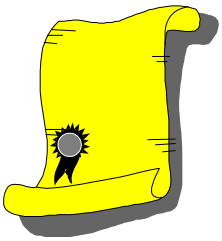


Пловдивски Университет
«Паисий Хилендарски»
Факултет по математика и информатика
Катедра Компютърни системи

Дипломна работа

ТЕМА:

«Един подход за описание на геометрична информация»

Дипломант:
Александър Пламенов Пенев
Факултетен номер: 921106

Научен ръководител:
гл.ас. д-р Д. Димов

Пловдив
1996 г.

Увод

Настоящата дипломна работа разглежда проблема за описание на геометрична информация на тела от реалния свят, чрез дефиниране на свойства на информационни обекти. Предложен е и е реализиран език за описание на геометрична информация. Програмната реализация е на Borland Pascal 7.0.

Дипломната работа се състои от увод, четири части, приложения и съдържание. ***Част I*** е кратко въведение в предметната област на проблема, задачите за решаване и целите на работата. В ***Част II*** са описани теоретичните основи за решаване на задачата. ***Част III*** е описание на програмната реализация. В ***Част IV*** се прави заключение за постигнатото от дипломната работа и възможностите за разширения и подобрения.

Към дипломната работа са приложени: изходните текстове на програмната реализация, примерни програми на дефинирания в тази работа език и изображенията получени от тях.

Част I

Въведение

1.1. Въведение в проблемите

Компютърната графика се развива като самостоятелна научна дисциплина от сравнително дълго време. Нейни главни задачи са генерирането, обработката и анализа на визуални образи с помощта на компютър.

Приложенията на компютърната графика са многобройни и са в най-разнообразни области от живота - от проектирането на сгради и автомобили до развлеченията с все по-впечатляващите във визуално-техническо отношение филми и компютърни игри.

Генерацията на едно изображение изисква предварителното описание на телата участващи в него. Този проблем се свежда до намирането на представяне на модел тяло (на част от реалният свят).

1.2. Постановка на задачата

Съществуват много и разнообразни решения за представянето на телата от реалния свят във вид удобен за компютъра.

Най-известните от тях са:

- гранично представяне - описващо телата като съвкупност от пространствени многоъгълници, определящи границите на телата;
- конструктивна геометрия - на базата на т. н. примитиви (тела като сфера, цилиндър, полупространство и др.) и краен брой

теоретико множествени операции над тях се описват тела с различна форма;

- изброяване на заетото пространство - пространството се разделя на краен брой сектори (паралелепипеди за тримерното пространство) и всяко тяло се описва като списък от секторите, с които то има сечение;
- замитане (sweeping) - телата се получават чрез постъпателни и/или въртеливо движение на тела с по-ниска размерност. Например цилиндърът може да се разглежда като обединение на всички кръгове при движението на основата му перпендикулярно на равнината си.

Всички тези решения с различна степен на явност кодират геометричната информация.

Задачата на дипломната работа е да постави началото на изследвания в насока явно представяне и използване на геометричната информация.

1.3. Цели и същност на дипломната работа

Цел: Да се дефинира език за описание на геометрична информация.

Цел: Да се направи връзка със съществуващ алгоритъм за визуализация (описан в [1]).

Много от най-добрите методи за обработка (в частност визуализация) на геометрична информация са много ресурсоотнемащи и изискващи голяма изчислителна мощност отстрана на компютъра.

Цел: Да се разгледат възможностите за прилагане на клиент-сървър подхода за работа на много потребители в обща графична система.

По своята същност дипломната работа е изследователска и не си поставя за цел преки практически резултати.

Част II

Теоретични основи

2.1. Основни понятия

Ще бъдат дефинирани понятията, използвани в дипломната работа, така както ще бъдат разбирани. Някои от дефинициите са за всеизвестни понятия и се препокриват с всеобщите схващания. Други дават представа за нови понятия свързани с разглежданата област.

2.1.1. Обект. Класове от обекти

def: **Обект** се нарича всичко, което може да се характеризира еднозначно с набор от свойства (атрибути).

Обектите се разглеждат като неделими, въпреки тяхната евентуална вътрешна структура. Два обекта се считат за неразличими или равни, ако всичките им свойства съвпадат.

Всяко свойство има име и стойност. Името е различно за различните свойства. Обикновено то носи семантична информация за свойството, например: ’цвят’, ’тегло’,... Стойността на свойството е също обект. Тя може да е известна или неизвестна. Ако свойствата на един обект са с известна стойност, то той се нарича единичен. Ако поне едно от свойствата на обект е с неизвестна стойност, обектът се нарича абстрактен обект. Всеки абстрактен обект е множество от единични обекти.

Множеството от всички обекти се нарича пространство на обектите и се означава с $\mathfrak{I}$.

Един частен случай на обект е обектът $\mathrm{nil} \in \Im$, който няма нито едно свойство, т. е. множеството от свойствата му е $\varnothing$.

Важен вид обекти са информационните обекти. Те са отражение на обекти от реалния свят, т. е. свойствата им съвпадат с част от свойствата на реалните обекти. При това свойствата могат да бъдат обективно или субективно отразени. Информационните обекти се използват за моделиране на света.

def: **Клас** от обекти се нарича всяко множество от обекти, групирани в него на базата на общи свойства.

Група от обекти, причислени към даден клас, придобиват едно общо свойство, а то е свойството за принадлежност към този клас. Това означава, че може да има клас от обекти, които предварително нямат общи свойства.

Един обект може да принадлежи към повече от един клас. Тогава той получава по едно свойство за принадлежност към всеки от класовете.

2.1.2. Списък

def: **Списък** се нарича всяко крайно линейно наредено множество L от наредени двойки (номер,обект)$\in \mathrm{N} \times \Im$. Наредбата се определя от първият елемент на наредената двойка.

Списъкът е динамична структура. Броят на елементите му не е предварително известен и не е фиксиран. Той може да се изменя и след създаването на списъка.

Основни операции над списъците са: създаване на нов списък, добавяне на обект (в края на списъка), извличане на обект по поредния му номер, премахване на обект с определен пореден номер, унищожаване на списък.

- нов: генерира се празен списък, т.е. $\mathrm{L} := \varnothing$;

- добавяне на обекта Obj в списъка: $L := L \cup \{(|L|+1,Obj)\}$;
- извличане на $i^{-ти}$ обект от списъка: втория елемент Obj на наредената двойка $(i,Obj)\in L$;
- премахване на $i^{-ти}$ обект от списъка:

$$L := \left(\bigcup_{\substack{j=1 \\ (j,x_j)\in L}}^{i-1} \{(j,x_j)\} \right) \bigcup \left(\bigcup_{\substack{j=i+1 \\ (j,x_j)\in L}}^{|L|} \{(j-1,x_j)\} \right);$$

За повече информация относно списъците може да бъдат ползвани [2] и [4].

2.1.3. Израз и функция

def: **Изображение** на $M_1\subseteq\Im$ в $M_2\subseteq\Im$ се нарича множеството ξ от наредените двойки $(m_1,m_2)\in M_1\times M_2$ за $\forall m_1\in M_1$.
Означение: $$\xi: M_1 \to M_2$$ Елементите $m_2\in M_2$ се наричат образи, а съответните им елементи $m_1\in M_1$ - първообрази.

Някои основни видове изображения са сюрективните, инективните и биективните. Сюрекция се нарича изображение, при което на всеки елемент от M_2 съответства поне един първообраз. Инекция се нарича изображение, при което всеки елемент от M_2 има не повече от един първообраз. Когато едно изображение е и сюрекция и инекция то се нарича биекция.

За всеки образ $y\in M_2$, може да бъде дефинирано множество от неговите първообрази: $\xi^{-1}(y)=\{x\in M_1|\xi(x)=y\}$. То се нарича пълен първообраз. Когато изображението е биекция това множество е едноелементно за всеки образ. В този случай съществува така нареченото обратно изображение на ξ и то се означава също с ξ^{-1}.

Важен вид изображения са функциите.

def: Функция се нарича изображение F, при което на всеки обект от множество на първообразите $M_1 \subseteq \Im$ се съпоставя точно един обект от множество на образите $M_2 \subseteq \Im$.

Означение: $y=F(x)$, където $x \in M_1$, $y \in M_2$.

Означение: когато $M_1=K_1 \times K_2 ... \times K_t$, $K_i \subseteq \Im$: $y=F(x_1,x_2,...,x_t)$, където $x_i \in K_i$ за $i=1..t$, $y \in M_2$. Тук $x_1,x_2,...,x_t$ се наричат аргументи, y - резултат на функцията, а F - име на функцията.

Основна операция над функциите е изчисляването им за определени фиксирани аргументи. Това става чрез намиране на наредената двойка (Първообраз,Образ), за която Първообраз съвпада със стойностите на аргументите и като резултат на функцията се взема елемента Образ.

На практика функциите не се задават с множеството от наредените двойки (Първообраз,Образ), а се описват с някакво правило, определящо как от аргументите да получим резултата. В този случай изчисляването може да се изпълни на три етапа:

- Предаване на аргументите към функцията;
- Прилагане на правилото;
- Изискване на резултата от функцията.

Тези етапи са логически и не е задължително да се спазват стриктно. Например вторият и третият могат да се обединят в един.

Частен случай за задаване на правилото за получаване на резултата от аргументите е израза.

def: **Израз** се нарича:

1) константа - единичен обект;

2) променлива - идентификатор (име) евентуално свързан с някакъв обект;

3) суперпозиция - функция с аргументи изрази.

Нека е въведен оператора, изчисляващ стойността на израза, calc(израз) по следния начин:

calc(c)=c, за ∀израз-константа c;

calc(v)=обекта свързан с v, за ∀израз-променлива v;

calc(f(e_1,e_2,...,e_n))=f(calc(e_1),calc(e_2),...,calc(e_n)), за ∀израз-суперпозиция.

Тогава изчисляването на функцията f, зададена чрез израза expr, става по следния начин:

- Предаване на аргументите към функцията - свързване на променливите в израза expr със съответните им аргументи на функцията;
- Прилагане на правилото - $f_{резултатно}$=calc(expr);
- Изискване на резултата от функцията - $f_{резултатно}$.

2.1.4. Координатна система

def: Координатна система върху множество M⊆ℑ се нарича еднозначната и обратима функция: $$sys: M \leftrightarrow R^n$$ която изобразява ∀x∈M в (x_1,x_2,...,x_n).
Тук x_1,x_2,...,x_n се наричат координати на обекта x в множеството M, а обекта x - точка относно координатната система sys.

Този вид функции задават имена на обектите в M. Имената са наредените n-торки от R^n. Когато е известно името на един обект name=(x_1,x_2,...,x_n), то съответният му обект е x=sys^{-1}(name)= sys^{-1}((x_1,x_2,...,x_n)). Обратно, за всеки обект може да се намери неговото име name=sys(x).

Във връзка с координатната система в множеството M, някои обекти имат специални названия. Обекта O=sys^{-1}((0,0,...,0)) се нарича център на координатната система.

2.1.5. Метрика

def: Метрика в $K \subseteq \Im$ се нарича всяка функция

$$\rho: K \times K \rightarrow R^{+}$$

за която са изпълнени следните три условия:

1) $\rho(x,y) \geq 0$, и $\rho(x,y)=0 \Leftrightarrow x=y$;

2) $\rho(x,y)=\rho(y,x)$;

3) $\rho(x,y) \leq \rho(x,z)+\rho(z,y)$

за $\forall x,y,z \in K$

Функциите-метрики определят доколко два обекта от един клас се различават в някакво отношение, зададено чрез метриката. Колкото стойността на функцията за два обекта е по-голяма, толкова по-различни са те или се казва, че са на по-голямо разстояние един от друг.

2.1.6. Пространство

def: **Метрично пространство** се нарича наредената двойка (K,ρ), където $K \subseteq \Im$ е произволно множество, а ρ е метрика.

Ако в същото множество K бъде въведена друга метрика ρ_1, то се получава ново метрично пространство (K,ρ_1).

def: Пространство $\mathbf{E^n}$ се нарича метричното пространство R^n с:

1) метрика $\rho(x,y)=\sqrt{\sum_{i=1}^{n}(x_i-y_i)^2}$;

2) координатна система $sys(x)=x$

за $\forall x,y \in R^n$

За нуждите на дипломната работа е необходимо е да бъде избрано едно пространство за базово (основно), което да е с добре познати свойства. Такова се явява E^n.

Структурата на множество R^n е подходяща за лесно въвеждане на една основна координатна система. Това става чрез функцията идентитет. В това множество всички други възможни координатни системи могат да бъдат разглеждани и като функции изобразяващи точките относно съответната координатна система в точките относно основната.

Под ε-околност на точка P се разбира множеството $A=\{x\in E^n|\rho(A,x)\leq\varepsilon\}$.

2.1.7. Точково множество

def: **Точково множество** се нарича всяко подмножество $A\subseteq E^n$

Най-важната операция за едно точково множество е проверката, дали една точка от пространството принадлежи и на даденото множество, т. е. предиката за принадлежност. Този предикат определя напълно множеството. Чрез него точките във всяко такова множество могат да бъдат разделени на три вида: вътрешни - за които съществува околност, която съдържа само точки на множеството; външни - за които съществува околност, която съдържа само точки не принадлежащи на множеството; гранични - всички останали.

Други възможни операции са:

- най-близка точка от множеството до дадена точка от пространството;
- нормален вектор в точка от границата на множеството;
- най-близка точка по направление;
- дали една точка е вътре, на границата или вън от множеството;
- и др.

2.1.8. Геометрична информация

def: **Геометрична информация** се нарича множеството от следните три групи свойства на обекта:

1) пространствени форми $\{s_1,s_2,...,s_t\}$;

2) метрически характеристики $\{m_1,m_2,...,m_q\}$;

3) местоположение и ориентация $\{p_1,p_2,...,p_r\}$.

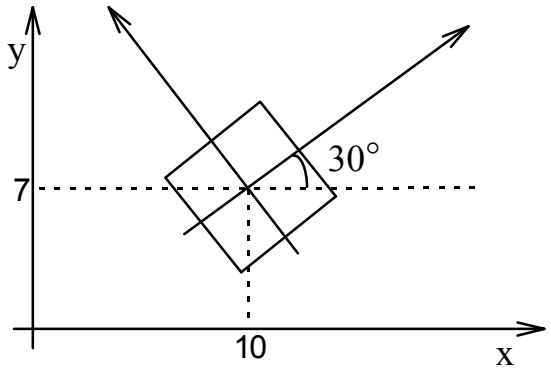
Всяка геометрична информация индуцира (определя, поражда) точково множество в E^n (или точкови множества).

Пространствените форми s_i представляват класове на еквивалентност в съвкупността от всевъзможните точкови множества в пространството. Пример за пространствени форми са: сфера, цилиндър, куб и т. н. Тези свойства описват формата на точковото множество.

Метрическите характеристики определят размерите на множеството. Те могат да се разделят на два вида: локални и глобални. Локалните задават стойностите си за всяка точка от множеството, а глобалните за множеството като цяло. Пример за глобални метрически характеристики са: обема, повърхността, центъра на тежестта и диаметъра на множеството.

Точковите множества могат да заемат различни положения в пространството, затова свойствата за местоположение и ориентация се избират така, че да характеризират една координатна система в E^n. Тази координатна система се нарича локална за множеството и формите се описват спрямо нея.

Ето един пример за геометрична информация индуцираща квадрат в E^2.



фиг. 1

g = {{квадрат},{5},{10,7,30}}

Геометричната информация не определя еднозначно едно точково множество, дори всички параметри да са фиксирани. В примера не става ясно дали се индуцира квадрата или контура му. При наличието на повече от една форма възможностите са още повече. Това налага да се определи точно как от геометричната информация се получава точковото множество. Това се постига чрез т. н. канонична геометрична информация. Тя е еднозначна, защото е определен начина по който може да се получи точково множество.

def: **Канонична геометрична информация** за обект се нарича геометрична информация на която:

1) $t=1$ т.е. зададена е само една пространствена форма $\{s\}$, и тя е определена чрез функцията $F(X)$, задаваща ориентираното разстояние от точка X до границата на множеството;

2) за точките на множеството предиката $F(X)\leq 0$ се удовлетворява;

3) местоположението и ориентацията са зададени чрез функция sys - координатна система в пространството, в което информацията поражда точковото множество.

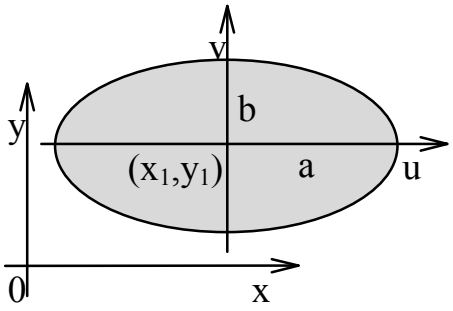
Когато е дадена канонична геометрична информация $g_k=\{F(X),\{m_1,m_2,...,m_q\},sys(Z)\}$, то породеното точково множество е $A=\{X\in E^n \mid F(X)\leq 0\}$.

Функцията $F(X)$ описва формата спрямо координатната система на E^n. Удобно е обаче формата да бъде описвана спрямо локална за множеството координатна система, каквато се явява sys. Центърът на локалната координатна система се нарича полюс на множеството. Той може да не принадлежи на множеството.

Нека $f(Z,m_1,m_2,...,m_q)$ е функция определяща точково множество в локалната координатна система sys. Тъй като sys е обратима по определение, то

$$\exists sys^{-1} \Rightarrow F(X)=f(sys^{-1}(X),m_1,m_2,...,m_q).$$

Пример: Да се опише каноничната геометрична информация в пространството E^2 на елипса с полуоси a и b, център (x_1,y_1) и ъгъл на завъртане $0°$:



фиг. 2

Формата описана спрямо локалната координатна система е $f((u,v))=\frac{u^2}{a^2}+\frac{v^2}{b^2}-1.$

Координатната система е $sys((u,v))=(u+x_1,v+y_1)$, а $sys^{-1}((x,y))=(x-x_1,y-y_1)$.

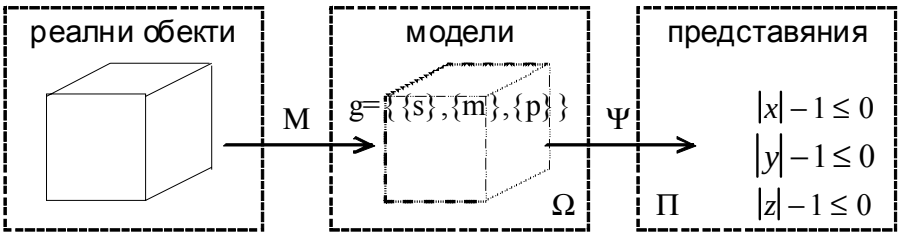
Следователно

$$g_k=\{F((x,y))=\frac{(x-x_1)^2}{a^2}+\frac{(y-y_1)^2}{b^2}-1,\ \{a,b\},\ sys((u,v))=(u+x_1,v+y_1)\}$$

За повече информация относно геометричната информация може да бъде ползвана [5].

2.1.9. Представяща схема

Описанието на реалния свят в компютрите може да се раздели на два етапа: моделиране(M) и представяне(Ψ). При първият от реалния обект се получава модел, а при втория на модела се съпоставя конкретно елемент от множеството на представянията Π.



фиг. 3

Пространството Ω е множеството на моделите, а Π - на представянията. Трябва да се прави разлика между процеса представяне (наречен още представяща схема) и всеки елемент от пространството Π, който също се нарича представяне.

def: Представяне се нарича множеството от конвенции за това, как да бъде описан един клас от обекти. Представянето използва конвенциите за описването на даден обект.
Компоненти на представянето: 1) Лексика - символите разрешени в представянето; 2) Структура - описва условията за групиране на символите; 3) Процедура - специфицира процедурите за достъп до описанието (създаване, модифициране, отговори); 4) Семантика - остановява по какъв начин и какво значение ще се предаде на структурите.

Много често символите в лексиката са от една крайна азбука. Тогава структурата се задава от граматика, която може да генерира Π.

def: **Представяща схема** се нарича функцията

$$\Psi: \Omega \rightarrow \Pi$$

изобразяваща моделите на реалните тела от пространството Ω в определено представяне в пространството Π.

Представящата схема извършва един вид именуване на моделите, като на всеки модел съпоставя негово име - един елемент от Π. Функцията Ψ е сложен процес и обикновено се извършва от човека.

Функцията Ψ^{-1} е семантиката на представянето.

2.1.10. Тела. Абстрактни тела

Понятието тяло е един вид обобщение на понятието геометрична информация.

def: **Тяло** се нарича обект, за който свойствата определящи геометричната му информация са определени еднозначно. Обекта може да притежава и допълнителни свойства.

Допълнителните свойства обикновено дават информация за някои физически характеристики на обектите от реалният свят. Това е необходимо за по точното моделиране на света.

Като пример за такива свойства могат да бъдат посочени цвета, прозрачността, якостта и масата на предметите. Някои от тях, като цвета е добре да бъдат дефинирани за всяка точка от тялото.

def: **Абстрактно тяло** се нарича обект, за който свойствата определящи геометричната му информация не са определени напълно (еднозначно).

Абстрактните тела представляват абстрактни обекти. Те са подходящи за създаване на библиотеки от готови тела. На тяхна база могат лесно да се дефинират конкретни тела, като се конкретизират всички свойства. Също така от тях могат да се получават други абстрактни тела, като се конкретизират само някои свойства или се добавят нови.

Например от абстрактното тяло сфера може да се получи ново абстрактно тяло сфера с радиус едно или конкретно тяло сфера с даден радиус и център.

2.1.11. Графична виртуална машина

def: **Виртуална машина** се нарича мислен (въображаем) или програмен аналог на физически съществуваща машина (или проект на такава).

Виртуалните машини са много по-гъвкави от физически реализираните. Те могат да бъдат променяни много по-бързо и "изработката" им е по-лесна и по-евтина. В последствие, когато машината се окаже полезна, може някои нейни компоненти (функции) да бъдат физически реализирани в специализирани модули. Цялостната реализация е нецелесъобразна и се прилага само в много редки случаи.

За средствата на компютърната графика е възможно да бъде реализирана графична виртуална машина.

def: **Графична виртуална машина** се нарича виртуална машина притежаваща следните възможности:

1) работа в пространство със зададена размерност;
2) описание на тела и абстрактни тела;
3) получаване на данни за дефинираните тела.

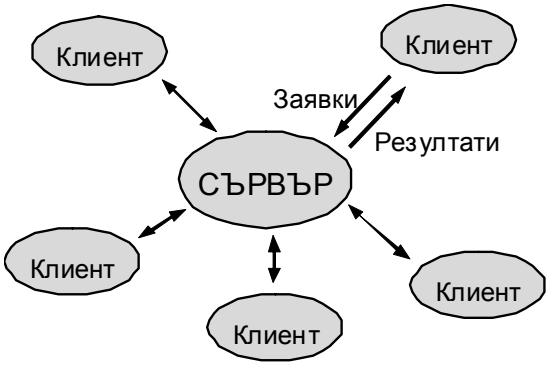
Графичната виртуална машина дава абстрактна работна среда, на един потребител, за дефиниране на тела и получаване на техните свойства. Това може да става в пространство с предварително избрана размерност или в помощни пространства евентуално с по-малки или по-големи размерности. Във всяко пространство съществува една текуща координатна система и една функция метрика. Те могат да бъдат произволно променяни.

Машината притежава вътрешно състояние, което се изменя в процеса на работа на потребителя с нея. То съдържа и всички вече дефинирани тела.

Управлението се извършва с помощта на специализиран език. Удобно е той да бъде обектно-ориентиран.

2.1.12. Клиент-Сървър организация

Този вид организация на работата се характеризира с два типа обекти влизащи в определени взаимоотношения: т. н. клиенти и сървър. Обменът на данни (и в частен случай - команди) се осъществява между един клиент и сървъра. На фиг. 4 е показана принципната схема.



фиг. 4

Клиентите изпращат заявки към сървъра, а той ги изпълнява и връща резултатите. Сървърът организира така работата си, че за всеки клиент “мисли”, че е единствен.

def: **Сървър** се нарича абстрактен (или физически) изпълнител на определен брой услуги.

Услугите са сложни последователности от действия, които постигат желаната от клиента цел.

Понятието сървър е поначало абстрактно, т. е. не е свързано с някаква конкретна реализация на система клиент-сървър, но обикновено такива системи се разполагат върху компютърни мрежи с множество компютри. Сървърът физически е мощен компютър(и), клиентите са други възли в мрежата, а заявките и резултатите се реализират като мрежова комуникация.

def: **Клиент** се нарича всеки обект, използващ услугите на даден сървър.

Един клиент може да комуникира и с повече от един сървър. Физически клиентите могат да са компютри, процеси, задачи. Възможно е клиентите и сървъра да са само на един компютър.

Компютърната графика е област изискваща големи компютърни мощности. Не всеки може да има на работното си място суперкомпютър.

Клиент-сървър подхода може да бъде приложен и при компютърната графика. Така много потребители (клиенти) работят едновременно с един графичен сървър, който поддържа за всеки по една графичната виртуална машина. Езика за връзка е този на виртуалната машина, разширен с няколко нови команди за регистрация на нови клиенти.

Така графичният сървър може да бъде физически разположен на един или повече мощни компютъра. Поради това че някои от най-

добрите алгоритми за визуализация са паралелни, тези компютри могат да бъдат многопроцесорни.

2.2. Цели и решения

Ще бъдат описани подцелите необходими за решаване на главната цел на дипломната работа, начините по които те трябва да се съчетаят, както и методите за решаването им.

2.2.1. Обекти

Подцел 1: Представяне и използване на единични информационни обекти.

Нека е даден единичен обект O със свойства $S_1,S_2,...,S_t$. Тогава O се представя като функция без аргументи и с резултат - списък от функции-свойства S_i.

Когато се наложи изчисляване на стойността на дадено свойство S, то в списъка се намира първата функция с име съвпадащо с името на S. Като стойност на свойството се получава резултата от изчислението на намерената функция с аргументите на свойството.

Подцел 2: Представяне и използване на абстрактни информационни обекти.

Абстрактните обекти могат да се разглеждат като обекти с параметри. Параметрите влияят на стойностите на свойствата. При задаване на конкретни стойности става доопределяне на неизвестните свойства.

Нека е даден абстрактен обект $A(p_1,p_2,...,p_k)$ със свойства $S_1,S_2,...,S_t$. Тогава O се представя като функция с аргументи $p_1,p_2,...,p_k$ и с резултат - списък от функции-свойства S_i.

От този абстрактен обект може да се получи друг абстрактен обект $A_1(p^1_1,p^1_2,...,p^1_m)$, чрез наследяване (виж подцел 3). При пълна конкретизация на параметрите на A се получава единичен обект.

Подцел 3: Наследяване на свойствата на обектите при дефиниране на нов обект.

Под наследяване на свойствата се разбира един обект да има не само нови свойства, но и да съхрани свойствата на друг обект, наречен негов предшественик или родител.

Нека е даден произволен обект X - родител. Тогава за да се дефинира наследник Y на X е необходимо към списъка от свойства на Y да се долепи в края списъка от свойствата на X. По този начин, ако има свойства с еднакви имена и в Y и в X, то се изчислява новодефинираното в Y, защото се изчислява първото срещнато свойство в списъка.

2.2.2. Израз и функция

Подцел 4: Описание на функции и изрази.

Функциите могат формално да бъдат разделени на два класа: вградени и потребителски. Вградените са тези които са дефинирани от системата и тяхното изчисляване става по известен за нея алгоритъм. На тяхна база в процеса на работа се дефинират потребителските функции. Това става като се опишат името на новата функция, имената на аргументите є (формалните є параметри) и израза, по който тя се изчислява. В този израз участват константи, променливи (и имена на аргументи, които също са променливи), вградени функции и други вече дефинирани потребителски функции.

За всяка функция се дефинират: приоритет и асоциативност. Те определят начина, по който функцията ще се записва в изразите.

Приоритета дава възможност за писане на изрази без много ненужни скоби. Той въвежда неявен ред за изчисляване на израза. Например 1+2*3 се изчислява като 1+(2*3), а не като (1+2)*3.

Асоциативността определя две неща. Първо - къде се пише името на функцията спрямо аргументите: префиксно, постфиксно, инфиксно или функционален запис (за по-подробна информация относно видовете функции виж [2]). Второ - за инфиксните функции, в какъв ред се изчисляват части от изрази съдържащи само този оператор. Например: 1+2+3 се пресмята като (1+2)+3, а не като 1+(2+3).

Потребителските функции е позволено да бъдат само във функционален запис.

Подцел 5: Изчисляване на функции и изрази.

Нека е даден изразът expr и списък от параметри a_param, съдържащ стойностите на променливите участващи в expr. Тогава за да бъде изчислен expr над него се прилага операторът calc (виж 2.1.3. Израз и функция).

Когато трябва да бъде изчислена променлива, то нейната стойност се търси в списъка a_param.

Функциите се изчисляват на два етапа:

- Създаване на нов параметричен списък a_param_1, който представлява a_param с добавени в началото променливи с имената на формалните параметри на функцията и стойности получени от изчисляването на подизразите на функцията;
- Пресмятане на стойността и получаване на резултата. За потребителските функции това става, чрез изчисляване на израза е при използването на a_param_1.

2.2.3. Необходими вградени функции

Подцел 6: Да се определят вградените функции и алгоритмите, по които да се пресмятат те.

Таблица 1 дава информация за необходимите вградени функции. Функциите с префиксен, постфиксен и инфиксен запис, ще бъдат наричани още оператори. По-големите стойности на приоритета определят по-висок приоритет на функцията (оператора). В полето Асоциативност символа **f** означава мястото на оператора спрямо аргументите си; **x** означава аргумент-подизраз, който съдържа в себе си оператори с по-висок приоритет; **y** означава аргумент-подизраз, който съдържа в себе си оператори с по-висок или равен приоритет на приоритета на текущият оператор; **a** означава произволен аргумент. Полето Коментар дава кратко описание за начина на изчисление на функцията.

Таблица 1

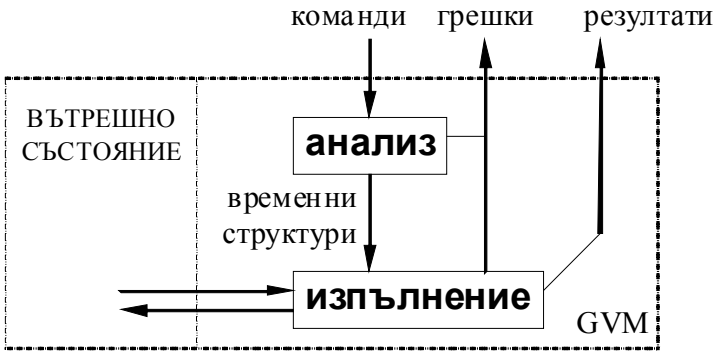
Име	Приоритет	Асоциативност	Пример	Резултат	Коментар
new	1	fx	new f(x)=x-1	≠0 - Да	запомня новата ф-я
get	1	fx	get 1+1.5	2.5	идентитет
del	1	fx	del f(x)	≠0 - Да	премахва функция
property	1	fx	property f()=1	свойство	дефинира свойство
=	2	xfx	f(x)=2^x	функция	дефинира функция
&	5	yfx	[1,2,3]&[4,5]	[1,2,3,4,5]	слепване на списъци
.	8	yfx	[3,2,1,0].3 obj.s(1)	1	елемент от списък изчислява свойство
~	8	fx	~(1+1)	-2	унарен минус
-	5	yfx	1-2 [1,2,3]-[1,0,0]	-1 [0,2,3]	разлика на числа разлика на вектори
+	5	yfx	1+2 [1,2]+[1,2]	3 [2,4]	сума на числа сума на вектори
*	6	yfx	2*6 [1,2,3]*[1,2,0]	12 6	произведение скаларно произвед.
/	6	yfx	1/2	0.5	частно
mod	6	yfx	4 mod 3	1	остатък от деление

div	6	yfx	4 div 3	1	целочислено деление
^	7	yfx	2^3	8	степенуване
!	7	yf	10!	3628800	факториел
not	8	fy	not (1==1)	0	отрицание (≠0 - Да)
and	6	yfx	(1<2) and (2>1)	1	конюнкция (≠0 - Да)
or	5	yfx	(1<0) or (0<1)	1	дизюнкция (≠0 - Да)
xor	5	yfx	(2=2) xor (1=1)	0	изключващо или
==	4	xfx	1==1	1	равно (сравнение)
<>	4	xfx	1<>1	0	различно
>=	4	xfx	1>=1	1	по-голямо или равно
<=	4	xfx	1<=2	1	по-малко или равно
>	4	xfx	1>1	0	по-малко
<	4	xfx	1<2	1	по-голямо
odd	-	f(a)	odd(2) odd(3)	0 1	нечетно
sqr	-	f(a)	sqr(3)	9	квадрат
sqrt	-	f(a)	sqrt(2)	1,41421...	квадратен корен
sin	-	f(a)	sin(pi())	-1	синус
cos	-	f(a)	cos(0)	1	косинус
arctan	-	f(a)	arctan(0)	0	аркустангенс
pi	-	f()	pi()	3.14159...	π
exp	-	f(a)	exp(1)	2,71828...	експоненциал (e^x)
ln	-	f(a)	ln(2)	0,69314...	логаритъм ($\log_e x$)
abs	-	f(a)	abs(1) abs(-1)	1 1	абсолютна стойност
sgn	-	f(a)	sgn(10) sgn(-1.5)	1 -1	сигнум
int	-	f(a)	int(1.234)	1	цяла част
round	-	f(a)	round(1.234) round(1.678)	1 2	закръгляне до цяло
frac	-	f(a)	frac(1.234)	0.234	дробна част
random	-	f()	random()	0.546785	случайно число

			random()	0.243428	в интервала [0..1)
if	-	f(a,a,a)	if(1<2,4,7)	4	условен израз
case	-	f(a,a,...)	case(2,5,6,7) case(0,5,6,7)	7 5	избор
min	-	f(a,...)	min(2,1,3)	1	минимум
max	-	f(a,...)	max(3,1)	3	максимум

2.2.4. Език на графичната виртуална машина

Графичната виртуалната машина (виж принципната схема на фиг. 5) интерпретира командите последователно една след друга. Всяка команда се анализира, изпълнява и резултата от изпълнението се извежда извън машината. Ако при анализа или изпълнението се получи грешка, то това се сигнализира.



фиг. 5

Командите трябва да бъдат правилни изречения на езика за дефиниране на геометрична информация, описан по нататък.

Подцел 7: Да се определят лексиката, синтаксиса и семантиката на езика за дефиниране на геометрична информация.

2.2.4.1. Азбука

Думите на езика са съставени от буквите на крайната азбука Σ съдържаща символите съответстващи на кодовете от 0 до 255 от ASCII[†] таблицата.

2.2.4.2. Лексика

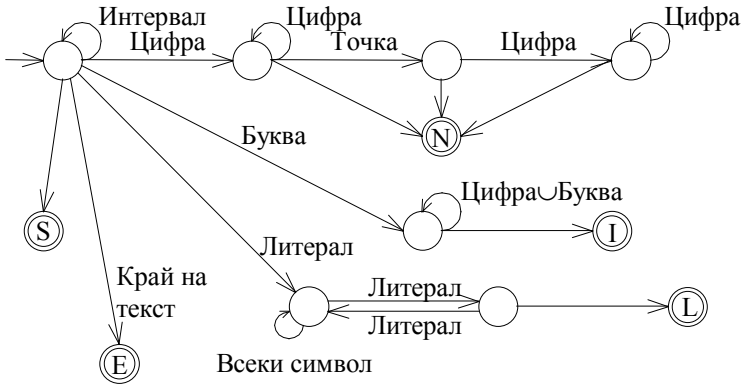
Лексикалният анализатор отделя от входния текст следните типове лексеми: край на текста, идентификатор, низ, число, символ, име на функция, оператор.

Нека буквите от азбуката са разделени на следните класове (под #Число се разбира буква с код Число):

- Край на текста = {#0}
- Литерал = {'}
- Точка = {.}
- ФункСимвол = {(}
- Интервал = {#9,#10,#13,#32}
- Цифра = {0,1,2,3,4,5,6,7,8,9}
- Символ = {!,",#,$,%,&,(,),*,+,#44,-,.,/,:,;,<,=,>,?,@,[,\,],^, `,{,|,},~}
- Буква = Σ\(Интервал∪Цифра∪Символ∪Край на текста)
- Всеки символ = Интервал∪Цифра∪Буква∪Символ

Тогава следния автомат определя поредната лексема:

[†] **A**merican **S**tandart for **C**haracter **I**nformation **I**nterchange



фиг. 6

Разпознаваните типове лексеми са: край на текста(E), идентификатор(I), низ(L), число(N) и символ(S).

Ако типа на разпознатата лексема е идентификатор или символ и стойността и е в списъка на дефинираните оператори, то типа се преобразува в тип оператор.

Ако типа на разпознатата лексема е идентификатор и следващия символ е от класа ФункСимвол, то типа се преобразува в тип име на функция.

2.2.4.3. Синтаксис

Нека е дадена една последователност LS от лексеми, получена от лексикалния анализатор. Задача на синтактичния анализ е да определи дали LS е правилно изречение от езика и да му построи синтактично дърво.

Синтаксисът на езика е определен от граматиката Γ_m, където **m** е максималния приоритет на операторите, които могат да се използват в изразите.

Терминални символи са:

- **; , ( ) [ ]** съответстващи на лексемите от тип символ;
- **ФункцияИме** за лексемите от LS с тип име на функция;
- **Идентификатор** за лексемите с тип идентификатор;
- **Число** за лексемите от LS с тип число;

- **Оп$_{i,xx}$ Оп$_{i,xy}$ Оп$_{i,yx}$ ЛОп$_{i,x}$ ЛОп$_{i,y}$ ДОп$_{i,x}$ ДОп$_{i,y}$** за лексемите от тип оператор, като **i** е приоритета на конкретния оператор; **Оп**, **ЛОп**, **ДОп** са съответно означения за инфиксните, префиксните и постфиксните оператори; **xx**, **xy**, **yx**, **x** и **y** са означения за различните видове асоциативност (например **yx** е за оператори изчислявани в израз без скоби отляво надясно).

Стартов символ на граматиката е Програма.

Правилата са:

Програма → ε | Команда | Команда **;** Програма

Команда → Израз$_0$

СписъкИзрази → Израз$_0$ | Израз$_0$ **,** СписъкИзрази

Списък → **[ ]** | **[** СписъкИзрази **]**

Израз$_m$ → **ФункцияИме (** СписъкИзрази **)**

Израз$_m$ → **ФункцияИме ( )**

Израз$_m$ → **(** Израз$_0$ **)**

Израз$_m$ → **Идентификатор**

Израз$_m$ → **Число**

Израз$_m$ → Списък

Израз$_{m-1}$ → Израз$_m$ **Оп$_{m-1,xx}$** Израз$_m$

Израз$_{m-1}$ → Израз$_m$ **Оп$_{m-1,xy}$** Израз$_{m-1}$

Израз$_{m-1}$ → Израз$_{m-1}$ **Оп$_{m-1,yx}$** Израз$_{m-1}$

Израз$_{m-1}$ → **ЛОп$_{m-1,x}$** Израз$_m$ | **ЛОп$_{m-1,y}$** Израз$_{m-1}$

Израз$_{m-1}$ → Израз$_m$ **ДОп$_{m-1,x}$** | Израз$_{m-1}$ **ДОп$_{m-1,y}$**

Израз$_{m-1}$ → Израз$_m$

...

Израз$_1$ → Израз$_2$ **Оп$_{1,xx}$** Израз$_2$

Израз$_1$ → Израз$_2$ **Оп$_{1,xy}$** Израз$_1$

$Израз_1 \rightarrow Израз_1$ **Оп$_{1,yx}$** $Израз_1$

$Израз_1 \rightarrow$ **ЛОп$_{1,x}$** $Израз_2$ | **ЛОп$_{1,y}$** $Израз_1$

$Израз_1 \rightarrow Израз_2$ **ДОп$_{1,x}$** | $Израз_1$ **ДОп$_{1,y}$**

$Израз_1 \rightarrow Израз_2$

$Израз_0 \rightarrow Израз_1$ **Оп$_{0,xx}$** $Израз_1$

$Израз_0 \rightarrow Израз_1$ **Оп$_{0,xy}$** $Израз_0$

$Израз_0 \rightarrow Израз_0$ **Оп$_{0,yx}$** $Израз_1$

$Израз_0 \rightarrow$ **ЛОп$_{0,x}$** $Израз_1$ | **ЛОп$_{0,y}$** $Израз_0$

$Израз_0 \rightarrow Израз_1$ **ДОп$_{0,x}$** | $Израз_0$ **ДОп$_{0,y}$**

$Израз_0 \rightarrow Израз_1$

Граматика Γ_m е контекстно свободна, следователно изречения от езика породен от нея могат да бъдат разпознавани от магазинен автомат.

Директния подход за възходящ синтактичен анализ, чрез непосредствено прилагане на правилата на граматиката върху входната последователност, не дава задоволителни резултати. Това налага някои промени над граматиката и организацията на алгоритъма за анализ:

Всички терминални символи предварително да бъдат заменени с нетерминални;

Нетерминалните символи да бъдат номерирани;

Правилата да бъдат преобразувани в такива водещи до по детерминиран алгоритъм за анализ;

Правилата да бъдат номерирани и хеширани;

Използване на евристики за отхвърляне на прилагането на правила, които е лесно да се останови, че няма да доведат до резултат и ще пропаднат;

Да се вземе под внимание, че $Израз_{i+1}$ е и $Израз_i$ (поради правилата $Израз_i \rightarrow Израз_{i+1}$)

Предварително определяне на командите в програмата и прилагане на анализа само върху тях.

След прилагането на тези изисквания се получава следната оптимизирана граматика Γ'_m с правила:

1 $Израз_m(2000+m) \rightarrow$ Идентификатор(1)

2 $Израз_m(2000+m) \rightarrow$ Число(3)

3 $Израз_m(2000+m) \rightarrow$ Списък(11)

4 $Израз_m(2000+m) \rightarrow$ ФункцияИме(5) '('(252) СписъкИзраз(10) ')'(254)

5 $Израз_m(2000+m) \rightarrow$ ФункцияИме(5) '('(252) $Израз_0(2000)$ ')'(254)

6 $Израз_m(2000+m) \rightarrow$ ФункцияИме(5) '('(252) ')'(254)

7 $Израз_m(2000+m) \rightarrow$ '('(252) $Израз_0(2000)$ ')'(254)

...

8 $Израз_i(2000+i) \rightarrow Израз_{i+1}(2001+i)$ $Оп_{i,xx}(10i5)$ $Израз_{i+1}(2001+i)$

9 $Израз_i(2000+i) \rightarrow Израз_{i+1}(2001+i)$ $Оп_{i,xy}(10i6)$ $Израз_i(2000+i)$

10 $Израз_i(2000+i) \rightarrow Израз_i(2000+i)$ $Оп_{i,yx}(10i7)$ $Израз_i(2000+i)$

11 $Израз_i(2000+i) \rightarrow ЛОп_{i,x}(10i1)$ $Израз_{i+1}(2001+i)$

12 $Израз_i(2000+i) \rightarrow ЛОп_{i,y}(10i2)$ $Израз_i(2000+i)$

13 $Израз_i(2000+i) \rightarrow Израз_{i+1}(2001+i)$ $ДОп_{i,x}(10i3)$

14 $Израз_i(2000+i) \rightarrow Израз_i(2000+i)$ $ДОп_{i,y}(10i4)$

...

15 СписъкИзраз(10) $\rightarrow$ СписъкИзраз(10) ','(251) $Израз_0(2000)$

16 СписъкИзраз(10) $\rightarrow$ $Израз_0(2000)$ ','(251) $Израз_0(2000)$

17 Списък(11) $\rightarrow$ '['(253) СписъкИзраз(10) ']'(255)

18 Списък(11) $\rightarrow$ '['(253) $Израз_0(2000)$ ']'(255)

19 Списък(11) $\rightarrow$ '['(253) ']'(255)

Резултат от синтактичният анализ е синтактичното дърво (на входната редица LS) в неявна форма. То представлява списък LP от правила и позиции, които са били приложени върху LS.

Програмната реализация на дипломната работа реализира Γ'_9.

След направените измервания над тестови примери необходимото време за анализ е намаляло средно над 250 пъти, и което е по-важно - експоненциалната зависимост от дължината на входната редица се е превърнала в линейна.

2.2.4.4. Семантика

Семантичният анализ трябва да придаде някакво значение на синтактичното дърво. При този етап от транслацията изречението на езика окончателно се превежда в изпълним вид. Този вид е вътрешна структура разбираема за изпълняващата компонента на виртуалната машина (това може да е и машинна програма за физическата машина).

За извършване на този превод, към всяко правило на Γ'_m трябва да бъде прикрепена по една генерираща процедура. Тези процедури обработват редицата от лексеми LS на определено място и заменят последователности от лексемите с частични изходни изпълними резултати. Заменяната поредица е с дължина равна на дължината на тялото на правилото.

За граматиката Γ'_m процедурите генерират следното:

- за правило 1: променлива с име равно на стойността на лексемата - първи елемент в поредицата;
- за правило 2: константа-реално число със стойност определена от лексемата - първи елемент в поредицата;
- за правило 3: константа-списък със стойност равна на образувания вече списък от другите генериращи процедури;

- за правило 4: израз-функция с име определено от първата лексема в поредицата и аргументи - списъкът, представляващ третият елемент в поредицата;
- за правило 5: израз-функция с име определено от първата лексема в поредицата и един аргумент - третият елемент в поредицата;
- за правило 6: израз-функция без аргументи и с име определено от първата лексема в поредицата;
- за правило 7: вторият елемент от поредицата;
- за правило 8, 9 и 10: израз-функция с име бинарния оператор и два аргумента - първият и третият елемент в поредицата;
- за правило 11 и 12: израз-функция с име унарния оператор и аргумент - вторият елемент в поредицата;
- за правило 13 и 14: израз-функция с име унарния оператор и аргумент - първият елемент в поредицата;
- за правило 15: списък равен на списъка първи елемент от поредицата с долепен в края третият елемент от поредицата;
- за правило 16: двуелементен списък съдържащ първия и третия елемент от поредицата;
- за правило 17: списъкът - втори елемент в поредицата;
- за правило 18: едноелементен списък, съдържащ втория елемент от поредицата;
- за правило 19: празен списък.

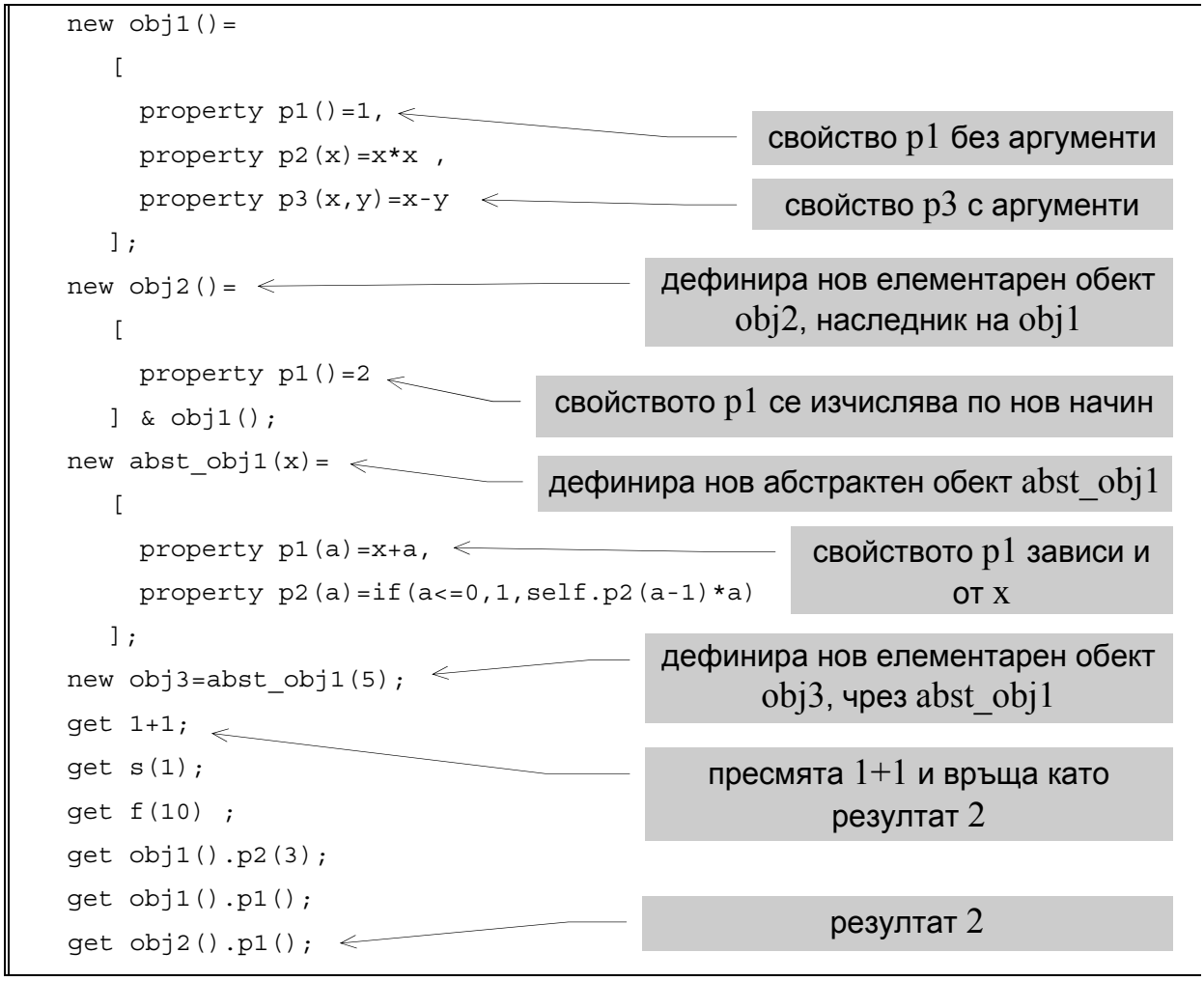
След прилагането на процедурите на всички правила от списъка LP, получаваме крайният резултат - изпълнима вътрешна структура.

2.2.4.5. Примерна програма на езика

```
new s(x)=x+1;
new f(x)=if(x<=0,1,f(x-1)*x) ;
```

дефинира нова функция s

дефинира елементарен обект obj1



2.2.5. Графичен сървър

Както вече е споменато (в точка 2.1.12. Клиент-Сървър организация), графичният сървър инициализира по една графична виртуална машина за всеки клиент. Той също изпълнява и специална начална програма AUTOEXEC.GVM, имаща за цел да дефинира предварително функции и обекти необходими за по-лесно дефиниране на геометрична информация, чрез наследяване на свойствата.

Във връзка с визуализацията са дефинирани и няколко абстрактни обекта влизащи в описанието на сцената заедно с телата.

2.2.6. Описание на геометрична информация

За целите на дипломната работа ще бъде разглеждана само каноничната геометрична информация.

Подцел 8: Да се определи начина за дефиниране на канонична геометрична информация със средствата на езика.

Необходимите свойства, които трябва да бъдат дефинирани за да описва обекта геометрична информация са:

- координатна система - свойства LocalSystem(point) и LocalSystemBack(point);
- форма - свойство Form(point), описващо разстоянията от точката point до тялото спрямо локалната координатна система.

2.2.7. Описание на тела и абстрактни тела

Подцел 9: Да се определят допълнителните (метрични) свойства необходими за визуализация.

Допълнителната информация за телата необходима на алгоритъма за визуализация е:

- коефициент на огледално отражение в дадена точка - свойство RR(point) с резултат списък от отделни коефициенти за всяка компонента на светлината - червена, зелена и синя;
- коефициент на пропускане на светлината в дадена точка - свойство BR(point);
- съпротивление на средата към светлината - свойство Res();
- коефициент на дифузно отражение в дадена точка - свойство KD(point);
- коефициент на отражение на дифузната светлина в дадена точка - свойство DiffuseR(point);
- степента на Тонг за пространствено разпределение на огледално отразената светлина - свойство Tn(point);

- коефициент на пречупване на светлината за тялото - свойство LtR();

Повече информация за коефициентите може да се намери в [1].

2.2.8. Описание на светлинните източници

Подцел 10: Да се определи как да бъдат описвани светлинните източници със средствата на езика.

2.2.8.1. Точкови

Всеки точков източник на светлина има определено местоположение в пространството и излъчва светлина във всички посоки с определена интензивност. Това показва, че тези източници на светлина могат да бъдат представяни като наследници на абстрактния обект:

```
PointLight(center,eng)=
   [
     property Position()=center,
     property Energyes()=eng
   ]
```

Например: `PointLight([0,0,0],[0,80,0])` е източник разположен в центъра на координатната система и излъчва зелена светлина.

2.2.8.2. Разсеяна светлина

По подобен начин се определя и разсеяната светлина за сцената:

```
DiffuseLight(eng,dist)=
   [
     property Energyes()=eng,
     property Distance()=dist
   ]
```

Повече информация за светлинните източници може да се намери в [1].

2.2.9. Визуализация‡

Алгоритъма за визуализация описан и реализиран в [1] може да бъде приложен за описаните чрез езика тела.

За целта трябва да се добави нова вградена функция Show(...) имаща за аргументи произволен брой тела и светлинни източници. Резултат от функцията е изображението получено след прилагането на алгоритъма, известен като Трасиране на лъчи (Ray Tracing).

Връзката с растеризиращите процедури изисква телата да дават определена информация, което води до необходимостта за решаване на следните подзадачи:

2.2.9.1. Положение на лъч спрямо тяло

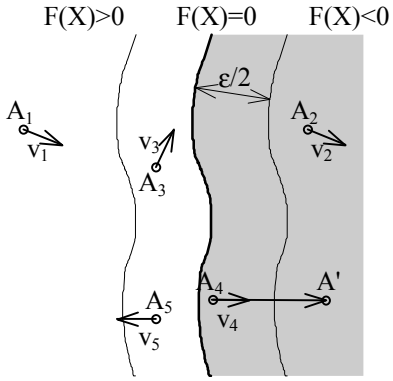
Нека е даден лъч L с начална точка A и посока v ($||v||=1$). Нека също е дадено тялото T, зададено чрез каноничната си геометрична информация $g_k=\{F(X),\{m_1,m_2,...,m_q\},sys(Z)\}$.

Подцел 11: Да се определи дали лъча L е външен, вътрешен, влизащ, излизащ или граничен за тялото T.

Поради неточното (непълното) представяне на реалните числа в компютрите трябва под граница на тялото да се разбира не само точките за които $F(X)=0$, а и някаква тяхна малка околност. Тогава под граница на тялото T ще се разбира $TL=\{X\in E^n \mid |F(X)|\leq\varepsilon/2\}$, за някое малко число ε.

---

‡ Голяма част от понятията в тази подточка имат отношение към алгоритъма за визуализация (виж [1]) и няма да бъдат описвани в настоящата дипломна работа.



фиг. 6

На фиг 6 са показани различните възможни положения на лъчите, като съответно (A_1,v_1) е външен, (A_2,v_2) е вътрешен, (A_3,v_3) е граничен, (A_4,v_4) е влизащ и (A_5,v_5) е излизащ.

Произволният лъч L може да бъде класифициран по следният алгоритъм:

Ако $F(A)<-\varepsilon/2$ то L е вътрешен

$F(A)>\varepsilon/2$ то L е външен

Иначе

Ако $F(A+\varepsilon.v)<-\varepsilon/2$ то L е влизащ

$F(A+\varepsilon.v)>\varepsilon/2$ то L е излизащ

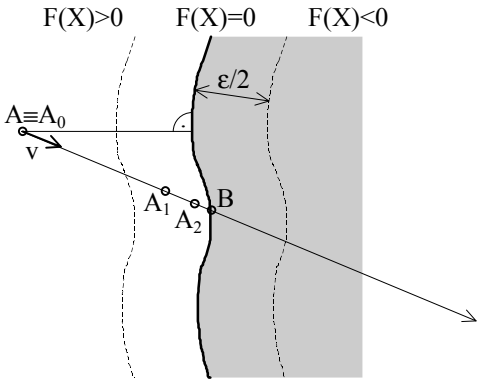
Иначе L е граничен

Например (A_4,v_4) е влизащ, защото началната му точка е от границата на тялото, а точката A'_4, представляваща началната точка отместена по посоката на лъча със стъпка ε, е вътрешна за тялото.

2.2.9.2. Пресечена точка на лъч с тяло

Нека е даден лъч L с начална точка A и посока v ($||v||=1$). Нека също е дадено тялото T, зададено чрез каноничната си геометрична информация $g_k=\{F(X),\{m_1,m_2,...,m_q\},sys(Z)\}$.

Подцел 12: Да се намери пресечената точка B между тялото T и лъчът L.



фиг. 7

Пресечената точка B се намира чрез итерацията:

$A_0=A$

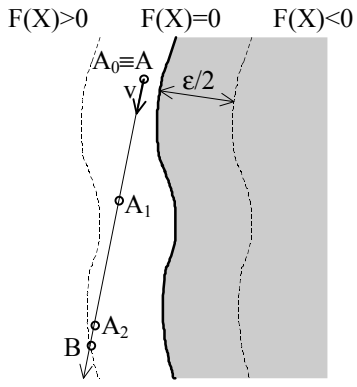
$A_{i+1}=A_i + v.|F(A_i)|$, докато $|F(A_i)|>\varepsilon/2$.

Ако пресечена точка B съществува, то редицата $\{A_i\}$ е сходяща към нея. В противен случай редицата не е сходяща и след изминаване на определено достатъчно голямо разстояние, итерацията може да бъде прекратена и даден отговор, че лъчът не пресича тялото.

2.2.9.3. Точка на отделяне от повърхността

Нека е даден гранитен лъч L с начална точка A и посока v ($||v||=1$). Нека също е дадено тялото T, зададено чрез каноничната си геометрична информация $g_k=\{F(X), \{m_1,m_2,...,m_q\},sys(Z)\}$.

Подцел 13: Да се намери точката на отделяне от повърхността на лъчът L.



фиг. 8

Точката на отделяне B се намира чрез итерацията:

$A_0=A$

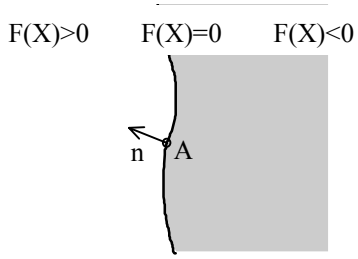
$A_{i+1}=A_i + \varepsilon.v$, докато $|F(A_i)|<\varepsilon/2$.

Когато точката A_{i+1} напусне границата на тялото, това е точка B. В противен случай след изминаване на определено достатъчно голямо разстояние, итерацията може да бъде прекратена и даден отговор, че няма точка на отделяне.

2.2.9.4. Нормален вектор в точка от повърхността

Нека е дадено тяло T, зададено чрез каноничната си геометрична информация $g_k=\{F(X),\{m_1,m_2,...,m_q\},sys(Z)\}$. Нека също е дадена точка A от границата на T.

Подцел 14: Да се намери нормалният вектор n в точката A.



фиг. 9

Нормалният вектор се намира по формулата: $n=\nabla F(A)$.

Приближено пресмятане за тримерния случай става по формулата:

$$n=\nabla F(A)=\nabla F((A_x,A_y,A_z))\approx([F((A_x+\varepsilon,A_y,A_z))-F(A)]/\varepsilon,$$
$$[F((A_x,A_y+\varepsilon,A_z))-F(A)]/\varepsilon,$$
$$[F((A_x,A_y,A_z+\varepsilon))-F(A)]/\varepsilon\ )$$

Част III

Реализация

Програмната реализация е под формата на 7 модула (Unit) и една програма на езика Borland Pascal 7.0. Общият брой програмни линии е ≈ 4600 при обем на файловете ≈145K.

Програмата G-Client.PAS

Демонстрира възможността за работа на един клиент с графичната виртуална машина.

При стартиране на програмата на екрана се извежда подканящо съобщение `:>` и се изчаква въвеждане на команда. За изход от програмата се използва командата **.** (точка). За стартиране на програма написана на езика на виртуалната машина се използва **!<име на файл>**, което предизвиква изпълнение на зададения файл, като последователност от команди. Подразбиращо се разширение на името на файла е **.GVM**. Всички останали входни данни се изпълняват като команда на графичната виртуална машина. След изпълнение на дадена команда резултатът от нея се визуализира.

Модул uL1.PAS

Реализира някои основни обекти (в описанието на реализацията обект ще се използва в смисълът на езика Borland Pascal) на програмната реализация.

Обектът RootObj е главен в йерархията и дефинира някои основни методи, общи за всичките му наследници, като: създаване на копие

на екземпляр на обект, освобождаване на динамичен екземпляр на обект и др.

Обектът ListObj реализира структурата списък с основните и някои улесняващи работата методи. Списъкът е полиморфен и може да се попълва с произволни обекти наследници на RootObj.

Обектът ParamObj се използва като елемент в списъците с параметри.

Обектът ParamListObj реализира списък от параметри. Той предвижда освен всички методи на ListObj и метод за търсене на стойност на параметър по неговото име.

Обектът ValueObj е родител на всички обекти, които ще се изчисляват. Методът Calc пресмята стойността на обекта в зависимост от актуалните параметри в ActualParamList.

Процедурата InitUnit се използва за първоначално инициализиране на модула.

Процедурата DoneUnit се използва за деинициализиране на модула.

Модул uL2.PAS

Реализира обектите за работа с изрази и функции. Поддържа списък с текущо дефинираните функции.

Обектът ExpressionObj е родител на всички видове изрази.

Обектът ExprVarObj реализира израз-променлива.

Обектът ExprRealObj реализира израз-константа със стойност реално число.

Обектът ExprListObj реализира израз-константа със стойност списък.

Обектът ExprFuncObj реализира израз-суперпозиция.

Обектът FunctionObj е родител на двата вида функции - вградени и потребителски.

Обектът InternalFunctionObj реализира вградени функции.

Обектът UserFunctionObj реализира потребителски функции зададени чрез израз.

Процедурата DefineFunction добавя нова или променя вече дефинирана функция.

Функцията DeleteFunction премахва функция с определено име от списъка с текущо дефинираните функции. Резултат: True - успешно премахване.

Функцията GetFunctionsList позволява достъп до списъкът с текущо дефинираните функции.

Процедурата SetFunctionsList установява нов списъкът с текущо дефинирани функции.

Функцията FindFunction търси функция с дадено име.

Процедурата InitUnit се използва за първоначално инициализиране на модула.

Процедурата DoneUnit се използва за деинициализиране на модула.

Модул uL2_1.PAS

Реализира обектите представляващи вградени функции.

Това са: ISelectObj, IPlusObj, IConcatObj, IMinusObj, IMulObj, IDivideObj, IUMinusObj, IPowerObj, IFactorialObj, IModObj, IDivObj, INotObj, IAndObj, IOrObj, IXorObj, IEqualObj, ILetObj, INoEqualObj, ILessOrEqualObj, IGreatOrEqualObj, ILessObj, IGreatObj, INewObj, IGetObj, IDelObj, IPropertyObj, IOddObj, ISqrObj, ISqrtObj, ISinObj, ICosObj,

IArcTanObj, IPiObj, IExpObj, ILnObj, IAbsObj, ISgnObj, IIntObj, IRoundObj, IFracObj, IRandomObj, IIfObj, ICaseObj, IMinObj и IMaxObj.

Процедурата InitUnit се използва за първоначално инициализиране на модула. Тя дефинира всички вградени функции.

Процедурата DoneUnit се използва за деинициализиране на модула.

Модул Lexical.PAS

Реализира лексикалният анализатор.

Обектът LexemaObj представлява лексема с определен тип и стойност.

Обектът LexBreakerObj е лексикален анализатор. Методът GetLex връща поредната лексема.

Процедурата InitUnit се използва за първоначално инициализиране на модула.

Процедурата DoneUnit се използва за деинициализиране на модула.

Модул Syntax.PAS

Реализира синтактичният анализатор и транслацията.

Обектът RuleTermSymbolObj се използва за създаване на терминални символи в описанието на правилата.

Обектът RuleNoTermSymbolObj се използва за създаване на нетерминални символи в описанието на правилата.

Типът CompileProcType определя интерфейса на генериращите процедури.

Обектът RuleObj се използва за създаване на правилата на граматиката. Те се състоят от глава, тяло и генерираща процедура.

Обектът SyntaxAnaliserObj е синтактичният анализатор и транслатор. Методът Compile преобразува текст на програма в списък от изразите в нея. Методът Test извършва синтактичният анализ. Методът CompileOne транслира една команда в израз.

Процедурата InitUnit се използва за първоначално инициализиране на модула.

Процедурата DoneUnit се използва за деинициализиране на модула.

Модул uL3.PAS

Реализира връзката с визуализиращите модули и дефинира вградената функция Show.

Процедурата InitUnit се използва за първоначално инициализиране на модула.

Процедурата DoneUnit се използва за деинициализиране на модула.

Модул uL4.PAS

Реализира графична виртуална машина.

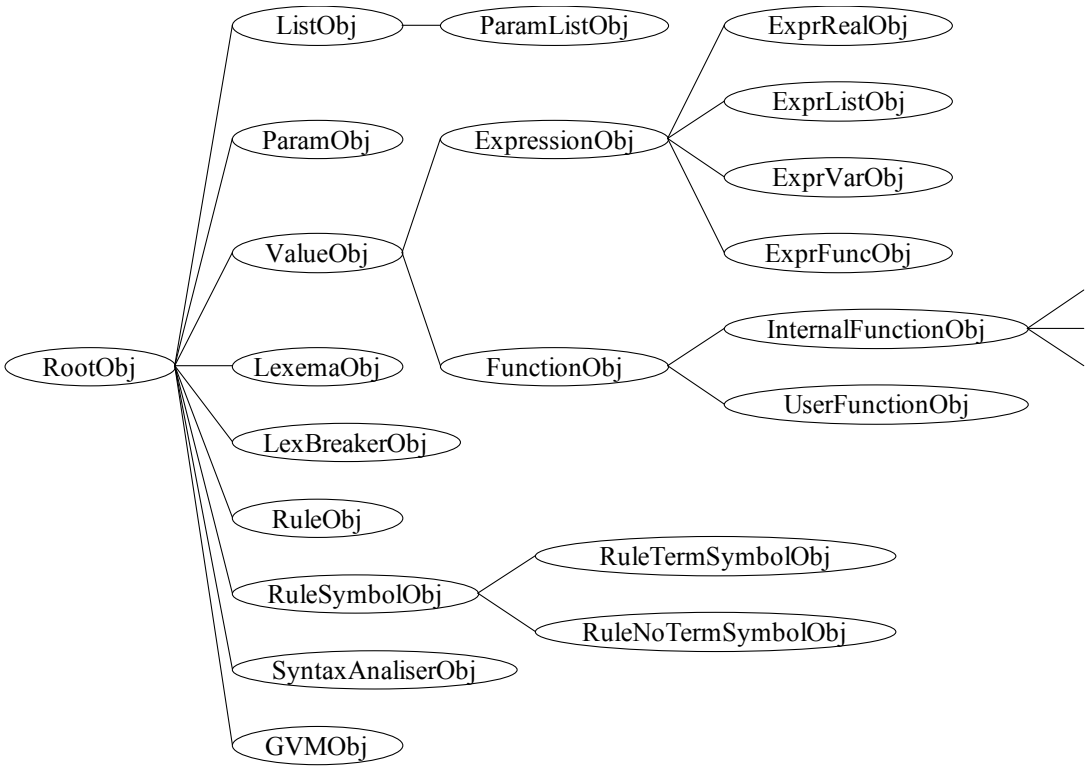
Обектът GVMObj представлява графична виртуална машина.

Методът InitHardware предвиден за настройване на хардуера за работа с определена виртуална машина. Методът ReturnResult извежда резултатите от изпълнението на командите и грешките.

Методът Execute изпълнява команда на езика на машината.

Методът Run изпълнява командите записани в даден файл.

Йерархия на обектните типове



Част IV

Заключение

Целите на дипломната работа са постигнати.

Дефинираният език може да бъде използван по няколко начина:

- Като език за описание и съхранение на сцени за визуализиране;
- Като машинен език на графична виртуална машина;
- Като средство за заявки към графичен сървър.

Възможно е лесно да бъде разширен езика с нови функции и оператори. Добре е също да се реализира компилатор с цел по-бързо изчисляване на изразите.

Връзката с визуализиращата компонента е възможна и е осъществена. Скоростта на растеризиране на сцените е ниска и затова бъдещите разширения могат да включват и други по-бързи (и даващи по-некачествено изображение) алгоритми, които да се използват за предварително преглеждане или при редактиране на сцената.

Теоретичната част и програмната реализация на дипломната работа дават добра основа за по нататъшно изследване на подхода за описание на геометрична информация.

Литература

1. Вълко Вълков - Дипломна работа “Построяване на реалистични изображения на тримерни тела”, ПУ “Паисий Хилендарски”, Пловдив, 1995 г.

2. Б. Боровски, Б. Янков, Г. Гочев, Д. П. Шишков, ... “Справочник по изчислителна техника”, “Техника”, София, 1990 г.

3. Б. Янков “Транслатори и операционни системи”, “Техника”, София, 1977 г.

4. П. Бърнев, С. Керпетжиев “Основни понятия в информатиката”, “Д-р Петър Берон”, София, 1988 г.

5. Стоян Ю. Г., Яновчев С. В. “Математические модели и оптимизационные методы геометрическое проектирова-ния”, “Наукова думка”, Киев, 1986 г.

6. В. Димова-Нанчева, Н. Стоянов “Висша математика” - част 1, “Техника”, София, 1973 г.

7. С. Манолов, А. Петрова-Денева, А. Генов, Н. Шополов “Висша математика” - част 2, “Техника”, София, 1977 г.

Приложения

Приложение 1 - изходни текстове на програмите

Изходен текст на модула uL1

```
Unit uL1; { Модул ниво първо }

Interface

Uses Strings; { Използва модул за работа с Null-низове }

Type { Базов обект в йерархията }
     RootObjPtr = ^RootObj;
     RootObj = Object
        Constructor Init;                  { Конструктор }
        Destructor  Done; virtual;         { Деструктор }
        Procedure Free;                    { Освобождава динамичен обект от типа }

        Function  Copy : Pointer; virtual;  { Създава копие на обекта }
        Function  _write : String; virtual; { Текстово изображение на обекта }
     End;

Type { Запис за реализиране на списъци от обекти наследници на RootObj }
     ListNodePtr = ^ListNodeRec;
     ListNodeRec = Record
        Next : ListNodePtr;            { Следващ възел }
        Obj : RootObjPtr;              { Данни - указател към обект }
     End;
     { Обект Списък от обекти произволни обекти }
     ListObjPtr = ^ListObj;
     ListObj = Object(RootObj)
        Last : ListNodePtr;            { Последен елемент от списъка }
        Count : LongInt;               { Брой на елементите в списъка }
        LastAccessed : LongInt;        { Номер на последния използван елем. }
        LastAccessedPtr : ListNodePtr; { Адрес на последния използван елем. }
        Constructor Init;                          { Конструира празен списък }
        Destructor  Done; virtual;                 { Деструктира списъка и елементите му }
        Function  Copy : Pointer; virtual;         { Копира списъка }
        Function  Get (Num : LongInt) : Pointer;   { Елемент по номер от 1,... }
        Function  Add (O : RootObjPtr) : Pointer;  { Добавя нов елемент-обект }
        { Вмъква нов елемент на Num-то място. Всички след него се изместват }
        Function  Insert (Num : LongInt; O : RootObjPtr) : Pointer;
        { Заменя елемента на Num-то място с дадения. Стария се деструктира }
        Function  Replace (Num : LongInt; O : RootObjPtr) : Pointer;
        Function  Remove (Num : LongInt) : Pointer; { Премахва елемент Num }
        Procedure Del (Num : LongInt);             { Премахва и деструктира елемент }
        Function  GetCount : LongInt;              { Връща броя на елементите }

        Function  _write : String; virtual;        { Текстово изображение на списъка }
     End;

Type { Обект Параметър в параметричен списък }
     ParamObjPtr = ^ParamObj;
     ParamObj = Object(RootObj)
        Name : PChar;       { име на параметъра }
        Value : RootObjPtr; { стойност на параметъра - nil за формален парам. }
        { Конструира параметър по име aName и стойност aValue }
        Constructor Init (aName : PChar; aValue : RootObjPtr);
        Destructor  Done; virtual;
```

```
        Function  Copy : Pointer; virtual;
        Function  _write : String; virtual; { Текстово изображение параметъра }
     End;

     { Обект Списък от параметри }
     ParamListObjPtr = ^ParamListObj;
     ParamListObj = Object(ListObj)
        { Стойност на параметър по зададено име }
        Function  GetParam (ParamName : PChar) : RootObjPtr;
     End;

Type { Обект Стойност }
     ValueObjPtr = ^ValueObj;
     ValueObj = Object(RootObj)
        { Пресмята на стойността евентуално зависеща от параметрите от списъка }
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
     End;

Procedure InitUnit; { Първоначална инициализация на модула }
Procedure DoneUnit; { Край на работата с модула }

Implementation

{ RootObj }

Constructor RootObj.Init;

Begin
End;

Destructor RootObj.Done;

Begin
End;

Procedure RootObj.Free;

Begin
     Dispose (RootObjPtr(@Self),Done);
End;

Function RootObj.Copy : Pointer;

Begin
     Copy := New(RootObjPtr,Init);
End;

Function RootObj._write : String;

Begin
     _write := '.';
End;

{ ListObj }

Constructor ListObj.Init;

Begin
     Inherited Init;
     Last := nil;
     Count := 0;
     LastAccessed := $7FFFFFFF;
     LastAccessedPtr := nil;
End;

Destructor ListObj.Done;

Var L : ListNodePtr;

Begin
     While Last<>nil do
           Begin
                L := Last^.Next;
                If L=Last Then Last := nil
                          Else Last^.Next := L^.Next;
                If L^.Obj<>nil Then L^.Obj^.Free;
                Dispose (L);
```

```
          End;

     Inherited Done;
End;

Function ListObj.Copy : Pointer;

Var L : ListObjPtr;
    N : LongInt;

Begin
     New (L,Init);
     For N := 1 to Count do L^.Add (RootObjPtr(Get(N))^.Copy);
     Copy := L;
End;

Function ListObj.Get (Num : LongInt) : Pointer;

Var L : ListNodePtr;
    Num1 : LongInt;

Begin
     If (Last<>nil) and (Num>0) and (Num<=Count) Then
        Begin
             If Num>LastAccessed Then
                Begin
                     L := LastAccessedPtr;
                     Num1 := Num;
                     Dec (Num,LastAccessed);
                     LastAccessed := Num1;
                End
             Else
                Begin
                     If Num=LastAccessed Then
                        Begin
                             Get := LastAccessedPtr^.Obj;
                             Exit;
                        End
                     Else
                        Begin
                             L := Last;
                             LastAccessed := Num;
                        End;
                End;

{
             Repeat
                   Dec (Num);
                   L := L^.Next;
             Until Num=0;
}
             Asm
                        Mov     Ax,Word Ptr [Num]
                        Mov     Bx,Word Ptr [Num+2]
                        LEs     Di,L
                   @1:
                        LEs     Di,Es:[Di]
                        Sub     Ax,1
                        Sbb     Bx,0
                        Jnz     @1
                        Or      Ax,Ax
                        Jnz     @1

                        Mov     Word Ptr [L],Di
                        Mov     Word Ptr [L+2],Es
             End;
             Get := L^.Obj;
             LastAccessedPtr := L;
        End
     Else
        Begin
             LastAccessed := $7FFFFFFF;
             LastAccessedPtr := nil;
             Get := nil;
        End;
End;
```

```
Function ListObj.Add (O : RootObjPtr) : Pointer;

Var L : ListNodePtr;

Begin
     New (L);
     L^.Obj := O;
     If Last<>nil Then
        Begin
             L^.Next := Last^.Next;
             Last^.Next := L;
        End
     Else L^.Next := L;
     Last := L;
     Inc (Count);
     Add := O;
End;

Function ListObj.Insert (Num : LongInt; O : RootObjPtr) : Pointer;

Var L : ListNodePtr;

Begin
     If Num>Count Then Add (O)
     Else
        Begin
             If Num<=0 Then Num := 1;
             Get (Num);
             New (L);
             L^ := LastAccessedPtr^;
             LastAccessedPtr^.Obj := O;
             LastAccessedPtr^.Next := L;
             If Last=LastAccessedPtr Then Last := L;
             Inc (Count);
        End;
     Insert := O;
End;

Function ListObj.Replace (Num : LongInt; O : RootObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     O1 := Get (Num);
     If O1<>nil Then O1^.Free;
     If LastAccessedPtr<>nil Then LastAccessedPtr^.Obj := O
                             Else Insert (Num,O);
     Replace := O;
End;

Function ListObj.Remove (Num : LongInt) : Pointer;

Var L : ListNodePtr;

Begin
     Remove := Get (Num);
     If LastAccessedPtr<>nil Then
        Begin
             L := LastAccessedPtr^.Next;
             If LastAccessedPtr=Last Then
                Begin
                     Get (Num-1);
                     Last := LastAccessedPtr;
                     If Last<>nil Then
                        Begin
                             Dispose (Last^.Next);
                             Last^.Next := L;
                        End
                     Else Dispose (L);
                End
             Else
                Begin
                     LastAccessedPtr^ := L^;
                     If L=Last Then Last := LastAccessedPtr;
                     Dispose (L);
                End;
             Dec (Count);
```

```
             LastAccessed := $7FFFFFFF;
             LastAccessedPtr := nil;
        End;
End;

Procedure ListObj.Del (Num : LongInt);

Var O : RootObjPtr;

Begin
     O := Remove (Num);
     If O<>nil Then O^.Free;
End;

{
Function ListObj.GetCount : LongInt;

Begin
     GetCount := Count;
End;
}

Function ListObj.GetCount : LongInt; Assembler;

Asm
        Mov     Ax,Word Ptr Es:[Di+Count]
        Mov     Dx,Word Ptr Es:[Di+Count+2]
End;

Function ListObj._write : String;

Var I : LongInt;
    S : String;
    O : RootObjPtr;

Begin
     S := '[';
     If GetCount>0 Then
        Begin
             For I := 1 to GetCount-1 do
                 Begin
                      O := Get(I);
                      If O<>nil Then S := S + O^._write + ','
                                Else S := S + 'nil,';
                 End;
             O := Get(GetCount);
             If O<>nil Then S := S + O^._write
                       Else S := S + 'nil';
        End;
     _write := S + ']';
End;

{ ParamObj, ParamListObj }

Constructor ParamObj.Init (aName : PChar; aValue : RootObjPtr);

Begin
     Inherited Init;
     Name := StrNew (aName);
     Value := aValue;
End;

Destructor ParamObj.Done;

Begin
     StrDispose (Name);
     If Value<>nil Then Value^.Free;
     Inherited Done;
End;

Function ParamObj.Copy : Pointer;

Begin
     If Value<>nil Then Copy := New(ParamObjPtr,Init(Name,Value^.Copy))
                   Else Copy := New(ParamObjPtr,Init(Name,nil));
End;
```

```
Function ParamObj._write : String;

Var S : String;

Begin
     _write := StrPas(Name);
End;

Function ParamListObj.GetParam (ParamName : PChar) : RootObjPtr;

Var N : LongInt;
    P : ParamObjPtr;

Begin
     GetParam := nil;
     For N := 1 to GetCount do
        Begin
             P := Get(N);
             If (P^.Name<>nil) and (StrIComp(ParamName,P^.Name)=0) Then
                Begin
                     GetParam := P^.Value;
                     Break;
                End;
        End;
End;

{ ValueObj }

Function ValueObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Begin
     Calc := nil;
End;


{}

Const UnitActive : Byte = 0; { семафор за инициализация на модула }

Procedure InitUnit;

Begin
     Inc (UnitActive);
     If UnitActive>1 Then Exit;
End;

Procedure DoneUnit;

Begin
     Dec (UnitActive);
     If UnitActive>0 Then Exit;
End;


End.
```

Изходен текст на модула uL2

```
Unit uL2; { Модул ниво второ }

Interface

Uses uL1,Strings; { Използва: модул ниво първо;
                              модул за работа с Null-низове. }

Type AssocType = (Assoc_functional,                    { видове асоциативност на операторите }
                  Assoc_fx,Assoc_fy,Assoc_xf,Assoc_yf,
                  Assoc_xfx,Assoc_xfy,Assoc_yfx);

Type FunctionObjPtr = ^FunctionObj;
{}
```

```
     { Базов обект за работа с израз }
     ExpressionObjPtr = ^ExpressionObj;
     ExpressionObj = Object(ValueObj)
        Function  _write : String; virtual;
     End;

     { Обект израз-променлива }
     ExprVarObjPtr = ^ExprVarObj;
     ExprVarObj = Object(ExpressionObj)
        Name : PChar;                        { име на променливата }
        Constructor Init (aName : PChar);    { Конструира променлива с дадено име. }
        Destructor  Done; virtual;
        Function  Copy : Pointer; virtual;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
        Function  _write : String; virtual;
     End;

     { Обект израз-реална константа }
     ExprRealObjPtr = ^ExprRealObj;
     ExprRealObj = Object(ExpressionObj)
        Value : Real;                        { стойност на константата }
        Constructor Init (aValue : Real);    { Конструира константа с дадена стойност. }
        Function  Copy : Pointer; virtual;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;

        Procedure SetVal (aValue : Real);    { Задава нова стойност /не се използва/. }
        Function  _write : String; virtual;
     End;

     { Обект израз-списък }
     ExprListObjPtr = ^ExprListObj;
     ExprListObj = Object(ExpressionObj)
        Value : ListObjPtr;                       { стойност-списък от обекти }
        Constructor Init (aValue : ListObjPtr);  { Конструира константа с дадена стойност. }
        Destructor  Done; virtual;
        Function  Copy : Pointer; virtual;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;

        Procedure SetVal (aValue : ListObjPtr);  { Задава нова стойност /не се използва/. }
        Function  _write : String; virtual;
     End;

     { Обект израз-суперпозиция }
     ExprFuncObjPtr = ^ExprFuncObj;
     ExprFuncObj = Object(ExpressionObj)
        Func : PChar;                { име на изпълняваната функция }
        LinkFunc : FunctionObjPtr;   { връзка с функцията - за по-бързо пресмятане }
        SubExprList : ListObjPtr;    { списък от подизрази }
        { Конструира израз по име на функция и подизрази. }
        Constructor Init (aName : PChar; aSubExprList : ListObjPtr);
        Destructor  Done; virtual;
        Function  Copy : Pointer; virtual;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
        Function  _write : String; virtual;
     End;
{}
     { Базов обект за работа с функции }
     FunctionObj = Object(ValueObj)
        Name : PChar;                        { име }
        Priority : Byte;                     { приоритет }
        Assoc : AssocType;                   { асоциативност }
        FormalParamList : ParamListObjPtr;  { списък от формални параметри }
        { Конструира функция по име, приоритет, ацоц. и формални параметри. }
        Constructor Init (aName : PChar; aPriority : Byte; aAssoc : AssocType;
                          aFormalParamList : ParamListObjPtr);
        Destructor  Done; virtual;
        Function  Copy : Pointer; virtual;
        { Осъществява предаването на параметрите. }
        { Свързва формалните параметри със стойностите на подизразите. }
        Function  MakeParamList (ActualParamList : ParamListObjPtr;
                                 SubExprList : ListObjPtr) : ParamListObjPtr; virtual;
{        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;}
        Function  _write : String; virtual;
     End;

     { Базов обект за създаване на нови вградени функции }
     InternalFunctionObjPtr = ^InternalFunctionObj;
```

```
     InternalFunctionObj = Object(FunctionObj)
     End;

     { Обект за дефиниране на потребителски функции }
     UserFunctionObjPtr = ^UserFunctionObj;
     UserFunctionObj = Object(FunctionObj)
        Expr : ExpressionObjPtr;  { израз, по който да се пресмята ф-ята }
        { Конструира потребителска функция по име, приоритет, ацоц.,
          формални параметри и израз. }
        Constructor Init (aName : PChar; aPriority : Word; aAssoc : AssocType;
                          aFormalParamList : ParamListObjPtr; aExpr : ExpressionObjPtr);
        Destructor  Done; virtual;
        Function  Copy : Pointer; virtual;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
        Function  _write : String; virtual;
     End;

Procedure InitUnit;  { Първоначална инициализация на модула }
Procedure DoneUnit;  { Край на работата с модула }

Procedure DefineFunction (DF : FunctionObjPtr); { Дефинира нова функция или подменя вече съществуваща. }
Function  DeleteFunction (Name : PChar) : Boolean; { Премахва (деактивира) съществуваща функция. }
Function  GetFunctionsList : ListObjPtr;       { Позволява достъп до списъка на дефинираните функции. }
Procedure SetFunctionsList (FL : ListObjPtr);  { Променя списъка на дефинираните функции. }
Function  FindFunction (Name : PChar) : FunctionObjPtr; { Търси функция със зададеното. }

Implementation

Uses uL2_1;  { използва модул за първоначално дефиниране на вградени функции }

Var FuncList : ListObjPtr;  { списък с текущо дефинираните функции }

Procedure DefineFunction (DF : FunctionObjPtr);

Var I : LongInt;
    F : FunctionObjPtr;

Begin
     For I := 1 to FuncList^.GetCount do
         Begin
              F := FuncList^.Get (I);
              If StrIComp(DF^.Name,F^.Name)=0 Then
                 Begin
                      If (F^.Assoc=Assoc_functional) and
                         (DF^.Assoc=Assoc_functional) Then
                         Begin
                              F^.Done;
                              F^ := DF^;
                              UserFunctionObjPtr(DF)^.Name := nil;
                              UserFunctionObjPtr(DF)^.FormalParamList := nil;
                              UserFunctionObjPtr(DF)^.Expr := nil;
                         End;
                      DF^.Free;
                      Exit;
                 End;
         End;
     FuncList^.Add (DF);
End;

Function DeleteFunction (Name : PChar) : Boolean;

Var I : LongInt;
    F : FunctionObjPtr;

Begin
     DeleteFunction := False;
     For I := 1 to FuncList^.GetCount do
         Begin
              F := FuncList^.Get (I);
              If (StrIComp(Name,F^.Name)=0) and (F^.Assoc=Assoc_functional) Then
                 Begin
                      UserFunctionObjPtr(F)^.Expr^.Free;
                      UserFunctionObjPtr(F)^.Expr := New(ExpressionObjPtr,Init);
                      DeleteFunction := True;
                      Exit;
                 End;
         End;
```

```
End;

Function GetFunctionsList : ListObjPtr;

Begin
     GetFunctionsList := FuncList;
End;

Procedure SetFunctionsList (FL : ListObjPtr);

Begin
     FuncList := FL;
End;

Function FindFunction (Name : PChar) : FunctionObjPtr;

Var I : LongInt;
    F : FunctionObjPtr;

Begin
     For I := 1 to FuncList^.GetCount do
         Begin
              F := FuncList^.Get (I);
              If StrIComp(Name,F^.Name)=0 Then
                 Begin
                      FindFunction := F;
                      Exit;
                 End;
         End;
     FindFunction := nil;
End;

{ ExpressionObj }

Function ExpressionObj._write : String;

Begin
     _write := '???';
End;

{ ExprVarObj }

Constructor ExprVarObj.Init (aName : PChar);

Begin
     Inherited Init;
     Name := StrNew (aName);
End;

Destructor ExprVarObj.Done;

Begin
     StrDispose (Name);
     Inherited Done;
End;

Function ExprVarObj.Copy : Pointer;

Begin
     Copy := New(ExprVarObjPtr,Init(Name));
End;

Function ExprVarObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var P : RootObjPtr;

Begin
     P := ActualParamList^.GetParam (Name);
     If P<>nil Then Calc := P^.Copy
               Else Calc := nil;
End;

Function ExprVarObj._write : String;

Begin
     _write := StrPas (Name);
End;
```

```
{ ExprRealObj }

Constructor ExprRealObj.Init (aValue : Real);

Begin
     Inherited Init;
     Value := aValue;
End;

Function ExprRealObj.Copy : Pointer;

Begin
     Copy := New(ExprRealObjPtr,Init(Value));
End;

Function ExprRealObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Begin
     Calc := Copy;
End;

Procedure ExprRealObj.SetVal (aValue : Real);

Begin
     Value := aValue;
End;

Function ExprRealObj._write : String;

Var S : String;

Begin
     Str (Value:0:10,S);
     While (Byte(S[0])>1) and
           (System.Copy(S,Byte(S[0]),1)='0') and
           (System.Copy(S,Byte(S[0])-1,1)<>'.') do Dec (Byte(S[0]));
     If Pos('.0',S)=Byte(S[0])-1 Then Dec (Byte(S[0]),2);
     _write := S;
End;

{ ExprListObj }

Constructor ExprListObj.Init (aValue : ListObjPtr);

Begin
     Inherited Init;
     Value := aValue;
End;

Destructor ExprListObj.Done;

Begin
     If Value<>nil Then Value^.Free;
     Inherited Done;
End;

Function ExprListObj.Copy : Pointer;

Begin
     If Value<>nil Then Copy := New(ExprListObjPtr,Init(Value^.Copy))
                   Else Copy := New(ExprListObjPtr,Init(nil));
End;

Function ExprListObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var L : ListObjPtr;
    I : LongInt;
    E : ExpressionObjPtr;

Begin
     If Value<>nil Then
        Begin
             New (L,Init);
             For I := 1 to Value^.GetCount do
                 Begin
                      E := Value^.Get (I);
```

```
                    If TypeOf(E^)<>TypeOf(UserFunctionObj)
                       Then L^.Add (E^.Calc(ActualParamList))
                       Else L^.Add (E^.Copy);
               End;
          Calc := L;
       End
    Else Calc := nil;
End;

Procedure ExprListObj.SetVal (aValue : ListObjPtr);

Begin
    If Value<>nil Then Value^.Free;
    Value := aValue;
End;

Function ExprListObj._write : String;

Begin
    If Value<>nil Then _write := Value^._write
                  Else _write := '[]';
End;

{ ExprFuncObj }


Constructor ExprFuncObj.Init (aName : PChar; aSubExprList : ListObjPtr);

Begin
    Inherited Init;
    Func := StrNew (aName);
    LinkFunc := nil;
    SubExprList := aSubExprList;
End;

Destructor ExprFuncObj.Done;

Begin
    StrDispose (Func);
    If SubExprList<>nil Then SubExprList^.Free;
    Inherited Done;
End;

Function ExprFuncObj.Copy : Pointer;

Begin
    Copy := New(ExprFuncObjPtr,Init(Func,SubExprList^.Copy));
End;

Function ExprFuncObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var L : ParamListObjPtr;

Begin
    If LinkFunc=nil Then LinkFunc := FindFunction (Func);
    If LinkFunc<>nil Then
       Begin
            L := LinkFunc^.MakeParamList (ActualParamList,SubExprList);
            Calc := LinkFunc^.Calc (L);
            L^.Free;
       End
    Else Calc := nil;
End;

Function ExprFuncObj._write : String;

Var I : LongInt;
    S : String;

Begin
    If Func^>' ' Then S := StrPas(Func) + '('
                 Else S := StrPas(Func+2) + '(';
    If SubExprList^.GetCount>0 Then
       Begin
            For I := 1 to SubExprList^.GetCount-1 do
                S := S + RootObjPtr(SubExprList^.Get(I))^._write + ',';
            S := S + RootObjPtr(SubExprList^.Get(SubExprList^.GetCount))^._write;
```

```
        End;
     _write := S + ')';
End;

{ FunctionObj }

Constructor FunctionObj.Init (aName : PChar; aPriority : Byte; aAssoc : AssocType;
                              aFormalParamList : ParamListObjPtr);
Begin
     Inherited Init;
     Name := StrNew (aName);
     FormalParamList := aFormalParamList;
     Priority := aPriority;
     Assoc := aAssoc;
End;

Destructor FunctionObj.Done;

Begin
     StrDispose (Name);
     If FormalParamList<>nil Then FormalParamList^.Free;
     Inherited Done;
End;

Function FunctionObj.Copy : Pointer;

Begin
     If FormalParamList<>nil Then
        Copy := New (FunctionObjPtr,Init(Name,Priority,Assoc,FormalParamList^.Copy))
     Else
        Copy := New (FunctionObjPtr,Init(Name,Priority,Assoc,nil));
End;

Function FunctionObj.MakeParamList (ActualParamList : ParamListObjPtr;
                                    SubExprList : ListObjPtr) : ParamListObjPtr;
Var N,M,M1 : LongInt;
    L : ParamListObjPtr;

Begin
     L := New(ParamListObjPtr,Init);
     If FormalParamList<>nil Then M := FormalParamList^.GetCount Else M := 0;
     M1 := M;
     N := SubExprList^.GetCount;
     If N<M Then M := N Else M1 := N;
     For N := 1 to M do
         L^.Add(New(ParamObjPtr,Init(ParamObjPtr(FormalParamList^.Get(N))^.Name,
                                     ValueObjPtr(SubExprList^.Get(N))^.Calc(ActualParamList))));
     If SubExprList^.GetCount>=M1 Then
        For N := M+1 to M1 do
            L^.Add(New(ParamObjPtr,Init(nil,ValueObjPtr(SubExprList^.Get(N))^.Calc(ActualParamList))));
     L^.Add(New(ParamObjPtr,Init('_ParamsCount',New(ExprRealObjPtr,Init(M1)))));
     For N := 1 to ActualParamList^.GetCount do L^.Add(ParamObjPtr(ActualParamList^.Get(N))^.Copy);
     MakeParamList := L;
End;

Function FunctionObj._write : String;

Var S : String;

Begin
     S := FormalParamList^._write;
     _write := StrPas(Name)+'('+System.Copy(S,2,Byte(S[0])-2)+')';
End;

{ InternalFunctionObj }

{ UserFunctionObj }

Constructor UserFunctionObj.Init (aName : PChar; aPriority : Word; aAssoc : AssocType;
                                  aFormalParamList : ParamListObjPtr;
                                  aExpr : ExpressionObjPtr);
Begin
     Inherited Init (aName,aPriority,aAssoc,aFormalParamList);
     Expr := aExpr;
End;

Destructor UserFunctionObj.Done;
```

```
Begin
     If Expr<>nil Then Expr^.Free;
     Inherited Done;
End;

Function UserFunctionObj.Copy : Pointer;

Var aExpr : ExpressionObjPtr;

Begin
     If FormalParamList<>nil Then
        Copy := New (UserFunctionObjPtr,Init(Name,Priority,Assoc,FormalParamList^.Copy,Expr^.Copy))
     Else
        Copy := New (UserFunctionObjPtr,Init(Name,Priority,Assoc,nil,Expr^.Copy));
End;

Function UserFunctionObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Begin
     Calc := Expr^.Calc (ActualParamList);
End;

Function UserFunctionObj._write : String;

Var S : String;

Begin
     S := FormalParamList^._write;
     _write := StrPas(Name)+'('+System.Copy(S,2,Byte(S[0])-2)+')='+Expr^._write;
End;

{}

Const UnitActive : Byte = 0;  { семафор за инициализация на модула }

Procedure InitUnit;

Begin
     Inc (UnitActive);
     If UnitActive>1 Then Exit;

     uL1.InitUnit;
     FuncList := New(ListObjPtr,Init);
     uL2_1.InitUnit;
End;

Procedure DoneUnit;

Begin
     Dec (UnitActive);
     If UnitActive>0 Then Exit;

     uL2_1.DoneUnit;
     If FuncList<>nil Then FuncList^.Free;
     uL1.DoneUnit;
End;


End.
```

Изходен текст на модула uL2_1

```
Unit uL2_1;

Interface

Uses uL1,uL2,Strings; { Използва: модул ниво първо;
                                  модул ниво второ;
                                  модул за работа с Null-низове. }
```

```
Type ISelectObjPtr = ^ISelectObj; { избор на елемент от списък или пресмятане стойността на свойство }
     ISelectObj = Object(InternalFunctionObj)
        Constructor Init;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
        Function  MakeParamList (ActualParamList : ParamListObjPtr;
                                 SubExprList : ListObjPtr) : ParamListObjPtr; virtual;
     End;

     IPlusObjPtr = ^IPlusObj; { събиране }
     IPlusObj = Object(InternalFunctionObj)
        Constructor Init;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
     End;

     IConcatObjPtr = ^IConcatObj; { слепване на списъци (наследяване на свойства между обекти) }
     IConcatObj = Object(InternalFunctionObj)
        Constructor Init;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
     End;

     IMinusObjPtr = ^IMinusObj; { изваждане }
     IMinusObj = Object(InternalFunctionObj)
        Constructor Init;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
     End;

     IMulObjPtr = ^IMulObj; { умножение }
     IMulObj = Object(InternalFunctionObj)
        Constructor Init;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
     End;

     IDivideObjPtr = ^IDivideObj; { деление }
     IDivideObj = Object(InternalFunctionObj)
        Constructor Init;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
     End;

     IUMinusObjPtr = ^IUMinusObj; { унарен минус }
     IUMinusObj = Object(InternalFunctionObj)
        Constructor Init;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
     End;

     IPowerObjPtr = ^IPowerObj; { степенуване }
     IPowerObj = Object(InternalFunctionObj)
        Constructor Init;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
     End;

     IFactorialObjPtr = ^IFactorialObj; { факториел }
     IFactorialObj = Object(InternalFunctionObj)
        Constructor Init;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
     End;

     IModObjPtr = ^IModObj; { остатък от деление }
     IModObj = Object(InternalFunctionObj)
        Constructor Init;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
     End;

     IDivObjPtr = ^IDivObj; { целочислено деление }
     IDivObj = Object(InternalFunctionObj)
        Constructor Init;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
     End;

     INotObjPtr = ^INotObj; { логическо отрицание }
     INotObj = Object(InternalFunctionObj)
        Constructor Init;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
     End;

     IAndObjPtr = ^IAndObj; { конюнкция }
     IAndObj = Object(InternalFunctionObj)
        Constructor Init;
```

```
   Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
End;

IOrObjPtr = ^IOrObj; { дизюнкция }
IOrObj = Object(InternalFunctionObj)
   Constructor Init;
   Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
End;

IXorObjPtr = ^IXorObj; { изключваща дизюнкция }
IXorObj = Object(InternalFunctionObj)
   Constructor Init;
   Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
End;

IEqualObjPtr = ^IEqualObj; { равенство }
IEqualObj = Object(InternalFunctionObj)
   Constructor Init;
   Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
End;

ILetObjPtr = ^ILetObj; { дефиниране на функции }
ILetObj = Object(InternalFunctionObj)
   Constructor Init;
   Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
   Function  MakeParamList (ActualParamList : ParamListObjPtr;
                            SubExprList : ListObjPtr) : ParamListObjPtr; virtual;
End;

INoEqualObjPtr = ^INoEqualObj; { различно }
INoEqualObj = Object(InternalFunctionObj)
   Constructor Init;
   Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
End;

ILessOrEqualObjPtr = ^ILessOrEqualObj; { по-малко или равно }
ILessOrEqualObj = Object(InternalFunctionObj)
   Constructor Init;
   Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
End;

IGreatOrEqualObjPtr = ^IGreatOrEqualObj; { по-голямо или равно }
IGreatOrEqualObj = Object(InternalFunctionObj)
   Constructor Init;
   Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
End;

ILessObjPtr = ^ILessObj; { по-малко }
ILessObj = Object(InternalFunctionObj)
   Constructor Init;
   Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
End;

IGreatObjPtr = ^IGreatObj; { по-голямо }
IGreatObj = Object(InternalFunctionObj)
   Constructor Init;
   Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
End;

INewObjPtr = ^INewObj; { дефиниране на нова функция }
INewObj = Object(InternalFunctionObj)
   Constructor Init;
   Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
End;

IGetObjPtr = ^IGetObj; { стойност на израз (идентитет) }
IGetObj = Object(InternalFunctionObj)
   Constructor Init;
   Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
End;

IDelObjPtr = ^IDelObj; { премахва функция }
IDelObj = Object(InternalFunctionObj)
   Constructor Init;
   Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
   Function  MakeParamList (ActualParamList : ParamListObjPtr;
                            SubExprList : ListObjPtr) : ParamListObjPtr; virtual;
```

```
      End;

      IPropertyObjPtr = ^IPropertyObj; { създава свойство на обект }
      IPropertyObj = Object(InternalFunctionObj)
         Constructor Init;
         Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
      End;
{}
      IOddObjPtr = ^IOddObj; { нечетно? }
      IOddObj = Object(InternalFunctionObj)
         Constructor Init;
         Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
      End;

      ISqrObjPtr = ^ISqrObj; { квадрат }
      ISqrObj = Object(InternalFunctionObj)
         Constructor Init;
         Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
      End;

      ISqrtObjPtr = ^ISqrtObj; { квадратен корен }
      ISqrtObj = Object(InternalFunctionObj)
         Constructor Init;
         Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
      End;

      ISinObjPtr = ^ISinObj; { синус }
      ISinObj = Object(InternalFunctionObj)
         Constructor Init;
         Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
      End;

      ICosObjPtr = ^ICosObj; { косинус }
      ICosObj = Object(InternalFunctionObj)
         Constructor Init;
         Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
      End;

      IArcTanObjPtr = ^IArcTanObj; { аркустангенс }
      IArcTanObj = Object(InternalFunctionObj)
         Constructor Init;
         Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
      End;

      IPiObjPtr = ^IPiObj; { пи }
      IPiObj = Object(InternalFunctionObj)
         Constructor Init;
         Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
      End;

      IExpObjPtr = ^IExpObj; { експоненциал }
      IExpObj = Object(InternalFunctionObj)
         Constructor Init;
         Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
      End;

      ILnObjPtr = ^ILnObj; { натурален логаритъм }
      ILnObj = Object(InternalFunctionObj)
         Constructor Init;
         Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
      End;

      IAbsObjPtr = ^IAbsObj; { абсолютна стойност (модул) }
      IAbsObj = Object(InternalFunctionObj)
         Constructor Init;
         Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
      End;

      ISgnObjPtr = ^ISgnObj; { знак (сигнум) }
      ISgnObj = Object(InternalFunctionObj)
         Constructor Init;
         Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
      End;

      IIntObjPtr = ^IIntObj; { цяла част на число }
      IIntObj = Object(InternalFunctionObj)
         Constructor Init;
```

```
      Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
   End;

   IRoundObjPtr = ^IRoundObj; { закръгляне до цяло }
   IRoundObj = Object(InternalFunctionObj)
      Constructor Init;
      Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
   End;

   IFracObjPtr = ^IFracObj; { дробна част на число }
   IFracObj = Object(InternalFunctionObj)
      Constructor Init;
      Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
   End;

   IRandomObjPtr = ^IRandomObj; { случайно число }
   IRandomObj = Object(InternalFunctionObj)
      Constructor Init;
      Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
   End;

   IIfObjPtr = ^IIfObj; { условно изчисляване на изрази }
   IIfObj = Object(InternalFunctionObj)
      Constructor Init;
      Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
      Function  MakeParamList (ActualParamList : ParamListObjPtr;
                               SubExprList : ListObjPtr) : ParamListObjPtr; virtual;
   End;

   ICaseObjPtr = ^ICaseObj; { изчисляване на израз с даден номер }
   ICaseObj = Object(InternalFunctionObj)
      Constructor Init;
      Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
      Function  MakeParamList (ActualParamList : ParamListObjPtr;
                               SubExprList : ListObjPtr) : ParamListObjPtr; virtual;
   End;

   IMinObjPtr = ^IMinObj; { минимална стойност }
   IMinObj = Object(InternalFunctionObj)
      Constructor Init;
      Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
   End;

   IMaxObjPtr = ^IMaxObj; { максимална стойност }
   IMaxObj = Object(InternalFunctionObj)
      Constructor Init;
      Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
   End;

Procedure InitUnit;
Procedure DoneUnit;

Implementation

{ ISelectObj }

Constructor ISelectObj.Init;

Begin
   Inherited Init ('.',8,Assoc_yfx,nil);
End;

Function ISelectObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var W : LongInt;
    O : RootObjPtr; EF : ExprFuncObjPtr absolute O;
    O1 : RootObjPtr;
    L : ListObjPtr;
    PL : ParamListObjPtr;
    F : FunctionObjPtr;

Begin
   Calc := nil;
   L := ListObjPtr(ParamObjPtr(ActualParamList^.Get(1))^.Value);
   O := ParamObjPtr(ActualParamList^.Get(2))^.Value;
   If (L<>nil) and (O<>nil) and (TypeOf(L^)=TypeOf(ListObj)) Then
      Begin
```

```
            If TypeOf(O^)=TypeOf(ExprRealObj) Then
               Begin
                    W := Trunc(ExprRealObjPtr(O)^.Value);
                    O1 := L^.Get(W);
                    If O1<>nil Then Calc := O1^.Copy;
               End
            Else If TypeOf(O^)=TypeOf(ExprFuncObj) Then
                    Begin
                         For W := 1 to L^.GetCount do
                             Begin
                                  F := L^.Get (W);
                                  If (F<>nil) and (TypeOf(F^)=TypeOf(UserFunctionObj)) and
                                     (StrIComp(EF^.Func,F^.Name)=0) Then
                                     Begin
                                          PL := F^.MakeParamList (ActualParamList,EF^.SubExprList);
                                          Calc := F^.Calc (PL);
                                          PL^.Free;
                                          Break;
                                     End;
                             End;
                    End;
       End;
End;

Function ISelectObj.MakeParamList (ActualParamList : ParamListObjPtr;
                                   SubExprList : ListObjPtr) : ParamListObjPtr;
Var L : ParamListObjPtr;

Begin
     L := ActualParamList^.Copy;
     L^.Insert (1,New(ParamObjPtr,Init('Self',ValueObjPtr(SubExprList^.Get(1))^.Calc(ActualParamList))));
     L^.Insert (2,New(ParamObjPtr,Init(nil,ValueObjPtr(SubExprList^.Get(2))^.Copy)));
     MakeParamList := L;
End;

{ IPlusObj }

Constructor IPlusObj.Init;

Begin
     Inherited Init ('+',5,Assoc_yfx,nil);
End;

Function IPlusObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var I,M : LongInt;
    O1 : RootObjPtr; L1 : ListObjPtr absolute O1;
    O2 : RootObjPtr; L2 : ListObjPtr absolute O2;
    L : ListObjPtr;
    R1,R2 : ExprRealObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If TypeOf(O2^)=TypeOf(ExprRealObj) Then
                   Calc := New(ExprRealObjPtr,Init(ExprRealObjPtr(O1)^.Value+ExprRealObjPtr(O2)^.Value));
           End
        Else If TypeOf(O1^)=TypeOf(ListObj) Then
                Begin
                     If TypeOf(O2^)=TypeOf(ListObj) Then
                        Begin
                             New (L,Init);
                             M := L1^.GetCount;
                             If M<L2^.GetCount Then M := L2^.GetCount;
                             For I := 1 to M do
                                 Begin
                                      R1 := L1^.Get (I);
                                      R2 := L2^.Get (I);
                                      If R1=nil Then L^.Add (R2^.Copy)
                                      Else If R2=nil Then L^.Add (R1^.Copy)
                                           Else If (TypeOf(R1^)=TypeOf(ExprRealObj)) and
                                                   (TypeOf(R2^)=TypeOf(ExprRealObj)) Then
                                                   L^.Add (New(ExprRealObjPtr,Init(R1^.Value+R2^.Value)))
```

```
                                        Else
                                           Begin
                                                L^.Free;
                                                Break;
                                           End;
                                   End;
                              Calc := L;
                         End;
                End;
End;

{ IConcatObj }

Constructor IConcatObj.Init;

Begin
     Inherited Init ('&',5,Assoc_yfx,nil);
End;

Function IConcatObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var I : LongInt;
    O1 : RootObjPtr; L1 : ListObjPtr absolute O1;
    O2 : RootObjPtr; L2 : ListObjPtr absolute O2;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ListObj) Then
           Begin
                If TypeOf(O2^)=TypeOf(ListObj) Then
                   Begin
                        L1 := L1^.Copy;
                        For I := 1 to L2^.GetCount do L1^.Add(RootObjPtr(L2^.Get(I))^.Copy);
                        Calc := L1;
                   End;
           End;
End;

{ IMinusObj }

Constructor IMinusObj.Init;

Begin
     Inherited Init ('-',5,Assoc_yfx,nil);
End;

Function IMinusObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var I,M : LongInt;
    O1 : RootObjPtr; L1 : ListObjPtr absolute O1;
    O2 : RootObjPtr; L2 : ListObjPtr absolute O2;
    L : ListObjPtr;
    R1,R2 : ExprRealObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If TypeOf(O2^)=TypeOf(ExprRealObj) Then
                   Calc := New(ExprRealObjPtr,Init(ExprRealObjPtr(O1)^.Value-ExprRealObjPtr(O2)^.Value));
           End
        Else If TypeOf(O1^)=TypeOf(ListObj) Then
                Begin
                     If TypeOf(O2^)=TypeOf(ListObj) Then
                        Begin
                             New (L,Init);
                             M := L1^.GetCount;
                             If M<L2^.GetCount Then M := L2^.GetCount;
                             For I := 1 to M do
                                 Begin
                                      R1 := L1^.Get (I);
```

```
                                        R2 := L2^.Get (I);
                                        If (R1=nil) and (TypeOf(R2^)=TypeOf(ExprRealObj)) Then
                                           L^.Add (New(ExprRealObjPtr,Init(-R2^.Value)))
                                        Else If R2=nil Then L^.Add (R1^.Copy)
                                             Else If (TypeOf(R1^)=TypeOf(ExprRealObj)) and
                                                     (TypeOf(R2^)=TypeOf(ExprRealObj)) Then
                                                     L^.Add (New(ExprRealObjPtr,Init(R1^.Value-R2^.Value)))
                                                  Else
                                                     Begin
                                                          L^.Free;
                                                          Break;
                                                     End;
                                    End;
                               Calc := L;
                          End;
                End;
End;

{ IMulObj }

Constructor IMulObj.Init;

Begin
     Inherited Init ('*',6,Assoc_yfx,nil);
End;

Function IMulObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var I : LongInt;
    O1 : RootObjPtr; L1 : ListObjPtr absolute O1;
    O2 : RootObjPtr; L2 : ListObjPtr absolute O2;
    O3 : RootObjPtr;
    O4 : RootObjPtr;
    S : Real;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If TypeOf(O2^)=TypeOf(ExprRealObj) Then
                   Calc := New(ExprRealObjPtr,Init(ExprRealObjPtr(O1)^.Value*ExprRealObjPtr(O2)^.Value));
           End
        Else If TypeOf(O1^)=TypeOf(ListObj) Then
                Begin
                     If TypeOf(O2^)=TypeOf(ListObj) Then
                        Begin
                             S := 0;
                             For I := 1 to L1^.GetCount do
                                 Begin
                                      O3 := L1^.Get (I);
                                      O4 := L2^.Get (I);
                                      If (O3<>nil) and (O4<>nil) and
                                         (TypeOf(O3^)=TypeOf(ExprRealObj)) and
                                         (TypeOf(O4^)=TypeOf(ExprRealObj))
                                         Then S := S+ExprRealObjPtr(O3)^.Value*ExprRealObjPtr(O4)^.Value;
                                 End;
                             Calc := New(ExprRealObjPtr,Init(S));
                        End;
                End;
End;

{ IDivideObj }

Constructor IDivideObj.Init;

Begin
     Inherited Init ('/',6,Assoc_yfx,nil);
End;

Function IDivideObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;
    O2 : RootObjPtr;
```

```
Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If TypeOf(O2^)=TypeOf(ExprRealObj) Then
                   Calc := New(ExprRealObjPtr,Init(ExprRealObjPtr(O1)^.Value/ExprRealObjPtr(O2)^.Value));
           End;
End;

{ IUMinusObj }

Constructor IUMinusObj.Init;

Begin
     Inherited Init ('~',8,Assoc_fx,nil);
End;

Function IUMinusObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                Calc := New(ExprRealObjPtr,Init(-ExprRealObjPtr(O1)^.Value));
           End;
End;

{ IPowerObj }

Constructor IPowerObj.Init;

Begin
     Inherited Init ('^',7,Assoc_yfx,nil);
End;

Function IPowerObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;
    O2 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If TypeOf(O2^)=TypeOf(ExprRealObj) Then
                   Calc := New(ExprRealObjPtr,Init(Exp(Ln(ExprRealObjPtr(O1)^.Value)
                                                       *ExprRealObjPtr(O2)^.Value)));
           End;
End;

{ IFactorialObj }

Constructor IFactorialObj.Init;

Begin
     Inherited Init ('!',7,Assoc_yf,nil);
End;

Function IFactorialObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var I : LongInt;
    O1 : RootObjPtr;
    R : Real;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
```

```
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                R := 1;
                For I := 2 to Trunc(ExprRealObjPtr(O1)^.Value) do R := I*R;
                Calc := New(ExprRealObjPtr,Init(R));
           End;
End;

{ IModObj }

Constructor IModObj.Init;

Begin
     Inherited Init ('Mod',6,Assoc_yfx,nil);
End;

Function IModObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var I : LongInt;
    O1 : RootObjPtr;
    O2 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If TypeOf(O2^)=TypeOf(ExprRealObj) Then
                   Begin
                        I := Trunc(ExprRealObjPtr(O1)^.Value);
                        I := I mod Trunc(ExprRealObjPtr(O2)^.Value);
                        Calc := New(ExprRealObjPtr,Init(I));
                   End;
           End;
End;

{ IDivObj }

Constructor IDivObj.Init;

Begin
     Inherited Init ('Div',6,Assoc_yfx,nil);
End;

Function IDivObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var I : LongInt;
    O1 : RootObjPtr;
    O2 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If TypeOf(O2^)=TypeOf(ExprRealObj) Then
                   Begin
                        I := Trunc(ExprRealObjPtr(O1)^.Value);
                        I := I div Trunc(ExprRealObjPtr(O2)^.Value);
                        Calc := New(ExprRealObjPtr,Init(I));
                   End;
           End;
End;

{ INotObj }

Constructor INotObj.Init;

Begin
     Inherited Init ('Not',8,Assoc_fy,nil);
End;
```

```
Function INotObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If ExprRealObjPtr(O1)^.Value=0
                   Then Calc := New(ExprRealObjPtr,Init(1))
                   Else Calc := New(ExprRealObjPtr,Init(0));
           End;
End;

{ IAndObj }

Constructor IAndObj.Init;

Begin
     Inherited Init ('And',6,Assoc_yfx,nil);
End;

Function IAndObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;
    O2 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If TypeOf(O2^)=TypeOf(ExprRealObj) Then
                   Begin
                        If (ExprRealObjPtr(O1)^.Value<>0) and (ExprRealObjPtr(O2)^.Value<>0)
                           Then Calc := New(ExprRealObjPtr,Init(1))
                           Else Calc := New(ExprRealObjPtr,Init(0));
                   End;
           End;
End;

{ IOrObj }

Constructor IOrObj.Init;

Begin
     Inherited Init ('Or',5,Assoc_yfx,nil);
End;

Function IOrObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;
    O2 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If TypeOf(O2^)=TypeOf(ExprRealObj) Then
                   Begin
                        If (ExprRealObjPtr(O1)^.Value<>0) or (ExprRealObjPtr(O2)^.Value<>0)
                           Then Calc := New(ExprRealObjPtr,Init(1))
                           Else Calc := New(ExprRealObjPtr,Init(0));
                   End;
           End;
End;

{ IXorObj }

Constructor IXorObj.Init;
```

```
Begin
     Inherited Init ('Xor',5,Assoc_yfx,nil);
End;

Function IXorObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;
    O2 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If TypeOf(O2^)=TypeOf(ExprRealObj) Then
                   Begin
                        If (ExprRealObjPtr(O1)^.Value<>0) xor (ExprRealObjPtr(O2)^.Value<>0)
                           Then Calc := New(ExprRealObjPtr,Init(1))
                           Else Calc := New(ExprRealObjPtr,Init(0));
                   End;
           End;
End;

{ IEqualObj }

Constructor IEqualObj.Init;

Begin
     Inherited Init ('==',4,Assoc_xfx,nil);
End;

Function IEqualObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var I : LongInt;
    O1 : RootObjPtr; L1 : ListObjPtr absolute O1;
    O2 : RootObjPtr; L2 : ListObjPtr absolute O2;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If TypeOf(O2^)=TypeOf(ExprRealObj) Then
                   Begin
                        If ExprRealObjPtr(O1)^.Value=ExprRealObjPtr(O2)^.Value
                           Then Calc := New(ExprRealObjPtr,Init(1))
                           Else Calc := New(ExprRealObjPtr,Init(0));
                   End;
           End
        Else If TypeOf(O1^)=TypeOf(ListObj) Then
                Begin
                     If TypeOf(O2^)=TypeOf(ListObj) Then
                        Begin
                             {???}
                             Calc := nil;
                        End;
                End;
End;

{ ILetObj }

Constructor ILetObj.Init;

Begin
     Inherited Init ('=',2,Assoc_xfx,nil);
End;

Function ILetObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr; H : ExprFuncObjPtr absolute O1;
    O2 : RootObjPtr;
    FPL : ParamListObjPtr;
```

```
    V : ExprVarObjPtr;
    I : LongInt;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ExprFuncObj) Then
           Begin
                New (FPL,Init);
                For I := 1 to H^.SubExprList^.GetCount do
                    Begin
                         V := H^.SubExprList^.Get (I);
                         If (V<>nil) and (TypeOf(V^)=TypeOf(ExprVarObj)) Then
                            FPL^.Add (New(ParamObjPtr,Init(V^.Name,nil)))
                         Else
                            Begin
                                 FPL^.Free;
                                 Exit;
                            End;
                    End;
                Calc := New(UserFunctionObjPtr,
                            Init(ExprFuncObjPtr(O1)^.Func,0,Assoc_functional,
                            FPL,ExpressionObjPtr(O2)^.Copy));
           End;
End;

Function ILetObj.MakeParamList (ActualParamList : ParamListObjPtr;
                                SubExprList : ListObjPtr) : ParamListObjPtr;
Var L : ParamListObjPtr;

Begin
     L := ActualParamList^.Copy;
     L^.Insert (1,New(ParamObjPtr,Init(nil,ValueObjPtr(SubExprList^.Get(1))^.Copy)));
     L^.Insert (2,New(ParamObjPtr,Init(nil,ValueObjPtr(SubExprList^.Get(2))^.Copy)));
     MakeParamList := L;
End;

{ INoEqualObj }

Constructor INoEqualObj.Init;

Begin
     Inherited Init ('<>',4,Assoc_xfx,nil);
End;

Function INoEqualObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var I : LongInt;
    O1 : RootObjPtr; L1 : ListObjPtr absolute O1;
    O2 : RootObjPtr; L2 : ListObjPtr absolute O2;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If TypeOf(O2^)=TypeOf(ExprRealObj) Then
                   Begin
                        If ExprRealObjPtr(O1)^.Value<>ExprRealObjPtr(O2)^.Value
                           Then Calc := New(ExprRealObjPtr,Init(1))
                           Else Calc := New(ExprRealObjPtr,Init(0));
                   End;
           End
        Else If TypeOf(O1^)=TypeOf(ListObj) Then
                Begin
                     If TypeOf(O2^)=TypeOf(ListObj) Then
                        Begin
                             {???}
                             Calc := nil;
                        End;
                End;
End;
```

```
{ ILessOrEqualObj }

Constructor ILessOrEqualObj.Init;

Begin
     Inherited Init ('<=',4,Assoc_xfx,nil);
End;

Function ILessOrEqualObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;
    O2 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If TypeOf(O2^)=TypeOf(ExprRealObj) Then
                   Begin
                        If ExprRealObjPtr(O1)^.Value<=ExprRealObjPtr(O2)^.Value
                           Then Calc := New(ExprRealObjPtr,Init(1))
                           Else Calc := New(ExprRealObjPtr,Init(0));
                   End;
           End;
End;

{ IGreatOrEqualObj }

Constructor IGreatOrEqualObj.Init;

Begin
     Inherited Init ('>=',4,Assoc_xfx,nil);
End;

Function IGreatOrEqualObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;
    O2 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If TypeOf(O2^)=TypeOf(ExprRealObj) Then
                   Begin
                        If ExprRealObjPtr(O1)^.Value>=ExprRealObjPtr(O2)^.Value
                           Then Calc := New(ExprRealObjPtr,Init(1))
                           Else Calc := New(ExprRealObjPtr,Init(0));
                   End;
           End;
End;

{ ILessObj }

Constructor ILessObj.Init;

Begin
     Inherited Init ('<',4,Assoc_xfx,nil);
End;

Function ILessObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;
    O2 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
```

```
           Begin
                If TypeOf(O2^)=TypeOf(ExprRealObj) Then
                   Begin
                        If ExprRealObjPtr(O1)^.Value<ExprRealObjPtr(O2)^.Value
                           Then Calc := New(ExprRealObjPtr,Init(1))
                           Else Calc := New(ExprRealObjPtr,Init(0));
                   End;
           End;
End;

{ IGreatObj }

Constructor IGreatObj.Init;

Begin
     Inherited Init ('>',4,Assoc_xfx,nil);
End;

Function IGreatObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;
    O2 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value;
     If (O1<>nil) and (O2<>nil) Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If TypeOf(O2^)=TypeOf(ExprRealObj) Then
                   Begin
                        If ExprRealObjPtr(O1)^.Value>ExprRealObjPtr(O2)^.Value
                           Then Calc := New(ExprRealObjPtr,Init(1))
                           Else Calc := New(ExprRealObjPtr,Init(0));
                   End;
           End;
End;

{ INewObj }

Constructor INewObj.Init;

Begin
     Inherited Init ('New',1,Assoc_fx,nil);
End;

Function INewObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(UserFunctionObj) Then
           Begin
                DefineFunction (FunctionObjPtr(O1)^.Copy);
                Calc := New(ExprRealObjPtr,Init(1));
           End;
End;

{ IGetObj }

Constructor IGetObj.Init;

Begin
     Inherited Init ('Get',1,Assoc_fx,nil);
End;

Function IGetObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
```

```
     If O1<>nil Then Calc := O1^.Copy;
End;

{ IDelObj }

Constructor IDelObj.Init;

Begin
     Inherited Init ('Del',1,Assoc_fx,nil);
End;

Function IDelObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;
    DName : PChar;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        Begin
             If TypeOf(O1^)=TypeOf(ExprFuncObj) Then DName := ExprFuncObjPtr(O1)^.Func
             Else DName := '';

             If DeleteFunction (DName)
                Then Calc := New(ExprRealObjPtr,Init(1))
                Else Calc := New(ExprRealObjPtr,Init(0));
        End;
End;

Function IDelObj.MakeParamList (ActualParamList : ParamListObjPtr;
                                SubExprList : ListObjPtr) : ParamListObjPtr;
Var L : ParamListObjPtr;

Begin
     L := ActualParamList^.Copy;
     L^.Insert (1,New(ParamObjPtr,Init(nil,ValueObjPtr(SubExprList^.Get(1))^.Copy)));
     MakeParamList := L;
End;

{ IPropertyObj }

Constructor IPropertyObj.Init;

Begin
     Inherited Init ('Property',1,Assoc_fx,nil);
End;

Function IPropertyObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr; F : UserFunctionObjPtr absolute O1;
    E1 : ExpressionObjPtr;

  Procedure PartialCalc (Var E : RootObjPtr);

  Var P,P1 : RootObjPtr;
      I : LongInt;
      L : ListObjPtr;

  Begin
       If E<>nil Then
          Begin
               If TypeOf(E^)=TypeOf(ExprListObj) Then
                  Begin
                       L := ExprListObjPtr(E)^.Value;
                       If L<>nil Then
                       For I := 1 to L^.GetCount do
                           Begin
                                P := L^.Get (I);
                                P1 := P; PartialCalc (P);
                                If P<>P1 Then L^.Replace (I,P);
                           End;
                  End
              Else If TypeOf(E^)=TypeOf(ExprFuncObj) Then
                       Begin
                            L := ExprFuncObjPtr(E)^.SubExprList;
                            If L<>nil Then
```

```
                              For I := 1 to L^.GetCount do
                                  Begin
                                       P := L^.Get (I);
                                       P1 := P; PartialCalc (P);
                                       If P<>P1 Then L^.Replace (I,P);
                                  End;
                         End
                      Else If TypeOf(E^)<>TypeOf(UserFunctionObj) Then
                              Begin
                                   P := ValueObjPtr(E)^.Calc (ActualParamList);
                                   If P<>nil Then
                                      Begin
                                           If TypeOf(P^)=TypeOf(ListObj) Then
                                              P := New(ExprListObjPtr,Init(ListObjPtr(P)));
                                           E := P;
                                      End;
                              End;
          End;
  End;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(UserFunctionObj) Then
           Begin
                F := F^.Copy;
                E1 := F^.Expr; PartialCalc (RootObjPtr(F^.Expr));
                If F^.Expr<>E1 Then E1^.Free;
                Calc := F;
           End;
End;

{ IOddObj }

Constructor IOddObj.Init;

Begin
     Inherited Init ('Odd',0,Assoc_Functional,nil);
End;

Function IOddObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If Odd(Trunc(ExprRealObjPtr(O1)^.Value))
                   Then Calc := New(ExprRealObjPtr,Init(1))
                   Else Calc := New(ExprRealObjPtr,Init(0));
           End;
End;

{ ISqrObj }

Constructor ISqrObj.Init;

Begin
     Inherited Init ('Sqr',0,Assoc_Functional,nil);
End;

Function ISqrObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                Calc := New(ExprRealObjPtr,Init(Sqr(ExprRealObjPtr(O1)^.Value)));
           End;
```

```
End;

{ ISqrtObj }

Constructor ISqrtObj.Init;

Begin
     Inherited Init ('Sqrt',0,Assoc_Functional,nil);
End;

Function ISqrtObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                Calc := New(ExprRealObjPtr,Init(Sqrt(ExprRealObjPtr(O1)^.Value)));
           End;
End;

{ ISinObj }

Constructor ISinObj.Init;

Begin
     Inherited Init ('Sin',0,Assoc_Functional,nil);
End;

Function ISinObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                Calc := New(ExprRealObjPtr,Init(Sin(ExprRealObjPtr(O1)^.Value)));
           End;
End;

{ ICosObj }

Constructor ICosObj.Init;

Begin
     Inherited Init ('Cos',0,Assoc_Functional,nil);
End;

Function ICosObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                Calc := New(ExprRealObjPtr,Init(Cos(ExprRealObjPtr(O1)^.Value)));
           End;
End;

{ IArcTanObj }

Constructor IArcTanObj.Init;

Begin
     Inherited Init ('ArcTan',0,Assoc_Functional,nil);
End;

Function IArcTanObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;
```

```
Var O1 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                Calc := New(ExprRealObjPtr,Init(ArcTan(ExprRealObjPtr(O1)^.Value)));
           End;
End;

{ IPiObj }

Constructor IPiObj.Init;

Begin
     Inherited Init ('Pi',0,Assoc_Functional,nil);
End;

Function IPiObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Begin
     Calc := New(ExprRealObjPtr,Init(Pi));
End;

{ IExpObj }

Constructor IExpObj.Init;

Begin
     Inherited Init ('Exp',0,Assoc_Functional,nil);
End;

Function IExpObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                Calc := New(ExprRealObjPtr,Init(Exp(ExprRealObjPtr(O1)^.Value)));
           End;
End;

{ ILnObj }

Constructor ILnObj.Init;

Begin
     Inherited Init ('Ln',0,Assoc_Functional,nil);
End;

Function ILnObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                Calc := New(ExprRealObjPtr,Init(Ln(ExprRealObjPtr(O1)^.Value)));
           End;
End;

{ IAbsObj }

Constructor IAbsObj.Init;

Begin
     Inherited Init ('Abs',0,Assoc_Functional,nil);
```

```
End;

Function IAbsObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                Calc := New(ExprRealObjPtr,Init(Abs(ExprRealObjPtr(O1)^.Value)));
           End;
End;

{ ISgnObj }

Constructor ISgnObj.Init;

Begin
     Inherited Init ('Sgn',0,Assoc_Functional,nil);
End;

Function ISgnObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If ExprRealObjPtr(O1)^.Value<0
                   Then Calc := New(ExprRealObjPtr,Init(-1))
                   Else If ExprRealObjPtr(O1)^.Value>0 Then Calc := New(ExprRealObjPtr,Init(1))
                        Else Calc := New(ExprRealObjPtr,Init(0));
           End;
End;

{ IIntObj }

Constructor IIntObj.Init;

Begin
     Inherited Init ('Int',0,Assoc_Functional,nil);
End;

Function IIntObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                Calc := New(ExprRealObjPtr,Init(Int(ExprRealObjPtr(O1)^.Value)));
           End;
End;

{ IRoundObj }

Constructor IRoundObj.Init;

Begin
     Inherited Init ('Round',0,Assoc_Functional,nil);
End;

Function IRoundObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     Calc := nil;
```

```
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                Calc := New(ExprRealObjPtr,Init(Round(ExprRealObjPtr(O1)^.Value)));
           End;
End;

{ IFracObj }

Constructor IFracObj.Init;

Begin
     Inherited Init ('Frac',0,Assoc_Functional,nil);
End;

Function IFracObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                Calc := New(ExprRealObjPtr,Init(Frac(ExprRealObjPtr(O1)^.Value)));
           End;
End;

{ IRandomObj }

Constructor IRandomObj.Init;

Begin
     Inherited Init ('Random',0,Assoc_Functional,nil);
End;

Function IRandomObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;

Begin
     Calc := New(ExprRealObjPtr,Init(Random));
End;

{ IIfObj }

Constructor IIfObj.Init;

Begin
     Inherited Init ('If',0,Assoc_Functional,nil);
End;

Function IIfObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;
    O2 : RootObjPtr;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                If ExprRealObjPtr(O1)^.Value<>0
                   Then O2 := ParamObjPtr(ActualParamList^.Get(2))^.Value
                   Else O2 := ParamObjPtr(ActualParamList^.Get(3))^.Value;
                If O2<>nil Then Calc := ValueObjPtr(O2)^.Calc(ActualParamList);
           End;
End;

Function IIfObj.MakeParamList (ActualParamList : ParamListObjPtr;
                               SubExprList : ListObjPtr) : ParamListObjPtr;
Var L : ParamListObjPtr;

Begin
```

```
     L := ActualParamList^.Copy;
     L^.Insert (1,New(ParamObjPtr,Init(nil,ValueObjPtr(SubExprList^.Get(1))^.Calc(ActualParamList))));
     L^.Insert (2,New(ParamObjPtr,Init(nil,ValueObjPtr(SubExprList^.Get(2))^.Copy)));
     L^.Insert (3,New(ParamObjPtr,Init(nil,ValueObjPtr(SubExprList^.Get(3))^.Copy)));
     MakeParamList := L;
End;

{ ICaseObj }

Constructor ICaseObj.Init;

Begin
     Inherited Init ('Case',0,Assoc_Functional,nil);
End;

Function ICaseObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;
    O2 : RootObjPtr;
    PCnt : RootObjPtr;
    N : LongInt;

Begin
     Calc := nil;
     O1 := ParamObjPtr(ActualParamList^.Get(1))^.Value;
     If O1<>nil Then
        If TypeOf(O1^)=TypeOf(ExprRealObj) Then
           Begin
                N := Trunc(ExprRealObjPtr(O1)^.Value);
                PCnt := ActualParamList^.GetParam ('_ParamsCount');
                If (Trunc(ExprRealObjPtr(PCnt)^.Value)>1) and
                   (N>=0) and (N<Trunc(ExprRealObjPtr(PCnt)^.Value)-1) Then
                   Begin
                        O2 := ActualParamList^.Get(N+1);
                        If O2<>nil Then
                           Begin
                                O2 := ParamObjPtr(O2)^.Value;
                                If O2<>nil Then Calc := ValueObjPtr(O2)^.Calc(ActualParamList);
                           End;
                   End;
           End;
End;

Function ICaseObj.MakeParamList (ActualParamList : ParamListObjPtr;
                                 SubExprList : ListObjPtr) : ParamListObjPtr;
Var L : ParamListObjPtr;
    N : LongInt;

Begin
     L := ActualParamList^.Copy;
     L^.Add(New(ParamObjPtr,Init('_ParamsCount',New(ExprRealObjPtr,Init(SubExprList^.GetCount)))));
     L^.Insert (1,New(ParamObjPtr,Init(nil,ValueObjPtr(SubExprList^.Get(1))^.Calc(ActualParamList))));
     For N := 2 to SubExprList^.GetCount do
         L^.Insert (N,New(ParamObjPtr,Init(nil,ValueObjPtr(SubExprList^.Get(N))^.Copy)));
     MakeParamList := L;
End;

{ IMinObj }

Constructor IMinObj.Init;

Begin
     Inherited Init ('Min',0,Assoc_Functional,nil);
End;

Function IMinObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;
    PCnt : RootObjPtr;
    N : LongInt;
    R : Real;

Begin
     Calc := nil;

     R := 1.7e38;
     PCnt := ActualParamList^.GetParam ('_ParamsCount');
```

```
     For N := 1 to Trunc(ExprRealObjPtr(PCnt)^.Value) do
         Begin
              O1 := ParamObjPtr(ActualParamList^.Get(N))^.Value;
              If (O1<>nil) and (TypeOf(O1^)=TypeOf(ExprRealObj)) Then
                 Begin
                      If ExprRealObjPtr(O1)^.Value<R Then R := ExprRealObjPtr(O1)^.Value;
                 End
              Else Exit;
         End;
     Calc := New(ExprRealObjPtr,Init(R));
End;

{ IMaxObj }

Constructor IMaxObj.Init;

Begin
     Inherited Init ('Max',0,Assoc_Functional,nil);
End;

Function IMaxObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Var O1 : RootObjPtr;
    PCnt : RootObjPtr;
    N : LongInt;
    R : Real;

Begin
     Calc := nil;

     R := -1.7e38;
     PCnt := ActualParamList^.GetParam ('_ParamsCount');
     For N := 1 to Trunc(ExprRealObjPtr(PCnt)^.Value) do
         Begin
              O1 := ParamObjPtr(ActualParamList^.Get(N))^.Value;
              If (O1<>nil) and (TypeOf(O1^)=TypeOf(ExprRealObj)) Then
                 Begin
                      If ExprRealObjPtr(O1)^.Value>R Then R := ExprRealObjPtr(O1)^.Value;
                 End
              Else Exit;
         End;
     Calc := New(ExprRealObjPtr,Init(R));
End;

{}

Const UnitActive : Byte = 0; { семафор за инициализация на модула }

Procedure InitUnit;

Begin
     Inc (UnitActive);
     If UnitActive>1 Then Exit;

     DefineFunction (New(ISelectObjPtr,Init));

     DefineFunction (New(IPlusObjPtr,Init));
     DefineFunction (New(IConcatObjPtr,Init));
     DefineFunction (New(IMinusObjPtr,Init));
     DefineFunction (New(IMulObjPtr,Init));
     DefineFunction (New(IDivideObjPtr,Init));
     DefineFunction (New(IUMinusObjPtr,Init));
     DefineFunction (New(IPowerObjPtr,Init));
     DefineFunction (New(IFactorialObjPtr,Init));
     DefineFunction (New(IModObjPtr,Init));
     DefineFunction (New(IDivObjPtr,Init));

     DefineFunction (New(INotObjPtr,Init));
     DefineFunction (New(IAndObjPtr,Init));
     DefineFunction (New(IOrObjPtr,Init));
     DefineFunction (New(IXorObjPtr,Init));
     DefineFunction (New(IEqualObjPtr,Init));
     DefineFunction (New(ILetObjPtr,Init));
     DefineFunction (New(INoEqualObjPtr,Init));
     DefineFunction (New(ILessOrEqualObjPtr,Init));
     DefineFunction (New(IGreatOrEqualObjPtr,Init));
     DefineFunction (New(ILessObjPtr,Init));
```

```
     DefineFunction (New(IGreatObjPtr,Init));

     DefineFunction (New(INewObjPtr,Init));
     DefineFunction (New(IGetObjPtr,Init));
     DefineFunction (New(IDelObjPtr,Init));
     DefineFunction (New(IPropertyObjPtr,Init));

     DefineFunction (New(IOddObjPtr,Init));

     DefineFunction (New(ISqrObjPtr,Init));
     DefineFunction (New(ISqrtObjPtr,Init));

     DefineFunction (New(ISinObjPtr,Init));
     DefineFunction (New(ICosObjPtr,Init));
     DefineFunction (New(IArcTanObjPtr,Init));
     DefineFunction (New(IPiObjPtr,Init));

     DefineFunction (New(IExpObjPtr,Init));
     DefineFunction (New(ILnObjPtr,Init));

     DefineFunction (New(IAbsObjPtr,Init));
     DefineFunction (New(ISgnObjPtr,Init));
     DefineFunction (New(IIntObjPtr,Init));
     DefineFunction (New(IRoundObjPtr,Init));
     DefineFunction (New(IFracObjPtr,Init));
     DefineFunction (New(IRandomObjPtr,Init));

     DefineFunction (New(IIfObjPtr,Init));
     DefineFunction (New(ICaseObjPtr,Init));
     DefineFunction (New(IMinObjPtr,Init));
     DefineFunction (New(IMaxObjPtr,Init));
End;

Procedure DoneUnit;

Begin
     Dec (UnitActive);
     If UnitActive>0 Then Exit;

End;


End.
```

Изходен текст на модула Lexical

```
Unit Lexical; { Модул лексикален анализатор }

Interface

Uses Strings,uL1,uL2; { Използва: модул за работа с Null-низове;
                                  модул ниво първо;
                                  модул ниво второ. }

Type { типове лексеми }
     LexemaType = (lex_End,lex_Ident,lex_String,lex_Number,lex_Symbol,
                   lex_FuncName,lex_Operator);
     { Обект лексема }
     LexemaObjPtr = ^LexemaObj;
     LexemaObj = Object(RootObj)
        LexType : LexemaType;
        Value : PChar;
        Constructor Init (aLexType : LexemaType; aValue : PChar);
        Destructor  Done; virtual;
     End;

     { Обект лексикален анализатор }
     LexBreakerObjPtr = ^LexBreakerObj;
     LexBreakerObj = Object(RootObj)
        InputText : PChar;                  { входен текст }
        Ptr : PChar;                        { текуща позиция в текста }
```

```
        Constructor Init (InpText : PChar);  { Конструира анализатор на текста InpText. }
        Destructor  Done; virtual;
        Function  GetLex : LexemaObjPtr;     { Отделя поредната лексема от входният текст. }
     End;


Procedure InitUnit;  { Първоначална инициализация на модула }
Procedure DoneUnit;  { Край на работата с модула }

Implementation

{ LexemaObj }

Constructor LexemaObj.Init (aLexType : LexemaType; aValue : PChar);

Begin
     LexType := aLexType;
     Value := StrNew (aValue);
End;

Destructor LexemaObj.Done;

Begin
     StrDispose (Value);
End;

{ LexBreakerObj }

Constructor LexBreakerObj.Init (InpText : PChar);

Begin
     Inherited Init;
     InputText := StrNew (InpText);
     Ptr := InputText;
End;

Destructor LexBreakerObj.Done;

Begin
     StrDispose (InputText);
     Inherited Done;
End;

Function LexBreakerObj.GetLex : LexemaObjPtr;

Const EndOfText = #0;
      Literal = '''';
      Point = '.';
      FuncSymbol = '(';
      Spaces = [#9,#10,#13,' '];
      Digits = ['0'..'9'];
      Symbols = ['!','"','#','$','%','&','(',')','*','+',
                 ',','-','.','/',':',';','<','=','>','?',
                 '@','[','\',']','^','`','{','|','}','~'];
      Letters = [#0..#$FF]-Spaces-Digits-Symbols-[EndOfText,Literal];
      AnySymbols = Spaces+Digits+Letters+Symbols;

Var P,P1,P2,PP : PChar;
    C,C1 : Char;
    LexT,OldLexT : LexemaType;
    I : LongInt;
    F : FunctionObjPtr;
    FList : ListObjPtr;
    Buff : Array[0..255] of Char;

Begin
     While Ptr^ in Spaces do Inc (Ptr);
     P := Ptr;
     If P^<>EndOfText Then
        If P^ in Letters Then
           Begin
                While P^ in Digits+Letters do Inc (P);
                LexT := lex_Ident;
           End
        Else
           If P^ in Digits Then
              Begin
```

```
                While P^ in Digits do Inc (P);
                If (P^=Point) and ((P+1)^ in Digits) Then
                   Begin
                        Inc (P);
                        While P^ in Digits do Inc (P);
                   End;
                LexT := lex_Number;
             End
          Else
             If P^=Literal Then
                Begin
                     While P^=Literal do
                           Begin
                                Inc (P);
                                While P^ in AnySymbols do Inc (P);
                                If P^=Literal Then Inc (P);
                           End;
                     LexT := lex_String;
                End
             Else
                Begin
                     Inc (P);
                     LexT := lex_Symbol;
                End
     Else LexT := lex_End;

     If (LexT=lex_Ident) or (LexT=lex_Symbol) Then
        Begin
             OldLexT := LexT;
             FList := GetFunctionsList;
             For I := 1 to FList^.GetCount do
                 Begin
                      F := FList^.Get (I);
                      If (F<>nil) and (F^.Assoc<>Assoc_functional) and
                         (StrLIComp(Ptr,F^.Name,StrLen(F^.Name))=0) Then
                         Begin
                              P := Ptr+StrLen(F^.Name);
                              Buff[0] := Char(F^.Priority);
                              Buff[1] := Char(F^.Assoc);
                              StrCopy (@Buff[2],F^.Name);
                              Ptr := Buff;
                              LexT := lex_Operator;
                              Break;
                         End;
                 End;
             If (LexT<>lex_Operator) and (OldLexT=lex_Ident) and
                (P^=FuncSymbol) Then LexT := lex_FuncName;
        End;

     C := P^;
     P^ := EndOfText;
     If LexT=lex_String Then
        Begin
             P1 := Ptr+1;
             P2 := StrNew (Ptr); PP := P2;
             If P1^<>EndOfText Then
                While (P1+1)^<>EndOfText do
                      Begin
                           If (P1^<>Literal) or ((P1+1)^=Literal) Then
                              Begin
                                   P2^ := P1^;
                                   Inc (P2);
                                   If (P1^=Literal) and ((P1+1)^=Literal) Then Inc (P1);
                              End;
                           Inc (P1);
                      End;
             C1 := P2^;
             P2^ := EndOfText;
             GetLex := New (LexemaObjPtr,Init(LexT,PP));
             P2^ := C1;
             StrDispose (PP);
        End
     Else GetLex := New (LexemaObjPtr,Init(LexT,Ptr));
     P^ := C;

     Ptr := P;
End;
```

```
{}

Const UnitActive : Byte = 0; { семафор за инициализация на модула }

Procedure InitUnit;

Begin
     Inc (UnitActive);
     If UnitActive>1 Then Exit;

     uL1.InitUnit;
     uL2.InitUnit;
End;

Procedure DoneUnit;

Begin
     Dec (UnitActive);
     If UnitActive>0 Then Exit;

     uL2.DoneUnit;
     uL1.DoneUnit;
End;


End.
```

Изходен текст на модула Syntax

```
Unit Syntax; { Модул синтактичен анализатор и транслатор }

Interface

Uses Strings,uL1,uL2,Lexical; { Използва: модул за работа с Null-низове;
                                модул ниво първо;
                                модул ниво второ;
                                модул лексикален анализатор. }

Type { Базов обект за символите използвани в правилата }
     RuleSymbolObjPtr = ^RuleSymbolObj;
     RuleSymbolObj = Object(RootObj)
        Function  CompareWith (Symb : RuleSymbolObjPtr) : Boolean; virtual;
     End;
     { Обект терминален символ }
     RuleTermSymbolObjPtr = ^RuleTermSymbolObj;
     RuleTermSymbolObj = Object(RuleSymbolObj)
        Term : PChar;                      { стойност на символа }
        Constructor Init (aTerm : PChar);  { Конструира терминален символ }
        Destructor  Done; virtual;
        { Сравнява терминалният символ с друг произволен символ }
        Function  CompareWith (Symb : RuleSymbolObjPtr) : Boolean; virtual;
        Function  Copy : Pointer; virtual;
     End;
     { Обект нетерминален символ }
     RuleNoTermSymbolObjPtr = ^RuleNoTermSymbolObj;
     RuleNoTermSymbolObj = Object(RuleSymbolObj)
        NoTerm : Word;                     { номер на символа }
        Constructor Init (aNoTerm : Word); { Конструира нетерминален символ }
        { Сравнява нетерминалният символ с друг произволен символ }
        Function  CompareWith (Symb : RuleSymbolObjPtr) : Boolean; virtual;
        Function  Copy : Pointer; virtual;
     End;

     { Интерфейс на генериращите процедури (Seq - поредица за редуциране, от позиция Index назад) }
     CompileProcType = Procedure (Seq : ListObjPtr; Index : LongInt);
     { Обект правило от граматика }
     RuleObjPtr = ^RuleObj;
     RuleObj = Object(RootObj)
        Head : RuleSymbolObjPtr;     { глава на превилото - нетерминален символ }
        Body : ListObjPtr;           { тяло - списък от терминални и нетерминални символи }
```

```
        Compile : CompileProcType;   { генерираща процедура }
        { Конструира правило по зададени: глава, тяло и генерираща процедура }
        Constructor Init (aHead : RuleSymbolObjPtr; aBody : ListObjPtr;
                          aCompile : CompileProcType);
        Destructor  Done; virtual;
        Function  Copy : Pointer; virtual;
     End;

     { Обект синтактичен анализатор и транслатор до израз }
     SyntaxAnaliserObjPtr = ^SyntaxAnaliserObj;
     SyntaxAnaliserObj = Object(RootObj)
        RulesList : ListObjPtr;                { списък от граматични правила }
        RulesHash,
        RulesHash1 : Array[Byte] of ListObjPtr; { хеш таблици за ускоряване на анализа }
        Constructor Init;
        Destructor  Done; virtual;
        { Анализира и транслира входен текст. Резултат списък от изрази. }
        Function  Compile (InputText : PChar) : ListObjPtr;
        { Построява синтактичното дърво на списък от лексеми. }
        Function  Test (Var Seq : ListObjPtr) : ListObjPtr;
        { Транслира списък от лексеми до един израз. }
        Function  CompileOne (Seq : ListObjPtr; ReduceRules : ListObjPtr) : ExpressionObjPtr;
     End;

Procedure InitUnit;  { Първоначална инициализация на модула }
Procedure DoneUnit;  { Край на работата с модула }

Implementation

Type { Обект елемент от магазина (стека) на магазинният анализатор }
     SaveSeqObjPtr = ^SaveSeqObj;
     SaveSeqObj = Object(RootObj)
        Seq : ListObjPtr;
        Index : LongInt;
        FromList : ListObjPtr;
        FromRu : LongInt;
        Constructor Init (aSeq : ListObjPtr; aIndex : LongInt;
                          aFromList : ListObjPtr; aFromRu : LongInt);
        Destructor  Done; virtual;
        Function  Copy : Pointer; virtual;
     End;

{ RuleSymbolObj }

Function RuleSymbolObj.CompareWith (Symb : RuleSymbolObjPtr) : Boolean;

Begin
     CompareWith := False;
End;

{ RuleTermSymbolObj }

Constructor RuleTermSymbolObj.Init (aTerm : PChar);

Begin
     Inherited Init;
     Term := StrNew (aTerm);
End;

Destructor RuleTermSymbolObj.Done;

Begin
     StrDispose (Term);
     Inherited Done;
End;

Function RuleTermSymbolObj.CompareWith (Symb : RuleSymbolObjPtr) : Boolean;

Begin
     CompareWith := (Symb<>nil) and (TypeOf(Self)=TypeOf(Symb^)) and
                    (StrIComp(Term,RuleTermSymbolObjPtr(Symb)^.Term)=0);
End;

Function RuleTermSymbolObj.Copy : Pointer;

Begin
     Copy := New (RuleTermSymbolObjPtr,Init(Term));
```

```
End;

{ RuleNoTermSymbolObj }

Constructor RuleNoTermSymbolObj.Init (aNoTerm : Word);

Begin
     Inherited Init;
     NoTerm := aNoTerm;
End;

Function RuleNoTermSymbolObj.CompareWith (Symb : RuleSymbolObjPtr) : Boolean;

Var NoTerm1 : Word;

Begin
     NoTerm1 := RuleNoTermSymbolObjPtr(Symb)^.NoTerm;
     CompareWith := (Symb<>nil) and (TypeOf(Self)=TypeOf(Symb^)) and
                    ((NoTerm=NoTerm1) or
                     ((NoTerm>=2000) and (NoTerm1>=NoTerm)));
End;

Function RuleNoTermSymbolObj.Copy : Pointer;

Begin
     Copy := New (RuleNoTermSymbolObjPtr,Init(NoTerm));
End;

{ RuleObj }

Constructor RuleObj.Init (aHead : RuleSymbolObjPtr; aBody : ListObjPtr;
                          aCompile : CompileProcType);

Begin
     Head := aHead;
     Body := aBody;
     Compile := aCompile;
End;

Destructor RuleObj.Done;

Begin
     Head^.Free;
     Body^.Free;
     Inherited Done;
End;

Function RuleObj.Copy : Pointer;

Begin
     Copy := New (RuleObjPtr,Init(Head^.Copy,Body^.Copy,Compile));
End;

{ SaveSeqObj }

Constructor SaveSeqObj.Init (aSeq : ListObjPtr; aIndex : LongInt;
                             aFromList : ListObjPtr; aFromRu : LongInt);

Begin
     Inherited Init;
     Seq := aSeq;
     Index := aIndex;
     FromList := aFromList;
     FromRu := aFromRu;
End;

Function SaveSeqObj.Copy : Pointer;

Begin
     Copy := New (SaveSeqObjPtr,Init(Seq^.Copy,Index,FromList,FromRu));
End;

Destructor SaveSeqObj.Done;

Begin
     If Seq<>nil Then Seq^.Free;
     Inherited Done;
```

```
End;

{ SyntaxAnaliserObj }

{ make_xxxx - генериращи процедури за граматиката }
Procedure make_VarExpr (Seq : ListObjPtr; Index : LongInt);
Begin
     Seq^.Replace (Index,New(ExprVarObjPtr,Init(LexemaObjPtr(Seq^.Get(Index))^.Value)));
End;

Procedure make_RealExpr (Seq : ListObjPtr; Index : LongInt);
Var R : Real;
    I : Integer;
Begin
     Val (LexemaObjPtr(Seq^.Get(Index))^.Value,R,I);
     Seq^.Replace (Index,New(ExprRealObjPtr,Init(R)));
End;

Procedure make_ListExpr (Seq : ListObjPtr; Index : LongInt);
Var L : ListObjPtr;
Begin
     L := Seq^.Remove (Index);
     Seq^.Insert (Index,New(ExprListObjPtr,Init(L)));
End;

Procedure make_BinarFuncExpr (Seq : ListObjPtr; Index : LongInt);
Var L : ListObjPtr;
Begin
     New (L,Init);
     L^.Add (Seq^.Remove(Index-2));
     L^.Add (Seq^.Remove(Index-1));
     Seq^.Replace (Index-2,New(ExprFuncObjPtr,Init(LexemaObjPtr(Seq^.Get(Index-2))^.Value,L)));
End;

Procedure make_LeftUnarFuncExpr (Seq : ListObjPtr; Index : LongInt);
Var L : ListObjPtr;
Begin
     New (L,Init);
     L^.Add (Seq^.Remove(Index));
     Seq^.Replace (Index-1,New(ExprFuncObjPtr,Init(LexemaObjPtr(Seq^.Get(Index-1))^.Value,L)));
End;

Procedure make_RightUnarFuncExpr (Seq : ListObjPtr; Index : LongInt);
Var L : ListObjPtr;
Begin
     New (L,Init);
     L^.Add (Seq^.Remove(Index-1));
     Seq^.Replace (Index-1,New(ExprFuncObjPtr,Init(LexemaObjPtr(Seq^.Get(Index-1))^.Value,L)));
End;

Procedure make_FuncExpr (Seq : ListObjPtr; Index : LongInt);
Var L : ListObjPtr;
Begin
     L := Seq^.Remove (Index-1);
     Seq^.Del (Index-1);
     Seq^.Del (Index-2);
     Seq^.Replace (Index-3,New(ExprFuncObjPtr,Init(LexemaObjPtr(Seq^.Get(Index-3))^.Value,L)));
End;

Procedure make_FuncExpr1 (Seq : ListObjPtr; Index : LongInt);
Var L : ListObjPtr;
Begin
     New (L,Init);
     L^.Add (Seq^.Remove(Index-1));
     Seq^.Del (Index-1);
     Seq^.Del (Index-2);
     Seq^.Replace (Index-3,New(ExprFuncObjPtr,Init(LexemaObjPtr(Seq^.Get(Index-3))^.Value,L)));
End;

Procedure make_FuncExpr0 (Seq : ListObjPtr; Index : LongInt);
Var L : ListObjPtr;
Begin
     New (L,Init);
     Seq^.Del (Index);
     Seq^.Del (Index-1);
     Seq^.Replace (Index-2,New(ExprFuncObjPtr,Init(LexemaObjPtr(Seq^.Get(Index-2))^.Value,L)));
End;
```

```
Procedure make_Open (Seq : ListObjPtr; Index : LongInt);
Begin
     Seq^.Del (Index);
     Seq^.Del (Index-2);
End;

Procedure make_ParamList (Seq : ListObjPtr; Index : LongInt);
Var L : ListObjPtr;
Begin
     L := Seq^.Get (Index-2);
     Seq^.Del (Index-1);
     L^.Add (Seq^.Remove(Index-1));
End;

Procedure make_ParamList2 (Seq : ListObjPtr; Index : LongInt);
Var L : ListObjPtr;
Begin
     New (L,Init);
     L^.Add (Seq^.Remove(Index-2));
     L^.Add (Seq^.Remove(Index-1));
     Seq^.Replace (Index-2,L);
End;

Procedure make_List (Seq : ListObjPtr; Index : LongInt);
Begin
     Seq^.Del (Index);
     Seq^.Del (Index-2);
End;

Procedure make_List1 (Seq : ListObjPtr; Index : LongInt);
Var L : ListObjPtr;
Begin
     New (L,Init);
     L^.Add (Seq^.Remove(Index-1));
     Seq^.Del (Index-1);
     Seq^.Replace (Index-2,L);
End;

Procedure make_EmptyList (Seq : ListObjPtr; Index : LongInt);
Var L : ListObjPtr;
Begin
     New (L,Init);
     Seq^.Del (Index);
     Seq^.Replace (Index-1,L);
End;

Constructor SyntaxAnaliserObj.Init;

  Procedure AddRule (Head : Word; Body : ListObjPtr; Comp : CompileProcType);
  Begin
       RulesList^.Add (New(RuleObjPtr,Init(New(RuleNoTermSymbolObjPtr,Init(Head)),Body,Comp)));
  End;

Var TermSym : RuleTermSymbolObjPtr;
    NoTermSym : RuleNoTermSymbolObjPtr;
    Sym : RuleSymbolObjPtr;
    Body : ListObjPtr;
    I : LongInt;
    J,W : Word;
    FL,L : ListObjPtr;
    F : FunctionObjPtr;
    Ru : RuleObjPtr;

Begin
     Inherited Init;
     New (RulesList,Init);

{ Записва правилата на граматиката в списъка с правила. }
{
<2009,exp9>::=<1,ident>
<2009,exp9>::=<3,number>
<2009,exp9>::=<11,list>
}
     New (Body,Init);
     New (NoTermSym,Init(1)); Body^.Add (NoTermSym);
     AddRule (2009,Body,make_VarExpr);
```

```
     New (Body,Init);
     New (NoTermSym,Init(3)); Body^.Add (NoTermSym);
     AddRule (2009,Body,make_RealExpr);
     New (Body,Init);
     New (NoTermSym,Init(11)); Body^.Add (NoTermSym);
     AddRule (2009,Body,make_ListExpr);

{
<2001,exp1>::=<2002,exp2><1015,o1xx><2002,exp2>
<2001,exp1>::=<2002,exp2><1016,o1xy><2001,exp1>
<2001,exp1>::=<2001,exp1><1017,o1yx><2002,exp2>
<2001,exp1>::=<1011,l1x><2002,exp2>
<2001,exp1>::=<1012,l1y><2001,exp1>
<2001,exp1>::=<2002,exp2><1013,r1x>
<2001,exp1>::=<2001,exp1><1014,r1y>
}
     For I := 2000 to 2008 do
         Begin
              New (Body,Init);
              New (NoTermSym,Init(I+1)); Body^.Add (NoTermSym);
              New (NoTermSym,Init(1000+(I-2000)*10+5)); Body^.Add (NoTermSym);
              New (NoTermSym,Init(I+1)); Body^.Add (NoTermSym);
              AddRule (I,Body,make_BinarFuncExpr);

              New (Body,Init);
              New (NoTermSym,Init(I)); Body^.Add (NoTermSym);
              New (NoTermSym,Init(1000+(I-2000)*10+6)); Body^.Add (NoTermSym);
              New (NoTermSym,Init(I+1)); Body^.Add (NoTermSym);
              AddRule (I,Body,make_BinarFuncExpr);

              New (Body,Init);
              New (NoTermSym,Init(I+1)); Body^.Add (NoTermSym);
              New (NoTermSym,Init(1000+(I-2000)*10+7)); Body^.Add (NoTermSym);
              New (NoTermSym,Init(I)); Body^.Add (NoTermSym);
              AddRule (I,Body,make_BinarFuncExpr);

              New (Body,Init);
              New (NoTermSym,Init(I+1)); Body^.Add (NoTermSym);
              New (NoTermSym,Init(1000+(I-2000)*10+1)); Body^.Add (NoTermSym);
              AddRule (I,Body,make_LeftUnarFuncExpr);

              New (Body,Init);
              New (NoTermSym,Init(I)); Body^.Add (NoTermSym);
              New (NoTermSym,Init(1000+(I-2000)*10+2)); Body^.Add (NoTermSym);
              AddRule (I,Body,make_LeftUnarFuncExpr);

              New (Body,Init);
              New (NoTermSym,Init(1000+(I-2000)*10+3)); Body^.Add (NoTermSym);
              New (NoTermSym,Init(I+1)); Body^.Add (NoTermSym);
              AddRule (I,Body,make_RightUnarFuncExpr);

              New (Body,Init);
              New (NoTermSym,Init(1000+(I-2000)*10+4)); Body^.Add (NoTermSym);
              New (NoTermSym,Init(I)); Body^.Add (NoTermSym);
              AddRule (I,Body,make_RightUnarFuncExpr);
         End;

{<2009,exp9>::=<5,funcname>'('<10,paramlist>')'
<2009,exp9>::=<5,funcname>'('<2000,exp0>')'
<2009,exp9>::=<5,funcname>'('')'
<2009,exp9>::='('<2000,exp0>')'}
     New (Body,Init);
     New (NoTermSym,Init(254)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(10)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(252)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(5)); Body^.Add (NoTermSym);
     AddRule (2009,Body,make_FuncExpr);
     New (Body,Init);
     New (NoTermSym,Init(254)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(2000)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(252)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(5)); Body^.Add (NoTermSym);
     AddRule (2009,Body,make_FuncExpr1);
     New (Body,Init);
     New (NoTermSym,Init(254)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(252)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(5)); Body^.Add (NoTermSym);
```

```
     AddRule (2009,Body,make_FuncExpr0);
     New (Body,Init);
     New (NoTermSym,Init(254)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(2000)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(252)); Body^.Add (NoTermSym);
     AddRule (2009,Body,make_Open);

{<10,paramlist>::=<10,paramlist>','<2000,exp0>}
     New (Body,Init);
     New (NoTermSym,Init(2000)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(251)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(10)); Body^.Add (NoTermSym);
     AddRule (10,Body,make_ParamList);
{<10,paramlist>::=<2000,exp0>','<2000,exp0>}
     New (Body,Init);
     New (NoTermSym,Init(2000)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(251)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(2000)); Body^.Add (NoTermSym);
     AddRule (10,Body,make_ParamList2);
{<11,list>::='['<10,paramlist>']'}
     New (Body,Init);
     New (NoTermSym,Init(255)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(10)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(253)); Body^.Add (NoTermSym);
     AddRule (11,Body,make_List);
{<11,list>::='['<2000,exp0>']'}
     New (Body,Init);
     New (NoTermSym,Init(255)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(2000)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(253)); Body^.Add (NoTermSym);
     AddRule (11,Body,make_List1);
{<11,list>::='['']'}
     New (Body,Init);
     New (NoTermSym,Init(255)); Body^.Add (NoTermSym);
     New (NoTermSym,Init(253)); Body^.Add (NoTermSym);
     AddRule (11,Body,make_EmptyList);

{ Хешира правилата }
     For I := 0 to 255 do
         Begin
              New (RulesHash[I],Init);
              New (RulesHash1[I],Init);
         End;
     For I := 1 to RulesList^.GetCount do
         Begin
              Ru := RulesList^.Get (I);
              W := RuleNoTermSymbolObjPtr(Ru^.Body^.Get(1))^.NoTerm;
              RulesHash[Byte(W)]^.Add (Ru);
              If (W>2000) and (W<=2009) Then
                 For J := W+1 to 2009 do RulesHash[Byte(J)]^.Add (Ru);
         End;
     For I := 1 to RulesList^.GetCount do
         Begin
              Ru := RulesList^.Get (I);
              NoTermSym := Ru^.Body^.Get(2);
              If (NoTermSym<>nil) and (NoTermSym^.NoTerm<2000) Then
                 Begin
                      RulesHash1[Byte(NoTermSym^.NoTerm)]^.Add (Ru);
                 End;
         End;
     For I := 1 to RulesList^.GetCount do
         Begin
              Ru := RulesList^.Get (I);
              NoTermSym := Ru^.Body^.Get(2);
              If NoTermSym=nil Then
                 For J := 0 to 255 do RulesHash1[J]^.Add (Ru);
         End;
End;

Destructor SyntaxAnaliserObj.Done;

Var I : Byte;

Begin
     For I := 0 to 255 do
         Begin
              While RulesHash[I]^.GetCount>0 do RulesHash[I]^.Remove (1);
```

```
              RulesHash[I]^.Free;
              While RulesHash1[I]^.GetCount>0 do RulesHash1[I]^.Remove (1);
              RulesHash1[I]^.Free;
          End;
     RulesList^.Free;
     Inherited Done;
End;

Function SyntaxAnaliserObj.Compile (InputText : PChar) : ListObjPtr;

Var LB : LexBreakerObjPtr;
    Lex : LexemaObjPtr;
    Seq,Seq1,ReduceList,Results : ListObjPtr;
    TermSym : RuleTermSymbolObjPtr;
    NoTermSym : RuleNoTermSymbolObjPtr;
    S : PChar;

Begin
     New (Seq,Init);
     New (Seq1,Init);
     New (Results,Init);

     New (LB,Init(InputText));
     Repeat
           Lex := LB^.GetLex;
           Case Lex^.LexType of
                lex_End : Begin
                               Lex^.Free;
                               If Seq^.GetCount>0 Then
                                  Begin
                                       ReduceList := Test(Seq);
                                       If ReduceList<>nil Then
                                          Begin
                                               Results^.Add(CompileOne(Seq1,ReduceList));
                                               ReduceList^.Free;
                                          End
                                       Else Results^.Add(nil);
                                  End;
                               Compile := Results;
                               Break;
                          End;
                lex_Ident : Begin
                                 New (NoTermSym,Init(LongInt(Lex^.LexType)));
                                 Seq^.Add (NoTermSym);
                            End;
                lex_String,
                lex_Symbol : Begin
                                  Case Lex^.Value^ of
                                       ';' : Begin
                                                  Lex^.Free;
                                                  ReduceList := Test(Seq);
                                                  If ReduceList<>nil Then
                                                     Begin
                                                          Results^.Add(CompileOne(Seq1,ReduceList));
                                                          ReduceList^.Free;
                                                     End
                                                  Else Results^.Add(nil);
                                                  Seq^.Free;
                                                  New (Seq,Init);
                                                  Seq1^.Free;
                                                  New (Seq1,Init);
                                                  Continue;
                                             End;
                                       ',','(','[',')',']' :
                                             Begin
                                                  New (NoTermSym,Init(250+Pos(Lex^.Value^,',([)]')));
                                                  Seq^.Add (NoTermSym);
                                             End;
                                        Else Begin
                                                  New (TermSym,Init(Lex^.Value));
                                                  Seq^.Add (TermSym);
                                             End;
                                  End
                             End;
                lex_Number,
                lex_FuncName : Begin
                                    New (NoTermSym,Init(LongInt(Lex^.LexType)));
```

```
                                   Seq^.Add (NoTermSym);
                              End;
                lex_Operator : Begin
                                   New (NoTermSym,
                                        Init(1000+10*Byte(Lex^.Value^)+Byte((Lex^.Value+1)^)));
                                   Seq^.Add (NoTermSym);
                              End;
            End;
            If Lex^.LexType=lex_Operator Then
               Begin
                    S := StrNew (Lex^.Value+2);
                    StrDispose (Lex^.Value);
                    Lex^.Value := S;
               End;
            Seq1^.Add (Lex);
      Until False;
      LB^.Free;

      Seq1^.Free;
      Seq^.Free;
End;

Function SyntaxAnaliserObj.Test (Var Seq : ListObjPtr) : ListObjPtr;

Var Lex : LexemaObjPtr;
    Seq1,Stack,RList : ListObjPtr;
    Index,I,J,RulesCount : LongInt;
    TermSym : RuleTermSymbolObjPtr;
    NoTermSym,NoTermSym1 : RuleNoTermSymbolObjPtr;
    Sym : RuleSymbolObjPtr;
    Ru : RuleObjPtr;
    Flag : Boolean;
    Save : SaveSeqObjPtr;
    W1,W2 : Word;

  { Проверява дали може да бъде приложено дадено правило. }
  Function Try (Seq : ListObjPtr; Ind : LongInt; Ru : RuleObjPtr) : Boolean;
  Var I,I1 : LongInt;
  Begin
       Try := False;
       I1 := Ru^.Body^.GetCount;
       If Ind>=I1 Then
          Begin
               For I := Ind-I1+1 to Ind do
                   Begin
                        If not RuleSymbolObjPtr(Ru^.Body^.Get(I1))^.CompareWith(Seq^.Get(I)) Then Exit;
                        Dec (I1);
                   End;
               Try := True;
               If (RuleNoTermSymbolObjPtr(Ru^.Head)^.NoTerm>=2000) and
                  (RuleNoTermSymbolObjPtr(Ru^.Head)^.NoTerm<2009) Then
                  Begin
                       NoTermSym := Seq^.Get(Ind-1);
                       NoTermSym1 := Seq^.Get(Ind+1);
                       If (NoTermSym1<>nil) and (TypeOf(NoTermSym1^)=TypeOf(RuleNoTermSymbolObj)) and
                          (NoTermSym^.NoTerm>=1000) and (NoTermSym^.NoTerm<2000) and
                          (NoTermSym1^.NoTerm>=1000) and (NoTermSym1^.NoTerm<2000) Then
                          Begin
                               W1 := ((NoTermSym^.NoTerm-1000) div 10);
                               W2 := ((NoTermSym1^.NoTerm-1000) div 10);
                               Case AssocType(NoTermSym^.NoTerm mod 10) of
                                    Assoc_yfx : Try := W1>=W2;
                                    Assoc_xfx : Try := W1>W2;
                                    Assoc_xfy : Try := W1>W2;
                                    Assoc_fx : Try := W1>=W2;
                                    Assoc_fy : Try := W1>W2;
                               End;
                          End;
                  End;
          End;
  End;

Begin
     New (Stack,Init);
     Index := 1;

     Sym := Seq^.Get (Index);
```

```
If Sym=nil Then
   Begin
        Test := Stack;
        Exit;
   End;
If TypeOf(Sym^)=TypeOf(RuleNoTermSymbolObj)
   Then RList := RulesHash[Byte(RuleNoTermSymbolObjPtr(Sym)^.NoTerm)]
   Else RList := RulesHash[240];
J := 1;
RulesCount := RList^.GetCount;
Repeat
      While J<=RulesCount do
            Begin
                 Ru := RList^.Get (J);
                 If Try(Seq,Index,Ru) Then
                    Begin
                         {push (Seq,Index,J)}
                         Stack^.Insert (1,New(SaveSeqObjPtr,Init(Seq,Index,RList,J)));
                         {reduce (Seq,Index,J)}
                         New (Seq1,Init);
                         For I := 1 to Index-Ru^.Body^.GetCount do
                             Seq1^.Add (RuleSymbolObjPtr(Seq^.Get(I))^.Copy);
                         Seq1^.Add (Ru^.Head^.Copy);
                         For I := Index+1 to Seq^.GetCount do
                             Seq1^.Add (RuleSymbolObjPtr(Seq^.Get(I))^.Copy);
                         Seq := Seq1; Dec (Index,Ru^.Body^.GetCount-1);

                         NoTermSym := Seq^.Get (Index-1);
                         If (NoTermSym<>nil) and
                            (TypeOf(NoTermSym^)=TypeOf(RuleNoTermSymbolObj)) and
                            (NoTermSym^.NoTerm<2000) Then
                            Begin
                                 RList := RulesHash1[Byte(NoTermSym^.NoTerm)];
                            End
                         Else
                            Begin
                                 Sym := Seq^.Get (Index);
                                 If TypeOf(Sym^)=TypeOf(RuleNoTermSymbolObj) Then
                                    RList := RulesHash[Byte(RuleNoTermSymbolObjPtr(Sym)^.NoTerm)]
                                 Else RList := RulesHash[240];
                            End;
                         J := 1;
                         RulesCount := RList^.GetCount;
                    End
                 Else Inc (J);
            End;

      If Index<Seq^.GetCount Then Inc (Index)
      Else
         Begin
              NoTermSym := Seq^.Get(1);
              If (Seq^.GetCount=1) and (TypeOf(NoTermSym^)=TypeOf(RuleNoTermSymbolObj)) and
                 (NoTermSym^.NoTerm>=2000) and (NoTermSym^.NoTerm<=2009) Then
                 Begin
                      Flag := True;
                      Break;
                 End
              Else
                 Begin
                      If Stack^.GetCount>0 Then
                         Begin
                              {pop(Seq,Index,J)}
                              Seq^.Free;
                              Save := Stack^.Remove (1);
                              Seq := Save^.Seq;
                              Save^.Seq := nil;
                              Index := Save^.Index;
                              RList := Save^.FromList;
                              RulesCount := RList^.GetCount;
                              J := Save^.FromRu+1;
                              Save^.Free;
                              Continue;
                         End
                      Else
                         Begin
                              Flag := False;
                              Break;
```

```
                              End;
                         End;
                   End;

              NoTermSym := Seq^.Get (Index-1);
              If (NoTermSym<>nil) and (TypeOf(NoTermSym^)=TypeOf(RuleNoTermSymbolObj)) and
                 (NoTermSym^.NoTerm<2000) Then
                 Begin
                      RList := RulesHash1[Byte(NoTermSym^.NoTerm)];
                 End
              Else
                 Begin
                      Sym := Seq^.Get (Index);
                      If TypeOf(Sym^)=TypeOf(RuleNoTermSymbolObj)
                         Then RList := RulesHash[Byte(RuleNoTermSymbolObjPtr(Sym)^.NoTerm)]
                         Else RList := RulesHash[240];
                 End;
              J := 1;
              RulesCount := RList^.GetCount;

     Until False;

     If Flag Then
        Begin
             New (RList,Init);
             While Stack^.GetCount>0 do RList^.Insert (1,Stack^.Remove(1));
             Test := RList;
        End
     Else Test := nil;
     Stack^.Free;
End;

Function SyntaxAnaliserObj.CompileOne (Seq : ListObjPtr; ReduceRules : ListObjPtr) : ExpressionObjPtr;

Var Ru : RuleObjPtr;
    Save : SaveSeqObjPtr;
    I : LongInt;
    E : ExpressionObjPtr;

Begin
     For I := 1 to ReduceRules^.GetCount do
         Begin
              Save := ReduceRules^.Get (I);
              Ru := Save^.FromList^.Get (Save^.FromRu);
              Ru^.Compile (Seq,Save^.Index);
         End;
     E := Seq^.Get(1);
     If E<>nil Then CompileOne := E^.Copy
               Else CompileOne := nil;
End;

{}

Const UnitActive : Byte = 0; { семафор за инициализация на модула }

Procedure InitUnit;

Begin
     Inc (UnitActive);
     If UnitActive>1 Then Exit;

     uL1.InitUnit;
     uL2.InitUnit;
     Lexical.InitUnit;
End;

Procedure DoneUnit;

Begin
     Dec (UnitActive);
     If UnitActive>0 Then Exit;

     Lexical.DoneUnit;
     uL2.DoneUnit;
     uL1.DoneUnit;
End;
```

```
End.
```

Изходен текст на модула uL3

```
Unit uL3;  { Модул ниво трето }

Interface

Uses uL1,uL2,Syntax,Strings;  { Използва: модул ниво първо;
                                         модул ниво второ;
                                         модул синтактичен анализатор и транслатор;
                                         модул за работа с Null-низове. }

Type { Функция за визуализация на дадена сцена }
     IShowObjPtr = ^IShowObj;
     IShowObj = Object(InternalFunctionObj)
        Constructor Init;
        Function  Calc (ActualParamList : ParamListObjPtr) : Pointer; virtual;
     End;

{ Константи определящи точността на изследване на телата. }
Const GeomBody_Eps : Real = 0.001;
      GeomBody_Eps2 : Real = 0.0005; { = GeomBody_Eps/2 }
      GeomBody_EpsNorm : Real = 0.00000001;
      GeomBody_WaitComPoint : Real = 5;

Procedure InitUnit;  { Първоначална инициализация на модула }
Procedure DoneUnit;  { Край на работата с модула }

Implementation

{ Използва модули за генериране на реалистични изображения. }
Uses uGlobal1,uPointLt,uRazLtC,MainUnit,uBeams,uGBeam,uList,uGVec,uEng,
     uAir;

Function CalcProp (L : ListObjPtr; PropName : PChar; Params : ListObjPtr;
                   ActualParamList : ParamListObjPtr) : RootObjPtr;

Var SEList : ListObjPtr;
    EF : ExprFuncObjPtr;

Begin
     New (SEList,Init);
     SEList^.Add (New(ExprListObjPtr,Init(L^.Copy)));
     SEList^.Add (New(ExprFuncObjPtr,Init(PropName,Params)));
     New (EF,Init('.',SEList));
     CalcProp := EF^.Calc (ActualParamList);
     EF^.Free;
End;

Type PGeomBody = ^TGeomBody;
     TGeomBody = Object(TBody)
        Props : ListObjPtr;
        ActualParamList : ParamListObjPtr;
        Constructor Init (aProps : ListObjPtr; aActualParamList : ParamListObjPtr; Var RM : TRM);
        Destructor  Done (Var RM : TRM);
        Procedure SetBeamPos (Var OB : TOBeam; Var RM : TRM); virtual;
        Procedure GiveComPoint (Const B : GBeam;
                                Var OP : TOBeam; Var RM : TRM); virtual;
        Procedure GiveSplitPoint (Const B : TOBeam;
                                  Var SB : TOBeam; Var RM : TRM); virtual;
        Procedure GiveRRAt (Const P : GPoint; Prv : PBody;
                            Var RR : EngValue; Var RM : TRM); virtual;
        Procedure GiveBRAt (Const P : GPoint; Prv : PBody;
                            Var BR : EngValue; Var RM : TRM); virtual;
        Procedure GiveRes (Var Res : EngValue; Var RM : TRM); virtual;
        Procedure GiveNormVecAt (Const P : TABeam;
                                 Var V : GVec; Var RM : TRM); virtual;
        Procedure GiveKDAt (Const P : TABeam; Prv : PBody;
                            Var KD : EngValue; Var RM : TRM); virtual;
```

```
        Procedure GiveRazRAt (Const P : TABeam;
                              Var RazR : EngValue; Var RM : TRM); virtual;
        Procedure GiveTnAt (Const P : TABeam;
                            Var Tn : Integer; Var RM : TRM); virtual;
        Procedure GiveLtR (Var LtR : EngValue; Var RM : TRM ); virtual;
      private
        Procedure GlobalToLoacalSystem (Glob : TVCoord; Var Loc : TVCoord);
        Procedure LoacalToGlobalSystem (Loc : TVCoord; Var Glob : TVCoord);
        Procedure GlobalToLoacalSystemV (Glob : TVCoord; Var Loc : TVCoord);
        Procedure LoacalToGlobalSystemV (Loc : TVCoord; Var Glob : TVCoord);
     End;

Constructor TGeomBody.Init (aProps : ListObjPtr; aActualParamList : ParamListObjPtr; Var RM : TRM);

Begin
     RM := OK;
     Name := 'Geometric';
     Props := aProps;
     ActualParamList := aActualParamList;
End;

Destructor TGeomBody.Done (Var RM : TRM);

Begin
     RM := OK;
End;

Const Coord0 : TVCoord = (0,0,0);

Procedure TGeomBody.GlobalToLoacalSystem (Glob : TVCoord; Var Loc : TVCoord);

Var L,L1 : ListObjPtr;
    R : ExprRealObjPtr;

Begin
     New (L,Init);
     L^.Add (New(ExprRealObjPtr,Init(Glob[1])));
     L^.Add (New(ExprRealObjPtr,Init(Glob[2])));
     L^.Add (New(ExprRealObjPtr,Init(Glob[3])));
     New (L1,Init);
     L1^.Add (New(ExprListObjPtr,Init(L)));
     L := Pointer(CalcProp(Props,'LocalSystemBack',L1,ActualParamList));
     If (L<>nil) and (TypeOf(L^)=TypeOf(ListObj)) Then
        Begin
             R := L^.Get (1);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then Loc[1] := R^.Value;
             R := L^.Get (2);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then Loc[2] := R^.Value;
             R := L^.Get (3);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then Loc[3] := R^.Value;
        End
     Else Loc := Glob;
     If L<>nil Then L^.Free;
End;

Procedure TGeomBody.LoacalToGlobalSystem (Loc : TVCoord; Var Glob : TVCoord);

Var L,L1 : ListObjPtr;
    R : ExprRealObjPtr;

Begin
     New (L,Init);
     L^.Add (New(ExprRealObjPtr,Init(Loc[1])));
     L^.Add (New(ExprRealObjPtr,Init(Loc[2])));
     L^.Add (New(ExprRealObjPtr,Init(Loc[3])));
     New (L1,Init);
     L1^.Add (New(ExprListObjPtr,Init(L)));
     L := Pointer(CalcProp(Props,'LocalSystem',L1,ActualParamList));
     If (L<>nil) and (TypeOf(L^)=TypeOf(ListObj)) Then
        Begin
             R := L^.Get (1);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then Glob[1] := R^.Value;
             R := L^.Get (2);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then Glob[2] := R^.Value;
             R := L^.Get (3);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then Glob[3] := R^.Value;
        End
```

```
     Else Glob := Loc;
     If L<>nil Then L^.Free;
End;

Procedure TGeomBody.GlobalToLoacalSystemV (Glob : TVCoord; Var Loc : TVCoord);

Var Loc0 : TVCoord;

Begin
     GlobalToLoacalSystem (Coord0,Loc0);
     GlobalToLoacalSystem (Glob,Loc);
     Loc[1] := Loc[1]-Loc0[1];
     Loc[2] := Loc[2]-Loc0[2];
     Loc[3] := Loc[3]-Loc0[3];
End;

Procedure TGeomBody.LoacalToGlobalSystemV (Loc : TVCoord; Var Glob : TVCoord);

Var Glob0 : TVCoord;

Begin
     LoacalToGlobalSystem (Coord0,Glob0);
     LoacalToGlobalSystem (Loc,Glob);
     Glob[1] := Glob[1]-Glob0[1];
     Glob[2] := Glob[2]-Glob0[2];
     Glob[3] := Glob[3]-Glob0[3];
End;

Procedure TGeomBody.SetBeamPos (Var OB : TOBeam; Var RM : TRM);

Var RM1 : TRM;
    OB1 : TOBeam;
    P : GPoint;
    R : ExprRealObjPtr;
    L,L1 : ListObjPtr;
    R1,Vec1,Vec2,Vec3 : Real;
    Coord,Coord1: TVCoord;

Begin
     RM := OK;
     OB.AB.BodyID := @Self;
     OB.AB.Place := unk;

     GlobalToLoacalSystem (OB.Coord,Coord);

     New (L,Init);
     L^.Add (New(ExprRealObjPtr,Init(Coord[1])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[2])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[3])));
     New (L1,Init);
     L1^.Add (New(ExprListObjPtr,Init(L)));
     R := Pointer(CalcProp(Props,'Form',L1,ActualParamList));
     If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj))
        Then R1 := R^.Value Else R1 := 0;
     If R<>nil Then R^.Free;
     If R1<-GeomBody_Eps2 Then OB.Pos := InP
     Else If R1>GeomBody_Eps2 Then OB.Pos := OutP
          Else
             Begin
                  GlobalToLoacalSystemV (OB.Vec.Coord,Coord1);
                  Vec1 := Coord1[1];
                  Vec2 := Coord1[2];
                  Vec3 := Coord1[3];
                  R1 := Sqrt(Sqr(Vec1)+Sqr(Vec2)+Sqr(Vec3));
                  New (L,Init);
                  L^.Add (New(ExprRealObjPtr,Init(Coord[1]+Vec1/R1*GeomBody_Eps)));
                  L^.Add (New(ExprRealObjPtr,Init(Coord[2]+Vec2/R1*GeomBody_Eps)));
                  L^.Add (New(ExprRealObjPtr,Init(Coord[3]+Vec3/R1*GeomBody_Eps)));
                  New (L1,Init);
                  L1^.Add (New(ExprListObjPtr,Init(L)));
                  R := Pointer(CalcProp(Props,'Form',L1,ActualParamList));
                  If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj))
                     Then R1 := R^.Value Else R1 := 0;
                  If R<>nil Then R^.Free;
                  If R1<-GeomBody_Eps2 Then OB.Pos := ToIn
                  Else If R1>GeomBody_Eps2 Then OB.Pos := ToOut
                       Else OB.Pos := Slide;
```

```
            End;
End;

Procedure TGeomBody.GiveComPoint (Const B : GBeam;
                                  Var OP : TOBeam; Var RM : TRM);
Var RM1 : TRM;
    R1,R2,RR : Real;
    R : ExprRealObjPtr;
    L,L1 : ListObjPtr;
    B1 : GBeam;
    Coord,Coord1,Coord2 : TVCoord;

Begin
     RM := OK;
     B.CopyTo (B1,RM1);
     B1.Vec.NormLen (RM1);
     OP.Init (RM1);
     OP.Coord := B.Coord;

     GlobalToLoacalSystem (OP.Coord,Coord);
     Coord2 := Coord;
     GlobalToLoacalSystemV (B1.Vec.Coord,Coord1);

     New (L,Init);
     L^.Add (New(ExprRealObjPtr,Init(Coord[1])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[2])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[3])));
     New (L1,Init);
     L1^.Add (New(ExprListObjPtr,Init(L)));
     R := Pointer(CalcProp(Props,'Form',L1,ActualParamList));
     If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj))
        Then R2 := R^.Value Else R2 := 0;
     If R<>nil Then R^.Free;
     R1 := Abs(R2);
     If R1<GeomBody_Eps2 Then R1 := GeomBody_Eps;
     RR := R1;
     Repeat
           Coord[1] := Coord[1] + Coord1[1]*R1;
           Coord[2] := Coord[2] + Coord1[2]*R1;
           Coord[3] := Coord[3] + Coord1[3]*R1;
           New (L,Init);
           L^.Add (New(ExprRealObjPtr,Init(Coord[1])));
           L^.Add (New(ExprRealObjPtr,Init(Coord[2])));
           L^.Add (New(ExprRealObjPtr,Init(Coord[3])));
           New (L1,Init);
           L1^.Add (New(ExprListObjPtr,Init(L)));
           R := Pointer(CalcProp(Props,'Form',L1,ActualParamList));
           If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj))
              Then R1 := Abs(R^.Value) Else R1 := 0;
           If R<>nil Then R^.Free;
           RR := RR + R1;
     Until (R1<GeomBody_Eps2) or (RR>GeomBody_WaitComPoint);

     Coord[1] := Coord2[1]+RR*Coord1[1];
     Coord[2] := Coord2[2]+RR*Coord1[2];
     Coord[3] := Coord2[3]+RR*Coord1[3];
     LoacalToGlobalSystem (Coord,OP.Coord);
     B.Vec.CopyTo (OP.Vec,RM1);
     B1.Done (RM1);

     If R1<GeomBody_Eps2 Then
        SetBeamPos (OP,RM1)
     Else
        Begin
             RM := gcpNE;
             OP.AB.BodyID := @Self;
             OP.AB.Place := unk;
             If R2>GeomBody_Eps2 Then OP.Pos := OutP
                                 Else OP.Pos := InP;
        End;
End;

Procedure TGeomBody.GiveSplitPoint (Const B : TOBeam;
                                    Var SB : TOBeam; Var RM : TRM);
Var RM1 : TRM;
    R1,R2,RR : Real;
    R : ExprRealObjPtr;
```

```
    L,L1 : ListObjPtr;
    Vec1 : GVec;
    Coord,Coord1: TVCoord;

Begin
     RM := OK;
     SB.Init (RM1);
     B.CopyTo (SB,RM1);
     SB.Vec.CopyTo (Vec1,RM1);
     Vec1.NormLen (RM1);

     GlobalToLoacalSystem (SB.Coord,Coord);
     GlobalToLoacalSystemV (Vec1.Coord,Coord1);

     RR := 0;
     Repeat
           Coord[1] := Coord[1] + Coord1[1]*GeomBody_Eps;
           Coord[2] := Coord[2] + Coord1[2]*GeomBody_Eps;
           Coord[3] := Coord[3] + Coord1[3]*GeomBody_Eps;
           RR := RR + GeomBody_Eps;
           New (L,Init);
           L^.Add (New(ExprRealObjPtr,Init(Coord[1])));
           L^.Add (New(ExprRealObjPtr,Init(Coord[2])));
           L^.Add (New(ExprRealObjPtr,Init(Coord[3])));
           New (L1,Init);
           L1^.Add (New(ExprListObjPtr,Init(L)));
           R := Pointer(CalcProp(Props,'Form',L1,ActualParamList));
           If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj))
              Then R2 := R^.Value Else R2 := 0;
           If R<>nil Then R^.Free;
           R1 := Abs(R2);
     Until (R1>=GeomBody_Eps2) or (RR>GeomBody_WaitComPoint);
     If R1>=GeomBody_Eps2 Then
        Begin
             SB.AB.BodyID := @Self;
             SB.AB.Place := unk;
             If R2>0 Then SB.Pos := ToOut
                     Else SB.Pos := ToIn;
        End
     Else
        Begin
             RM := gspNE;
             SB.AB.BodyID := @Self;
             SB.AB.Place := unk;
             SB.Pos := Slide;
        End;

     LoacalToGlobalSystem (Coord,SB.Coord);
End;

Procedure TGeomBody.GiveRRAt (Const P : GPoint; Prv : PBody;
                              Var RR : EngValue; Var RM : TRM);
Var RM1 : TRM;
    EV : TEVData;
    L,L1 : ListObjPtr;
    R : ExprRealObjPtr;
    Coord: TVCoord;

Begin
     RM := OK;
     RR.Init (RM1);
     EV[1] := 0;  EV[2] := 0;  EV[3] := 0;

     GlobalToLoacalSystem (P.Coord,Coord);

     New (L,Init);
     L^.Add (New(ExprRealObjPtr,Init(Coord[1])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[2])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[3])));
     New (L1,Init);
     L1^.Add (New(ExprListObjPtr,Init(L)));
     L := Pointer(CalcProp(Props,'RR',L1,ActualParamList));
     If (L<>nil) and (TypeOf(L^)=TypeOf(ListObj)) Then
        Begin
             R := L^.Get (1);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[1] := R^.Value;
             R := L^.Get (2);
```

```
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[2] := R^.Value;
             R := L^.Get (3);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[3] := R^.Value;
        End;
     If L<>nil Then L^.Free;
     RR.TakeData (EV,RM1);
End;

Procedure TGeomBody.GiveBRAt (Const P : GPoint; Prv : PBody;
                              Var BR : EngValue; Var RM : TRM);
Var RM1 : TRM;
    EV : TEVData;
    L,L1 : ListObjPtr;
    R : ExprRealObjPtr;
    Coord: TVCoord;

Begin
     RM := OK;
     BR.Init (RM1);
     EV[1] := 0;  EV[2] := 0;  EV[3] := 0;

     GlobalToLoacalSystem (P.Coord,Coord);
     New (L,Init);
     L^.Add (New(ExprRealObjPtr,Init(Coord[1])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[2])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[3])));
     New (L1,Init);
     L1^.Add (New(ExprListObjPtr,Init(L)));
     L := Pointer(CalcProp(Props,'BR',L1,ActualParamList));
     If (L<>nil) and (TypeOf(L^)=TypeOf(ListObj)) Then
        Begin
             R := L^.Get (1);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[1] := R^.Value;
             R := L^.Get (2);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[2] := R^.Value;
             R := L^.Get (3);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[3] := R^.Value;
        End;
     If L<>nil Then L^.Free;
     BR.TakeData (EV,RM1);
End;

Procedure TGeomBody.GiveRes (Var Res : EngValue; Var RM : TRM);

Var RM1 : TRM;
    EV : TEVData;
    L : ListObjPtr;
    R : ExprRealObjPtr;

Begin
     RM := OK;
     Res.Init (RM1);
     EV[1] := 0;  EV[2] := 0;  EV[3] := 0;

     L := Pointer(CalcProp(Props,'Res',New(ListObjPtr,Init),ActualParamList));
     If (L<>nil) and (TypeOf(L^)=TypeOf(ListObj)) Then
        Begin
             R := L^.Get (1);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[1] := R^.Value;
             R := L^.Get (2);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[2] := R^.Value;
             R := L^.Get (3);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[3] := R^.Value;
        End;
     If L<>nil Then L^.Free;
     Res.TakeData (EV,RM1);
End;

Procedure TGeomBody.GiveNormVecAt (Const P : TABeam;
                                   Var V : GVec; Var RM : TRM);
Var RM1 : TRM;
    OB1 : TOBeam;
    R : ExprRealObjPtr;
    L,L1 : ListObjPtr;
    R1,Rx,Ry,Rz : Real;
    Coord: TVCoord;
```

```
Begin
     RM := OK;
     GlobalToLoacalSystem (P.Coord,Coord);
     New (L,Init);
     L^.Add (New(ExprRealObjPtr,Init(Coord[1])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[2])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[3])));
     New (L1,Init);
     L1^.Add (New(ExprListObjPtr,Init(L)));
     R := Pointer(CalcProp(Props,'Form',L1,ActualParamList));
     If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj))
        Then R1 := R^.Value Else R1 := 0;
     If R<>nil Then R^.Free;

     New (L,Init);
     L^.Add (New(ExprRealObjPtr,Init(Coord[1]+GeomBody_EpsNorm)));
     L^.Add (New(ExprRealObjPtr,Init(Coord[2])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[3])));
     New (L1,Init);
     L1^.Add (New(ExprListObjPtr,Init(L)));
     R := Pointer(CalcProp(Props,'Form',L1,ActualParamList));
     If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj))
        Then Rx := R^.Value Else Rx := 0;
     If R<>nil Then R^.Free;

     New (L,Init);
     L^.Add (New(ExprRealObjPtr,Init(Coord[1])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[2]+GeomBody_EpsNorm)));
     L^.Add (New(ExprRealObjPtr,Init(Coord[3])));
     New (L1,Init);
     L1^.Add (New(ExprListObjPtr,Init(L)));
     R := Pointer(CalcProp(Props,'Form',L1,ActualParamList));
     If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj))
        Then Ry := R^.Value Else Ry := 0;
     If R<>nil Then R^.Free;

     New (L,Init);
     L^.Add (New(ExprRealObjPtr,Init(Coord[1])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[2])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[3]+GeomBody_EpsNorm)));
     New (L1,Init);
     L1^.Add (New(ExprListObjPtr,Init(L)));
     R := Pointer(CalcProp(Props,'Form',L1,ActualParamList));
     If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj))
        Then Rz := R^.Value Else Rz := 0;
     If R<>nil Then R^.Free;

     Coord[1] := (Rx-R1)/GeomBody_EpsNorm;
     Coord[2] := (Ry-R1)/GeomBody_EpsNorm;
     Coord[3] := (Rz-R1)/GeomBody_EpsNorm;
     LoacalToGlobalSystemV (Coord,V.Coord);
End;

Procedure TGeomBody.GiveKDAt (Const P : TABeam; Prv : PBody;
                              Var KD : EngValue; Var RM : TRM);
Var RM1 : TRM;
    EV : TEVData;
    L,L1 : ListObjPtr;
    R : ExprRealObjPtr;
    Coord: TVCoord;

Begin
     RM := OK;
     KD.Init (RM1);
     EV[1] := 0;  EV[2] := 0;  EV[3] := 0;

     GlobalToLoacalSystem (P.Coord,Coord);
     New (L,Init);
     L^.Add (New(ExprRealObjPtr,Init(Coord[1])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[2])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[3])));
     New (L1,Init);
     L1^.Add (New(ExprListObjPtr,Init(L)));
     L := Pointer(CalcProp(Props,'KD',L1,ActualParamList));
     If (L<>nil) and (TypeOf(L^)=TypeOf(ListObj)) Then
        Begin
             R := L^.Get (1);
```

```
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[1] := R^.Value;
             R := L^.Get (2);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[2] := R^.Value;
             R := L^.Get (3);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[3] := R^.Value;
        End;
     If L<>nil Then L^.Free;
     KD.TakeData (EV,RM1);
End;

Procedure TGeomBody.GiveRazRAt (Const P : TABeam;
                                Var RazR : EngValue; Var RM : TRM);
Var RM1 : TRM;
    EV : TEVData;
    L,L1 : ListObjPtr;
    R : ExprRealObjPtr;
    Coord: TVCoord;

Begin
     RM := OK;
     RazR.Init (RM1);
     EV[1] := 0;  EV[2] := 0;  EV[3] := 0;

     GlobalToLoacalSystem (P.Coord,Coord);
     New (L,Init);
     L^.Add (New(ExprRealObjPtr,Init(Coord[1])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[2])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[3])));
     New (L1,Init);
     L1^.Add (New(ExprListObjPtr,Init(L)));
     L := Pointer(CalcProp(Props,'DiffuseR',L1,ActualParamList));
     If (L<>nil) and (TypeOf(L^)=TypeOf(ListObj)) Then
        Begin
             R := L^.Get (1);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[1] := R^.Value;
             R := L^.Get (2);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[2] := R^.Value;
             R := L^.Get (3);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[3] := R^.Value;
        End;
     If L<>nil Then L^.Free;
     RazR.TakeData (EV,RM1);
End;

Procedure TGeomBody.GiveTnAt (Const P : TABeam;
                              Var Tn : Integer; Var RM : TRM);
Var RM1 : TRM;
    L,L1 : ListObjPtr;
    R : ExprRealObjPtr;
    Coord: TVCoord;

Begin
     RM := OK;

     GlobalToLoacalSystem (P.Coord,Coord);
     New (L,Init);
     L^.Add (New(ExprRealObjPtr,Init(Coord[1])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[2])));
     L^.Add (New(ExprRealObjPtr,Init(Coord[3])));
     New (L1,Init);
     L1^.Add (New(ExprListObjPtr,Init(L)));
     R := Pointer(CalcProp(Props,'Tn',L1,ActualParamList));
     If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj))
        Then Tn := Trunc(R^.Value) Else Tn := 0;
     If R<>nil Then R^.Free;
End;

Procedure TGeomBody.GiveLtR (Var LtR : EngValue; Var RM : TRM);

Var RM1 : TRM;
    EV : TEVData;
    L : ListObjPtr;
    R : ExprRealObjPtr;

Begin
     RM := OK;
     LtR.Init (RM1);
```

```
     EV[1] := 0;  EV[2] := 0;  EV[3] := 0;

     L := Pointer(CalcProp(Props,'LtR',New(ListObjPtr,Init),ActualParamList));
     If (L<>nil) and (TypeOf(L^)=TypeOf(ListObj)) Then
        Begin
             R := L^.Get (1);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[1] := R^.Value;
             R := L^.Get (2);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[2] := R^.Value;
             R := L^.Get (3);
             If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then EV[3] := R^.Value;
        End;
     If L<>nil Then L^.Free;
     LtR.TakeData (EV,RM1);
End;

{ IShowObj }

Constructor IShowObj.Init;

Begin
     Inherited Init ('Show',0,Assoc_Functional,nil);
End;

Function IShowObj.Calc (ActualParamList : ParamListObjPtr) : Pointer;

Const aRes : TEVData = (1,1,1);
      aLtR : TEVData = (1.1,1.1,1.1);
      AirKD1 : TEVData = (0,0,0);
      aBR : TEVData = (0.99,0.99,0.99);
      aRR : TEVData = (0,0,0);
      AirTn2 : Integer = 1;

      MnI : TEVData = (0,0,0 );
      MxI : TEVData = (0.7,0.7,0.7);

      OCoord : TVCoord = (0,0,0);
      OxCoord: TVCoord = (1,0,0);
      OyCoord: TVCoord = (0,1,0);
      OzCoord: TVCoord = (0,0,1);
      ViewPointCoord : TVCoord = (0,0,1);

Var RM1,RM2 : TRM;
    ErrSum : TSumRM;
    GBody : ^TGeomBody;
    Light : ^TPointLight;
    RLt : TRazLtPoint;
    AirBody : TAir2;
    AirRes : EngValue;
    AirLtR : EngValue;
    I : LongInt;
    J : Integer;
    P : Pointer;

    lCenter : TVCoord; fCenter : Boolean;
    lPower : TEVData; fPower : Boolean;
    rePower : TEVData;
    reDist : Real;

    O1 : RootObjPtr; L1 : ListObjPtr absolute O1;
    L : ListObjPtr;
    R : ExprRealObjPtr;

Begin
     ErrSum := OK; RM2 := OK;
     Echo := True;

     Bodies.Init(RM1);  AddErr(ErrSum,RM1);
     Lights.Init(RM1);  AddErr(ErrSum,RM1);

     WSizeX := 0;
     WSizeY := 0;

     rePower[1] := 0;
     rePower[2] := 0;
     rePower[3] := 0;
     reDist := 3;
```

```
O1 := ActualParamList^.GetParam ('_ParamsCount');
For I := 1 to Trunc(ExprRealObjPtr(O1)^.Value) do
    Begin
         O1 := ParamObjPtr(ActualParamList^.Get(I))^.Value;
         If (O1<>nil) and (TypeOf(O1^)=TypeOf(ListObj)) Then
            Begin
                 L := Pointer(CalcProp(L1,'Position',New(ListObjPtr,Init),ActualParamList));
                 fCenter := (L<>nil) and (TypeOf(L^)=TypeOf(ListObj));
                 If fCenter Then
                    Begin
                         R := L^.Get (1);
                         If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj))
                            Then lCenter[1] := R^.Value Else lCenter[1] := 0;
                         R := L^.Get (2);
                         If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj))
                            Then lCenter[2] := R^.Value Else lCenter[2] := 0;
                         R := L^.Get (3);
                         If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj))
                            Then lCenter[3] := R^.Value Else lCenter[3] := 0;
                    End;
                 If L<>nil Then L^.Free;
                 L := Pointer(CalcProp(L1,'Energyes',New(ListObjPtr,Init),ActualParamList));
                 fPower := (L<>nil) and (TypeOf(L^)=TypeOf(ListObj));
                 If fPower Then
                    Begin
                         R := L^.Get (1);
                         If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj))
                            Then lPower[1] := R^.Value Else lPower[1] := 0;
                         R := L^.Get (2);
                         If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj))
                            Then lPower[2] := R^.Value Else lPower[2] := 0;
                         R := L^.Get (3);
                         If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj))
                            Then lPower[3] := R^.Value Else lPower[3] := 0;
                    End;
                 If L<>nil Then L^.Free;

                 If fCenter and fPower Then
                    Begin
                         New (Light,Init(RM1)); AddErr(ErrSum,RM1);
                         Light^.Power.TakeData(lPower,RM1); AddErr(ErrSum,RM1);
                         Light^.Center.TakeCoord(lCenter,RM1); AddErr(ErrSum,RM1);
                         Lights.Take(Light,RM1);  AddErr(ErrSum,RM1);
                   End
                 Else If fPower Then
                         Begin
                              R := Pointer(CalcProp(L1,'Distance',
                                           New(ListObjPtr,Init),ActualParamList));
                              If (R<>nil) and (TypeOf(R^)=TypeOf(ExprRealObj)) Then
                                 Begin
                                      rePower := lPower;
                                      reDist := R^.Value;
                                 End;
                              If R<>nil Then R^.Free;
                         End
                       Else
                         Begin
                              R := Pointer(CalcProp(L1,'Res'{Form},
                                           New(ListObjPtr,Init),ActualParamList));
                              If R<>nil Then
                                 Begin
                                      New (GBody,Init(L1,ActualParamList,RM1));
                                      Bodies.Take(GBody,RM1); AddErr(ErrSum,RM1);
                                 End;
                              If R<>nil Then R^.Free;
                         End;
            End
         Else If (O1<>nil) and (TypeOf(O1^)=TypeOf(ExprRealObj)) Then
                 Begin
                      If WSizeX=0 Then WSizeX := Abs(Trunc(ExprRealObjPtr(O1)^.Value))
                                  Else WSizeY := Abs(Trunc(ExprRealObjPtr(O1)^.Value));
                 End;
    End;

RLt.Init(RM1);      AddErr(ErrSum,RM1);
RLt.Power:=rePower;
```

```
     RLt.rDist:=reDist;
     RE:=@RLt;

     AirRes.Init(RM1);               AddErr(ErrSum,RM1);
     AirRes.TakeData(aRes,RM1);      AddErr(ErrSum,RM1);
     AirLtR.Init(RM1);               AddErr(ErrSum,RM1);
     AirLtR.TakeData(aLtR,RM1);      AddErr(ErrSum,RM1);
     AirBody.Init(RM1);              AddErr(ErrSum,RM1);
     AirBody.airKD:=AirKD1;
     AirBody.airTn:=AirTn2;
     AirBody.airBR:=aBR;
     AirBody.airRR:=aRR;
     AirBody.Prozr:=True;

     AirBody.TakeRes(AirRes,RM1);    AddErr(ErrSum,RM1);
     AirBody.TakeLtR(AirLtR,RM1);    AddErr(ErrSum,RM1);
     Air:=@AirBody;

     MinInt.Init(RM1);
     MinInt.TakeData(MnI,RM1);
     MaxInt.Init(RM1);
     MaxInt.TakeData(MxI,RM1);

     O.Init(RM1); AddErr(ErrSum,RM1);
     O.TakeCoord(OCoord,RM1); AddErr(ErrSum,RM1);
     Oz.Init(RM1);
     Oz.TakeCoord(OzCoord,RM1); AddErr(ErrSum,RM1);
     Ox.Init(RM1);
     Ox.TakeCoord(OxCoord,RM1); AddErr(ErrSum,RM1);
     Oy.Init(RM1);
     Oy.TakeCoord(OyCoord,RM1); AddErr(ErrSum,RM1);

     ScreenMinX := 0;  ScreenMinY := 0;
     MaxColor := 255;
     LenX := 320;    LenY := 200;
     MinPosX := 0;   MinPosY := 0;
     WMinX :=-1.0;    WMinY :=-1.0;
     WMaxX := 1.0;    WMaxY := 1.0;
     If WSizeX=0 Then WSizeX := 10;
     If WSizeY=0 Then WSizeY := WSizeX;
     ProjMode := Usp;
     ViewPoint.Init(RM1);  AddErr(ErrSum,RM1);
     ViewPoint.TakeCoord(ViewPointCoord,RM1); AddErr(ErrSum,RM1);
     OutputFileName := 'SW_IMAGE';
     DelayTime := 0;
     MaxGen := 2;
     OutputFileMode := RGB;
     Warning := False;
{}
     OpenIt(RM1); AddErr(ErrSum,RM1);
     If ErrSum=OK Then
        Begin
             CheckParam(Warning,RM1);AddErr(ErrSum,RM1);
             If ErrSum=OK Then MakePicture(RM2);  AddErr(ErrSum,RM2);
        End;
     CloseIt(RM1); AddErr(ErrSum,RM1);
{}
     Lights.GiveCount (J,RM1); AddErr(ErrSum,RM1);
     For I := 0 to J-1 do
         Begin
              Lights.GiveAt (I,Pointer(Light),RM1); AddErr(ErrSum,RM1);
              Dispose (Light,Done(RM1)); AddErr(ErrSum,RM1);
         End;
     Lights.Done(RM1); AddErr(ErrSum,RM1);

     Bodies.GiveCount (J,RM1); AddErr(ErrSum,RM1);
     For I := 0 to J-1 do
         Begin
              Bodies.GiveAt (I,Pointer(GBody),RM1); AddErr(ErrSum,RM1);
              Dispose (GBody,Done(RM1)); AddErr(ErrSum,RM1);
         End;
     Bodies.Done(RM1); AddErr(ErrSum,RM1);

     If (RM2<>OK) or (ErrSum<>OK)
        Then Calc := nil
        Else Calc := New (ExprRealObjPtr,Init(1));
End;
```

```
{}

Const UnitActive : Byte = 0;  { семафор за инициализация на модула }

Procedure InitUnit;

Begin
     Inc (UnitActive);
     If UnitActive>1 Then Exit;

     uL1.InitUnit;
     uL2.InitUnit;
     Syntax.InitUnit;

     DefineFunction (New(IShowObjPtr,Init));
End;

Procedure DoneUnit;

Begin
     Dec (UnitActive);
     If UnitActive>0 Then Exit;

     Syntax.DoneUnit;
     uL2.DoneUnit;
     uL1.DoneUnit;
End;


End.
```

Изходен текст на модула uL4

```
Unit uL4; { Модул ниво четвърто }

Interface

Uses uL1,uL2,Syntax;  { Използва: модул ниво първо;
                                  модул ниво второ;
                                  модул за работа с Null-низове. }

Type { Обект графична виртуална машина }
     GVMObjPtr = ^GVMObj;
     GVMObj = Object(RootObj)
        FuncList : ListObjPtr;     { списък от дефинираните функции (, тела, обекти) }
        Synt : SyntaxAnaliserObj;  { анализатор на командите }

        Constructor Init (BaseFunctions : ListObjPtr);
        Destructor  Done; virtual;
        Function  Copy : Pointer; virtual;

        { Настройка на средата }
        Procedure InitHardware; virtual;
        { Извеждане на резултатите }
        Procedure ReturnResult (Res : ListObjPtr); virtual;
        { Изпълнява команда(и) }
        Procedure Execute (Command : PChar; Echo : Boolean);
        { Изпълнява програма (Echo - дали да се съобщават резултатите) }
        Procedure Run (FileName : PChar; Echo : Boolean);
     End;

Procedure InitUnit; { Първоначална инициализация на модула }
Procedure DoneUnit; { Край на работата с модула }

Implementation

{ GVMObj }

Constructor GVMObj.Init (BaseFunctions : ListObjPtr);
```

```
Begin
     Inherited Init;
     FuncList := BaseFunctions;
     Synt.Init;
     Run ('AUTOEXEC.GVM',False);
End;

Destructor GVMObj.Done;

Begin
     Synt.Done;
     FuncList^.Free;
     Inherited Done;
End;

Function GVMObj.Copy : Pointer;

Begin
     InitHardware;
     Copy := New (GVMObjPtr,Init(FuncList^.Copy));
End;

Procedure GVMObj.InitHardware;

Begin
     SetFunctionsList (FuncList);
End;

Procedure GVMObj.ReturnResult (Res : ListObjPtr);

Var I : LongInt;
    O : ExpressionObjPtr;

Begin
     For I := 1 to Res^.GetCount do
         Begin
              O := Res^.Get (I);
              Write ('Result(',I,')=');
              If O<>nil Then WriteLn (O^._Write)
                        Else WriteLn ('Error!');
         End;
End;

Procedure GVMObj.Execute (Command : PChar; Echo : Boolean);

Var Res,Res1 : ListObjPtr;
    F : File;
    L,I : LongInt;
    O : ExpressionObjPtr;
    P : ParamListObjPtr;

Begin
     InitHardware;
     Res := Synt.Compile (Command);
     New (P,Init);
     New (Res1,Init);
     For I := 1 to Res^.GetCount do
         Begin
              O := Res^.Get (I);
              If O<>nil Then O := O^.Calc (P);
              Res1^.Add (O);
         End;
     Res^.Free;
     P^.Free;
     If Echo Then ReturnResult (Res1);
     Res1^.Free;
End;

Procedure GVMObj.Run (FileName : PChar; Echo : Boolean);

Var Res,Res1 : ListObjPtr;
    F : File;
    S : PChar;
    L,I : LongInt;
    O : ExpressionObjPtr;
    P : ParamListObjPtr;
```

```
Begin
     Assign (F,FileName);
     {$I-}
     Reset (F,1);
     I := IOResult;
     If I<>0 Then
        Begin
             If Echo Then WriteLn ('DOS error #',I);
             Exit;
        End;
     {$I+}
     L := FileSize(F);
     GetMem (S,L+1);
     BlockRead (F,S^,L);
     (S+L)^ := #0;

     Execute (S,Echo);

     FreeMem (S,L+1);
     Close (F);
End;

{}

Const UnitActive : Byte = 0; { семафор за инициализация на модула }

Procedure InitUnit;

Begin
     Inc (UnitActive);
     If UnitActive>1 Then Exit;

     uL1.InitUnit;
     uL2.InitUnit;
     Syntax.InitUnit;
End;

Procedure DoneUnit;

Begin
     Dec (UnitActive);
     If UnitActive>0 Then Exit;

     Syntax.DoneUnit;
     uL2.DoneUnit;
     uL1.DoneUnit;
End;


End.
```

Изходен текст на програмата G-Client

```
Program G_Client;

Uses Strings,uL1,uL2,Lexical,Syntax,uL3,uL4;

Var GVM : GVMObjPtr;
    S : Array[0..255] of Char;

Begin
     WriteLn (#10#13'G-Client v1.0'#10);

     uL1.InitUnit;
     uL2.InitUnit;
     Lexical.InitUnit;
     Syntax.InitUnit;
     uL3.InitUnit;
     uL4.InitUnit;

     New (GVM,Init(GetFunctionsList));
```

```
     Repeat
           Write (':>');
           ReadLn (S);
           Case S[0] of
{ изход }        '.' : Break;
{ програма }     '!' : Begin
                           If StrScan(S,'.')=nil Then StrCat (S,'.GVM');
                           GVM^.Run (@S[1],True);
                      End;
{ команда }      Else  GVM^.Execute (S,True);
           End;
     Until False;

     SetFunctionsList (nil);
     GVM^.Free;

     uL4.DoneUnit;
     uL3.DoneUnit;
     Syntax.DoneUnit;
     Lexical.DoneUnit;
     uL2.DoneUnit;
     uL1.DoneUnit;
End.
```

Изходен текст на AUTOEXEC.GVM

```
new nil()=[];

new BaseDimension()=3;

new System(point)=point;
new SystemBack(point)=point;
new Distance(point1,point2)=sqr((point2-point1)*(point2-point1));

new Solid()=
       [
        property LocalSystem(point)=point,
        property LocalSystemBack(point)=point,
        property Form(point)=0,
        property RR(point)=[0,0,0],
        property BR(point)=[0,0,0],
        property Res()=[0,0,0],
        property KD(point)=[0,0,0],
        property DiffuseR(point)=[1,1,1],
        property Tn(point)=0,
        property LtR()=[0,0,0]
       ] & nil();

new PointLight(center,eng)=
       [
        property Position()=center,
        property Energyes()=eng
       ] & nil();

new DiffuseLight(eng,dist)=
       [
        property Energyes()=eng,
        property Distance()=dist
       ] & nil();
```

Приложение 2 - демострационни примери

```
new LS(point,pos,vx,vy,vz,num)=[((point+pos)*vx.num),
                                ((point+pos)*vy.num),
                                ((point+pos)*vz.num)];
new Solid3D(pos,vx,vy,vz)=
       [
        property LocalSystem(point)=[LS(point,pos,vx,vy,vz,1),
                                     LS(point,pos,vx,vy,vz,2),
                                     LS(point,pos,vx,vy,vz,3)],
        property LocalSystemBack(point)=[vx*point,vy*point,vz*point]-pos
       ] & Solid();

new Plane(pos,vx,vy,vz)=
       [
        property Form(point)=abs(pos.3)
       ] & Solid3D(pos,vx,vy,vz);

new Checker(pos,vx,vy,vz,xLen,yLen,k,RR_C,BR_C,KD_C)=
       [
        property IFPos(point,A)=If(Odd(Round(Point.1/xLen)+Round(Point.2/yLen)),A,(A*k)),
        property RR(point)=Self.IFPos(point,RR_C),
        property BR(point)=Self.IFPos(point,BR_C),
        property KD(point)=Self.IFPos(point,KD_C)
       ] & Plane(pos,vx,vy,vz);

new StarN(pos,vx,vy,vz,r,N)=
       [
        property Form(point)=Sqrt(point*point)-
                             r*(2-1.3*Abs(Sin(N/2*ArcTan(point.2/point.1))))
       ] & Solid3D(pos,vx,vy,vz);

new Star8()=
       [
        property RR(point)=[0.5,0.5,0.1],
        property BR(point)=[0.9,0.9,0.1],
        property Res()=[1,1,1],
        property KD(point)=[0.1,0.1,0.9],
        property DiffuseR(point)=[1,1,1],
        property Tn(point)=8,
```

```
        property LtR()=[1.3,1.3,1.3]
       ] & StarN([~0.2,0.3,1.3],[1,0,0],[0,1,0],[0,0,1],0.35,8);
new Checker1()=
       [
        property Res()=[1,1,1],
        property Tn(point)=5,
        property LtR()=[1.3,1.3,1.3]
       ] & Checker([0,0,2],[1,0,0],[0,1,0],[0,0,1],0.2,0.2,0.5,
                   [0.5,0.5,0.5],[0.5,0.4,0.7],[0.8,0.5,0.5]);
new Light1()=PointLight([0,3,~2],[70,70,70]);
new Light2()=PointLight([0,~3,~2],[10,10,10]);
new LightD()=DiffuseLight([0,0,0],3);

show(320,200,LightD(),Light1(),Light2(),Checker1(),Star8());
```

```
new LS(point,pos,vx,vy,vz,num)=[((point+pos)*vx.num),
                                ((point+pos)*vy.num),
                                ((point+pos)*vz.num)];
new Solid3D(pos,vx,vy,vz)=
       [
        property LocalSystem(point)=[LS(point,pos,vx,vy,vz,1),
                                     LS(point,pos,vx,vy,vz,2),
                                     LS(point,pos,vx,vy,vz,3)],
        property LocalSystemBack(point)=[vx*point,vy*point,vz*point]-pos
       ] & Solid();

new HalfSpace(pos,vx,vy,vz)=
       [
        property Form(point)=pos.3
       ] & Solid3D(pos,vx,vy,vz);

new Plane(pos,vx,vy,vz)=
       [
        property Form(point)=abs(pos.3)
       ] & Solid3D(pos,vx,vy,vz);

new Checker(pos,vx,vy,vz,xLen,yLen,k,RR_C,BR_C,KD_C)=
       [
        property IFPos(point,A)=If(Odd(Round(Point.1/xLen)+Round(Point.2/yLen)),A,(A*k)),
        property RR(point)=Self.IFPos(point,RR_C),
        property BR(point)=Self.IFPos(point,BR_C),
        property KD(point)=Self.IFPos(point,KD_C)
       ] & Plane(pos,vx,vy,vz);

new Sphare(pos,vx,vy,vz,r)=
       [
        property Form(point)=Sqrt(point*point)-r
       ] & Solid3D(pos,vx,vy,vz);

new Sphare1()=
       [
        property BR(point)=[0.7,0.5,0.8],
        property Res()=[1,1,1],
        property KD(point)=[0.2,0.4,0.2],
        property DiffuseR(point)=[1,1,1],
        property Tn(point)=10,
        property LtR()=[1.3,1.3,1.3]
       ] & Sphare([~0.2,0.3,1.3],[1,0,0],[0,1,0],[0,0,1],0.5);
```

```
new Checker1()=
       [
        property Res()=[1,1,1],
        property Tn(point)=2,
        property LtR()=[1.3,1.3,1.3]
       ] & Checker([2,0,0],[~1,1,~1.5],[1,2.5,~1],[1,0,1],0.1,0.1,0.5,
                   [0.5,0.5,0.5],[0.7,0.5,0.8],[0.5,0.2,0.2]);
new HalfSpace1()=
       [
        property RR(point)=[0.6,0.6,0.6],
        property BR(point)=[0.7,0.5,0.8],
        property Res()=[1,1,1],
        property KD(point)=[0.2,0.2,0.5],
        property Tn(point)=5,
        property LtR()=[1.3,1.3,1.3]
       ] & HalfSpace([~2,0,0],[1,1,0],[1,~1,~1],[~1,0,1]);
new Light1()=PointLight([0,3,~2],[90,100,80]);
new LightD()=DiffuseLight([0,0,0],3);

show(320,200,LightD(),Light1(),Sphare1(),Checker1(),HalfSpace1());
```

Съдържание