("Megad egy ujabb fv.-t? (i/n): "); valasz = beker(valasz);
        valasz = toupper(valasz);
      }  while (!(valasz=='N'));
}
//predikatumok megadasa
do
{
      int pred;
      printf("Predikatumszimbolum neve: "); pred = beker(pred);
      while (ellenoriz(pred)==0)  //ha mar van ilyen nevu bejegyzes
      {
        printf("Hiba! mar van ilyen nevu bejegyzes.\n");
        printf("Predikatumszimbolum neve: "); pred = beker(pred);
      }
      struct predikatumelem *hova = predikatum.berak(pred);
      int darab;
      printf("Hany darab operandusa van?: "); darab = beker(darab)-'0';
```

```
        for (int i=1; i<=darab; ++i)
        {
          int optipus;
          printf("A(z) %2d. op. tipusa: ",i); optipus = beker(optipus);
          while (ellenoriz(optipus))  //igazat ad vissza, ha nincs ilyen bejegyzes
          {
            printf("Hiba! Nincs ilyen tipus!\n");
            printf("A(z) %2d. op. tipusa: ",i); optipus = beker(optipus);
          }
          predikatum.berak(hova, optipus);
        }
        printf("Megad egy ujabb predikatumot? (i/n): "); valasz = beker(valasz);
        valasz = toupper(valasz);
    }  while (!(valasz=='N'));
    printf("Szeretne file-ba menteni a beallitasokat? (i/n): ");
    valasz = beker(valasz); valasz = toupper(valasz);
    while (!(valasz=='I' || valasz=='N'))
    {
        printf("Hibas karakter.\n");
        printf("Szeretne file-ba menteni a beallitasokat? (i/n): ");
        valasz = beker(valasz); valasz = toupper(valasz);
    }
    if (valasz=='I') fileba_ment();
  }//end of "kezdeti else ag", vagyis: ha nem file-bol adjuk meg a nyelvet
}
```

Tehát megadtunk egy nyelvet. További lehetőségek:

a) kimentjük file-ba (ajánlott)
b) betöltjük file-ból

a) file-ba való kimentés

Ez a következőt jelenti: a megfelelő osztályobjektumok (**Tipus tipus; Fuggveny fuggveny; Predikatum predikatum;** [ezek a Formula osztály tagjai]) fel vannak töltve, s csupán ezeket a láncolt listákat, multilistákat kellene bejárni, s az adatokat valamilyen alkalmas módon kiírni file-ba úgy, hogy azt legközelebb el is tudjuk olvasni.

Maga a függvény:

```
void Formula::fileba_ment()
{
  FILE *fp;
  char nev[8+1+3+1];                    //egy file hossza ennyi max.
  printf("A file neve (amibe mentjuk): ");
  fgets(nev, sizeof(nev), stdin); nev[strlen(nev)-1]='\0';
  while (!( fp=fopen(nev,"wt") ))
  {
    printf("Hiba. Nem tudom letrehozni. Adjon meg egy masik nevet!\n");
    printf("A file neve (ahonnan betoltjuk): "); fgets(nev, sizeof(nev), stdin);
nev[strlen(nev)-1]='\0';
   }
```

```
{
  fprintf(fp,"Srt/Cnst\n");
  struct tipuselem *akt = tipus.getFej();

  while (akt)
  {
      fprintf(fp,"%s(", akt->nev);
      struct listaelem *mutato = akt->val;
      while (mutato)
      {
        fprintf(fp,"%s%s", mutato->nev, (mutato->kov)?(","):(""));
        mutato = mutato->kov;
      }
      fprintf(fp,"|"); mutato = akt->kons;
      while (mutato)
      {
        fprintf(fp,"%s%s", mutato->nev, (mutato->kov)?(","):(""));
        mutato = mutato->kov;
      }
      fprintf(fp,")\n");
      akt = akt->kov;
  }
}
fprintf(fp,"\n");
//ez iratta ki a Tipus osztaly tartalmat
{
  fprintf(fp,"Fv\n");
  struct fuggvenyelem *akt = fuggveny.getFej();
  while (akt)
  {
      fprintf(fp,"%s(%s|", akt->nev, akt->ertek);
      struct listaelem *mutato = akt->op_tipusa;
      while (mutato)
      {
        fprintf(fp,"%s%s", mutato->nev, (mutato->kov)?(","):(""));
        mutato = mutato->kov;
      }
      fprintf(fp,")\n");
      akt = akt->kov;
  }
}
fprintf(fp,"\n");
//ez iratta ki a Fuggveny osztaly tartalmat
{
  fprintf(fp,"Pred\n");
  struct predikatumelem *akt = predikatum.getFej();
  while (akt)
  {
      fprintf(fp,"%s(", akt->nev);
      struct listaelem *mutato = akt->op_tipusa;
      while (mutato)
```

```
      {
        fprintf(fp,"%s%s", mutato->nev, (mutato->kov)?(","):(""));
        mutato = mutato->kov;
      }
      fprintf(fp,")\n");
      akt = akt->kov;
    }
  }
  fprintf(fp,"\n\n__END__\n"
          "eme file felepitese:\n"
          "Srt/Cnst\ntipus_neve(valtozok_neve|[konstansok_neve])\t\t#konstansok:
opcionalis\n\t\t\t\t\t\t#egy ures sor\n"
          "[Fv\nfuggveny_neve(eredmeny_tipusa|operandusok_tipusai)\n\t\t\t\t\t\t#ez az
egesz opc.\n]\t\t\t\t\t\t#egy ures sor a vegen\n"
          "Pred\npredikatum_neve(operandusok_tipusai)\n\t\t\t\t\t\t#egy ures sor\n");
  fflush(fp);
  fclose(fp);
}
```

Ennek eredményeképpen a következő lehetséges output-ot kapjuk:

```
Srt/Cnst
p(x,y|)

Fv
f(p|p,p)
g(p|p)

Pred
P(p,p,p)
Q(p,p)


__END__
eme file felepitese:
Srt/Cnst
tipus_neve(valtozok_neve|[konstansok_neve])          #konstansok: opcionalis
                                                            #egy ures sor
[Fv
fuggveny_neve(eredmeny_tipusa|operandusok_tipusai)
                                                            #ez az egesz opc.
]                                                           #egy ures sor a vegen
Pred
predikatum_neve(operandusok_tipusai)
                                                            #egy ures sor
```

Ez a file a következő információkat hordozza:

- egyetlen típus van, a **p**
- 2 változó: **x** és **y**, típusuk: **p**
- nincs konstans megadva
- 2 fv.-ünk van: **f** és **g**. **f** visszatérési értékének típusa **p**, 2 operandusa van, mindkettő **p** típusú**;** a **g** fv. visszatérési értékének típusa **p**, s egyetlen operandusa van, melynek típusa **p.**
- 2 predikátum, az egyik 3 operandusú (**P**), a másik 2 operandusú (**Q**). Mindkettőnek **p** típusúak az operandusai.

b) file-ból való beolvasás

A megadott file-t az előbb leírt módon kell feldolgozni, s a láncolt listákat, multilistákat fel kell tölteni:

```
void Formula::filebol_epit()
{
  FILE *fp;
  char nev[8+1+3+1];                          //egy file-nev max. hossza (DOS)
  char sor[MAX_SORHOSSZ];
  printf("A file neve (ahonnan betoltjuk): "); fgets(nev, sizeof(nev), stdin);
nev[strlen(nev)-1]='\0';
  while (!( fp=fopen(nev,"rt") ))
  {
    printf("Hiba. Nem tudom megnyitni. Adjon meg egy masik nevet!\n");
    printf("A file neve (ahonnan betoltjuk): "); fgets(nev, sizeof(nev), stdin);
nev[strlen(nev)-1]='\0';
  }
  fgets(sor,MAX_SORHOSSZ,fp); sor[strlen(sor)-1]='\0';
  if (strcmp(sor,"Srt/Cnst")==0)  //egyezes
  {
    fgets(sor,MAX_SORHOSSZ,fp); sor[strlen(sor)-1]='\0';
    while (strcmp(sor, ""))                   //amig nem talalkozik az ures sorral
    {
        char *i = sor;
        struct tipuselem *hova = tipus.berak(*i++);
        ++i; // "(" atugrasa
        while (*i != '|')
        {
          tipus.berak(hova,*i++, VALTOZO);
          if (*i == ',') ++i;                 // "," atlepese
        }
        ++i; // "|" atlepese
        while (*i != ')')
        {
          tipus.berak(hova,*i++, KONSTANS);
          if (*i == ',') ++i;                 // "," atlepese
        }
        fgets(sor,MAX_SORHOSSZ,fp); sor[strlen(sor)-1]='\0';
        int j=0;                 //a Borland C++ 3.1 fordito miatt van itt
    }                            //enélkül ↑ furcsa ugrásokat csinált a sorok közt
  }

  fgets(sor,MAX_SORHOSSZ,fp); sor[strlen(sor)-1]='\0';
  if (strcmp(sor,"Fv")==0)                    //egyezes
  {
    fgets(sor,MAX_SORHOSSZ,fp); sor[strlen(sor)-1]='\0';
    while (strcmp(sor, ""))                          //amig nem talalkozik az ures sorral
    {
        char *i = sor;
        struct fuggvenyelem *hova = fuggveny.berak(*i++);
```

```
        ++i;                                        // "(" atugrasa
        fuggveny.berak(hova, *i++, ERTEK);
        ++i;                                        // "|" atlepese

        while (*i != ')')
        {
          fuggveny.berak(hova, *i++, OPERANDUS);
          if (*i == ',') ++i;                       // "," atlepese
        }
        fgets(sor,MAX_SORHOSSZ,fp); sor[strlen(sor)-1]='\0';
        int j=0;                //a Borland C++ 3.1 fordito miatt van itt
    }                           //enélkül ↑ furcsa ugrásokat csinált a sorok közt
  }

  fgets(sor,MAX_SORHOSSZ,fp); sor[strlen(sor)-1]='\0';
  if (strcmp(sor,"Pred")==0)                         //egyezes
  {
    fgets(sor,MAX_SORHOSSZ,fp); sor[strlen(sor)-1]='\0';
    while (strcmp(sor, ""))                          //amig nem talalkozik az ures sorral
    {
        char *i = sor;
        struct predikatumelem *hova = predikatum.berak(*i++);
        ++i;                                        // "(" atugrasa

        while (*i != ')')
        {
          predikatum.berak(hova, *i++);
          if (*i == ',') ++i;                       // "," atlepese
        }
        fgets(sor,MAX_SORHOSSZ,fp); sor[strlen(sor)-1]='\0';
    }
  }
  fclose(fp);
}
```

A nyelv megadásánál még alkalmaztunk egy fv.-t, amelynek az a feladata, hogy a beolvasott karaktert leellenőrizze aszerint, hogy korábban megadtunk-e már ilyen karaktert. Ha már szerepelt, akkor hibát jelez, ha pedig még nem szerepelt, akkor fel lehet venni:

```
int Formula::ellenoriz(int mit) {
  char temp[2] = {mit, '\0'};
  char *str = strdup(temp); if (!str) hiba();

  return (!( tipus.eleme(str) || fuggveny.eleme(str) || predikatum.eleme(str) || lista.eleme(str) ));
}
```

És itt látható, hogy miért is kellett a korábbi 4 osztály mindegyikénél egy **eleme()** fv.: a **Formula::eleme()** mostmár csak ezeket fogja megkérdezni. Ha még sehol sem szerepelt ez a jel, szimbólum, akkor a (0 || 0 || 0 || 0) [4 db 0 logikai VAGY-ja] 0-t fog jelenteni. Ezt negáljuk; lesz belőle 1, ami majd a hívó prg.-rész számára azt jelzi, hogy ez a szimbólum felvehető.

A későbbiek során egy beolvasott karakterről el kell tudni dönteni, hogy az mi is valójában (konstans, változó, stb.). A **miez()** fv. ezt fogja nekünk megnézni:

```
char Formula::miez(char *mit)
{
  char p;
  p = lista.miez(mit);
  if (p=='h') p = tipus.miez(mit);
  if (p=='h') p = fuggveny.miez(mit);
  if (p=='h') p = predikatum.miez(mit);
  return p;
}
```

Ez is csupán a már meglévő objektumok megfelelő metódusait hívja meg sorban egymás után.

/***/
Az eddigi prg.-rész eddig összefoglalva a következőt tette: bekért egy nyelvet (billentyűzet-ről vagy file-ból), s figyelte, hogy egy szimbólum nem fordult-e elő kétszer. Ha igen (elő-fordult), akkor azt újrakérte, vagyis nem engedett be hibás adatokat.

Tehát megvan a nyelv eddig.
/***/

Beolvassuk a formulát:

```
void Formula::formula_beolvasasa()
{
  clrscr();
  printf("Logikai osszekoto jelek:\n"
          "------------------------\n"
          "negacio:\t\t¬ (Alt+191)\nES:\t\t\t&\nVAGY:\t\t\tl\nkonjunkcio:\t\t>\n\n"
          "Kvantorok:\n"
          "----------\n"
          "Univerzalis:\t\tV\nEgzisztencialis:\tE\n\n\n");
  printf("a teljesen bezarojelezett formula:\n\n");
  char temp[FORM_HOSSZ];
  char vegleges[FORM_HOSSZ]; int hova=0;
  fgets(temp,FORM_HOSSZ,stdin); temp[strlen(temp)-1]='\0';
  char *i = temp;
  while (*i)                                   //szóközök eltávolítása
  {
    if ((*i)!=' ') vegleges[hova++] = *i;
    ++i;
  }
  vegleges[hova] = '\0';
  formula = strdup(temp);                  //ez a Formula osztaly adattagja
}
```

/***/
Sikerült eddig megadni a nyelvet, ill. beolvasni a formulát. Most már le is kellene ellenőrizni, s felépíteni belőle egy formulafát.
/***/

```
void Formula::eloellenorzes()
{
  //zarojelek ellenorzese
  int count=0;
  char betu;
  char *i = formula;
  while (*i)
  {
    if (*i == '(') ++count;
    else if (*i == ')') --count;
    betu = miez(i);
    if (betu == 'h') hiba(1,*i);                    //ismeretlen szimbólumok kiszűrése
    if (count<0) hiba(0);
    ++i;
  }
  if (count!=0) hiba(0);
}
```

Ez a fv. leellenőrzi a zárójelek számát, ill. figyeli a formula egyes szimbólumait: ha ismeretlen szimbólummal találkozik, akkor hibaüzenet.
Hogyan állapítja meg egy karakterről, hogy az mi? (változó, konstans, fv.-szimbólum, vagy egyéb, ismeretlen jel). A **miez()** fv. segítségével.

Ezután felépítjük belőle a formulafát, amely fv. majd *a fa gyökerével tér* majd *vissza*.

```
struct faelem * Formula::epit(char *str, int lista_lesz)
{
  Verem verem;

  char *i = str;
  char betu, temp, temp2;
  char koztes_resz[FORM_HOSSZ];
  struct faelem * teteje;

  while (*i)
  {
    betu = miez(i);
    switch (betu)
    {
        case 'k' :  verem.berak(*i); ++i;    //a valtozon van
                    temp=miez(i); if (temp!='v') hiba(3);
                    verem.berak(*i); ++i; //a '('-en van
                    if ((*i)=='(')
                    {
                        eloallit(i, koztes_resz);
                        verem.berak(epit(koztes_resz));
```

```
                        break;
                    }
                    else hiba(0);
        case 's' :  if ((*i)!='(') break;         //ha '(' nyito zarojel
                    int k=0;
                    eloallit(i, koztes_resz);    //koztes_reszt es i-t modositja!!!
                    verem.berak(epit(koztes_resz));
                    break;
        case 'l' :  if ((*i)=='ż')
                    {
                        verem.berak(*i);
                        if ((*(i+1))!='(') hiba(0);
                    }
                    else
                    {
                        verem.berak(*i);
                        break;
                    }
                    break;
        case 'p' :  verem.berak(*i);
                    if ((*(i+1))=='(')
                    {
                        ++i;                    //a '('-re allitottuk
                        teteje = verem.top();
                        eloallit(i, koztes_resz);
                        teteje->le = epit(koztes_resz, LISTA);
                    }
                    break;
        case 'f' :  verem.berak(*i);
                    if ((*(i+1))!='(') hiba(0);
                    /*else if temp=='('*/
                    ++i;                    //a '('-re allitottuk
                    teteje = verem.top();
                    eloallit(i, koztes_resz);
                    teteje->le = epit(koztes_resz, LISTA);
                    if (lista_lesz)
                        if ((*(i+1))!=',' && (*(i+1))!='\0') hiba(7);  //ha ez is egy listaban van, ak-
kor ha van valami utana, kell utana a vesszo
                    break;
        case 'v' :
        case 'c' :  if (!lista_lesz) hiba(4, betu);
                    verem.berak(*i);
                    if ((*(i+1))!=',' && (*(i+1))!='\0') hiba(5,betu);
                    break;
    } //end of 'switch (betu)'
    ++i;
  } //end of while (*i)

  int verem_merete = verem.meret;
  struct faelem *akt, *masodik, *bal, *kozep, *jobb;
  if (lista_lesz)
```

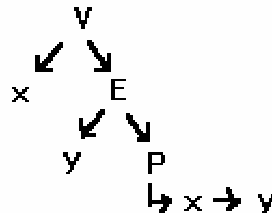
```
  {
    verem.osszefuz();
    akt = verem.alja();
    return akt;
  }

  switch (verem_merete)
  {
    case 1  :  akt = verem.alja();
               return akt;
    case 2  :  jobb = verem.kivesz();          //negacio
               kozep = verem.kivesz();
               kozep->jobb = jobb;
               return kozep;
    case 3  :  akt = verem.alja();
               masodik = verem.masodik();
               temp  = miez(akt->nev);
               temp2 = miez(masodik->nev);
               if (temp=='k' && !(temp2=='v' || temp2=='l')) hiba(10);
               if (temp=='k' && temp2!='l')
               {
                 jobb=verem.kivesz();
                 bal = verem.kivesz();
                 kozep = verem.kivesz();
                 kozep->bal = bal; kozep->jobb = jobb;
                 return kozep;
               }
               //else
               jobb=verem.kivesz();
               kozep = verem.kivesz();
               bal = verem.kivesz();
               kozep->bal = bal; kozep->jobb = jobb;
               return kozep;
    default :  hiba(6);
  } //end of switch
  return NULL;             //a fordito miatt, mar hamarabb visszater amugy
}
```

Mivel teljesen be van zárójelezve a formulánk, ezért az egyes zárójeles részekre fog meg-hívódni a fv. rekurzívan, pl.:
(Vx(Ey(P(x,y)))) *V - univerzális kvantor; E - egzisztenciális k.*
elindul az elejétől egy mutató (i névre hallgat), s nézi, mire mutat. Itt ez egy '**(**' lesz, ezért eme zárójel, és a párja közti részre meghívja önmagát, s a visszaadott mutatót berakja a verembe. A mutatót pedig a '**(**' zárójel párja után állítjuk. Ezt a munkát (köztes rész kisze-dése, mutató átállítása) végzi el az **eloallit()** fv.
Vx(Ey(P(x,y)))
Ezzel hívtuk meg a fv.-t. Első karakter: kvantor. Utána csak változó állhat, ezt ellenőrizzük. Mindkettőt berakjuk. Változó után '**(**' kell legyen a teljes bezárójelezés miatt, ha nem: hiba. Verembe be a köztes részből előállított részfa gyökerét.
Ey(P(x,y))

Előzőekhez hasonló.
P(x,y)
Ez predikátum, amelynek kettőnél több (!!!) operandusa is lehet. Én így az operandusokat egy láncolt listában tárolom le. Direkt erre a célra a **faelem** struktúrának van egy **le** nevű mutatója, amelyet csak a függvények és a predikátumok használnak ki. Ez mutat majd az operandusok alkotta láncolt listára. Valahogy viszont jelezni kellene, hogy az átadott köztes részből:
x,y
egy listát kellene csinálni. Erre való a **LISTA** nevű konstans, amely értéke alapértelmezésben 0, elhagyható (C++ specialitás), de ki is írható, s ekkor felülírjuk az alapértelmezett értékét.
A megkapott karaktersorozatot ismét elkezdi értelmezni: változó. Mivel be van kapcsolva a **LISTA** mód, ezért nem fog hibát jelezni, hanem bepakolgatja őket a verembe. Vége a "sztringnek", kilépünk a **while (*i)** ciklusból. A veremben ott van **x** és **y**, s mivel ezekből lista lesz, ezért összefűzzük őket, amit a **verem.osszefuz()** metódus el is végez: **x** kov mutatóját **y**-ra állítja, s mivel nincsenek többen a veremben, **y** kov mutatója NULL lesz. Visszatér **x** címével, amit **P le** mutatója meg is kap miután visszaléptünk a rekurzióban. Veremben csak **P** van, verem mérete 1 ⇒ visszatér **P** címével. Rekurzióban vissza, verembe **P** címe be.
3-an vannak a veremben: **EyP**. A verem alján kvantor: megfelelő sorrendben kivesszük őket, fát építünk (középen **E**, tőle balra **y**, jobbra **P**). A középső címével vissza, rekurzióban vissza, **E** címe be a verembe.
Verem: **VxE**. Ugyanígy fát épít, középső címével (**V**) vissza, s ezt kapja majd meg a **Formula::gyoker** nevű tag!
Ha a fa építése közben valami gond volt: hibaüzenet. A **gyoker** tag csak akkor kap értéket, ha sikerült a faépítés.

A tárban hogy néz ki a fa?



A függvény által felhasznált "segédfüggvények":

```
void Formula::eloallit(char *&i, char atadando[])
{
  int count=1, hova=0; ++i;                    //i a '(' utanra lett allitva
  while (count!=0)
  {
    if (*i == '(') ++count;
    else if (*i == ')') --count;
    if (count!=0) atadando[hova++] = *i;
    ++i;
  }
  atadando[hova] = '\0';
  --i;          //most i a zaro ) jelre mutat, az epit fv. vegen noveljuk majd
}
```

Működésének lényegéről már volt szó (egy nyitó '(' zárójelhez tartozó csukó ')' zárójel *közti* rész "kiszedése", s eközben **i**-t úgy növeljük, hogy az kihat az **epit()** fv.-re is (ui. a mutatót "cím" szerint adtuk át, C++).
i tehát a végén arra a csukó '**)**'-re fog mutatni, amely a kezdeti '**(**' párja.

A következő fv. csupán visszaállítja a fából a teljesen bezárójelezett formulát, pl. az előző ábrán látható fán alkalmazva ezt kapnánk:
(Vx(Ey(P(x,y))))

```
void Formula::bejar(struct faelem *gyoker)
{
  char betu, temp;
  struct faelem *akt;
  if (gyoker)
  {
    betu = miez(gyoker->nev);
    switch (betu)
    {
         case 'l' :  cout<<"(";
                     bejar(gyoker->bal);
                     cout<<gyoker->nev;
                     bejar(gyoker->jobb);
                     cout<<")"; break;
         case 'k' :  cout<<"("; cout<<gyoker->nev;
                     bejar(gyoker->bal); bejar(gyoker->jobb);
                     cout<<")"; break;
         case 'p' :
         case 'f' :  cout<<"("; cout<<gyoker->nev;
                     if (gyoker->le)
                     {
                          cout<<"(";
                          bejar(gyoker->le);
                          cout<<")";
                     }
                     cout<<")";
                     if (gyoker->kov)
                     {
                          cout<<","; bejar(gyoker->kov);
                     }
                     break;
         case 'v' :
         case 'c' :  cout<<gyoker->nev;
                     if (gyoker->kov)
                     {
                          cout<<","; bejar(gyoker->kov);
                     }
                     break;
    }//end of switch
  }
}
```

A Verem osztály megadását a legvégére hagynám; a metódusok nevei "elég beszédesek", végülis azt valósítják meg, amire a nevük is utal.

Végül meg kell vizsgálni, hogy a fv.-ek és predikátumok operandusainak száma és típusa renden van-e:

```
void Formula::utoellenorzes(struct faelem *gyoker)
{
  char betu, temp;
  char eredeti[OPERAND_SZAMA], itteni[OPERAND_SZAMA];
  struct faelem *akt;
  if (gyoker)
  {
    betu = miez(gyoker->nev);
    switch (betu)
    {
         case 'l' :  utoellenorzes(gyoker->bal);
                     utoellenorzes(gyoker->jobb);
                     break;
         case 'k' : utoellenorzes(gyoker->bal); utoellenorzes(gyoker->jobb);
                     break;
         case 'p' :
         case 'f' :  operandus_lista(gyoker->nev, betu, eredeti);
                     itteni_lista(gyoker,itteni);
                     if (strcmp(itteni,eredeti)) hiba(9,(gyoker->nev)[0]);

                     if (gyoker->le)  utoellenorzes(gyoker->le);
                     if (gyoker->kov) utoellenorzes(gyoker->kov);
                     break;
         case 'v' :
         case 'c' :  if (gyoker->kov) utoellenorzes(gyoker->kov);
                     break;
    }//end of switch
  }
}
```

Elkezdjük bejárni a fát, s ha valahol találunk egy predikátum- vagy fv.-szimbólumot, akkor beindul az ellenőrző rész: egy "sztring"-be kigyűjtjük azon típusneveket, amiknek szerepelnie kellene (a nyelv szerint), egy másikba pedig azt, hogy mi szerepel valójában. A két sztring összehasonlításából kiderül, hogy egyeznek-e, vagy sem.

```
void Formula::operandus_lista(char *fv_pred_nev, char miez, char str[])
{
  switch (miez)
  {
    case 'p' :  predikatum.opTipusa(fv_pred_nev, str); break;
    case 'f' :  fuggveny.opTipusa(fv_pred_nev, str);   break;
  }
}
```

Ez a fv. azt gyűjti ki, hogy a megadott *nyelv szerint* milyen típusú argumentumok szükségesek.

Ez pedig: a felhasználó mit adott meg.

```
void Formula::itteni_lista(struct faelem *gyoker, char str[])
{
  int hova=0;
  char betu, temp;
  struct faelem *akt = gyoker->le;
  while (akt)
  {
    betu = miez(akt->nev);
    if (betu=='c' || betu=='v') temp = tipus.getTipusa(akt->nev);
    else if (betu=='f') temp = fuggveny.getTipusa(akt->nev);
    else hiba(8,(gyoker->nev)[0]);
    str[hova++] = temp;
    akt = akt->kov;
  }
  str[hova] = '\0';
}
```

/***/
Ha itt sincs hiba, akkor helyes a formula.
/***/

Az egyes hibaüzenetek:

```
void Formula::hiba(int kod, char betu)
{
  switch (kod)
  {
    case 0    :  fprintf(stderr,"Zarojelezesi hiba.\n"); break;
    case 1    :  fprintf(stderr,"Ismeretlen szimbolum: %c.\n",betu); break;
    case 2    :  fprintf(stderr,"Hiba a kov. fv.-nel v. pred.-nal: %c.\n",betu); break;
    case 3    :  fprintf(stderr,"Hiba a kvantor utan.\n"); break;
    case 4    :  fprintf(stderr,"Valtozo vagy konstans (%c) nem megfelelo "
                                "helyen.\n",betu); break;
    case 5    :  fprintf(stderr,"Valtozo vagy konstans (%c) utan nem megfelelo "
                                "karakter.\n",betu); break;
    case 6    :  fprintf(stderr,"Hibas.\n"); break;
    case 7    :  fprintf(stderr,"Fv. vagy pred. argumentumlistaja nem megfelelo.\n");
         break;
    case 8    :  fprintf(stderr,"Hibas a %c fv. vagy pred. operanduslistaja.\n",betu);
         break;
    case 9    :  fprintf(stderr,"Nem megfelelo operandusszam vagy tipushiba a %c fv. "
                          "vagy pred. eseten.\n",betu); break;
    case 10   : fprintf(stderr,"Hibas.\n"); break;
  }
  exit(-1);
}
```

A végére maradt a Verem osztály megadása:

```
struct veremelem {
  struct faelem    *fa;                    //fára mutató mutatókat tárolunk le láncolt listában
  struct veremelem *kov;
};

class Verem {
  public:
    Verem() : fej(NULL), vege(NULL), meret(0) {};                    //inicializálás
    ~Verem();
    void berak(struct faelem *cim);
    void berak(char mit);
    struct faelem * top();
    struct faelem * alja();
    struct faelem * kivesz();
    void osszefuz();
    void hiba(int kod);
    int meret;
  private:
    struct veremelem *fej, *vege;
};

Verem::~Verem()                //destruktor. Csak a listát számolja fel, amire viszont
{                              // mutatnak, a faelemeket, azokat nem törlik
   struct veremelem *akt;
   while (fej)
   {
      akt=fej; fej=fej->kov; free(akt);
   }
}

void Verem::hiba(int kod)            //milyen hibák lehetnek egy verem esetén
{
   switch (kod)
   {
      case 0 :  fprintf(stderr,"Mem.-fogl. hiba (Verem osztaly).\n"); break;
      case 1 :  fprintf(stderr,"Ures verembol probalt kivenni.\n"); break;
   }
   exit(-1);
}
```

A köv. fv. a verembe berakja egy faelem címét, a *faelem már létezik, neki nem foglal még egyszer mem.-területet.*

```
void Verem::berak(struct faelem *cim) {
   struct veremelem *uj = (struct veremelem *)malloc(sizeof(struct veremelem));
   if (!uj) hiba(0);
   uj->fa = cim; uj->kov=NULL;
   if (!fej) fej=vege=uj;
   else vege->kov=uj, vege=uj;
   ++meret;
}
```

Az előző fv.-nyel ellentétben itt a faelem még nem létezik, vagyis tárterületet foglalunk neki, s a címét berakjuk a verembe.

Miért eme megkülönböztetés? Amikor "befele" megyünk a rekurzióba, akkor még nincs nekik tárterület foglalva, viszont amikor "kifelé" jövünk belőle, akkor már elég csak a faelem címét berakni, már van neki fenntartott tárterület.

```
void Verem::berak(char mit)
{
  struct veremelem *uj = (struct veremelem *)malloc(sizeof(struct veremelem));
  if (!uj) hiba(0);
  uj->kov=NULL;
  if (!fej) fej=vege=uj;
  else vege->kov=uj, vege=uj;

  uj->fa = (struct faelem *)malloc(sizeof(struct faelem)); if (!uj) hiba(0);
  char str[2] = {mit, '\0'};
  struct faelem *ujfa = uj->fa;
  ujfa->nev = strdup(str); if (!(ujfa->nev)) hiba(0);
  ujfa->bal = ujfa->jobb = ujfa->le = ujfa->kov = NULL;
  ++meret;
}

struct faelem * Verem::top()               //verem legfelső elemének címe
{
  if (!vege) hiba(1);
  return (vege->fa);
}

struct faelem * Verem::alja() {            //verem legalsó elemének címe
  if (!fej) hiba(1);
  return (fej->fa);
}

struct faelem * Verem::kivesz()            //veremből törli legfelső elemet, s
{                                          //annak címével visszatér
  if (!fej) hiba(1);
  --meret;                                 //verem mérete 1-gyel csökken
  struct veremelem *akt    = fej;
  struct faelem    *vissza;
  if (fej==vege)                           //1 elemu
  {
    vissza = fej->fa;
    free(fej); fej = vege = NULL;
    return vissza;
  }
  while (akt->kov != vege) akt = akt->kov;     //min. 2 elemu
  vissza = vege->fa;
  free(vege); vege = akt;
  return vissza;
}
```

A veremben lévő elemeket összefűzi, azaz a **kov** mutatóikat egymásra állítja (egy egyirányú láncolt listát készít belőlük).

```
void Verem::osszefuz()
{
  struct veremelem *akt = fej;
  struct faelem *egy, *ketto;
  while (akt)
  {
    if (akt->kov)
    {
         egy=akt->fa;
         ketto=(akt->kov)->fa;
         egy->kov = ketto;
    }
    akt = akt->kov;
  }
}

struct faelem * Verem::masodik()          //a verem aljától számított 2. elem címe
{
  if (!fej) hiba(1);
  if (!(fej->kov)) hiba(1);
  return ((fej->kov)->fa);
}
```

Ami kimaradt: a főprogram:

```
void main()
{
  Formula form;
  form.vezerlo();
}
```

/**/
Kód vége
/**/

Futási eredmény:

Adjon meg egy elsorendu mat.log. nyelvet!
Szeretne file-bol betolteni egy nyelvet? n
Tipus neve: p
Valtozo neve: x
Megad egy ujabb valtozot? (i/n): i
Valtozo neve: y
Megad egy ujabb valtozot? (i/n): n
Megad konstansokat ehhez a tipushoz? (i/n): n
Megad egy ujabb tipust? (i/n): n
Megad fuggvenyeket? (i/n): i
A fv. neve: f

```
Hany darab operandusa van?: 2
A(z) 1. op. tipusa: p
A(z) 2. op. tipusa: p
A fv. visszateresi ertekenek tipusa: p
Megad egy ujabb fv.-t? (i/n): i
A fv. neve: g
Hany darab operandusa van?: 1
A(z) 1. op. tipusa: p
A fv. visszateresi ertekenek tipusa: p
Predikatumszimbolum neve: P
Hany darab operandusa van?: 3
A(z) 1. op. tipusa: p
A(z) 2. op. tipusa: p
A(z) 3. op. tipusa: p
Megad egy ujabb predikatumot? (i/n): i
Predikatumszimbolum neve: Q
Hany darab operandusa van?: 2
A(z) 1. op. tipusa: p
A(z) 2. op. tipusa: p
Szeretne file-ba menteni a beallitasokat? (i/n): i
A file neve (amibe mentjuk): mentes1.dat

Logikai osszekoto jelek:
----------------------------
negacio:            ¬ (Alt+191)
ES:                 &
VAGY:               |
konjunkcio:         >

Kvantorok:
-------------
Univerzalis:        V
Egzisztencialis:    E


a teljesen bezarojelezett formula:
((Vy(Ex(P(x,y,f(x,y)))))&(¬(Vx(Q(y,g(x))))))

fa felepitese OK
a fv.-ek es/vagy pred.-ok operandusainak tipusa(i) OK

----------------------------------------------------------------
a teljesen bezarojelezett formula:
((Vy(Ex(P(x,y,f(x,y)))))&(¬(Vx(Q(y,g(x,y))))))

fa felepitese OK
Nem megfelelo operandusszam vagy tipushiba a g fv. vagy pred. eseten.
```