Mesterséges intelligencia 2.

gyakorlat

6. feladat

(kvantor legkisebb hatáskörének megállapítása)

Készítette:

Szathmáry László
III. PTM.

6. feladat:

Olyan program megírása, amely elvégzi egy megadott kvantoros formulán a legkisebb hatáskör eljárást, azaz "bevisz" minden kvantort annak legkisebb hatáskörébe.

Megoldás:

Az első feladatot fogjuk kibővíteni, mégpedig azért, mert ott egy alkalmas reprezentációt használtunk a formula tárolására: formulafa. Vagyis az első feladat beolvasta a formulát, szintaktikailag leellenőrizte, majd fát épített belőle.
A továbbiakban ezen fával fogunk dolgozni.
Az első feladatban volt egy **Verem**, ill. egy **Formula** osztály. Ezeket néhány fv.-nyel ki kell egészíteni:

```
class Verem {
  public:
    Verem() : fej(NULL), vege(NULL), meret(0) {};
    ~Verem();
    void init();                                          /*** UJ ***/
    void berak(struct faelem *cim);
    void berak(char mit);
    int eleme(struct faelem *mi);                         /*** UJ ***/
    struct faelem * top();
    struct faelem * alja();
    struct faelem * kivesz();
    struct faelem * Verem::masodik();
    void osszefuz();
    void hiba(int kod);
    struct veremelem * getVege() { return vege; }         /*** UJ ***/
    struct veremelem * getFej()  { return fej;  }         /*** UJ ***/
    struct veremelem * getElozo(struct veremelem *akt);   /*** UJ ***/
    int meret;
  private:
    struct veremelem *fej, *vege;
};
```

Az új fv.-ek:

`init()`: ez a verem inicializálását végzi el. Erre a konstruktoron kívül azért lesz szükség, mert majd ugyanazt a vermet (ugyanazt az objektumot) egy helyen többször is fel akarjuk használni, s minden új használat előtt inicializálni kell majd a vermet, azaz tartalmát töröljük, mutatóit beállítjuk, mérete 0 lesz:

```
void Verem::init()
{
   struct veremelem *akt;
   while (fej)
   {
      akt=fej; fej=fej->kov; free(akt);
   }
   fej = vege = NULL;
   meret=0;
}
```

`eleme():` ebben a veremben mutatókat tárolunk láncolt listában, s az előző törlés során is csak ezen láncszemeket töröltük ki, nem pedig a láncszemek által mutatott facsomópontokat. Az `eleme()` fv. segítségével azt tudjuk megállapítani, hogy ebben a láncolt listában szerepel-e egy adott faelem címe:

```
int Verem::eleme(struct faelem *mi)
{
   struct veremelem *akt=fej;
   while (akt)
   {
      if (akt->fa == mi) return 1;        //megvan
      akt=akt->kov;
   }
   return 0;                              //nincs meg
}
```

`getVege(), getFej():` ezek csupán a megfelelő mutatókkal térnek vissza. Mivel private elérési jogú a két mutató, így kívülről csak eme két publikus fv. beiktatásával kérhető le az értékük.

`getElozo():` a láncolt listánk szerkezete a következő volt:

```
struct veremelem {
  struct faelem    *fa;
  struct veremelem *kov;
};
```

Látható, hogy a lista csak egyirányú láncolást tesz lehetővé, viszont nekünk majd később visszafelé is kellene tudnunk haladni. A reprezentációs változtatás elkerülhető a következő fv. közbeiktatásával:

```
struct veremelem * Verem::getElozo(struct veremelem *akt)
{
   struct veremelem *keres = fej;
   if (akt==fej) return NULL;
   while (keres->kov != akt) keres = keres->kov;
   return keres;
}
```

Vagyis ha legelöl vagyunk (`akt==fej`) akkor annak már nincs megelőzője, NULL-lal térünk vissza. Különben megkeressük a megelőző listaelemet, s annak a címét adjuk vissza.

Szükség lesz még egy konstansra is. (Az értékére csak annyi megkötés van, hogy ne 0 legyen).

```
#define  FIRST_CALL    7
```

Az első feladatban leírt **Formula** osztályt szintén ki kell egészíteni azon fv.-ekkel, amelyek majd elvégzik a legkisebb hatáskör eljárást.

```
class Formula {
  public:
    Formula() {};
    void vezerlo();
  private:
    Tipus tipus;
    Fuggveny fuggveny;
    Predikatum predikatum;
    Lista lista;
    char *formula;
    struct faelem *gyoker;

    int volt_modositas;                          /*** UJ ***/
    Verem nem_tud_lepni, osszes_kvantor;         /*** UJ ***/

    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 szetszed();
    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[]);
    void literalla_alakit();
    struct faelem* lit_alakra_hoz(struct faelem *, struct faelem *, int);
    void lit_kiir(struct faelem *gyoker);
    void kv_hataskor();
    void kv_elim(struct faelem *gyoker);
    void kv_torol(struct faelem *mit);
    int  elofordul(char *, struct faelem *gyoker, int elso_hivas=0);
    void valtozok_gyujt(struct faelem *, char [], int=0);
    void bemasol(struct faelem *kvantor, struct faelem *hova);
    void megprobal_bevinni(struct faelem *kvantor);
    void kv_bevitel();
    void kvantorok_kigyujtese(struct faelem *gyoker);
    void kvantorok_probaja();
};
```

(Az új fv.-neveket vastagon szedtem).

```
struct faelem {
  char *nev;
  struct faelem *vissza;                 /*** UJ ***/
  struct faelem *bal, *jobb;
  struct faelem *le;
  struct faelem *kov;
};
```

Itt kell egy kis reprezentációs változtatás: a rekordstruktúrát egy új mezővel bővítjük: egy visszafelé mutatóval. Így a fában visszafelé is tudunk majd lépkedni. Ez a változtatás egyáltalán nem érinti a szintaktikai ellenőrző részt, majd csak a kvantorok bevitelénél használjuk majd.

```
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);
   berak("l","*",3); berak("l","+",3);
   berak("k","Δ",5); berak("k","∇",5);
   berak("j","t");   berak("j","f");
}
```

A Lista osztály is bővítésre szorul (vastagított rész), ui. literálformára hozás esetén a fában új szimbólumok fognak megjelenni. A `miez()` fv. segítségével tudjuk mindig megállapítani egy facsomópontról, hogy az mi is valójában. A fv. által visszaadott értékek:

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

Lesz 2 új kvantorunk (Δ,∇), 2 új logikai összekötő jelünk (+, *), ill. 2 új jelünk: a **t** és az **f**, melyek a jelölt formulák előtt állnak. Ezeket a tárban külön faelemként tárolom. Nézzük a vezérlőt:

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

    literalla_alakit(); putchar('\n');                /*** UJ ***/
    printf("Literalla alakitva:\n");                  /*** UJ ***/
    lit_kiir(gyoker);   putchar('\n');  getch();      /*** UJ ***/
    kv_hataskor();                                    /*** UJ ***/
}
```

Eddig csak olyan változtatásokat, bővítéseket végeztünk, amelyek csupán előkészítették a prg.-ot a tényleges feladatra:

```
void Formula::literalla_alakit()
{
    int jel;

    cout<<"Milyen jelet rak ele? (\"t\" vagy \"f\"): "; jel = beker(jel);
    while (!(jel=='t' || jel=='f'))
    {
        cout<<"Milyen jelet rak ele? (\"t\" vagy \"f\"): "; jel = beker(jel);
    }
    lit_alakra_hoz(gyoker, NULL, jel);
}
```

A felhasználótól bekérjük, hogy milyen jelet szeretne a formula elé tenni (**t** vagy **f**). Ezáltal jelölt formulát csinálunk belőle, amit literálalakra kell hozni, hogy a későbbiekben tudjunk vele dolgozni.

Literális alakú pp-kifejezés: olyan pp-kifejezés, amely a következőkből állhat: $\uparrow, \downarrow$ szimbólum, fA, tA alakú jelölt formulák (ahol A atomi formula); ill. ezek összege, szorzata is lit. alakú pp-kifejezés. Továbbá ΔxE és ∇xE is lit. alakú pp-kif., ahol E lit. alakú pp-kif.

A literális alakra hozást ez a rekurzív fv. végzi el, amely a fa átalakítását helyben végzi el:

```
struct faelem* Formula::lit_alakra_hoz(struct faelem *gyoker, struct faelem *os, int
jel)
{
   char betu, temp;
   struct faelem *mutat, *uj;

   if (gyoker)
   {
      gyoker->vissza=os;         //itt vegezzuk el a visszafele lancolast
      betu = miez(gyoker->nev);
      temp = (gyoker->nev)[0];
      switch (betu)
      {
        case 'l':                //logikai osszekoto jel (ES,VAGY,IMPL.,NEG.)
        {
           switch (temp)
           {
              case '&':if (jel=='t')
                       {
                          (gyoker->nev)[0]='*';
                          gyoker->bal =lit_alakra_hoz(gyoker->bal, gyoker, 't');
                          gyoker->jobb=lit_alakra_hoz(gyoker->jobb,gyoker, 't');
                       }
                       else
                       {
                          (gyoker->nev)[0]='+';
                          gyoker->bal =lit_alakra_hoz(gyoker->bal, gyoker, 'f');
                          gyoker->jobb=lit_alakra_hoz(gyoker->jobb,gyoker, 'f');
                       }
                       return gyoker;
                       break;
```

```
      case '|':if (jel=='t')
               {
                  (gyoker->nev)[0]='+';
                  gyoker->bal =lit_alakra_hoz(gyoker->bal, gyoker, 't');
                  gyoker->jobb=lit_alakra_hoz(gyoker->jobb,gyoker, 't');
               }
               else
               {
                  (gyoker->nev)[0]='*';
                  gyoker->bal =lit_alakra_hoz(gyoker->bal, gyoker, 'f');
                  gyoker->jobb=lit_alakra_hoz(gyoker->jobb,gyoker, 'f');
               }
               return gyoker;
               break;
      case '>':if (jel=='t')
               {
                  (gyoker->nev)[0]='+';
                  gyoker->bal =lit_alakra_hoz(gyoker->bal, gyoker, 'f');
                  gyoker->jobb=lit_alakra_hoz(gyoker->jobb,gyoker, 't');
               }
               else
               {
                  (gyoker->nev)[0]='*';
                  gyoker->bal =lit_alakra_hoz(gyoker->bal, gyoker, 't');
                  gyoker->jobb=lit_alakra_hoz(gyoker->jobb,gyoker, 'f');
               }
               return gyoker;
               break;
      case '¬':mutat = gyoker->jobb;
               free(gyoker);
               if (jel=='t') return(lit_alakra_hoz(mutat, os,'f'));
               else          return(lit_alakra_hoz(mutat, os,'t'));
               break;
   }//switch (temp)
}//case 'l'
case 'k':          //kvantor (V, E)
{
   switch (temp)
   {
      case 'V':if (jel=='t')
               {
                  (gyoker->nev)[0]='Δ';
                  gyoker->jobb=lit_alakra_hoz(gyoker->jobb, gyoker,'t');
               }
               else
               {
                  (gyoker->nev)[0]='∇';
                  gyoker->jobb=lit_alakra_hoz(gyoker->jobb, gyoker,'f');
               }
               return gyoker;
               break;
      case 'E':if (jel=='t')
               {
                  (gyoker->nev)[0]='∇';
                  gyoker->jobb=lit_alakra_hoz(gyoker->jobb, gyoker,'t');
               }
               else
               {
                  (gyoker->nev)[0]='Δ';
                  gyoker->jobb=lit_alakra_hoz(gyoker->jobb, gyoker,'f');
               }
               return gyoker;
               break;
   }//switch (temp)
}//case 'k'
```

```
        case 'p':                                       //predikatumszimbolum
           uj = (struct faelem *) malloc(sizeof(struct faelem));
           if (!uj) hiba();
           char str[2] = {jel, '\0'};
           uj->nev = strdup(str);
           uj->bal = uj->le = uj->kov = NULL;
           uj->jobb = gyoker; gyoker->vissza=uj;
           uj->vissza=os;
           return uj;
      }//switch (betu)
   }//if (gyoker)
   return NULL;  //a fordito miatt; amugy mar hamarabb visszater
}
```

A felhasználandó multiplikatív törvények:

$t(A \wedge B) \sim tA * tB$	$f(A \wedge B) \sim fA + fB$
$t(A \vee B) \sim tA + tB$	$f(A \vee B) \sim fA * fB$
$t(A \supset B) \sim fA + tB$	$f(A \supset B) \sim tA * fB$
$t\neg A \sim fA$	$f\neg A \sim tA$
$t\forall xA(x) \sim \Delta x tA(x)$	$f\forall xA(x) \sim \nabla x fA(x)$
$t\exists xA(x) \sim \nabla x tA(x)$	$f\exists xA(x) \sim \Delta x fA(x)$

A fv. működése: bejárjuk a fát, s az egyes ágakban csak a predikátumszimbólumokig megyünk le, azok argumentumlistájába már nem. Vagyis van egy faelemünk, s megvizsgáljuk, hogy mi is. Ha logikai összekötő jel, akkor őt magát átírjuk "+"-ra vagy "*"-ra (ez persze attól függ, hogy melyik multiplikatív törvényt tudjuk használni), majd a megfelelő jellel (**f** vagy **t**) rekurzívan meghívjuk a fv.-t a logikai összekötő jel bal és jobb ágára is. Negáció és kvantor esetén is a multiplikatív törvényeket alkalmazzuk.

Ha lejutottunk egy predikátumszimbólumig, akkor az elé berakjuk a megfelelő jelet (**f** vagy **t**) úgy, hogy a jelnek külön tárterületet foglalunk.

Ahhoz, hogy a visszafelé mutatók is be legyenek állítva, a rekurzív hívásoknál mindig átadjuk az aktuális gyökeret, ami a következő hívás során előálló új aktuális gyökér őse! Eme aktuális gyökér visszafelé mutatóját erre az ősre ráállítva elvégezhető a láncolás. A fv. legelső hívása így néz ki:

```
lit_alakra_hoz(gyoker, NULL, jel);
```

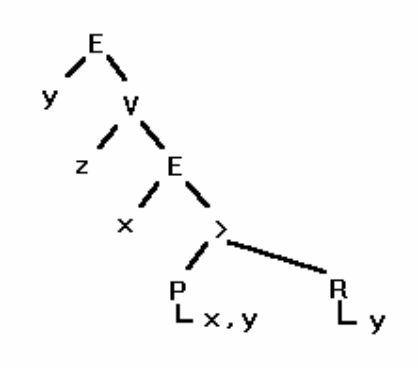
Az ős itt NULL, ui. a tényleges gyökérnek nincs őse, így azt NULL-ra kell majd állítanunk.

Példa:

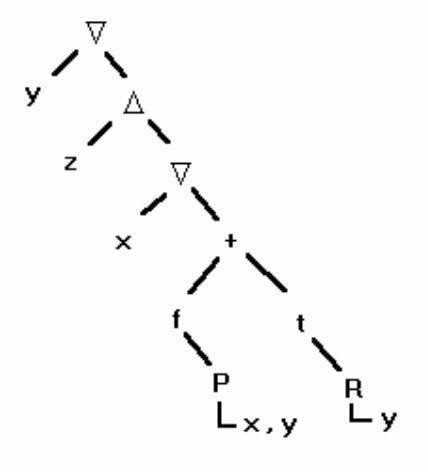
Nézzük meg az előző fv. működését "élőben":

Adott a köv. formula: $\exists y \forall z \exists x (P(x,y) \supset R(y))$

Ez a formulafában a következőképpen lesz letárolva:



Erre futtatjuk le a `lit_alakra_hoz()` fv.-t, amely a következőképpen alakítja át a fánkat, ha az egész formula elé egy "t" szimbólumot teszünk (azaz ha "t" jelölt formulává alakítjuk):



Szükség lenne rá, hogy ezt a fát meg is tudjuk jeleníteni a képernyőn valamilyen formában. Ehhez kell a köv. `lit_kiir()` nevű fv.-t fogjuk felhasználni, amely rekurzív módon bejárja a fát, s a fában tárolt literális alakú pp-kifejezést megjeleníti.

Az előző fára alkalmazva a következőt kapnánk eredményül:

$$\nabla y \Delta z \nabla x((fP(x,y)+tR(y))$$

```
Void Formula::lit_kiir(struct faelem *gyoker)
{
   char betu, mi_a_jobb_ag;

   if (gyoker)
   {
      betu = miez(gyoker->nev);
      switch (betu)
      {
       case 'l':cout<<"(";
                lit_kiir(gyoker->bal);
                cout<<gyoker->nev;
                lit_kiir(gyoker->jobb);
                cout<<")";
                break;
       case 'k':cout << gyoker->nev;
                lit_kiir(gyoker->bal);
                mi_a_jobb_ag = miez((gyoker->jobb)->nev);
                if (mi_a_jobb_ag!='l') lit_kiir(gyoker->jobb);
                else
                {
                   cout<<"(";
                   lit_kiir(gyoker->jobb);
                   cout<<")";
                }
                break;
       case 'j':cout<<gyoker->nev;
                lit_kiir(gyoker->jobb);
                break;
       case 'p':
       case 'f':cout<<gyoker->nev;
                if (gyoker->le)
                {
                   cout<<"(";
                   bejar(gyoker->le);
                   cout<<")";
                }
                if (gyoker->kov)    //'predikatumban fuggvenyek' esete
                {
                   cout<<","; bejar(gyoker->kov);
                }
                break;
       case 'v':
       case 'c':cout<<gyoker->nev;
                if (gyoker->kov)
                {
                   cout<<","; bejar(gyoker->kov);
                }
                break;
      }//switch (betu)
   }//if (gyoker)
}
```

A formulafánk most már kész arra, hogy megállapítsuk a kvantorok legkisebb hatáskörét. Az ehhez szükséges eljárásokat egy külön kis "vezérlő"-ben fogjuk össze:

```
void Formula::kv_hataskor()        //ez lesz itt a 'lokalis vezerlo'
{
   kv_elim(gyoker);
   printf("\nFiktiv kvantorok eliminalva:\n");
   lit_kiir(gyoker);    putchar('\n'); getch();

   cout<<"\nA kvantorbevitel lepesei (legszukebb hataskor,
billentyuleutesre tovabblep):\n";
   kv_bevitel();
}
```

Első menetben eltávolítjuk a fiktív kvantorokat, azaz ahol:

$$QxE \sim E \quad \text{,ahol } x \notin Fv(E)$$

```
void Formula::kv_elim(struct faelem *gyoker)
{
   struct faelem *akt;

   if (gyoker)
   {
      if (miez(gyoker->nev)=='k')
      {
       if (elofordul((gyoker->bal)->nev, gyoker->jobb, FIRST_CALL))
            kv_elim(gyoker->jobb);
       else /* most jon a torles */
       {
          akt = gyoker->jobb;
          kv_torol(gyoker);
          kv_elim(akt);
       }
      }
      else /* ha nem kvantor */
      {
       kv_elim(gyoker->bal);
       kv_elim(gyoker->jobb);
      }
   }//if (gyoker)
}
```

Ez a fv. bejárja a fát, s minden kvantor esetén "akcióba lép", azaz meghívja az `elofordul()` fv.-t, amely megnézi, hogy a kvantor által kötött változó `((gyoker→bal)→nev)` előfordul-e a kvantor jobb részfájában. Ha előfordul, akkor nem fiktív kvantor, marad, megyünk tovább a fában (keressük a következő kvantort).

Ha a kvantor jobb részfájában (ami lényegében a kvantor hatásköre!) nem fordul elő a kvantor által kötött változó, akkor fiktív, törölhető.

Ezt a törlést végzi el a `kv_torol()` fv., amely megkapja egy faelem címét (a kvantorét), s kitörli. Ügyelni kell arra, hogy az oda-vissza láncolás megmaradjon!

```
void Formula::kv_torol(struct faelem *mit)
{
   struct faelem *elozo = mit->vissza;
   if (elozo==NULL)
   {
      Formula::gyoker=mit->jobb;          //itt gyoker az egesz fa gyokere
      Formula::gyoker->vissza=NULL;
   }
   else
   {
      if (elozo->bal==mit) elozo->bal  = mit->jobb;
      else           elozo->jobb = mit->jobb;
      (mit->jobb)->vissza = elozo;
   }
   free(mit->nev); free(mit);
}
```

A változó előfordulását a köv. fv. vizsgálja:

```
int Formula::elofordul(char *valtozo_nev, struct faelem *gyoker, int
elso_hivas)
{
   char str[OPERAND_SZAMA];
   static int megvan_e;

   if (elso_hivas) megvan_e=0;
   if (gyoker)
   {
      if (miez(gyoker->nev)=='p')
      {
       valtozok_gyujt(gyoker->le, str, FIRST_CALL);
       if (strstr(str,valtozo_nev)) megvan_e=1;
      }
      else /* ha nem predikatumszimbolummal van dolgunk */
      {
       elofordul(valtozo_nev, gyoker->bal);
       elofordul(valtozo_nev, gyoker->jobb);
      }
   }//if (gyoker)
   return megvan_e;
}
```

A fv. kap egy változónevet, ill. egy faelem címét, ami számára a részfa gyökerét jelenti. A változót ebben a fában kell keresni. A fv.-nek egy logikai értékkel kell visszatérni. Mivel nem akartam bevezetni egy "globális" változót (azaz a Formula osztályban még egy attribútumot), így egy statikus változót fogunk használni, amelynek az a különlegessége, hogy két hívás között megőrzi az értékét! DE! Ennek a fv.-nek úgy kellene működnie, hogy "tiszta lappal" induljon minden esetben, amikor egy kvantor vizsgálata esetén meghívjuk. Viszont két hívás közt megőrzi az értékét, így ha egyszer egy kvantor esetén az értéke 1 lett, onnantól kezdve ez is marad az értéke. Ezért ha egy kvantorvizsgálat esetén hívjuk meg, mindig ki kellene nullázni. A FIRST_CALL konstans pontosan erre való. A fv. 3. paraméterének van egy alapértelmezett értéke (0, C++ specialitás). Ezt a legelső hívásnál módosítjuk, így a legelső hívásnál a **`megvan_e`** változó kinullázódik, de a rekurzív hívásoknál már ismét az alapértelmezett értékkel fog meghívódni, így a változó értékét nem fogjuk módosítgatni.

Tehát a fát bejárjuk, s kigyűjtjük a változókat. Változó hol lehet? A predikátumok argumentumlistájában. Ezeket kigyűjtjük egy sztringbe, s megnézzük, hogy ebben a sztringben benne van-e a változó (mint karakter). Ha benne van, akkor van találat.

A változókat eme fv. gyűjti ki:

```
void Formula::valtozok_gyujt(struct faelem *akt, char str[], int
kivulrol_valo_hivas)
{
   static int i;

   if (kivulrol_valo_hivas) i=0;
   if (akt)
   {
      if (akt->le)  valtozok_gyujt(akt->le, str);
      if (miez(akt->nev)=='v') str[i++] = (akt->nev)[0];
      if (akt->kov) valtozok_gyujt(akt->kov, str);
   }
   str[i]='\0';
}
```

Ez is rekurzív; ez egy sztringbe gyűjtöget. Kell egy index a sztring feltöltéséhez, amit a legelső híváskor ki kell nullázni. Itt a probléma ill. annak feloldása ugyanaz, mint az előző fv. esetén. Ez kap egy sztringet, s azt módosítja. A fv. befejezése után a változások megmaradnak, ui. C-ben a tömbök cím szerint adódnak át (s itt a sztringet karaktertömbként adtuk át).

```
/* ************************************************************** */
/*    Most jön a kvantorok bevitele                               */
/* ************************************************************** */

void Formula::kv_bevitel()
{
   volt_modositas=0;                     //false;    (globalis valtozo)

   kvantorok_kigyujtese(gyoker);
   kvantorok_probaja();
   lit_kiir(gyoker);   putchar('\n'); getch();
   while (volt_modositas)
   {
      volt_modositas=0;
      osszes_kvantor.init();
      kvantorok_kigyujtese(gyoker);
      kvantorok_probaja();
      if (volt_modositas)
      {
       lit_kiir(gyoker);   putchar('\n');   getch();
      }
   }
}
```

Azaz a fában már nincsenek fiktív kvantorok. Kigyűjtjük az összeset, s megpróbáljuk mindegyiket bevinni a legkisebb hatáskörük felé. Ezt egész addig csináljuk, amíg tudjuk a fát valamilyen módon módosítani. Ha már minden kvantor a helyén van, ill. egy kvantort nem tudunk bentebb vinni, akkor vége, nem tudunk továbblépni.

A Formula osztályban van egy ilyen bővítésünk:

```
int volt_modositas;
Verem nem_tud_lepni, osszes_kvantor;
```

Ezekre itt és most lesz szükség. Egy Verem osztályba tartozó obj. egy láncolt lista, amelyben faelemek címei vannak összeláncolva egyirányban.

```
void Formula::kvantorok_kigyujtese(struct faelem *gyoker)
{
   if (gyoker)
   {
      if (miez(gyoker->nev)=='k')
      {
       osszes_kvantor.berak(gyoker);
       kvantorok_kigyujtese(gyoker->jobb);
      }
      else /* ha nem kvantor */
      {
       kvantorok_kigyujtese(gyoker->bal);
       kvantorok_kigyujtese(gyoker->jobb);
      }
   }//if (gyoker)
}
```

Ha pl. adott a köv. lit. alakú formula: $\nabla y \Delta z \nabla x((fP(x,y)+tR(y))$, akkor először a ∇x -et, majd a Δz -t, s végül a ∇y -t kell megpróbálni bevinni. Azaz: "belülről haladunk kifelé". Ezt hogy érhetjük el? Bejárjuk a fát, s a kvantorok címeit egy VEREMBE (!!!) rakjuk be.

```
void Formula::kvantorok_probaja()
{
   struct veremelem *akt   = osszes_kvantor.getVege();
   struct veremelem *keres = osszes_kvantor.getFej();
   int nem_megy;

   while (!(akt==NULL || volt_modositas))
   {
      nem_megy = nem_tud_lepni.eleme(akt->fa);
      if (nem_megy)
      {
       akt = osszes_kvantor.getElozo(akt);
      }
      else
      {
       megprobal_bevinni(akt->fa);
       akt = osszes_kvantor.getElozo(akt);
      }
   }
}
```

Egy veremben tehát ki van gyűjtve az összes kvantor. Viszont lehet olyan eset, amikor egy kvantort semmiképp sem tudunk bevinni, mert nincs rá szabály. Pl.:

$$\Delta x(G(x)+H(x))$$
$$\nabla x(G(x)*H(x))$$

Ezekben az esetekben a kvantort nem tudjuk bentebb vinni. Ezen kvantorokat (amik már nem vihetők be) egy másik Verem obj.-ban (**nem_tud_lepni**) gyűjtjük ki. A kvantorokat vesszük sorba a veremben. Ha elfogytak, vagy ha volt módosítás a fában, akkor kilépünk a fv.-ből. Ha a kvantor szerepel a **nem_tud_lepni** obj. listájában, akkor azzal nem kell foglalkoznunk, vesszük a köv. veremelemet. Ha nem szerepel, akkor megpróbáljuk bevinni, s vesszük a köv. veremelemet. Ezt ismételgetjük egy ciklusban, amiből tehát akkor lépünk ki, ha nincs további veremelem, vagy ha a fában volt módosítás.

```
void Formula::megprobal_bevinni(struct faelem *kvantor)
{
   struct faelem *akt=kvantor->jobb;
   char betu, log_jel;
   int van_bal_oldalt=0, van_jobb_oldalt=0;

   betu = miez(akt->nev);
   while (!(betu=='l' || betu=='j'))     //'+', '*', 't', 'f'
   {
      akt=akt->jobb;
      betu = miez(akt->nev);
   }
   betu = (akt->nev)[0];
   if (betu=='t' || betu=='f') return;   // ez a helyen van
   if (elofordul((kvantor->bal)->nev, akt->bal,  FIRST_CALL))
      van_bal_oldalt=1;
   if (elofordul((kvantor->bal)->nev, akt->jobb, FIRST_CALL))
      van_jobb_oldalt=1;

   if (van_bal_oldalt + van_jobb_oldalt == 0) kv_torol(kvantor);
   else if (van_bal_oldalt + van_jobb_oldalt == 1)
//vagyis csak az egyik reszfaban fordul elo a kv. altal kotott valtozo
   {
      if (van_bal_oldalt)     bemasol(kvantor, akt->bal);   //mit hova
      else                    bemasol(kvantor, akt->jobb);
      kv_torol(kvantor);
      Formula::volt_modositas=1;
   }
   else  /* if (van_bal_oldalt + van_jobb_oldalt == 2) */
   {
      betu    = (kvantor->nev)[0];
      log_jel = (akt->nev)[0];
      if ((betu=='Δ' && log_jel=='*') || (betu=='∇' && log_jel=='+'))
      {
       bemasol(kvantor, akt->bal);
       bemasol(kvantor, akt->jobb);
       kv_torol(kvantor);
       Formula::volt_modositas=1;
      }
      else
      {
       nem_tud_lepni.berak(kvantor);
       Formula::volt_modositas=1;
       return;    //kulonben nem tudunk vele most mit kezdeni
      }
```

```
   }
}
```

Megkapjuk egy kvantor címét, ezt kellene bevinni. A kvantor jobb oldali részfájában addig megyünk lefele, amíg nem találunk egy logikai összekötő jelet vagy egy spec. jelet (**t** vagy **f**). Ha ez utóbbit találunk, akkor a kvantor már a legszűkebb hatáskörében van, nincs mit tennünk.
Különben, ha "+"-ot vagy "*"-ot találunk, akkor a köv. szabályokat (törvényeket) kell alkalmaznunk:

$$Qx(E \circ G(x)) \sim E \circ QxG(x), \quad \text{ahol } x \notin Fv(E),\ Q \in \{\forall,\exists\},\ \circ \in \{+,*\}$$

$$\Delta x(G(x)*H(x)) \sim \Delta xG(x) * \Delta xH(x)$$
$$\nabla x(G(x)+H(x)) \sim \nabla xG(x) + \nabla xH(x)$$

Tehát meg kell nézni, hogy a "+" vagy "*" bal ill. jobb oldali részfájában előfordul-e a kvantor által kötött változó.
Ha egyik oldalt sem fordul elő, akkor a kvantor fiktív. Mivel korábban már korábban ezeket eliminátuk, így ez az eset nem fog előfordulni. (Meg lehetett volna azt is csinálni, hogy csak itt szedjük ki a fiktív kv.-okat).
Ha az egyik oldalt előfordul, a másikon viszont nem, akkor bevisszük oda, ahol előfordult, s a kv.-t töröljük.
Ha mindkét oldalt előfordul, akkor vizsgálat a legutóbbi két szabály alapján. Ha valamelyiknek eleget tesz, akkor mindkét oldalra berakjuk a kvantort, s töröljük az eredeti helyéről. Ha a szabálynak nem tesz eleget, akkor nem lehet bevinni, így ezt a "tiltott" kvantorok közé fel is vesszük, így legközelebb már meg sem próbáljuk bevinni.

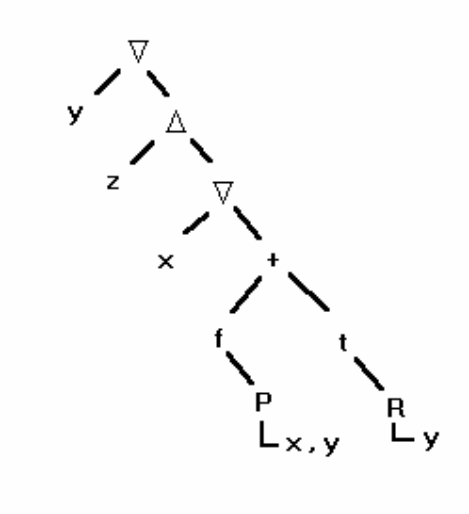
```
void Formula::bemasol(struct faelem *kvantor, struct faelem *hova)
{
   char betu;
   struct faelem *uj_kv, *uj_valtozo, *akt=hova, *elozo;
   uj_kv      = (struct faelem *) malloc(sizeof(struct faelem));
   uj_valtozo = (struct faelem *) malloc(sizeof(struct faelem));
   if (!uj_kv || !uj_valtozo) hiba();

   *uj_kv = *kvantor; *uj_valtozo = *(kvantor->bal);
   uj_kv->nev = strdup(kvantor->nev);
   uj_valtozo->nev = strdup((kvantor->bal)->nev);
   uj_kv->bal = uj_valtozo; uj_valtozo->vissza = uj_kv;
   uj_kv->jobb = akt; uj_kv->vissza=akt->vissza;

   elozo = akt->vissza;
   if (elozo->bal == akt) elozo->bal=uj_kv;
   else                   elozo->jobb=uj_kv;
   akt->vissza=uj_kv;
}
```

Ez a fv. megkapja a beszúrandó kvantor címét, ill. ama faelem címét ("+" vagy "*"), amely alá balra vagy jobbra be kell másolni.
A kvantorról készítünk egy másolatot (új tárterület foglalása), s a mutatókat sorra beállítjuk.

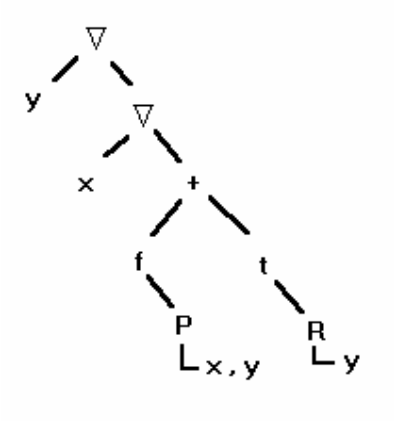
A program működése a korábbi példánál maradva:

fiktív kvantorok eliminációja:

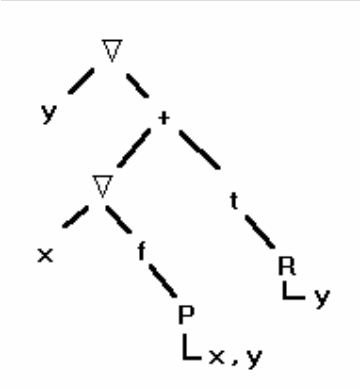
elimináció előtt:



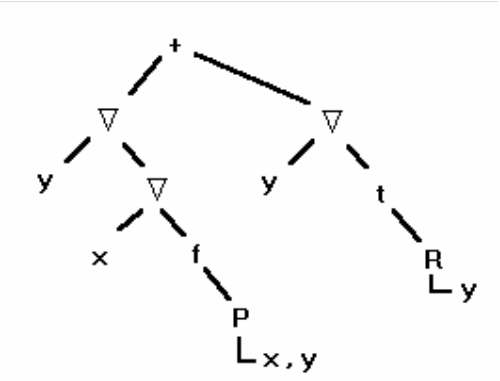
elimináció után:



∇_x bevitele:



∇_y bevitele:



Futási eredmény:

```
a formula:
Ey(Vz(Ex(P(x,y)>R(y))))

fa felepitese OK
(Ey(Vz(Ex((P(x,y))>(R(y))))))
a fv.-ek es/vagy pred.-ok operandusainak tipusa(i) OK

Milyen jelet rak ele? ("t" vagy "f"): t

Literalla alakitva?
∇y∆z∇x((fP(x,y)+tR(y)))

∇∆
Fiktiv kv.-ok eliminalva:
∇y∇x((fP(x,y)+tR(y)))

A kv.-bevitel lepesei (legszukebb hataskor, billentyuleutesre tovabblep):
∇y((∇xfP(x,y)+tR(y)))
(∇y∇xfP(x,y)+∇ytR(y))
```