СУ "СВ. КЛИМЕНТ ОХРИДСКИ"

ПАВЕЛ ХРИСТОВ БОЙЧЕВ

Д И С Е Р Т А Ц И Я

СОФИЯ, 2000

СОФИЙСКИ УНИВЕРСИТЕТ "СВЕТИ КЛИМЕНТ ОХРИДСКИ"
ФАКУЛТЕТ ПО МАТЕМАТИКА И ИНФОРМАТИКА
КАТЕДРА "ИНФОРМАЦИОННИ ТЕХНОЛОГИИ"

ПАВЕЛ ХРИСТОВ БОЙЧЕВ

ЕЗИК И СИСТЕМА, БАЗИРАНИ НА LOGO

Д И С Е Р Т А Ц И Я

за присъждане на образователната и научна степен "доктор"

Научен ръководител:
доцент д-р Божидар Сендов

София, 2000

Резюме на съдържанието

СЪДЪРЖАНИЕ

Таблици

Фигури

Примери

I ВЪВЕДЕНИЕ

I.1 Увод

По света съществуват десетки опити да се проектира и реализира интерактивна езиково-ориентирана компютърна среда, чрез която да могат да бъдат създавани сложни геометрични конструкции и да се изучава поведението им при промяна на първоначалните параметри. Освен това, от подобни системи се изисква да са достатъчно интуитивни, за да могат да се ползват и от ученици в средните училища и дори от деца [13]. Една от най-успешните подобни системи е Система Планиметрия - Геомландия [7, 12, 21, 22]. Тя е създадена през 1985 г. в Лабораторията по автоматизирани системи за обучение (сега Катедра "Информационни Технологии") към Факултета по Математика и Информатика на СУ "св. Климент Охридски", под ръководството на доц. Божидар Сендов. Езикът на системата представлява геометрично разширение на езика Logo. Разширенията са насочени към използване на средства за директно манипулиране с богат набор от базови обекти.

Главна особеност на софтуера, особено този, който се използва в образованието, е, че изискванията към него постоянно се увеличават. Независимо от прогресивността на Система Планиметрия, новите изисквания към нея налагат следните промени:

- ядрото на системата да се поддържа по-лесно, като се замени асемблероподобния интерпретируем програмен език, поддържащ работа с атоми и списъци, с по-приемлива алтернатива;
- интерфейсът да се модернизира така, че да следва тенденциите и новостите при дизайна на графични среди;
- системата да използва пълната налична памет, независимо от наличните хардуерни ресурси;
- потребителят да може създава пълноценни собствени обекти и да генерира своите конструкции не само от вградените обекти, но и от новосъздадените от него;
- връзките между обектите да позволяват циклична зависимост на обектите;
- системата да се отвори и да стане приложима не само за планиметрия, но и за други предмети.

През есента на 1992 бе решено да се проектира и реализира нова система, която би задоволила текущите и бъдещите изисквания на потребителите към система от ранга на Система Планиметрия.

I.2 Цели и задачи на дисертацията

Целта на настоящата работа е да се проектира и реализира система, която от една страна да съдържа функционалността на система Планиметрия, а от друга – да представя много по-добри възможности за разширения и надстройки.

Смисълът на подобна система е в това, че към настоящият момент използването на софтуерни системи в обучението и в изследването е крайно ограничено. Това се поражда от факта, че подобни системи или са хубави, но прекалено тесни в смисъла на материала, който обхващат, или пък са в противоположния край – обхващат широк кръг от теми, но са използваеми само от напреднали.

Така поставената цел, може да бъде преформулирана и по следния начин: ***Необходимо е да се създаде софтуерна система за образованието, чрез която да могат да се създават конкретни модули по отделните дисциплини (когато става въпрос за образованието) или модели на процеси и устройства (когато става въпрос за изследователска дейност).***

Системата, която би удовлетворила тази цел, ще има предимството да предостави унифицирани мултидисциплинарни средства, част от които са насочени към начинаещите, а друга част – към напредналите.

За постигането на целта, дисертантът си постави следните основни задачи:

- Да се проектира нова идеология за описване и манипулиране на обекти;
- Да се проектира нов език за програмиране – диалект на езика Logo;
- Да се реализира транслатор за проектирания език;
- Да се създадат необходимо количество библиотеки, с чиято помощ транслаторът на езика да бъде практически използваем;
- Да се създадат подходящи примери, които да илюстрират новите подходи при работата с обекти.

За да се решат тези задачи, а също и поради необходимостта да се приложат идеи и решения, несъвместими с вече реализираната Геомландия, бе взето решение да се проектира и реализира нова система, която изпълнява колкото се може по-голяма част от следните неформални изисквания:

- Модерен, ергономичен и потребителски настроен графичен интерфейс, лесна преносимост и удобна помощна система;
- Относително лесно поддържане на ядрото на системата дори и от програмисти, неучаствали в първоначалното проектиране и реализация на системата;
- Възможност за изграждане на модели от различни области на науката;
- Лесно създаване на нови обекти и пълноценна работа в обектно-ориентирана среда;
- Интерактивен, команден и програмен достъп до създадените обекти.

I.3 Структура на дисертацията

Дисертацията е организирана във въведение, две глави, заключение и списък с използвана литература. Включени са и списък от публикациите на автора, свързани с дисертацията, списък от полезни web адреси, както и кратка професионална автобиография.

Пълният обем на дисертацията е 189 страници, от които първа глава заема 19 страници, втора глава 56 страници, а трета глава 97 страници. Съдържанието и списъкът на примерите, таблиците и фигурите е 8 страници. Заглавните страници, списъка на използваната литература, допълнителни WEB връзки, публикации, авторска справка и автобиография на автора са разположени на 9 страници.

Използваната литература включва 28 заглавия (5 на български език и 23 на английски). Представени са и 55 адреса на WWW сайтове с допълнителна информация. Списъкът от публикации на дисертанта, отразяващи пряко резултати от дисертацията, съдържа 9 заглавия. Работата съдържа 12 таблици, 72 фигури и 54 примера.

В първа глава се описват целите и задачите на дисертацията, прави се обзор на софтуерни системи, подобни в една или друга степен с тази, която е предмет на дисертацията и кратко описание на реализираната система.

Във втора глава се прави пълно описание на езика за програмиране от високо ниво Elica Logo. Представени са основите на езика, запазените символи и думи, дефиниране и използване на събития, правила и обекти, описание на потребителските библиотеки. Особено внимание е отделено на унификацията на метаобектите. Представени са някои от по-интересните примери.

В трета глава е описана реализацията на системата – как са проектирани и реализирани отделните модули, участващи в процеса на транслация на Elica Logo програми. Описани са и решенията на специфични проблеми и решения, възникнали в процеса на работа.

В заключението се посочват приносите, според дисертанта, както и къде са представени и публикувани резултати от работата.

В използваната литература са изброени източниците, цитирани в работата.

В списъка с публикации са включени публикациите на автора, които са пряко свързани с дисертацията.

Кратка автобиография съдържа резюме на професионалната автобиография на дисертанта, а в авторската справка е описан приносът на дисертанта.

I.4 Обзор

Система Elica е уникална според много фактори и затова е трудно тя да бъде сравнена пълноценно с други системи. Към настоящия момент такива просто не съществуват. Информационната ниша на образователния софтуер, която запълва системата, е специално подбрана така, че да не си пречи с подобни системи.

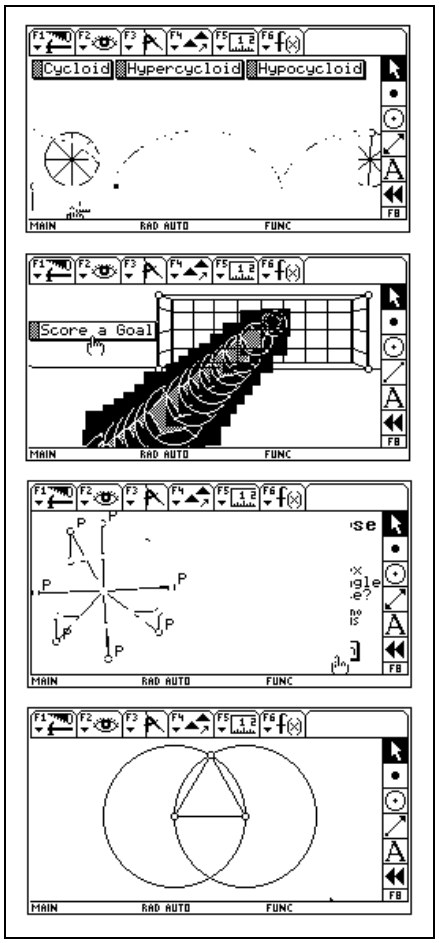
В настоящия обзор са включени кратки функционални описания на някои от сега съществуващите системи, които биха могли да имат общи възможности с реалните или потенциалните възможности на Elica. Системите са разделени в две групи. Първата обхваща системи за математически пресмятания и визуализация, а втората – различни реализации на езика за програмиране Logo.

I.4.1 Системи за математически пресмятания

I.4.1.1 The Geometer's Sketchpad

Системата GSP [14] съдържа версии за Macintosh и Windows. Предназначена е за обучение на ученици по Евклидова геометрия в равнина. Със Sketchpad може да се построяват конструкции в Евклидовото пространство чрез използване на средствата за изчертаване от toolbox и с помощта на командите от менюто Construct.

Фигура I-1 Система Geometer's Sketchpad



Командите от менюто Transform позволяват да се извършва транслация, ротация и хомотетия с фиксирана, изчислена или динамична величина. Менютата Measure и Graph ни пренасят в областта на аналитичната геометрия, където може да измерваме свойствата на нашата конструкция и да работим в правоъгълни или полярни координати. Комбинирайки менютата Edit и Display със средствата за изписване на текст може да се добавят етикети и заглавия, да се променят свойствата на изобразяване на обектите и да се създават анимации. С помощта на Script може да капсуловат сложни конструкции в единични стъпки, разширявайки възможностите на Sketchpad. Sketchpad е динамична геометрична среда. Позволява dragging на поддържаните основни обекти - точка, окръжност, права, отсечка, лъч. Дава възможност и за геометричните преобразувания транслация, ротация и хомотетия и работа с релации между обекти: точка върху даден обект, пресечна точка, среда, перпендикулярна права, успоредна права, ъглополовяща, дъга от окръжност, ГМТ (locus), запълване на вътрешността на обект и други.

В Sketch всичко се изчертава с мишката. В Script може да се запази последователността на изчертаване на конструкция. След което тя може рекурсивно да се приложи.

Основните недостатъци на системата са, че не може да се надгражда с нови обекти, не могат да се задават други релации между обектите т.е. всичко е фиксирано. Няма език за програмиране. След като се намери множеството от точки не може да се извършват изследвания над него. Няма елипси, хиперболи, параболи, стереометрия и т.н.

Предимствата на системата са, че е доста стабилна, красива и бърза; не е претрупана с много бутони и лесно се усвоява. Работи на различни платформи. Има версии и за по-стари компютри.

I.4.1.2 Algebra II

Algebra II [9] е система за Macintosh. Предназначена е за обучение на ученици по алгебра с изрази, равенства, матрици, таблици и графики. Има обекти, които могат да се придвижват, да си променят формата, да са свързани помежду си и т.н.

Системата предоставя средства за изчертаване на графики (максимум на 3 графики на една координата ос); поле за пресмятане на стойности по формула; пресмятане на стойност на функция в дадена точка; калкулатор; уравнения на конични сечения; матрици; аритметични операции (+ - / *), таблици; случайни числа; сравнение за равенство на две входни стойности и други.

I.4.1.3 Function Analyzer

Системата Function Analyzer [15] има версии за IBM PS/2, PC DOS. Предназначена е за обучение на ученици по алгебра от 7 до 12 клас за изучаване на зависимостите между различните представяния на алгебричните функции: символно, графично и таблично, за да се разберат ефектите, които се получават при манипулиране с изразите и графиките на функциите - да се видят връзките в класове от функции. Системата може да се използва и за да се разбере важността на мащаба и неговия ефект върху изобразяването на функцията, а също така и да се изследва поведението и формата на полиномни функции.

Function Analyzer предоставя средства за изследване и манипулиране на функции, представени като изрази или графики. Функциите са разположени на три взаимосвързани мрежи: голям изглед; уголемен мащаб, намален мащаб. Може да се обработват изразите на функциите, като се сменят техните коефициенти. Освен това могат да се обработват графиките на функциите, като се транслират, увеличава се интервалът им или се създава техен огледален образ, а също да се проверяват стойности на функция, да се изследват точки в координата равнина или да се променя мащабът на координатната равнина.

Има библиотека с функции, които могат да се избират случайно и да се показват графиките им на екрана, без да се показва символното им представяне. Ученикът трябва сам да го открие.

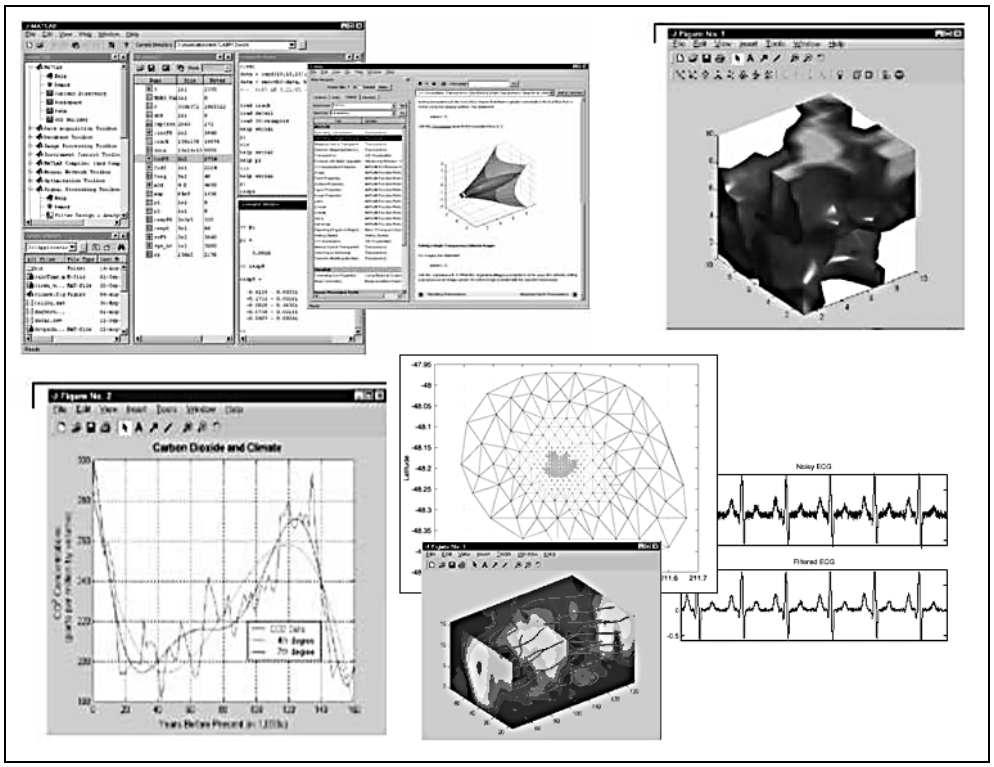
Основният недостатък на системата е, че е стара и не поддържа работа с мишка.

I.4.1.4 MathLab

Системата MATLAB се използва за моделиране и визуализиране на данни, за ускоряване на изследванията, анализите и намиране на по-ефективни решения. Системата включва средства за анализ и обработка на данни, визуализация и манипулиране на изображения, създаване на прототипи на алгоритми и тяхното разработване, създаване на модели и симулации, създаване на приложения и др.

Приложението на MATLAB е в различни приложни сфери, включващи обработване на изображения и сигнали, дизайн на системи за контрол, финанси, икономика и редица природни науки.

Фигура I-2 Система MATHLAB



Особеност на системата е, че включва интерактивен език и среда за програмиране. В езика се поддържат различни структури от данни, обектно-ориентирано програмиране, потребителски графичен интерфейс и средства за трасиране. Има и възможност да се използват процедури написани на C, C++, FORTRAN и Java.

Повечето от математическите функции в MATLAB оперират директно с вектори и матрици. В резултат на това много алгоритми се записват с малко команди и се избягва съставянето на множество вложени цикли.

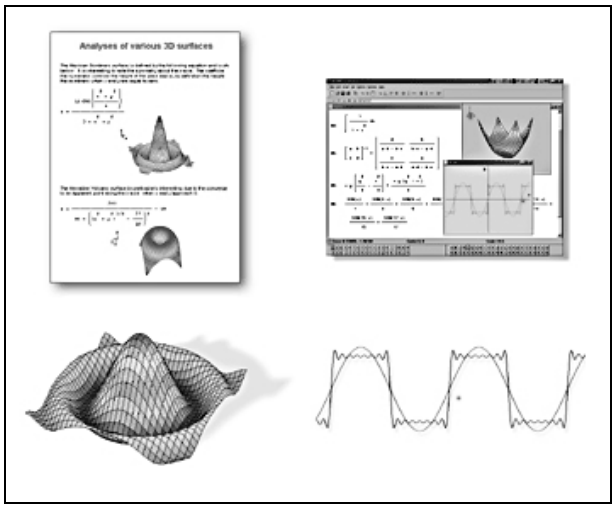
I.4.1.5 Derive

Чрез системата Derive могат да се създават матрици, да се интегрира, диференцира, да се намират гранични стойности и т.н. Системата е насочена към алгебра, тригонометрия, векторни пресмятания, матрици, числени методи и други. Математическите изрази се показват в стандартен двумерен формат. Могат да се решават символни и числени задачи.

Derive включва богата библиотека от помощни средства за дефиниране на функции за решаване на уравнения от първа и втора степен както алгебрично, така и приближено, чрез подходящи числени методи, за изчисляване на Bessel, Fresnel и елиптични интеграли и много други специални функции.

Файловете с необходимите функции се зареждат автоматично и при нужда, така че те са винаги налични.

Фигура I-3 Система Derive



Системата може да създава двумерни и тримерни графики – създава 2D и 3D чертежи за анализиране и изследване на уравненията в декартови и полярни координати. Дава възможност да се контролират аспектът, мащабът и осите, а също и визуализиране на реалните и имагинерните части.

Системата позволява трасирането на графики и показване на точните координати на точки от крива. Derive може и да начертае логическите комбинации при неравенства с две променливи.

I.4.1.6 Maple

Maple е мощно средство за аналитична математика, 2D/3D графики и изчисления с произволна точност. Най-голямото предимство на системата е в множеството от пакети, които поддържат всичко, започвайки от тензорните пресмятания до финансовите изчисления. Съдържа подобен на Pascal език за програмиране и може да използва код, написан и компилиран със C или FORTRAN. За съжаление, тази възможност е слабо документирана и трудно използваема.

Мощта на Maple може понякога да доведе до неудобни проблеми в нотациите. Ако пресметнат израз трябва да се използва в други изчисления, автоматично създаваните членове от типа на $O(x^n)$ трябва да бъдат ръчно премахвани.

I.4.1.7 Cabri-geometre

В системата Cabri [4] потребителите могат да описват чрез действия различни геометрични обекти и конструкции. Веднъж създадени те могат да се модифицират чрез промяна на едни или други от базовите им параметри.

Системата има възможност да определя някои от свойствата между обектите като паралелност, перпендикулярност и др.

Cabri е подходящо средство и за учителя, и за ученика особено в учебния процес. Според нуждите, някои от функциите на системата могат да забранят или пък да се добавят нови.

Ценно свойство на системата е, че може да запомня активността на ученика и да позволи на учителя да я проиграва, за да изследва процеса на учене и стъпките, които са били направени за достигане на окончателното решение.

I.4.1.8 Mathematica

Matematica [18] е система за изчисления, включваща специализиран език за програмиране, за създаване на математически и други приложения. Със системата могат да се правят числени и символни изчисления и преобразувания. Най-простият начин на използване на системата е като много мощен калкулатор с произволна точност.

Системата поддържа числени изчисления не само над числа, но и над структури като вектори и матрици. За тях е реализиран пълния набор от функции от намирането на обратна матрица до изчисляването на собствените стойности. Чрез вградените методи може да се прави числено интегриране, минимизиране и линейно програмиране.

Mathematica е особено силна в символните пресмятания и работата с формули. Системата може да преобразува формули, да опростява, да разкрива скоби, да привежда във вид, удобен за логаритмуване и много други. Може да работи с полиноми и рационални изрази и да намира алгебрични решения на полиномиални уравнения и на системи от уравнения.

Графичните средства на системата също са мощни. Тя може да създава двумерни и тримерни графики като потребителят може да настройва параметрите според нуждите си. Освен това, при тримерните визуализации може да се настройва режимът на сенки, оцветяване и осветяване.

За разлика от повечето среди за изчисления, Mathematica има и вграден език за програмиране от високо ниво. На него могат да се пишат и изпълняват програми. Интерес представлява фактът, че чрез езика могат да се използват няколко различни програмни стила: процедурен, чрез описване на алгоритми и блокове от команди, функционален, чрез функции и операции над тях, и стил с използване на правила и шаблони.

I.4.2 Кратка история на езика Logo

I.4.2.1 Началото

В средата на седемдесетте Seymour Papert, математик, работещ с Piaget в Женева, заминава за САЩ и заедно с Marvin Minsky основават MIT Artificial Intelligence Laboratory. През 1967 той и неговият екип създават първата версия на Logo.

Най-известното приложение на Logo е костенурковата графика. Името идва от първата реализация на подобна костенурка, която всъщност е била малък робот. Скоро той бива заместен от костенурката на екрана.

I.4.2.2 На бял свят

Широката употреба на Logo започва едва след разпространението на персоналните компютри. MIT Logo Group разработва Logo за Apple II [1] и за TI 99/4 на Texas Instruments. През 1980 пилотен проект, спонсориран от MIT и TI, започва в училище в Далас с начална база от 50 компютъра и 450 ученика. В същото време в Ню Йорк започва проектът "Компютри в училището" с дванадесет TI 99/4 компютъра в 6 училища. В последствие са добавени и няколко Apple II.

Прототипите на Logo, използвани в тези проекти, постепенно се превръщат в комерсиални продукти. Texas Instruments пускат TILOGO, а създадената през 1977 Terrapin Software предлагат Terrapin Logo, а по-късно и Logo PLUS.

През 1980 се основава нова компания – Logo Computer Systems Inc. (LCSI) с председател Seymour Papert. Компанията разработва Apple Logo. През същата година

една публикация на Seymour Papert вдъхновява хиляди преподаватели по света с интелектуалната и съзидателната мощ на Logo [28]. Именно техния ентусиазъм подпалва бума на Logo.

Създават се много нови версии на езика. В Европа, Южна Америка и Япония става известно Logo за MSX компютри, а в Северна Америка - Atari Logo и Commodore Logo. След подкрепата на фирмите производителки на хардуер на бял свят се появяват IBM Logo и Logo Learner.

След 1985 започват първите опити да се третира езикът Logo като сериозен език за програмиране, особено за новите Macintosh компютри. Coral Software разработват ObjectLogo.

I.4.2.3 Първи разширения

През 1985 LCSI предлагат LogoWriter. В системата се предлагат няколко нови възможности – обработване на текст, използване на движещи се картинки, повече от една костенурки и локализация на различни езици.

Друго разширение е създаването на LEGO Logo в MIT Media Lab. При тази система Logo програмата работи с реални мотори, сензори, светлини.

Създаденият от Terrapin Software TerrapinLogo през 1988 е усъвършенстван в Logo PLUS. Harvard Associates разработват PC Logo for DOS, а по-късно и за Windows. Сега тези две компании са слети и предлагат класическата форма на езика.

Поради липсата на нововъведения в LogoWriter в началото на 1990 ентусиазмът в САЩ към езика Logo спада, за разлика от Южна Америка, където на всеки две години ентусиасти провеждат конгреса Congreso Logo. В Япония LogoWriter се надстройва до LogoWriter2. През 1995 във Великобритания започва поредицата конференции EuroLogo. Може би и благодарение на тях в Европа се създават WinLogo (Испания) и Comenius Logo (Словакия).

I.4.2.4 Новата вълна

През последните години се забелязва бум на средите за програмиране на базата на Logo. В САЩ и Канада широко разпространение получава MicroWorlds. Системата съдържа много допълнителни свойства – средства за рисуване, редактор на форми, композитор на мелодии, поддържа многозадачна работа, която в последствие е вградена и в PCLogo for Windows. На базата на ядрото на MicroWorlds са създадени Control Lab и Control System.

В посока на паралелизма се разработва StarLogo. Хиляди костенурки се управляват в паралелен режим и взаимодействат помежду си. Системата е предназначена за изследване на децентрализирани модели.

В близките години са създадени и редица други комерсиални Logo системи, като Logo Grafico (Аржентина) и Mach Turtles Logo (Канада). Развиват се и безплатните версии. Brian Harvey създава UCBLogo за Macintosh, MSDOS и Unix. На негова база George Mills създава MSWLogo, използващ новите възможности в Windows.

I.4.3 Обзор на различни реализации на езика LOGO

I.4.3.1 PGS/Geomland

PGS (Plane Geometry System) и по-новата й версия Geomland са работни среди на базата на Logo, ориентирани към конструиране, решаване и изследване на геометрични задачи

в равнината. Работата със системата прилича на работа в лаборатория – построяват се обекти, променят се, изследват се техните свойства, изграждат се хипотези, които се проверяват дали са верни.

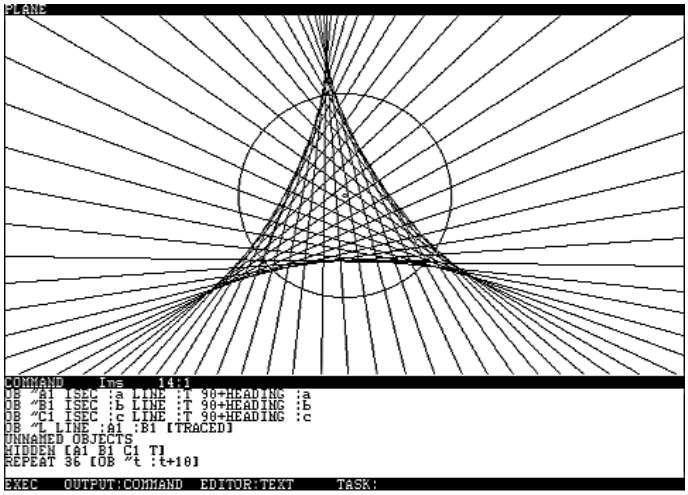
Това дава възможност да се преподава математика в по-различен стил. В системата са вградени основните геометрични обекти, използвани в училищния курс, а също и богат набор от средства за тяхното манипулиране.

Основно преимущество на системата е, че командите могат да се изпълняват в директен режим, но могат и да се съхраняват в библиотеки. За по-голямо удобство конструкциите могат да се задават и интерактивно, като в резултат на действията на потребителя се създават съответните команди на Logo.

Особеност на реализацията е, че между геометричните обекти могат да се създават връзки. По този начин могат лесно да се параметризират сложни конструкции, които се прегенерират при промяна на базисните им параметри, които могат да са не само числа, а и каквито и да е други обекти.

Към системата са създадени различни допълнителни библиотеки – за стереометрия, за оптика, за решаване на Аполониевите задачи, система с помощна информация, библиотека от модели механизми, и много други.

Фигура I-4 Система Планиметрия (PGS, Geomland)



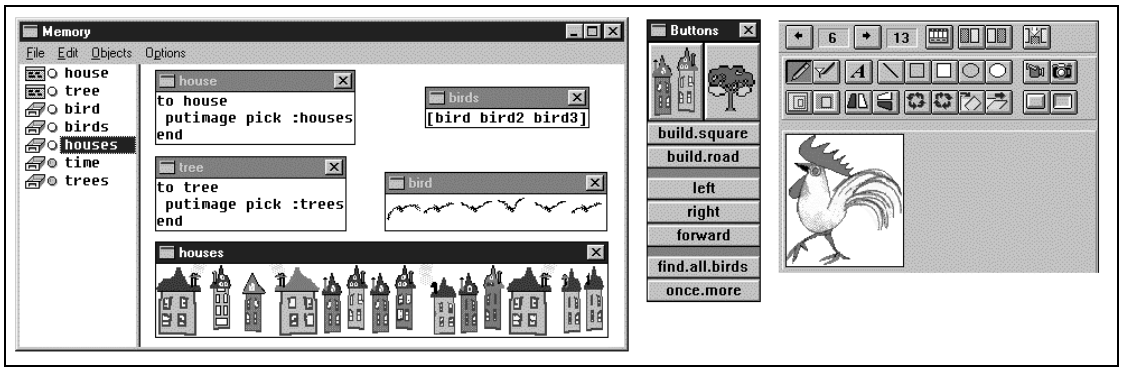
I.4.3.2 Comenius Logo

Системата Comenius Logo [2] е създаден от Университета Комениус, Словакия. Тя е предназначена за Windows и е с добри възможности за анимация. Локализирана е за различни страни, под различни имена – Унгария, Полша, България, Гърция, Чехия, Португалия, Бразилия, Великобритания, Холандия, Белгия и Германия.

Спрямо традиционното Logo системата въвежда нови типове данни, възможности за показване на мултимедийно съдържание, графични процедури и др. Идеологически, авторите на системата разширяват полиморфизмът на данните в езика Logo така, че да покрива и новите обекти.

Друга особеност е, че използването на костенуркова графика е силно застъпено, като броят, видът и поведението на костенурките може да се определя от потребителя.

Фигура I-5 Система Comenius Logo

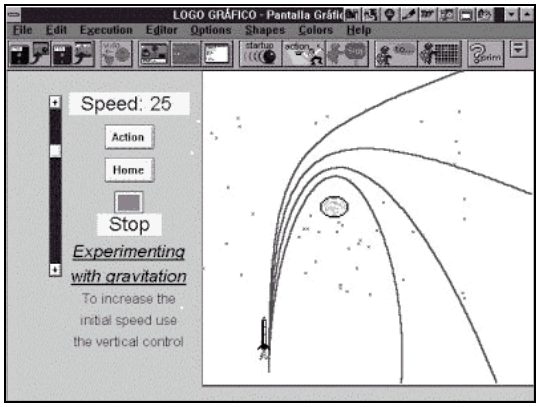


I.4.3.3 Logo Grafico / Graphic Logo

Logo Grafico и Graphic Logo са една и съща система, разработена от Fundaustral (Аржентина). Първата е версията на испански и португалски, а втората – на английски. Системата е подходяща за създаване на игри, анимации и симулации.

Системата набляга на симулациите и на специален режим на анимация, наречен "симулационна анимация", защото според авторите тези средства са също толкова ценни като тези, предоставяни от костенурковата графика.

Фигура I-6 Система Logo Grafico / Graphic Logo



Режимът на анимация добавя динамичен и кинетичен смисъл към математическия, който вече съществува и дава възможност на потребителите да изследват по-близко до реалността поведение на моделите, като например да работят с еластични колизии, да изследват триенето или пък гравитацията.

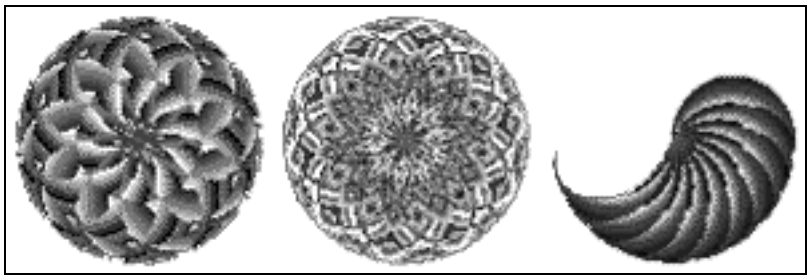
Програмирането в системата набляга на събитията. За отделните обекти могат да се дефинират начини на поведения, които да се следват при настъпване на отделни събития.

I.4.3.4 StarLogo

StarLogo е специализирана версия на езика за програмиране Logo. Системата разширява концепциите на традиционната костенуркова графика с паралелното командване на множество костенурки. В допълнение на това, StarLogo дава възможност програмите да се написват като набор от хиляди по-малки фрагменти, които определят поведението на костенурките и на средата, в която те се движат. Поради това, системата често се използва за създаването на поведенчески системи и за системи с Изкуствен интелект.

В системата са разработени примери, които демонстрират използването й в предметни области като биологията, графиката, математиката, физиката и различни социални системи.

Фигура I-7 Система StarLogo



I.4.3.5 MicroWorlds

Компанията LCSI е разработила системата MicroWorlds за Windows и Macintosh компютри. Системата има възможности за създаване на динамични мултимедийни и Web-базирани проекти. Съществува и по-усъвършенствана версия – MicroWorlds Pro, която са включени редица подобрения в програмната среда, в графичните средства и в обработването на мултимедийни данни.

Фигура I-8 Система MicroWorlds

I.4.3.6 LogoWriter Win

LogoWriter Win [8] е най-последният продукт от дългата редицата продукти на Logo Japan, която е разработила и системата The Logo Robot Control System. Тя работи с програмируеми RCX тухлички на LEGO.

Фигура I-9 Система Logo Writer Win

I.4.3.7 Terrapin Logo

Основната цел при създаването на Terrapin Logo е да се предоставя възможност за използване на традиционните средства: костенуркова графика и езика за програмиране Logo. Системата се предлага от Terrapin Software и включва модерен MacOs/Windows интерфейс. За по-старите компютри могат да се използват някои от предишните версии: PC Logo и Logo PLUS.

I.4.3.8 Mach Turtles Logo

Системата Mach Turtles Logo (Learning Edition) е за Windows и се предлага безплатно за лични или образователни цели. Тя съдържа пълна версия на езика Logo с допълнителни мултимедийни възможности като анимация, звук, поддръжка на MIDI и на пълноцветна графика.

Системата позволява създаването на фонове и на анимационни картинки, които лесно могат да се накарат да се движат.

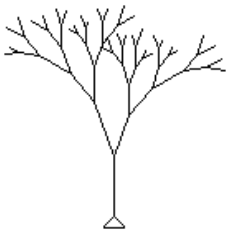
Фигура I-10 Mach Turtles Logo

I.4.3.9 TinyLogo

TinyLogo е език за програмиране и работна среда. Той е удобен за начинаещи потребители и за всички, които искат да направят първоначално запознаване с езика Logo и с програмирането въобще.

Системата е транслатор на езика Logo и поддържа пълна костенуркова графика.

Фигура I-11 Система TinyLogo



Особеното на системата е, че след инсталация програмата може да работи изцяло на Palm компютри, като изисква от тях да имат поне 27k свободна памет. Създадените програми могат да се съхраняват като memo и да се обменят с други притежатели на Palm компютри.

I.4.3.10 ObjectLogo for Macintosh

Системата ObjectLogo for Macintosh [5] е създадена 1986 година и е една от първите реализации на обектноориентираното програмиране в среда на езика Logo.

Обектите в ObjectLogo съдържат методи и полета, могат да се наследяват, създават и премахват. Могат да се създават и интерактивни обекти, които реагират на действията на потребителите.

По начина на третиране на обектите тази система е най-прогресивна, но при нея традиционното Logo е утежнено със синтактични конструкции (вградени команди), с които става възможна работата с обекти. Това утежнение не намалява отличните функционални възможности на ObjectLogo.

I.4.3.11 UCBLogo

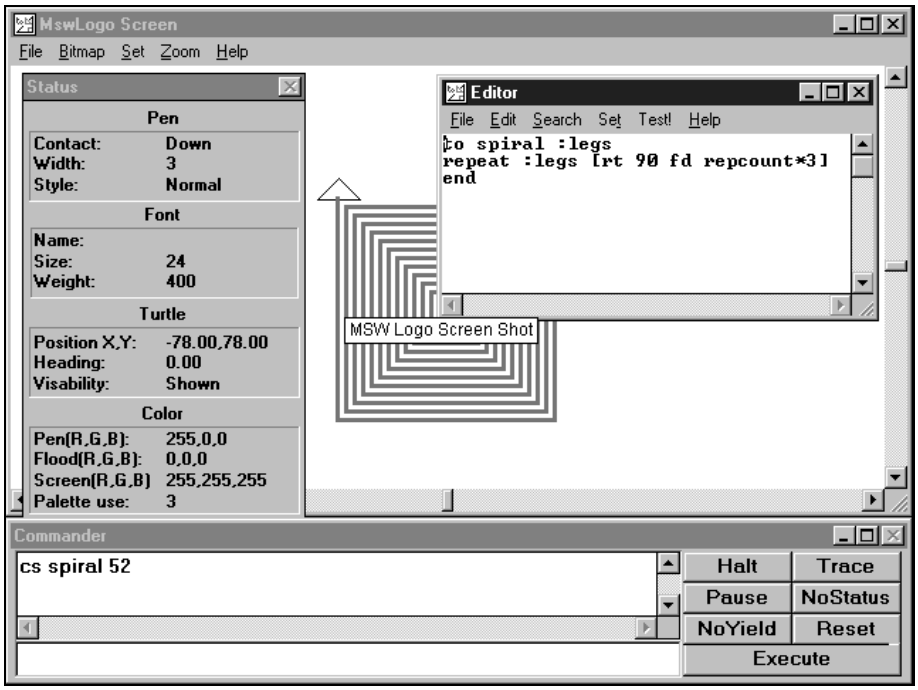
UCBLogo е функционално пълна версия на Logo. Разработена е от Brian Harvey и неговите студенти в University of California в Бъркли. Съществуват няколко версии за различни платформи.

I.4.3.12 WMSLogo

Системата WMSLogo е създадена от George Mills въз основа на ядрото на UCBLogo. Включва мултимедийни и други възможности. Работи под Windows. За системата са

създадени и локализирани версии (френска и немска). Версията на френски се нарича MSWLOGO a Genuve.

Фигура I-12 Система WMSLogo

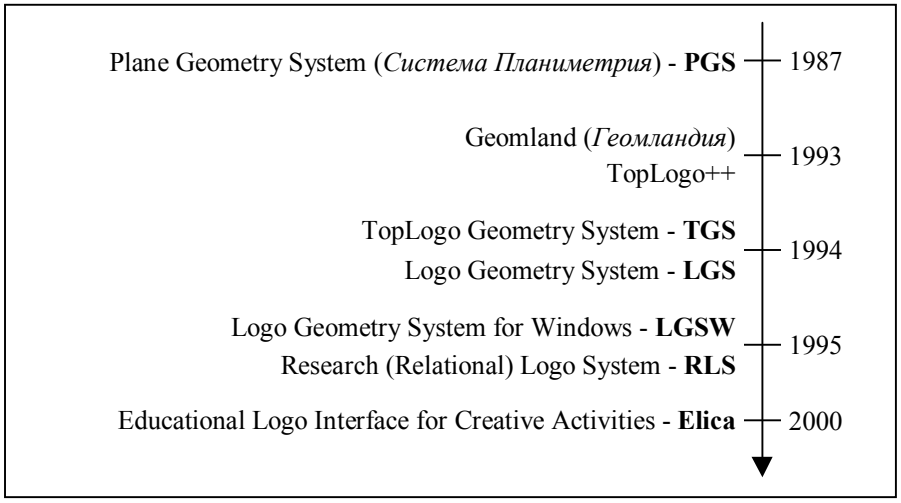


I.5 Система Elica

I.5.1 Кратка история на Elica

Вече бе споменато, че Геомландия (Система Планиметрия) може и трябва да се счита като концептуален предшественик на Elica. Всъщност, Геомландия е следващата версия на Система Планиметрия. В края на 1992 и началото на 1993 бе решено да се проектира и създаде Logo-подобен програмен език, наречен TopLogo++. Приликата с Logo беше главно на синтактично ниво. TopLogo++ бе написан изцяло на 80x86 Assembler. За този език бе написан компилатор и работна среда.

Фигура I-13 История на Elica



Синтактичната прилика с Logo се оказа недостатъчна за реализирането на желаната система, затова, година по-късно бе написана нова среда и интерпретатор за нов език – TopLogo Geometry System (TGS). За базова среда за разработване на TGS бе използван TopLogo++.

Поради проблеми със скоростта, TGS бе пренаписана на Turbo Pascal и съществено променена. Новата версия получи името Logo Geometry System (LGS). В последствие се създаде Logo Geometry System for Windows (LGSW), с което бе поставено началото на графичния интерфейс на системата.

Независимо от положителните качества на LGSW, в края на 1995 г. започна теоретичната разработка на нова концепция, за това как трябва да се реализира подобна система. Първоначалното име, Relational Logo System (RLS), подсказва основната характеристика на модела – релационен. Няколко месеца по-късно моделът беше леко видоизменен, с цел по-лесна реализация. Реалното създаване на интерпретатора на езика и на работната среда, стана в края на същата година. Името на системата бе запазено – RLS, но с по-различно тълкование – Research Logo System.

В продължение на 5 години RLS беше развивана както като концепция, така и като реализация. С всяка нова версия системата се доближаваше все повече до първоначалните изисквания.

За първи път през лятото на 1999 г. системата беше забелязана от чуждестранни специалисти в образованието и след получаване на финансова подкрепа за по-малко от година системата бе съществено подобрена, доразвита и завършена. Новата система получи името Elica (Educational Logo Interface for Creative Activities).

I.5.2 Основни свойства на система Elica

Повечето от свойствата на системата Elica са предварително заложени още в процеса на нейното проектиране, но други са реализирани в процеса на разработване на системата. Основните свойства на системата са:

- **обектна ориентираност**: базовият език на системата е диалект на Logo, в който има възможност да се създават и манипулират обекти;
- **RISC-технология**: базовият език на системата съдържа минималното количество запазени думи, на базата на които се създават процедури и функции, реализиращи "липсващата" функционалност;
- **отвореност**: системата предоставя възможност отделните модули да се създават независимо от системата, като използването им става с динамично свързване;
- **гъвкавост**: едни и същи модели могат да бъдат реализирани по различни начини и потребителят може да избира с каква комбинация от модули ще работи;
- **графична атрактивност**: графичните възможности на системата се доближават до най-новите технологии във визуализирането, с което се създава възможност да се създават фотореалистични модели и симулации;
- **идеологическа уникалност**: начинът, по който се третират различните метаобекти в системата, предлага съществено различна гледна точка, която не се среща в никоя известна система.
- **пълнота**: работата с числа допуска използването на безкрайности и неопределености във всички математически операции както като аргументи, така и като резултати.

I.5.3 Архитектура на системата

Архитектурата на система Elica може да се разглежда в няколко аспекта, но основното е, че системата е изградена на принципа на модулността. Всеки от модулите може да бъде променян, без промяна в останалите модули.

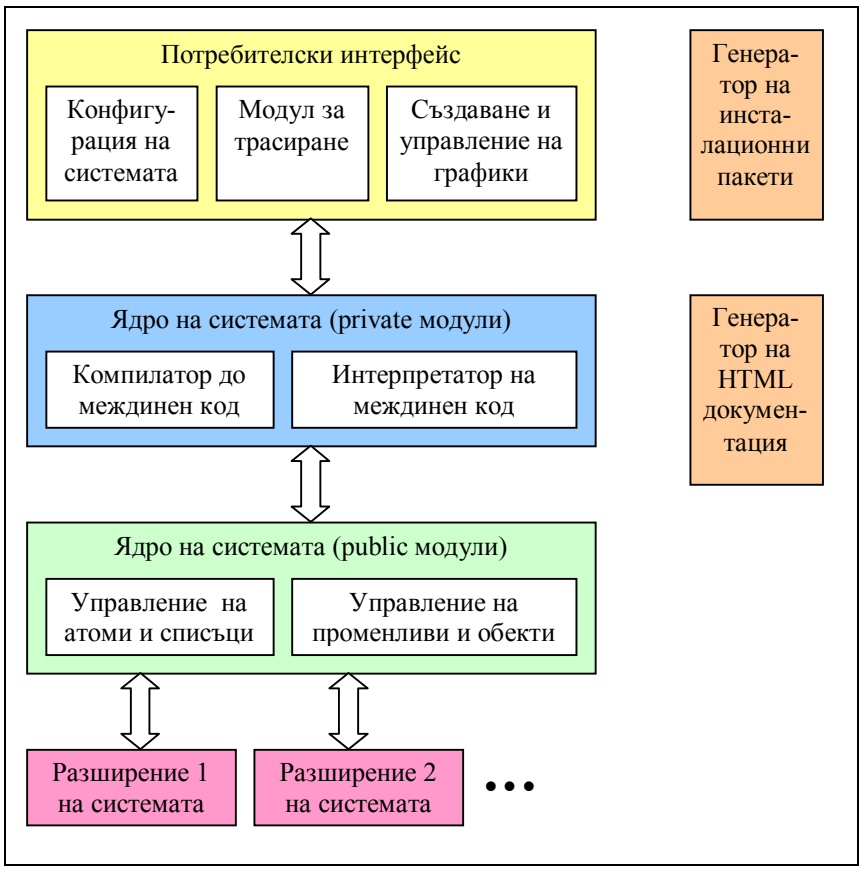
Централният модул на системата е нейното ядро с *private* модули. Това са модули, до които потребителят няма пряк програмен достъп. Тези модули изпълняват основните функции по интерпретацията на потребителските програми:

- Конвертиране на текста на дадена програма до списък;
- Конвертиране на списък до междинния вътрешен код на Elica;
- Интерпретиране на вече създаден междинен вътрешен код.

Функционално тези модули обменят данни с модулите на интерфейса, които също не са програмно достъпни за потребителя. Интерфейсните модули отговарят за дейностите, които се визуализират по един или друг начин върху екрана:

- Конфигуриране на системата, предоставяне на възможности да се работи с няколко програми едновременно, помощна информация и други;
- Трасиране на програма, следене на състоянието на паметта, локалния стек и др.;
- Създаване и управление на графични изображения, предоставяне на обратна интерактивна връзка с потребителя.

Фигура I-14 Архитектура на Elica



Освен private модули ядрото съдържа и модули, които са достъпни за потребителя. Това се налага, когато се прави разширение от ниско ниво към системата. В този случай потребителят ще трябва да може да създава, извлича, обработва и изтрива информация за данните, създадени по време на работа на системата. Public модулите са два вида:

- За базово управление на атоми и списъци;
- За пълноценно управление на променливи, обекти и други метаструктури.

Разширенията към системата могат да бъдат написани на високо ниво, но при нужда, те могат да бъдат написани на по-ниско, спрямо системата, ниво – например на Delphi. В този случай разширенията използват пряк интерфейс към част от ядрото на Elica. Броят и функционалността на разширенията не е ограничен.

В системата съществуват и два независими модула, които се използват по време на разработването. Единият модул комплектова и създава инсталационнен пакет на системата, а другият е генератор на помощна информация в HTML формат, подходяща за автоматично създаване на web страници.

I.5.4 Графичен интерфейс

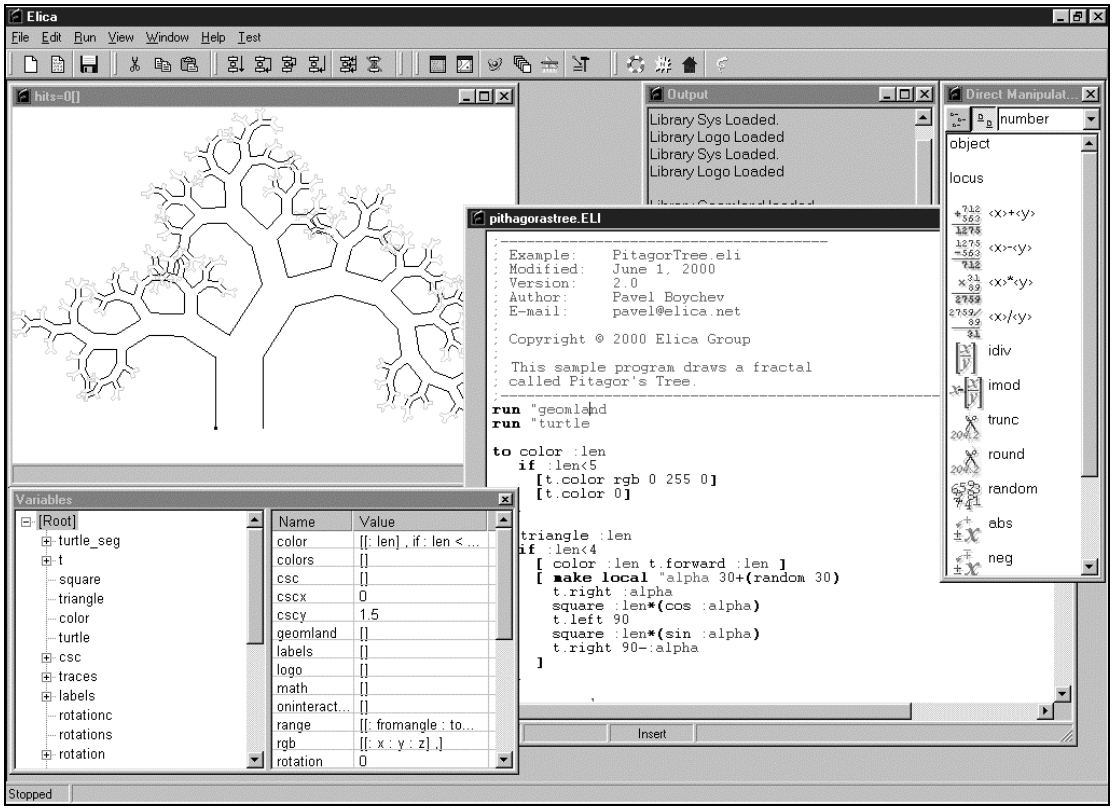
Графичният интерфейс на системата е съобразен със съвременните изисквания за проектиране и създаване на графични интерфейси. За база е използвана WMS (Window Management System) на Windows и стандартните и разширените контроли.

Основните функции, които са достъпни на потребителя на системата, са изнесени в менютата, а паралелно с това са дублирани и като бутони.

Основните дейности в системата се извършват в няколко по вид прозореца:

- Редактори – в тези прозорци потребителят може да разглежда, редактира, изпълнява и трасира програми;
- Графичен – това е прозорец, в който се визуализират образите на графичните обекти и се осъществява интерактивният режим;
- Изход – в този прозорец се извеждат текстовите резултати на изпълняваните програми
- Помощни прозорци – съществува набор от помощни прозорци, с чиято помощ се улеснява работата на потребителя. Информацията в помощните прозорци дава възможност да се проследява състоянието на паметта, стойностите на локалните и глобалните променливи, стекът на изпълнение. Освен това те подсигуряват бърз достъп до най-често използваните езикови конструкции, процедури и функции

Фигура I-15 Работна среда Elica



I.5.5 Elica Logo

Една от основните новости е проектирането на език за програмиране, наречен Elica Logo, който е базов език за системата. Като диалект на Logo, Elica Logo възприема изцяло синтаксиса и идеологията му. Особеността в Elica Logo е наличието на силно намален набор от запазени думи, без това да оказва влияние на функционалността и мощността на езика. Независимо от това на пръв поглед ограничение, езикът дава възможности, които са стандартни за най-новите поколения езици, а дори и такива, които липсват при тях:

- Обекти и поддържане на обектно-ориентирано програмиране. Обектите могат да се дефинират, използват, унищожават, наследяват. Динамично могат да се създават и премахват полета и методи на обектите. Поддържа се полиморфизъм и множествено наследяване;

- Дефиниране на произволни бинарни и унарни оператори, както и оператори с по-висока кратност. Операторите могат да са префиксни, суфиксни и смесени. Асоциативността и приоритетът на всеки от тях се определят от потребителя и могат да се променят;
- Уникална унификация на различните метаелементи на езика. Оператори, процедури, функции, дефиниции на обекти, инстанции на обекти, домейни, променливи и масиви се обработват по един и същ начин и са взаимно заменяеми.

Elica Logo е език, който не е обвързан с никоя конкретна предметна област. Възможностите за разширението му го правят подходящ за реализиране на модели на различни природни, научни или абстрактни процеси и явления.

II ЕЗИКЪТ ЗА ПРОГРАМИРАНЕ ELICA LOGO

II.1 Основи на езика

II.1.1 Типове данни

Традиционните Logo езици поддържат поне трите основни типове данни – *числа*, *думи* и *списъци*. Понякога, те са разделени на две групи – атоми и списъци. Числата и думите са атоми, поради общото мнение, че те представляват най-малките значими частици от данни.

В Elica Logo атоми са всички типове данни, които се поддържат от ядрото на системата. То от своя страна, "има представа" за следните типове данни: числа, думи и списъци, като не поддържа никакви операции с тях с изключение на конвертирането им от и до текст.

Езикът дава възможност да се дефинират сложни обекти, но те не могат да се обработват пряко от ядрото на системата, а от допълнителните библиотеки и разширения на езика.

II.1.1.1 Числа

Числата в Elica Logo са един от основните типове данни. Единственото, което ядрото на системата може да прави с числа, е да "прочете" число от текста на програмата и да изведе число на екрана. Математическите операции са реализирани извън ядрото на системата. В Elica Logo не се прави разлика между цели и реални числа. За системата всички числа са реални, но тези, които могат да се представят като цели се представят пред потребителя като такива. В следващата таблица са показани няколко примера на правилно записани числа. В Elica Logo не е предвидена възможност за четене на числа в двоична, осмична или шестнадесетична бройна система.

Пример II-1 Примери за числа

Команди	Резултати
`print 5`	`5`
`print -9.3`	`-9.3`
`print 1253.5`	`1253.5`
`print -1.187E3`	`-118.7`

II.1.1.2 Думи

Думите са редици от знаци, съдържащи само видими знаци, но в някои се налага да се ползват и такива с по-особено значение. В тези случаи, за да се укаже на транслатора да не интерпретира тези знаци като специални символи, цялата дума се загражда с апострофи. Ако трябва да се използва апостроф вътре в дума, тя трябва да се дублира.

Пример II-2 Примери за думи

Команди	Резултати
`print "hello`	`HELLO`
`print "'hello'`	`Hello`
`print "'long word'`	`long word`
`print [list of words]`	`[LIST OF WORDS]`

Когато дума се ограничава от апострофи, знаците в нея се запазват същите, в противен случай се конвертират автоматично до главни.

Има две специални думи: `"True` и `"False`. Те се използват за обозначение на булевите стойности *истина* и *лъжа*. Условни команди като `WHILE` и `IF` очакват резултата от условията да е булева стойност.

Пример II-3 Примери за логически думи

Команди	Резултати
`print "true and "false`	`FALSE`
`print "true or "false`	`TRUE`

`if equal? 5 4`
`  [print "equal']`
`  [print "'not equal']`
	`not equal`

II.1.1.3 Списъци

Списъците в Elica Logo са структури от данни, които съдържат линейно свързани елементи. Всеки елемент от списък може да е число, дума или друг списък. Списъците имат жизненоважна роля в системата, понеже изпълнимата програма се представя като списък. Елементите от даден списък са разделени с интервал или друг разделител. Началото на списъка се задава с [, а краят с]. Празният списък се обозначава с [ ]. Допуска се неограничено влагане на списъци.

Пример II-4 Примери за списъци

Команди	Резултати
`print [a b c]`	`[A B C]`
`print [ ]`	`[ ]`
`print se [b] [d [f]]`	`[B D [F]]`
`print [f g] = [ ]`	`FALSE`

Съществува формална дефиниция на списък, представена на следващата фигура.

Фигура II-1 Дефиниция на списък

1. [ ] е списък (празен списък)
2. Ако X_1 е число, дума или списък,
   то $[X_1]$ е списък.
3. Ако X_N е число, дума или списък, а $[X_1\ X_2\ ...\ X_{N-1}]$ е списък,
   то $[X_1\ X_2\ ...\ X_{N-1}\ X_N]$ е също списък.

II.1.1.4 Съвместимост на типовете данни

Както и други реализации на Logo, Elica Logo е слабо типизиран програмен език. Едно от последствията на това е, че всяка променлива може да съдържа стойности от произволни типове. Това се отнася и за параметрите на подпрограмите, затова, всички дефиниции на процедури, функции и оператори не дефинират типа данни, с които работят.

В някои случаи, когато очакваният тип се различава от реалния, системата може да приложи автоматично преобразуване. Това се прилага най-често при използване на думи и числа. При необходимост, всяко число се конвертира до дума, а всяка дума до число (ако това е възможно).

Пример II-5 Съвместимост на типове

Команди	Резултати
`print "5 + bf "a12`	`17`
`repeat bf "b4` `  [print "ready]`	`READY` `READY` `READY` `READY`
`print (word "pi "= pi)`	`PI=3.14159`

II.1.2 Запазени символи

II.1.2.1 Кавички "

Символът *кавички "* се използва, за да обозначи, че следващата лексема трябва да се третира като дума, а не като име на променлива или подпрограма.

Пример II-6 Кавички "

Команди	Резултати
`make "w "rose`	
`make "l [a word]`	
`print :w :l`	`ROSE [A WORD]`
`print "word`	`WORD`
`print :word`	`[: X : Y OUTPUT LOGO.25]`

Забележка: Последната команда `PRINT` в по-горния пример, отпечатва дефиницията на функцията `word`, дефинирана в библиотеката Logo.

II.1.2.2 Двоеточие :

Символът *двоеточие :* се използва, за да обозначи, че дадена лексема трябва да се третира като име на променлива. Ако това е има на процедура или функция, двоеточието обозначава дефиницията на процедурата или функцията..

Пример II-7 Двоеточие :

Команди	Резултати
`make "a 5`	
`print "var "a "is :a`	`VAR A IS 5`
`to pi2`	
`  output mul pi pi`	
`end`	
`print pi2`	`3.1415926536`
`print :pi2`	`[OUTPUT LOGO.9]`

Независимо от това, че двоеточието е синтактичен елемент (използва се по време на фазата на синтактичния анализ), то може да се използва и като функция. В този случай, аргументът трябва да е израз в кръгли скоби.

Пример II-8 Двоеточието като функция

Команди	Резултати
`make "ab 5`	`5`
`print :(word "a "b)`	

Използването на двоеточието като функция е идеално за реализиране на указатели в Logo-стил. Естествено е, че Elica Logo не поддържа указатели в конвенционалния смисъл на този термин (т.е. указатели към конкретни адреси в паметта).

Понеже системата адресира всички променливи чрез тяхното име, а не чрез адреса, където са разположени в паметта на компютъра, то указателите в Elica съдържат името на "указаната" променлива. Следващият пример показва как се създават и използват указатели.

Пример II-9 Указатели

Команди	Резултати
`make "x [recursive]`	
`make "y "x`	
`make "z "y`	
`print :z :(:z) :(:(:z))`	`Y X [RECURSIVE]`

II.1.2.3 Запетая ,

Езикът за програмиране Logo, в нито един от своите диалекти, не използва някакъв символ за отделяне на последователните команди (както например в Pascal се използва ;). Вместо това, езикът се осланя на запазените символи като кръгли скоби, квадратни скоби, а също и на символите за нов ред, за да определи кои лексеми в кои команди участват.

В Elica Logo има два начина за изписване на няколко команди на един ред, за които нито един от "обичайните" терминиращи символи не е приложим. Единият начин е когато командата изчерпи полагащите й се аргументи. Останалите принадлежат към следващи команда. Вторият начин е да се използва *запетая* ,. В следващия пример запетаята указва, че втората команда `PRINT` е само с един аргумент, а `LOCAL?` е отделна команда.

Пример II-10 Запетая ,

Команди	Резултати
`print "a local? "b`	`A FALSE`
`print "a, local? "b`	`A`

Особено важно е да се спомене, че има случаи, в които запетаите се игнорират от лексическия анализатор. Правилото е следното: ако запетая или символ за нов ред (CR или LF) са в част от текста на програма, разположен пряко в кръгли скоби, то те се игнорират. Ако са разположени пряко в квадратни скоби или не са под влиянието на никакви скоби – те не се игнорират.

Причината за това е, че в Elica Logo всяка функция, процедура, оператор или команда могат да се изпълнят с повече или с по-малко аргументи от дефинираните. В този случай, когато аргументите са много и е неудобно да се изписват на един ред, те се поставят на последователни редове, а командата се загражда с кръгли скоби. По този начин новите редове и запетаите се игнорират.

II.1.2.4 Фигурни скоби {…}

С цел улесняване четенето на програми, е необходимо на определени места да се поставят обяснения (коментари). В Elica Logo това се прави като текстът, който не съдържа команди и следователно не трябва да се транслира, се загради с *фигурни скоби {...}*. По този начин може да се обозначи и част от програмата, която не трябва да се изпълнява.

Пример II-11 Фигурни скоби

Команди	Резултати
print 1 {skip this text} print 2 {multiline and {nested} comments} print 3	1 2 3

Всички символи, включително и интервалите, табулациите, символите за нов ред, вътре в {…} се игнорират, затова този начин е подходящ за коментиране на големи сегменти от текста на програмата. Коментарите могат да се влагат един в друг и единствените символи, които се "интерпретират" в рамките на коментар са {…}.

II.1.2.5 Точка и запетая ;

Използването на фигурни скоби за коментиране на част от текст на програма не е винаги удобно. В случаи, когато се налага бързо да се коментират единични редове, се използва *точка и запетая ;* . С този символ се коментира текста до края на реда.

Пример II-12 Точка и запетая ;

Команди	Резултати
print 1 ;skip this text print 2 ;and this one	1 2

II.1.2.6 Апострофи '…'

За повече удобство Elica транслаторът преобразува всички лексеми в главни букви. Това не дава възможност да се извежда по лесен начин текст с главни и малки букви. За да се преодолее този проблем, могат да се използват апострофи.

Пример II-13 Апострофи '…'

Команди	Резултати
print "Test print "'Test' make "a 5 make "'a' 10 print :a :'a' :A :'A'	TEST Test 5 10 5 5

Апострофите се използват, за да се комбинират няколко думи в една. Това е особено важно, когато думата съдържа "нелегални" букви като интервал. По този начин може да се дефинира променлива с име, съдържащо интервал.

Пример II-14 Апострофи и интервали

Команди	Резултати
print "Ah "some "words print "'Ah some words' make "' ' [a b] print :' '	AH SOME WORDS Ah some words [A B]

II.1.2.7 Кръгли скоби (…)

Logo е език за програмиране, който използва процедури, функции и оператори с нефиксиран брой параметри, независимо че всички те имат подразбиращ се брой параметри. Например, може да се направи процедура `sum`, която е с два аргумента и изчислява сумата на две числа. Възможно е `sum` да се напише така, че да поема произволен брой аргументи. За да се укаже, че `sum` се използва с повече аргументи, целия израз се загражда с кръгли скоби.

Пример II-15 Кръгли скоби

Команди	Резултати
`print sum 5 7`	`12`
`print sum 3 5 7 9`	`8 7 9`
`print (sum 3 5 7 9)`	`24`

Важна особеност на кръглите скоби е, че всички запетаи и символи за нов ред в тях, ако не са вложени в квадратни скоби, се игнорират. Това дава възможност да се пише команда на няколко реда.

Пример II-16 Кръгли скоби и нови редове

Команди	Резултати

`make "myvar "myvalue`
`(print`
`:myvar)`
	`MYVALUE`

II.1.2.8 Квадратни скоби […]

Квадратните скоби са едни от най-важните и най-използваните специални символи. Те изпълняват съществена роля в структурирането на текста на Logo програми. Основната им употреба е да маркират началото и края на списък при дефинирането на елементите от него.

Пример II-17 Квадратни скоби

Команди	Резултати
`print [a b c]`	`[A B C]`
`print [ ]`	`[ ]`
`print [a 6 [f+g [9 8][]]]`	`[A 6 [F+G [9 8][]]]`

Понеже текстът на Logo програма е списък, то и самите програми изглеждат като списък, чийто елементи са командите в програмата. Например, когато команда очаква списък от команди, те са наистина списък.

Пример II-18 Квадратни скоби и команди

Команди	Резултати
`if :a = 5` `[ print "ok ]`	`OK`

`repeat 5`
`[ make "a :a+1`
`print :a`
`]`
	`6`
	`7`
	`8`
	`9`
	`10`

Дефинирането на процедури, функции и обекти се прави с команди, но самите дефиниции не изглеждат като списък. Това се налага от традиционния за Logo начин за дефиниране на подпрограми. Независимо от това, Elica Logo автоматично конвертира подобни дефиниции в аналогични, но с използването на списък.

Пример II-19 Квадратни скоби и подпрограми

Команди	Резултати

`to proc :x`
`  to func :y`
`  output :y`
`  end`
`  print func :x`
`end`	

`proc "abc`	`ABC`
`print :proc`	`[[: X], MAKE "FUNC [[: Y] , OUTPUT : Y ,]` `, PRINT FUNC : X ,]`

II.1.3 Запазени думи

В Elica Logo има само девет запазени думи и две псевдозапазени думи, означаващи команди. Всички останали команди, които са запазени думи в другите реализации на езика Logo, са реализирани в библиотеките на системата. Независимо, че броят на запазените думи е силно ограничен, част от тях също могат да бъдат реализирани във външни библиотеки. Запазените думи в Elica Logo са: `PRINT`, `MAKE`, `IF`, `REPEAT`, `WHILE`, `LOCAL`, `OUTPUT` и `RUN`. Псевдозапазените думи са `TO` и `END`, които по време на лексическата обработка се конвертират до еквивалентни конструкции с `MAKE` и `LOCAL`.

II.1.3.1 Команда `PRINT`

Общият вид на командата е следният:

```
print
print «value»
print «value» «value» ...
```

където «value» е произволен израз.

Командата `PRINT` се използва, за да се отпечатат една или повече стойности на екрана. Всяка `PRINT` команда отпечатва стойност на един ред. Ако се използва без аргументи, командата оставя празен ред.

Пример II-20 Команда PRINT

Команди	Резултати
`print "a [b] 5`	`A [B] 5`
`print`	
`print [no more]`	`[NO MORE]`

II.1.3.2 Команда IF

Общият вид на командата е следния:

```
if «cond» «then»
if «cond» «then» «else»
```

където «cond» е израз, «then» и «else» са изрази или константи от тип списък.

Командата IF се използва когато трябва да се изпълнят команди само при удовлетворяване или неудовлетворяване на определено условие. Командата има две форми: кратка и пълна. Кратката форма се използва, когато не се указва какви команди трябва да се изпълняват, когато условието не е удовлетворено.

Пример II-21 Команда IF

Команди	Резултати
`make "a 5`	
`if :a=5 [print "yes]`	YES
`if :a=6 [print "ok]`	
`if :a=7`	
`  [print "equal]`	
`  [print "not]`	NOT

Ако условието не е нито TRUE, нито FALSE, приема се, че е FALSE.

Командата IF може да се изпише на няколко реда, без да се налага заграждането й с кръгли скоби. Ако «then» е константа от тип списък, тя може да се разположи на същия ред, веднага след условието, но може да се постави и на следващия ред. Аналогично правило важи и за «else». По този начин, командата IF може да се напише по шест различни начина

```
if «cond» [ ]
if «cond»
   [ ]
if «cond» [ ] [ ]
if «cond» [ ]
   [ ]
if «cond»
   [ ] [ ]
if «cond»
   [ ]
   [ ]
```

II.1.3.3 Команда REPEAT

Общият вид на командата е следния:

```
repeat «count» «commands»
```

където «count» е числов израз, а «commands» е израз или константа от тип списък.

Командата REPEAT се използва, за да се повтори изпълнението на фрагмент от кода, определен брой пъти. Броят повторения се задава в «count» и трябва да е цяло положително число. Ако то е нула или е отрицателно, командите в «commands» не се изпълняват. Ако е положително и реално, използва се само цялата му част.

Пример II-22 Команда REPEAT

Команди	Резултати
`make "a 0`	
`repeat 10`	
`  [make "i sum :i 1]`	
`print :i`	10

Ако «commands» е константа от тип списък, тя може да се разположи и на следващия ред, без да се налага използването на кръгли скоби. По този начин, командата REPEAT може да се запише по два начина:

```
repeat «count» [ ]
repeat «count»
  [ ]
```

II.1.3.4 Команда WHILE

Общият вид на командата е следния:

```
while «cond» «commands»
```

където «cond» е булев израз, «commands» е израз или списък от команди.

Командата `WHILE` се използва, за да се изпълни фрагмент от кода определен от «cond» брой пъти. В общия случай броят повторения не се знае отначало. Преди всяко изпълнение на «commands» се преизчислява стойността на «cond». Ако тя е TRUE, командите се изпълняват и отново се преизчислява условието. Този цикъл продължава до момента, в който стойността на «cond» стане различна от TRUE.

Пример II-23 Команда WHILE

Команди	Резултати
`make "i 10`	
`while :i>0`	
`  [ make "i :i/2-1, print :i ]`	`4`
	`1`
	`-0.5`
`print :I`	`-0.5`

Ако «commands» е константа от тип списък, тя може да се разположи и на следващия ред, без да се налага използването на кръгли скоби. По този начин, командата `WHILE` може да се запише по два начина:

```
while «cond» [ ]
while «cond»
  [ ]
```

II.1.3.5 Команда LOCAL

Общият вид на командата е следния:

```
local "«name»
local "«name» "«name» ...
```

където «name» е име на идентификатор.

Командата `LOCAL` се използва, за да се дефинира променлива, процедура или функция като локална за текущо изпълнявания списък от команди. След приключване на изпълнението на командите от списъка, всички локални дефиниции се премахват и стават недостъпни. Командата може да се използва и в комбинация с `MAKE` за постигане на по-къс и елегантен код.

Пример II-24 Команда LOCAL

Команди	Резултати
`to semisum :x :y`	
`   local "m, make "m (:x+:y) / 2`	
`   make local "n :m`	
`   print :n`	
`end`	
`semisum 5 12`	`8.5`

II.1.3.6 Команда OUTPUT

Общият вид на командата е следния:

```
output
output «expr»
```

където «expr» е израз.

Командата OUTPUT прекратява изпълнението на текущата подпрограма и предава управлението на мястото, от където първоначално е била извикана. Ако се използва втората форма на командата, стойността на израза се връща на извикващата команда. Използва се реализиране на функциите в Elica Logo.

Системата реагира по различни начини според начина на извикване на подпрограма и начина на завършването й. Поради сложността, това е предмет на отделна глава,

Пример II-25 Команда OUTPUT

Команди	Резултати

`to test :x`
`  output less? :x 0`
`end`
 	
`if not test 5`
`  [print "less]`
	LESS

II.1.3.7 Команда RUN

Общият вид на командата е следния:

```
run «commands»
run «libraryname»
```

където «commands» е израз със стойност списък, а «libraryname» е израз със стойност дума.

Командата RUN се използва в два основни случая. Първият е, когато трябва да се изпълнят команди, записани в списък, а вторият – изпълнение на команди, записани в библиотека.

Типът на израза на RUN определя, кой от двата варианта се осъществява. Ако е списък, той се изпълнява направо, а ако е дума, тя се третира като името на файла, съдържащ библиотеката.

Пример II-26 Команда RUN

Команди	Резултати
`make "a [print "boza]`	
`run :a`	BOZA
`run "ex_local`	8.5

II.1.3.8 Команда TO..END

Общият вид на командата е следния:

```
to «arg» «arg»... «name» «arg» «arg»...
   «commands»
end
```

където «name» е име на процедура, функция, оператор или обект, «arg» са аргументите (било то отляво или отдясно на името), а «commands» е списъка от командите асоциирани със създаваната подпрограма.

Аргументите са идентификатори, предхождани от един от двата символа: двоеточие или кавички. Аргумент дефиниран с двоеточие се предава по стойност, а такъв с кавички – по име.

Пример II-27 Команда TO..END

Команди	Резултати

`to :x !`
`  if :x>1`
`  [output :x*((:x-1)!)]`
`  [output 1]`
`end`	

`print 5 !`	`120`
`print 500 !`	`1.2201E1134`

Ако подпрограма се дефинира в рамките на друга подпрограма, първата става локална за втората. Това се дължи на факта, че командата `TO..END` се превежда до `MAKE` по следния начин:

```
make local "«name» [[«arg»...] [«arg»...] «commands»]
```

II.1.3.9 Команда MAKE

Общият вид на командата е следния:

```
make «name» «value»
make «name»
```

където «name» е израз, обозначаващ име на променлива, а «value» е произволен израз.

Командата `MAKE` се използва, за да се създаде глобална променлива и/или да се промени стойността на глобална или локална променлива. Ако се използва късият вариант на командата, само се създава променлива, без да й се присвоява стойност.

Най-често командата се използва с два аргумента, но има и разширен синтаксис, при който тя може да се използва с три и с четири.

Пример II-28 Команда MAKE

Команди	Резултати
`make "a 5`	
`make word "i :a sin :a`	
`print :i5`	`0.087156`

II.1.3.10 Команда OB

Общият вид на командата е следния:

```
ob «name» «value»
```

където «name» е израз, обозначаващ име на променлива, а «value» е произволен израз.

Командата `OB` се използва, за да се създаде глобална променлива и/или да се промени стойността на глобална или локална променлива. Командите `OB` и `MAKE` са идентични, като единствената (и съществена) разлика е, че при командата `OB` автоматично се създават връзки от използваните променливи към променливата с име «`name`».

Най-често командата се използва с два аргумента, но има и разширен синтаксис, при който тя може да се използва с три и с четири.

II.1.4 Програми и библиотеки

Програмите на Elica Logo са обикновени текстови файлове без фиксирана структура. За удобство командите се изписват по една на ред, като броят на разделителите не е съществен.

Всяка програма на Elica Logo е идентична по структура с тялото на подпрограмите, т.е. за потребителите няма разлика в начина на задаване на командите от подпрограма и на програма.

Библиотеките в Elica Logo са също програми, които могат да се използват и като библиотеки и като самостоятелни приложения. Когато програма иска да използва библиотека, тя се стартира. Ако в библиотеката се дефинират подпрограми, те ще бъдат достъпни и от главната програма.

Библиотеките в Elica Logo са два вида – вътрешни и външни. Вътрешните библиотеки са реализирани изцяло на Elica Logo. Тези библиотеки се използват с командата `RUN`. Допустимо е библиотека да зарежда други библиотеки, а също и библиотеки да се използват взаимно.

Вътрешните библиотеки се характеризират с това, че лесно се разглеждат от потребителите. Те от своя страна могат да ги модифицират без използването на допълнителни средства.

Външните библиотеки са тези, които не са написани на Elica Logo, а на друг език (например Delphi). Тези библиотеки трябва да са предварително компилирани до DLL файлове, а експортираните функции да са в строго определен формат. Външните библиотеки могат да се използват директно от програма на Elica Logo, но също така и от други библиотеки. Добра практика е за всяка външна библиотека да се създаде по една или няколко вътрешни, които да капсуловат външните подпрограми и им задават удобни имена и параметри. Връзката на системата с външните библиотеки се осъществява чрез използването на специален обект, наречен `DLL`. Този обект съдържа имената на външните библиотеки под формата на имена на полета в обекта. Всяко от полетата съдържа като подполета имената на функциите от външната библиотека.

Особеност на обекта `DLL` е, че това е виртуален обект – в реалност той не съществува. Изпълняването на подпрограмата `DLL.LOGO.DELETE` (това означава, че се изпълнява метода `DELETE` от обекта `LOGO` от обекта `DLL`) извършва следните дейности:

- намира файл с име `LOGO.DLL;`
- намира в него подпрограма с име `DELETE;`
- изпълнява я.

Подпрограмите от външните библиотеки (от гледна точка на потребителя) се извикват без никакви параметри. Всички данни, които трябва да се предадат на външна подпрограма се зареждат в локални променливи. Външната процедура "знае" от кои локални променливи трябва да зареди параметрите си. В следващия пример се показва как може да се дефинира операторът + като се използва функцията `logosum` от файла `logo.eli`.

Пример II-29 Използване на външни подпрограми

Команди

```
to :x '+' :y
   output dll.logo.logosum
end
```

II.2 Разширен синтаксис на командата MAKE

Стандартната форма на командата MAKE е съвместима със структурата й при всички останали реализации на Logo. Особеностите на Elica Logo изискват синтаксисът на командата да бъде разширен, като при това се запазва пълна съвместимост.

II.2.1 Разширени команди MAKE и OB

Общият вид на разширените команди е съответно:

```
make «predcond» «name» «value» «postcond»
```

и

```
ob «predcond» «name» «value» «postcond»
```

където «name» е израз, обозначаващ име на променлива, «value» е произволен израз, а «predcond» и «postcond» са списъци с имена на допълнителни процедури.

Допълнителните процедури трябва да са дефинирани с един аргумент. Ролята, която изпълняват тези процедури се вижда от следния пример:

Пример II-30 Разширена команда MAKE

Команди	Еквивалентни команди

`make [list dump] "x 5 [denote]`	`list "x`
	`dump "x`
	`make "x 5`
	`denote "x`

От примера ясно се вижда, че двата допълнителни аргумента се използват, предимно за улеснение – дават възможност на потребителя да изпълнява избрани от него процедури веднага преди и след присвояването на нова стойност на променлива.

II.2.2 Събития OnBeforeMake и OnAfterMake

Практическото използване на разширения вид на командата MAKE може да създаде неудобства, понеже се оказва, че голяма част от MAKE командите имат едни и същи разширения.

В такива случаи могат да се използват събитията OnBeforeMake и OnAfterMake. Формално, това са променливи, които имат за стойност списък от имена на процедури. Всеки път когато се изпълнява командата MAKE системата проверява стойностите на OnBeforeMake и OnAfterMake и разширява командата.

Пример II-31 Събития OnBeforeMake и OnAfterMake

Команди	Еквивалентни команди
`make "x 5 [dump]`	`make "onaftermake [dump]`
`make "done :x=5 [dump]`	`make "x 5`
`make :name func :x [dump]`	`make "done :x=5`
	`make :name func :x`

II.3 Събития

Система Elica може да реагира на определени събития – както такива, породени от действията на потребителя, така и такива, породени от изпълнението на програма. Elica разпознава 12 различни събития – част от тях са валидни само в рамките на определен обект (за който са създадени), а други – са глобални и важат за цялата програма.

Събитията се представят като стандартни променливи и/или процедури и техните имена не са част от запазените идентификатори в системата.

II.3.1 Събития OnChange и OnPlan

OnChange е най-сложното и най-използвано събитие, затова е описано в отделно. Същността му е да се изпълняват определени от потребителя команди при промяната на стойността на променливите.

Общият вид на събитието е следния:

```
make "«name».onchange «commands»
```

или:

```
make "«name».onchange.«name» «commands»
```

Вторият вид на събитието се използва при автоматично генерираните с командата `OB` обекти.

Когато възникне нужда от преизчисляване на свързани обекти, системата автоматично активира събитието `OnPlan`, което съдържа реда на `OnChange` командите, които трябва да се изпълнят, за да се актуализират обектите в необходимия ред.

II.3.2 Събитие OnDrawImage

Това събитие е най-често срещаното в системата. То се активира, когато се промени стойността на обект, който има изображение. След промяната, изображението на обекта трябва да се преизчисли и да се визуализира на екрана.

Elica показва на екрана само глобалните обекти, в които има дефиниран метод `OnDrawImage`. При нужда от прерисуване, системата изпълнява този метод.

Общият вид на събитието е следния:

```
to «name»
   ...
   to ondrawimage
      «commands»
   end
end
```

или

```
make "«name».ondrawimage «commands»
```

Първият начин на записване е когато се дефинира обект и начина на визуализирането му се вгражда в самата дефиниция. Вторият начин на записване е когато трябва да се дефинира или модифицира начинът на рисуване на вече създаден обект.

В следващия пример се показва как се дефинира обекта `point`, в който е описано как се рисува точка на екрана. Самото рисуване се реализира в процедурата `drawpoint` от външната библиотека `graphix.dll`.

Пример II-32 Събитие OnDrawImage

Команди

`to point :x :y`
`  to ondrawimage`
`  output dll.graphix.drawpoint`
`  end`
`end`

Стойността, която се връща от `drawpoint` се използва автоматично от системата, за да нарисува обекта. Стойността не е стандартен тип данни и не може да се използва пряко от потребителя. За по-сложните обекти (тези, които съдържат други обекти) се налага да се дефинират по-сложни събития за рисуване. Но за повечето случаи може да се използва методът на автоматично създаване на изображения. Той се прилага, когато изображението на главния обект се състои от изображенията на подобектите му.

Пример II-33 Автоматично генерирано събитие OnDrawImage

Команди

`to points :p1 :p2`
`  to ondrawimage end`
`end`

Ако тялото на `OnDrawImage` събитието е празно, при рисуване на обекта системата автоматично обхожда всички подчинени полета и композира всичките техни изображения в едно.

II.3.3 Събития OnPriority и OnLeftAssociation

Общият вид на събитията е следния:

```
make "«name».onpriority «number»
make "«name».onleftassociation «boolean»
```

Поради унификацията на метаобектите, която е описана в отделна глава, всички функции, процедури и оператори могат да бъдат дефинирани с допълнителни параметри, които определят поведението им в изрази. Тези параметри са приоритета и асоциативността. Приоритетът е цяло положително число, а асоциативността е логическа стойност `TRUE` (лява) или `FALSE` (дясна). Събитията OnPriority и OnLeftAssociation възникват когато системата компилира изрази и изчислява в какъв ред да се извършват пресмятанията. Нито едно от събитията не е задължително.

Пример II-34 Събития OnPriority и OnleftAssociation

Команди

`make "'+.onpriority  40`
`make "'*.onpriority  45`
`make "'-.onpriority  40`
`make "'/.onpriority  45`
`make "'^.onpriority  48`
`make "idiv.onpriority  30`
`make "imod.onpriority  30`
`make "not.onpriority  17`
`make "and.onpriority  16`
`make "or.onpriority  15`
`make "not.onleftassociation "false`

Пример II- показва как са зададени приоритетът и асоциативността на някои от операторите в библиотеката Logo. Операторите, за които не е зададен приоритет, получават приоритет по подразбиране 20. При пропускане на асоциативността, тя се приема за `TRUE`.

II.3.4 Събитие OnInteractive

Събитието `OnInteractive` възниква когато потребителят иска да използва възможностите за улеснено създаване на команди. Променливата `OnInteractive` съдържа списък в определен формат.

Общият вид на събитието е следния:

```
make "«name».oninteractive.«word»
     [«function» «image» «syntax»
      «function» «image» «syntax»
      ...]
```

където «word» е името на група от функции, показано в прозореца Direct Manipulation, «function» е име на подпрограма, «image» е или празна дума или име на BMP файл, записан в поддиректорията Images на главната директория на Elica, а «syntax» е дума, която представя синтактичен шаблон на подпрограмата, заедно с параметрите й.

Пример II-35 Събитие OnInteractive

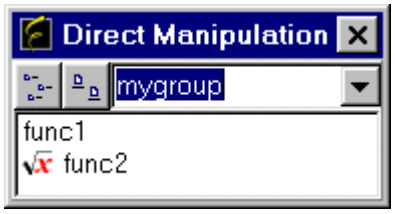
Команди

```
to func1 :x
   output sqrt abs (:x * 3)
end

to func2 :x
   output sqrt abs (:x * 5)
end

make "oninteractive.mygroup
 [ func1 ''      'func1 <number>'
   func2 dm_sqrt 'func2 <number>']
```

Резултати



II.3.5 Събития OnBeforeMake и OnAfterMake

Събитията `OnBeforeMake` и `OnAfterMake` се генерират точно преди и след присвояването на стойност на променлива с командата `MAKE`. Подробности са описани в II.2.2.

II.3.6 Събития OnMouseDown, OnMouseUp и OnMouseMove

Общият вид на събитията е следния:

```
to onmousedown :«x» :«y»
   ...
end
to onmouseup :«x» :«y»
   ...
end
to onmousemove :«x» :«y»
   ...
end
```

където аргументите `x` и `y` са логическите координати, където е станало съответното събитие. `OnMouseDown` се активира когато потребителят натисне левия или десния бутон на мишката в рамките на прозореца Plane. `OnMouseUp` възниква когато бутон на мишката се отпусне. При движение на мишката в прозореца Plane, системата генерира `OnMouseMove` събития.

Пример II-36 Събития OnMouseDown, OnMouseUp и OnMouseMove

Команди	Резултати

`to onmousemove :x :y`
`  make [trace] "p point :x :y`
`end`
	

II.3.7 Събития OnHistoryCreate и OnHistoryUpdate

Събитията `OnHistoryCreate` и `OnHistoryUpdate` се използват за създаване и актуализиране на историята на дадена променлива (това включва поддържането на списък от всички стойности). Двете събития се извикват автоматично от системата при промяна на която и да е променлива, за която е указано да се съхранява история.

И двете събития са реализирани в библиотека на Elica и потребителят може да промени поведението им. По подразбиране, историята на всички променливи, за които се иска да се пази история, се съхраняват в глобалния обект `History`.

II.4 Правила

Правилата в система Elica са списъци от команди, които са прикрепени към определени променливи и определят поведението на системата при промяна на тези променливи и активиране на OnChange събитията над тях.

Поддържат се два вида правила – прости и поименни. Простите правила се отнасят към конкретна променлива като адресантът на промяната е анонимен. Поименните правила са тези, при които адресантът на промяната е известен.

II.4.1 Прости правила

Най-простият начин да се използват правилата, е да се дефинира какво да стане, като се промени определена променлива.

Пример II-37 Прости правила

Команди	Резултати
`make "a 5`	
`make "a.onchange [print "'a=' :a]`	
`make "a 6`	`a= 6`
`make "a "boza`	`a= boza`
`make "a [this is a test]`	`a= [this is a test]`

В горния пример е дефинирано правилото "Ако се промени променливата `a`, да се отпечата текстът `a=` следван от новата стойност на `a`".

Ако разгледаме подробно примера, ще стане ясно, че правилото се записва като подчинено поле на променливата `a`, без значение дали тя съдържа обект или е обикновена променлива с атомарна стойност. Представянето на правилата по този начин улеснява потребителя, като той има възможността да променя създадените вече правила.

II.4.2 Свойства на простите правила

Основните свойства на простите правила са *адитивност* и *самозащита*.

Адитивността при правилата означава, че полетата `OnChange` имат по-различно поведение спрямо останалите променливи. Когато за първи път се присвои стойност на `OnChange` поведението е нормално, но когато се присвои стойност за втори път, новата стойност се обединява с предишната. Формално погледнато, ако променлива има някакви команди в `OnChange` полето си и се зададат нови команди, те ще се добавят към вече съществуващите. Това поведение на простите правила се използва, когато трябва да се създадат множество промени, които не се дефинират наведнъж, а едно по едно в процеса на работа на програмата. Отделните сегменти с команди, които се обединяват, са разделени със запетая, за да не може една команда да интерферира синтактично с друга.

Основното неудобство на простите правила се проявява, когато те се създават в цикъл, но за една и съща променлива. Новите команди, които най-вероятно са едни и същи, се акумулират и се получава дълъг списък от еднакви команди.

Самозащитата при правилата е свойството те да не се самоактивират при промяна. Понеже `OnChange` е променлива, промяната й би трябвало да я реактивира. Това може да доведе до неприятни последици и затова правилата се самозащитават.

II.4.3 Рекурентни правила

Една от важните особености на правилата е, че те могат да бъдат рекурентни – два или повече обекта могат да си влияят взаимно. На следващия пример е показано как двете променливи `a` и `b` могат да се дефинират така, че да спазват определена математическа връзка помежду си, независимо коя от тях променяме.

Пример II-38 Рекурентни правила

Команди	Резултати

`make "a 5`
`make "b :a*:a`
 	
`make "a.onchange [make "b :a*:a]`
`make "b.onchange [make "a sqrt :b]`
 	
`make "a 10`
`print :a :b`
`make "b 36`
`print :a :b`
	
	10 100
	
	6 36

При промяната на променливата `a`, се изпълняват командите, записани в `a.onchange`. Те от своя страна променят `b`. В този случай командите в `b.onchange` се изпълняват, но стойността на `a` не се променя и не се активира повторно събитието `OnChange` за променливата `a`. При промяна на променливата `b` процесът е аналогичен, като в резултат се преизчислява `a`, така че връзката между двете променливи да се запази.

Системата Elica дава възможност в подобна схема да се свържат повече от няколко променливи, като при промяната на всяка се преизчисляват по няколко други.

II.4.4 Пряко активиране на правила

Активирането на правилата се извършва автоматично от системата при промяна стойността на променлива. Понякога се налага правилата да бъдат активирани по желание на потребителя. Има два начина за това, като единият е приложим и за обекти.

Пример II-39 Активиране на правила

Команди	Резултати
`make "a.onchange [print "changed]` `a.onchange`	 changed

Първият начин е показан на примера по-горе. Той използва пряко активиране на правилото. Този начин за активиране е приложим и за обекти. Особеното при обектите е, че командите в техните правила имат достъп до другите полета и методи на обекта. Прякото извикване на `OnChange` метода дава възможност на командите от правилото да използват обекта.

Пример II-40 Стартиране на правила

Команди	Резултати
`make "a.onchange [print "changed]` `run :a.onchange`	 changed

Вторият начин на активиране на правила е по-скоро стартирането им. Понеже те са записани под формата на списък от команди, то е възможно да се изпълнят с командата `RUN`. Особеното на този начин на действие е, че командите не се изпълняват в

контекста на обекта, а в контекста на викащата подпрограма. Поради тази причина данните за обекта са скрити и недостъпни.

II.4.5 Поименни правила

Вторият вид правила са *поименните*. Те описват едностранна връзка между две променливи, макар че има и изключения от това. Поименните правила се създават като полета в `OnChange` правилото. Името на полето е името на променливата, която се преизчислява. Естествено, това е само условно, понеже кои променливи реално се преизчисляват, зависи изцяло от командите в правилото.

Пример II-41 Поименни правила

Команди	Резултати
`make "a.onchange.b [make "b word :a :a]`	
`make "a "x`	
`print :b`	`xx`
`make "a "xor`	
`print :b`	`xorxor`

Управлението на поименните правила е по-лесно. За тях не важи адитивността. Промяната на поименно правило го заменя, а не го допълва. И за поименни правила е в сила самозащитата – те не се самоактивират при промяна.

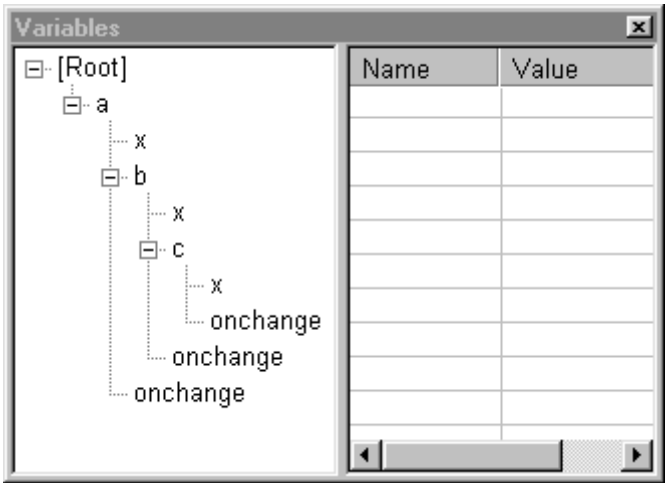
Изискването тези правила да се записват като полета в `OnChange` дават възможността да се създадат множество правила, като всяко от тях може да бъде променено независимо от останалите.

Системата Elica дава възможност простите и поименните правила да се използват съвместно. Една променлива може да има списък от команди в полето си `OnChange`, а в същото време да има няколко независими списъка, записани като стойности на полета на `OnChange`. Решението кога кои правила да се използват се предоставя изцяло на потребителя.

II.4.6 Йерархия на правилата

Особен интерес представлява поведението на обектите, когато се променят полета от тях, а в същото време са дефинирани множество правила на различни нива от йерархията на обектите. Когато се промени променлива, системата се опитва да активира правилата, асоциирани с нея.

Фигура II-2 Примерен обект за йерархия на правилата



Независимо дали има правила или не, ако променливата е поле от обект, системата се обръща към обекта-родител и се опитва да активира неговите правила. За да се илюстрира това, ще използваме сложният обект, показан на Фигура II-.

Обектът се казва `a` и съдържа полето `x`, полето `b` и метода `onchange`. От своя страна `b` е също обект, съдържащ `x`, `c` и `onchange`, където `c` е обект, съдържащ `x` и `onchange`. Промяната на някое от `x` полетата ще предизвиква активирането на различни правила.

На следващия пример е показано действието на йерархията в правилата. Променянето на стойността на `x` от най-високото ниво, предизвиква активиране само на правилото от най-високото ниво.

Пример II-42 Йерархия на правилата

Команди	Резултати
`make "a 1`	
`make "a.onchange [print 1]`	
`make "a.b.onchange [print 2]`	
`make "a.b.c.onchange [print 3]`	
`make "a.x 5`	1
`make "a.b.x 5`	2
	1
`make "a.b.c.x 5`	3
	2
	1

Ако се промени `x` от второто или третото, най-вътрешно ниво, активират се всички правила от съответното ниво до най-горното, като редът на активиране е от по-вътрешното към по-външното. Това трябва да се има предвид при създаването на сложни обекти с множество полета и правила, прикачени както към главния обект, така и към полетата.

II.5 Обекти

Езикът Elica Logo дава възможност да се дефинират обекти в стандартният смисъл на ООП. Обектите си имат дефиниции и инстанции. Съдържат полета и методи, могат да се наследяват, поддържа се полиморфизъм. Най-важното в обектите в системата е, че тяхното използване не изисква никакво допълнение в синтаксиса на езика Elica Logo.

II.5.1 Дефиниране на обекти

В системата могат да се създават обекти по два различни начина. При единия – *пряко* създаване – обектът и полетата му се създават директно с отделни команди. Всъщност така се създава конкретна инстанция на обекта. При другия начин първо се създава *дефиниция* на обекта (в други ООП среди това се нарича *клас*) и тази дефиниция се използва за създаването на неограничен брой инстанции.

Пример II-43 Пряко създаване на обекти

Команди	Резултати
`make "a.x 5`	
`make "a.y 10`	
`make "a.dump [print :x+:y]`	
`a.dump`	`15`

При прякото създаване на обекти в програмата трябва изрично да се дефинират всички методи и полета. В по-горния пример се създава обекта `a`, който има две полета `x` и `y`, и един метод `dump`. Методът отпечатва сумата от двете полета. Понеже методът е вътрешен за обекта, той може да използва директно полетата, без да се цитира името на целия обект.

Важно е да се отбележи, че по посочения начин се създава директно инстанция на обекта, без да се използва негова дефиниция.

Конвенционалният начин за създаване на обекти е да се създаде клас (т.е. описание на обекта) и в последствие се създават необходимия брой инстанции.

Пример II-44 Създаване на класове

Команди	Резултати
`to class_a :x :y`	
`  to dump`	
`  print :x+:y`	
`  end`	
`end`	
`make "a class_a 5 10`	
`a.dump`	`15`

Създаването (дефинирането) на класове в Elica не се различава по нищо от създаването на процедури. От това следва, че дефиницията на всяка процедура може да се използва като дефиниция на обект (обратното също е вярно).

Между елементите на дефиницията на процедура и дефиницията на клас съществува тясна връзка. Следващата таблица показва терминологичната връзка.

Таблица II-1 Терминологична връзка между процедури и класове

Дефиниция на процедура	Дефиниция на клас
Име на процедура	Име на клас
Параметри	Инициализационни параметри
Локални променливи	Полета
Локални процедури	Методи
Команди в процедурата	Инициализационни команди

II.5.2 Използване на обекти

Използването на обекти се базира на следните дейности: създаване, модифициране и изтриване на инстанции, достъп до полетата и достъп до методите.

Създаването на инстанции, както бе споменато по-рано, може да стане директно, но може и да се използва дефиниция на клас. В Пример II- е показано как с командата `MAKE` се създава инстанция на класа `class_a`. Името на инстанцията е `a`. По време на създаването се подават инициализиращите параметри, които служат за първоначално инициализиране на инстанцията. По време на изпълнението на `class_a` командите в тялото на дефиницията се изпълняват при на инициализацията на инстанцията. В конкретния случай се създава локална процедура (метод) с име `dump`.

Достъпът до елементите се осъществява със запазеният символ *точка*. Тя служи за създаването на идентификатор за достъп до поле на обект. Ако има обект `X` с поле `Y`, полето се цитира със следния идентификатор: `X.Y` . По аналогичен начин се прави достъп и до методите на обекта.

Модифицирането на инстанция се осъществява по стандартния за Elica начин. Независимо от това, че инстанцията вече е създадена, всички полета и методи на обекта могат да се модифицират. Не само това, но и по всяко време могат да се добавят нови полета или методи, а също и да се изтриват. Това води до уникалната възможност в Elica един обект да се трансформира в друг. Чрез последователно изтриване на излишните полета и методи и чрез вмъкване на липсващите е възможно да се трансформира инстанция от произволен клас в инстанция от друг произволен клас.

Полиморфизмът в Elica е естествена възможност поради факта, че аргументите на процедурите и класовете не са типизирани. Когато процедура очаква конкретен тип обект като параметър, а реално подадената стойност е от друг тип обект, това не представлява никакъв проблем. Ако в стойността липсва очакваното поле или метод, системата ще генерира грешка по време на работа, а ако полето или методът присъстват, те ще бъдат използвани.

Задължение на програмиста е да се подсигури по подходящ начин срещу получаването на параметри с неизползваем тип на стойността.

Когато от метод на обект се прави достъп до променлива, търсенето на тази променлива става в строго определен ред. Първо се претърсват всички локални полета на метода. Ако променливата не се намери там се преминава към прекия родител и търсенето се извършва из неговите локални полета. Ако пак не се намери, се преминава към все по-горно ниво, докато не се стигне до нивото на глобалните променливи.

Този начин на претърсване гарантира, че всеки подобект може да използва по лесен начин променливите на родителя си, без значение дали родителя е глобална променлива или е поле в друг обект.

II.5.3 Наследяване

Наследяването на обекти в Elica Logo се осъществява с командата `RUN`. Формално погледнато, наследяването се извършва като по време на инициализирането на обект се стартират телата на обектите-родители.

Пример II-45 Наследяване

Команди	Резултати

`to fly`
`  make local "wings 2`
`end`
 	
`to human`
`  make local "ears 2`
`end`
 	
`to mutant`
`  run bf :fly`
`  run bf :human`
`  to have`
`  ( print "'I have' :wings`
`  "'wings and' :ears`
`  "'ears')`
`  end`
`end`
 	
`make "me mutant`
`me.have`
	`I have 2 wings and 2 ears`

В примера са дефинирани два обекта – `fly` (с едно поле `wings`) и `human` (с поле `ears`). Третият обект – `mutant` – наследява едновременно и двата обекта, като получава всички дефинирани в тях полета. Отделно от тях се дефинира и метод `have`, който дава информация за обекта. Тази информация се извлича от наследените от двата родителя полета.

При множественото наследяване могат да настъпят конфликти между наследените идентификатори. В тези случаи интерпретаторът ще генерира грешка за дублиране на име на променливи.

При неправилна употреба, проблем могат да създадат наследяванията на обекти, които имат инициализационни параметри. В такива случаи, използването на `RUN` не позволява да се зададат параметрите и те трябва да се създадат и инициализират в рамките на наследяващия обекта.

II.5.4 Масиви

По традиция реализациите на езика Logo не включват използването на масиви. Тяхната идеология не е подходяща за идеологията на Logo. Независимо от това, в няколко от реализациите се правят опити да се използват масиви [5].

В Elica Logo също се дава възможност на потребителите да използват масиви, които не са масиви в стандартният смисъл на термина. Масивите в Elica са просто начин за записване и адресиране на обекти. Всеки обект може да се разглежда като масив индексите на който са имената на полетата в него. Обратното също е вярно – всеки, масив може да се разглежда като обект с полета – индексите на масива.

Основните свойства и особености на масивите в Elica са следните:

- Индексите могат да са числа или думи. Всяко име, което е подходящо за име на променлива, може да бъде индекс в масив;
- Числовите индекси могат да не са последователни;
- Допуска се неограничено влагане на индексите – това съответства на влагането на обекти;
- Типът на всеки елемент на масива е независим от типа на другите елементи;
- Допуска се някои елементи да имат стойност, която е масив. В резултат на тази функционалност, дълбочината на индексите на масива може да е различна за различните елементи;
- Елементи от масива могат да се създават и изтриват динамично, по време на работа на програмата;
- Масивите в Elica са по-скоро начин на записване на обектите, отколкото начин на представяне на еднородни елементи в паметта;
- Масивите в Elica не предоставят по-бърз достъп до елементите си. Те не са подходящи да се използват за оптимизация на производителността;
- Стойности могат да се присвояват не само на най-дълбоките елементи на масива, а и на всяко ниво на индексите. Тази възможност позволява да се дава допълнителна информация и за по-високите "слоеве" в масива.

Пример II-46 Масиви

Команди	Резултати
`make "a(5) 10`	
`make "a(9)(1) "car`	
`make "a(:a(9)(1)) [disp lay]`	
`print :a(5)`	`10`
`print :a(9)(1)`	`car`
`print :a("car)`	`[disp lay]`

На примера е показано създаването на 3 елемента на масива `a`. Два от елементите имат числови индекси, а третият – символен. Елементът a(9) е масив, който има един елемент. Освен с константи, индексите мога да се задават и с изрази. Допустим е и по-сложният вариант – самото име на масива да е израз (т.нар. *косвени масиви*).

Пример II-47 Косвени масиви

Команди	Резултати
`make "x "arr`	
`make (:x)(1) "one`	
`make (:x)(2) "two`	
`print :arr(1) :arr(2)`	`one two`

Косвените масиви дават невероятна мощ на програмиста и го подпомагат да записва сложни изрази по по-прост начин. Независимо от това те трябва да се използват с внимание, защото могат да предизвикат объркване на начинаещите.

II.5.5 Домейни

Използването на множество библиотеки може да доведе до неудобство, особено ако всяка от тях създава голям брой глобални процедури и функции. Неудобството се

поражда от ниското ниво на модулност. За да се реши този проблем, в Elica са въведени домейн области.

Домейните представляват обекти, чиито полета и методи са достъпни и видими като глобални променливи и подпрограми. Всички основни библиотеки на Elica дефинират процедурите и функциите си в свои домейни. По този начин се постигат следните неща:

- В рамките на един домейн са събрани тематично свързани променливи и подпрограми;
- Могат да се зареждат библиотеки, в които са дефинирани подпрограми с едни и същи имена;
- Пространството на глобалните променливи не се "задръства" с множество идентификатори – всички идентификатори от домейна се представят в глобалното пространство с името на домейна.

Пример II-48 Домейни

Команди	Резултати

`make "dom.x 5`
`domain "dom`
 	
`print :x`
	
	5

Всеки обект може да бъде направен на домейн. Това става със системната процедура `domain`. От този момент нататък (докато не се премахне свойството да е домейн), обектът отваря директен достъп до своите полета и подпрограми.

II.5.6 Виртуални полета

За разлика от другите езици за програмиране базирани на Logo, Elica Logo поддържа виртуални полета. Основната причина да се създадат е изискването да може да се направи библиотека, която колкото се може по-пълно да съвпада със съществуващата система Геомландия.

Първоначалното приложение на виртуалните полета е използването на подпрограма, която да може да извлича или променя полета от обекти. Най-простият пример е командата `ABSC` от Геомландия [12], която може да се използва и като двуаргументна процедура в случай, че полето се променя, или като едноаргументна функция в случай, че се извлича стойността на полето. И без въвеждането на допълнителни свойства, в Elica Logo е възможно да се направи това. Единственото неудобство е, че в един от двата варианта потребителят ще трябва да загражда командата с кръгли скоби.

За да се избегне това, в Elica Logo се обработват и виртуални свойства. На практика, за всяко виртуално свойство се създава по една двуаргументна процедура за промяна и по една едноаргументна функция за извличане.

Тези две подпрограми трябва да имат специално избрани имена: `put_<име-на-поле>` и `get_<име-на-поле>`. Когато в последствие потребителят иска да има достъп до полето, той използва само името му като име на функция или процедура, а компилаторът автоматично намира реалната функция или процедура, която трябва да се използва.

Виртуалността си проличава от факта, че потребителят не се интересува как се извлича полето от обекта – то може да съществува в обекта, но може и да не съществува, а да се изчислява в реално време.

Пример за използване на виртуални полета е реализацията на виртуалното поле `length` на обекта `segment`. С командата `length "segname :newlen` може да се променя дължината на отсечка, а с функцията `length :seg` се връща дължината на отсечката. Интересното в примера е, че обектът `segment` няма поле `length`, а при всяко извличане или променяне на дължината се правят необходимите изчисления.

II.6 Синтактични диаграми (БН нотация)

«program» ::= «commands»

«commands» ::= { «command» | **(**« command»**)** }*

«command» ::= «if» | «local» | «make»| «output» | «print» | «repeat »|
«run» | «to» | «while »| «user» | , | «cr»

«if» ::= **IF** «expr» «expr» {«expr»}

«local» ::= **LOCAL** (**"** «word»)*

«make» ::= **MAKE** {«expr»} «expr» {«expr»{«expr»}}

«output» ::= **OUTPUT** {«expr»}

«print» ::= **PRINT** {«expr»}*

«repeat» ::= **REPEAT** «expr» « expr»

«run» ::= **RUN** «expr»

«to» ::= **TO** {(**:** | **"**) «word»}* «word» {(**:** | **"**) «word»}* «commandlist» **END**

«while» ::= **WHILE** «expr» « expr»

«user» ::= «word» {«expr»}*

«expr» ::= «list» | {**:**} **(**«expr»**)**| « number» | **"**«word» | **:**«word» |
LOCAL «expr» | «user»

«list» ::= **[** {«list» | «word » | **:** | **"** | **,**}* **]**

«word» ::= («visible ascii character» \ **'**)* | **'**(«word» | **' '**)* **'**

«number» ::= {**-**}«digits»{**.** «digits»{(**E**| **e**){+ | **-**}«digits»}}

«digits» ::= (**0** .. **9**)*

«cr» ::= «ascii CR character»

II.7 Унификация на метаобектите

Унификацията на метаобектите в Elica е резултат от желанието да се проектира език за програмиране със силно ограничен набор от запазени думи, но с възможност да се дефинират функции, процедури, оператори и обекти. Въпреки че синтактичните правила в езика са прости, има възможност да се конструират много сложни програми и структури, които не са достъпни за другите езици.

II.7.1 Въведение

По традиция за Logo езиците няма единна спецификация, въпреки, че някои синтактични структури са едни и същи във всичките им реализации. Интерес представлява да се проследи връзката между Elica Logo и другите реализации на Logo.

Проектирането на Elica Logo и унификацията на метаобектите е осъществено на няколко стъпки.

- Премахнати са повечето запазени думи, докато техният брой не бе сведен до 9. От тях също има "ненужни" думи, но те са запазени по исторически причини;
- Втората стъпка е да се премахнат всички сложни типове данни и да се запазят само най-основните – числа, думи и списъци;
- Премахване на всички процедури и функции, както дефинирани от потребителя, така и системни. В резултат, ще липсва дефиниция на функция, която събира две числа;
- Опростяване на вътрешното представяне на данните и свеждането му до използването на универсални блокове от памет.

Имайки предвид всичките тези опростявания, може да се зададе въпросът какво се очаква да прави подобна ограничена система. Тя дава възможност:

- да дефинира всички премахнати запазени думи, функции и процедури;
- да дефинира оператори с техните приоритети и асоциативности;
- да дефинира обекти (с полета, методи, множествено наследяване) и всички останали премахнати сложни структури (масиви, множества, и т.н.);
- да дефинира събития, виртуални и невиртуални свойства, библиотеки и домейни.

Поради унификацията на вътрешното представяне,променливите, процедурите, функциите, обектите, операторите, масиви, домейни и библиотеки се представят по един и същ начин, което естествено води до идеята, че тази унификация на данните може да е видима и от страна на потребителя. На практика, това води до едно от най-високите постижения на Elica – за потребителя няма разлика между процедура и променлива или пък между оператор или обект.

II.7.2 Унификация на данните

Система Elica представя всички данни с еднотипни блокове от памет. Всеки блок съдържа две части: *информационна* и *релационна*. Информационната част може да съдържа числа, думи или елементи на списъци. Релационната част съдържа данни, описващи вътрешната йерархия и релации между блоковете памет.

Всички блокове са организирани под формата на дърво. Това изисква наличието на вертикална релация от типа родител-дете. Всеки блок може да има, но може и да няма, родител. Същото се отнася и за децата. Друга налична релация е хоризонталната, от тип брат-брат. Всички деца на един и същ родител задължително са в хоризонтална релация помежду си.

Теоретичните изследвания и практическите тестове показват, че подобна структура и връзки между блоковете са напълно достатъчни.

II.7.3 Обединяване на функции и процедури

Началото на унификацията на метаобектите в Elica Logo започва с унификацията на процедурите и функциите. В езика Logo това се осъществява сравнително лесно, понеже и двата метаобекта имат еднаква дефиниция. Единствената разлика е, че процедурите завършват с изпълнението на последната команда от тялото на процедурата, или пък се изпълни командата `OUTPUT` без аргумент. Функциите задължително завършват с командата `OUTPUT`, но с аргумент.

На следващия пример са показани команди, правещи едно и също, но едната използва процедура, а другата – функция.

Пример II-49 Обединяване на функции и процедури

Процедура	Функция

`to sqr :x`
`  make "x :x*:x`
`  print :x`
`end`
`sqr 5`
	`to sqr :x`
	`  make "x :x*:x`
	`  output :x`
	`end`
	`print sqr 5`

Примерът провокира следния въпрос: ако имаме подпрограма, която в някои случаи (чрез използването на командата `IF`), завършва своето изпълнение с безаргументна команда `OUTPUT`, а в други случаи `OUTPUT` е с аргумент, каква е подпрограмата – функция или процедура? Отговорът е: и двете, защото ще може да се използва и като функция и като процедура.

Има два независими фактора, които биха помогнали на стандартно мислещия потребител да различи процедури от функции:

- Фактор 1: Връща ли подпрограмата някаква стойност?
- Фактор 2: Очаква ли викащата команда да получи някаква стойност?

В повечето езици за програмиране двата фактора задължително са силно синхронизирани, в противен случай се генерира грешка. В последно време някои езици позволяват *полусинхронизация*, която е естествена за Elica. Полусинхронизацията е когато подпрограмата връща стойност, а викащата команда не я очаква. В такива случаи тази стойност се игнорира.

Особеното в Elica е, че е реализирана и пълна *десинхронизация*. На практика, това означава, че могат да се правят всички възможни комбинации от двата фактора. Десинхронизацията (по-точно използване на процедура като функция) е табу за повечето програмни езици и противоречи на унификацията.

II.7.4 Обединяване на процедури, функции и променливи

Понеже езикът Logo е създаден като предимно синтактично опростяване на LISP, за него е естествено да представя командите и данните по един и същ начин. Тази възможност е реализирана в повечето реализации на Logo, включително и в Elica. Програмите се представят като списък от команди, а командите са съставени от думи (запазени символи и думи, потребителски идентификатори, числа и т.н.), затова всяка програма може да се представи като списък.

Това не означава, че вътрешното представяне на процедурите и функциите е същото като на списъците, но поне за система Elica това е вярно – всички команди се

съхраняват като списъци и се обработват като такива. Това дава възможност променливите със стойност списък да се използват като процедури и функции, а също и обратното – процедурите и функциите да се използват като променливи.

В първият случай - променливите като подпрограми – на променлива се присвоява списък. Ако списъка съдържа валидни команди и може да се изпълни, може да се изпълни и променливата като че ли е подпрограма.

Изпълняването на последната команда от следващия пример ще накара Elica да третира идентификатора `a` като име на процедура. Тялото на процедурата е всъщност съдържанието на списъка, който е присвоен на променливата.

Пример II-50 Променливи като подпрограми

Команди	Резултати
`make "a [print "Hello]`	
`run :a`	`Hello`
`a`	`Hello`

Обединяването на променливите с подпрограмите е напълно симетрично. Всяка подпрограма може да бъде използвана и като променлива, която има стойност списък от командите съставляващи тялото на подпрограмата. Понеже стойността на променливите може да се променя свободно по време на изпълнение на програмата, възможно е да се предефинират и самите процедури и функции.

Пример II-51 Подпрограми като променливи

Команди	Резултати

`to b :x`
`  print "OK :x`
`end`	

`b "Tom`	`OK Tom`
`print :b`	`[[: x] , print " OK : x ,]`

Важно следствие на това обединение е, че всички процедури и функции могат да се създават, без да се използва `TO…END` конструкцията. По исторически причини в Elica тази конструкция може да се използва, независимо че вътрешно тя се конвертира в еквивалентна `MAKE` команда.

II.7.5 Обединяване на процедури, функции и оператори

Една от най-интересните възможности на Elica е, че дефинициите на подпрограми не са твърдо фиксирани. При традиционната форма името на подпрограмата е следвана от списък с имената на параметрите. При Elica е възможно да се премести името на подпрограмата така, че да е между или дори и след имената на параметрите.

Следващият пример показва два начина на дефиниране на подпрограма за събиране на две числа.

Пример II-52 Обединяване на подпрограми и оператори

Функция	Оператор
`to + :x :y`	`to :x + :y`
`  …`	`  …`
`end`	`end`

Левият пример показва как може да се дефинира + като функция, а десният – като оператор. Elica може да работи и с двете дефиниции. Всъщност, Elica може да работи независимо къде е споменато името на подпрограмата (оператора) – вътрешните

механизми на компилация на Elica Logo програми не правят разлика между функции и оператори. По този начин процедурите и функциите са частен вид оператори и като следствие, в Elica може да се дефинира процедурен оператор. Друго приложение на тази възможност е смесването на префиксни, инфиксни и суфиксни оператори в един и същ израз, като реда на изчислението се определя от приоритетите им и от асоциативностите им. Най-интересното приложение, което е недостъпно за повечето езици за програмиране е, че могат да се дефинират оператори с по няколко аргумента вляво и вдясно от оператора.

II.7.6 Обединяване на обекти, масиви и променливи

Съществуват редица изследвания относно най-удачният начин за семантичното и физическото представяне на обектите. Един оптимален начин е представянето чрез дърво [16]. Всъщност, вътрешната структура на представянето на обектите в Elica е дървовидна.

В точка II.7.3 бяха дискутирани двата фактора, които влияят на различаването на процедури от функции. Понеже тези фактори са независими, различните комбинации са общо 4. За три от тях вече е дадено обяснение.

Таблица II-2 Фактори, определящи типа на метаобект

		Извикваната подпрограма връща стойност	
		Не	**Да**
Викащата команда очаква резултат	**Не**	Подпрограмата е **процедура**	Подпрограмата е **процедура**, а стойността се **игнорира**
	Да	Подпрограмата е **обект**	Подпрограмата е **функция**

Интересният вариант е, когато извикващата команда очаква стойност, а извикваната подпрограма не предоставя такава. В този случай след завършване на изпълнението на подпрограмата системата не изтрива локалните данни, а ги запазва и ги предава автоматично като резултат. На практика резултатът е променлива, която съдържа в себе си всички локални променливи и подпрограми, които са съществували в момента, в който извикваната подпрограма е завършила изпълнението си. Стойността на тази променлива е точно това, което би се очаквало ако подпрограмата се извика като клас с цел създаване на инстанция на обект.

Важно следствие е, че не съществува начин, по който да се разбере дали една дефиниция е дефиниция на обект или на процедура. Всичко зависи от това как тя се използва. По тази причина процедурите и класовете са едно и също нещо. Но процедурите са и променливи, затова и класовете са променливи. Не бива да се забравя, че както е описано по-горе, инстанциите на даден клас са също променливи. Това означава, че може да се напише оператор, чийто резултат да е дефиниция на клас, нещо, невъзможно в останалите езици за програмиране.

Дефинирането на класове става с командите `TO`…`END`, но те не се използват от ядрото на системата, понеже още в началните фази на транслиране се конвертират до еквивалентни конструкции на базата на командата `MAKE`.

Масивите, разглеждани като по-особена нотация на обектите, са също нормални променливи, които имат подчинени променливи. Ето защо този факт може да се използва от по-умелите програмисти по следния начин: дефинирането и инициализирането на масив, може да се направи чрез обект и инстанцията на обекта

после се използва като масив. По този начин се спестява многократното изписване на името на масива, понеже инициализацията ще става "вътре" в него.

Интересно приложение на унификацията е създаването на метаобекти, за които няма подходяща терминология. Това не означава, че те са безсмислени. Например, система Elica позволява по естествен начин да се създаде структура, която е едновременно и клас и оператор, или пък променлива с числова стойност, която е и инстанция на обект и масив.

II.7.7 Заключение

Почти невъзможно е да се опишат всички метаструктури, които могат да се създадат в Elica и да се обясни тяхното приложение. Всичко това зависи от фантазията на програмиста и от умението му да работи с непознати структури. Затова е по целесъобразно да се представи тримерна диаграма на възможните метаобекти и техните основни характеристики. За улеснение, в диаграмата не са включени масивите и домейните. На следващата фигура са показани двадесет и четирите възможни интерпретации на променливите в Elica. Трябва да се отбележи, че само шест от тях са реално достъпни в другите езици за програмиране.

Поради терминологични проблеми са поставени имена само на тези интерпретации, за които има съответствия в другите езици.

Самите интерпретации са структурирани в тримерен паралелепипед. Стойностите по трите измерения определят еднозначно свойствата и типа на съответната интерпретация. Семантиката на измеренията е следната:

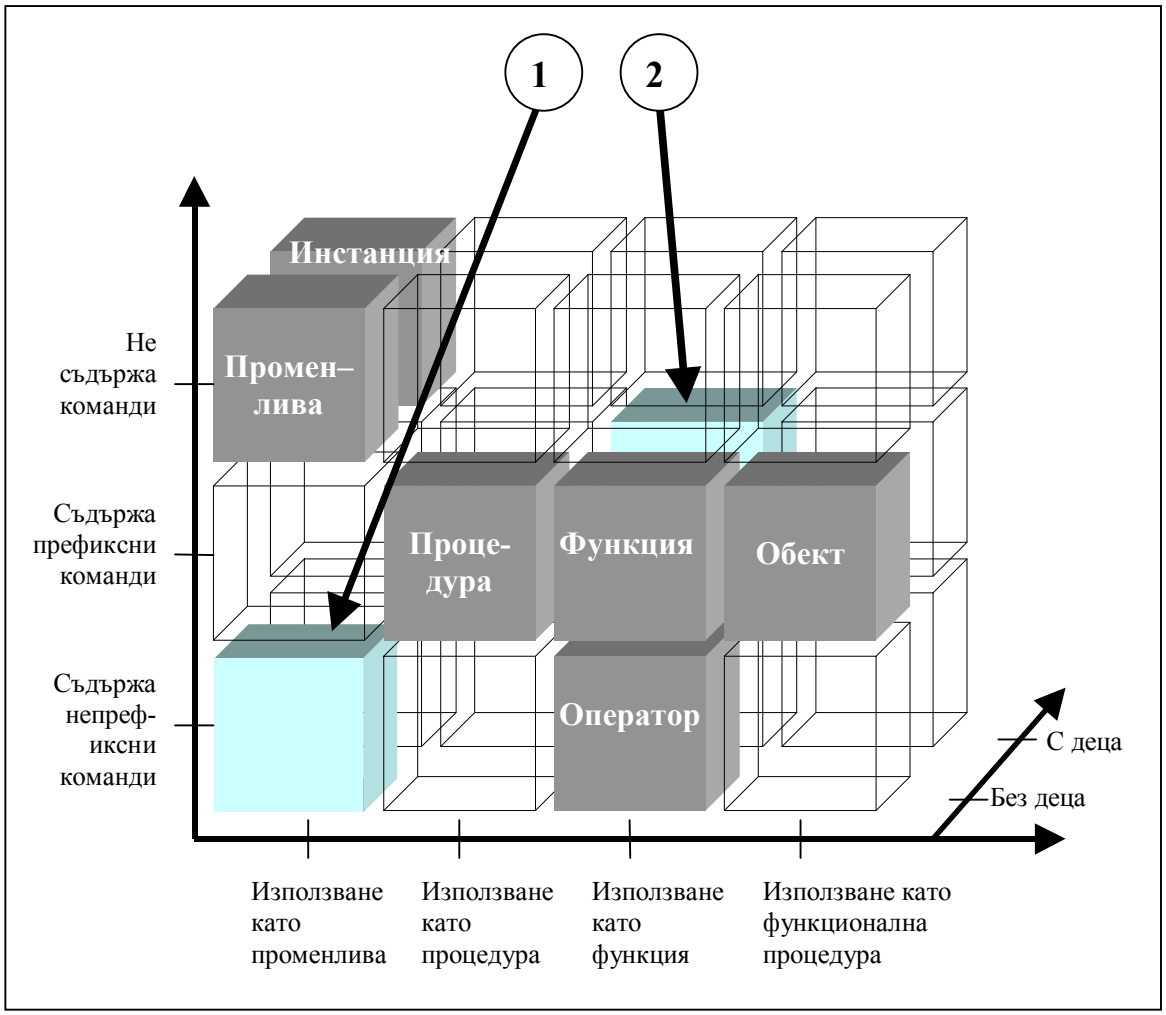
- Най-простото свойство, по което можем да различаваме променливите е това, дали те имат деца или не. Първият, най-близък слой от диаграмата е за интерпретации за променливи, които нямат деца. Вторият слой е за променливи, които имат едно или повече деца;
- Колоните в схемата определят начина на използване на променливата: като обикновена променлива, като процедура, като функция или като процедура използвана като функция (*функционална процедура*);
- Редовете показват прякото съдържанието на променливите (децата не се считат за част от прякото съдържание). Трите различни случая са, когато съдържанието не е списък от команди или е списък от команди, като във втория случай се прави разграничение между префиксен и непрефиксен формат. Префиксният формат е когато дефиницията започва с името на метаобекта, следвано от списък с параметри. Непрефиксният вариант е когато има поне един параметър от ляво на името.

Шестте посивени блока са метаобектите, които другите езици предлагат: променливи, процедури, функции, оператори, класове и инстанции. Всички останали блокове са достъпни единствено за Elica.

Например, блокът, маркиран с (1) на Фигура II-, описва метаобект, който няма деца, съдържа инфиксна (или суфиксна) дефиниция и се използва като променлива. Независимо, че е трудно да се опише подобен метаобект [10], в контекста на Elica той може да се интерпретира по следния начин:

Нека имаме оператора `div`. Ако изпълним командата `print :div` ще използваме `div` като променлива, която няма деца и има инфиксно съдържание (предполага се, че дефиницията на `div` е нещо подобно на `to :x div :y … end`). Крайният резултат ще е, че дефиницията на `div`, ще се интерпретира като списък от думи и в този си вид ще бъде отпечатана на екрана.

Фигура II-3 Структуриране на метаобектите



Вторият пример, отбелязан с (2) на диаграмата, е малко по-сложен. Разглеждайки "координатите" на блока, ще се открие, че той съответства на променлива, която има деца, използва се като функция и има префиксно съдържание. За да успеем да идентифицираме подобен метаобект в някой друг език, ще трябва да намерим инстанция на обект, която се използва като функция и връща резултат след изчисляването си.

Както бе споменато, всеки метаобект в Elica може да се използва по различни начини и да бъде модифициран по време на работа на програмата. Това дава възможност на потребителя да променя стойностите на променливите така, че те да станат инстанции на обект, като оригинално са били създадени като дефиниции на оператори, а по-късно, инстанциите да бъдат използвани като функции. Тази гъвкавост обуславя защо в Elica променливите, процедурите, функциите, обектите, класовете, масивите, домейните и операторите са едно и също нещо и поради тази причина не трябва да се прилага стандартната терминология, без да се разбира същността. А именно, когато се казва, че `A` е обект, това означава, че в този конкретен момент променливата с име `A` се използва като че ли е обект.

Тази мултифункционалност може да създаде проблеми на конвенционално мислещите ООП програмисти, понеже преминаването към семантиката на метаобектите в Elica не е тривиален процес [3]. Но веднъж осъзнавайки взаимосвързаността на всички метаобекти, всички "екзотични" възможности на Elica стават съвсем естествени и позволяват на програмиста да се концентрира на смисъла, а не на формата [17].

II.8 Разширения и потребителски библиотеки

Поради ограничения набор от запазени думи, ядрото на системата Elica не може да работи пряко с типовете данни. Това е предвидено да се прави с библиотеки, които надстройват функционалността на системата. Някои от библиотеките са системни, като например библиотеката Sys – за достъп до системни функции, Logo – за надстройване на Elica Logo до пълнофункционален Logo език, Win – за създаване на бутони, IO – за вход и изход. Други от библиотеките са на по-високо ниво: Geomland – система за планиметрия, Graphix – библиотека за тримерна графика, Turtle – за костенуркова графика, Astro – за небесна механика.

Примери за отделните процедури и функции в библиотеките могат да се намерят на сайта на система Elica: http://www.elica.net.

II.8.1 Библиотека Sys

Библиотеката Sys включва дефиниции на подпрограми, с които се управляват характеристиките на променливите, прави се достъп до имената на дефинираните идентификатори и други. За да се стартира библиотеката трябва в началото на програмата да се напише `run "sys`. Библиотеката може да се включва направо – тя не изисква предварителното зареждане на други библиотеки.

Понеже библиотеката Sys е от особена важност (тя се използва от всички останали библиотеки), по подразбиране работната среда я включва в изпълнението.

След зареждането си Sys създава домейн с име `sys`, в който са записани всички процедури и функции.

II.8.1.1 Функции за достъп до имената

Съществуват няколко функции за достъп до идентификаторите в Elica. Част от тях са за извличане на списъци с дефинираните идентификатори, други се използват за проверка на конкретни идентификатори, а трети – за изтриване.

`names :x`	Връща списък от всички полета и методи на обекта x
`globalnames`	Връща списък от всички глобални идентификатори
`localnames`	Връща списък от всички локални идентификатори спрямо текущото ниво на изпълнение
`local? :#x`	Проверява дали определена дума е име на локална променлива
`exists? :#x`	Проверява дали определена дума е има на достъпна променлива
`delete "#x`	Изтрива идентификатор с дадено име
`deletesub :#x`	Изтрива всички подчинени полета и методи на обект

II.8.1.2 Системни флагове

Библиотеката Sys предоставя достъп до някои системни флагове, които са фиксирани в ядрото на системата и не трябва да се променят от потребителя. Тези флагове се използват във всички променливи, като указват тяхното поведение в различни случаи.

Битовите константи, съответстващи на флаговете, са декларирани в обекта `userbit` и са следните:

Таблица II-3 Системни битове

Поле	Бит	Значение
lock	0	Заключване на променлива
definition	1	Създаване на дефиниция на променлива
historical	2	Създаване на история на променлива
syntax	3	Създаване на синтактични зависимости между променливи
hidden	5	Скриване на визуалния образ на променлива
domain	6	Обозначаване на променлива като домейн

`setbit :variable :bit` Установява даден бит

`clearbit :variable :bit` Изчиства даден бит

`getbit :variable :bit` Връща стойността на даден бит

Функциите за достъп до системните битове не би трябвало да се ползват пряко от потребителите, освен, ако те не си дефинират собствени битове и съответно поведение на системата.

Използването на системните битове се осъществява от специални функции, по три за всеки отделен бит.

В бъдещите версии на системата, могат да бъдат добавени още битове, затова, ако потребителят иска да дефинира и използва собствени битове, е добре да се заемат битове с номера от 16 до 31.

II.8.1.3 Функции за достъп до системните флагове

За работа със системните флагове са предоставени набори от функции. Първата група е за работа с флага `lock`. По подразбиране променливите не са заключени и могат да се променят по всяко време. Когато променлива се заключи, промяната й се забранява.

`lock :variable` Заключване на променлива

`nolock :variable` Отключване на променлива

`lock? :variable` Проверка дали променлива е заключена или не

Дефиниция на променлива е командата `MAKE`, с който тя е била създадена. По подразбиране променливите не пазят дефиницията си, но ако за някоя от тях флагът `definition` е установен, към променливата ще се създаде подчинена променлива с име `definition`, която ще съдържа последната дефиниция на променливата като команда на Elica Logo. Създаването на дефиниции забавя изпълнението на програмата и затова трябва да се използват само при нужда.

`definition :variable` Указва, че дадена променлива трябва да запазва дефиницията си

`nodefinition :variable` Премахва задължението на променлива да запазва дефиницията си

`definition? :variable` Проверява дали променлива създава дефиницията си

Полето `definition` се създава след изпълнението на първата `MAKE` команда след използването на процедурата `definition`. След изпълнението на `nodefinition` полето не се премахва.

Флагът `syntax` се използва, когато променлива трябва автоматично да създаде връзки между себе си и използваните променливи в команда MAKE, с която променливата се променя или създава. Връзките са еднопосочни и указват преизчисляване на тази променлива, ако някоя от използваните се промени.

Използването на този флаг трябва да става внимателно, защото той забавя изпълнението на програмата.

`syntax :variable`	Указва, че за дадена променлива трябва автоматично да се генерират правила OnChange
`nosyntax :variable`	Указва, че за дадена променлива не трябва автоматично да се генерират правила OnChange
`syntax? :variable`	Проверява дали за променлива трябва автоматично да се генерират правила OnChange или не трябва

Когато за променлива флагът `hidden` е установен, независимо дали тази променлива има метод `ondrawimage`, изображението й не се показва на екрана. По подразбиране този флаг е нулиран за всички променливи.

`hidden :variable`	Скрива изображението на променлива
`nohidden :variable`	Показва изображението на променлива
`hidden? :variable`	Проверява дали променлива се скрива насилствено

Флагът `domain` определя дали определен обект ще се използва като домейн от системата. Когато обект е домейн, всички негови полета и методи са достъпни като че ли са глобални.

`domain :variable`	Създава домейн
`nodomain :variable`	Премахва домейн
`domain? :variable`	Проверява дали обект е домейн

Историята на променливите се създава само за тези от тях, за които флагът `history` е установен. Всъщност дали ще се създава история или не зависи и от реализацията на двете събития, които се активират при създаване на история и при актуализация на история.

`historical :variable`	Указва да се пази история на променлива
`nohistorical :variable`	Указва да не се пази история на променлива
`historical? :variable`	Проверява дали се пази история на променлива
`onhistorycreate :name`	Процедура за обработване на събитието, което се активира при първоначално създаване на история
`onhistoryupdate :name`	Процедура за обработване на събитието, което се активира при всяко последващо актуализиране на история

II.8.1.4 Други процедури

Останалите системни процедури се използват за прерисуване на екрана или отделни обекти и за генериране на потребителски съобщения за грешки.

`error :x`	Процедура за генериране на дефинирано от потребителя съобщение за грешка
`repaint`	Изпраща заявка към системата да пречертае графичния екран
`draw "#x`	Изпраща заявка към системата да пречертае конкретна променлива

II.8.2 Библиотека Logo

Библиотеката Logo включва дефиниции на процедури и функции, с които се създава пълноценна Logo среда. За да се стартира библиотеката трябва в началото на програмата да се напише `run "logo`. Библиотеката не може да се включва направо, понеже изисква предварителното зареждане на библиотеката Sys.

Понеже библиотеката Logo е от особена важност (тя се използва от всички останали библиотеки), по подразбиране работната среда я включва в изпълнението.

След зареждането си Logo създава домейни с имена `logo` и `math`, в който са записани всички процедури и функции.

II.8.2.1 Оператори в Logo

В домейна `logo` са дефинирани 5 оператора за основните математически операции и 6 логически оператора за сравнение. Зададени са и приоритетите им така, че редът на изпълнение на операторите да е естествения ред, приет в математиката.

`:x + :y`	Оператор за събиране на две числа
`:x * :y`	Оператор за умножение на две числа
`:x - :y`	Оператор за изваждане на две числа
`:x / :y`	Оператор за деление на две числа
`:x ^ :y`	Оператор за степенуване на две числа

С най-висок приоритет е операторa `^`, следват равноприоритетните `*` и `/`, а с най-нисък приоритет са `+` и `-`. И петте оператора могат да работят с безкрайности и неопределени числа.

Операторите за сравнение връщат стойности `"True` или `"False` според резултата на сравнението. Аргументи могат да са както числа и думи, така и списъци.

`:x = :y`	Сравнение за "равно"
`:x < :y`	Сравнение за "по-малко"
`:x > :y`	Сравнение за "по-голямо "
`:x <= :y`	Сравнение за "по-малко или равно"
`:x >= :y`	Сравнение за "по-голямо или равно"
`:x <> :y`	Сравнение за "неравно"

На всички оператори за сравнение приоритетите са по-ниски, както от тези за математически операции, така и от подразбиращия се приоритет на потребителските подпрограми.

Библиотеката Logo дефинира и 3 логически оператора. Те връщат стойност "True или "False според стойностите на аргументите си.

`not :x`	Логическо отрицание "НЕ"
`:x and :y`	Логическо "И"
`:x or :y`	Логическо "ИЛИ

От логическите оператори с най-висок приоритет е `not`, следван от `and` и накрая е `or`. И трите оператора са "по-силни" от операторите за сравнение, но "по-слаби" от обикновените потребителски подпрограми, на които също може да се дефинира приоритет.

Всички споменати до момента оператори са дефинирани в домейна `logo`, а единствено следващите два – в `math`.

`:x idiv :y`	Целочислено делене
`:x imod :y`	Остатък при целочислено делене

II.8.2.2 Математически функции

Математическите функции в библиотеката Logo са наследени идеологически от система Геомландия. Те се използват в различни пресмятания. На базата на тях потребителите могат да създават по-сложни функции.

`pi`	Връща стойността на константата π
`sqrt :x`	Изчислява квадратен корен от число
`trunc :x`	Изрязва дробната част от число
`abs :x`	Връща абсолютната стойност на число
`sign :x`	Определя знака на число
`neg :x`	Променя знака на число на обратния
`exp :x`	Изчислява натурална експонента
`logn :x`	Изчислява натурален логаритъм
`sin :x`	Изчислява sin от ъгъл в градуси
`cos :x`	Изчислява cos от ъгъл в градуси
`tan :x`	Изчислява tg от ъгъл в градуси
`cotan :x`	Изчислява cotg от ъгъл в градуси
`arcsin :x`	Изчислява arcsin в градуси
`arccos :x`	Изчислява arccos в градуси
`arctan :x`	Изчислява arctg в градуси
`arccotan :x`	Изчислява arccotg в градуси
`random :x`	Генерира случайно реално число от 0 до 1 (ако се използва без аргументи), или цяло число от 0 до :x-1

`round :x :y`	Закръглява число с определена точност

II.8.2.3 Функции за работа с числа, думи, списъци и множества

Основният набор от функции, характерни за почти всички реализации на езика Logo, са тези за работа с числа, думи и списъци. Същите функции в Logo библиотеката могат да се използват и за множества. Функциите, които работят на ниво елементи, третират цифрите на числата и буквите от думите за елементи.

Множествата в Logo библиотеката са обекти, чийто елементи имат имена във вида #0, #1, #2, и т.н. Поради унификацията, множествата могат да се използват като множества (чрез съответните функции) или като масиви (чрез скобна-нотация) или като обекти (чрез точкова нотация). Всичко зависи от потребителя – при какъв начин на представяне се чувства най-комфортно.

`number? :x`	Проверява дали стойност е число
`word? :x`	Проверява дали стойност е дума
`list? :x`	Проверява дали стойност е списък
`set? :#x`	Проверява дали стойност е множество
`word :x :y`	Създава дума
`list :x :y`	Създава списък
`se :x :y`	Създава плосък списък
`set`	Създава множество
`count :x`	Преброява елементите
`bf :x`	Премахва първите един или няколко елемента
`bl :x`	Премахва последните един или няколко елемента
`first :x`	Връща първите един или няколко елемента
`last :x`	Връща последните един или няколко елемента
`item :y :x`	Връща елемент, определен от индекса му
`fput :x :y`	Добавя елемент в началото
`lput :x :y`	Добавя елемент в края
`setmax :x`	Връща максималния индекс на елемент в множество
`setmin :x`	Връща минималния индекс на елемент в множество

II.8.2.4 Функции за работа със списъци-множества

Списъците-множества са обикновени списъци, над които могат да се прилагат някои функции за множества. Елементите в множествата са наредени. Всички функции за работа със списъци-множества работят само с елементите от най-високото ниво.

`member :x :y`	Намира къде елемент присъства в списък
`member? :x :y`	Проверява дали елемент присъства в списък
`sunion :x :y`	Извършва обединение на елементите на два списъка
`ssection :x :y`	Извършва сечение на елементите на два списъка

`sminus :x :y`	Извършва изваждане на елементите от два списъка

II.8.2.5 Други функции

`eps :x`	Променя точността на сравнявания на числа. По подразбиране точността е 10^{-10}
`pow :a :i`	Повдига число на цяла степен
`ascii :x`	Определя ASCII кода на буква
`char :x`	Определя буква според ASCII кода й
`ms`	Връща изминалото време в милисекунди от някакъв фиксиран момент, определен от операционната система
`wait :x`	Прави пауза за определено време
`INF? :x`	Проверява дали число е безкрайност
`NAN? :x`	Проверява дали число е неопределено

II.8.3 Библиотека Geomland

Библиотека Geomland е направена като функционален заместител на система Геомландия. Тя дефинира основните линейни обекти и криви от втора степен, а също и операциите с тях. Основните обекти в Geomland са точка, права, отсечка, лъч, вектор, окръжност, елипса, парабола, хипербола и ъгъл. Всички дефиниции на обекти са създадени в домейна `geomland`. Създаден е и още един домейн за по-лесен достъп до предварително дефинирани цветове.

II.8.3.1 Обект point (точка)

Обектът `point` е интерпретация на геометричното понятие "точка" в равнината. Полетата на обекта съдържат `x`, `y` и `z` координатите на точката (`z` = 0), `type` съдържа думата "`point`, а `color` – подразбиращият се за точките цвят. Функциите за работа с точки са:

`point :x :y`	Конструктор на обект от тип точка
`point? :x`	Проверява дали обект е точка
`normalize :p1`	Нормализира точката (премества я по посока на (0,0) докато радиус-векторът стане с единична дължина)
`get_absc :obj`	Извлича стойността на виртуалното поле absc (абсциса на точка)
`put_absc "name :value`	Променя стойността на виртуалното поле absc
`get_ord :obj`	Извлича стойността на виртуалното поле ord (ордината на точка)
`put_ord "name :value`	Променя стойността на виртуалното поле ord
`distance :p1 :p2`	Изчислява разстоянието между две точки

За по-лесно изписване на математически изрази, библиотека Geomland дава възможност да се използва точкова (или векторна) аритметика. Векторите в

библиотеката се представят чрез точките и векторната аритметика се реализира чрез точкова аритметика.

`:p1 + :p2`	Събира координатите на две точки
`:p1 - :p2`	Изчислява разликата на координатите две точки
`:p1 * :p2`	Умножава координатите на точка с число
`:p1 / :p2`	Дели координатите на точка с число
`:p1 ° :angle`	Завърта точка около (0,0) на определен ъгъл

II.8.3.2 Обекти line, segment, ray и vector (линия, отсечка, лъч и вектор)

Поведението на тези обекти в библиотека Geomland е почти идентично. Основната разлика е в начина на изобразяване. Вътрешната структура и на трите обекта е една и съща – начална и крайна точка (`initial` и `final`), тип (`type`) и цвят (`color`).

`line :initial :y`	Конструктор на обект от тип линия
`segment :initial :y`	Конструктор на обект от тип отсечка
`ray :initial :y`	Конструктор на обект от тип лъч
`vector :x :y`	Конструктор от тип вектор
`line? :x`	Проверява дали обект е права
`segment? :x`	Проверява дали обект е отсечка
`ray? :x`	Проверява дали обект е лъч
`vector? :x`	Проверява дали обект е вектор
`dx :obj`	Изчислява разликата на абсцисите на началната и крайната точки
`dy :obj`	Изчислява разликата на ординатите на началната и крайната точки
`get_heading :obj`	Извлича стойността на виртуалното поле heading (ъгъл, който се сключва с оста Ox)
`get_initial :obj`	Извлича стойността на виртуалното поле initial (начална точка)
`get_final :obj`	Извлича стойността на виртуалното поле final (крайна точка)
`get_length :obj`	Извлича стойността на виртуалното поле length (разстояние между началната и крайната точка)
`put_heading "name :value`	Променя стойността на виртуалното поле heading
`put_initial "name :value`	Променя стойността на виртуалното поле initial
`put_final "name :value`	Променя стойността на виртуалното поле final
`put_length "name :value`	Променя стойността на виртуалното поле length

II.8.3.3 Обект circle (окръжност)

Обектът `circle` е софтуерен еквивалент на геометричното понятие "окръжност" в равнината. Полетата на обекта съдържат `center` (център на окръжността), `radius` (радиуса й), `type` съдържа думата `"circle`, а `color` – подразбиращият се за окръжностите цвят. Функциите за работа с окръжности са:

`circle :center :y`	Създава окръжност по център и радиус или център и точка, през която минава
`circle? :x`	Проверява дали обект е окръжност
`get_center :obj`	Извлича стойността на виртуалното поле center (център на окръжността)
`get_radius :obj`	Извлича стойността на виртуалното поле radius (радиус на окръжността)
`put_center "name :value`	Променя стойността на виртуалното поле center
`put_radius "name :value`	Променя стойността на виртуалното поле radius

За обекта окръжност свойството `initial` също е достъпно само за четене.

II.8.3.4 Обекти ellipse и hyperbola (елипса и хипербола)

Двата обекта са идентични като вътрешна структура и като функции за достъп. Единствената разлика е в начина на изобразяване. И двата обекта съдържат двата си фокуса `focus1` и `focus2`, `center` – център на фигурата, два радиуса – `radiusx` и `radiusy`, `spin` – наклон на фигурата, тип `type`, който съдържа думите `"ellipse` или `"hyperbola`, и `color`, съдържащ цвета на обекта.

`ellipse :focus1 :focus2 :x`	Създава елипса по двата й фокуса и ексцентритет или трета точка, през която минава
`hyperbola :focus1 :focus2 :x`	Създава хипербола по двата й фокуса и ексцентритет или трета точка, през която минава
`ellipse? :x`	Проверява дали обект е елипса
`hyperbola? :x`	Проверява дали обект е хипербола
`get_focus :obj`	Извлича стойността на виртуалното поле focus, което е множество от двата фокуса

За обектите елипса и хипербола свойството `initial` също е достъпно само за четене.

II.8.3.5 Обект parabola (парабола)

Обектът парабола съответства на геометричния обект парабола. Полетата на обекта са фокуса на параболата, формиращата директриса, център, радиус, завъртяност, тип и цвят. Завъртяността, центърът и радиусът се пресмятат на базата на фокуса и директрисата.

`parabola :foc :directrissa`	Създава парабола
`parabola? :x`	Проверява дали обект е парабола

II.8.3.6 Сечения на геометрични обекти

Библиотеката Geomland съдържа функции за намиране на сечението на някои от обектите – точки, линии и окръжности. Допустимо е да се използват отсечки и лъчи, при изчисляване на сечението те се третират като прави. Повечето функциите за намиране на сечение са изнесени в отделен файл – `intersection.eli`.

`isec :obj1 :obj2`	Основна функция за намиране на сечение. Проверява типа на аргументите, и ако за такава комбинация от типове съществува функция за намиране на сечение, то тя се използва
`isec_pointpoint :p1 :p2`	Изчислява сечение на точка с точка
`isec_pointline :p :l`	Изчислява сечение на точка с линия
`isec_linepoint :l :p`	Изчислява сечение на линия с точка
`isec_pointcircle :p :c`	Изчислява сечение на точка с окръжност
`isec_circlepoint :c :p`	Изчислява сечение на окръжност и точка
`isec_lineline :l1 :l2`	Изчислява сечение на линия и линия
`isec_circlecircle :a :b`	Изчислява сечение на окръжност и окръжност
`isec_linecircle :l :k`	Изчислява сечение на линия и окръжност
`isec_circleline :c :l`	Изчислява сечение на окръжност и линия

II.8.3.7 Функции за точки върху обекти

Функциите за определяне на точки от геометричен обект са изнесени в отделен файл – `pointon.eli`, който се зарежда автоматично от библиотеката Geomland.

`pointon :x :y`	Определя точка от отсечка, линия, лъч, окръжност, елипса, хипербола, парабола или ъгъл, съответстваща на дадени число
`pointon? :x :y`	Проверява дали точка лежи на друга точка, отсечка, линия, лъч или окръжност
`pointon# :x :y`	Намира съответстващото число на разположението на точка върху отсечка, линия, лъч или окръжност

II.8.3.8 Етикети на геометрични обекти

Всички геометрични обект по подразбиране се визуализират без имена. Подпрограмите за създаване на етикети се съдържат във файла labels.eli, който се зарежда автоматично при зареждането на Geomland.

Етикетите са самостоятелни обекти, свързани чрез правила с обектите, към които са прикрепени. За целта се използват и няколко помощни обекта

`font :fontname :depth :bold :italic`
: Създава обект от тип тримерен шрифт на базата на име на шрифт, дълбочина на буквите, удебеленост и наклоненост

`text :string :font :center :radiusx :radiusy :focus :spin`
Създава обект, който показва текст на екрана с определен шрифт, разположение, размер и ориентация

`label :parent`

Създава обект от тип label етикет, който може да променя своите характеристики – текст, който се появява, шрифт, размер, цвят, отдалеченост от обекта, отместеност, ориентация и положение по протежение на обекта.

Подпрограмите за работа с етикети са:

`labeled "objname`	Създава етикет към обект
`def_position :obj`	Изчислява позицията по подразбиране на етикет спрямо обект
`label_position :obj :pos :dis`	Изчислява реалната позиция на етикет спрямо обект

Всички етикети се съхраняват в глобалния обект `labels` като полета с име името на обекта, към който са прикрепени.

Поради начина на представяне на текстовите изображения от графичната подсистема, е желателно обектите от тип `label` да се използват само по предназначение, понеже прекомерната им употреба може да доведе до по-бавно изпълнение на програмата.

II.8.3.9 Други функции, процедури и обекти

Цветовете в Geomland са обекти, които имат три елемента, съответно за червения, зеления и синия свят. Обект от тип цвят се създава като се използва класът `rgb`. Създаден е и домейнът `colors`, който съдържа 18 предварително дефинирани цвята: `black`, `maroon`, `green`, `olive`, `navy`, `purple`, `teal`, `gray`, `silver`, `red`, `lime`, `yellow`, `blue`, `fuchsia`, `aqua`, `ltgray`, `dkgray` и `white`.

Всеки от обектите може да се нарисува с произволен цвят (допустим от операционната система). Новосъздадените обекти са с подразбиращия се според типа на обекта цвят. Ако потребителят не укаже цвета на обект, ще се използва този по подразбиране.

В подпрограмите на потребителя често се налага да се проверява типът на геометричния обект. За целта могат да се използват следните функции, които работят и за примитивните типове данни.

`type :obj`	Връща типа на геометричен обект или променлива с стойност число, дума, списък или множество. При подходящо проектиране на нови обекти, те също могат да бъдат обхванати от тази функция.
`suptype :obj`	Връща надтипа на геометричен обект. В Geomland типът `line` е надтип на `segment` и `ray`.
`Oxy`	Създава координатна система
`lookat :x :y`	Променя гледната точка към равнината с обекти
`setmatrix :matrix`	Прилага дефинирана от потребителя проекционна матрица
`"scale`	Променлива, указваща текущо използваното мащабиране

`"csc`	Променлива, указващата координатите на центъра на прозореца за изобразяване на графични обекти
`"rotation`	Променлива, указваща текущо използваното завъртане
`"traces`	Променлива, съдържаща множество от всички следи на геометрични обекти
`locus "pntname "argname :step :count`	Създава множество от обекти от геометричното място на обект при промяна на параметър в определени граници
`polygon :points`	Създава отворен многоъгълник на базата на множество от точки.
`clrdr`	Изтриват се всички следи на обекти и екранът се пречертава

Освен описаните до момента обекти, в библиотеката Geomland са дефинирани и обектите range (интервал) и angle (ъгъл). Първият е базисен обект, съдържащ две числа – начало и край на интервал от градуси.

`range :fromangle :toangle`	Създава интервал от ъгли
`angle :initial :center :final`	Създава обект от тип ъгъл
`angle? :x`	Проверява дали обект е ъгъл
`angle3 :q :r :p`	Изчислява ъгъл, определен от 3 точки

II.8.4 Библиотека Graphix

Библиотеката Graphix е прекия интерфейс на система Elica към базовата графична система. Обектите и функционалните възможности на Graphix се използват в Geomland при изобразяването на геометричните обекти.

Всички обекти се дефинират с параметри, които да позволяват бързото им визуализиране и не целят близост с математическото представяне на обекта. Друга особеност на обектите е, че те се контролират по почти един и същ начин.

Библиотеката създава домейн `graphix`.

II.8.4.1 Параметри на обектите

Съществува един единствен набор от параметри, които се използват при визуализирането на всички обекти от Graphix. Естествено, някои обекти използват само част от предвидените параметри – тези, които са принципно характерни за него. Например, текстурите не трябва да се прилагат върху точки, защото няма да има никакъв ефект.

Някои от по-сложните параметри са представени като обекти.

`vector :x :y :z`	Създава вектор в общия смисъл на наредена тройка
`rgb :x :y :z`	Създава цветови вектор
`range :fromangle :toangle :mode`	Създава ъглов диапазон и начина на свързване на двата края. Ако `mode` е 0 не се прави свързване, ако е 1 се свързват пряко, а ако е 2 се свързват през центъра на фигурата, към която се прилага ъгловия диапазон.

Самите параметри на обектите контролират начина им на показване като определят цвета, дебелината, осветяването, текстурата и други характеристики. Нито един от параметрите не е задължителен. Ако някои параметър се използва от даден обект, и ако не е представен в конкретната инстанция на обекта, системата ще използва подразбиращата се стойност.

Ето и самите параметри:

`color` Обект от тип `rgb`. Определя цвета на точка, линия, отсечка, едноцветен многоъгълник. Дефинираният цвят се смесва с текстурата, ако има такава, и определя цвета на контурите на тримерен обект в режим на рисуване на ъглови точки или контури.

- `x` Червена компонента на цвета
- `y` Зелена компонента на цвета
- `z` Синя компонента на цвета

`width` Ширина на точка или линеен компонент

`smooth` Булев параметър, които определя колко фино да се рисуват обектите. Колкото по-фино се изобразяват, толкова по-бавно става самото изобразяване.

`stipple` Коефициент на мащабиране на едномерни шаблони

`pattern` Ако стойността е дума, тя се използва за едномерен 16-битов шаблон, който се прилага на линейни компоненти. Ако стойността е списък от думи, те се използват за двумерен 32x32-битов шаблон, който се прилага на планарни компоненти.

`precision` Прецизност при изобразяването на сложни обекти. Колкото е по-голяма стойността на параметъра, толкова повече точки ще формират контролните точки на обекта.

`mode` Режим на визуализиране на обект. Ако е 0 – показват се само контролните точки на обекта, ако е 1 – контурните линии, ако е 2 – повърхността на обекта се рисува като набор от плътни многоъгълници

`texture` Съдържа име на графичен файл със текстура. Поддържа се единствено формата BMP.

- `scale` Мащаб при прилагането на текстурата
- `spin` Завъртяност (в градуси) на текстурата спрямо повърхността, над която се прилага
- `smooth` Определя колко точно да се изчислява оцветяването при разтягане или свиване на текстурата. Ако е "true се прави прецизна интерполация, което забавя визуализирането.

`range` Обект от тип `range`, който определя каква част от многоъгълник или обемен обект трябва да се нарисува

- `fromangle` Начален ъгъл
- `toangle` Краен ъгъл
- `mode` Определя начина на свързване на началните и крайните точки

`light`	Булева стойност, която определя дали да се използва осветяване
`hollow`	Определя за някои обемни обекти, дали да бъдат затворени или не. Например, ако се отвори цилиндър, той ще се показва като тръба
`normal`	Ако е число, определя коефициент на мащабиране на нормалите (това оказва влияние на осветеността) за неполигонни компоненти. Ако е вектор – дефинира самата нормала за полигонални компоненти.
`multicolor`	Булева стойност, ако е "true всяка контурна точка на многоъгълник се рисува със собствен цвят. Ако многоъгълникът се запълва, се прави интерполация на цветовете за всички междинни точки.

II.8.4.2 Линейни обекти

Линейните обекти, дефинирани в Graphix са два вида – такива, които се дефинират директно, и такива, които се дефинират косвено.

Преките линейни обекти се дефинират с реалните им координати в пространството. Това означава, че за всеки от тях се знае кои точки в пространството заема, и позволява да се обработват лесно колизии между обектите.

`point :x :y :z`	Създава точка на базата на координатите й
`segment :initial :final`	Създава отсечка, свързваща две точки
`line :initial :final`	Създава линия, минаваща през две точки
`ray :initial :final`	Създава лъч, започващ от една точка и минаващ през друга

Косвените линейни обекти се създават в тяхната нормализирана форма, но в последствие се прилагат различни трансформации – транслация, мащабиране, завъртане и ориентиране. Параметърът `center` определя транслацията, `spin` определя ротацията спрямо нормалата на обекта, а `focus` определя посоката на нормалата.

`square :center :radius :focus :spin`	Създава квадрат
`rectangle :center :radiusx :radiusy :focus :spin`	Създава правоъгълник
`polygon :points`	Създава многоъгълник

II.8.4.3 Криви от втора степен

Кривите от втора степен са само косвени обекти. Те се създават в нормализиран вид и инициализационните параметри определят трансформациите, които привеждат кривата в желания вид. Поддържаните криви от втора степен са окръжност, елипса, фрагмент от елипса, парабола и хипербола.

`circle :center :radius :focus`	Създава окръжност
`ellipse :center :radiusx :radiusy :focus :spin`	Създава елипса
`subellipse :center :radiusx :radiusy :range :focus`	Създава фрагмент от елипса

`parabola :center :radius :focus :spin` Създава парабола

`hyperbola :center :radius :focus :spin` Създава хипербола

II.8.4.4 Тримерни обекти

Тримерните обекти в библиотеката Graphix са също само косвени. В библиотеката са дефинирани следните обекти: куб, паралелепипед, цилиндър, конус, пресечен конус и сфера.

`cube :center :radius :focus :spin` Създава куб

`paralelogram :center :radiusx :radiusy :radiusz :focus :spin` Създава паралелепипед

`cylinder :center :radiusx :radiusy :radiusz :focus` Създава цилиндър

`cone :center :radiusx :radiusy :radiusz :focus :spin` Създава конус

`cutcone :center :radiusx :radiusy :radiusz :ratio` Създава пресечен конус

`sphere :center :radius :focus :spin` Създава сфера

II.8.4.5 Други функции, процедури и обекти

Освен представените дефиниции на обекти, библиотеката Graphix създава някои други класове за работа с текстове и текстури, процедури за промяна на гледната точка и на перспективата и една функция за промяна стила на обект.

`font :fontname :depth :bold :italic` Създава обект-шрифт със зададена тримерност, дебелина и наклон

`text :string :font :center :radiusx :radiusy` Създава тримерен текст, визуализиран на екрана

`texture :texturename` Създава текстура, която в последствие може да се наложи на различни двумерни и тримерни обекти

`lookat :eye :focus :forehead` Променя гледната точка в пространството. `Eye` дефинира къде се намира гледната точка, `focus` определя в каква посока се гледа, а `forehead` е вектор, указващ посоката нагоре.

`setmatrix :matrix` Прилага дефинирана от потребителя проекционна матрица

`styled :obj :attr` Създава или променя стиловите характеристики на обект

II.8.5 Библиотека IO

Библиотеката за вход и изход IO съдържа дефинициите на функции и процедури за четене от клавиатурата. Всички дефиниции са в рамките на домейна `io`.

`getkeybuffer`	Функция от ниско ниво за извличане на текущия системно дефиниран буфер
`readkey?`	Проверява дали е натиснат клавиш
`readkey`	Ако има натиснат клавиш, тази функция го връща, а ако няма – функцията изчаква да се въведе.
`readword`	Функция за четене на дума (последователност от букви, завършваща с CR)
`clearkey`	Изчиства текущия буфер на клавиатурата
`printkey :x`	Отпечатва буква на екрана. Клавишът Backspace връща назад и изтрива по една буква.
`wordtolist :x`	Конвертира дума в списък (използва вградения конвертор на система Elica)
`readline`	Прочита ред от клавиатурата (последователност от букви, конвертирана в списък)

В библиотеката са дефинирани няколко константи на някои системни "букви" – `space`, `escape`, `return` и `backspace`.

II.8.6 Библиотека Turtle

Съществена характеристика за всяка графична реализация на езика за програмиране Logo е, че се поддържа в една или друга костенуркова графика. Този метод на рисуване се базира на обект, наречен *костенурка*, който може да бъде командван като робот (всъщност първите реализации на костенуркова графика са били именно с робот, а компютърният модел се е появил по-късно).

Библиотеката дефинира един единствен обект от тип `turtle`, в който е съсредоточена цялата логика реализацията на костенуркова графика. Това, че костенурката е обект, означава, че потребителят може да си създаде неограничен брой костенурки и всяка от тях да е с независими от другите параметри.

Създаването на костенурка става с команда от вида `make "a turtle`, без да се задават аргументи. В последствие потребителят може да променя характеристиките на костенурката.

Следва списък от полетата и методите на обекта `turtle`.

`"pos`	Текущи координати на костенурката
`"ang`	Направление (насоченост)
`"pen`	Флаг дали е в режим на рисуване или само на местене
`"col`	Цвят на костенурката
`"showit`	Флаг дали костенурката е показана на екрана
`home`	Инициализира костенурката
`forward :dist`	Премества костенурката напред
`backward :dist`	Премества костенурката назад
`left :a`	Завърта костенурката наляво
`right :a`	Завърта костенурката надясно
`show`	Показва костенурката

`hide`	Скрива я
`color :x`	Променя цвета на костенурката
`penup`	Превключва в режим на местене
`pendown`	Превключва в режим на местене с рисуване
`setx :x`	Установява абсцисата на костенурката
`sety :y`	Установява ординатата на костенурката
`setxy :x :y`	Променя местоположението на костенурката

II.8.7 Библиотека Win

Библиотеката `Win` е една от най-малките библиотеки. В нея има дефиниран само един обект – `button`. С този обект могат да се създават бутони в рамките на прозореца за рисуване, като на всеки бутон се определя положението му на екрана, размерите и изписаният текст. `Button` е дефиниран в домейна `win`.

II.8.8 Библиотека Astro

Библиотеката `Astro` съдържа дефинициите на различни обекти, които могат да се използват за симулиране на модели от небесна механика. Могат да се дефинират различни обекти-планети с техните параметри (координати, маса, вектор-скорост и вектор-ускорение) и да се симулира тяхното гравитационно взаимодействие.

`position :x :y`	Конструктор на обект радиус-вектор
`velocity :x :y`	Конструктор на обект скорост-вектор
`acceleration :x :y`	Конструктор на обект ускорение-вектор
`mass :x`	Шаблонна функция (не извършва никаква дейност, а само подобрява описанието на програмата)
`with :x`	Шаблонна функция
`gravity :p1 :p2`	Гравитационно влияние между планети
`planet :m :p :v`	Създава обект от тип планета

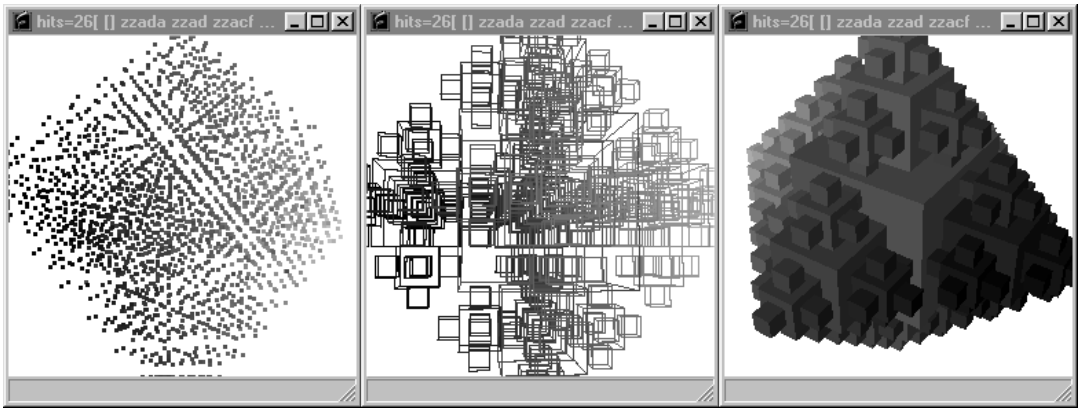
II.9 Разработени примери

Съществуването на проект за език и неговата реализация, не означават, че той е практически приложим. За да се провери реално работата на подобна система – не само дали работи по принцип както е специфицирано, а дали е подходяща за решаването на реални примери, е необходимо да се направят множество примери.

Съществуването на подобни примери се използва не само за да се провери годността на системата, а и за да се демонстрират нейните функционални възможности.

След инсталацията на система Elica, в поддиректория samples се копират различни по сложност и големина примерни програми. В настоящата глава ще бъдат показани някои от най-интересните примери (някои от тях са разположени и на сайта на системата).

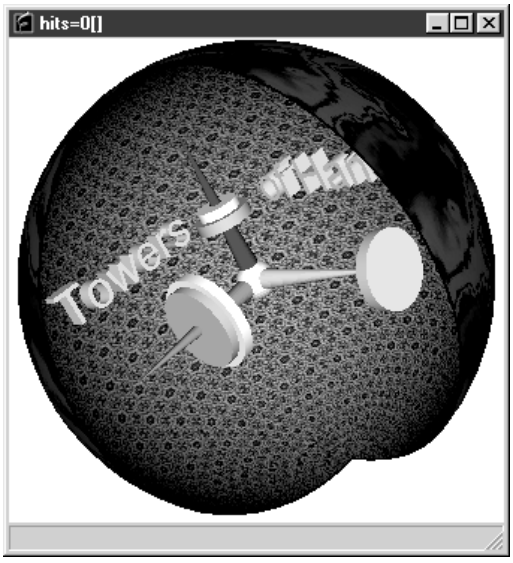
II.9.1 Фрактал от кубове



Фракталът от кубове е една красива рекурсивна фигура. Започва се от куб и на всяка от стените му се поставя по един по-малък куб. На петте свободни стени на по-малките кубове се поставят още по-малки и т.н.

Примерът създава вложеност от 4-то ниво, като по време на създаване цялата конфигурация се върти в пространството. Самото създаване не става изведнъж, а плавно. Първоначално се натрупват кубовете като всеки от тях е представен с върхните си точки. След като се натрупат всичките четири нива кубчетата започват да се променят – появяват им се ръбовете. След като завърши и този процес, те започват да се оцветяват, като цвета на всяко кубче зависи от пространствените му координати.

На трите кадъра по-горе са показани три от фазите на демонстрацията. След като те завършват, кубчетата започват едно по едно да се премахват, докато останат шестте най-върхни, които се свързват с отсечки.



II.9.2 Ханойски кули

Задачата за ханойските кули е една от най-традиционните в програмирането и се използва за демонстрация на рекурсивните процедури. Самата реализация на алгоритъма на Ханойските кули не представлява особен интерес, затова в този пример е наблегнато на визуалното представяне на задачата.

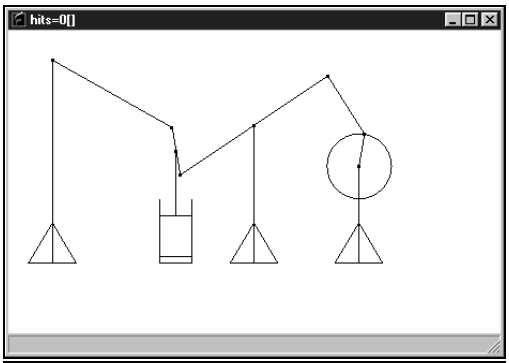
След пускането на примера се появява

мраморна сфера, която се върти на екрана. Въртенето продължава до края на демонстрацията.

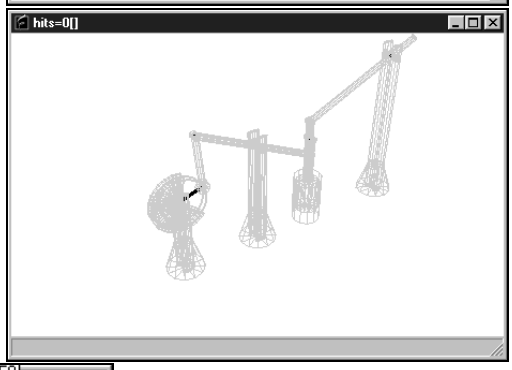
След известно време в сферата започва да се отваря процеп, а във вътрешността се забелязва система от три взаимно перпендикулярни оси. Това са осите, които се използват в задачата. В процеса на демонстрацията се решават последователно задачите с три диска, с четири, с пет и с шест. Дисковете се нанизват на осите, а при преместването на други оси те се отнизват, прелитат до съответната ос и се нанизват на нея. Движенията на дисковете са доста сложни, понеже се получават от наслагването на няколко по-прости движения:

- прелитането от една ос до друга;
- преориентирането в пространството, понеже осите са перпендикулярни;
- глобално въртене на цялата сфера с всички обекти в нея.

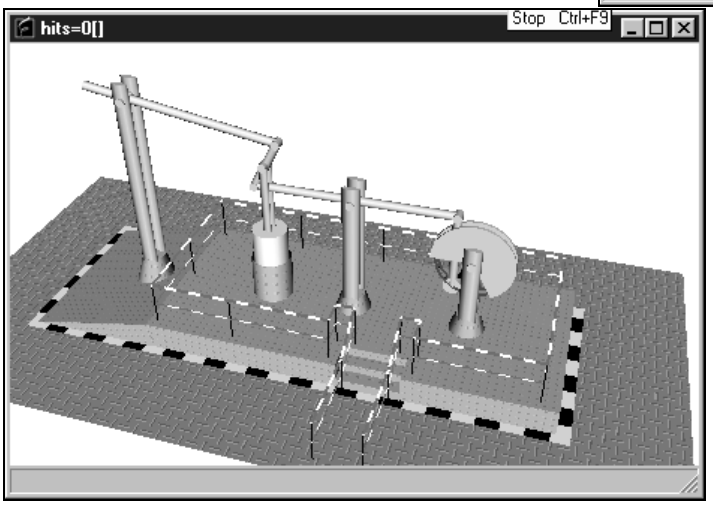
II.9.3 Парна машина



Това е най-красивия и най-сложния от предоставените примери. Идеята е взета от реализация на парна машина в система Геомландия. На първия екран е показана реализация, подобна на оригиналната.



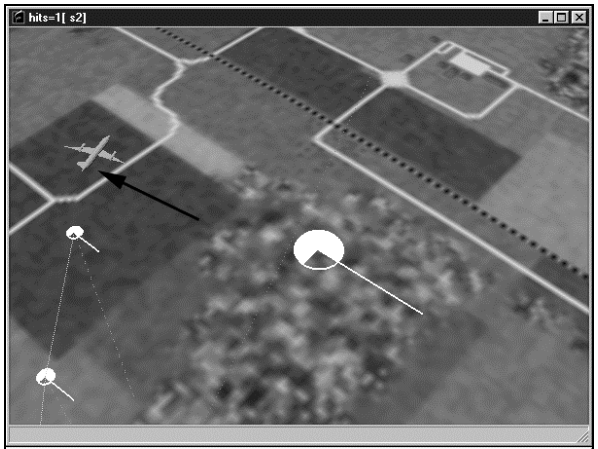
Особеното в примера на парната машина е, че постепенно се преминава в по-висока степен на реалистичност. Първоначалната фаза е плоска равнина реализация, която постепенно става обемна, като се запазва в една равнина. Продължавайки да работи, машината и елементите й започват да придобиват обемност, като тази, показана на втория екран. В последствие обемните елементи се оцветяват и им се наслагат различни по цвят текстури. След като и тази стъпка завърши, без да спира работата си, започват да се рисуват допълнителните елементи – постамент, предпазна зона, стълби и т.н.



Особеността на реализацията на парната машина е, че елементите, които са използвани и в плоския, и в тримерния случай, са все обекти, създадени в Elica, за които е описан начина на преминаването им от двумерни в обемни.

На последния кадър е показана парната машина в крайната фаза. Дори и в този си сложен вид, тя продължава да работи с достатъчна скорост, за да се приемат всички движения като непрекъснати.

От самото начало на тримерност на примера, до края цялата парна машина освен че работи, и се върти, за да може да бъде огледана от всички страни.



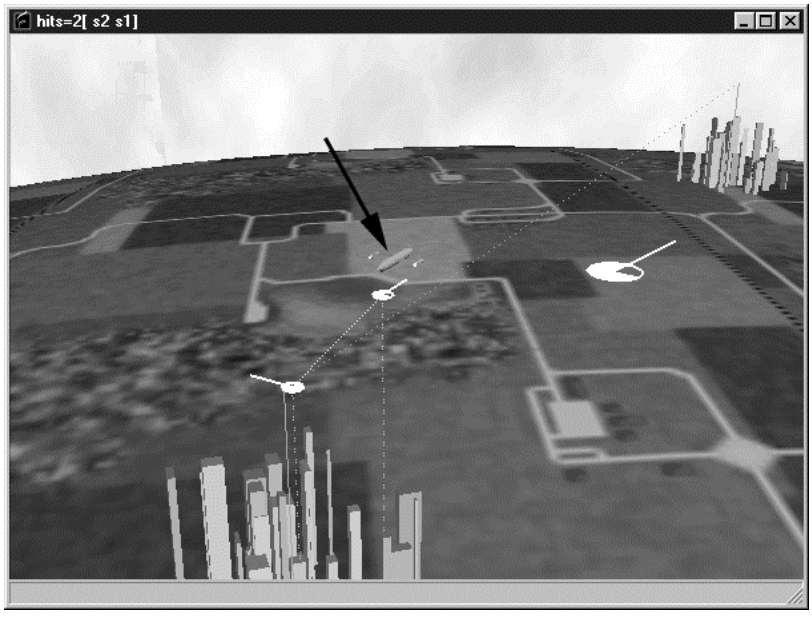
II.9.4 Полет на самолет

Един от най-естествените примери, които са реализирани в система Elica е симулацията на полет на самолет.

Сцената представлява земя, чиято закривеност поради големите разстояния е ясно забележима.

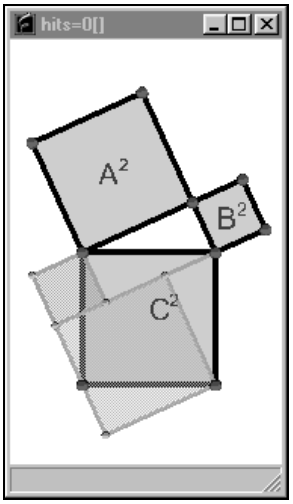
Върху земята са разположени пътища, ниви, гори, постройки, езера и тяхното показване в перспектива следва физичните закони.

Показани са и два града, всеки от които съставен от множество различни небостъргачи и по-ниски сгради. Самолетът излита от един такъв град и целта е да се приземи в друг. Началната скорост и посока са така определени, че той да пристигне в точното време на точното място. И на трите екрана позицията на самолета е показана с черна стрелка. По време на полета самолетът е подложен на влиянието на вятъра, който духа с променлива посока. Специален метеорологичен маркер е поставен на земята и показва текущите параметри на вятъра. Вятърът променя положението на самолета. За да може по-лесно да се следи неговата траектория и влиянието на вятъра в отделните фази, периодично се рисува изминатия път като върху него са отбелязани метеорологичните маркери, които са били валидни по време на тази част на полета.



С тънка линия е отбелязан "идеалния" полет – този, който се извършва при тихо време, без никакви външния влияния. Идеята на примера възникна след дискусии относно проекта "Navigational Vectors", разработен от Center for Improved Engineering and Science Education, Stevens University (www.ciese.org)

II.9.5 Теорема на Питагор

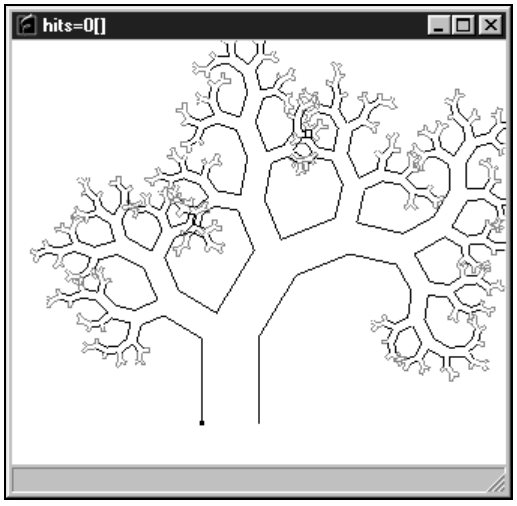


Теоремата на Питагор е една от най-често използваните в училищният курс по геометрия. Самата теорема има стотина различни доказателства, някои от които са чисто алгебрични, а други – чисто геометрични.

Примерът за теоремата на Питагор представя геометрично решение, предложено от автора, без претенции за оригиналност. Доказателството е конструктивно - чрез продължаване на стените на квадратите, построени върху катетите на триъгълника, се конструират еднакви на тях квадрати, от които след подходящо разрязване се сглобява квадрата на хипотенузата.

Отначало се показва каква допълнителна конструкция се прави. Триъгълникът се променя динамично, а конструкцията се изменя плавно. Втората фаза показва как да се направи разрязването и как да се насложат отделните части, така, че от двата по-малки квадрата да се получи по-големия.

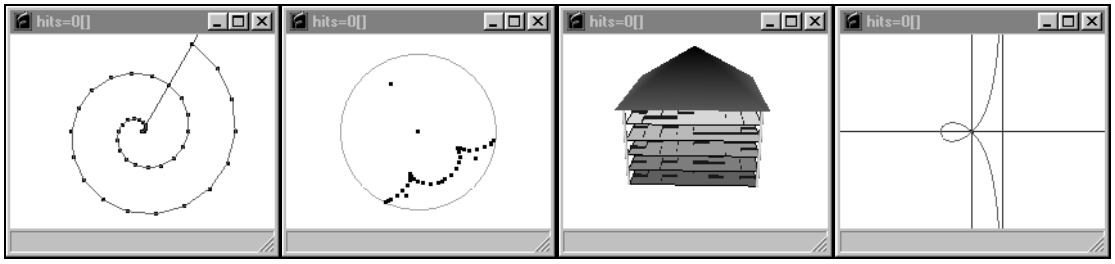
II.9.6 Питагорово дърво



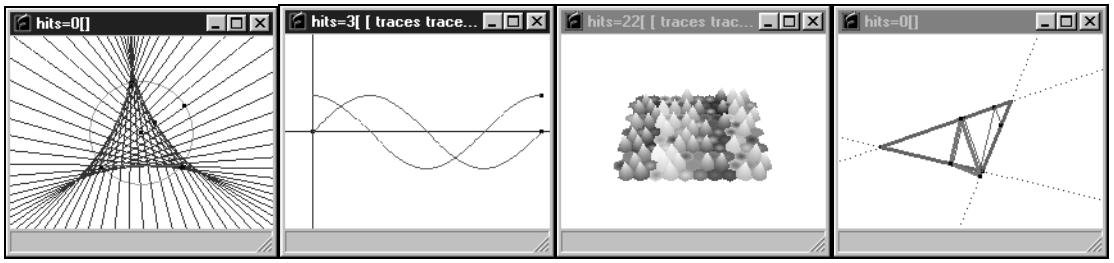
Един от най-илюстративните примери, съчетаващ рекурсия, изкуство и компютърна графика е създаването на Питагоровото дърво. Това е вид рекурсивен фрактал, който има формата на дърво. Името е Питагорово, защото дървото се конструира от правоъгълни триъгълници и квадрати върху катетите. Ако правоъгълните триъгълници са с различни ъгли, то дървото ще е несиметрично, както е в случая. За този пример е използвана костенурковата графика в Elica, благодарение на което текстът на програмата на примера е къс, елегантен и лесно разбираем.

II.9.7 Галерия

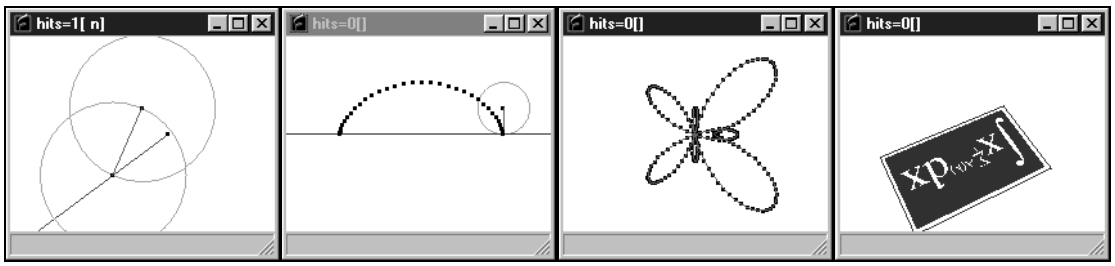
В следващата галерия са показани някои от другите примери, разпространявани с Elica. На първият ред са показани примери на архимедова спирала, конструкцията "Крепост", тримерна къща и строфоида.



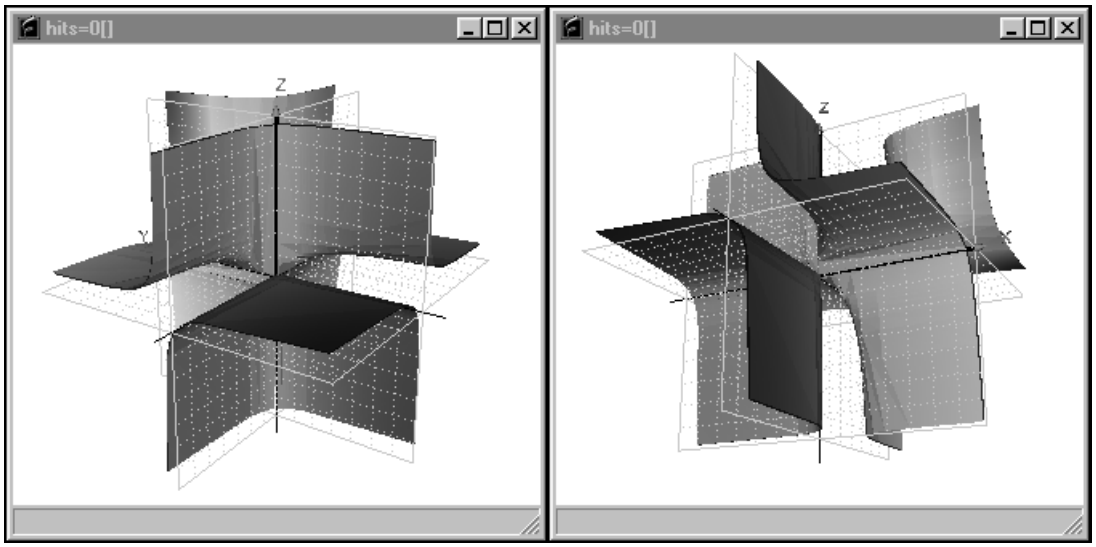
На следващия ред са показани сноп прави, построени по определен начин, графиките на функциите sin и cos, мозаечна конструкция от пирамиди с шестоъгълни основи и решението на задачата на Рижик.



На третият ред от галерията са показани други четири примера: интерактивна геометрична конструкция, построяване на циклоида, рисуване на "пеперуда" и визуализиране на формула.



На следващия ред е показана въведената от Г. Станилов повърхнина, определена от уравнението: $\frac{1}{x}+\frac{1}{y}+\frac{1}{z}=1$ [24]. Изненадата при построяването на примера бе, че графичния резултат е крайно интересен и неочакван, особено като се има предвид, че преди използването на система Elica, същият пример е бил пробван на други системи, където или е било невъзможно да се нарисува повърхнината или пък това е ставало доста трудно.



III РЕАЛИЗАЦИЯ НА СИСТЕМАТА

III.1 Вътрешно представяне на Elica Logo данните

Проектирането и реализирането на език за програмиране изискват внимателно разглеждане на въпроса как да се представят вътрешно данните, които се обработват. Известно е, че колкото по-прости са данните, толкова по-сложни са алгоритмите, които ги обработват. Обратното също е вярно, ако целта е да се реализират прости алгоритми, най-често "пропусната" сложност рефлектира върху данните.

Концепцията при проектирането на вътрешното представяне на данните в система Elica следва от глобалната унификацията в езика, при която различните метаобекти са взаимно заменяеми.

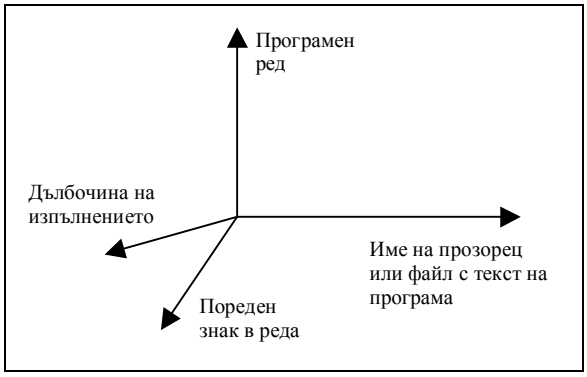
III.1.1 Позиция в програмата по време на изпълнение

В редица случаи се налага да се знае точната позиция в програмата по време на изпълнението й. Някои от основните случаи са:

- при трасиране на програма, за да може потребителят да следи коя команда се изпълнява
- при грешка, за да може системата за покаже на потребителя къде е станала грешката.

Текущата информация за позицията се съхранява в структурата TPosition, съдържаща четири полета. Система Elica може да поддържа работа с няколко прозореца с програмен текст, затова, позицията включва името на прозореца с програмния текст, в който се намира дадената позиция. В рамките на вече фиксирания прозорец, позицията се доуточнява като се зададе номера на реда и позицията на символа в реда. Номерът на реда се използва при трасиране на програмата. Позицията на символа служи за точно определяне на мястото, където е текущата позиция.

Фигура III-1 Пространствено-времева координатна система



Всички описани до момента характеристики определят физически "координати" в активното пространството на програмните текстове. За целите на трасирането, е необходимо да се съхранява и още една характеристика, която е динамична и зависи от изпълнението на програмата. Това е текущото ниво на вложеност на изпълнението. Ако една процедура се изпълнява рекурсивно, всяко ново извикване минава през едни и същи позиции, но с различни нива на вложеност.

```
TPosition = record
   Caption : string;    // Заглавие на прозореца
   Line    : integer;   // Ред
   CharPos : integer;   // Позиция на символа
   Level   : integer;   // Ниво на вложеност
end;
```

Предлаганата структура дефинира еднозначно пространствено-времеви координати. Координатите са четири мерни – три измерения са отделени на пространствеността, т.е. физическите пространствени координати, а едно измерение е времевото.

III.1.2 Структура на атомите

III.1.2.1 Реализирани предишни варианти

В процеса на разработване на предишните версии на система Elica, са били изпробвани няколко начина на представяне на данните в паметта, понеже само с теоретично разглеждане на проблема не винаги може да се намери оптималното решение. Проектирането на структурата се обуславя главно от няколко взаимно противоречащи се фактора – компактност на структурите, лекота на използването и гъвкавост при надстройването им.

В исторически план, реализирани са следните структури: уникални записи, стандартни записи, обекти в йерархия и унифициран обект без йерархия.

- *Уникалните записи* са били използвани в най-първите реализации на системата. На практика, те означават, че за всеки обект е проектиран собствен запис от данни (става въпрос за запис в смисъла на Pascal, който се дефинира със служебната дума *record*). Уникалните записи са били прилагани към всеки от обектите. Поради това, не е било възможно да се създават нови обекти, без да се компилира цялата система – всички обекти са били вградени в нея.

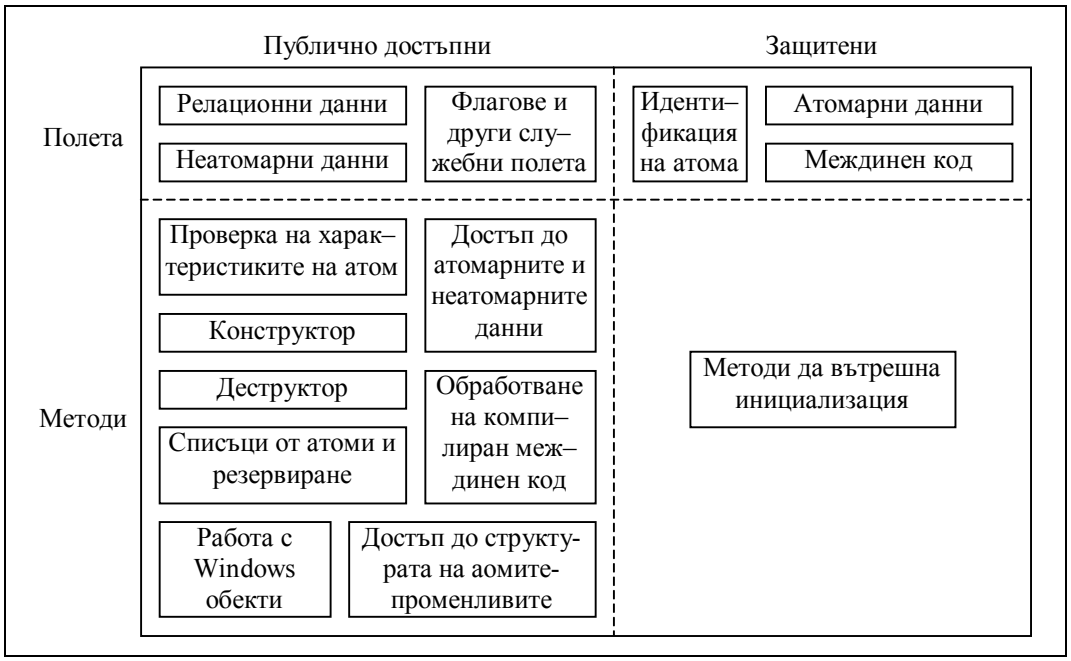
- В последствие, това ограничение е било избегнато с въвеждането на *стандартните записи*. Тези записи не съответстват на различните обекти в системата, а на мета обектите. Общата част на записите се "наследява" в посока от по-прости към по-сложни записи. Избраните основни типове стандартни записи са давали възможност да се създават динамично нови обекти. Основните разлики са в това, че в стандартните записи могат да се съхраняват числа, думи, списъци и променливи. На това място, в теоретичните разработки на концепциите на Elica се е стигнало до идеята за представянето на обектите като обикновени променливи. Ето защо, в записите за променливи е можело да се съхраняват произволни обекти.

- Въвеждането на обектно-ориентираното програмиране създаде условия за "конвертиране" на записите в обекти. Освен йерархията, пораждаща *йерархични обекти*, създаването и капсуловането на методите за работа с различните блокове памет в самите обекти, също е допринесло за съществено по-добро структуриране и изчистване на механизмите за работа с метаобекти. За съжаление, този начин на представяне на данните не е последният, въпреки, че е най-удобният. Основният проблем е, че използването на обектно-ориентирано програмиране изчиства концепциите при създаването на програма, но забавя производителността. Проблемът със скоростта е единственият, който наложи реализацията на друга вътрешна структура на атомите.

- Ускоряването е постигнато с унифициране на различните видове обекти в един единствен, който не наследява никого и не се наследява от никого. Използването на *унифицирания нейерархичен обект* е начина, който е избран за крайната реализация

на вътрешната структура на атомите в Elica. На следващата схема е показана общата структура на атомите в системата.

Фигура III-2 Обща структура на атомите



III.1.2.2 Вътрешна структура на атомите

Окончателната вътрешна структура на атомите е унифицирания нейерархичен обект. На практика това е един единствен клас, чиито инстанции могат да бъдат както числови атоми, така и текстови, списъчни, или пък да съдържат променливи, процедури, функции, оператори, масиви и домейни.

Дефиницията на структурата е планирана да се използва от външни за системата модули (най-често това са външните библиотеки), затова е използвана конвенцията за достъпност до полетата и методите на класа.

Вътрешните полета са декларирани като private. Те са скрити за всички други модули, които използват атоми. Това е направено с цел да се предостави гъвкавост при промяна на вътрешната структура. Достъпът до тези защитени полета се извършва с функции и ако полетата в бъдеще се променят, а функциите се запазят, това няма да наложи преправяне на останалите части от програмата.

III.1.2.2.1 Защитени полета

Защитените полета на атомите представляват най-вътрешните данни на атома, които са достъпни само от методите на самия атом. По своята роля те се делят на даннови и служебни. Полетата за данни са две – едното може да съдържа текст, а другото – число с плаваща точка. За да се унифицира структурата на атома, се налага да се дефинират и двете полета, независимо от това, че във всеки момент се използва най-много едно от тях.

```
_Name  : string; // текст с общо предназначение
_Float : float;  // реално число
```

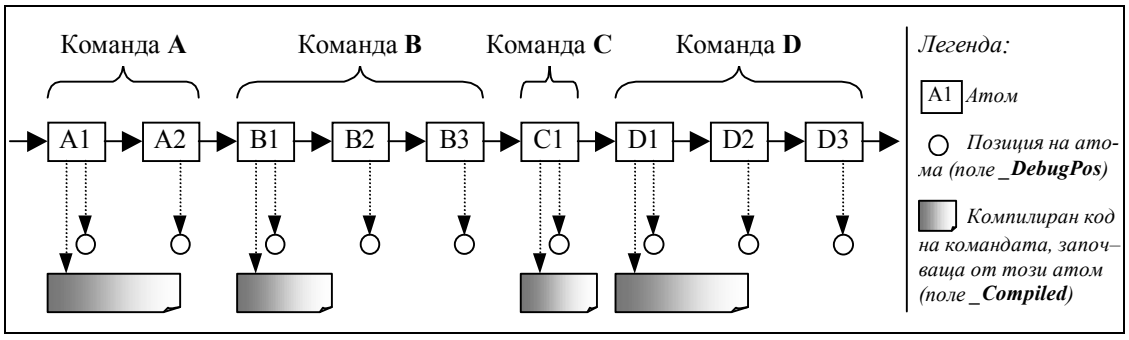
Полетата за служебна информация се използват по време на работа на системата за поддържане целостта на данните в паметта. Всички атоми в системата са свързани в

няколко списъка. Затова едно от системните полета е _NextAtom. То осъществява еднопосочната връзка към следващия атом от същия списък. Двата най-основни списъка в Elica са списъкът на заетите атоми и списъкът на свободните атоми. Те ще бъдат разгледани в точка "Структура на контекста". Следващото поле съдържа броя референции към този атом. Тези референции определят само изричните връзки, създадени от ядрото на системата, а не всички връзки. Основната роля на полето е да указва на подсистемата за управление на паметта, кои атоми не трябва да се освобождават, в случай че никой друг атом не сочи към тях. Тази защита се налага поради факта, че когато се изпълнява списък от команди, в определен момент може да се окаже, че той не е свързан с нито един атом и по тази причина подлежи на освобождаване. Но понеже в момента този списък се изпълнява, временно той се защитава и едва след изпълнението на всички команди от него става възможно изтриването му.

```
_NextAtom : TAtom;    // Връзка към следващия атом от списъка
_Ref      : integer; // Брой изрични референции към атома
```

Последните две от защитените полета контролират изпълнението на програмата. Те се попълват само когато съдържат атоми, които са част от текста на програмата. В едното поле, _Compiled, ядрото на системата разполага компилирания междинен код на текущата команда, а в полето _DebugPos се поместват данни за позицията на атома в пространствено-времевата схема на програмата.

Фигура III-3 Позиция на атом и компилиран междинен код



На фигурата е показано кога се попълват _Compiled и _DebugPos. Полето _DebugPos се попълва за всеки атом, който е създаден пряко от текста на изпълняваната програма. Най-често това се извършва в момента на конвертиране на текст в списък от команди. На фигурата _DebugPos данните са отбелязани с малки кръгчета.

Полето _Compiled се попълва на ниво команда. Затова на фигурата са показани само четири блока, съдържащи компилирана команда до междинен код. Дефиницията на полетата е следната:

```
_Compiled : TList;     // Компилиран междинен код
_DebugPos : PPosition; // Текуща позиция в програмата
```

III.1.2.2.2 Публично достъпни полета

Описаните полета не са достатъчни за създаване на функционални атоми. Останалите необходими полета се декларират като публично достъпни. Това означава, че те ще могат да се ползват пряко и от разработчиците на външни библиотеки.

Две от най-важните полета са _Id и _Flags. В първото се записва стойност, която определя типа на атома. Проверката на типа на атома е единствения начин, с който се разбира кои полета от атома за какво да се използват, а също, и кои полета са валидни и кои не. Причината за това е, че в някои типове атоми някои полета не се използват, а други полета имат друг смисъл.

Таблица III-1 Вътрешни типове на атомите

Тип	Значение
nAtom	Числов атом
wAtom	Текстов атом
lAtom	Атом, съдържащ звено от обикновен списък […]
eAtom	Атом, съдържащ звено от списък-израз (…)
oAtom	Обектен атом – използва се само за Windows обекти
vAtom	Атом, съдържащ променлива
mAtom	Атом-памет, съдържащ произволни по размер и съдържание данни

В полето _Flags се съхраняват не само флаговете, достъпни на потребителя, но и вътрешните флагове. Всеки флаг е отделен бит. В следващата таблица са описани текущо дефинираните служебни флагове.

Таблица III-2 Системни флагове на атомите

Име	Значение
OBJECT	Флаг, определящ дали атомът е атом на променлива, създадена с командата `OB`
ACTIVE	За атомите, съдържащи текущо активни променливи, този бит е 1
CONTEXT	Този флаг се установява на 1, когато атомът е и контекстна променлива (т.е. базова променлива на активен контекст)
LOCKED	Вътрешно-системно заключване на процедури, в периода на изпълнението им
GARBAGE	Използва се при прочистване на паметта от модула за управление на паметта
EXPANDED	Флагът определя дали в прозореца на променливите променливата в този атом е разпакетирана (т.е. показани са и подчинените променливи)

Потребителските флагове също определят поведението на променливите, защото те са приложими само за атомите на променливите. Потребителят има пряк достъп за четене и писане в тези флагове.

Таблица III-3 Потребителски типове на атомите

Име	Значение
USERLOCK	Потребителско заключване на идентификатор, за защита на промяната му с командите `MAKE` или `OB`
USERDEF	При 1, системата запомня дефиницията на променливата
USERHIST	Когато е установен, този флаг предизвиква създаване на историята на променлива
USERSYNTAX	Флагът е 1, когато системата трябва да създава изрични синтактични връзки между променлива и дефиниращите променливи
HIDDEN	С този флаг може да се контролира дали променливите с графичен образ да се изобразява на екрана или образа й да се игнорира
USERDOMAIN	За създаване на домейни, този флаг трябва да се 1

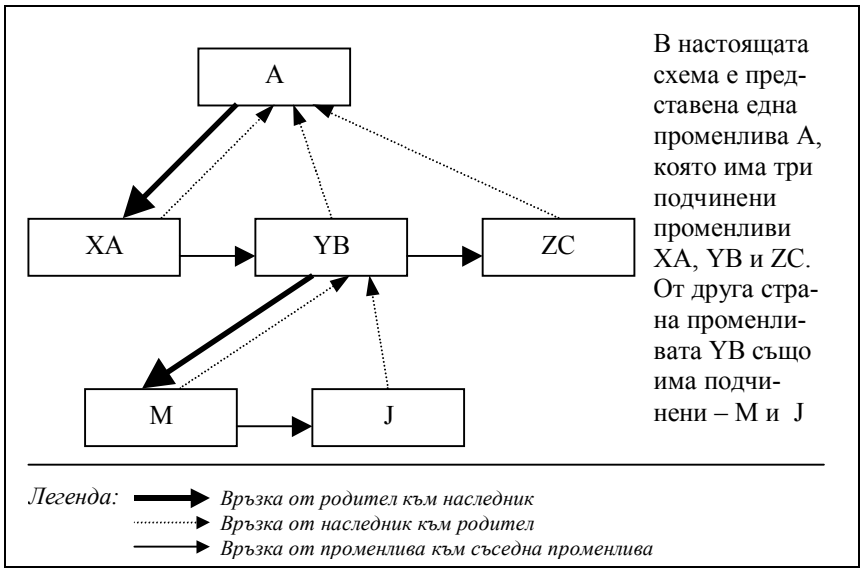
Реализирането на списъци в Elica става по стандартният начин – списъкът е линеен еднопосочен списък от звена, всяко от които сочи към следващото звено и към съответен елемент от списъка. Връзките се съхраняват в полетата _CAR и _CDR. Имената са наследени и като име, и като семантика от далечния прародител на LOGO – езика LISP.

```
_CAR : TAtom; // Указател към елемент от списъка
_CDR : TAtom; // Указател към следващото звено
```

Тези полета се използват и за други видове атоми. За Windows обекти в _CAR се съхранява манипулаторът. За променливи там се записва атомът със стойността на променливата. За атоми-памет в _CAR се записва указател към паметта, а в _CDR адресът на функцията, която освобождава заетата памет.

Поменливите в Elica (т.е. блоковете памет, които ги съдържат) могат да са в хоризонтална и вертикална релация с други променливи (блокове). Полетата в атомите, отговорни за релационните връзки, са Parent, FirstChild и Brother. В тях се записват, съответно родителския блок и първият подчинен блок (заедно те определят вертикалната йерархия) и следващия блок, който има същият родител (хоризонтална йерархия)

Фигура III-4 Релационна връзка между променливите



И трите релационни полета се използват само при връзки между променливите. Съществува една главна променлива, родител на всички останали. Тя е единствената променлива, която няма родител.

```
Parent     : TAtom; // Родителски атом
FirstChild : TAtom; // Подчинен атом
Brother    : TAtom; // Съседен атом
```

Всеки родител сочи само към една от подчинените си променливи – първата. Останалите подчинени променливи са достъпни през полето за съседство Brother. Обратната връзка, към родителя се осъществява от всяка подчинена променлива.

Последните три публично достъпни полета са за ускорен достъп до определени подчинени променливи и за определяне възрастта на променливата. За обектите в Elica често се налага да се проверяват дали съдържат променливи с имена OnDrawImage и OnChange.

```
OnDrawImage : TAtom;
OnChange    : TAtom;
```

В тези две полета винаги се пази пряк указател към подчинените променливи със съответните имена. Ускорението, което се постига по този начин, е значително, независимо от допълнителната работа, която трябва да се извършва за поддържане актуалността на полетата.

```
TimeStamp    : integer;
```

В полето TimeStamp се съдържа информация за "възрастта" на променлива. Възрастта не се измерва в някакви времеви единици, а съдържа стойността на глобален самоувеличаващ се брояч. Възрастта на променливите се използва при правилата, за да може да се следи, кои са променяни след започване действието на правилата и кои – не.

III.1.2.3 Методи

В обектите на атомите в Elica са дефинирани над 60 метода, които обезпечават всички необходими действия с атоми. Част от по-вътрешните методи, особено тези за инициализация, са защитени и не могат да се ползват извън атомите. Всички останали методи са публично достъпни.

III.1.2.3.1 Защитени методи

Основните действия на защитените методи е инициализацията на атомите. Методите Free и Initialize се използват за изчистване на всички "висящи" данни към атома. Създаването на атом от определен вид изисква подходящото му инициализиране. За целта са създадени методите InitializeAsList, InitializeAsWord, InitializeAsNumb, InitializeAsVar, InitializeAsObj и InitializeAsMem, които инициализират атома.

III.1.2.3.2 Публични методи за инициализация

Публично достъпните методи са много повече и по-разнообразни. Те се използват за всички аспекти на управление на атомите – създаване, модифициране, премахване, извличане на характеристики и т.н.

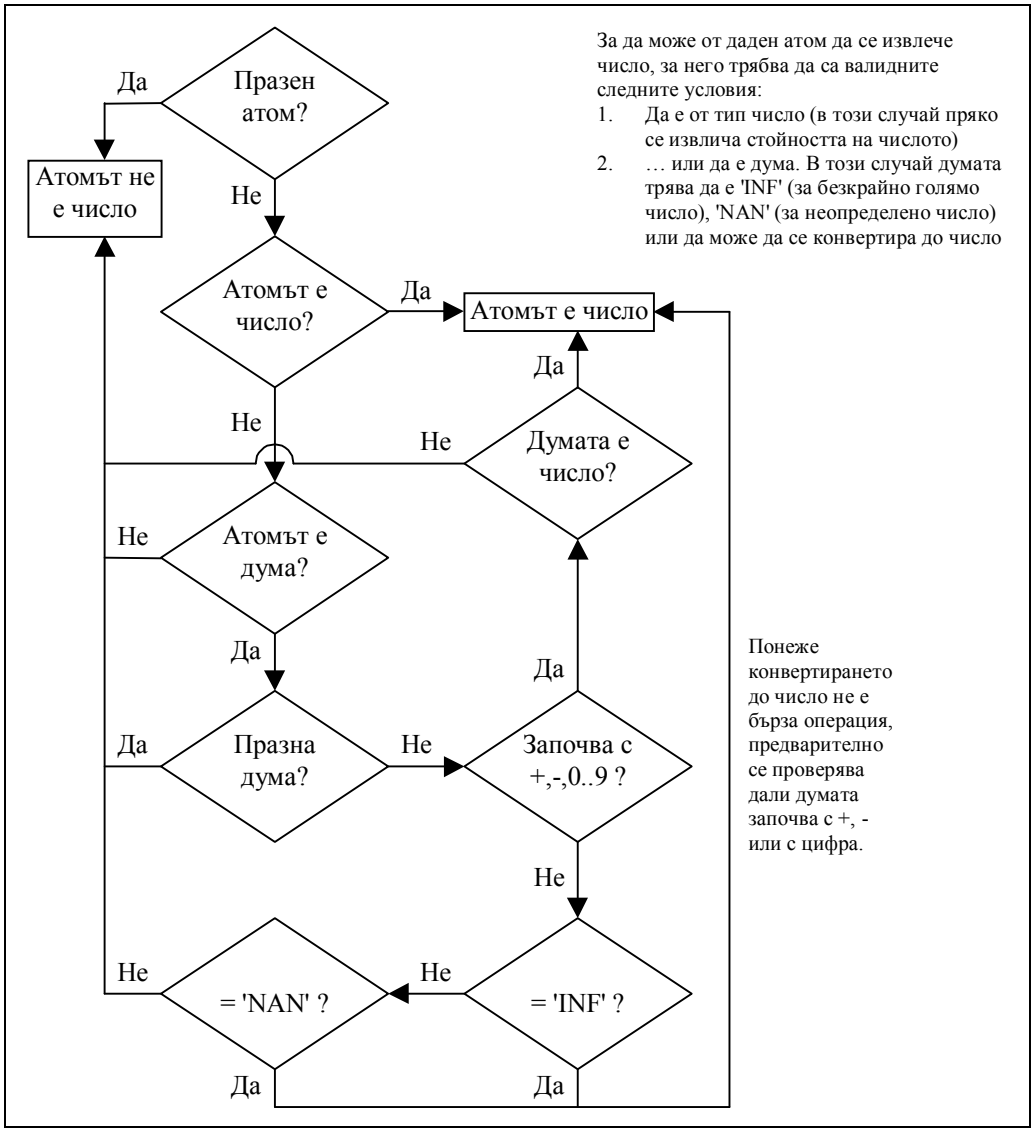
Основните методи за създаване на атоми на ниско ниво са Create и Destroy. Те са конструктора и деструктора на обект от тип атом. Резервирането на атом става с метода Use, който увеличава стойността на полето _Ref с единица. В резултат, атомът няма да се освободи при изчистване на паметта, независимо дали се ползва от други атоми или не. Освобождаването на атом става с обратния метод Deuse. Двата метода могат да се прилагат многократно. Deuse рядко трябва да се използва директно. За предпочитане е да се ползват FreeAtom и FreeNextAtom, които осъществяват необходимите проверки и прехвърляния на атома от един списък в друг.

Ядрото на системата поддържа няколко основни списъка с атоми, които ще бъдат описани по-късно. Единият от списъците е този на свободните атоми. Методът GetFreeAtom се използва за извличане на първия свободен атом от този списък и изключването му от него. Съществува и още един метод – FreeObject. Той се използва за освобождаване на ресурсите, заети от Windows обекти или от атоми-памет. Този метод се използва автоматично.

III.1.2.3.3 Методи за проверка на типа и флаговете на атомите

Проверката на типа на атомите може да стане чрез проверка на стойността, записана в _Id на атома, но за удобство са реализирани функции за всеки отделен тип: IsEmpty, IsAtom, IsList, IsExpr, IsVar, IsObj, IsMem, IsNumb и IsTrue. По особен е методътIsNumb, който не само проверява дали в атом се съдържа число, но и ако се съдържа текст, който може да се представи като число, методът прави необходимата конверсия. IsTrue проверява дали в атом се съдържа стойността `"True`.

Фигура III-5 Проверка дали атом съдържа число



Достъпът до флаговете на атом може да стане директно с метода Flags, който връща всички флагове наведнъж, или пък с по-общия метод GetBit или с по-конкретните IsLocked и IsLockedByUser. Промяната на флаговете може да стане наведнъж със SetFlags или поединично със SetBit.

III.1.2.3.4 Методи за стойността на атомите

Атомите, според вида им, могат да имат различни стойности. Извличането и променянето им става със съответни методи. Това се налага поради факта, че вътрешното представяне на данните беше сменяно на няколко пъти, докато интерфейсът към атомите се е запазил почти непроменен от последните няколко реализации на системата.

Както вече бе описано, някои полета в атомите се използват за различни цели. Затова, при употребата на методите, потребителят се освобождава от изискването да помни за кои видове атоми какво се записва в полетата на атома.

Ако атом съдържа променлива, стойността й се извлича и променя с Value и SetValue, а се конвертира до число или текст с NumberOf и StringOf. За работа с името на променливата се използват VarName, FullName и SetVarName. Претърсването на

подчинените променливи на дадена променлива се извършва с ChildIndex и ChildPtr-Index, които намират подчинена променлива, за която се знае името или адреса й.

Когато атомът съдържа Windows обекти, достъпът до тях става с метода Obj, който връща обект от тип TwinControl, и ObjType, с който се проверява типът на обекта.

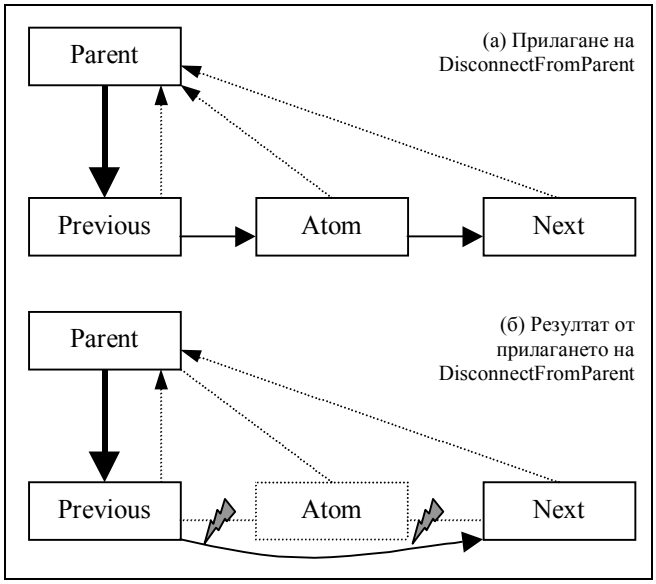
За удобство са дефинирани няколко метода, които се използват често. Методът Count връща броя елементи в атома – ако е списък – броя на елементи, ако е текст – броя на буквите, а ако е число – броя на буквите при текстовото представяне на числото. Методът LastNode намира последния елемент от списък, а IsUsed проверява дали атом е резервиран и забранен за освобождаване.

За вътрешни цели се използва методът Dump, който преобразува атома в текстов низ, независимо от стойността му и метода NextAtom, който задава следващия атом от списъка, в който се намира текущия атом.

III.1.2.3.5 Методи за работа с променливи

Методите за работа с променливи са едни от най-сложните като концепция и история. Тяхното идеологическо изчистване е продължило няколко версии на системата и то в посока на опростяване и унифициране.

Фигура III-6 Премахване на пряка вертикална релация



Най-простият метод е IsGlobal, с който се определя дали променлива е глобална или не. Това става с проверка дали променливата е подчинена на главната променлива-корен, която няма родител.

Създаването и премахването на вертикалната релация се контролира с методите DisconnectFromParent и ConnectToParent. Първият от двата метода проверява дали между две променливи съществува пряка вертикална релация и ако има я премахва.

Методът ConnectToParent е симетричен, с тази разлика, че атомът, който се определя като подчинен на друг, се записва като първия подчинен. Единствената причина за това е по-високата скорост. Обратната връзка между "премахнатият" атом и родителя му не се премахва, защото тя става автоматично невалидна поради една от следните причини:

- Премахването на родителска връзка се налага при изтриване на атома или неговия родител. В такива случаи връзката ще бъде естествено инициализирана при освобождаването на атома.

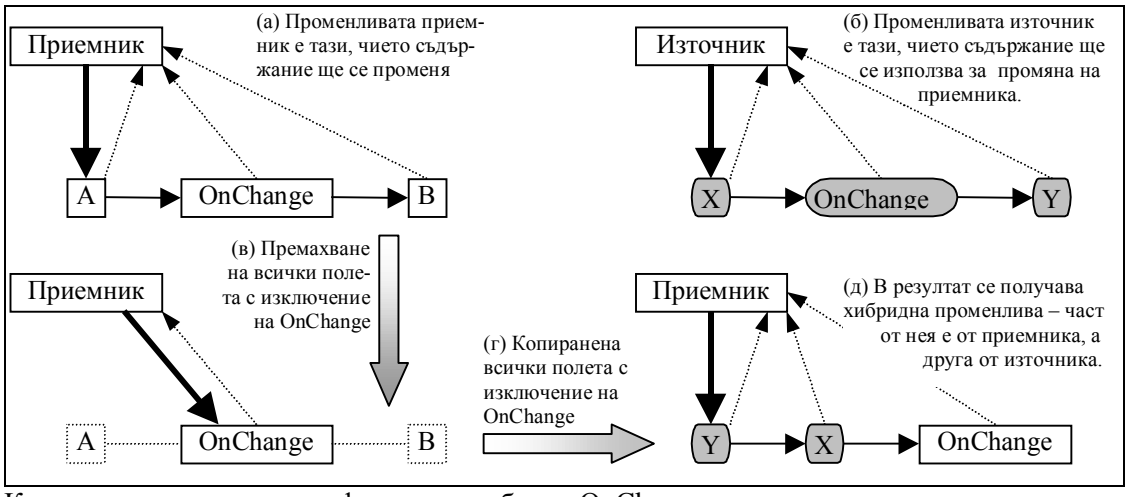
- Премахването на родителската връзка се налага при прехвърлянето на една променлива от един родител към друг. В такива случаи прикачването към втория родител автоматично ще актуализира обратната връзка.

Методите DeleteVarChildrenOnly, DeleteVarButGlobalRules и CopyVarButGlobalRules са трите най-важни метода на атомите. Те се прилагат, когато стойност или части от една променлива трябва да се прехвърлят в друга променлива. Проблемите възникват когато променливите имат подчинени променливи. В някои случаи само част от тях трябва да се прехвърлят от дарителя към приемника.

Методът DeleteVarChildernOnly премахва връзката от променлива към първата й подчинена променлива. А това автоматично я лишава и от всички останали променливи. По този начин бързо и ефикасно се премахват всички подчинени променливи. Интересно е, че техните обратни връзки се запазват, но това не е проблем, понеже при пълното премахване на подчинените променливи всички те подлежат на освобождаване.

При случаите, когато трябва на един обект да се присвои друг обект, се използват методите DeleteVarButGlobalRules и CopyVarButGlobalRules. Присвояването между методи не трябва да се извършва чрез просто премахване на полетата и методите на единия и създаване на полета и методи, които са идентични с тези на другия обект. Всъщност по тази процедура могат да се обработват почти всички подчинени променливи, с изключение на тези, свързани с OnChange и OnPlan събитията.

Фигура III-7 Присвояване между обекти



Когато променлива има дефинирано събитие OnChange и на тази променлива присвоим друга променлива, то събитието не трябва да се премахва или заменя с друго. Същото се отнася и за вече създадения план на изчисление, съхранен в OnPlan. От друга страна, когато от променлива се изисква да даде своята стойност заедно с полетата и методите си (ако е обект), тя трябва да предаде всичко с изключение на OnChange и OnPlan, които са характерни за конкретната инстанция.

Процесът на присвояване на обекти се извършва на две стъпки. През първата от променливата-приемник се премахват всички подчинени променливи, освен тези, свързани с правилата. За целта се използва методът DeleteVarButGlobalRules. Втората стъпка копира трансферируемите полета и методи от източника към приемника чрез използването на CopyVarButGlobalRules.

III.1.2.3.6 Методи за поддържане на междинен код

Междинният код в Elica се използва предимно за ускорение на изпълнението на програмите. Системата компилира текста на програмите до междинен код, който в

последствие се интерпретира. Компилацията се извършва на малки стъпки – команда по команда и получените команди на ниско ниво се "прикачат" към реалната команда. Атомите съдържат 9 метода, които помагат работата с междинния код. Тези методи добавят или извличат междинен код според спецификата на командата, а също променят или извличат позицията на атома в пространство-времето на програмата.

III.1.2.4 Допълнителни функции

Освен методите в класа на атомите, в Elica са дефинирани и няколко помощни функции, които са външни, но се използват в тясна връзка с атомите. Част от функциите са за създаване и инициализиране на атоми от различен вид: NewAtom, NewList, NewVar, NewWord, NewNumb, NewObj и NewMem. Възниква въпросът защо е необходимо да се дублира създаването и инициализирането на атомите, като поначало в класа има дефинирани съответните конструктори и методи. Отговорът е, че в Elica се използват допълнителни алгоритми за създаване на атоми.

Основната идея е в това, че след като атом е бил използван и е освободен, за неговото бъдеще има две възможности:

- заетата от атома памет да бъде върната на операционната система
- заетата от него памет да не се върне на операционната система, а да се запомни като вече свободна памет.

Използването на втория вариант дава съществено преимущество в скоростта на работа, понеже след като се натрупа достатъчно количество свободни атоми, Elica не се обръща към операционната система за заделяне на памет, а използва вече заделената и фрагментирана памет.

Този начин на работа става възможен чрез използването на специален списък на свободните атоми. След като Elica не се нуждае от някой атом, той се "закача" в началото на списъка. Това е операция с константна сложност. Когато Elica се нуждае от нов атом, първо се проверява дали има такива в списъка от свободни атоми. Ако има, извлича се най-първият – това също е операция с константна сложност, а ако няма – тогава се прави обръщение към операционната система да задели допълнително памет.

Списъците на свободните и на заетите атоми се поддържат в структурата на изпълнението и ще бъдат описани в някоя от следващите глави.

Останалите допълнителни функции и процедури са:

DefaultRelease	Процедура за общо освобождаване на памет, заета от атом-памет. За по-специалните случаи, когато освобождаването изисква конкретни действия, потребителят трябва сам да дефинира съответната процедура.
AddToListAt	Добавя елемент към края на списък. За да може да се използва, трябва списъкът да се създава от самото си начало с тази процедура, или пък да се пази указател към последното му звено
Position	Създава четиримерна позиция според зададените име на прозорец, номер на ред и знак, и дълбочина на изпълнението.
Lock87	Заключва числовия копроцесор и възникването на прекъсвания при изчисленията не спира изпълнението на програмата.
Error	Генерира съобщение за грешка с произволен текст. Създателите на външни библиотеки могат да използват тази процедура, за да си създават собствени грешки.

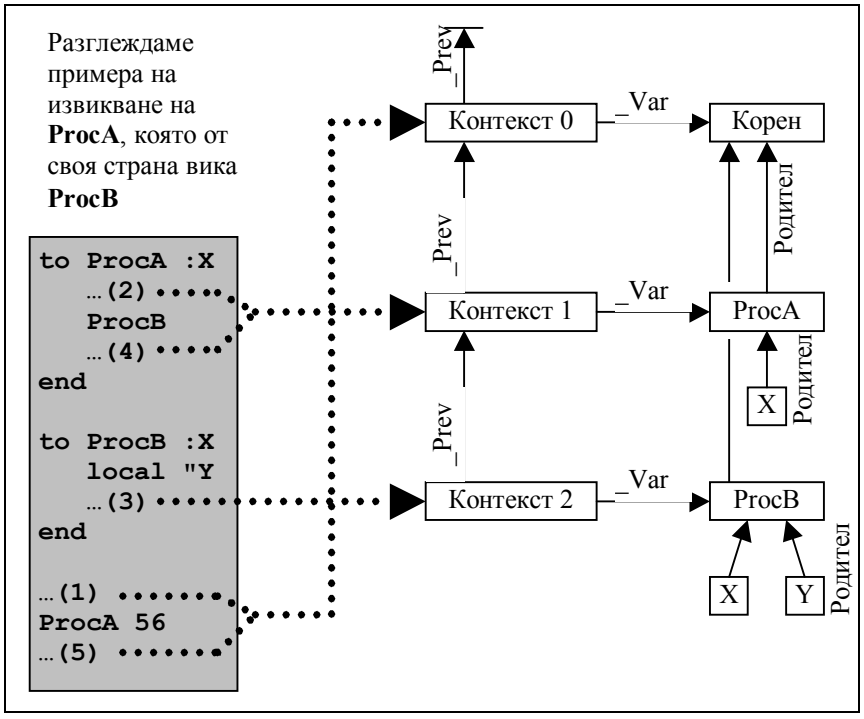
Compare	Сравнява два произволни атома. Ако атомите са списъци или изрази, сравнението е рекурсивно. Резултатът определя кой от двата атома-аргументи е по-голям, или пък определя че са равни. Функцията може да определи и кога се сравняват несравними атоми (например списък и атом-памет).

III.1.3 Структура на контекста

Изпълнението на програма на Elica е сложен процес и за неговото изпълнение са дефинирани други две структури – структура на контекста и структура на изпълнението. Както подсказва името на първата, тя се използва за да капсулира текущия контекст по време на изпълнението. Под *контекст*, в Elica се разбира текущото ниво (или дълбочина) на изпълнение, които дават еднозначна определеност на видимите и достъпните променливи.

Поради унификацията в Elica, основните дейности и данни, които са необходими за контекста са налични в дефинициите на атомите-променливи. Затова и за контексти се използват самите променливи.

Фигура III-8 Списък от контексти



Когато трябва да се извика процедура или функция, променливата, която съдържа нейната дефиниция става текущ контекст. От този момент нататък, докато не се извика нова функция или пък не се прекрати работата на текущата, всички локално създадени променливи, ще са създадени като локални на текущия контекст.

При завършване на работата, текущият контекст се премахва, без да се унищожава съответната променлива, и предишният контекст става отново текущ.

Подобен в една или друга степен механизъм съществува при всички реализации на транслатори и най-често, при тях този механизъм носи името *стек на изпълнението*. Реализацията в Elica също е чрез стек, но на базата на списък.

Дефиницията на структурата на контекста е проста – съдържат се две защитени полета: _Var – указател към променливата, реализираща контекста, и _Prev – указател към предишния контекст. Достъпът до двете полета се осъществява с публично достъпните

методи Variable и PreviousContext. Естествено, дефиниран е и конструктор на контексти, който автоматично свързва новосъздаденият контекст със списъка от контексти, като по този начин го прави текущ.

В примера от Фигура III-8 се показва състоянието на стека от контексти по време на изпълнение на проста програмка. В момента отбелязан с (1), текущият контекст е Контекст 0 (другите два контекста още не съществуват). Когато се извика процедура ProcA, се създава нов текущ контекст Контекст 1, който сочи към вече предишния, а контекстната променливата е ProcA, в която има дефинирана една локална променлива X. Тази позиция е отбелязана с (2). След като се извика и ProcB (момент (3)) се създава нов текущ контекст, сочещ към променливата ProcB. След като ProcB завърши изпълнението си в момент (4) текущ контекст става отново номер 1, а след като и ProcA завърши, текущ става отново първият контекст – този, с номер 0.

Първият контекст, който сочи към променливата-корен, се създава автоматично от системата преди изпълнението на програмата. Кореновата променлива не е достъпна за потребителя, но всички нейни подчинени променливи се виждат като глобални.

Най-простият пример, в който се вижда ползата от контекстите е следният. Използвайки примера на същата фигура, ако в момент (3) се изисква да се работи с променливата X, то ще се използва тази X, която е подчинена на ProcB. Но ако в моменти (2) или (4) се работи отново с X, ще се използва другата X – тази, подчинена на ProcA. Именно по този ефикасен начин се управлява не само стекът на изпълнението, а и видимостта на променливите.

III.1.4 Структура на изпълнението

Структурата на изпълнението има няколко важни функции, които в по-предишните реализации на системата са били разпределени в няколко различни структури. Обединяването, освен че продължава концепцията на унификацията, се използва и за значително опростяване на комуникацията на системата с външните библиотеки.

Основните функции на структурата на изпълнението са следните:

- унифицира достъпа на външните библиотеки да ресурсите на системата;
- унифицира достъпа до променливите и текущото състояние на системата;
- капсулира на най-високо ниво контрола върху изпълнението на програмите.

III.1.4.1 Защитени полета

Защитените полета в структурата се използват единствено от другите методи. Достъпът до някои от тях се осъществява със специални методи.

Както бе споменато преди, в системата се поддържат два основни списъка на атоми. В единият списък са свързани всички използвани атоми, а в другия – всички свободни. Независимо, че си приличат и че са реализирани по един и същ начин, двата списъка се използват поради различни причини.

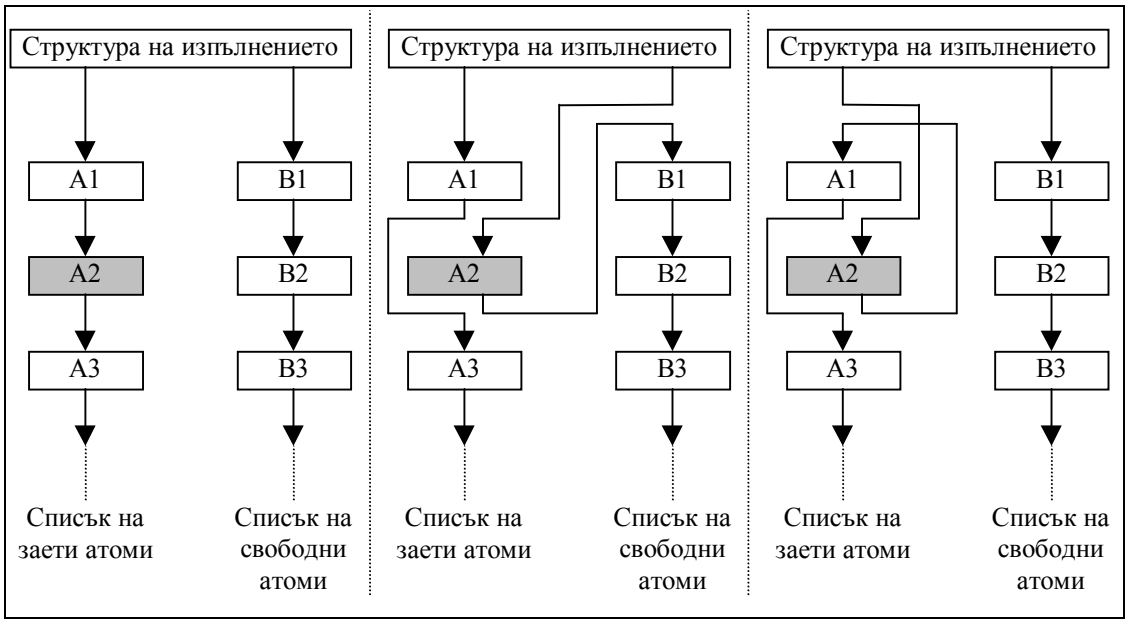
```
_UsedAtoms : TAtom; // Списък на заети атоми
_FreeAtoms : TAtom; // Списък на свободни атоми
```

Списъкът на заетите атоми съдържа всички атоми, които се използват от системата в рамките на конкретния контекст на изпълнение. Тази информация се използва при намиране на неизползваните атоми и тяхното освобождаване. Следователно, по време на работата, атоми, които не се използват вече, остават за известно време в списъка на заетите, докато специален процес не ги открие и освободи.

Освобождаването на атомите е просто премахването им от списъка на заетите и включването им в списъка на свободните. Това определя и главната роля на втория списък – той е източник на свободни, вече алокирани в паметта атоми. Причината за

това е, че заемането на памет за атом от паметта на операционната система, а също и връщането на тази памет е бавна операция.

Фигура III-9 Освобождаване и заемане на атоми



Поставянето на елемент в началото на списък и извличането на елемент от началото на списък са неимоверно бързи операции. Работата на Elica създава и унищожава по няколко стотина хиляди атома в минута, което определя и смисъла на използването на списъка на свободните атоми.

На горната фигура е разгледан примерът, при който се освобождава атома A2 (показан в левия сегмент). След освобождаването му (централния сегмент), той се изключва от списъка на заетите атоми и се включва в началото на списъка на свободните атоми. Ако в този момент системата се нуждае от атом, той се взема и връща в началото на списъка на заетите атоми. Десният сегмент показва крайното положение.

Други две от защитените полета: _Context и _RootContextVar се използват за достъп до променливите и контекстите.

```
_Context        : TContext; // Текущ контекст
_RootContextVar : TAtom;    // Коренова променлива
```

Полето _Context представлява текущия контекст. Както е показано на Фигура III-8 от текущият контекст може да се достигне до всички останали контексти. По този начин може да се обходи пълният стек на изпълнението. Второто поле е променливата, асоциирана с най-първия контекст. Това е кореновата променлива, която няма родител, а същевременно всички останали променливи са преки или косвени нейни наследници.

В по-предишните реализации на системата последното от защитените полета _TempVar се е използвало за даване на уникално име на променлива. В настоящата система се използва като генератор на уникални числа. При всяко използване на стойност от полето, то трябва да се увеличи с единица.

```
_TempVar : cardinal;    // Генератор на уникални числа
```

Понеже полето вече се използва за множество цели, номерата, отделени за всяка конкретна цел може да не са последователни. Не е предвидена защита от препълване, защото за практически цели полето е с достатъчна големина. Ако всяка секунда се генерират по 1000 уникални числа, първото повторение ще се случи след около два месеца непрекъсната работа. Независимо от това, подобно повторение също не е проблем, защото то е опасно само ако се случи в рамките на едно и също използване, и

то само ако уникалността е съществен критерий. В последната реализация на Elica, използването на _TempVar не създава проблеми относно уникалността.

III.1.4.2 Публични полета

Публичните полета в структурата на изпълнението съдържат предимно параметри за настройка на системата. Полето `Epsilon` съдържа точността, с която да се работи с реални числа при сравняване, `CaseSensitive` е флаг, който указва дали за идентификаторите главните букви да се третират като различни от съответните малки. Този флаг е достъпен от работната среда и не може да се променя програмно.

Полето KeyBuffer съдържа въведените данни от клавиатурата, които все още не са обработени от програмата, която работи в момента. Използването на това поле е главно от страна на модула за управление на вход и изход.

```
Epsilon       : float;     // Точност на сравнения
CaseSensitive : boolean;   // Разлика между главни и малки букви
KeyBuffer     : string;    // Буфер на въведените символи
```

Останалите публични полета се използват за системни цели и са поставени на това място, поради съображения за видимост.

`FoundInParent` и `LeftName` се използват от методите за търсене на променлива във фазите, когато име на пълно променлива се декомпозира на съставните си елементи, които се претърсват поединично.

```
FoundInParent  : TAtom;    // Родителя, от най-ниското възможно ниво,
                           //   съдържащ променливата
LeftName       : string;   // Необработена част от името
```

Публичното поле `GraphCall` е от особена важност. То съдържа адреса на call-back функция, която създадените от потребителя външни библиотеки могат да използват, за да имат достъп до графичните възможности на системата.

```
GraphCall : TGraphCall;    // Call-back графична функция
```

Последните две полета се използват за вътрешни цели. С тях се проследява броя на заетите атоми и броя на наличните свободни. В исторически план те са се използвали за проследяване на проблеми с паметта и отстраняване на грешки. Понастоящем, след като подобни проблеми са вече изчистени, те ще се използват епизодично, след евентуални промени в управлението на паметта, за да се провери коректността на промените.

```
Used : integer;    // Заети атоми
Free : integer;    // Свободни атоми
```

III.1.4.3 Методи

Всички методи в структурата на изпълнението са публично достъпни. Част от тях се използва за достъп до защитените полета, а друга – за работа с променливите. Именно тази част от методите е ключова в работата на системата.

III.1.4.3.1 Методи за работа с контексти

Контекстите, както вече бе обяснено, са структури, които съдържат описание на дълбочината, до която е стигнало изпълнението на програма, а също и определят кои променливи са достъпни и кои не. Достъпът до контекстите се осъществява с три метода – за добавяне на нов контекст, за премахване на контекст и за проверка на контекст.

Методът `PushContext` създава нов контекст, поставя го в списъка на контексти и го обозначава като текущ контекст. Тази операция трябва да се изпълнява всеки път, когато се налага или се очаква да се прави поне едно от следните неща:

- извикване на подпрограма изисква създаването на контекст, в който да се съхраняват локалните полета и процедури. Това се налага не само за да могат после лесно да се премахнат, но и ако се създава обект, тези локални полета и процедури ефективно да се преобразуват на полета и методи.
- ще се създават временни локални променливи, които после лесно и бързо ще трябва да се премахнат. Променливите се създават в новия контекст, и когато контекстът се премахне, всички те престават да са достъпни и в последствие се освобождават.
- насилствено сменяне на контекста на изпълнението. Това е една много честа операция при извикването на вътрешен метод на определен обект. По време на изпълнението на командите от метода, програмата трябва да има достъп до полетата и методите дефинира в преките и непреките родители на метода или пък в друг, независим обект.

В края на изпълнението на програмата, броят на създадените и премахнатите контексти трябва да е един и същ. За премахването на контекст трябва да се използва методът `PopContext`. Премахването на контекст автоматично предефинира предишния контекст като нов текущ, а освен това и премахва всички локални променливи, освен ако те не се ползват и за други цели. Често се налага да се знае текущия контекст. Информация за това може да се получи с метода `CurrentContext`. След получаването на контекста, могат да се използват неговите полета и методи директно.

Използването на `CurrentContext` трябва да става внимателно и особено със знанието, че текущият контекст да може да се променя не само от главната програма, а и от странични дейности на системата. Най-често срещаният случай е, когато се промени размерът на прозореца или пък се открият скрити до момента части от него, особено ако в момента се изпълнява програма. В такива случаи системата автоматично пречертава екрана и ако се налага, извиква потребителски подпрограми за пресмятане на графичното изображение. Именно в този момент, системата създава свои временни контексти, които се премахват след приключване на рисуването. В крайна сметка контекстите са отново същите, но е имало период, в който те не са били.

В настоящия момент това не води до проблеми в системата понеже няма паралелно изпълнение на програми, но ако се правят отделни процеси или нишки, ще трябва програмистът да се погрижи и да следи за контекста.

Когато една променлива се използва като контекст, тя става активна, защото нейните локални полета стават достъпни за обкръжението. В структурата на всяка променлива съществува флаг, който определя дали променливата е активна. Този флаг се използва в редица случаи, но понякога се налага да се изчиства изрично. За тези цели се използва методът `ClearActiveFlag`, която обхожда рекурсивно цялото дърво, започвайки от кореновата променлива и нулира флага във всички променливи.

III.1.4.3.2 Методи за създаване на променливи

Съществува един единствен метод за създаване на променливи `AddLocalVar` и негов опростен вариант, използван за създаване на уникални локални променливи - `AddUniqueVar`. Създаването на променлива изисква определяна на няколко параметъра:

- къде да бъде създадена променливата, т.е. след създаването й коя друга променлива ще й е родител.

- пълно, кратко или относително име на променливата. Методът `AddLocalVar` може да създава не само обикновени променливи, но и при нужда да създава всички междинни липсващи променливи. Например, ако съществува променлива `A` и в нея има само поле `B`, създаването на променлива `A.B.C.D` в родителя на `A` ще навлезе в `A.B` и от там ще създаде `C`, а в нея и `D`. Ако създаването се прави не в родителя на `A`, а в `A.B`, ще се създаде променливата `A.B.A.B.C.D`.
- стойност на променливата. Когато стойността не е известна към момента на създаване, се подава празен указател, който съответства на празния списък.
- ниво на очакване за уникалност на променливата. Това е един важен параметър, който бе добавен сравнително късно и служи за съществено ускорение при създаването на някои променливи. Ако се очаква променливата да не е уникална (както в примера по-горе, където A.B не се създава, а се използват наличните), преди създаването на всяко ниво от междинни променливи се проверява дали евентуално те не съществуват. Ако не съществуват – то се създават, а ако съществуват – използват се наготово. В много случаи, когато системата създава свои вътрешни временни променливи, тя знае предварително, че те са уникални и няма нужда да се правят каквито и да е проверки за съществуване. Тази функционалност се използва активно при правене на дубликат на обект – полетата на оригинала се дублицират едно по едно, като се знае, че в рамките на новосъздаващия се дубликат те ще са уникални.

В схемата на алгоритъма на създаване на променливи се споменава изискване за уникалност. Това изискване е налице в два отделни случая, които могат да настъпят и едновременно. Първият случай е когато като параметър към функцията е споменато, че се очаква създадената променлива да е уникална.

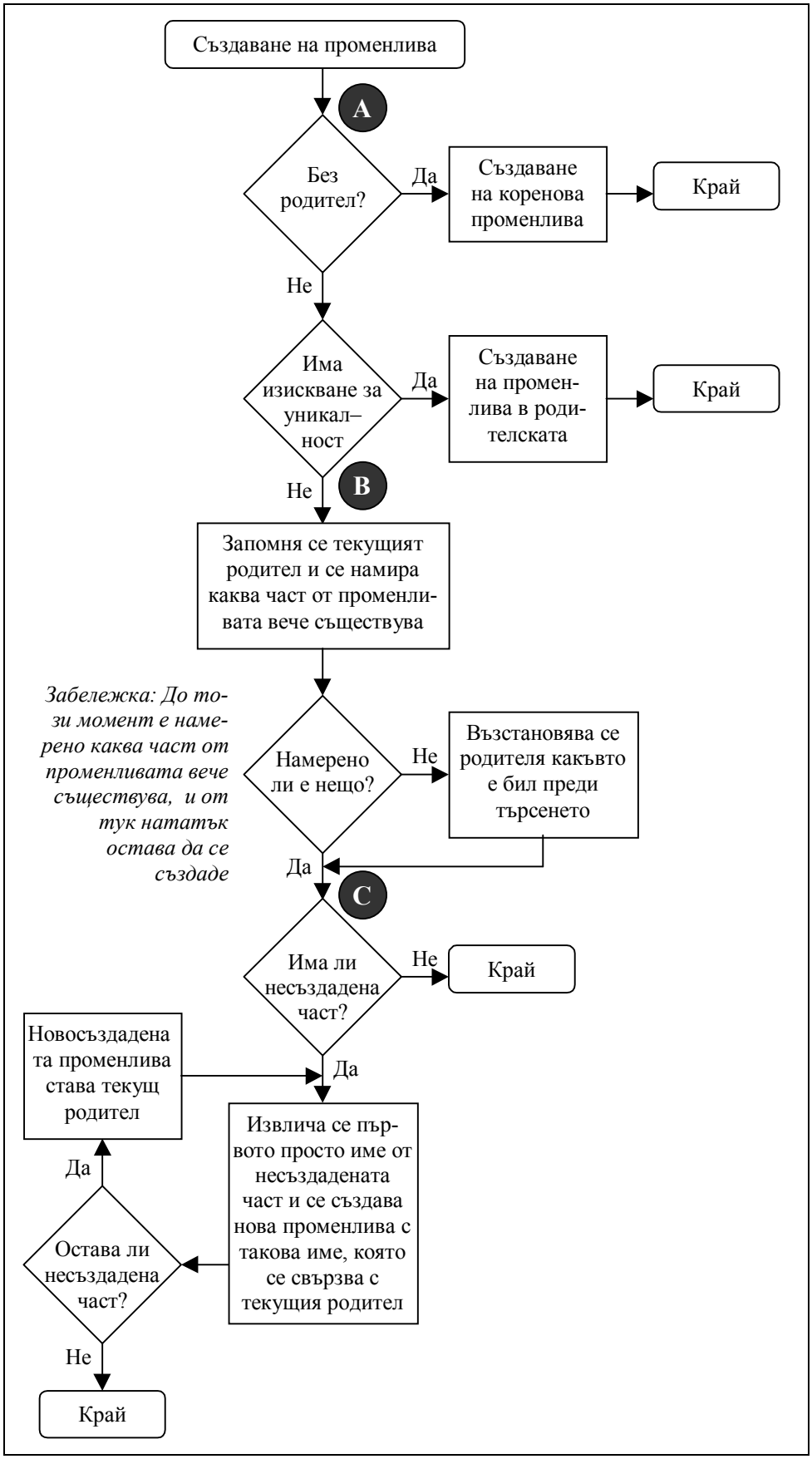
Вторият случай е малко по-сложен. Всички уникални променливи в системата имат едно и също стандартно име, което е фиксирано като константа и не може да се променя. Това не ограничава уникалните променливи, понеже те могат да са в различни контексти (т.е. да имат различни родители). Подобни променливи, с име равно на уникалното име не могат в общия случай да се създават пряко от потребителя, а само от системата. Тези променливи са винаги временни – т.е. след приключване на работата с тях те се изтриват.

Създаването на уникална променлива става с метода `AddUniqueVar`, който има един единствен параметър – стойността на променливата. Новата променлива се създава като подчинена на кореновата променлива. Ако се налага да се създава уникална променлива в рамките на друга променлива, може да се използва основният метод за създаване на произволна променлива, като се укаже очакваната уникалност.

Процесът по създаване на променлива може грубо да се раздели на три независими фази (на схемата началото на всяка фаза е отбелязан с черно кръгче с името на фазата).

- Фаза A. По време на тази фаза се определя дали променливата не попада в някои от особените случаи и ако да, те се обработват. Особените случаи са когато променливата ще е без родител (тогава се създава коренова променлива) или пък когато се изпълнява изискването за уникалност, обяснено по-горе.
- Фаза B. Тази фаза служи за определяне на това каква част от името на променливата вече съществува и естествено, това се прилага само когато името е съставно. Ако се намери такава част, тя става родителя, в който ще се създава остатъкът от името на променливата
- Фаза C. Това е последната фаза, в която, ако има все още несъздадени променливи, те се създават итеративно.

Фигура III-10 Създаване на променлива



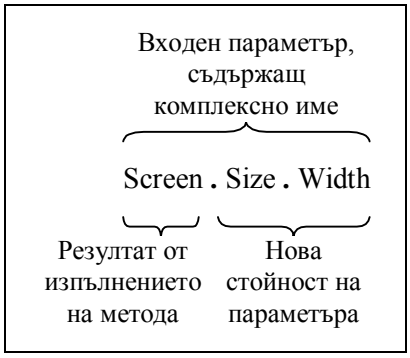
Най-честият случай е създаването на проста уникална променлива. Тогава първите две фази се прескачат (т.е. освен необходимите проверки не се прави нищо друго), а цикълът от последната фаза се изпълнява само веднъж.

Тази проверка за съществуване на уникалност в името се удовлетворява и когато име на променлива съвпада с уникалното име.

III.1.4.3.3 Методи за търсене на променливи

Тази група от методи е другата група, която е от особена важност не само за системата като цяло, но и за създаването на външни библиотеки, които имат достъп до променливите. Дефинирани са 5 метода за търсене на променливи или техни стойности и един помощен метод – `GetChildName` – с който съставните имена на променливи се декомпозират последователно.

Фигура III-11 Декомпозиране на съставни имена



Базовият метод за търсене на променливи е `FindChildVar`. С негова помощ се търси променлива в рамките на конкретен родител. Променливата може и да е комплексна. В такъв случай търсенето се извършва рекурсивно в съответните подчинени на родителя променливи.

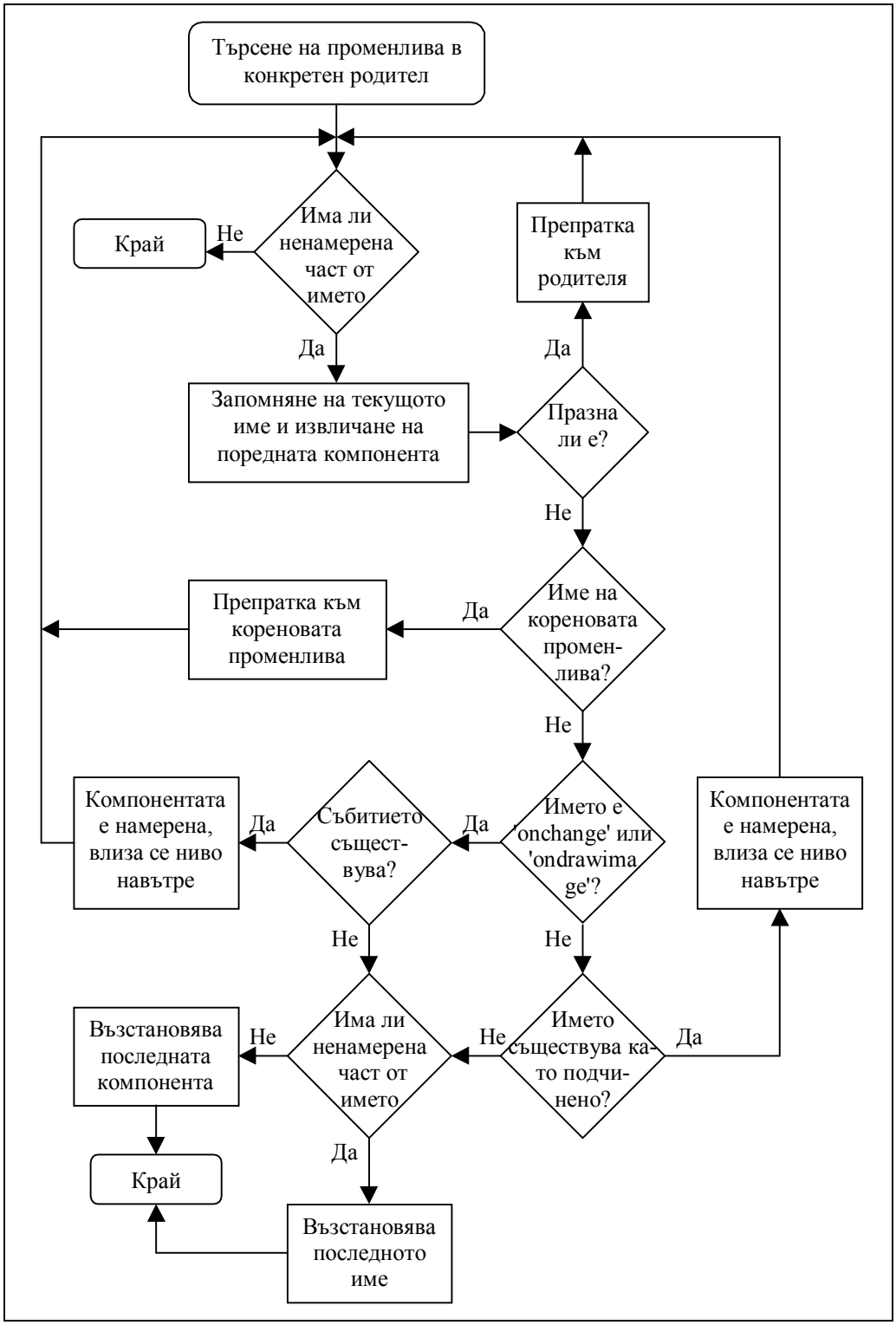
Ако името на търсената променлива е съставно и само част от него може да бъде идентифицирана, методът определя най-вътрешната точка от структурата на дървото на променливите, до която е имало съвпадение. Това се прави с цел да може лесно да се създаде "остатъкът" от променливата. Подобна технология се използва при създаването на променливи и бе по-подробно обяснено в предишната точка.

Като входни данни `FindChildVar` трябва да получи родителя, в който да се търси променливата и името на променливата. Името на променливата може да е просто или съставно. Ако е съставно, то може да съдържа препратка и към по-предишни родители.

Препращането към родителя става с вмъкването на празно име. Всеки път когато се срещне празен елемент от съставно име, търсенето на остатъка от името се извършва не в рамките на текущия родител, а в родителя на родителя. Допуска се неколкократно използване на препратка, като при всяка от тях търсенето ще се прехвърля на по-високо ниво.

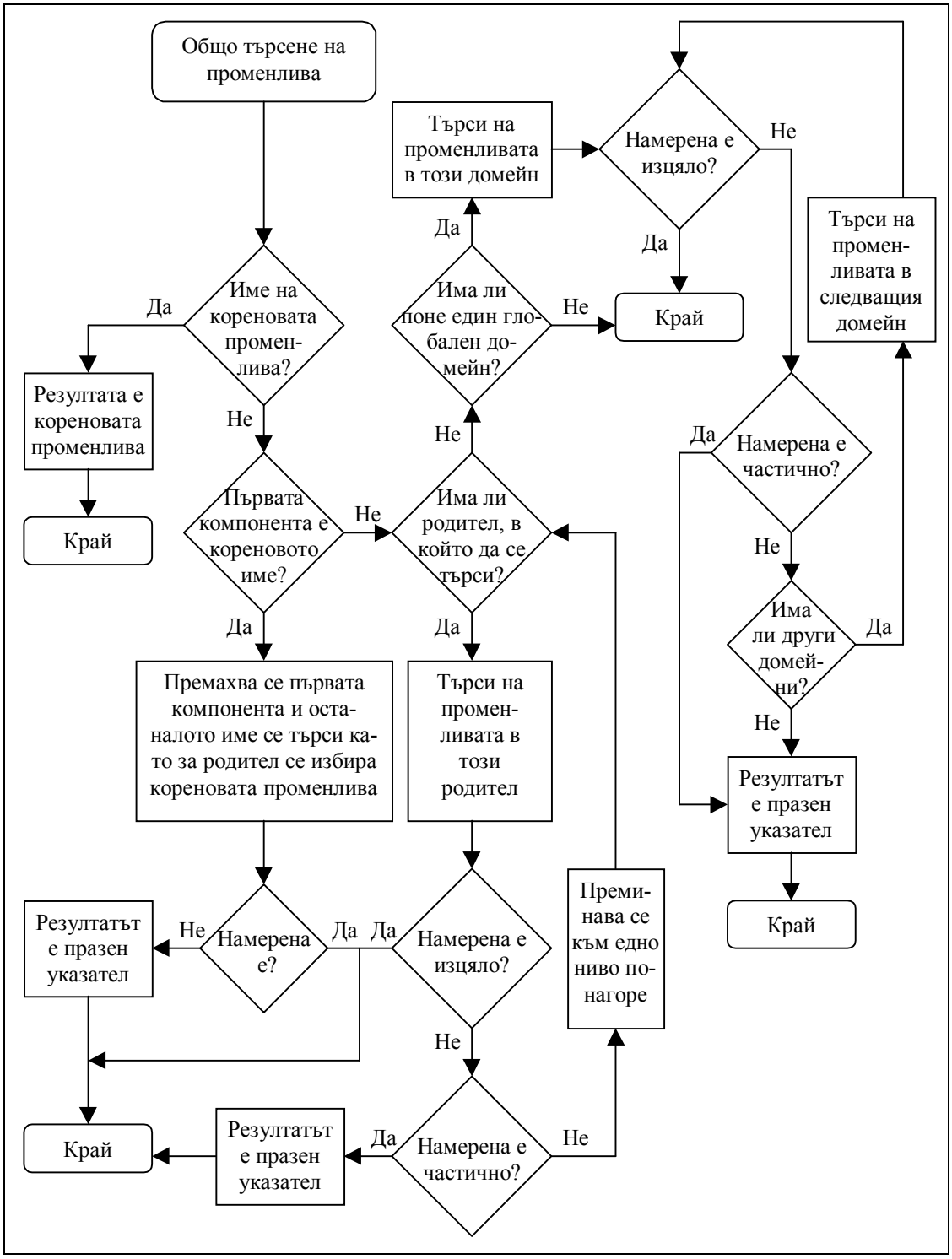
Използването на препратки в общия случай не е необходимо, защото търсенето автоматично преминава на по-горно ниво, ако на текущото търсенето е без успех. Все пак, в някои случаи употребата му е задължителна, особено ако се търси променлива от по-горното ниво на йерархията, за която се предполага, че може да има име, съвпадащо с име на променлива от текущото ниво.

Фигура III-12 Търсене на променлива в конкретен родител



Адресирането на променливата може да започне и от кореновата променлива. В такива случаи като компонента на името трябва да се съдържа името на кореновата променлива – тя винаги има фиксираното системно-дефинирано име `"'[Root]'`.

Фигура III-13 Общо търсене на променлива



Реализираният алгоритъм е получен след усилено изследване как по най-лесен начин може да се постигне следната функционалност:

- ако променливата съществува, да се намери адреса й в паметта
- ако променливата съществува частично, да се намери адресът на най-вътрешната намерена променлива и остатъкът от името, за което не е намерено съответствие
- ако променливата не съществува, нищо не се променя.

Поради честата употреба на локални променливи, могат да се използват следващите два метода `FindLocalVar` и `FindLocalVarValue`. И двата имат за аргумент име на променлива, а за родител се използва променливата, към която е асоцииран текущият контекст. Първият метод намира самата променлива, а вторият връща директно стойността ѝ. В случай, че такава променлива няма, и при двата метода резултата е празен указател.

Всички останали четири метода за търсене на променливи се базират на току-що описания `FindChildVar`. Най-простият от тях е `FindChildVarRO`, който извършва същите дейности и връща същите резултати като `FindChildVar`, но не променя никой от параметрите си. Методът се използва, когато търсенето на променлива не се придружава от създаването ѝ. Последният метод за търсене на променливи е `FindVar` и функционално е най-общият. Той също се използва масово при общото търсене на променливите. Функционално то се разделя на редица от търсения в конкретни родители. Ако търсената променлива е с абсолютна адресация (т.е. името ѝ започва с името на кореновата променлива), тя се търси като глобална. В противен случай тя се търси в променливата на текущия контекст, но и ако там не се намери, преминава се към родителя на променливата и търсенето се повтаря.

Ако след обхождане на всички родители променливата не се намери, тя търси последователно във всички глобални променливи, които са дефинирани като домейни. По този начин се реализира концепцията за домейните – обекти, чиито полета и методи са достъпни на системата, все едно че ли са глобални.

III.1.4.3.4 Други методи за достъп до променливи

При работата с променливите се налага да се знае коя е кореновата променлива. За тази цел се използва методът `RootVariable`. Кореновата променлива е по-особена не само поради факта, че няма родител, но и поради това, че не може да се премахне като другите променливи. Когато системата приключва своята работа или се стартира нова програма на Logo, кореновата променлива трябва да се изтрие и създаде наново. За изтриването се използва специалният метод `ClearRootVariable`.

III.1.4.3.5 Методи за достъп до броя на заети и свободни атоми

Поради факта, че защитените полета, съдържащи броя на заетите и свободните атоми, не се използват пряко от системата, те са оставени за случаите, когато се налага трасиране на механизмите на създаване и освобождаване на атоми. Методите, които могат да се използват за целта, са `SetUsedAtoms` (за променяне броя на заетите атоми), `GetUsedAtoms` и `GetFreeAtoms` (за извличане на броя на заетите и свободните атоми).

III.2 Управление на паметта

Управлението на паметта е един от най-сериозните проблеми, които трябва да се решат при проектирането и реализирането на система от ранга на Elica. Основният проблем се състои в това, че съществуващите начини за работа с динамични структури могат да се оптимизират да са или бързи или икономични, но е много трудно да се постигат и двете цели. Среда, която работи усилено със списъци (каквито са Logo, Lisp, Prolog) изпитва сериозни проблеми при реализирането на някои "обикновени" операции, като присвояването, например. Това е така, защото присвояването на списък с много елементи може да е доста бавна и тежка операция, особено и ако списъкът съдържа много подсписъци. Съществуват два вида реализации:

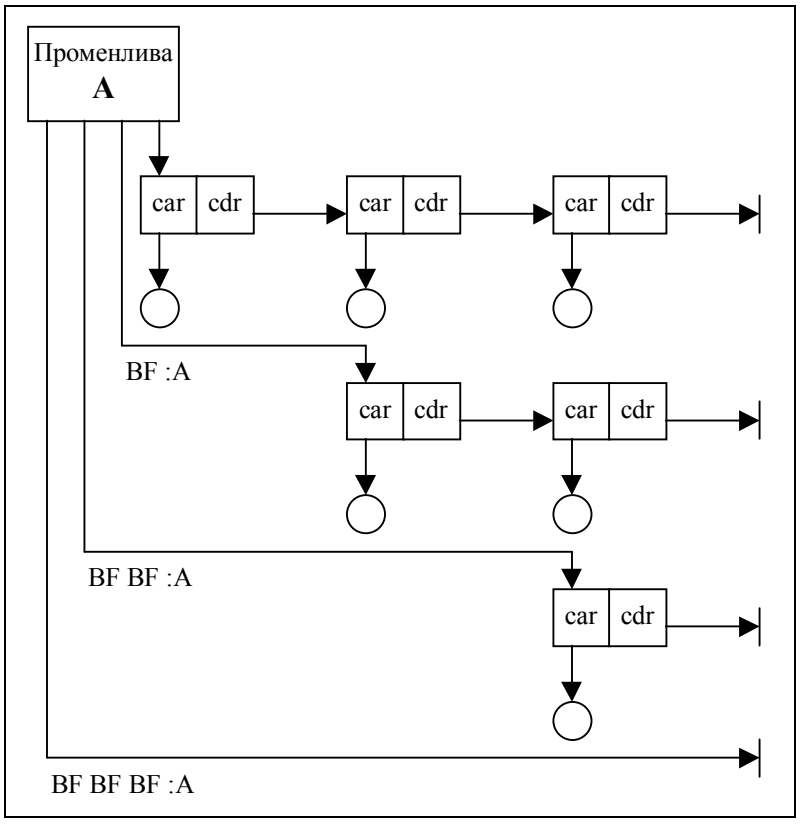
- всяко присвояване копира данните;
- всяко присвояване се извършва с минимален за момента разход на памет.

При първият начин няма проблеми да се управлява паметта, но пък недостатъците са прекалено силни, и на практика не се използва. За вторият начин са разработени няколко метода, два от които са приложени в Elica.

III.2.1 Присвояване чрез копиране на данни

Този начин на управление на паметта е най-лесният от гледна точка на проектиране и реализация, но води до неимоверно много проблеми. От една страна, присвояването става много бавна операция, а от друга – заема много памет.

Фигура III-14 Обхождане на списък чрез копиране на данни



Езикът Logo използва списъци. Една от най-често срещаните операции е да се обходи списък от началото до края, като на всяка стъпка се извършва някаква допълнителна дейност. Без значение дали обхождането се извършва итеративно или рекурсивно, при всеки вход ще трябва да се прави копие на всички елементи на списъка от текущия до

последния. Това е значително забавяне и разход на памет. На Фигура III-14 е показано обхождането на списък от три елемента с команда от вида на `MAKE "A BF :A`.

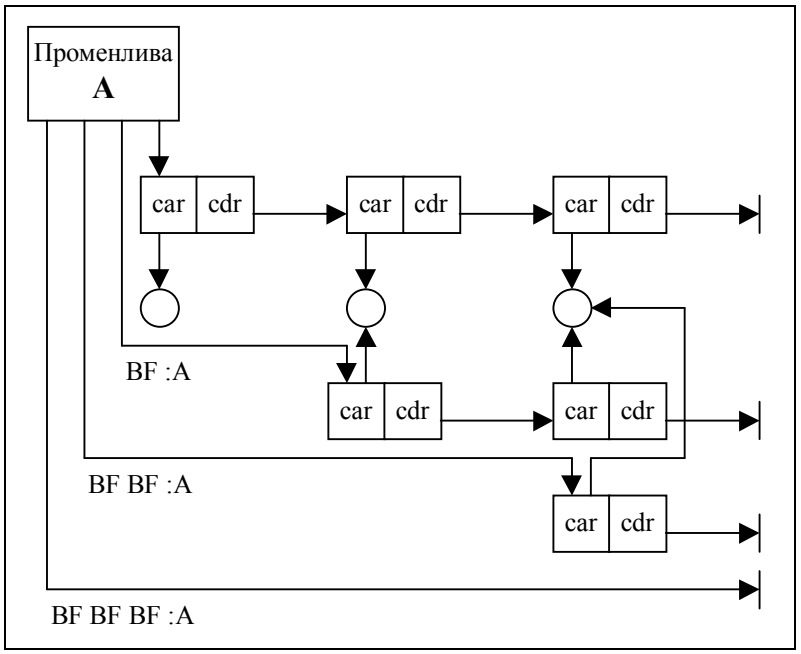
Към положителните свойства на този начин на управление на списъци може да се спомене лесното модифициране. Когато трябва да се модифицира част от списъка, това се извършва върху негово копие и за разлика от другия метод, не се налага оригиналният списък да "се защитава".

Другото положително свойство е лесното определяне кои блокове памет вече не се използват и могат да бъдат освободени става бързо и лесно – премахването на променлива или променянето на стойността й са единствения критерий при освобождаването на паметта.

III.2.2 Присвояване чрез промяна на указател

Противоположният по идеология начин на управление на паметта е да се пести време и място в максимална степен. Методът, описан в предната точка подлежи на очевидно оптимизиране – вместо да се копира целия списък, могат да се копират само възлите на списъка, а атомите, съдържащи същинските елементи на списъка се запазват (Фигура III-15).

Фигура III-15 Оптимизирано обхождане на списък

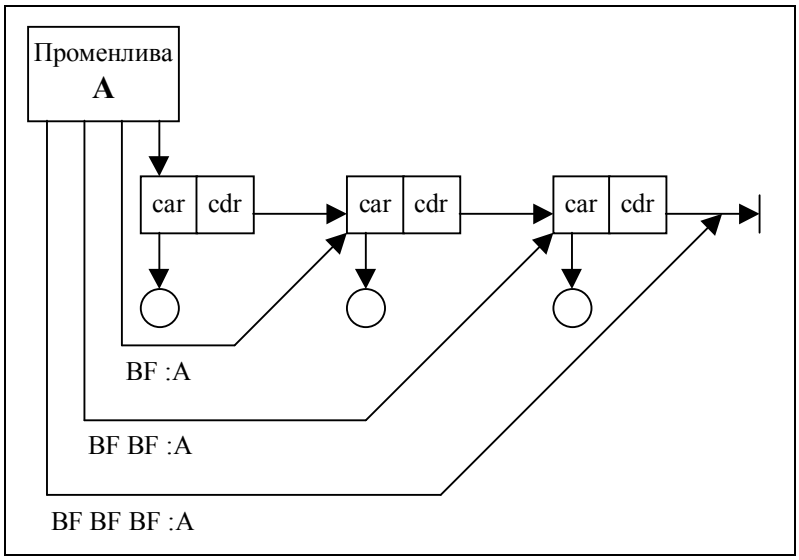


Продължавайки в същата посока, процесът на обхождане на списъка може да се оптимизира още повече - Фигура III-16. Вместо да се копират възлите на списъка, може просто указателя от променливата A да се премества последователно към края на списъка. Подобен начин на обхождане е най-бързият и с никакъв разход на памет.

Естествено, това не означава, че е оптимален за общия случай. Работата с паметта се състои не само в заделянето и манипулирането й, а също и освобождаването й. В третия начин на обхождане тежестта е пренесена върху освобождаването. Когато указателят се премества "навътре" по списъка, отминатите елементи вече не се използват, но това не означава, че те могат да бъдат освободени. Причината най-често се крие в това, че прилагайки аналогични начини за бързо присвояване на списъци, различни елементи на един и същ списък могат да бъдат сочени от различни променливи. Ако вместо примера `MAKE "A BF :A` се изпълни `MAKE "B BF :A`,

променливата B ще сочи към втория елемент на списъка и ако в този момент освободим първият, стойността на A ще бъде развалена.

Фигура III-16 Бързо обхождане на списък



Дори и да не се освобождава паметта има вероятност да възникнат други проблеми. Ако след командата `MAKE "B BF :A` изпълним `MAKE "C BL :A`, C сочи към първия елемент на A, но последният елемент от новата стойност на C е премахнат. Проблемът в случая е, че премахването на елемента от C го премахва и от A и от B.

III.2.3 Управление на паметта в Elica

Управлението на паметта в система Elica се извършва по хибриден начин – от една страна се копират възлите на списъците, а от друга при нужда, това не се прави. Подобна техника се използва и при управлението на обектите, които могат да се третират като дървета и за тях са приложими същите проблеми и решения.

След щателен анализ са идентифицирани всички случаи, при които може да се използва присвояване на указатели. Целта е колкото се може повече обработки да се извършват по този начин. За съжаление, той не може да се приложи навсякъде поради споменатите вече причини, но се използва поне в следните случаи:

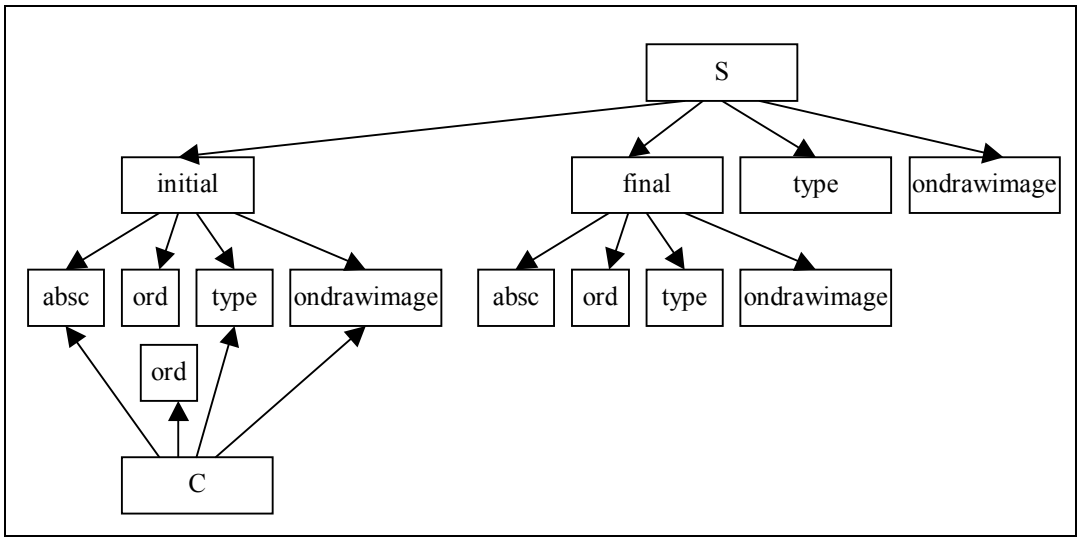
- при безопасни операции, като например извличане на списък без първият му елемент или без първите няколко елемента. Подобни операции са вградени във външните библиотеки и не са част от ядрото на системата. Тяхната безопасност ще бъде анализирана по-късно.
- при кратки служебни функции, вградени в ядрото на системата, като например тези за определяне дължината на списък или за композиране на пълното име на неглобална променлива. При подобни функции не се налага да се прави подсигуровка, която също ще бъде описана по-късно.
- при обекти, които са създадени временно и няма да бъдат легализирани (т.е. към тях няма да се асоциира име на променлива и потребителят няма да може да ги използва). Подобни обекти се създават постоянно в системата, особено когато междинните резултати в процеса на изчислението не са прости стойности, а инстанции на обекти.

За съжаление, съществуват случаи, при които не може да се използва само присвояване на указатели. При някои от тях се налага частично дублиране на блокове памет, а при други – цялостно дублиране. Случаите на частично дублиране са:

- при опасни функции за работа със списъци във външни библиотеки, като тези за изключване на последния или няколко последни елемента на списък. В тези случаи се дублират минималния брой възли на списъка, а елементите не се дублират.
- при временни обекти, които са част от други обекти, в които не се променя нищо и не се чете нищо. Подобни локални обекти не могат да се обработват само с преместване на указатели поради факта, че всяка променлива трябва да има най-много един родител. Частичното дублиране гарантира именно това изискване, като останалите връзки от йерархията не предизвикват проблеми и за елементите, които ги използват не се налага дублиране

Най-тежкият относно памет и време случай е, когато трябва да се дублира пълната структура. За списъци това никога не се налага, но за обекти се среща често. Най-елементарният пример е присвояването на глобални обекти. Ако имаме отсечката S и изпълним командата `MAKE "C :S.initial`, полетата в полето initial на S трябва да се дублират. Това се налага, защото впоследствие може да се променят някои полета от C и това не трябва да промени S.

Фигура III-17 Дублиране на обекти



Нека разгледаме примера от Фигура III-17. Нека след създаването на отсечка S и точка C, която съвпада в първата точка на отсечката да променим ординатата на C. В "оптималния", връзките в паметта могат да изглеждат както са показани на фигурата – C използва всички непроменени полета от S, а само за новопроменените е направено дублиране. Причините, поради които този начин на поведение не е оптимален са две:

- една променлива може да има няколко родителя. Това само по себе си не предизвиква проблеми, но изисква усложняване на алгоритмите в следните насоки:
  - реорганизиране на всички системни функции, които се основават на бързо обхождане на пътя от променлива до нейния родител. Ярък и често използван пример за това е проверката дали променлива е заключена (т.е. тя самата е заключена или поне един от родителите й е заключен);
  - реорганизиране на алгоритъма за търсене, който се основава на стандартен за системата еднопосочен път на търсене – от подчинена променлива към родителя й. Въвеждането на няколко родителя вмъква неяснота при търсенето на променлива. Очевиден пример за това е когато променлива с едно и също име се среща едновременно в две родителски разклонения. В такъв случай не съществува формален начин за определяне на "правилната" променлива, който да е верен за всички случаи, дори и тривиалните;

- реорганизиране на самия процес по създаване на променливи и поддържане на йерархията между тях. Проблеми възникват не само във вертикалната йерархия, а и в хоризонталната Ако две променливи от едно ниво имат различни родители, премахването на променлива или прехвърлянето й към друг родител изисква значителен брой операции, за да се гарантира коректността на данните.
- множествеността при родителите създава и още един проблем – вътрешната йерархия се преобразува от дърво в граф, който може да съдържа цикли. Това е доста неудобна структура при динамични среди като LOGO, защото създава проблеми при освобождаването на паметта. Те са преодолими, но с цената на използване на допълнителни ресурси.

- променянето на указатели и дублиране само на полетата, които в последствие са променени, усложнява алгоритъма по поддържането на променливите в паметта. Анализът коя част от структурата трябва да се дублира и коя не, не е бърза операция. В една от междинните реализации беше използван именно такъв метод, но практическото му използване е силно ограничено.

III.2.4 Освобождаване на паметта

Компромисът, който се прави в системата относно управлението на паметта с цел по-бързата й работа създава редица неудобства. За щастие всички те са преодолими, като необходимите ресурси за тяхното решаване са по-малки отколкото ако компромисите не бяха направени.

Най-важният проблем, който трябва да се реши е как да бъде реализиран механизъм, чрез който да се освобождава паметта, която не се използва, но е заета. Такава памет се генерира при операциите с преместването на указатели. Когато указател към един блок от памет се пренасочи към друг блок, първият трябва да се освободи, ако нито един друг указател не го сочи.

За да се реши този проблем в система Elica е реализиран леко модифициран вариант на механизъм за Garbage Collection.

По традиция, Garbage Collection се прави на няколко фази, като за помощни цели от всеки атом се използва по един служебен флаг:

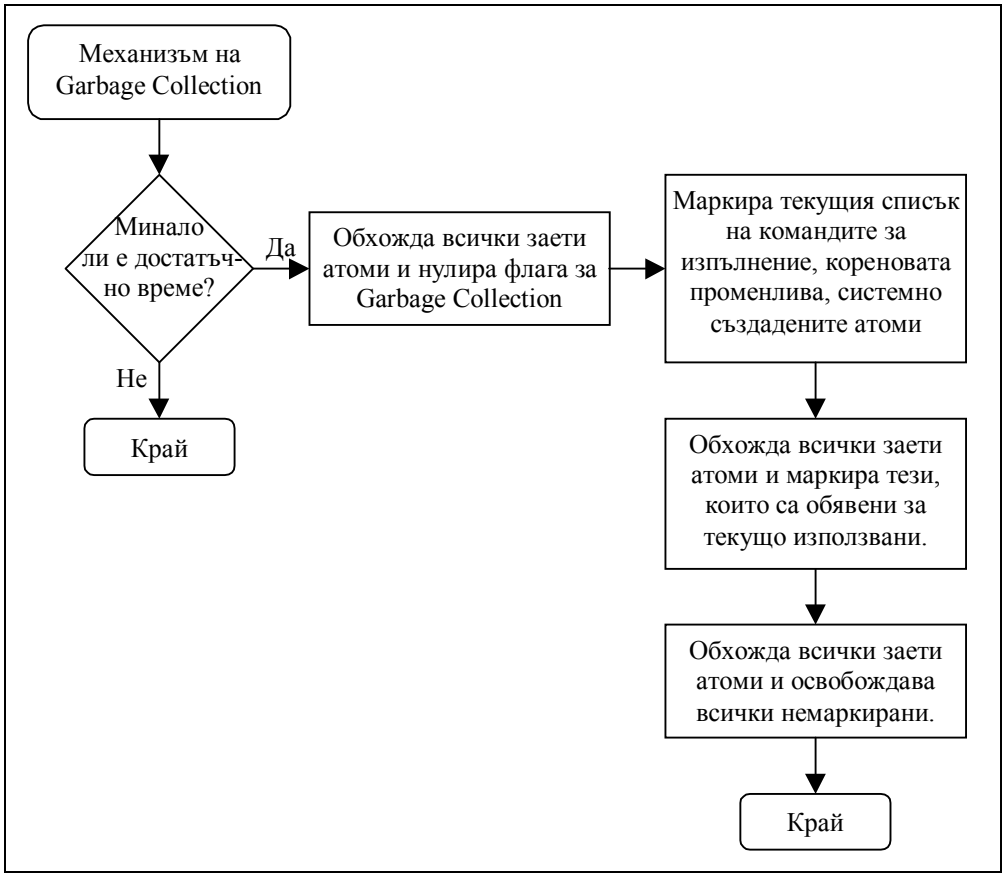
- последователно се обхождат всички заети атоми и флаговете им се нулират;
- обхождат се всички легално достъпни и реално използвани атоми и техните флагове се "вдигат";
- отново се обхождат последователно всички заети атоми и тези, които са със свален флаг и се освобождават.

Модификацията, която се налага е главно във втората фаза, при определянето на това кои атоми се използват реално и кои не.

На Фигура III-18 ясно си личат разликите. Първата от тях е, че освен да се маркира кореновата променлива (и всички други променливи, до които има достъп през нея), се маркират текущия списък от команди, системните атоми и текущо използваните атоми. Това се налага, защото използваните атоми не са само тези, които са асоциирани с променливите, но и тези, които по един или друг начин се използват от системата, без потребителя да има достъп до тях.

Списъкът от команди е указател, който се предава от процедура в процедура и сочи командата, която предстои да се изпълнява. Тази команда може да се програмно генерирана или да е междинен резултат и да не "принадлежи" на никоя променлива. Затова, защитавайки началото на списъка се защитава и целият списък.

Фигура III-18 Механизъм на Garbage Collection



За по-голяма скорост, системата създава няколко атома, които държи постоянно заети. Например, това са атомите, които съдържат думите `TRUE` и `FLASE`. Когато те са създадени, интерпретирането на потребителска функция, която връща булев резултат връща директно указател към някоя от тях. По този начин се пести памет и време.

Последният клас маркирани атоми са тези, които са декларирани за временно използвани. Всеки път когато системата създава указател, сочещ към атом, който ще бъде използван за повече време, атомът временно се декларира за използван. По този начин, ако евентуално настъпи Garbage Collection, атомът няма да бъде освободен. Реализирането на това става чрез увеличаване и намаляване на брояч в самия атом. Стойността на брояча показва от колко места атома се използва системно. По време на Garbage Collection, ако броячът е 0, този атом може да бъде освободен, освен ако трябва да се запази поради друг критерий.

Другата модификация спрямо стандартния алгоритъм е начинът, по който се определя кога да се прави Garbage Collection. Стандартният критерий не е приложим. Неговата същност е да се използва Garbage Collection когато свободната памет намалее под определена граница. Проблемите да се използва подобен критерий се крият във факта, че системата е работи под Windows. Тази операционна система се грижи приложението да има повече налична памет, отколкото физически е възможно. Тази памет, наречена виртуална, създава главния проблем. Когато свободната памет намалява, Windows се опитва да компенсира това, като записва запълнени блокове данни върху диска (в специални swap файлове). Когато приложението поиска да ползва памет, която вече е свалена на диска, Windows я прочита отново, като преди това й освобождава място в реалната памет като прехвърли други данни на диска.

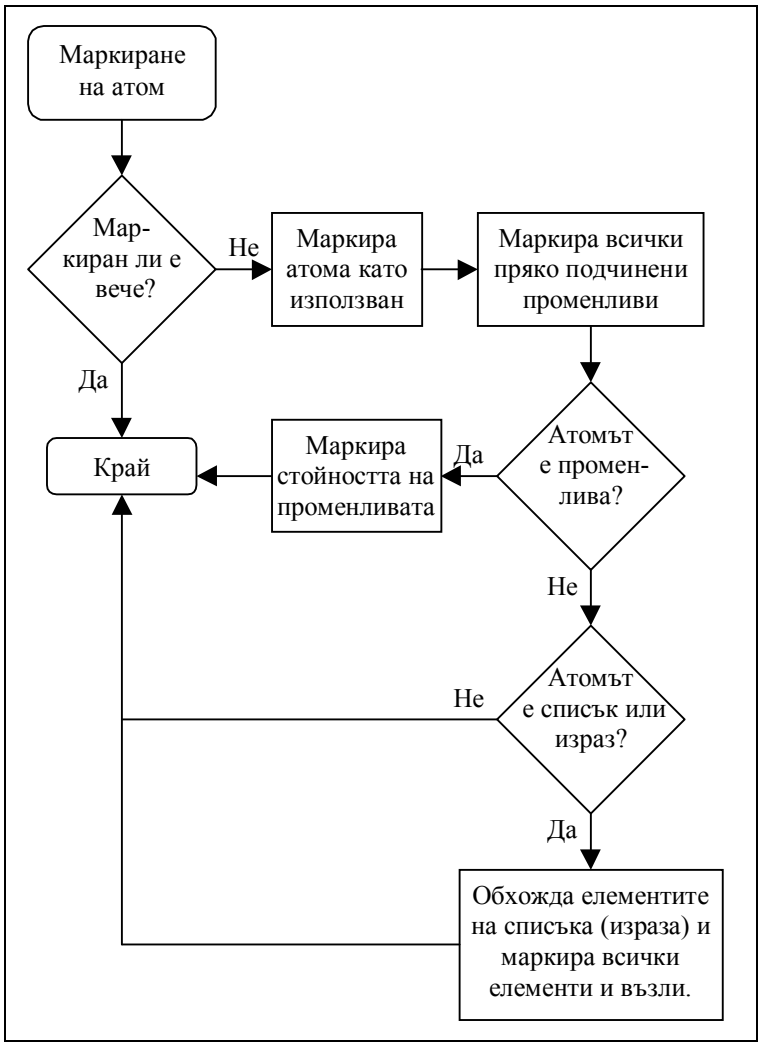
Решението кога и коя памет да се прехвърля на и от диска не зависи и не може да се контролира от Elica. И именно тука си проличава неприятният ефект – от даден момент

нататък, системата започва да работи много бавно, понеже 80%-90% (а понякога и 100%) от системата се използва само за прехвърляне на данни между диска и паметта. Няма естествен начин Elica да разбере кога Windows ще започне приоритетно да прехвърля данни, затова и стандартният критерий за активиране на Garbage Collection не е приложим.

Решението, което е реализирано в ядрото на системата, е да има заявка за Garbage Collection след изпълнението на всяка команда, но някои заявки да се пропускат, така че да не се правят два Garbage Collection прекалено близо във времето. Границата, която експериментално е установена като почти оптимална за различни конфигурации на хардуера и софтуера е около 5 секунди. Това означава, че между два последователни Garbage Collection процеси има свободно време от поне 5 секунди.

Ако този интервал се увеличи, започва напълване на паметта и може да се стигне до неприятната положение системата да е заета единствено със swap файловете си. Ако интервалът се намали пак се получава забавяне, което е породено от това, че малко време системата е генерирала малко атоми за освобождаване и времето се хаби предимно за обхождане на заетите атоми, които наистина се използват и няма нужда да се освобождават.

Фигура III-19 Маркиране на атом



В алгоритъма на Фигура III-18 е използван терминът *маркиране*. В Elica под маркиране на атом се разбира заявяването, че той не трябва да се освобождава от текущия Garbage

Collection. Също така, маркирането се пропагандира по всички връзки, които излизат от този атом.

Процедурата по маркиране на атом трябва да бъде много ефективна, понеже се изпълнява в рекурсия. При всеки Garbage Collection с нея се обхожда цялото дърво на променливите и други допълнителни списъци. За повишаване на скоростта са предвидени няколко критерия, които прекъсват рекурсията. Ако по време на работа се достигне до атом, през който Garbage Collection е минал и маркирал като използван, няма нужда той да се маркира отново и да се проверяват връзките, които излизат от него.

Това се гарантира от изискването докато се прави Garbage Collection системата да не работи – т.е. нито създава, нито променя, нито изтрива атоми.

Съществуват някои по-сложни възможности да се реализира и подобен механизъм - паралелен постъпков Garbage Collection, но това се оказва излишно, защото за следващите версии на системата е планирано да се проектира и реализира разпределен Garbage Collection, при който неизползваните атоми ще се освобождават веднага.

Друг начин да се ускори Garbage Collection е вече използваният механизъм за буфериране на свободните атоми. Както бе обяснено в предишни глави, системата съхранява списъци на свободните атоми. Когато се освобождава атом, той се свързва в тези опашки и при нужда се изважда от там. И двете операции са многократно по-бързи в сравнение с взимането и връщането на памет на операционната система.

III.3 Транслатор

Транслаторът в система Elica е може би най-главната и основна част, чието проектиране и разработване е отнело най-много време и ресурси [10]. В своето развитие системата е преминала през всички основни видове транслатори – както хомогенни, така и хетерогенни.

Като първична реализация на системата, езикът TopLogo++ е чист компилатор. Изходният текст на програмите се компилира директно до машинен код, с което се гарантира бързодействието на изпълнението.

Следващите системи, TGS, LGS и LGSW бяха реализирани като интерпретатори. Съществените преимущества на интерпретаторите при реализиране на език като Logo, е че позволяват да се реализира пълната им функционалност. Някои от възможностите на езика се реализират трудно с компилатори. Например, адресирането на променлива по име, изпълняване на динамично създадена програма, създаване и изтриване на променливи в реално време и т.н.

От следващия вариант на системата – RLS, транслаторът става хетерогенен. Текстът на програмата се компилира до междинен код, който в последствие се интерпретира. По този начин се постига компромисен вариант, понеже въпреки положителните възможности на интерпретаторите, скоростта на изпълнение спада значително.

Компромисът се състои в това, че хетерогенния транслатор предоставя същата гъвкавост като интерпретатора, и по-добра производителност, но все пак тя е по-слаба от тази на компилатора. Цената за постигането на това е необходимостта да се проектират, реализират и поддържат два транслатора – един интерпретатор и един компилатор.

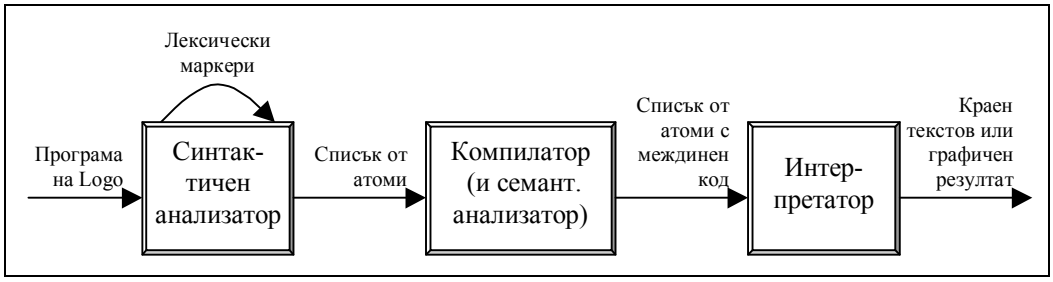
III.3.1 Обща архитектура на транслатора

Архитектурата на транслатора в система Elica следва традиционните представи за подобен модул, с леки промени, породени от естеството на процесите в Elica. Архитектурата може да се разглежда в два аспекта – като описание на елементите (т.е. дескриптивна архитектура) и като взаимовръзката им (функционална архитектура).

III.3.1.1 Дескриптивна архитектура

Дескриптивната архитектура на транслатор описва елементите и функциите им, но не и начина на взаимодействие с останалите елементи. Входните данни за транслатора са програми на езика Logo, които са записани като обикновени ASCII текстове или RTF текстове, съдържащи форматиране. За транслатора форматирането не е от значение и се игнорира.

Фигура III-20 Дескриптивна архитектура на транслатора



Синтактичният анализатор получава тези филтрирани текстове, обхожда ги и създава списък от лексически маркери. Това е списък от данни, показващи положението и размера на всяка лексема от входния текст. С помощта на този списък лексемите се

извличат една по една и се конструира списък от атоми, който представя програмата. Създаването на списъка е рекурсивно, т.е. в списъка може да има подсписъци, като броят на нивата не е ограничен.

Така създаденият списък се подава на компилатора с вграден семантичен анализатор, който определя мястото и размер на всяка команда и идентифицира параметрите й. Имайки тази информация, компилаторът създава междинен код за всяка команда поотделно, който се прикача към нея.

Интерпретаторът не получава директно този код, а списък от команди, към които той е прикачен. Решението да се използва подобен начин на транслация е породено от изискването за динамична (т.е. такава, която се осъществява в реално време) и частична (т.е. не се компилира цялата програма наведнъж) компилация.

III.3.1.2 Функционална архитектура

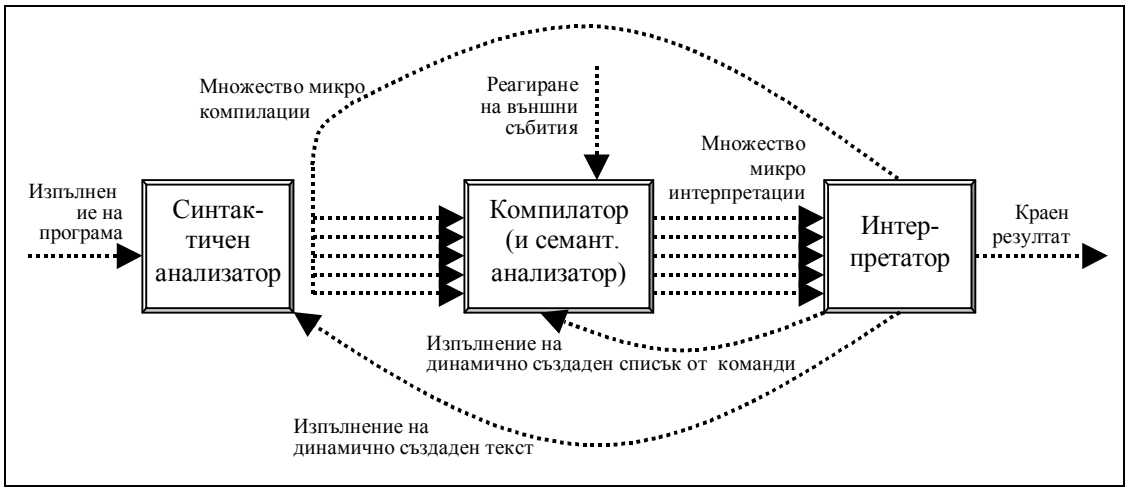
Функционалната архитектура на системата е по-сложна поради няколко фактора:

- необходимост от интерпретиране в реално време
- необходимост от компилиране в реално време.

Синтактичният анализатор, отговорен за конвертирането от текст до списък се използва не само при стартиране на програма от работната среда, но и в случаите, когато трябва да се изпълни динамично създаден текст, който е представен като дума (в термините на Logo), а не като списък от команди. Това налага динамичното използване на синтактичния анализатор.

Създаденият списък от команди не се компилира изцяло поради факта, че някои от конструкциите в него могат да се анализират семантично едва след изпълнението предходните команди. И още по-важно, процесът на компилация се активира от интерпретатора, независимо дали става въпрос за компилиране на динамично създаден списък от команди или за компилиране на основната програма.

Фигура III-21 Функционална архитектура на транслатора



Множествената микрокомпилация поражда и множествена микроинтерпретация. След изпълнение на подадения му междинен код интерпретаторът на Elica изисква от компилатора да компилира следващата команда, която после той интерпретира и т.н.

За да се спести време, компилираният междинен код не се губи, а се запазва. По този начин когато един и същ текст на програмата се изпълни повторно, той не се прекомпилира.

III.3.2 Синтактичен конвертор

Първата стъпка при стартирането на програма на Logo е тя да се преобразува в списък от команди, което е естествения формат на данните в системата. Това конвертиране се извършва от синтактичния анализатор, който е двуфазов.

По време на първата фаза се анализира текста на програмата и се определя началото и дължината на всяка лексема – т.нар. лексически маркери. Втората фаза използва тези маркери, извлича лексемите и генерира списък от команди, съответстващ на текста на програмата.

III.3.2.1 Лексически маркери

Лексическите маркери са структура, чиято цел е да определи мястото и размера на всяка лексема. Основната цел е това да става максимално бързо. В първите реализации на синтактичния анализатор подобни маркери не са използвани. Едва в последната версия, с цел ускоряване на процеса по транслиране на програмите, механизмът на лексическите маркери бе измислен, проектиран и реализиран.

Основата на създаването на лексическите маркери е на всеки знак да се присвои даден тип. На следващата таблица са показани използваните маркери. Особеното е, че за някои от знаците има по два или повече дефинирани маркера. Това се налага поради факта, че в някои случаи знакът може да има друга функционалност. Например, знакът '[' е самостоятелен (SINGLE), но в коментар се преобразува в COMMENT или LINECOMMENT, а ако е заграден в апострофи е IDENT

Таблица III-4 Лексически маркери

Маркер	Значение
IDENT	Знакове за идентификатори. Включват всички знакове, които не спадат към никоя от другите категории.
DELIM	Знакове за разделители. Включват интервал, табулация и край на ред.
PLUS	Знакът '+'.
MINUS	Знакът '-'.
MUL	Знакът '*'.
DIV	Знакът '/'.
POW	Знакът '^'.
COMPARE	Знакове за композиране на оператори за сравнения. Включва '<', '>' и '='.
SINGLE	Знакове, които винаги образуват самостоятелни лексеми. Включват ';', '{', '}', '[', '] ', '(', ')', '"', ':', ',', '\', '!' и CR
COMMENT	Знакът '{' и всички, намиращи се между '{' и '}'.
APOSTRO	Знакът апостроф '.
LINECOMMENT	Знакът ';' и всички, намиращи се до края на реда.

Процесът на генериране на маркери включва еднократно обхождане на текста на програмата и определяне актуалния тип на всеки знак. Определянето на типа не е тривиално особено за случаите, когато един и същ знак може да бъде маркер от различен тип, а също така, маркирането трябва да се справи и със следните допълнителни проблеми:

- при използването на ';' да изолира всички знакове до края на реда
- при използването на '[' и '}' да изолира всички знакове помежду им
- да поддържа многократно влагане на коментари от вида '{'…'}'

- да определя тип IDENT на всички знакове, обградени с апострофи, а вътре в тях всяка двойна употреба на апострофи да третира като единична

Реализацията на подобно маркиране изисква използването на някои допълнителни параметри, като колкото по-малко на брой са, толкова по-лесно е поддържането им, а съответно и скоростта на маркиране.

В системата съществува таблица, в която на всеки знак с ASCII код от 0 до 255 е обозначен стандартният тип на маркер. Най-често, определянето на типа на знака се свежда до еднократна проверка в масива, като за индекс се използва самият знак. За останалите случаи системата поддържа текущата дълбочина по три направления: дълбочина на APOSTRO, на COMMENT и на LINECOMMENT. Първата и третата дълбочини могат да са само на едно ниво, докато втората – може да е с произволен брой нива. Използването на тези дълбочини преопределя типа на всеки знак и единствено, ако и трите са 0, се прави проверка в масива със стандартни типове.

В крайна сметка така установените маркери еднозначно определят лексемите. Всяка поредица от знакове с един и същ тип на маркера се третират като една самостоятелна лексема. В специална вътрешна структура се записват позицията и дължината на всяка лексема, а също и номера на реда, в който се среща. С тези данни се определят част от пространствените координати на лексемите.

III.3.2.2 Генериране на списъци

Наличието на лексически маркери и създадената структура с частични пространствени координати на лексемите са достатъчни за осъществяването на втората фаза от процеса на синтактичен анализ. През тази фаза се конструира списък от думи, който представя текста на програмата. Списъкът може да съдържа други списъци в зависимост от структурата на програмата.

При съставянето на списъка се определят и пространствените координати на всеки от елементите му. Лексемите, се изчитат последователно и еднократно, използвайки лексическите маркери.

По време на тази фаза се прави частичен анализ и промяна на командите. Това се налага от факта, че ядрото на системата не може да обработва конструкцията `TO..END`, която е отпаднала по идеологически причини (по-точно поради унификацията), но е достатъчно стандартна и общоприета. По този начин потребителят ще може да я използва, но системата ще я конвертира в еквивалентна `MAKE` команда.

На Фигура III-22 е показан опростен вариант на конвертирането до списък, израз или подпрограма. И в трите случаи физическата организация е като списък, като разликата е, че при (…) списъкът се нарича *израз*, при TO..END се нарича *подпрограма*, а при […] се нарича *списък*. Другата разлика е, че при различните видове списъци се реагира по различен начин на терминиращите лексеми, които са съответно ")", "END" и "]".

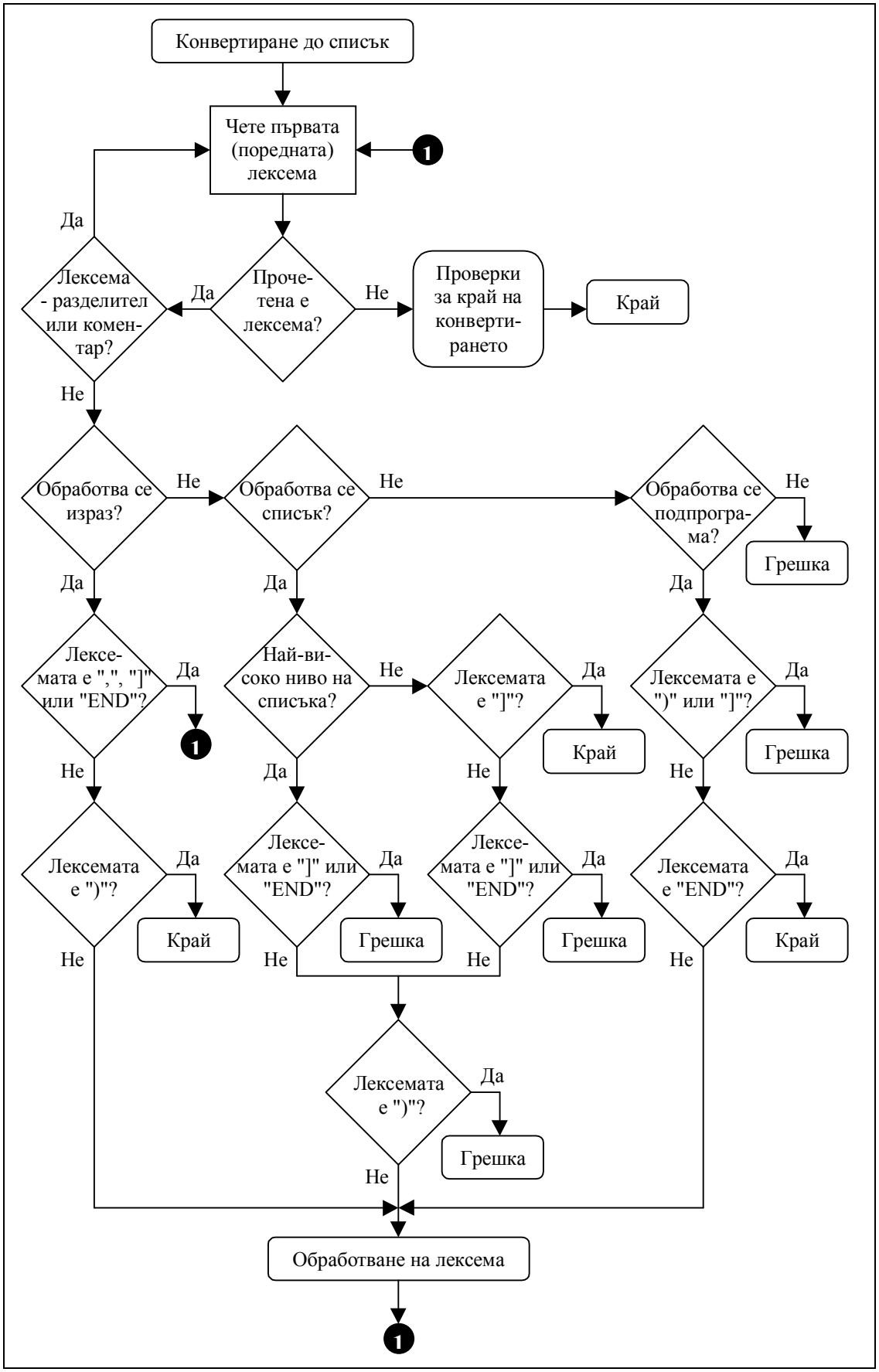
Проверката за край на конвертирането проверява дали всички отворени списъци (без този на най-високо ниво) са затворени по подходящ начин. Ако това не е така, се генерира съответна грешка.

Основните действия със "стандартните" лексеми (т.е. тези, чиято обработка не е показана на Фигура III-22) включват:

- ако лексемата е "TO", се извиква рекурсивно обработване на списък в подпрограма и конвертирането на TO...END в еквивалентна MAKE команда
- ако лексемата е "[" или "(" се извиква рекурсивно обработване на списък или израз

във всички останали случаи лексемата се добавя към текущия списък.

Фигура III-22 Конвертиране до списък



III.3.3 Междинен код

При проектирането на компилатора и семантичният анализатор една от основните цели бе да се проектира и реализира такъв междинен код, че да не се налага той да се променя, или поне промените в него да се сведат до минимум. Другата основна цел бе кодът да е максимално прост и лесен за интерпретиране.

По своята същност междинният код прилича много на асемблер. Междинният код се състои от код за инструкция евентуално последван от допълнителни аргументи, чийто брой и тип зависи от кода на инструкцията.

За компилирането на произволен програмен текст на Elica Logo, системата използва 19 инструкции. Следващият списък дава кратко описание на инструкциите и параметрите им.

1. **NOP**
   Празна инструкция. Без аргументи.
2. **POP**
   Извлича адрес от стека с адреси на атомите. Без аргументи.
3. **PUSH «addr»**
   Вмъква адрес на атом в стека с адреси на атоми. С един аргумент – адреса, който ще се постави на върха на стека.
4. **CALL «addr» «largs» «rargs» «type» «params»**
   Извиква потребителска процедура, чиято дефиниция е в атом с даден адрес, има определен брой леви и десни аргументи, с предварително зададен начин на изпълнение (определящ дали ще се очаква резултат от изпълнението) и списък от параметрите. Всеки от параметрите се задава с двойка числа **«type» «addr»**. Типът определя дали параметъра се предава по стойност или по име, а адресът е адрес на атом, който съдържа името на параметъра.
5. **RET «addr»**
   Прекратява изпълнението на подпрограма. В аргумента се съдържа адреса на атома от който започва следващата команда за изпълнение.
6. **QUOTE**
   Указва, че във върха на стека се намира адрес на атом, съдържащ дума, която трябва да се третира като дума. Към настоящия момент тази инструкция не се използва, понеже компилаторът е проектиран така, че никога да не я генерира. Съответства на Logo функцията **"** (кавички).
7. **EVAL «largs» «rargs»**
   Извлича атом от върха на стека, съдържащ име на променлива. Ако това е обект, създава нова променлива, копира в нея всички полета и стойности от обекта и слага адреса й в стека. В противен случай слага само стойността на променливата. Съответства на Logo функцията **:** (двоеточие).
8. **PERIOD «largs» «rargs»**
   Извлича два атом от върха на стека, съдържащ думи. Конкатенира ги, като помежду им слага **.** (точка) и връща новосъздадената дума в стека. Съответства на Logo функцията **.** (точка).
9. **PRINT «largs» «rargs»**
   Извлича от стека съответния брой атоми и отпечатва стойностите им на екрана. Ако атомът е променлива се отпечатва стойността на променливата, а ако не е – отпечатва се самият атом. Последната отпечатана стойност се връща в стека. Съответства на Logo командата `PRINT`.

10. **MAKE «largs» «rargs» «frominstr» «toinstr»**
    Извлича една, две, три или четири стойности от стека. Съответства на Logo командата `MAKE`. Третия и четвъртия параметър на инструкцията показва къде започва и къде свършва командата. Според броя на извлечените атоми, техните стойности се използват по следния начин:

    - една извлечена стойност: тя е името на променливата, а съответната Logo команда е от вида: `MAKE "Name`
    - две извлечени стойности: ако първата е списък, значи съответства на Logo команда от вида `MAKE [Attribs] "Name`, а в противен случай съответства на `MAKE "Name :Value`
    - три извлечени стойности: ако първата е списък, значи съответства на Logo команда от вида `MAKE [Attribs] "Name :Value`, а в противен случай съответства на `MAKE "Name :Value [Attribs]`
    - четири извлечени стойности: те съответстват на пълния синтаксис на командата и съответства на `MAKE [Attribs] "Name :Value [Attribs]`

11. **RUN «largs» «rargs»**
    Извлича атом от върха на стека. Ако стойността му е дума, се изпълняват командите записани в файл, с име дадената дума. Ако е списък, се изпълняват командите в списъка. Съответства на Logo командата `RUN`.

12. **OUTPUT «largs» «rargs»**
    Извлича атом от върха на стека и неговата стойност става резултат на текущо изпълняваната процедура. Двата параметъра показват колко аргумента да се използват. Допустимо е техния брой да е нула и в такъв случай нищо не се извлича от стека. Съответства на Logo командата `OUTPUT`.

13. **IF «largs» «rargs»**
    Според броя на аргументите извлича два или три атома от върха на стека. Първият атом трябва да съдържа логическа стойност и според нея се решава кои команди да се изпълнят. Тези команди се намират в единия или двата останали извлечени атома. Съответства на Logo командата `IF`.

14. **WHILE «largs» «rargs» «instruction»**
    Извлича два атома от върха на стека. Ако първият съдържа `"TRUE` изпълнява командите, записани във втория и не преминава към следващата инструкция, а се връща и повтаря командата. Важно е да се отбележи, че третия параметър на инструкцията показва на коя инструкция да се върне управлението. Тази инструкция е първата, при която започва изчисляването на аргументите на Logo командата `WHILE` и не съвпада със самата инструкция WHILE.

15. **REPEAT «largs» «rargs»**
    Извлича два атома от върха на стека. Изпълнява командите във втория според числото, записано в първия. Инструкцията съответства на Logo командата `REPEAT`.

16. **LOCAL «largs» «rargs»**
    Извлича от стека съответния брой атоми и създава локални променливи с имена - стойностите на атомите. Съответства на Logo командата `LOCAL`.

17. **DLLCALL «addr»**
    Извиква външна процедура, написана на друг програмен език и компилирана в DLL файл. Аргументът съдържа същинския адрес на процедурата и не трябва да се третира като адрес на атом.

18. **OB «largs» «rargs» «frominstr» «toinstr»**
    Извлича една, две, три или четири стойности от стека. Съответства на Logo

командата OB. Третия и четвъртия параметър на инструкцията показва къде започва и къде свършва командата. Според броя на извлечените атоми, техните стойности се използват по следния начин:

- една извлечена стойност: тя е името на променливата, а съответната Logo команда е от вида: `OB "Name`
- две извлечени стойности: ако първата е списък, значи съответства на Logo команда от вида `OB [Attribs] "Name`, а в противен случай съответства на `OB "Name :Value`
- три извлечени стойности: ако първата е списък, значи съответства на Logo команда от вида `OB [Attribs] "Name :Value`, а в противен случай съответства на `OB "Name :Value [Attribs]`
- четири извлечени стойности: те съответстват на пълния синтаксис на командата `OB` и съответства на `OB [Attribs] "Name :Value [Attribs]`

19. **DUMP**
Помощна инструкция, която предизвиква отпечатването на всички променливи и стойностите им, които са дефинирани в момента.

III.3.4 Компилатор до междинен код

III.3.4.1 Основни изисквания

Компилаторът до междинен код конвертира списък от команди до редица от инструкции. Командите са записани в стандартен списък на Logo, като по този начин може лесно да се компилира списък, създаден в реално време от програмата на потребителя.

Както вече бе описано, процесът на компилацията се разпределя в множество микрокомпилации. При всяка от тях се компилира само по една команда.

При проектирането на компилатора, основните изисквания бяха:

- да се компилира само по една команда
- потребителските процедури да могат да бъдат извиквани с по-малко, с повече или със стандартния брой аргументи
- потребителските процедури да могат да имат приоритет и асоциативност, които определят начина по който аргументите ще се групират и асоциират към процедурите
- потребителските процедури да могат да имат аргументи както отляво, така и отдясно или пък и от двете страни едновременно
- аргументите, които са в повече (или в по-малко) могат да са както отляво, така и от дясно или пък и от двете страни
- за допълнителните аргументи се задават служебни имена, получавани по стандартен начин
- колкото се може по-голяма част от проверките да се направи по време на компилация
- скоростта на компилиране да е голяма
- ако се подадат повече от една команди, то те всичките се компилират

III.3.4.2 Структура TExprRec

Процесът на компилиране създава временни масиви от елементи, в които се съхраняват характеристиките на всички елементи на израза за компилиране. Тази структура съдържа дванадесет полета, всяко от които е важно. Структурите на елементите са индексирани.

- `Name` - атом, съдържащ името/текста на елемента
- `StoredRL` - възел от списъка от команди, към който е прикачен текущия елемент
- `DefLeft` - брой леви аргументи (по подразбиране)
- `DefRight` - брой десни аргументи (по подразбиране)
- `Priority` - изчислен приоритет
- `UsedBy` - индекс на елемент, използващ текущия
- `Lefts` - списък от индекси на елементите, използвани като леви аргументи на текущия елемент
- `Rights` - списък от индекси на елементите, използвани като десни аргументи на текущия елемент
- `LeftArgs` - списък от имената и типа на левите аргументи
- `RightArgs` - списък от имената и типа на десните аргументи
- `Push` - показва дали елементът е проста стойност и като такава, трябва да се компилира до PUSH, или е сложна и трябва да се компилира до CALL.
- `System` - код на инструкция (попълва се само за служебните думи)

III.3.4.3 Фази на компилатора

Процесът на компилиране на команда е сложен и се разделя на няколко фази:

- Зареждане на елементите
- Семантичен анализатор (анализиране на елементите)
- Генериране на машинен код

Най-проста е първата фаза, при която елементите се четат от подадения списък, поставят се в локален масив и се инициализират характеристиките на всеки елемент. Втората фаза е най-сложната и по своята дейност реализира семантичния анализатор в системата. Именно в тази фаза се създава вътрешната йерархия и се реализират повечето от изискванията към компилатора. Последната фаза съдържа генератора на междинен код.

III.3.4.4 Зареждане на елементите

Зареждането на елементите е най-простата и най-бърза фаза в процеса на компилация. Входният списък, съдържащ командата за компилиране, се обхожда еднократно, като за всяка прочетена лексема се създава по един запис от тип TExprRec.

Всеки елемент се анализира самостоятелно и се определя приоритета му. Съставните елементи, тези, които са изрази в (…) или подсписъци от команди в […] се третират като един елемент, който се анализира после рекурсивно. Целта на това е обработването на вложени изрази и команди да не натоварва процеса на компилиране,

и да се обработва по естествен начин. Семантичният анализ през следващата фаза базира своите действия предимно на приоритетите на елементите, затова тяхното изчисляване е от значителна важност.

Факторите, които определят приоритета са няколко:

P – стандартен приоритет на елемента

T – текуща позиция на елемента в израза (индекс на елемента)

A – асоциативност на елемента

Таблица III-5 Характеристики на елементите

Елемент	P	Def Left	Def Right	Left Assoc	System
Подизраз	150	0	0	Да	iNOP
Подсписък	150	0	0	Да	iNOP
Число	150	0	0	Да	iNOP
Дума след "	150	0	0	Да	iNOP
" (кавички)	145	0	1	Не	iQUOTE
: (двоеточие)	140	0	1	Не	iEVAL
. (точка)	149	1	1	Да	iPERIOD
"PRINT"	0	0	200	Не	iPRINT
"MAKE"	0	0	4	Не	iMAKE
"OB"	0	0	4	Не	iOB
"RUN"	0	0	1	Не	iRUN
"OUTPUT"	0	0	1	Не	iOUTPUT
"IF"	0	0	3	Не	iIF
"WHILE"	0	0	2	Не	iWHILE
"REPEAT"	0	0	2	Не	iREPEAT
"LOCAL"	0	0	1	Не	iLOCAL
Идентификатор, дефиниран от потребителя	Зависи от дефиницията. По подразбиране е 20.	Зависи от дефиницията.	Зависи от дефиницията.	Зависи от дефиницията. По подразбиране е "Да"	iNOP

От тях P и A могат да бъдат дефинирани за елемента, но могат и да не бъдат. В такъв случай се приемат стойностите по подразбиране.

След редица експерименти и анализи, формулата за изчисляване на приоритета е определена по следния:

1000*P+T, ако асоциативността е лява

1000*P-T, ако асоциативността е дясна

Коефициентът 1000 е избран така, че да задава подприоритети в рамките на главния приоритет P. Подприоритетите се определят от позицията на елемента и тяхното влияние трябва да е толкова малко, че да не променя главния приоритет. Целта на подприоротетите е да доопредели приоритетната разлика на елементите в случаите когато те имат еднакъв главен приоритет.

Освен приоритета на елементите, за всеки от тях се определят и другите характеристики, по начин показан в Таблица III-.

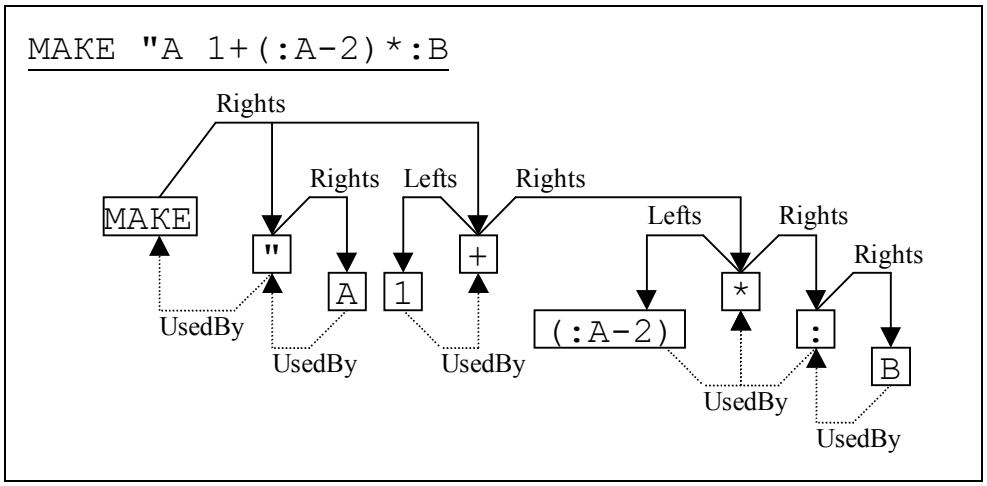
Повечето от останалите полета от TExprRec се инициализират с нулеви стойности, които се актуализират и използват през някоя от следващите фази на компилацията.

III.3.4.5 Семантичен анализатор

Втората фаза на компилатора е семантичния анализатор. В тази фаза се определят взаимоотношенията на отделните елементи и се построява йерархията между тях. По време на предходната фаза полетата Lefts, Rights и UsedBy са неинициализирани. Те определят вътрешната йерархия между елементите.

За построяването на йерархията се използва предимно приоритетът, който определя реда на обработване на елементите. При обработването на всеки елемент се намират всички неизползвани елементи, които могат да му бъдат аргументи. Това, естествено зависи от наличието на свободни елементи, от подразбиращият се брой на леви и десни параметри на елемента и от типа на компилация.

Фигура III-23 Йерархия на елементите при компилация



Типът на компилация може да е *нормален* или *пълен*. При нормалната компилация към всеки елемент се прикачват толкова аргументи, колкото може, но не повече от подразбиращия се брой. По този начин изразът `:a+:b :c` се декомпозира на два отделни подизраза `:a+:b` и `:c`. Пълната компилация се стреми да създаде само един израз, като според приоритетите на свободните елементи прикачва "по-слабите" изрази към "по-силните". По този начин същият израз би се компилирал като с една водеща операция +, която има един ляв аргумент `:a` и два десни `:b` и `:c`. Част от този анализ се извършва и през следващата фаза.

На Фигура III-23 е показана вече построената йерархия на командата `MAKE "A 1+(:A-2)*:B`. От функционална гледна точка няма нужда от поддържането на двойно-свързана йерархия (или две независими йерархии), но от практическа се налага, поради приложимите оптимизации и ускорения.

Йерархията отгоре-надолу е ориентирана и разделя подчинените елементи на всеки елемент на леви и десни. За обратната йерархия не е нужно такова разделение, понеже всеки елемент може да има най-много един родител.

III.3.4.6 Генериране на машинен код

Последната фаза на компилатора е генерирането на междинния код. След построяването на йерархията на елементите, генерирането на код е относително просто, като се изключат някои особености.

На най-първо ниво, генерирането се извършва по два различни начина, според типа на компилация. Както вече бе обяснено, при нормална компилация операторите, функциите и процедурите не използват насилствено повече аргументи отколкото им

трябват. При пълното компилиране целта е командите за компилация да се сведат до един единствен израз.

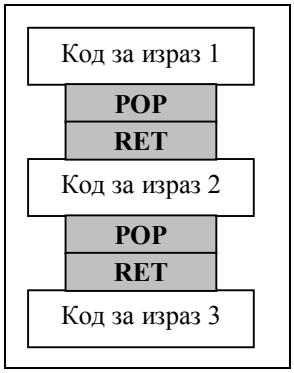
В процеса на компилация винаги се използва нормалният тип, но когато се налага да се компилира израз в скоби (…), се използва пълна компилация. Това се дължи на синтактичната особеност на езика Logo да се подават повече (или по-малко) аргументи чрез заграждане на израза в скоби.

За генерирането на кода се използва функция, която се стартира от даден елемент и генерира междинния код, съответстващ на него и на всичките му подчинени елементи.

При пълната компилация кореновият елемент е единствен и затова тази функция се изпълнява за него.

При нормалната компилация е възможно да има няколко коренови елемента и функцията се изпълнява за всеки един от тях поотделно. Между инструкциите за изразите се поставят допълнителните инструкции **POP** и **RET**, понеже резултатите от всички изрази без първият трябва да се игнорират от системата.

Фигура III-24 Нормална компилация



Генерирането на код за конкретен елемент става по еднотипен начин. Първоначално се генерират рекурсивно междинните кодове на всички аргументи на елемента, като се започне от най-десния и се стигне до най-левия. Създаването в обратен ред е избрано да бъде така, защото после при интерпретация, стойностите на аргументите да се изваждат от стека в правилния ред.

Следващата стъпка зависи от типа на елемента. Ако той е израз (т.е. задължително е бил заграден с кръгли скоби), извиква се нова микрокомпилация, която да го обработи. Ако елементът е такъв, че трябва да се постави в стека (т.е. е число, дума или списък), то се генерира инструкцията **PUSH**.

Останалите случаи се делят на три класа: елементът съдържа служебна дума, извикване на функция от външен DLL модул или пък е извикване на потребителско дефинирана функция.

Ако елементът е служебна дума се генерира съответната инструкция, като за инструкциите **WHILE**, **MAKE** и **OB** се генерират и допълнителните аргументи.

Ако елементът съдържа обръщение към външна функция, името на елемента, което трябва да е от вида DLL.<FileName>.<FuncName> се декомпозира на име на файл и име или номер на функция. След това се претърсва дадения файл за наличие на функция с исканото име или номер и ако се намери, се генерира съответната **DLLCALL** инструкция с аргумент реалният адрес на функцията.

Ако елементът е потребителска функция се генерира съответната инструкция **CALL**, към която се добавят следните данни:

- указател към атома на променлива на функцията

- брой на реално подадени леви и десни аргументи
- последователно за всички леви аргументи се отбелязва типа им и името им
- същото се прави и за десните аргументи

При генерирането на описанията на левите и десните аргументи, проблемите със скобите, с липсващите или с допълнителните аргументи са вече решени. За липсващите аргументи не се създават описания, а за допълнителните за име се определят служебните имена `#1`, `#2`, `#3` … за десните допълнителни аргументи и `#-1`, `#-2`, `#-3` … за допълнителните леви аргументи.

III.3.5 Интерпретатор на междинен код

Цялата подготовка и създаване на междинен код имат една основна цел – да направят интерпретирането на програмата колкото се може по-бързо. Затова и междинният код е проектиран така, че интерпретаторът да е прост, бърз и ефективен.

Следва списък на инструкциите и начина на интерпретацията им:

1. **NOP**
   Към настоящият момент тази инструкция не се генерира, затова и не се обработва от интерпретатора.
2. **POP**
   Извлича данни от работния стек.
3. **PUSH**
   Поставя стойност в работния стек.
4. **CALL**
   Създава нов контекст и дефинира в него локални променливи по една за всеки наличен аргумент. В рамките на новия контекст изпълнява командите от процедурата. Ако извикването е процедурно, локалните променливи и контекстът се премахват. Ако извикването е функционално и има получен резултат (с командата `OUTPUT`) той се добавя в стека. В противен случай се създава нова променлива и локалните променливи се преместват в нея. Тази локална променлива се връща в стека и в последствие се използва като инстанция на новосъздаден обект.
5. **RET**
   Прекратява изпълнението на подпрограма и определя списъка със следващите команди.
6. **QUOTE**
   Към настоящият момент тази инструкция не се генерира, затова и не се обработва от интерпретатора. В процеса на генериране на междинния код всяко срещане на служебната дума **"** се игнорира, понеже след работата семантичния анализатор не носи никаква допълнителна информация.
7. **EVAL**
   Извлича атом от върха на стека, съдържащ име на променлива. Ако променливата е проста (т.е. няма подчинени променливи), нейната стойност се връща в стека. В противен случай се създава нова променлива, в която се копират полетата и тази променлива се връща като инстанция на обект обратно в стека.
8. **PERIOD**
   Извлича два атома от стека, които съдържат имена на променливи. Двете имена се сливат, като между тях се слага точка. Това новокомпозирано име се връща обратно в стека.

9. **PRINT**
   Извлича от стека съответния брой атоми и отпечатва стойностите им на екрана. Ако атомът е променлива се отпечатва стойността на променливата, а ако не е – отпечатва се самият атом. Последната отпечатана стойност се връща в стека, но ако е променлива се "откача" от родителя си и става самостоятелна променлива.

10. **MAKE**
    Извлича една, две, три или четири стойности от стека. С тяхна помощ създава нова променлива или се променят характеристиките й. Броят и типът на аргументите определят данните, които се подават на универсална системна функция за създаване на променлива.
    Ако е наличен само един аргумент, създава се празна променлива.
    Ако аргументите са два и първият е дума, осъществява се обикновено създаване на променлива и присвояване на стойност, но ако първият аргумент е списък, се прави създаване на променлива.
    При три аргумента, ако, първият е дума, създава се променлива, присвоява й се стойност и се задават допълнителни характеристики. Ако първият аргумент е списък, първо се създава променливата, после се задават характеристиките й и едва след това се присвоява стойност.
    Последният случай е когато аргументите са четири. Тогава имаме всички данни за пълната разширена команда `MAKE`. При нея се създава променлива, задават се характеристиките й, присвоява й се стойност и отново се задават характеристики. И в четирите случаи към създаване на променлива се преминава само ако тя не съществува, в противен случай се използва наготово.
    Друга основна дейност, която се извършва при интерпретирането на инструкцията MAKE, е да се поддържат връзките между новопроменен обект и свързаните с него обекти.

11. **RUN**
    Извлича атом от върха на стека. Ако стойността му е дума, се изпълняват командите записани в файл, с име дадената дума. Ако е списък се изпълняват командите от списъка. В първия случай се компилира и интерпретира списъкът, а във втория първо се прочита файл, който се подава на лексическия анализатор, който от своя страна го конвертира до списък от команди, подаван в последствие на компилатора и на интерпретатора.

12. **OUTPUT**
    Извлича атом от върха на стека и неговата стойност става резултат на текущо изпълняваната процедура. С тази инструкция не се излиза от текущата процедура, а само се дефинира резултата, който ще се върне при излизането от нея. Излизането се осъществява от инструкцията **RET**.

13. **IF «largs» «rargs»**
    Според броя на аргументите извлича два или три атома от върха на стека. Проверява се стойността на първият атом и ако е `"True`, списъкът във втория атом се компилира и изпълнява. В противен случай това се прави със списъка в третия аргумент.

14. **WHILE**
    Извлича два атома от върха на стека. Ако първият съдържа `"TRUE` изпълнява командите, записани във втория и не преминава към следващата инструкция, а се връща и повтаря командата. За да може командата да се повтори е необходимо не само управлението да се върне в началото на инструкцията, а още по-напред – там, където се намират инструкциите за изчисляване на аргументите. Номерът на инструкцията, към която трябва да се върне управлението, се записва като аргумент на инструкцията **WHILE**.

15. **REPEAT**
    Извлича два атома от върха на стека. Изпълнява командите във втория според числото, записано в първия.

16. **LOCAL**
    Извлича от стека съответния брой атоми и създава локални променливи с имена - стойностите на атомите. Не се прави проверка дали няма повторение на създаваните имена. Името на последната създадена променлива се използва като резултат от командата и се връща обратно в стека.

17. **DLLCALL**
    Извиква външна процедура, написана на друг програмен език и компилирана в DLL файл. Аргументът съдържа същинския адрес на процедурата. Като аргументи на функцията се подават структури, които позволяват на външната функция да има достъп до текущото състояние на системата и глобалните и локалните променливи, достъпни в момента.

18. **OB**
    Обработва се от същата функция, която се използва при инструкцията MAKE.

19. **DUMP**
    Помощна инструкция, която предизвиква отпечатването на всички променливи и стойностите им, които са дефинирани в момента. Отпечатването започва от кореновата променлива и рекурсивно се спуска по всички други достъпни променливи. Системно създадените променливи, които не са вързани към дървото на променливите не се обхождат и за тях не се отпечатва никаква информация.

III.4 Поддържане на връзки между обектите

Създаването и използването на обекти и инстанции не представлява теоретичен и практически интерес, ако не се добавя нещо ново и уникално, което или липсва в други обектноориентирани среди или е слабо развито.

Относно обектите в система Elica може да се твърди, че новостите са три:

- Начинът на дефиниране, създаване, управление и унищожаване на обекти със средствата на езика Logo;
- Начинът на реализация на обектите, за постигане на пълната им унификация с променливите, операторите, процедурите, функциите и масивите;
- Връзките между обектите.

В настоящата глава ще бъде описана реализацията на връзките между обектите, които позволяват да се използват свойствата описани в предишни глави.

III.4.1 Основни проблеми и изисквания

Основните проблеми и изисквания към реализацията на връзките между обектите се отнасят не само до начина на реализация, но и до начина, по който те ще се използват от потребителя.

Връзките между обектите трябва да са разбираеми за програмиста и да са максимално прости. Те трябва да се базират на съществуващите средства в езика и да не изискват въвеждането на нови запазени думи и синтактични конструкции. Достъпът до връзките трябва да е по същия начин, както и достъпът до коя да е друга променлива. Трябва да има начин за стандартно създаване на връзки, а също така да се дава възможност на програмиста да проектира и създава собствени връзки между обектите.

От гледна точка на вътрешното представяне, връзките не трябва да изискват нови типове структури, а да се базират на вече съществуващите.

От функционална гледна точка връзките между обектите могат да са двупосочни, циклични, ациклични, да бъдат създавани и забранявани. Връзките да могат да се отнасят не само за обекти, но и за променливи, функции, процедури, оператори и т.н. дори и за описанието на обекти.

Проблемите, които се срещат при реализацията на тези изисквания са липсата на подобни системи, при които се създават връзки над обекти, за които системата не "знае" нищо предварително.

Съществуват редица системи с връзки между обектите, но при тях обектите са фиксирани и връзките между тях също са фиксирани [4, 12, 22]. Потребителят не може да създава нови обекти и нови видове връзки между тях.

III.4.2 Предишни реализации

III.4.2.1 Връзки в PGS

От системите, предшественици на Elica, първата, която поддържа връзки между обектите е PGS [12, 25, 26, 27]. При нея връзките се създават единствено от ядрото на системата и начинът на връзка зависи от обектите.

Фиксираността на типовете обекти и възможните връзки ограничават потребителя. Автоматичното свързване на обекти е на база на синтактичните им зависимости. Т.е. ако променливата A участва пряко в израз, формиращ стойността на B, то само тогава ще се създаде връзка от A към B.

Съществено ограничение е и невъзможността един обект да зависи от друг по няколко начина, а също така и да се създава циклична зависимост.

Въпреки тези недостатъци, връзките в PGS имат и съществени положителни характеристики:

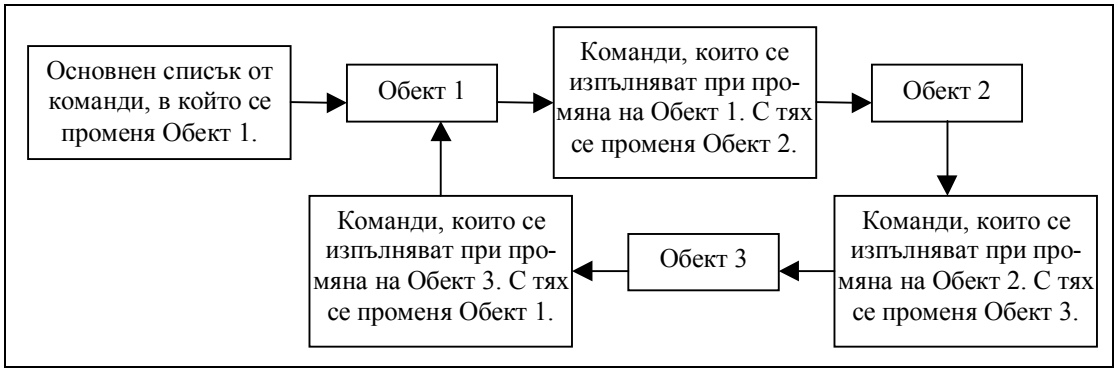
- те са лесни за разбиране и не представляват проблем дори и за начинаещи
- те са лесни за използване – на потребителя не се налага да прави нищо, освен да указва кога да се създава връзка и кога не
- при нужда, потребителят може да контролира някои от параметрите на връзките – може изцяло да се прекъсва връзката между обектите, а също така и частично. В PGS връзките са двойствени – т.е. всяка връзка между две променливи се записва като две релации. При едната, първата променлива е зависеща от втората, а при другата – втората влияе на първата. Всяка от тези две релации може да се манипулира независимо от другата
- чрез разработените примери на връзките в PGS се доказва, че въпреки че функционалността е по-ограничена (спрямо тази, която е била търсена), практическото й използване е успешно.
- независимо, че вътрешната реализация на връзките в PGS е недокументирана и неизвестна, малкото данни, които бе възможно да се съберат даде тласък при проектирането и реализирането на връзките в Elica. Всъщност моделът на връзките в Elica съдържа основните компоненти на модела в PGS, но с редица подобрения.

III.4.2.2 Връзки в предишни версии на Elica

При първите системи, реализирани след PGS, връзка между обектите не е била проектирана и е липсвала изцяло. В някои от по-късните версии (до RLS включително) е имало връзки между обектите, различаващи се от сега реализираните.

Първите опити за връзки са били на принципа да предоставят базисна функционалност, която да може да се разширява от потребителя. За яснота, ще разгледаме пример с циклична зависимост, показан на следващата фигура.

Фигура III-25 Циклична зависимост



Изборът на цикличен пример не е случаен. Оказва се, че това е един от ключовите примери, които създават проблеми. В общия случай циклите може да не са толкова елементарни. В примера са използвани три обекта, като третия зависи от втория, втория от първия, а първият от третия.

Най-първата реализация на връзки изпълняваше правилата (това са командите, които се активират при промяна на обект) веднага след промяна на обекта, към който са асоциирани. Такъв механизъм за връзки създава проблеми при циклите. За да се защити от подобни проблеми, във всяка променлива се дефинира флаг за активност, който

показва дали променливата е била променена и в резултат е започнало изпълнението на правилата към нея, а в същото време те още не са завършили.

Активните променливи разрушават циклите и не позволяват един и същ обект да се активира многократно. Дейностите, които се изпълняват са следните:

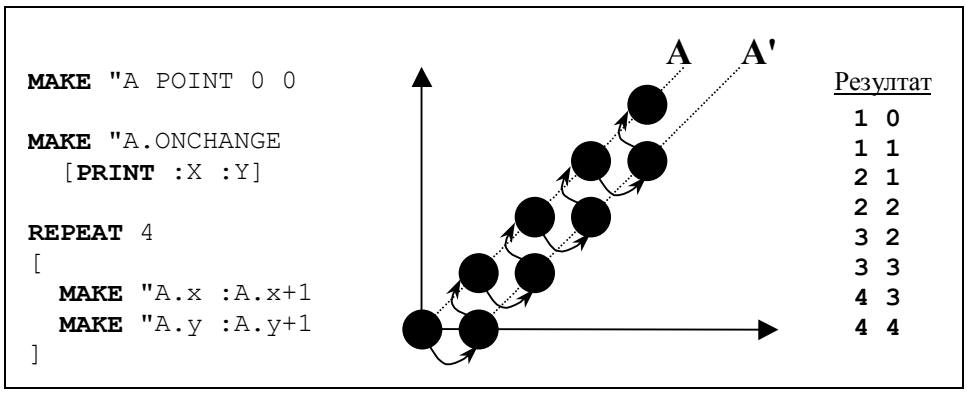
1. При изпълняване на основния списък от команди се променя Обект 1.
2. Обект 1 се активира.
3. Изпълняват се правилата на Обект 1 и в резултат се променя Обект 2.
4. Обект 2 се активира.
5. Изпълняват се правилата на Обект 2 и в резултат се променя Обект 3.
6. Обект 3 се активира.
7. Изпълняват се правилата на Обект 3, но Обект 1 не се променя, защото е вече активен.
8. Обект 3 се деактивира.
9. Обект 2 се деактивира.
10. Обект 1 се деактивира.
11. Изпълнява се следващата команда от основния списък от команди.

Понеже в примера правилата са симетрични, ако промяната започне не от първия, а от някой от другите два обекта, процесът по удовлетворяването на правилата ще е аналогичен.

Независимо от своята елегантност и естественост, представеният алгоритъм създаваше редица проблеми при практическото му използване. Най-очевидният от тях е при промяна две или повече полета на обект, към който има свързани правила.

Ако си представим, че към точка A е дефинирано правило `[PRINT :X :Y]`, с което при промяна на точката се отпечатват координатите й, плъзгането на точката по диагонална права **A** ще генерира двойно повече събития от необходимото.

Фигура III-26 Генериране на "излишни" точки



"Излишните" точки са по диагонала **A'**. Те са страничен резултат от факта, че за всяка промяна събитието `OnChange` се изпълнява веднага.

За да се избегне този проблем, се наложи да се направи нова модификация в механизма за обработване на връзките между обектите. При него, на всяка промяна не се реагира веднага, а заявките се буферират и периодично се изпълняват. При буферирането се запомня само последната заявка към дадена променлива. По този начин, ако тя се промени неколкократно, само последната заявка ще е валидна, а всички преди нея се игнорират. Съществен недостатък на тази идея е трудността при определянето на интервала от време, за който ще се прави буферирането и изчистването на буфера.

Оказва се, че съществуват редица примери, които си противоречат един на други – т.е. ако интервал, избран по един начин работи добре за един пример, прави другия пример негоден и обратно. Примерите, за които се получава това противоречие са тривиални и проблемът не може да бъде класифициран като рядко срещана "особеност".

След дълги експерименти, най-успешният интервал (т.е. този, при който качеството на работа е най-задоволително) е между изпълненията на всеки две команди от най-високо ниво.

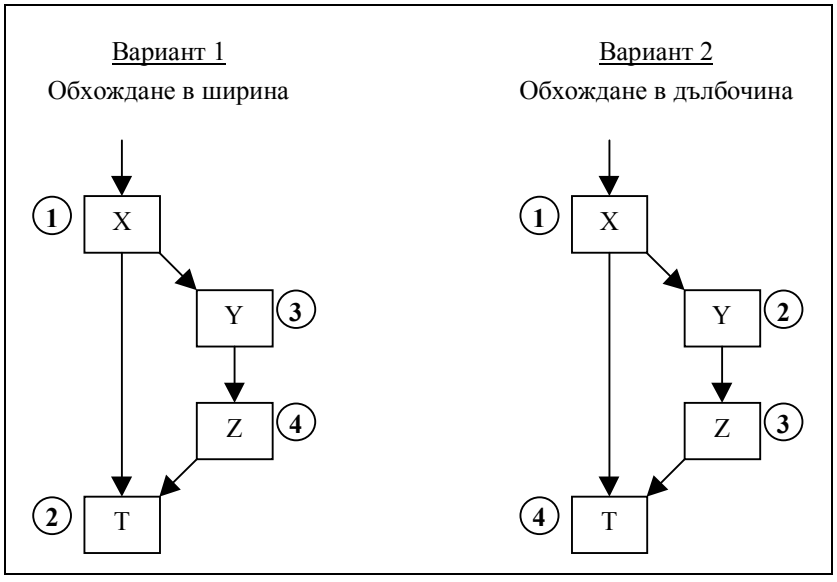
За съжаление, основният недостатък на подобно решение бе, че един и същ програмен текст като програма и като тяло на подпрограма не работи по еднакъв начин – нещо, което би създало съществени трудности на потребителите.

Следващите модификации на връзките включваха използването на именувани и анонимни връзки. Първоначалната цел, поради която те са въведени, бе да може да се изтриват и променят конкретни правила. При анонимните правила всички команди се събират в един списък, към който може само да се добавя. С въвеждането на именуваните правила, всяко отделно правило може да бъде променено или изтрито. Естествено, анонимните правила се запазват, заедно с тяхната специфика.

Известно време след въвеждането на двата вида правила, бе установено къде е основният проблем при връзки. Той се забелязваше при конструкции, където между два обекта съществуват две или повече преки или косвени връзки. При някои примери удовлетворяването на връзките трябва да обхожда графа на връзките в дълбочина, а при други – в ширина. За системата по това време бе невъзможно да реши в кой случай кой начин на обхождане да се избере, още повече, че съществуваха примери, при които трябваха и двата начина на обхождане – единият за една група от обекти, а другият – за друга.

Два от най-елементарните противоречащи се примери са следните: конструкцията триъгълник и двойната зависимост. Всеки от тях ще бъде описан накратко по-долу. При двойната зависимост се конструират следните връзки: $X \to Y$, $Y \to Z$ и $(X, Z) \to T$ - т.е. промяната на X влияе на Y, Y – на Z, а X и Z - на T.

Фигура III-27 Двойна зависимост



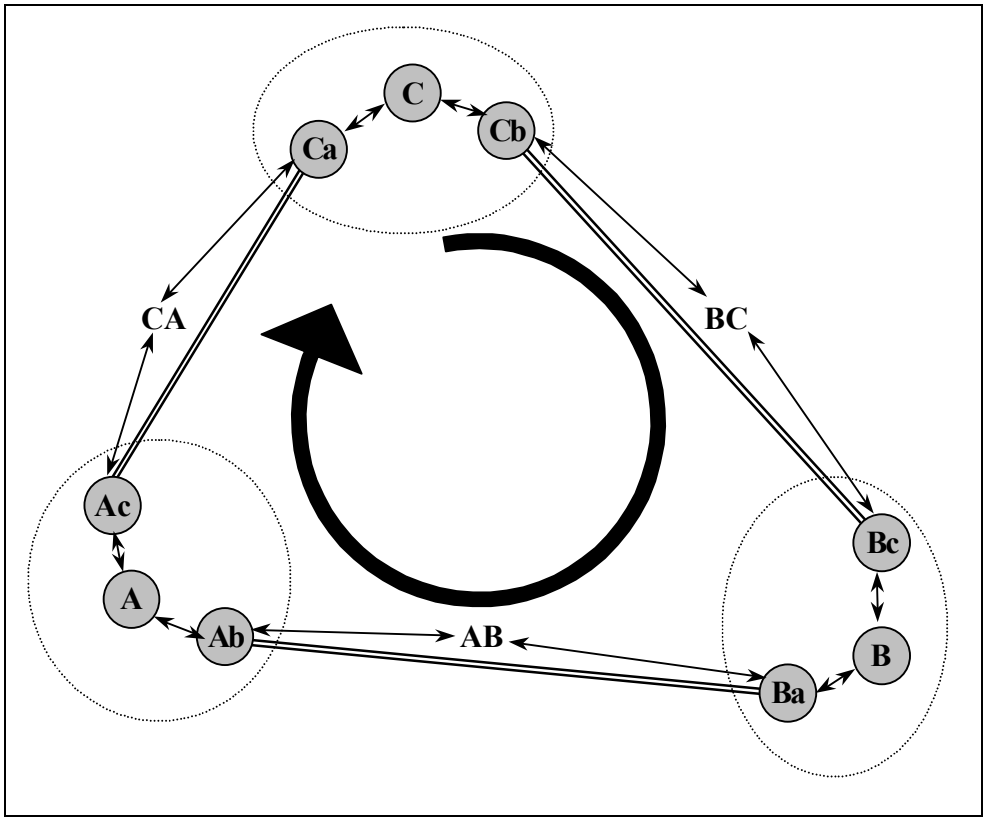
При първият вариант (Обхождане в ширина) крайният резултат, който ще се получи ще е грешен, понеже стойността на T ще бъде пресметната за актуалното X и за все още неактуализираното Z, но понеже T вече ще е пресметнато, когато обхождането стигне до Z няма да предизвика повторна промяна на T.

При вторият вариант (Обхождане в дълбочина) всичко е наред. Редът на актуализиране X, Y, Z, T е именно този, който трябва да бъде. Въпреки това, съществува друга реална опасност – крайният резултат може и да е грешен, в зависимост от това в какъв ред се обхождат връзките. На примера по-горе умишлено е избрана дясната връзка, при която

резултатът е коректен, но ако обхождането започне от лявата – крайният ефект ще е същият като при първия вариант.

Това беше пример, когато единствено обхождането в дълбочина дава верен резултат, но за съжаление, не винаги. Другият пример, при който правилен резултат се получава при обхождане в ширина, е конструкцията на триъгълника. В нея има три основни точки, три отсечки, като всяка отсечка се определя от крайните си точки. По този начин конструкцията се състои от 9 точки и 3 отсечки. Във всеки връх на триъгълника има по три съвпадащи точки – истинския връх на триъгълника и двете крайни точки на отсечките към този връх.

Фигура III-28 Циклична зависимост



На фигурата по-горе зоните в пунктираните елипси са точките, които трябва да съвпадат поради дефинираните правила. Правилата са двупосочни и са обозначени със стрелки.

Показаната конструкция не е стабилна, ако връзките се обхождат в дълбочина. Като контрапример може да се опише следният сценарий:

1. Променя се точка C.
2. Без да се ограничава общността, избираме в коя посока да започне обхождането. В случая е избрано по часовниковата стрелка, но това не е съществено, защото конструкцията е симетрична. В резултат се променя Cb.
3. От това следва, че се променя BC, а после Bc, B, Ba, AB, Ab, A, Ac, CA и Ca и то именно в този ред.
4. Промяната на Ca, не предизвиква промяна на C, понеже C е вече активно (т.е. променено).
5. С това, обхождането в дълбочина завърши. Нищо друго не се променя, защото всички обекти са били вече променени по веднъж.

От гледна точка на потребителя, реална промяна е настъпила само в точката C и отсечката BC. Всички останали обекти не са си променили позицията на екрана, което означава, че връзката между точка C и отсечка CA е разрушена и C не съвпада с края на CA.

Тези два проблема, описват главните проблеми, които трябва да бъдат решени. Намирането на механизъм за поддържане на връзките между обектите, който работи добре с именно тези два примера, би работил и с всички останали, които са били пробвани в процеса на разработка на системата. Естествено, това не гарантира, че този механизъм е универсален, но поне до сега не е намерен контрапример.

III.4.3 Окончателен механизъм на връзките

III.4.3.1 История на създаването му

Създаването на окончателния механизъм на връзки, който работи добре с всички изпробвани примери, и засега показва най-добра устойчивост, е проектиран на базата на два процеса.

Първият процес бе опит да се анализират и класифицират проблемите и опит да се намери общ начин, който да ги решава. Вторият процес бе нещо като реверсивно проектиране. За реализацията на връзките в PGS се знаеше единствено, че то се прави чрез два списъка към всеки обект [12]. В единия списък се описват променливите, на които влияе, а в другия – от които се влияе. Понеже хората, реализирали механизмите на връзки в PGS не бяха достъпни, реверсивното проектиране целеше да се намери най-правдоподобното обяснение на смисъла на двата списъка. Във всички предишни реализации на връзки в Elica се използва само това, на кои променливи влияе дадена променлива, като няма никакво значение от кои зависи [25].

В резултат на тези два процеса се проектира и реализира механизъм, който също се базира на двойка списъци, асоциирани към всяка променлива, и който решава проблема. Естествено, начинът на използване на тези списъци не е един и същ, най-малко поради това, че в PGS не могат да се създават циклични и множествени връзки, докато в система Elica това е възможно.

Другата поне външна близост със система PGS е тази, че този механизъм наложи разделянето на командата `MAKE` в два варианта. До този момент всичко се реализираше само с `MAKE`, но новата обработка на изискваше нова подобна команда и тя естествено стана командата `OB`. По този начин приликата в PGS на високо ниво става още по-голяма. Тази прилика бе увеличена още повече, след като се оказа, че с командата `OB` се създават автоматични връзки, докато с `MAKE` – те не се създават. Това позволи голяма част от примерите в PGS да бъдат използвани в Elica без почти никакви промени.

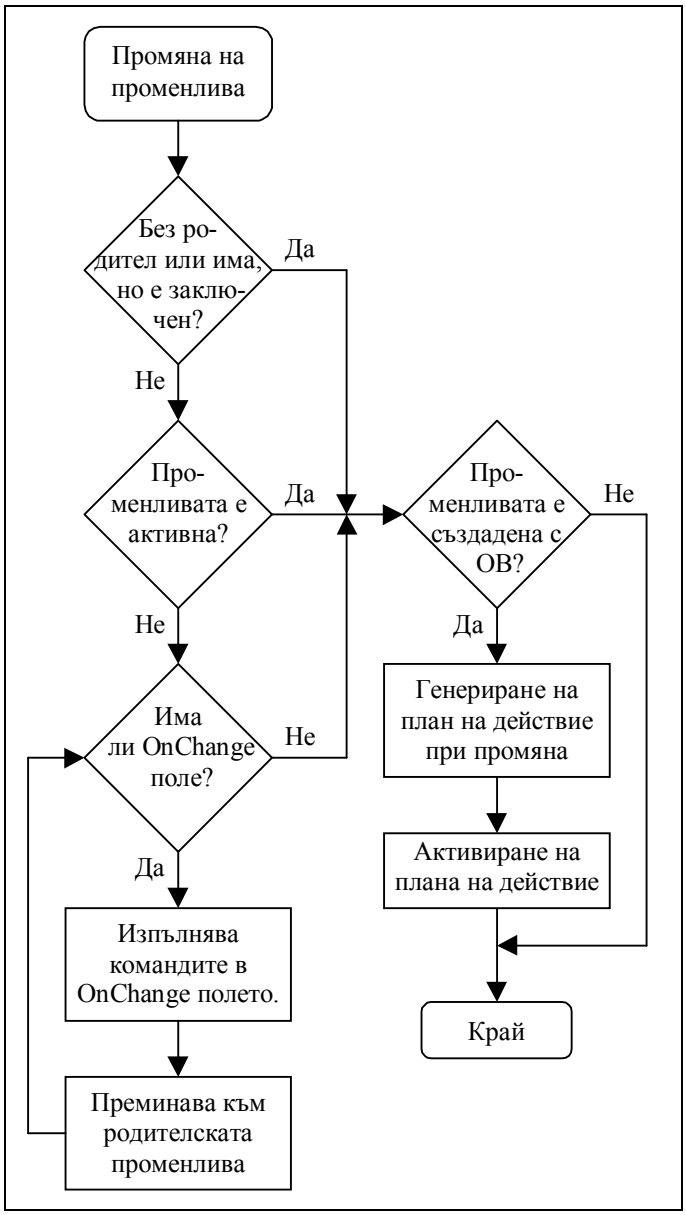
III.4.3.2 Общо описание на механизма

Особеностите на реализацията подчертават съвместимостта на избрания в Elica механизъм с предишно реализираните механизми [26]. Както и при тях съществуват анонимни и именувани правила. Анонимните правила се акумулативни, т.е. присвояването на нова стойност не премахва старата, а я допълва. Именуваните правила се "държат" като нормални променливи.

Вътрешната структура на събитията е същата. Те се записват като полета с имена `OnChange` или `OnChange.<name>`. Разликите са главно в изпълняването на командата OB в начина, по който се реагира след промяна на променлива. Както се вижда от следващата фигура, промяната на променлива се обработва бързо и без използването на сложен алгоритъм. Това е направено така, за да може системата да работи по-бързо. Тежестта е пренесена в отделна процедура, която генерира плана на

промяната. Тя се извиква за всеки обект, за който няма създаден план – т.е. в повечето случаи се изпълнява еднократно.

Фигура III-29 Механизъм на промяна



Генерирането на плана е най-отговорната част в механизма на връзките. По негово време се анализират взаимоотношенията между обектите и се създават автоматични команди, които се прикрепят към `OnChange` събитията. По този начин, за всеки обект се определя какви изменения ще настъпят в цялата конструкция след промяната му.

III.4.3.3 Активиране на план на действие

Планът на действие, чиято генерация ще бъде описана в следващата точка, съдържа списък от команди, които трябва да се изпълнят след промяната на дадена променлива.

Активирането на плана се осъществява винаги след евентуалното му генериране. По този начин, различните сценарии може да са няколко:

- за обекта има стар генериран план, генерацията не създава нов план, активира се стария план – това е най-често срещаната ситуация

- за обекта няма генериран план, генерацията създава нов план, който се активира – този сценарий се случва при първоначалното създаване на план
- за обекта няма генериран план, по време на генерацията не се създава нов (примерно, защото не се налага, или пък обекта няма връзки с други обекти и т.н.) и впоследствие не се активира нищо

В крайна сметка, активирането на план става само тогава, когато има съществуващ план, било то нов или стар. Самото активиране не може да се осъществи директно. Въпреки, че в плана се съдържат команди, всяка от тях трябва да се изпълни в контекста на променливата, спрямо която са генерирани. Поради тази причина, изпълнението става команда по команда, като за всяка се създава подходящия контекст.

III.4.3.4 Генериране на план на действие

Създаването на плана на действие е процес, който се стреми да определи взаимовръзките, които са валидни при промяна на конкретна променлива. Планът винаги се отнася за конкретна променлива и за това за всяка променлива, за която има нужда от съставяне на план, се създава уникален план.

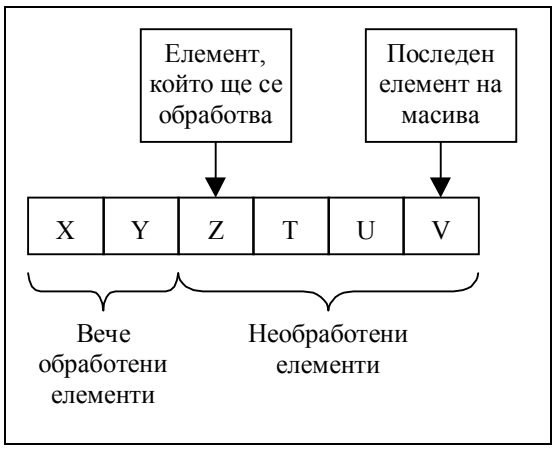
Алгоритъмът на генерация прилича на този, който се използва при сортиране на частично наредено множество. Ако приемем, че връзките между обектите създават подредба от вида "ако промяната на A изисква промяна на B, то A<B", генерирането на план за променлива X означава да се подредят всички променливи, които са в релация с X и са по-големи от X. За алгоритъма се твърди, че е почти същият, защото връзките между обектите могат да са циклични, което би довело до две взаимно противоречащи твърдения – може да се окаже, че A<B и в същото време B<A.

За да се избегне подобни проблеми, по време на генерация автоматично се елиминират част от връзките, а именно тези, които биха направили проблем. Тази елиминация е валидна само по време на съставянето на план. Ако имаме три променливи, за които са валидни следните връзки: A<B, B<C и C<A, и променим променливата B, при изчисляването на нейния план ще се запазят само връзките B<C и C<A, а първата връзка A<B ще бъде елиминирана, понеже тя създава цикъл. Ако пък променим A, тогава ще се запазят A<B и B<C, а елиминирана ще е връзката C<A.

Алгоритъмът на генериране използва масив от атоми, към който са прикрепени два указателя. Масивът е динамичен. Първият указател показва кой елемент от масива подлежи на обработка, а вторият - края на масива. По време на обработката на елемент могат да се създават нови елементи на масива, които винаги се поставят в края му, като променят крайния указател. Процесът по генерацията завършва, когато са обработени всички елементи от масива (т.е. двата указателя съвпадат). Такъв момент винаги настъпва, защото в масива може да има само краен брой елементи. Причината за това е факта, че атомите в масива са променливите, които участват в плана, и то подредени в подходящия ред. Ето защо всяка променлива може да се съдържа само веднъж в масива.

Началото на процеса е поставянето на променената променлива като първи елемент на масива. Нейното обработване може да не създаде нови елементи на масива, но може и да създаде. При първият случай се оказва, че промяната на променливата не изисква промени в други променливи и с това процесът на генериране на план завършва. Във другия случай се преминава към втория елемент на масива (той е първият от новосъздадените) и се повтаря същата обработка.

Фигура III-30 Генериране на план



На горната фигура е показано едно примерно текущо състояние по време на генерирането на план. Променливата, за която се генерира планът е X. Това, което е известно до момента е, че в резултат на промяната на X, ще трябва да се променят Y, Z, T, U и V. Променливата Y е вече обработена. Това означава, че нейната промяна няма да изисква промяната на други променливи освен тези, след нея. Обработването на Z, T, U и V може да изисква създаването на нови елементи към масива.

Основните действия при обработването на елемент са следните:

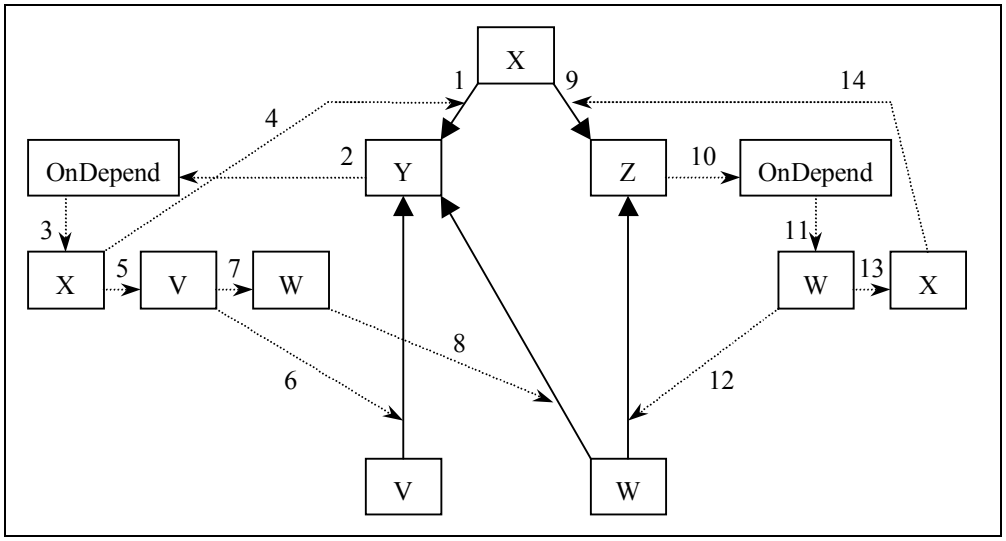
1. Намира се променливата с име, което е стойност на текущо обработвания елемент
2. Намира се OnChange полето на тази променлива
3. Ако има такова поле се намира първото му подчинено поле (т.е. намира се първото именувано правило). Ако няма, обработването на елемента от масива е завършено.
4. Ако то не е елиминирано, името на правилото се добавя към масива и се дава заявка за елиминиране му в рамките на променливата, с която се работи
5. Преминава се към следващото именувано правило и се повтаря точка 4, а ако няма следващо, обработването на елемента от масива е завършено.

Интерес в представения алгоритъм представляват двете действия, описани като "елиминиране на променлива в рамките на друга променлива" и "добавяне към масива".

Елиминирането на променлива е тривиален процес, при който се премахват потенциално опасни връзки по такъв начин, че за времето на генериране на плана те вече няма да се третират като съществуващи. По същество, ако имаме променливата X, към която има именувано правило Y (следователно пълното име на правилото е `X.onchange.Y`), елиминирането на X в рамките на Y означава това правило да се елиминира и да не може да се използва втори път. Подобна елиминация е естествена, но се оказва, че не е достатъчна.

Необходимо е и още едно еднократно елиминиране на връзките, но само за първия елемент от масива – това е променливата, за която се съставя планът. При нея елиминацията е по-сложна и именно тука се използва вторият списък, този, идеята за който възникна от реализацията на връзките в система PGS.

Фигура III-31 Елиминиране на правила за главната променлива



Нека приемем, че главната променена променлива е X и нейната промяна изисква промяна на Y и Z. За да състави верен план, е необходимо да се елиминират всички връзки, които сочат към Y и Z. Това се осъществява по следния начин: Започва се от една от променливите, например Y (1). Намира се от кои променливи зависи (2). В случая се използва наготово полето `OnDepend`, в което са описани всички променливи, които влияят на Y. От там се взима името на първата променлива (3), която влияе на Y. Това се оказва променливата X (4) и затова се елиминира връзката от X до Y. Преминава се по (5) до следващата променлива, която влияе на Y. Това е V и чрез (6) се премахва връзката от V към Y. По аналогичен начин през (7) и (8) се премахва връзката от W към Y. По същия метод се обработва и следващата променлива, на която влияе X. Чрез (9), (10) и (11) се намира, че W също влияе на Z и чрез (12) тази връзка се елиминира. Връзката от X към Z се елиминира чрез (13) и (14).

Независимо, че представеният по-горе алгоритъм изглежда сложен, в действителност той е показан опростено. Например, няма пряка връзка между X и Y. В реалност X има указател към полето си OnChange, а то има указател към X. Връзката между X и Z е още по-дълга: от X се преминава към OnChange, от там към Y и едва от там към Z. Това допълнение от своя страна е също известно опростяване, защото от X не винаги може директно да се намери OnChange. Това става само ако OnChange е първата подчинена променлива на X. В противен случай трябва да се обходят всички подчинени променливи на X, които са преди OnChange. По аналогичен начин връзките от V към Y и от W към Y и Z са също опростени и не са директни.

Ако се анализира внимателно схемата, ще се разбере, че единствената причина за използването на OnDepend е скоростта. Ако такова поле не съществува и не се поддържа, механизмът пак може да бъде реализиран, но намирането на това, кои променливи влияят на конкретно избрана променлива ще изисква обхождането на цялото дърво на променливите. Чрез използването на OnDepend това търсене се свежда до обхождане на списък от атоми.

III.4.3.5 Разширяване на плана

Последната дейност по генерирането на плана, която не е описана, е разширяването на плана. Вече бе споменато, че по време на генерация планът се съхранява в масив. При обработване на елементите на масива се налага да се добавят нови елементи.

Добавянето на елемент се допуска само тогава, когато не се образува цикъл. Прави впечатление фактът, че "борбата" срещу циклите се извършва на няколко фронта. Ако премахването на цикли се осъществява на едно единствено място, това ще изисква

сложен (или поне много бавен) алгоритъм, понеже трябва да се премахнат не само преките цикли, но и косвените. При добавянето на елемент се проверява дали той не би създал пряк цикъл с останалите елементи в масива. Досега не беше споменато, че от данните за елемента в масива може да се определи към кое правило се отнася той. Всъщност, в елемента не се пази име на променлива, а самата променлива, която съответства на именувано правило. По този начин родителят й ще има име OnChange, а прародителят ще е променливата, за която е дефинирано това именувано правило. Ако в масива вече се съдържа елемент, съответстващ на `A.onchange.B`, добавянето на `B.onchange.A` ще бъде игнорирано.

Ако не се образува пряк цикъл, елементът се добавя към края на масива, но това все още не означава, че съдбата му е еднозначно определена. Налага се да се направи и още една проверка. Ако планът се прави за променливата X и вече има създаден елемент отговарящ на `X.onchange.Y` и новият елемент съответства на `Z.onchange.Y`, правилото `X.onchange.Y` се маркира като неизползваемо. Причината за това е, че една и съща променлива се променя по два различни начина, чрез различни други променливи и системата запазва само последния начин на променяне, с което се гарантира, че променливата ще се промени само веднъж.

Елементите, които са определени като неизползваеми не се премахват от масива. Те със пълна сила се използват при следващите негови разширения, особено при проверката за преки цикли. Използваемостта на елемент се ползва единствено при окончателното композиране на плана.

III.4.3.6 Композиране на крайния план

Окончателното съставяне на плана изисква да се създаде списък от команди, които да се изпълняват при промяната на променливата, за която се генерира план. Това се осъществява като се създава списък от пълните имена на всички атоми, намиращи се в масива, за които е указано, че се използват. Естествено, в този списък не фигурира първият елемент от масива. Така създаденият план, може да изглежда, примерно, така:

```
[X.onchange.Y Y.onchange.K X.onchange.Z Z.onchange.F]
```

което може да се разтълкува по следния начин: след промяната на X трябва да се изпълнят командите, записани в `X.onchange.Y`, така се променя Y и затова трябва да се изпълни `Y.onchange.K`. Едва сега може да се промени и Z - другата зависима от X променлива, чрез изпълнение на `X.onchange.Z`, което от своя страна ще изисква изпълнение и на `Z.onchange.F`.

Планът винаги се съставя така, че по време на неговото прилагане да няма нужда да се допълва и променя. Т.е. изпълнените команди са всичките, които трябва да се изпълнят при промяната на X.

III.5 Изобразяване на обектите

Една от най-видимите дейности на система Elica е създаването на графични изображения и анимации. Поради високите изисквания на съвременните потребители, на проектирането и на реализирането на методите за управление и генериране на изображения бе отделено значително време.

Като цяло, в развитието си системата е преминала през три фази. Всяка от тях, отразява модерното за времето си мислене и методи за визуализиране.

Най-първите версии на Elica, тези, които съществуваха по времето на Mono, CGA, EGA и VGA мониторите, имаха вградени драйвери за директен достъп до видеопаметта. Това гарантираше висока скорост, още повече, че графичните процедури бяха написани на асемблер. Но това доведе и до няколко неудобства:

- за всеки вид видеоплатка трябваше нов набор от функции, понеже на ниско ниво видеопаметта се различава, а портовете за управлението й също бяха различни
- модифицирането и поддържането на код на асемблер изисква съществени човешки и мисловни ресурси

Не бива да се забравя, че тези първични системи бяха реализирани преди навлизането на Windows.

След като Elica премина на Windows (тогава тя се е казвала LGS, а версията за Windows – LSGW) директният достъп до видеопаметта става крайно нежелателен, още повече, че новата операционна система предоставя богат и достатъчен набор от примитивни графични функции. Освен това, грижата да се поддържат различните видеорежими и цветови модели отпадна изцяло и се прехвърли на Windows.

Тази втора фаза просъществува в две подфази. При първата, рисуването на графичните изображения ставаше направо върху съответния прозорец, което създаваше визуални и естетически проблеми и движението на обектите по екрана [20]. След усъвършенстване на графичния апарат, започна втората подфаза. При нея рисуването на обектите става в специални метаобекти (наричани в Windows метафайлове). Това доведе до съществото преимущество генерираният образ на даден обект да се съхранява в капсулован вид.

Третата фаза, засега последна, избягва употребата на всички графични средства на Windows, а използва тези на OpenGL. Предимствата на тази графична библиотека и начина й на използване са описани в някоя от следващите глави.

III.5.1 Изрязване на изображения (clipping)

III.5.1.1 Въведение

Реализирането на графичните функции в предишните системи, не представлява съществен интерес, но е изисквало разработването на собствени методи. Това се е наложило поради факта, че изобразяването на криви от втора степен трябва да се реализира от системата – Windows не предоставяше подобна функционалност.

Един от основните проблеми при визуализирането на криви е този, че рисуването им не е толкова просто и бързо, като рисуването на линия. Всеки фрагмент на линията е идентичен на всеки друг фрагмент от същата линия, но за криви същото не може да се твърди. Затова, кривите се декомпозират на множество от свързани отсечки, всяка от които може да се нарисува по сравнително прост начин.

Изчисляването на кривата може да заеме доста ресурси (най-вече изчислително време, а понякога и памет). Затова с цел ускоряване, всички графични системи се опитват да

избегнат излишните изчисления. Система Elica не прави изключение. В нея също е вграден алгоритъм за изсичане на ненужните части от кривите (тези, които са извън рамките на видимата област). Процесът по изрязването се нарича clipping.

Най-грубо, съществуват два често използвани алгоритъма за изрязване на криви [11]. При единият се рисуват всички сегменти (т.е. отсечки, съставящи приближение на кривата), а операционната система прилага стандартното за нея изрязване. При този вариант, управление на изрязване от страна на системата не се прави, но пък се изчисляват междинните точки на кривата независимо от това, каква част от нея видима. При другият алгоритъм се изчисляват пресечните точки на кривата с контурите на екрана и само видимите фрагменти се декомпозират. При този алгоритъм времето за рисуване на крива е голямо, поради това, че не винаги изчисленията са елементарни и бързи.

За да се реализира този алгоритъм, кривата трябва да се представи в параметричен вид. Ако точка P(x,y) е точка от кривата, трябва да съществува t, което да удовлетворява системата:

$$\left| \begin{array}{l} X = X(t) \\ Y = Y(t) \end{array} \right. \qquad \textit{(III.5-1)}$$

Изрязването на кривата е равносилно, на решаване на системата от неравенства:

$$\left| \begin{array}{l} MinX \le X(t) \le MaxX \\ MinY \le Y(t) \le MaxY \end{array} \right. , \qquad \textit{(III.5-2)}$$

където MinX, MaxX, MinY, MaxY са константи, определящи рзмера на прозореца. Проблемите при решаването на (III.5-2) са в това, че за някои криви те включват изрази като $a + bt + ct^2$ или $\sqrt{a + bt + ct^2}$, за които няма бързи решения. Пълното решение на системата дава 8 решения, които по двойки образуват четири интервала. Няма очевиден критерий как да се сдвоят решенията, особено в случаите, когато има еднакви решения, или пък нереални решения. Ако системата е вече решена, а интервалите – определени, рисуването на кривата е достатъчно елементарен процес и няма да бъде дискутиран.

III.5.1.2 Нов алгоритъм за изрязване

Както вече бе обяснено и двата начина на изрязване не са приложими за конкретните цели на Elica. Наложи се да се измисли нов начин, който да обединява положителните характеристики и на двата метода. Новият метод е двустъпков и комбинира предишните два метода.

Първата стъпка е да се решат неравенствата:

$$\left| \begin{array}{l} MinX \le \overline{X}(t) \\ MaxX \ge \underline{X}(t) \\ MinY \le \overline{Y}(t) \\ MaxY \ge \underline{Y}(t) \end{array} \right. \qquad \textit{(III.5-3)}$$

където

(III.5-4) $\underline{X}(t) \le X(t) \le \overline{X}(t)$, и

(III.5-5) $\underline{Y}(t) \le Y(t) \le \overline{Y}(t)$

И четирите функциите $\underline{X}(t)$, $\overline{X}(t)$, $\underline{Y}(t)$ и $\overline{Y}(t)$ са линейни функции и по този начин позволяват (III.5-3) да се реши лесно и бързо. Много е важно да се намерят колкото се може по-рестриктивни функции, удовлетворяващи (III.5-4) и (III.5-1). Колкото функциите са по-рестриктивни (т.е. колкото по-силно ограничават отгоре и отдолу параметричното уравнение на кривата), толкова по-точно изрязване ще се получи.

Важното в случая е, че не се изисква изрязването да е точно – може част от изрязаната крива да излиза извън границите на екрана. По-важното е, че изрязването не премахва видимата част от кривата, а излишната част (тази, която не се изрязва, но и не се вижда) е ограничена и достатъчно малка.

III.5.1.3 Линейно ограничаване

Проектирането на подходяща рестриктивна функция е особено важна дейност. За повечето криви е удобно да се раздели кривата на няколко части и за всяка поотделно да се приложат двете стъпки. Най-често използваните интервали са $[-\infty,-1]$, $[-1,0]$, $[0,1]$ и $[1,\infty]$. Тези интервали са избрани така, че да са подходящи за използваните неравенства. В случаите, когато функциите се базирани на тригонометричните функции, интервалите са $[0,\frac{1}{2}\pi]$ и $[\frac{1}{2}\pi,\pi]$.

Тези интервали са напълно достатъчни за разделянето на кривите от втора степен – окръжност, елипса, парабола и хипербола. За други криви е много вероятно да са по-подходящи други интервали.

III.5.1.4 Основни характеристики на кривите

За да се изясни въпросът относно кои са важните (относно Elica) характеристика на кривите, е важно да се определи кои уравнение ще се използват. За всяка от кривите съществуват по няколко различни уравнение, но не всяко от тях е подходящо.

Следват частично нормализираните уравнения на кривите [23]:

(III.5-6) $(x-x_0)^2+(y-y_0)^2=r^2$ за окръжности

(III.5-7) $\frac{(x-x_0)^2}{a^2}+\frac{(y-y_0)^2}{b^2}=1$ за елипси

(III.5-8) $y-y_0=p(x-x_0)^2$ за параболи

(III.5-9) $\frac{(x-x_0)^2}{a^2}-\frac{(y-y_0)^2}{b^2}=1$ за хиперболи

Уравненията (III.5-6), (III.5-7), (III.5-8) и (III.5-9) не са параметричните уравнения, използвани по време на рисуване, но чрез тях се обясняват лесно характеристиките на обектите.

Всички криви от втора степен (в смисъла на Elica) могат да се характеризират с център, ъгъл на завъртане, радиус и точност. За всяка крива съществува централна точка,

наричана за кратко *център*. Тази точка определя отместването на кривата спрямо началото на координатната система. За парабола, центъра е във върха й, за останалите криви, той съвпада с геометричния център. Центърът на крива се отбелязва с $P(x_0, y_0)$, където x_0 и y_0 са тези от уравнения (III.5-6), (III.5-7), (III.5-8) и (III.5-9).

Ъгъл на завъртане е параметър който показва завъртяността на кривата спрямо нейното "естествено" положение. Само за окръжностите този параметър не е от значение. Уравненията (III.5-6), (III.5-7), (III.5-8) и (III.5-9) са зададени при ъгъл на завъртане $\varphi = 0$.

Радиусът е друга основна характеристика. За окръжностите той съвпада с техния радиус и в (III.5-6) е отбелязан с r. За елипсите (III.5-7) и хиперболите (III.5-9) радиусът е двойка числа: a и b. За параболите (III.5-8) радиусът съответства на параметъра p.

Точността не е геометрична характеристика. Тя определя колко прецизно да се рисува кривата. В системата, всяка от кривите се разделя на четири *подкриви* и точността определя на колко линейни фрагмента ще се раздели всяка подкрива. Точността се използва по следния начин: всяка подкрива се изрязва с (III.5-3) независимо от останалите подкриви и в последствие се разделя на съответния брой точки. Ако точността е N, броят на изчислените точки е 4xN (по N за всяка подкрива). Всяка подкрива се рисува самостоятелно, като набор от отсечки, свързващи последователно N-те точки от нея. По този начин точността се прилага не върху цялата крива, а само върху тази част, която остава след изрязването.

III.5.1.5 Трансформиране до линейни неравенства

За да се илюстрира методът на преобразуване на нелинейните неравенства в линейни, ще разгледаме в детайли два примера.

Нека първият пример е с неравенството:

(III.5-3a) $\quad At + B\frac{1}{t} \geq C$

Изрязването на хиперболи става с решаването на няколко неравенства, като (III.5-3a), където A, B и C са реални числа (положителни, отрицателни или нула – ако знакът на неравенството е $\leq$ вместо $\geq$, на коефициентите трябва да се смени знакът). Трансформирането в линейно неравенство зависи от допустимия интервал на t. Нека разгледаме случая $t \in [1, \infty)$. Ако $B = 0$, (III.5-3a) е еквивалентно на:

(III.5-4a) $\quad At \geq C$

Ако $B > 0$, можем да заместим $B\frac{1}{t}$ с B защото $B \geq B\frac{1}{t}$. По този начин (III.5-3a) се трансформира в неравенство (III.5-5a), което е по-свободно, но за сметка на това е линейно.

(III.5-5a) $\quad At + B \geq C$

Ако $B < 0$, можем да заменим $B\frac{1}{t}$ с 0 защото $0 \geq B\frac{1}{t}$. Ситуацията е същата както в случая B=0, и така, неравенство (III.5-3a) се трансформира в (III.5-4a). По аналогичен

начин се процедира за интервала $t \in (0,1)$, като разликата е, че се трансформира At вместо $B\frac{1}{t}$.

Вторият пример е за трансформиране на тригонометрични неравенства, каквито възникват при изрязването на елипсите. Нека разгледаме следния общ случай:

(III.5-6a) $A\sin\theta + B\cos\theta + C \geq 0$

За интервала $\theta \in [0, \frac{1}{2}\pi]$ функциите sin и cos се ограничават със следните неравенства:

(III.5-7a) $\underline{\sin}\,\theta = \frac{2}{\pi}\theta \leq \sin\theta \leq \theta = \overline{\sin}\,\theta$

(III.5-8a) $\underline{\cos}\,\theta = 1 - \frac{2}{\pi}\theta \leq \cos\theta \leq \theta - \frac{\pi}{2} = \overline{\cos}\,\theta$

За другия интервал $\theta \in [\frac{1}{2}\pi, \pi]$ неравенствата (III.5-7a) и (III.5-8a) ще са по-различни. Нека се върнем на (III.5-6a). Според знаците на A и B може да се избере кое линейно ограничение на sin и cos да се избере. Например, ако $A < 0$ и $B > 0$, функцията sin трябва да се замени със $\underline{\sin}$, а функцията cos - с $\overline{\cos}$. В този случай, (III.5-6a) се трансформира в (III.5-9a):

(III.5-9a) $$\frac{2A}{\pi}\theta + B\left(\theta - \frac{\pi}{2}\right) + C \geq 0 \quad \Leftrightarrow \left(\frac{2A}{\pi} + B\right)\theta + C - \frac{B\pi}{2} \geq 0$$

III.5.1.6 Производителност

Следвайки аналогични разсъждения, старите версии на системата съдържаха набор от функции за бързо и ефективно рисуване на криви от втора степен. За да се прецени скоростта на работа, бе реализирана специална програма. В таблицата по-долу е показан резултат от тестването на производителността.

Таблица III-6 Рисуване на криви от втора степен

	Окръжности	Елипси	Параболи	Хиперболи
	$\sigma(x_0, y_0, R)$	$\varepsilon(x_0, y_0, \varphi, a, b)$	$\pi(x_0, y_0, \varphi, p)$	$\chi(x_0, y_0, \varphi, a, b)$
Тестови условия $T \in [1, 10^4]$	$x_0 = y_0 = 0$ $R = 10 + \frac{T}{20}$	$x_0 = y_0 = 0$ $\varphi = T$ $a = 10 + \frac{T}{20}$ $b = 10 + \frac{T}{50}$	$x_0 = y_0 = 0$ $\varphi = T$ $p = 10 + \frac{T}{20}$	$x_0 = y_0 = 0$ $\varphi = T$ $a = 10 + \frac{T}{20}$ $b = 10 + \frac{T}{50}$
Общо време	55.8 сек	85.6 сек	42.6 сек	48.0 сек
Секунди за една крива	0.006 сек	0.009 сек	0.004 сек	0.005 сек
Брой криви за секунда	179 окръжности	117 елипси	235 параболи	208 хиперболи

Тестовата система включва IBM PC Compatible, 80486 DX2/66 MHz, 8 MB RAM, Windows 3.10, Borland Pascal for Windows 7.0.

Всички тестове са извършени при една и съща точност от 40 точки за крива (т.е. $N \approx 10$). Параболите и хиперболите се рисуват по-бързо, понеже процесът на тяхното рисуване е оптимизиран относно математическите операции. Аналогична оптимизация, за окръжностите и елипсите също може да се постигне скорост от поне 200 криви в секунда.

III.5.1.7 Примерна програма

В последните версии на системата демонстрираният алгоритъм за изрязване не се използва, затова е добре да се покаже кратък, но завършен пример на програма, рисуваща хипербола. Програмата може да се компилира с Borland Pascal 7.0, но с минимални промени може да се адаптира и на други езици за програмиране.

```
program Hyperbola;
uses OWindows,WinTypes,WinProcs;
const MaxX = 200;
      MinX = -350;
      MaxY = 200;
      MinY = -100;
      N    = 9;

var MyApp : TApplication;
    DC    : HDC;

procedure Hyp(DC:HDC; A,B,X0,Y0,F:real);
  {-Draws hyperbola}
  var T1,T2,TS : real;
      S,C,AS,AC,BS,BC : real;
      K1,K2,K3,K4,KB1,KB3,KS1,KS3 : real;
      I   : integer;
      H,V : longint;

  function Solve(K1,K2,M:real):boolean;
    {-Solves the equation K1*t+K2/t>=M}
    begin
      Solve:=True;
      if K1>0 then M:=M-K1;
      if K2=0
        then begin if M>0 then exit; end
        else if K2<0
               then if M>=0
                      then exit
                      else begin
                             M:=K2/M;
                             if T1<M then T1:=M;
                           end
               else if M>0
                      then begin
                             M:=K2/M;
                             if T2>M then T2:=M;
                           end;
      Solve:=False;
    end; {-Solve}
  procedure Draw;
    {-Draws a one forth of a hyperbola}
    var I:integer;
    begin
      K1:=AC+BS;
      K2:=AC-BS;
      K3:=-AS+BC;
      K4:=-AS-BC;
```

```
      T1:=0;
      T2:=1;
      if Solve(K1,K2,MinX-X0) then exit;
      if Solve(-K1,-K2,-MaxX+X0) then exit;
      if Solve(K3,K4,MinY-Y0) then exit;
      if Solve(-K3,-K4,-MaxY+Y0) then exit;
      if T1>T2 then exit;

      TS:=(T2-T1)/Pred(N);
      if T1=0 then T1:=T1+TS;
      KB1:=K1*T1; KS1:=K1*TS;
      KB3:=K3*T1; KS3:=K3*TS;
      for I:=1 to Succ(N) do
        begin
          H:=Round(400+(KB1+K2/T1+X0));
          V:=Round(300-(KB3+K4/T1+Y0));
          if I=1
            then MoveTo(DC,H,V)
            else LineTo(DC,H,V);
          T1:=T1+TS;
          KB1:=KB1+KS1;
          KB3:=KB3+KS3;
        end; {-for}
    end; {-Draw}

  begin {-Hyp}
    F:=F*Pi/180;
    S:=Sin(F);
    C:=Cos(F);
    A:=A/2;
    B:=B/2;

    AC:=A*C; AS:=A*S; BC:=B*C; BS:=B*S;
    Draw;
    S:=-S; C:=-C; AS:=-AS; AC:=-AC; BS:=-BS; BC:=-BC;
    Draw;
    B:=-B; BS:=-BS; BC:=-BC;
    Draw;
    S:=-S; C:=-C; AS:=-AS; AC:=-AC; BS:=-BS; BC:=-BC;
    Draw;
  end; {-Hyp}

begin {-Main program}
  MyApp.Init('');
  MyApp.MainWindow^.Show(sw_ShowMaximized);

  DC:=GetDC(MyApp.MainWindow^.HWindow);
  Rectangle(DC,MinX+400,300-MinY,MaxX+400,300-MaxY);
  Hyp(DC,120,80,10,-10,30);
  ReleaseDC(MyApp.MainWindow^.HWIndow,DC);

  MyApp.Done;
end. {-Main program}
```

III.5.2 Механизъм на изобразяване

Механизмът на изобразяване в система Elica не се отнася до конструирането на образа на единичен обект, а до цялостната концепция съдържаща критериите за това кога да се преизчисли образът на обект и кога да се прерисува на екрана.

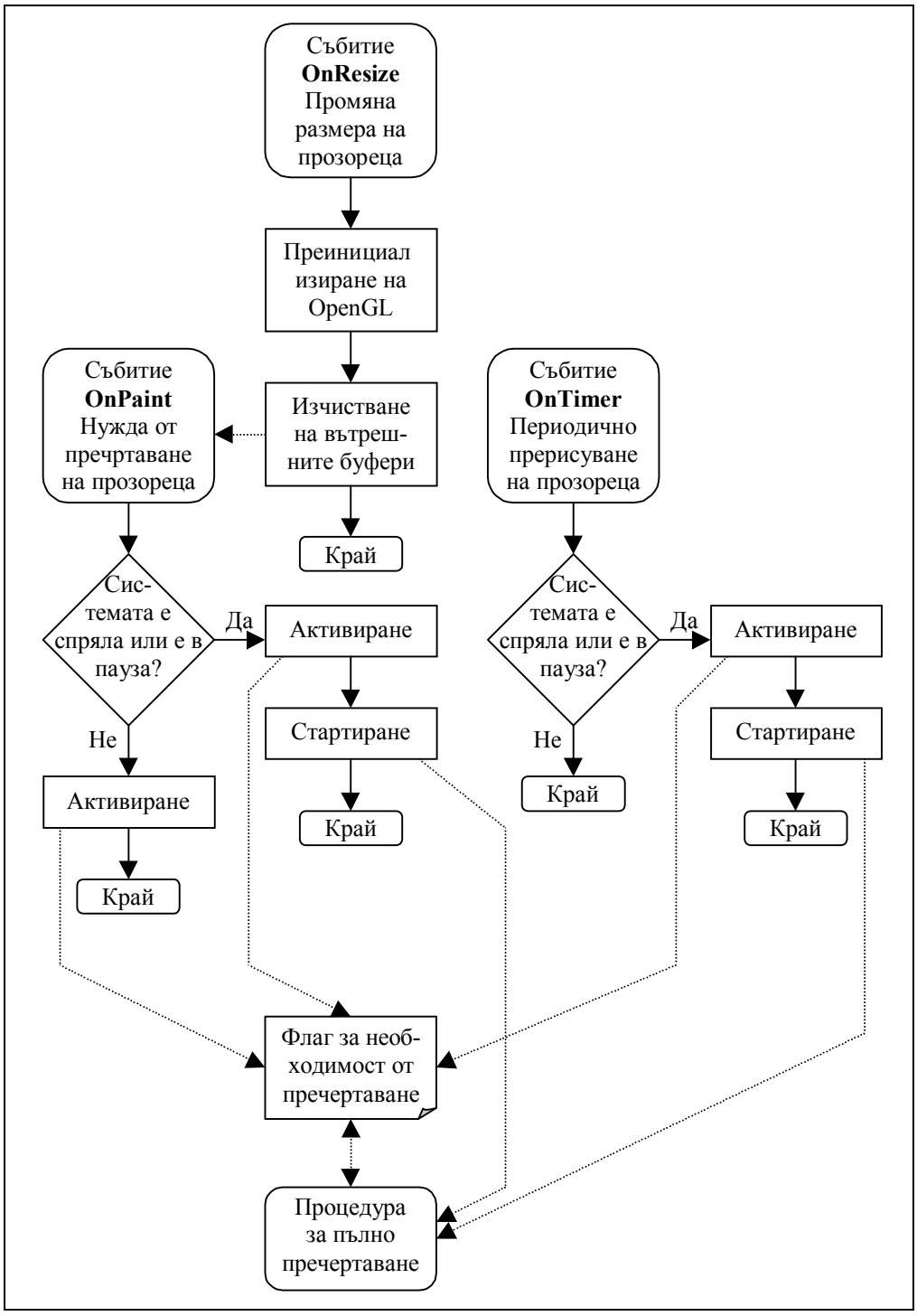
Очевидно е, че прерисуване трябва да се извърши най-малко по две различни причини:

- *Външна* – когато се промени размера на прозореца или негови скрити части станат видими (т.е. незасенчени от други прозорци), някои или всичките обектите с изображения, ще трябва да се пречертаят
- *Вътрешна* – когато се променя стойността на графичен обект.

III.5.2.1 Външни критерии за прерисуване

Външната причина не представлява съществен интерес, поради това, че е ясно кога точно възниква и как трябва да реагира системата. Системата разбира, кога е възникнала външната причина от системните съобщения, които получава от Windows.

Фигура III-32 Пречертаване след външни събития



В единия случай, ако съобщението е за променен размер на прозореца (независимо дали е станал по-малък или по-голям или просто по-различен), не е достатъчно само да се прерисуват обектите, а също и да се преинициализират някои характеристики на графичната система OpenGL:

- изчиства се фона
- заменят се активния с пасивния буфер
- зарежда подходящи стойности в проекционната матрица според размера на прозореца
- определя видимата част от прозореца и настройва логическата координатна система
- зарежда единичната матрица в матрицата на гледната точка

Важна особеност е, че това събитие се последва автоматично от друго – събитието, което обозначава, че част от прозореца трябва да се прерисува. Това събитие може да възникне и самостоятелно, затова на него трябва да се реагира адекватно и независимо от предходното събитие.

Изискването да се пречертае прозореца не означава, че системата трябва да направи това веднага. Според текущото състояние, това може да се отложи за по-подходящ момент. Критерий за това е текущото състояние на системата – ако в момента тя не изпълнява програма или е спряла (например в паузата при трасиране), пречертаването може да се осъществи веднага. В противен случай, системата работи и не трябва да се прекъсва работата й докато не настъпят подходящи за това условия.

Съществува и още едно събитие, което изисква пречертаване на екрана. В системата има таймер, който периодично изисква от системата да обнови екрана. Това събитие се игнорира, ако системата работи, а ако не работи на нея се реагира по същия начин както и на предишното съобщение.

III.5.2.2 Вътрешни критерии и пълно прерисуване

Физическото прерисуване на прозореца се осъществява от една единствена процедура, която не може да пречертава отделни обекти, а всички заедно. Това е следствие от концепцията на използване на OpenGL.

Както е показано на Фигура III-33 процедурата изчиства флага за необходимост от пречертаване.

Освен при някои външни събития, пречертаването се активира и при някои вътрешни:

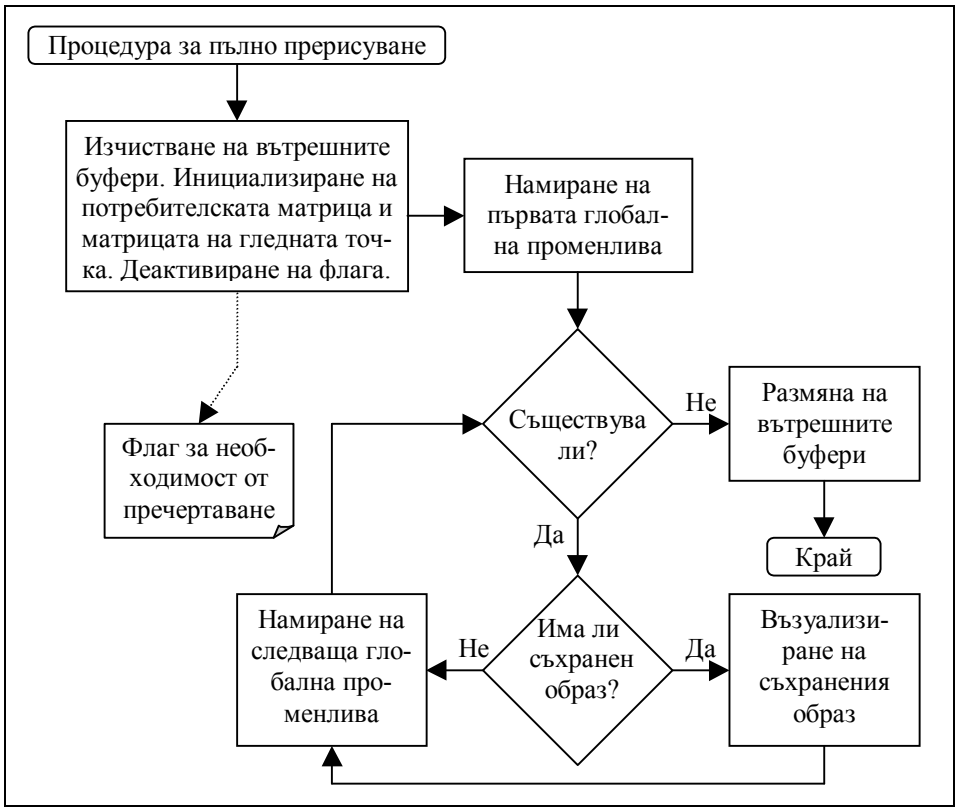
- След изпълнението на команда на най-високо (глобално ниво) и междувременно има променени глобални променливи с графично изображение
- Преди всяко изпълнение на списък от команди, ако флагът за необходимост е активиран
- След приключване на работа и след изчистване на всички променливи
- След приключване на работа в случаите, когато променливите се запазват
- При настъпване на пауза при трасиране на програмата
- При заявка от външна библиотека

Изборът на това при кои вътрешни събития да се прави пречертаване на екрана е важен, защото от една страна на потребителя не трябва да се показват "остарели", непълни или неточни изображения, нито пък актуализирането на прозореца да става ненужно често и да забавя значително изпълнението на програмата. Избраните събития гарантират именно това и са получени в резултат на продължителни тестове.

Компромисът, който е направен, е да не се реагира с пречертаване при промяна на локална променлива, дори и тя да има изображение. При опитите, когато се пречертават всички променливи, се забелязва съществен недостатък. Скоростта не само че е влошена, но и допълнителните изображения объркват потребителя. Затова е взето

решението да се показват само глобалните променливи, а всички локални (т.е. временни) да са без графичен образ.

Фигура III-33 Пълно прерисуване



Разбира се, това решение се отнася само до визуализирането на обектите, а не до вътрешното генериране на образите. Именно това е част от разликите между система Elica и други подобни графични системи.

III.5.2.3 Графична структура и йерархия на изображенията

При Elica, за всеки графичен обект се създава образ (набор от OpenGL команди), но той не винаги се визуализира на екрана. Това поражда резонния въпрос, защо е необходимо да се създава образ, който няма да се визуализира.

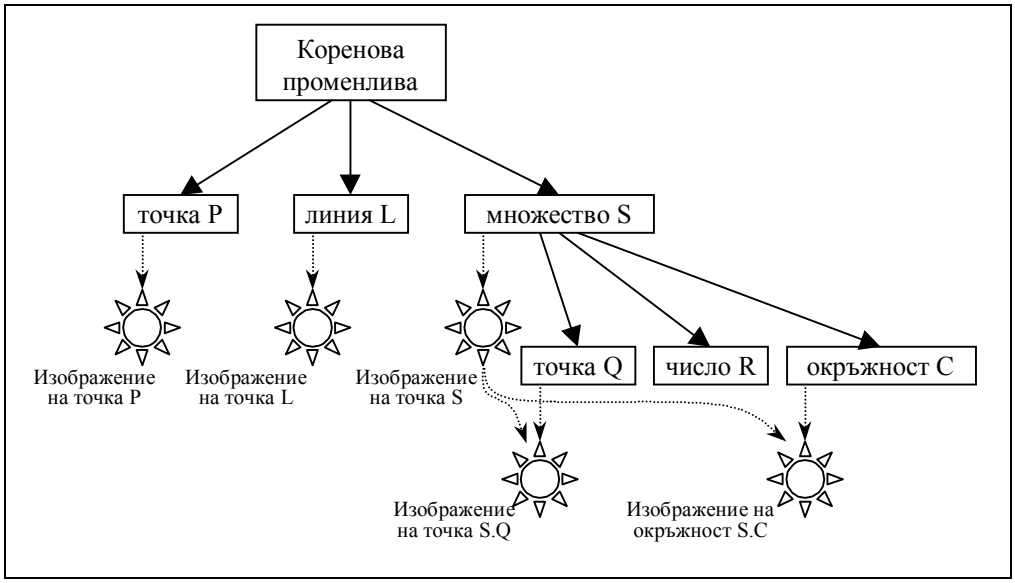
Отговорът е, че системата не може да разбере, кога образът на обект ще й потрябва и кога не. Много често сложни обекти се създават като множество от по-прости. Например, геометричното място от точки може да се визуализира чрез образите на отделните точки, чрез които потребителят да определи вида и формата на геометричното място. В подобни случаи, няма нужда да се създава отделен образ на цялото множество, а просто да се обозначи от кои самостоятелни подобрази се изгражда. По този начин при промяната на един елемент на конструкция, няма нужда да се прегенерират и всички останали елементи.

При създаването или променянето на променлива, която има изображение, системата изпълнява съответните методи, за да създаде списък от OpenGL команди, които всъщност са вътрешния образ на променливата. Няма пряка връзка между обект и образа му, понеже списъците от команди в OpenGL се адресират по име, а не по адрес. За щастие, името е 32 битово число и за настоящата реализация на система Elica, адресът на обекта може да се използва и като име на образа му.

На Фигура III-34 са показани три глобални променливи – P, L и S. Първите две си имат собствени изображения, докато третата е съставна. Тя се състои от три подполета, две

от които са с образи, а едно – без. Ако за променливата S е дефиниран методът `ondrawimage`, за нея също ще се създаде образ, но физически, той няма да съдържа образите на S.Q и S.C, а само препратка (по име) към тях.

Фигура III-34 Йерархия на изображенията



Целта на точно такова представяне е да има възможност при промяна на една променлива, например C, да не се налага прегенериране на изображението на Q. Ползата от въвеждането на тази функционалност следва от факта, че генерирането на вътрешните изображения става с изпълнение на методи написани на Elica Logo и често използват външни библиотеки, което е много по-бавно от самото рисуване на изображенията върху прозореца.

Ефективността си проличава най-вече при създаването на обекти. По време на създаване на полетата на обекта, за всеки от тях, за когото има дефиниран метод за изобразяване, се създава вътрешно изображение. В този момент системата "не знае", дали се създава обект или това са просто локални променливи, и затова не рисува тези изображения на екрана. След завършване създаването на целия обект, ако за него не е дефиниран специален метод за рисуване, но се изисква да има образ, системата генерира малко по обем вътрешно изображение, което на практика съдържа единствено OpenGL имената на вътрешните изображения на полетата на обекта.

Ето защо може да се говори за йерархия на изображенията. Тя може да е на произволен брой нива и на всяко ниво да има произволен брой вътрешни изображения.

Връзката "адрес на променлива" – "име на вътрешно изображение" е съществена и необходима, най-малкото поради три причини:

- подпрограмата за рисуване на обектите върху екрана, има достъп само до променливите, но чрез адреса им може да провери и евентуално използва техните вътрешни изображения.
- при премахването на променлива трябва да се премахне и нейният вътрешен образ (ако има такъв). Без подобна връзка, това не може да бъде осъществено. Временно на съществуване на вътрешното изображение не трябва да излиза извън интервала на съществуване на обекта, чието изображение е.
- когато модулът OpenGL е в режим на избор, може да се проследи кои обекти са нарисувани в определена зона от прозореца. OpenGL връща списък с имената им, но поради връзката, от имената еднозначно се получават адресите на променливите. Подобен механизъм може да се използва при реализацията на drag-and-drop.

Както бе споменато в описанието на езика за програмиране Elica Logo, потребителят може да избере по какъв начин ще се визуализират сложните обекти. Единият начин е да се зададе собствен метод за изобразяване. В такъв случай изображенията на подобектите не се използват. Вторият начин, засега по-често срещан, е методът да се остави празен, но съществуващ. Това служи за индикатор за системата да състави изображението на обекта като обединение от изображенията на отделните му подобекти. Този метод е частен случай и може да бъде реализиран и по първия начин, но пък предоставя удобство на програмиста и спестява съществени усилия при създаването на по-сложни обекти.

III.6 Технологии

Реализирането на система Elica не може и не трябва да се прави без да се използват тези от съвременните технологии, които биха били от съществена полза. Една от технологиите е работата с числа с плаваща точка и по-точно: ефективното използване на стандартният вече числов процесор. Благодарение на това Elica може безпрепятствено да работи с безкрайности и неопределени числа.

В първоначалните разработки на системата, визуализирането на графичните обекти е било правено със собствени алгоритми за рисуване, базирани на стандартните графични възможности на Windows. Това създава редица трудно преодолими проблеми при създаването на динамични фотореалистични симулации.

В настоящият момент съществуват няколко графични технологии, най-разпространението от които са OpenGL (Open Graphics Language) и DirectX. Избрана е първата технология, понеже тя е платформено независимо, докато втората се поддържа единствено за платформата Windows. Консорциумът, управляващ стандарта OpenGL, предвижда в най-скоро време да го разшири. Новият стандарт ще се казва OpenML (Open Multimedia Language) и ще покрива всички аспекти в създаването на мултимедийни системи.

III.6.1 Работа с числа с плаваща точка

III.6.1.1 Терминологични бележки

Терминът *плаваща точка* ще бъде предпочетен, в настоящия текст, поради факта, че в софтуерната индустрия той е по-разпространен, от термина *плаваща запетая*. Независимо от факта, че някои среди предоставят локализация на начина на изписване на нецелите числа, формата на записване с плаваща точка е де факто общоприет стандарт. Като синоним на число с плаваща точка, ще бъде използван и термина *реално число*, независимо, че то не покрива понятието *реално* в математическия му смисъл.

III.6.1.2 Въведение

По време на създаване на версии на продуктите, предшественици на Elica, работата с числа с плаваща точка, се е извършвала по един от трите най-разпространени тогава начини. Независимо от плюсовете и минусите на всеки от тях, към настоящият момент само един от тях е приел широко разпространение и е използван в Elica.

Главният проблем, поради който се налага да се отдели внимание на работата с числа с плаваща точка, е наличието на възможности за получаване на невалидни резултати – прекалено големи или прекалено малки числа, а също и нечислови резултати.

Единият от начините за работа с тези числа е да се изгради софтуерен апарат от високо ниво, който да следи за операции, които могат да доведат до проблеми. Друг начин е да се ползва стандартния числов процесор, който е вграден в по-новите Intel и съвместими с Intel чипове. При по-стари системи, могат да се използват софтуерни емулатори числовия процесор.

Към настоящият момент, е прието за стандарт системите с процесори на Intel да съдържат и копроцесори, задължително поне един – числов, а в някои конфигурации и графичен, или някой още по-специализиран.

Интерес предизвиква проблемът относно възможностите на числовите копроцесори. По-основно запознаване с техните възможности поражда уместният въпрос защо толкова малка част от тях се използва! В много малко продукти, работата с реални

числа използва стандартните вече възможности на числовите копроцесори, а дори и тези, които го използват, използват само една малка, базова част от тях.

Естествено, в някои среди за програмиране са предоставени средства, с които при добро желание и подходяща документация, програмистът може да използва пълните възможности, но все пак, това не е дейност, подходяща за начинаещи. За разлика от тези среди, Elica предоставя възможност за доста по-пълно използване на тези възможности. Основните изисквания, по време на реализацията са:

- системата да е устойчива към грешки породени от "неправилна" употреба на реални числа (като например, умножаване на твърде големи числа, деление на нула, коренуване на отрицателни числа и т.н.)
- системата да е пригодена за изграждане на различни модели на природни и абстрактни явления, където влиянието на физическите ограничения на компютъра върху точността на модела са сведени до минимум. Потребителите, които ползват системата трябва да могат да прикрият всяка "неправилна" употреба на реални числа.

III.6.1.3 Първоначални разработки

В предишните реализации на системата, са изпробвани различни начини за работа с реални числа, всички те, без последният, който е приложен в Elica, са отхвърлени, поради една или друга причина.

III.6.1.3.1 Цели и реални числа

При този разработка, системата прави разграничение между целите и реалните числа. Основната идея е, като се разделят хомогенните от хетерогенните операции, може да се постигне по-голяма икономия на паметта и по-голяма производителност. Хомогенните операции са тези, при които аргументите и резултата са или само цели, или само реални. Икономията на памет се постига главно поради факта, че целите числа заемат по-малко памет за своето представяне, а по-голямата производителност се постига само при хомогенни операции, особено при целочислените.

Независимо от тези два положителни параметъра, недостатъците на този начин на работа с числа са доста по-сериозни. На първо място, не се предоставя начин за ефективно справяне с грешките. На второ, разделянето на операциите на хомогенни и нехомогенни, води до съществено усложняване на алгоритмите:

- повечето операции трябва да се реализират в поне два варианта – целочислен и реален
- при хетерогенните операции трябва да се конвертира типа на аргументите или резултата, за да стане операцията хомогенна
- за някои операции хомогенността зависи от параметрите – делението на целочислени числа може да създаде както целочислен, така и реален резултат.
- допустимият интервал на целите числа е доста по-малък от този на реалните и една целочислена хомогенна операция, като например, умножение на цели числа, може да генерира резултат, който е извън допустимите стойности.

Всички тези усложнения, правят крайната производителност доста по-ниска от очакваната, а поддържането и по-нататъшното развитие на подобна система доста по-трудно.

III.6.1.3.2 Само реални числа

Естествена стъпка за премахването на по-голямата част от недостатъците, описани в III.6.1.3.1 е да се елиминира разграничението между хомогенни и нехомогенни операции. Начинът е да се премахне поддържането на целите числа, като по този начин всички операции ще станат хомогенни, с реални аргументи и реален резултат.

Това, разбира се, не означава, че разликата между цели и реални числа е заличена – относно потребителя, тези два различни типа числа си съществуват, но конвертирането при входа и изхода на някои реални числа до цели се прави от системата автоматично.

Преимуществата на този начин е, че се премахват доста от проблемите при работа с числа, като в крайна сметка производителността е по-добра. Решава се и неудобството с по-малкия диапазон на целите числа. Независимо от това, си остава проблемът, че няма никаква защита от "нелегални" операции.

III.6.1.3.3 Профилактични действия

Един от първите начини да се направи защита от неправомерни операции с числа с плаваща точка, е да се използват така наречените *профилактични* проверки. По своята същност това са проверки, които установяват дали при конкретните аргументи дадената операция ще се извърши добре, или ще предизвика грешка.

Знаейки каква е горната граница за положителните реалните числа и долната за отрицателните, може да се предвиди механизъм, с който да се проверява дали дадена операция ще е нелегална или не. За операциите с един аргумент това е доста лесно, но за тези с два аргумента, проблемите са по-сериозни.

Да вземем най-простия пример – събиране на две числа. Ако те са с различен знак, проблеми няма – резултатът ще е винаги по-малък (по модул) от по-големият (по модул) от двата аргумента. Ако знаците са едни и същи, проблемите са по-сериозни. Най-синтезирано, те се изразяват с факта, че проверката или ще използва много повече време от самото пресмятане на операцията, или пък ще действа с голямо закръгляне (при някои аргументи, водещи до "нормален" резултат, проверките ще сигнализират, че резултатът няма да е "нормален").

При една естествена разработка, повечето от операциите ще са легални, а много малко ще водят до проблеми. Защитата от нелегални аргументи на операциите, ще снижи значително производителността на системата, затова, начинът с профилактичните действия е практически неизгоден.

III.6.1.3.4 Завършващи инструкции

В една от предишните версии на Elica [25], е използвана техника, която независимо от своите предимства, е използвана твърде рядко от програмистите. Много среди дават възможност на програмистите да се подсигурят, че всички ресурси, които те използват, ще бъдат освободени независимо по какъв начин програмата спира своята работа. Реализацията на това става, като потребителят може да укаже коя процедура да се изпълни при завършване на програмата.

Програмата има начини да разбере дали когато завършва, това е станало по естествен път, или пък поради настъпване на грешка (един от най-лесните начини е да се използва флаг, който при успешен край се променя, а при грешка остава непроменен).

Процедурата, съдържаща завършващите инструкции може да върне управлението на програмата обратно към някоя начална точка и програмата да продължи да работи. По този начин процесът на грешките става до някаква степен управляем и се премахва нуждата постоянно да се следи дали е настъпила някаква грешка.

Това значително опростяване на програмата си има своята цена. След настъпване на грешка, управлението не може да се върне и да се продължи програмата след мястото на прекъсването, а винаги се връща в някакво начално състояние. Всички локални променливи са безвъзвратно загубени.

В повечето случаи стекът може да се препълни, поради факта, че при прекъсване, не винаги има възможност той да се възстанови в първоначалното състояние.

III.6.1.4 Използване на числов копроцесор

Окончателният вариант, избран за работа с реални числа, се базира основно на възможностите, които предоставят стандартните вече числови копроцесори. Независимо от факта, че по този начин могат да се избегнат всички проблеми, описани в III.6.1.3, използването на такъв копроцесор дава някои допълнителни предимства:

- всяка математическа операция, която се поддържа от копроцесора, може да бъде маскирана, т.е. използването й с невалидни аргументи да не доведе до спиране на програмата;
- точността на работа е доста по-висока, понеже операциите се извършват след като операндите се представят в разширения вътрешен формат;
- поддържат се редица "числа" – безкрайности (както положителни, така и отрицателни), безкрайно малки числа, неопределени числа, нормализирани и денормализирани числа и т.н.;
- всички операции работят както със стандартните числа, така и с нестандартните (като безкрайностите, например), т.е. делението на нула ще произведе безкрайност (или неопределеност, ако делимото също е било нула).

III.6.1.4.1 Общи положения

Реализацията на една компютърна система за работа с числа с плаваща точка е свързана с уточняването на много различни детайли и избиране на начин, по който да се избягват възникващите проблеми. Американският институт IEEE (Institute of Electronical and Electronics Engineers) е разработил няколко стандарта за аритметика с реални числа, с цел да се подобри точността, надеждността, преносимостта и съвместимостта на приложенията. IEEE стандартите описват добре дефинирани изисквания, които еднозначно определят резултата от всяка операция на реални числа.

Стандартите IEEE нямат задължителен характер нито за производителите на хардуер, нито за разработчиците на софтуер, но на практика, всички се съобразяват с тях. Разработените числови копроцесори от фамилията 80x87 на Intel, и съвместимите им копроцесори, произведени от други компании, поддържат изцяло характеристиките, свойствата и спецификациите, определени от IEEE:

- ограничения върху параметрите, дефиниращи стойности на основните и разширените формати за числа с плаваща точка;
- операции за събиране, изваждане, умножение, деление, тригонометрични, логаритмични и експоненциални функции, операции за сравнение;
- конвертиране от цели числа в реални числа, както и конвертиране на реално число от един формат в друг;
- изключителни ситуации, свързани с числа с плаваща точка и тяхната обработка, включително и на т.нар. неопределени числа (NaN).

III.6.1.4.2 Управляващи регистри

Копроцесорът съдържа два 16-битови управляващи регистъра. Първият, наречен *управляваща дума*, определя реакциите на копроцесора в различни ситуации, а вторият – *дума на състоянието* – извлича текущото състояние на копроцесора и последния резултат.

III.6.1.4.2.1 Управляваща дума

Управляващата дума предоставя възможност да се избере един от двата режима на работа: проективна затвореност на числовата система (по подразбиране) или афинна затвореност. В първия режим копроцесорът третират положителната и отрицателната безкрайности като една и съща безкрайност.

По време на своята работа копроцесорът може да се натъкне на особена ситуация. Например, при деление на нула са възможни две коренно различни реакции: да се предизвика прекъсване или да се върне резултат някоя от двете безкрайности (ако затвореността е афинна).

Значението на всеки от битовете от управляващата дума е показан на следната таблица:

Таблица III-7 Управляваща дума

Бит(ове)	Значение
0	**Невалидна операция**
1	**Денормализиран операнд**
2	**Деление на нула**
3	**Препълване**
4	**Машинна нула**
5	**Точност**
6	Запазен бит
7	Маска за разрешаване на прекъсванията
8-9	**Управление на точността**
10-11	Управление на закръглянето
12	**Управление на безкрайността (0=проективна, 1=афинна)**
13-15	Запазени битове

В Elica не се използват пряко всички битове, а само тези с номера от 0 до 5, 8, 9 и 12. В инициализационната си част, в тези битове се записват единици, с което се определя невалидните операции, деленията на нула, препълванията, машинната нула и препълванията да не генерират прекъсвания, безкрайностите да бъдат две – положителна и отрицателна, а точността да се управлява по-прецизно.

III.6.1.4.2.2 Дума на състоянието

Този регистър показва какво е текущото състояние на копроцесора и вида на последния резултат, изчислен от него. В регистъра има четири специални бита (C0-C3), които могат да се променят от аритметичните операции.

Таблица III-8 Дума на състоянието

Бит(ове)	Име	Значение
0		Невалидно действие
1		Денормализиран операнд
2		Деление на нула
3		Препълване
4		Машинна нула
5		Точност

6		Запазен бит
7		Заявка за прекъсване
8	**C0**	**Код на условията**
9	**C1**	**Код на условията**
10	**C2**	**Код на условията**
11-13		Указател на върха на стека
14	**C3**	**Код на условията**
15		Бит на заетостта

Особен интерес представляват кодовете на условията. Те се формират от набор от четири бита и дават допълнителна информация за типа на числото, записано във върха на стека на числовия копроцесор. Комбинациите от възможни състояния на тези битове, са показани в следващата таблица:

Таблица III-9 Кодове на условията

C3	C2	C1	C0	Значение
0	0	0	0	положително ненормализирано число
0	0	0	1	положително неопределено число
0	0	1	0	отрицателно ненормализирано число
0	0	1	1	отрицателно неопределено число
0	1	0	0	положително нормализирано число
0	1	0	1	положителна безкрайност
0	1	1	0	отрицателно нормализирано число
0	1	1	1	отрицателно безкрайност
1	0	0	0	положителна нула
1	0	0	1	(празен регистър)
1	0	1	0	отрицателна нула
1	0	1	1	(празен регистър)
1	1	0	0	положително денормализирано число
1	1	0	1	(празен регистър)
1	1	1	0	отрицателно денормализирано число
1	1	1	1	(празен регистър)

III.6.1.5 Обобщение

Избраният начин за работа с числа с плаваща точка, задоволява изцяло изискванията на система Elica. Прекъсванията поради невъзможност за изпълнение на математическа операция са изцяло маскирани. Потребителят разполага с разширено множество на реалните числа. Освен стандартните аргументи и резултати, всички операции поддържат работа с неопределени числа, а също и с безкрайно големи и безкрайно малки числа. От страна на системата не се изискват никакви допълнителни проверки и защити, поради това, че тези дейности се възлагат изцяло на числовия копроцесор.

III.6.2 Използване на графичен модул OpenGL

III.6.2.1 Какво е OpenGL

OpenGL е софтуерен интерфейс към графичен хардуер. Интерфейсът се състои от множество от няколкостотин процедури и функции, които позволяват на програмиста да определя тримерни обекти и операции, използвани в реализирането на висококачествена графика [6].

От гледна точка на програмиста, OpenGL е набор от команди за специфициране на двумерни или тримерни обекти и алгоритмите за тяхното контролиране. В най-честия случай програмистът може да отвори прозорец, да го свърже с OpenGL и да рисува директно върху него (чрез използване на интерфейса). Някои от командите на интерфейса се използват за рисуване на примитивни геометрични обекти (точки, отсечки, многоъгълници), други определят начина на растеризиране на геометричните примитиви (как се рисуват, оцветяват и осветяват)

III.6.2.2 Основи на OpenGL

OpenGL се занимава само с рендирането в графичен фреймбуфер и не поддържа никакви периферни устройства – мишки, клавиатури, писалки, таблети и т.н. OpenGL може да нарисува само примитивите: точка, отсечка, многоъгълник и правоъгълна зона от пиксели в някои от основните режими. Всички примитиви са дефинирани от група от един или повече върхове. Единичен връх дефинира точка, край на отсечка или връх на многоъгълник. Към всеки връх могат да се асоциират различни данни – координати в пространството, параметри на цвета, координати на нормалния вектор, координати на текстурата и т.н.)

Командите към OpenGL винаги се обработват в реда на издаването им, независимо че не се гарантира за колко време ще се изпълни всяка команда. OpenGL предоставя директен контрол над фундаменталните операции в двумерната и тримерната графика: трансформационни матрици, коефициенти на уравнения на осветяване, методи за антиалиасинг и други.

Моделът на интерпретация на OpenGL команди е клиент-сървер. Програмата-клиент изпраща команди, които се интерпретират и изпълняват от OpenGL-сървера. Сървърът може да се намира на същия компютър като клиента, но може и да работи на отдалечена система. В този смисъл OpenGL е мрежово прозрачен. OpenGL е проектиран да работи на широк кръг от платформи с различни графични възможности и производителност. Това ще позволи добре написаните приложения относително лесно да се прехвърлят от една платформа на друга.

III.6.2.3 Конвенция Begin-End

Основната конвенция при дефинирането на обекти е използването на Begin-End. Началото на всяка дефиниция започва с командата glBegin, а краят се обозначава с glEnd. В зоната между тези две команди се поставя редица от команди за дефиниране на върхове. Как те ще бъдат групирани, зависи единствено от режима, зададен в glBegin.

III.6.2.3.1 Точки

В този режим всички дефинирани върхове в рамките на един Begin-End блок са самостоятелни точки в пространството.

III.6.2.3.2 Начупена линия

Съществуват три режима на рисуване на начупени линии. Най-простият е точките, създадени в рамките на един Begin-End блок, да формират върховете на отворена начупена линия. Следващият режим е да се създаде затворена начупена линия. Третият режим е да се групират върховете два по два и да се създаде набор от независими отсечки.

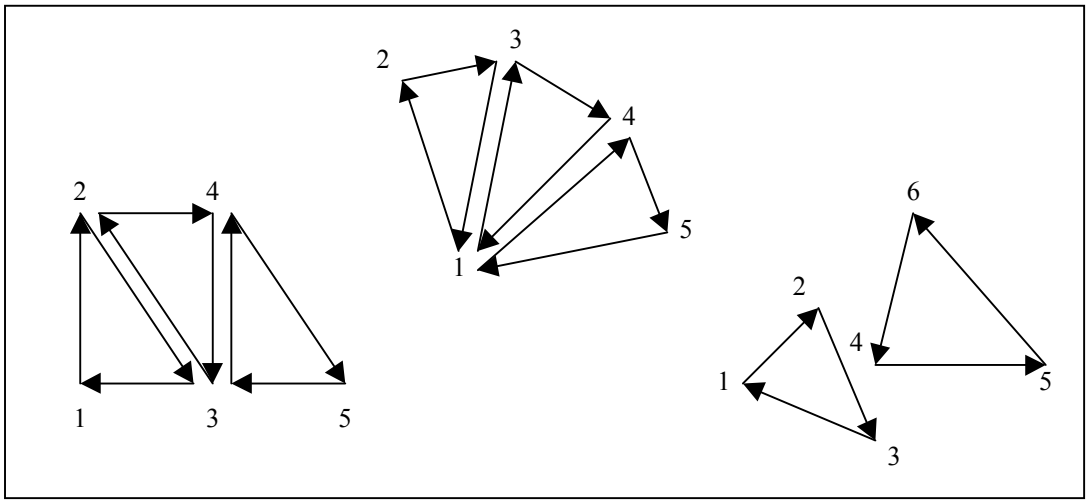
III.6.2.3.3 Многоъгълници

Създаването на многоъгълници в OpenGL става по абсолютно същия начин като създаването на затворена начупена линия. Единствената разлика е, че за многоъгълниците може да се дефинират свойства на вътрешността, а не само на контура.

III.6.2.3.4 Триъгълници

Създаването на триъгълници е един от основните начини за създаване на сложни повърхнини. Те се разбиват на малки триъгълници, които приближават повърхнината. Блоковете Begin-End могат да се използват за дефинирането на три вида редици от триъгълници. Разработчикът на графични приложения трябва сам да избере точно кой вид да използва според спецификата на повърхнината, която ще се дефинира.

Фигура III-35 Триъгълници в OpenGL



Най-вляво на Фигура III-35 е показано дефинирането на ивица от триъгълници. Особеното, е че това разполагане спестява създаването на върхове с едни и същи координати. Първите три точки от редицата дефинират първия триъгълник. Всяка следваща точка дефинира по един нов триъгълник, долепен до предния.

В средата на фигурата е показано ветрило от триъгълници. И при него с първите три точки се дефинира триъгълник, а всяка следваща – по един нов. Особеното е че също се пестят дефиниции на точки, а създадените триъгълници са с общ връх.

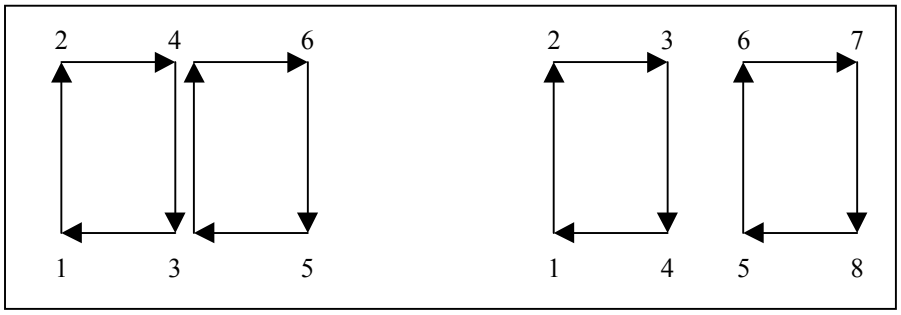
Вдясно са показани самостоятелни триъгълници. При тях всеки един се задава точно с три точки.

Независимо кой начин на дефиниране на триъгълници е избран, разработчикът трябва да отчита ориентацията на триъгълниците, понеже това влияе на запълването му и за определянето коя е лицевата страна на триъгълника.

III.6.2.3.5 Четириъгълници

В някои случаи описването на повърхнина с триъгълници е малко по-сложно и по-неестествено спрямо описването ѝ с четириъгълници (било те правоъгълници или не). OpenGL предоставя възможност за дефиниране на два вида набори от четириъгълници – ивици и самостоятелни.

Фигура III-36 Четириъгълници в OpenGL



На Фигура III-36 вляво е показано как се използват точките в Begin-End блок за дефинирането на ивица от четириъгълници, а вдясно – независими четириъгълници. Ако точките не са достатъчни за пълното дефиниране на съответния вид фигура, последният, непълен четириъгълник се игнорира.

Както и при дефинирането на триъгълници, разработчикът трябва да е наясно с ориентацията на четириъгълниците.

III.6.2.4 Контролиране на характеристиките на обектите

Създаването на сложни обекти в OpenGL се прави като обектите "се раздробяват" на набор от триъгълници и четириъгълници. Всеки от тях може да бъде дефиниран със собствени характеристики. В общия случай, тези характеристики определят цвета, осветеността, дебелината и стила на рисувания примитив.

Цветът се определя с командата glColor. Както повечето команди в OpenGL и тази има множество варианти според типа на аргументите си, но в Elica се използва само този, при който числовите стойности на компонентите на цветовете са в интервала от 0 до 255. Избраният цвят е базисен за обекта, към който е дефиниран. Допълнителните обработки като осветяване и прилагане на текстури могат да променят цвета. Осветяването се определя от режима на осветяване. Ако се зададе командата glShadeModel(GL_SMOOTH), прилага се осветяване, а при аргумент GL_FLAT всички примитиви се рисуват с единичен цвят. Естествено, осветеността зависи и от много други фактори – тип и разположение на светлинните източници, тип на материята на осветявания обект, начин на осветяване. Друга особеност е, че с GL_SMOOTH се определя начинът, по който да се променя цветът около ръбовете на триъгълниците, изграждащи даден обект, така че цветът да се променя плавно. Вътрешните механизми осъществяват това като променят и преизчисляват нормалния вектор в близост до ръбовете. Когато това не може да бъде осъществено от OpenGL, потребителят може изрично да определи нормалният вектор към връх на триъгълник с командата glNormal.

Когато се налага да се дефинират по-големи точки или по-дебели линии, трябва да се използват командите glPointSize и glLineWidth. Дебелината може и да не е цяло число. В такива случаи, ако е разрешен режимът GL_LINE_SMOOTH, те се рисуват с максимално допустимия антиалиасинг, за да представят зададената дебелина. В противен случай дебелината се закръгля до цяло число. За да се повиши функционалността на рисуваните примитиви могат да се задават едномерни и двумерни шаблони с командите glLineStipple и glPolygonStipple. Те определят кои точки, принадлежащи на визуалния образ на линия или многоъгълник да бъдат реално визуализирани и кои не. Следвайки конвенцията на OpenGL, тези режими могат да се използват само ако са разрешени с GL_LINE_STIPPLE и GL_POLYGON_STIPPLE. Тази конвенция на графичната система се е наложила поради факта, че прилагането на някои характеристики забавя съществено рендирането на обектите и за тази цел, тези характеристики трябва да бъдат забранени, ако не се прилагат.

Най-богати на характеристики са многоъгълниците. Освен изброените тука характеристики, за тях може да се зададе дали представляват част от двустранни повърхности или само едностранни. Двустранните повърхности в OpenGL изискват повече ресурси, понеже и двете страни се прилага набора от разрешени характеристики. Видът на изобразяване на многоъгълниците с контролира с командата glPolygonMode. С нея освен броя на страни, се определя кои части от многоъгълника да бъдат изобразени – само върхните точки, само ръбовете или целите многоъгълници. В някои случи се налага и да се използва командата glEdgeFlag, с която се определя кои ръбове на многоъгълник са и ръбове на повърхнината, която го "съдържа".

III.6.2.5 Тримерни трансформации

Графичният модул дава възможност на потребителите да променят пространствените параметри на обектите. Това се използва от Elica за улесняване на рисуването на някои по-сложни обекти. В тези случаи, системата дефинира нормализиран вид на обекта и с подходящи трансформации, които касаят не само върховете, но и ръбовете, нормалите и текстурите, преобразува обекта в желания вид.

Всички трансформации се извършват с построяването на съответната трансформационна матрица, която се умножава системно по текущата проекционна матрица. Задаването на трансформацията може да стане директно, като се опишат всички параметри на трансформационната матрица или индиректно, като се подадат параметрите на гледната точка с командата glLookAt. В този случай, се подават координатите на три вектора: радиус-вектор на точката от която "гледаме", радиус-вектор на точката към която "гледаме" и вектор, обозначаващ посоката "нагоре"

Когато трансформацията е по-проста, могат да се използват наготово някои от конструкторите на матрици: glTranslate, glScale и glRotate. Те се използват за създаването на транслационна матрица, на матрица за мащабиране и матрица на завъртане. По-особена е командата за ротация. С използването й се генерира матрица, която осъществява завъртане на някакъв ъгъл около линия минаваща през точка (0,0,0) и избрана друга точка в пространството.

III.6.2.6 Текстури и букви

Една от най-ефектните възможности на OpenGL е да прилага текстури върху многоъгълници и повърхнини. Естествено, тези възможности не са пряко достъпни, а използването им изисква добро познаване на OpenGL. В паметта текстурите се представят като правоъгълна картинка с размери на страните равни на степен на двойката. Прилагането на текстура върху повърхност изисква задаването на множество характеристики с командата glTexParameteri. Най-важните от тях определят цикличността на текстурата в хоризонтална и вертикална посока, и начина, по който да се пресмята съответствието между пикселите от текстурата и визуалния образ на обекта в случаите, когато то не е биективно (както е в повечето случаи).

Осветеността и модела на оцветяване на текстура се определя от glTexEnv, а начина на съответствие на координатната система на текстурата и координатната система върху повърхнината се определя с командите glTexImage2D и glTexCoord2f, които трябва да се използват при дефинирането на всеки връх.

Визуализирането на текстове в OpenGL е един от малкото елементи, които изглеждат по-лесни, отколкото са. Понеже OpenGL е платформено независима и платформено инвариантна, невъзможно е да се гарантира еднаквият външен вид на визуализирания текст. За да се реши този проблем, в OpenGL няма пряка възможност да се показва текст. Начинът за създаване на текстови обекти в Windows е следният: използва се функцията CreateFontIndirect за създаване на описание на шрифт. След това се използва

командата wglUseFontOutlines, с която се създава тримерно изображение на всяка от буквите на избрания шрифт. Визуализирането на текст се реализира, като се създават последователно всяка от буквите като OpenGL дефиниции на тримерни обекти.

III.7 По-важни проблеми и решения при реализацията на разширенията и потребителските библиотеки

III.7.1 Заложени концепции в Graphix

Библиотеката Graphix е втората по важност в Elica след библиотеката Logo. Благодарение на Graphix в системата е възможно да се рисуват тримерни обекти, сцената да се върти, да се оцветяват и осветяват повърхнини, да се скриват линии, да се наслагват текстури и т.н.

Поради своята сложност и големина, реализацията на Graphix използва няколко концепции, които улесниха изграждането и ще направят по-лесно бъдещото й разширяване.

III.7.1.1 Централизирано обработване на характеристиките

Една от използваните концепции е да се използва централизирано обработване на характеристиките на рисуваните обекти. В библиотеката са дефинирани около 25 обекта, с най-различни характеристики, чийто брой е над 20. Ако на всеки обект характеристиките се обработват индивидуално, това ще изисква написването и поддържането на огромен програмен код, който в една или друга степен ще се дели на почти еднакви фрагменти. За тази цел характеристиките са групирани и за всяка група е създадена процедура, която ги обработва.

Например, групата на текстурите обработва характеристики, които определят коя е текстурата, в какъв мащаб да се прилага, на какъв ъгъл да се завърти и колко прецизно да се интерполират междинните точки. Много от обектите (т.е. всички двумерни и тримерни) могат да бъдат нарисувани с текстура. Затова процедурата за зареждане на характеристиките на текстурите е една и се ползва от много места.

По аналогичен начин е решен и въпросът с линиите. Процедурата за четене на характеристиките на линейните обекти определя цвета, дебелината, пунктираността и гладкостта на рисуваните линии. Тези характеристики се използват всеки път, когато се рисува рамков обект, без значение от броя на измеренията му, или се рисува линеен обект.

Същата концепция се прилага и над всички останали характеристики. По този начин отделните процедури за генериране образа на геометрични обекти извиква онези процедури, които зареждат параметри отнасящи се до рисувания обект, след което обектът се рисува, използвайки вече заредените параметри.

Централизираната обработка на характеристиките се използва и за задаване на подразбиращи се стойности на характеристиките, които не са зададени явно от потребителя. Това се отнася само до тези характеристики, които са задължителни.

Ще разгледаме само един пример – четене на характеристиките при генерирането на изображението на цилиндър. Процесът включва следните стъпки:

- *Обработка на радиусите*: Проверява се зададен ли е радиус. Ако не е зададен, приема се, че е 1. Проверява се дали са зададени три радиуса – два за основата и един за височината. Ако не са, "пропуснатите" радиуси се приемат, че са със стойност равна на тази на първия радиус. Когато радиусите на основата са различни, тя е елипса.
- *Обработка на нормалата*: Проверява се дали е зададен коефициент, който се използва при определяне големината на нормалния вектор.

- *Обработка на ориентацията*: Проверява се до какви координати да се транслира обекта, как да се завърти в пространството, така че главната му ос да съвпадне със зададена от потребителя посока, и как да се завърти около вече определената нова главна ос.
- *Обработка на стил на точка*: Проверява се, ако ще се рисуват точки, какъв да бъде техният размер.
- *Обработка на стил на линия*: Проверява се, ако ще се рисуват линии, каква да бъде тяхната дебелина, колко фино да се рисуват, ако се използва шаблон – какъв е той, на кои точки влияе и какъв ще е мащаба му.
- *Обработка на стил на полигон*: Проверява се необходимия брой точки за представяне на многоъгълника и режима на рисуването му (чрез точки, чрез линии или чрез запълнени многоъгълници). Ако има включени светлинни източници се проверяват техните параметри и влиянието им на осветяването на многоъгълника. Проверява се наличието на текстура, данни за самата картинка, задаваща текстурата, нейния мащаб, завъртяност и прецизност при интеполирането на междинните точки. Ако оцветяването е плоско се определят дали е зададен двумерен шаблон, и дали всеки връх ще се характеризира със собствен цвят или всички върхове ще имат един и същ цвят.
- Обработка на цвят: Проверява се дали потребителят е задал цвят на обекта. Ако липсва цвят се използва цвета по подразбиране. Ако обектът се рисува от точки, цветът определя цвета на точката, ако е от линии – цвета на линиите, ако е от полигони – цвета на полигоните, а ако е от полигони с наложена текстура – цветовия оттенък на текстурата.
- *Обработка на интервални ограничения*: Проверява се дали ще се рисува завършен цилиндър или ще има ограничения върху пълнотата му. Ако няма ограничения се приема, че ще се рисува целия, ако има – се проверява как да се попълват липсващите области.
- *Обработка на тип на обекта*: Проверява се дали цилиндърът да бъде затворен отгоре и отдолу или да се представя като тръба. Освен това се проверява какво е отношението между основата и горната част на цилиндъра. Ако отношението е 1 се рисува цилиндър, ако е 0 се рисува конус, а ако е някаква междинна стойност се рисува скосен конус.

III.7.1.2 Нормализирано представяне на обектите

Втората концепция, която се използва при реализирането на библиотеката Graphix е, че вътрешният програмен код, който генерира описанието на обектите, винаги се базира на тяхната нормална форма.

Причината за това е, че този програмен код трябва да апроксимира повърхността на обекта с триъгълници или четириъгълници. Ако той трябва да апроксимира произволна допустима вариация на обекта (т.е. да се отчита положението, размера и ориентираността му в пространството), изчисленията ще са доста по-сложни и по-бавни.

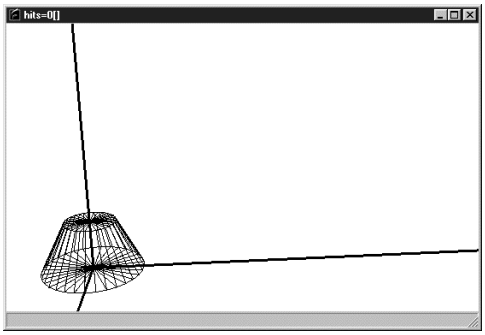
Именно поради тази причина всички обекти в Graphix, с изключение на точката, линията, лъча и отсечката, се генерират в тяхната нормализирана форма и после чрез трансформационна матрица се определя техния реален размер, положение в пространството и ориентираност [19].

Прилагането на трансформационната матрица се извършва на ниво на координати и се осъществява от базовата графична система, която е оптимизирана и използва пълните

възможности на процесора. Ако той има възможности за MMX операции, графичната система ги използва за бързо умножение на матрици и вектори.

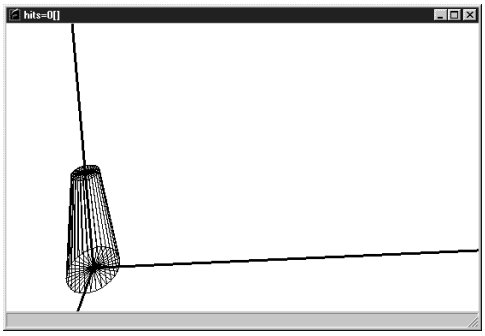
Най-често обектите се рисуват на 6-7 стъпки, при всяка от които се използват и прилагат определени негови характеристики.

Фигура III-37 Нормална форма на скосен конус



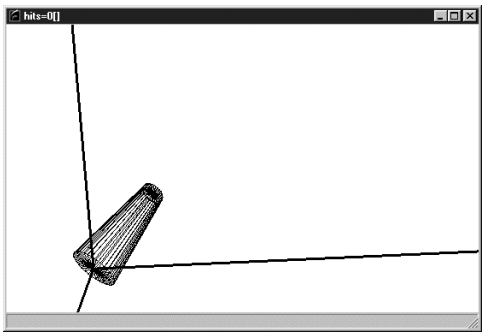
Първата стъпка е да се генерира нормализираният вид на обекта. В случая това е пресечен конус с височина 1 и основа - окръжност с радиус 1.

Фигура III-38 Прилагане на мащабиране



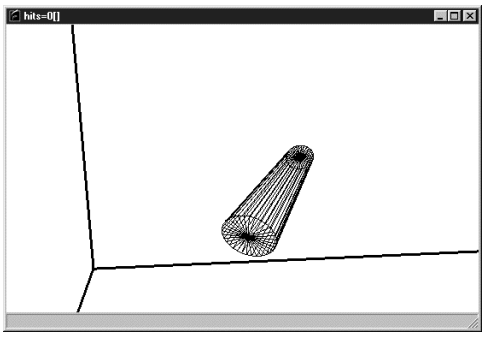
Втората стъпка е да се приложи мащабиране. При тази операция се променят размерите на обекта. Мащабирането се извършва с различен коефициент за всяко измерение. Както е показано на примера, след мащабирането, конусът е станал по-висок и странично сплескан, като основата му вече е елипса.

Фигура III-39 Прилагане на завъртане в пространството



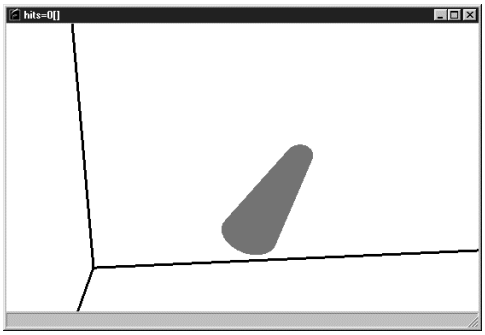
Следващата стъпка е да се завърти обекта така, че основната му ос да сочи в определена посока в пространството. В Graphix тази посока се задава с вектор. Освен тази ориентация, към обекта може да се приложи и завъртане около неговата ос, но в примера със скосен конус тази ротация няма да предизвика промяна.

Фигура III-40 Транслация на фигура



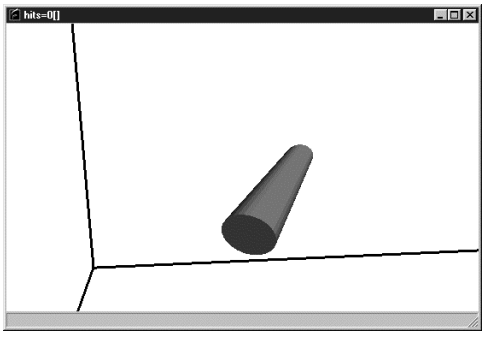
Последната стъпка относно пространствените характеристики на обекта е да се транслира, така че базовата му точка, най-често това е (0,0) в нормалната форма на обекта, да съвпада с предварително определена друга точка в пространството. При тази транслация размерите и ориентацията на обекта се запазват.

Фигура III-41 Оцветяване на геометричен обект



Петата стъпка е обекта да се оцвети. Разбира се, ако се изисква рисуването на обекта да е само на ниво контури, тази и следващите стъпки се прескачат. Оцветяването на обекта означава, че многоъгълниците (най-често триъгълници и четириъгълници) на които е декомпозирана повърхността му се оцветяват и частите от обекта, които остават скрити не се рисуват.

Фигура III-42 Осветяване и светлосенки

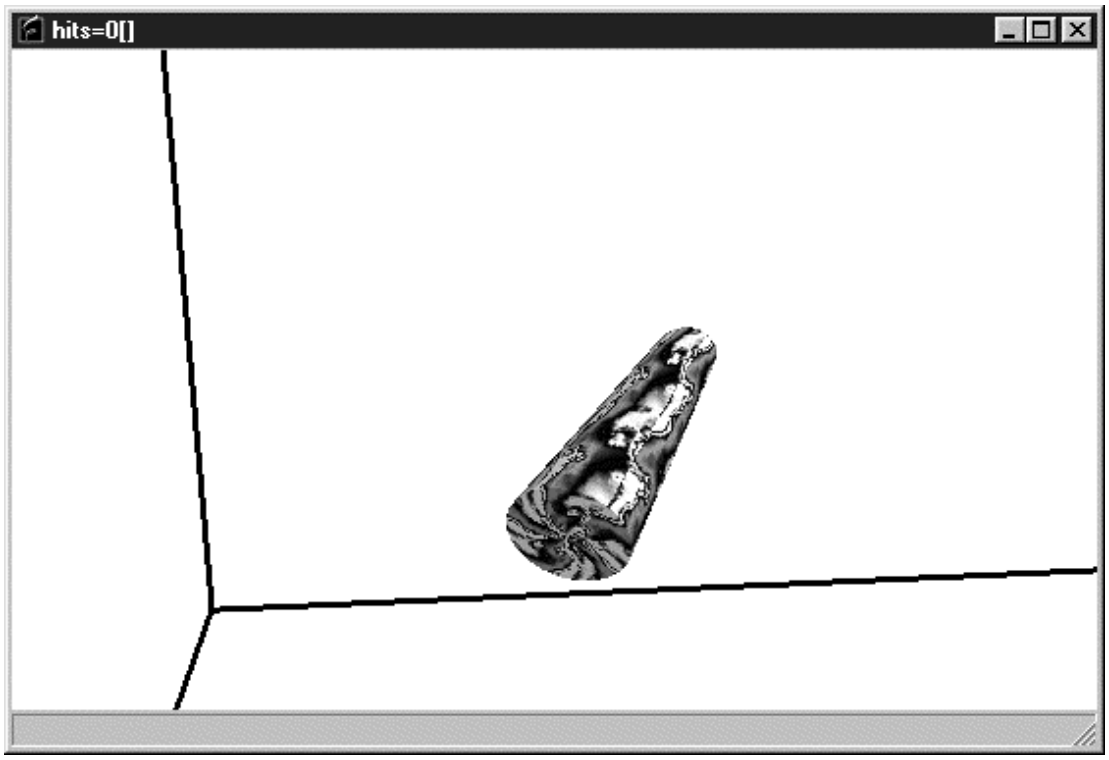


За някои обекти оцветяването не е достатъчно за тяхното визуализиране. Особено това важи за тримерните обекти. Когато оцветяването е хомогенно, извивките на повърхността са трудно забележими и за тази цел се прилага осветяването. За да може един обект да бъде осветен, за всяка негова точка трябва да бъде зададен нормален вектор. Силата на осветяване зависи от ъгъла, който сключва нормалния вектор с посоката от светлинния източник.

Гладкостта на осветяването може да се контролира по два начина – чрез увеличаване на броя апроксимиращи полигони или чрез интерполация на нормалните вектори. И двата

начина са заложени в реализацията, като тяхното използване се определя от потребителя.

Фигура III-43 Прилагане на текстура



Последната стъпка е да се наложи текстура върху повърхността на обекта. Текстурата се прилага не върху образа на екрана, а върху обекта и се забелязва изкривяването и деформирането и според кривината на повърхнината и пространственото положение и ориентация на обектите.

III.7.1.3 Обобщени обекти

Генерирането на част от изображенията на 25-те обекта в Graphix могат да се унифицират. Например, с една и съща параметрично управляема процедура може да се генерират точките, описващи цилиндър, тръба, конус и пресечен конус. По аналогичен начин се реализират паралелепипед и куб, елипса и окръжност, елипсоид и сфера и т.н.

Част от генериращите процедура са дори и още по-мощни. По този начин те могат да се използват за създаването на повече и по-разнообразни графични обекти.

В настоящата точка ще разгледаме единствено параметризацията, която определя формата на два от базовите обекти.

III.7.1.3.1 Обобщен цилиндър

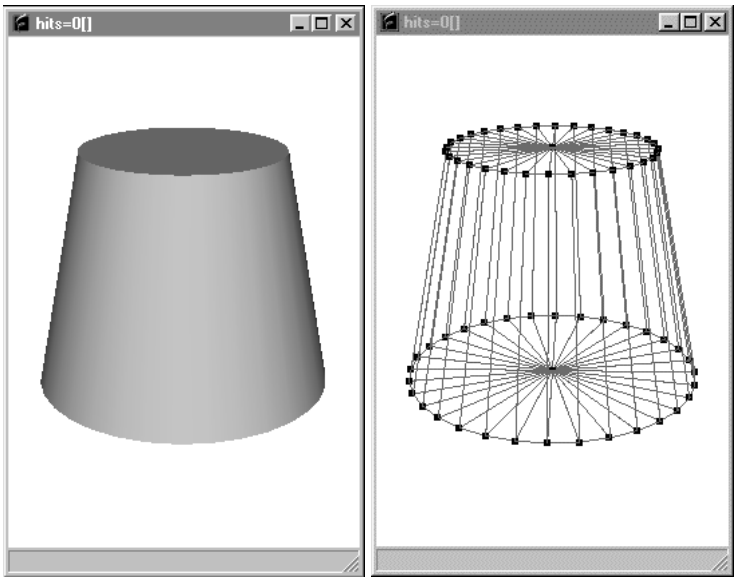
Първият обект, който ще разгледаме е цилиндър. При него може да се определя съотношението на радиусите в горния и долния му край. Ако съотношението е 1:1, то те са равни и рисуваната фигура е цилиндър. Ако съотношението е 0:1, то горния край на цилиндъра се трансформира в точка и фигурата е конус.

На Фигура III-44 е показан случаят на съотношение 0.68:1 при което се получава пресечен конус. Особеността е, че височината на фигурата е височината на пресечения конус така както е пресечен – т.е. ако потребителят иска да създаде конус с височина 2

и да го пресече с равнина на височина 1, трябва да зададе височина 1 и съотношение 0.5:1.

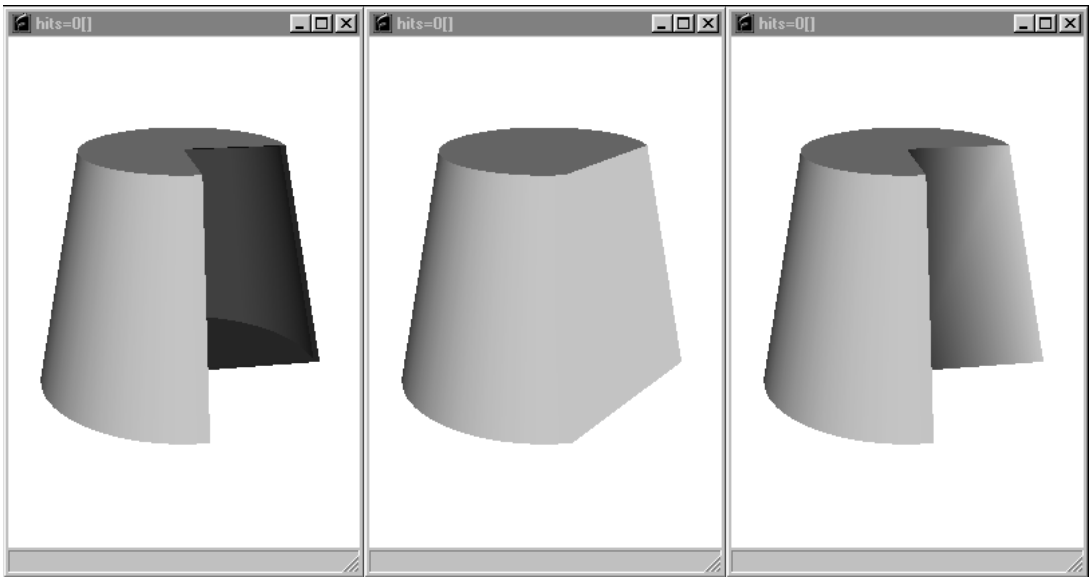
При генерирането на повърхността, се обхождат равномерно разположени точки по периметъра на две окръжности – едната, определяща основата, а другата – противоположната й стена. Стените, съответстващи на тях се апроксимират с две ветрилообразни множества от триъгълници. Страничната стена се получава с четириъгълници, свързващи съответните точки от горната и долната окръжности. Ако отношението е 0:1 вместо горната окръжност се използва точка и в такъв случай вместо четириъгълници се създават отново триъгълници.

Фигура III-44 Пълен пресечен конус



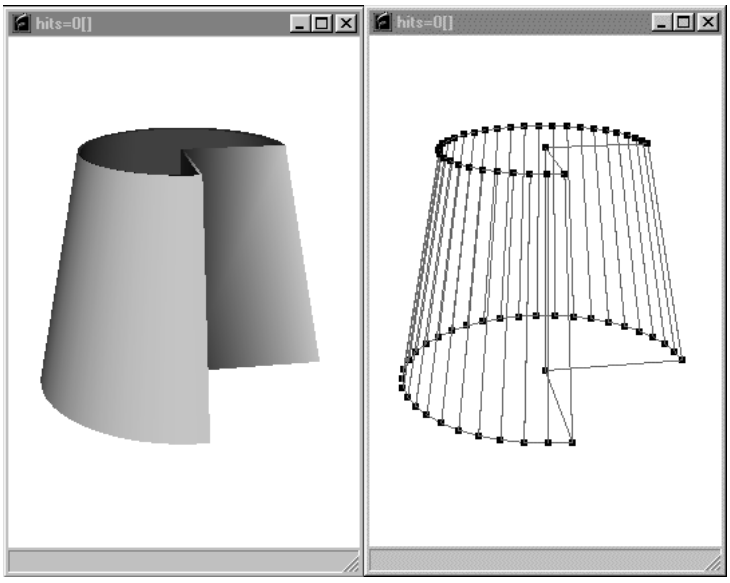
Втората характеристика, която може да се параметризира от потребителя е дали да се използва пълна окръжност или сегмент от нея да липсва. На следващата фигура са показани три варианта на изрязване на вертикален сегмент. Големината и положението на сегмента се определя с двойка ъгли – начален и краен, които се прилагат върху окръжностите. При тази операция броят на точките, формиращи дъгите, не се намаля, а просто те се сгъстяват.

Фигура III-45 Изрязване на сегмент от пресечен конус



Първият начин на изрязване е без да се съединяват краищата – в този случай се вижда вътрешността на пресечения конус. Вторият начин е те да се свържат директно, както това е показано на втората картинка и третият – да се свържат през централната ос на конуса. Реализацията на трите случая става чрез добавянето на 0, 1 или два четириъгълника (при цилиндър и пресечен конус) или триъгълника (при конус).

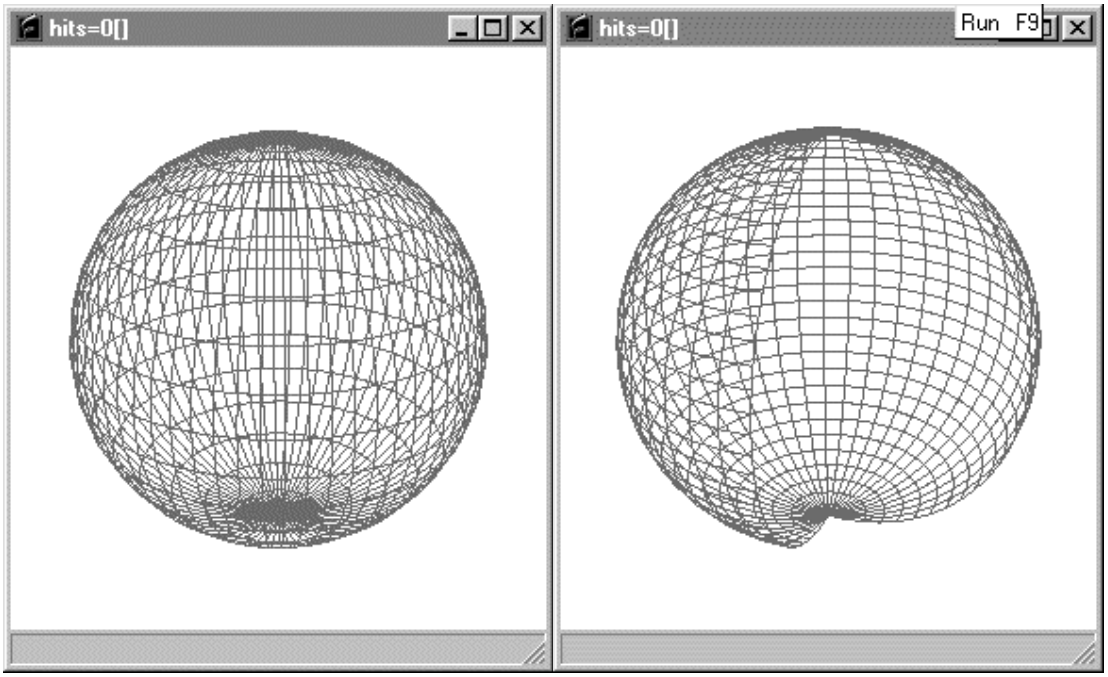
Фигура III-46 Отворен пресечен конус



Последната параметризация, достъпна на потребителя, е дали да се рисуват горната и долната част (при конуса – само долната). Ако те липсват, фигурата става отворена отгоре и отдолу. Това не променя по никакъв начин разположението на точките, а влияе единствено на това кои триъгълници и четириъгълници ще се създават и кои не.

III.7.1.3.2 Обобщена сфера

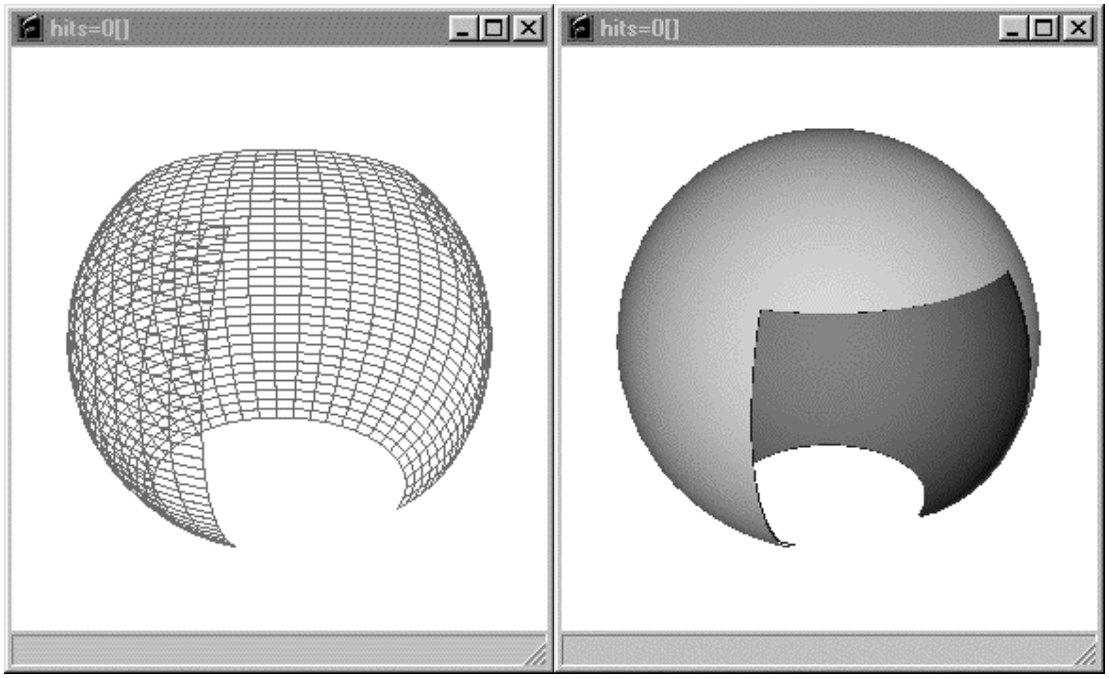
Фигура III-47 Пълна и вертикално изрязана сфера



Обобщената сфера се използва за генерирането на сфери, елипсоиди, сферични сегменти, дискове, пояси и други фигури, за които няма стандартно име.

На лявата част от Фигура III-47 е показано как се апроксимира сфера. Базовите точки са разположени по окръжности, описващи различни слоеве от сферата. Една от възможностите за промяна на формата е да се зададе вертикален сегмент, подобно на пресечения конус, който се изключва от рисуването, за сметка на сгъстяване на останалите точки.

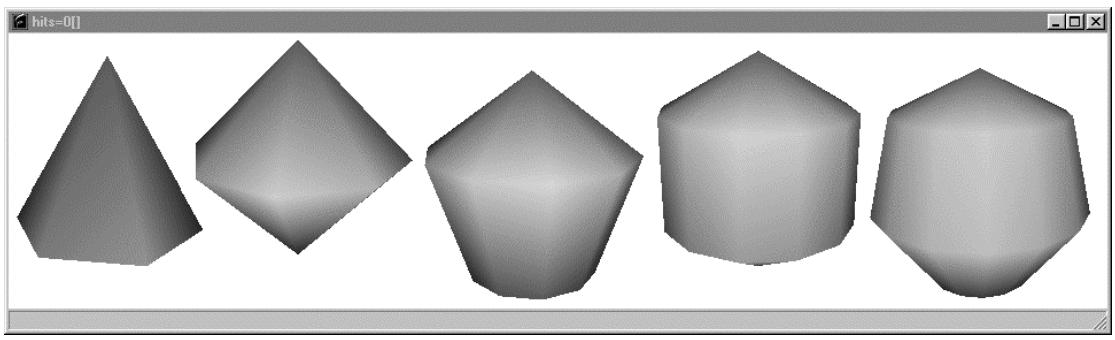
Фигура III-48 Вертикално и хоризонтално изрязване на сфера



Освен вертикалното изрязване от сферата могат да се изключат горната или долната зона. Размерът на изрязване се определя от потребителя и се задава в градуси. Изрязването може да се използва и с обратен знак, при което се получава фигурата "шлем" показана по-горе вдясно. Ако и двата края на изрязването са с обратен знак, може да се получи сфера с изрязан "прозорец" - сферичен правоъгълник.

Както и при пресечения конус, хоризонталните ивици, съставящи повърхността се задават като свързани области. Това се налага, за да може да се използва интерполация на нормалния вектор при осветяване на обекта за постигане на плавно преливане на светлосенките.

Фигура III-49 Занижен брой базови точки



Особено интересни пространствени обекти се получават при намаляне на броя на базовите точки. Обикновено техният брой се увеличава за постигането на по-точни

апроксимации, но и при тяхното намаляване също се получават полезни обекти. На Фигура III-49 е показан един и същ обект – сфера.

Броят на базовите точки за петте фигури – "пирамида", "диамант", "солница", "шатра" и "абажур" е съответно 3, 4, 5, 6 и 7. При тези фигури промяната на ориентацията в пространството би имало визуално значение, докато за сферата ориентацията е без значение.

Всички обемни обекти, като обобщения конус, и всички тримерни, които се рисуват чрез апроксимация на крива имат характеристика, с която се определя колко триъгълници, четириъгълници или отсечки да се използват при генериране на образа им. Естествено този брой не може да бъде точно зададен, а се определя само порядъкът. Това е така, защото всеки обект се рисува чрез няколко фрагмента и броят на точки се отнася само за един от фрагментите. Например, хиперболата се рисува като четири отделни клона, т.е. ако базовите точки са N, хиперболата се рисува чрез отсечки свързващи 4N точки.

III.7.2 Особености на реализация на библиотека Geomland

III.7.2.1 Виртуални полета

Библиотеката Geomland е реализирана на Elica Logo, без пряко използване на външни библиотеки. Geomland съдържа дефинициите на обекти, с които се реализират графичните обекти и операциите над тях в PGS. Основната част от библиотеката – рисуването на графичните обекти, е изнесена извън нея. Като цяло Geomland използва Graphix за рисуването, като това, което прави е да контролира по подходящ начин параметрите.

За почти всички обекти в PGS са направени еквивалентни обекти в Geomland, които са дефинирани така, че да са синтактично и семантично едни и същи.

Главно поради особености на PGS, в система Elica са въведени виртуалните полета. Най-простият пример за това е командата и функцията `ABSC`. Използвана като процедура, с нея се променя абсцисата на точка, а използвана като функция я връща.

Без използването на виртуални полета подобна функционалност може да се реализира в Elica, защото е възможно броя на аргументите да е променлив, а също и една и съща подпрограма да се използва и като процедура и като функция, но реализацията ще има следната особеност:

- Ако подпрограмата `ABSC` е дефинирана с един параметър, като функция тя ще се ползва без проблеми, а като процедура - трябва да се загражда в кръгли скоби, за да може вторият аргумент да се вземе под внимание
- Ако подпрограмата `ABSC` е дефинирана с два параметъра, като процедура използването й ще е без особеност, докато като функция ще трябва да се загради в кръгли скоби, за да не игнорира вторият параметър, винаги, когато не е в края на израз.

Причината за тази особеност се крие в това, че за разлика от PGS библиотеките в Elica не са вградени и няма начин да се укаже, че броят на параметри зависи от това, дали подпрограма се използва като функция или като процедура.

Виртуалните полета в Elica са функции/процедури, които формално не са дефинирани, но са дефинирани техни варианти – един функционален и един процедурен. Имената на вариантите са фиксирани и затова системата знае как да ги търси. Функционалният вариант е с име, започващо с "`get_`", следвано от истинското име на виртуалния параметър. Процедурният вариант започва с "`put_`".

Следвайки тази концепция, ABSC е реализирано по начин, показан във Втора глава. На това място трябва да се отбележи, че не е задължително параметрите на get_ и put_ подпрограмите да са точно такива, като в примера – в някои случаи може да се наложи достъпа до виртуалните полета да изисква повече или по-различни параметри.

Пример III-1 Виртуални полета за достъп до реални полета

Команди

```
to get_absc :obj
   output :obj.x
end
to put_absc "name :value
   make (:name).("x) :value
end
```

В случая get_absc извлича стойността на полето ABSC от определен обект, а put_absc го променя. Когато в програма се срещне идентификаторът ABSC, системата се опитва да намери променлива с такова име. Ако подобна променлива не се намери, системата определя като как в подпрограма би се използвал този идентификатор. Ако е като функция, прави се второ търсене, но пред името, което се търси се слага "get_", в противен случай се слага "put_".

Ползата от виртуалните полета се оказва, че е по-голяма от това само да се постигне синтактична съвместимост с PGS. Чрез виртуалните полета могат да се "симулират" полета, които в реалност не съществуват. Четенето на виртуално поле става като неговата стойност се пресмята в реално време, а промяната му се свежда до промяна на едно или няколко реални полета.

Един от най-показателните примери за това е дефиницията на length.

Пример III-2 Създаване на виртуални полета

Команди

```
to get_length :obj
   output distance :obj.initial :obj.final
end

to put_length "name :value
   make local "i :((:name)("initial))
   make local "f :((:name)("final))
   make (:name)("final) :i+(:f-:i)*:value/(distance :i :f)
end
```

III.7.2.2 Сечение на геометрични обекти

Сечението на геометрични обекти в Geomland са осъществява от набор от функции, дефинирани в отделен файл, които се използват от функцията isec. Единствената роля на isec е да определи типа на двата си аргумента (това са обектите, чието сечение се търси) и да извика съответната функция. Истинските функции за сечения са на точка с точка, точка с линия, линия и точка, точка и окръжност, окръжност и точка, линия и линия, окръжност и окръжност, линия и окръжност и последната функция – сечение на окръжност и линия.

От изброените 9 вида сечения основните са 6. Останалите 3 се получават от основните. Реализацията на 6-те основни функции следва от аналитичното изчисление на сечението.

III.7.2.2.1 Сечение на точка с точка

Нека са дадени две точки $P_1(X_1, Y_1)$ и $P_2(X_2, Y_2)$. Ако е изпълнено:

(III.7-1a) $$\begin{vmatrix} X_1 = X_2 \\ Y_1 = Y_2 \end{vmatrix}$$

сечението е точка, която съвпада и с двете точки. В противен случай сечението е празно множество.

III.7.2.2.2 Сечение на точка с линия

Нека са дадени точката $P(X, Y)$ и линията $L(X_1, Y_1, X_2, Y_2)$. Изчисляваме разликите между проекциите на точките, определящи правата:

(III.7-2b) $$\begin{vmatrix} \Delta X = X_2 - X_1 \\ \Delta Y = Y_2 - Y_1 \end{vmatrix}$$

Точката *P* лежи на правата *L*, тогава и само тогава, когато е изпълнено

(III.7-3b) $$\frac{X - X_1}{X_2 - X_1} = \frac{Y - Y_1}{Y_2 - Y_1},$$

което, използвайки (III.7-2b), може да се развие така:

(III.7-4b) $$\frac{X - X_1}{\Delta X} = \frac{Y - Y_1}{\Delta Y},$$

(III.7-5b) $$(X - X_1)\Delta Y = (Y - Y_1)\Delta X$$

Последното равенство (III.7-5b) се използва за проверка, дали точка лежи на права. Всички тези преобразувания се налагат, за да се избегне изследването на подслучаи. Ако равенството е изпълнено, сечението е точка, която съвпада с *P*. В противен случай сечение е празно множество.

III.7.2.2.3 Сечение на точка с окръжност

Нека са дадени точката $P(X, Y)$ и окръжността $C(X_C, Y_C, R)$. Точката ще лежи на окръжността тогава и само тогава, когато разстоянието ѝ до центъра е равно на радиуса. От изчислителна гледна точка е по-добре да сравняваме квадратите на разстоянието и на радиуса:

(III.7-6c) $$R^2 = (X - X_C)^2 + (Y - Y_C)^2$$

Ако равенството (III.7-6c) е изпълнено, сечението е точка, съвпадаща с *P*. В противен случай сечението е празно множество.

III.7.2.2.4 Сечение на линия с линия

Нека са дадени двете линии $L(X_1, Y_1, X_2, Y_2)$ и $L(X_3, Y_3, X_4, Y_4)$. Изразявайки сечението параметрично, то може да бъде намерено, ако се определят коефициентите *t* и *q* от следната система:

(III.7-7d) $$\begin{vmatrix} X_1 + t(X_2 - X_1) = X_3 + q(X_4 - X_3) \\ Y_1 + t(Y_2 - Y_1) = Y_3 + q(Y_4 - Y_3) \end{vmatrix},$$

която е равносилна на:

(III.7-8d) $$\begin{vmatrix} (X_1 - X_3) + t(X_2 - X_1) = q(X_4 - X_3) \\ (Y_1 - Y_3) + t(Y_2 - Y_1) = q(Y_4 - Y_3) \end{vmatrix}$$

Като умножим първото равенство с $(Y_2 - Y_1)$, а второто с $(X_2 - X_1)$, ще получим системата:

(III.7-9d) $$\begin{vmatrix} (X_1 - X_3)(Y_2 - Y_1) + t(X_2 - X_1)(Y_2 - Y_1) = q(X_4 - X_3)(Y_2 - Y_1) \\ (X_2 - X_1)(Y_1 - Y_3) + t(X_2 - X_1)(Y_2 - Y_1) = q(X_2 - X_1)(Y_4 - Y_3) \end{vmatrix}$$

Ако положим:

(III.7-10d) $$\begin{aligned} a &= (X_1 - X_3)(Y_2 - Y_1) - (Y_1 - Y_3)(X_2 - X_1) \\ b &= (X_4 - X_3)(Y_2 - Y_1) - (Y_4 - Y_3)(X_2 - X_1) \end{aligned}$$

От (III.7-9d) можем да елиминираме *t* и получаваме уравнението:

(III.7-11d) $a = bq$**,**

за което разглеждаме три случая. Ако $a = b = 0$ сечението съвпада с правите, които също съвпадат помежду си. Ако $a \neq 0$ и $b = 0$, сечението е празно множество. Ако $b \neq 0$ сечението е точката $(X_3 + q(X_4 - X_3), Y_3 + q(Y_4 - Y_3))$

III.7.2.2.5 Сечение на линия с окръжност

Нека са дадени линията $L(X_1, Y_1, X_2, Y_2)$ и окръжността $C(X_C, Y_C, R)$. Уравнението на правата се задава с (III.7-3). Като положим:

(III.7-12e) $$\begin{aligned} a &= Y_2 - Y_1 \\ b &= X_2 - X_1 \\ c &= X_2 Y_1 - X_1 Y_2 \end{aligned}$$

уравнението на правата може да се запише:

(III.7-13e) $ax + by + c = 0$

Ако $a = b = 0$, то сечението е празно множество.

Ако $a = 0$, но $b \neq 0$, то се преобразува в $by + c = 0$ имащо решение $y = -\frac{c}{b}$. Случаят, който се разглежда, е за хоризонтална права, за която можем лесно да определим дали пресича окръжността или не. Ако $y > Y_C + R$ или $y < Y_C - R$, сечението е празно множество. Ако $y = Y_C + R$, сечението на правата с окръжността е точката (X_C, y). При последният случай $Y_C + R \geq y \geq Y_C - R$ решенията са две. За удобство полагаме

$d = \sqrt{R^2 - (y - Y_C)^2}$, с което координатите на двете пресечни точки на правата с окръжността могат да се запишат така: $(X_C \pm d, y)$.

Третият основен случай е при $a \neq 0$. Чрез използване на уравнението на окръжността:

(III.7-14e) $\quad (x - X_C)^2 + (y - Y_C)^2 - R^2 = 0$

след умножаване на двете страни с a^2 получаваме:

(III.7-15e) $\quad (ax - aX_C)^2 + (ay - aY_C)^2 - a^2 R^2 = 0$

Чрез (III.7-13e) можем да заместим ax и да получим:

(III.7-16e) $\quad (-c - by - aX_C)^2 + (ay - aY_C)^2 - a^2 R^2 = 0,$

което се свежда до следното квадратно уравнение:

(III.7-17e) $\quad (a^2 + b^2)y^2 + (b(c + aX_C) - a^2 Y_C)y + (c + aX_C)^2 + a^2 (Y_C^2 - R^2)$

Полагаме:

(III.7-18e)
$$\begin{aligned} A &= a^2 + b^2 \neq 0 \\ B &= b(c + aX_C) - a^2 Y_C \\ C &= (c + aX_C)^2 + a^2 (Y_C^2 - R^2) \\ D^2 &= B^2 - AC \end{aligned}$$

и разглеждаме три случая. Ако $D^2 < 0$ квадратното уравнение няма решение и сечението е празно множество. Ако $D^2 = 0$ сечението на правата и окръжността е една точка, която е с координати $\left(\frac{bB - cA}{aA}, -\frac{B}{A}\right)$. Ако $D^2 > 0$ решенията са две, които съответстват на сечение - множество от двете точки $\left(\frac{b(B - D) - cA}{aA}, \frac{-B + D}{A}\right)$ и $\left(\frac{b(B + D) - cA}{aA}, \frac{-B - D}{A}\right)$.

III.7.2.2.6 Сечение на окръжност с окръжност

Нека са дадени окръжностите $A(X_A, Y_A, R_A)$ и $B(X_B, Y_B, R_B)$. Техните уравнения са:

(III.7-19f) $\quad \begin{cases} (X_A - x)^2 + (Y_A - y)^2 = R_A^2 \\ (X_B - x)^2 + (Y_B - y)^2 = R_B^2 \end{cases}$

Изваждаме второто уравнение от първото и след кратки преобразувания получаваме:

(III.7-20f) $\quad (X_A - X_B)(X_A + X_B - 2x) + (Y_A - Y_B)(Y_A + Y_B - 2y) = R_A^2 - R_B^2,$

което се свежда до уравнение на права:

(III.7-21f) $Ax + By = C$ **,**

където:

$$A = -2\left(X_A - X_B\right)$$
(III.7-22f) $$B = -2\left(Y_A - Y_B\right)$$
$$C = R_A^2 - R_B^2 - X_A^2 + X_B^2 - Y_A^2 + Y_B^2$$

Разглеждат се два случая. При първият $A = 0$ и (III.7-21f) се трансформира до

(III.7-23f) $By = C$**.**

Ако $B = C = 0$ двете окръжности съвпадат и тяхното сечение също съвпада с тях. Ако $B = 0$ и $C \neq 0$ сечението е празно множество.

Разглеждаме (III.7-23f) при $B \neq 0$ и в първото уравнение от (III.7-19f) заместваме $y = \frac{C}{B}$. Получаваме:

(III.7-24f) $$\left(X_A - x\right)^2 = \left(R_A - Y_A + \frac{C}{B}\right)\left(R_A + Y_A - \frac{C}{B}\right)$$

и решавайки го стигаме до:

(III.7-25f) $$x_{1,2} = X_A \pm \sqrt{\left(R_A - Y_A + \frac{C}{B}\right)\left(R_A + Y_A - \frac{C}{B}\right)}$$

с което се определя сечението - множество от двете точки $\left(x_1, y\right)$ и $\left(x_2, y\right)$.

Остана да разгледаме втория случай, при който $A \neq 0$. От (III.7-21f) изразяваме *x* чрез *y* по следния начин:

(III.7-26f) $$x = \frac{C}{A} - \frac{By}{A}$$

и замествайки в първото уравнение от (III.7-19f) получаваме:

(III.7-27f) $$\left(X_A - \frac{C}{A} + \frac{By}{A}\right)^2 + \left(Y_A - y\right)^2 = R_A^2$$

След известни преобразувания и следните полагания:

$$P = B^2 + A^2$$
(III.7-28f) $$Q = \left(X_A A - C\right)B - Y_A A^2$$
$$R = \left(X_A A - C\right)^2 + \left(Y_A^2 - R_A^2\right)A^2$$

се стига до квадратното уравнение:

(III.7-29f) $Py^2 + 2Qy + R = 0$

Полагаме:

(III.7-30f) $D^2 = Q^2 - PR$

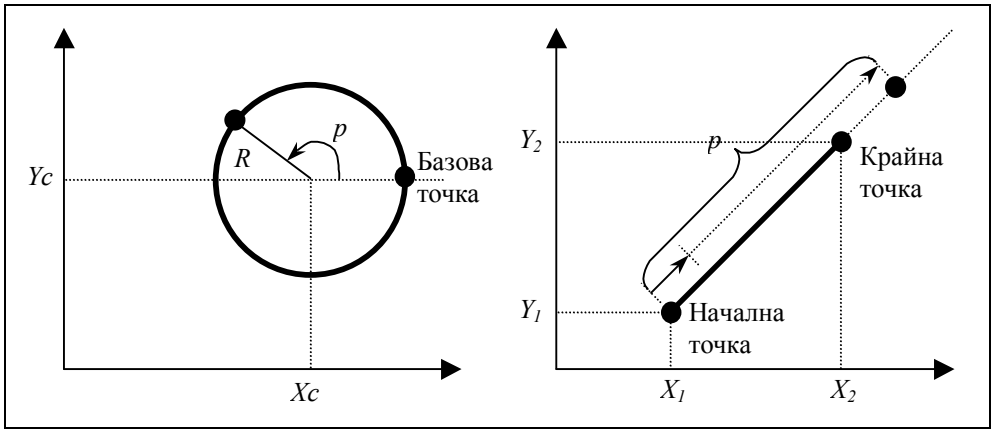
Ако $D^2 < 0$, квадратното уравнение няма решение и сечението е празно множество. Ако $D^2 = 0$, сечението на окръжностите е една точка, която е с координати $\left(\frac{CP + BQ}{AP}, -\frac{Q}{P}\right)$. Ако $D^2 > 0$, решенията са две, които съответстват на сечение - множество от двете точки $\left(\frac{CP + B(Q - D)}{AP}, \frac{-Q + D}{P}\right)$ и $\left(\frac{CP + B(Q + D)}{AP}, \frac{-Q - D}{P}\right)$.

III.7.2.3 PointOn функции

В библиотеката Geomland са реализирани три PointOn функции, с които се създава, намира или проверява положението на точка върху геометричен обект.

Най-лесно се реализира функцията PointOn? При нея проверката за положението на точка върху точка, линия, отсечка, лъч или окръжност е аналогично на начина, по който се търси сечение на точка с геометричен обект.

Фигура III-50 Реализация на PointOn#



Функцията PointOn# е реализирана само за окръжност и за линейните обекти линия, отсечка и лъч. За окръжност позицията на точката се изчислява като на ъгъла, на който е завъртяна базовата точка спрямо центъра на окръжността. За линейни обекти позицията се определя от разстоянието на точката до началната определяща точка. Ако позицията е положителна, точката е в положителната посока на линейния обект.

Функцията PointOn е реализирана за почти всички геометрични обекти: за линейните линия, отсечка и лъч, за окръжност, елипса, парабола, хипербола и за ъгъл. Изчисляването на геометричната позиция на точка според зададена числова позиция се извършва по различен начин за различните обекти. Във всички случаи се използва точковата (векторната) и ъгловата аритметика в Geomland.

Точката на позиция *p* спрямо окръжност $C(X, Y, R)$ се задава от израза:

(III.7-31g) $(X, Y) + (R, 0) \circ p$**,**

където оператора $\circ$ е дефиниран в II.8.3.1. Точката на позиция *p* спрямо окръжност отсечка, линия или лъч $L(X_1, Y_1, X_2, Y_2)$ се задава от израза:

(III.7-32g) $$(X_1,Y_1)+p\frac{\overrightarrow{(X_1,Y_1)(X_2,Y_2)}}{\left|\overrightarrow{(X_1,Y_1)(X_2,Y_2)}\right|}$$

Точката на позиция p спрямо елипса $E(X,Y,R_X,R_Y,\alpha)$, където α е ъгъла на завъртане на елипсата около центъра ѝ (X,Y), се задава от израза:

(III.7-33g) $$(X,Y)+(R_X\cos p,R_Y\sin p)\circ\alpha$$

По аналогичен начин точката на позиция p спрямо хипербола $H(X,Y,R_X,R_Y,\alpha)$, където α е ъгълът на завъртане на хиперболата около центъра ѝ (X,Y), се задава от израза:

(III.7-34g) $$(X,Y)+\left(\frac{R_X}{\cos p},R_Y\tan p\right)\circ\left(\alpha+\frac{\pi}{2}\right)$$

Точката на позиция p спрямо парабола $P(X,Y,R,\alpha)$, където α е ъгъла на завъртане на параболата около центъра ѝ (X,Y), се задава от израза:

(III.7-35g) $$(X,Y)+(p^2,Rp)\circ\alpha$$

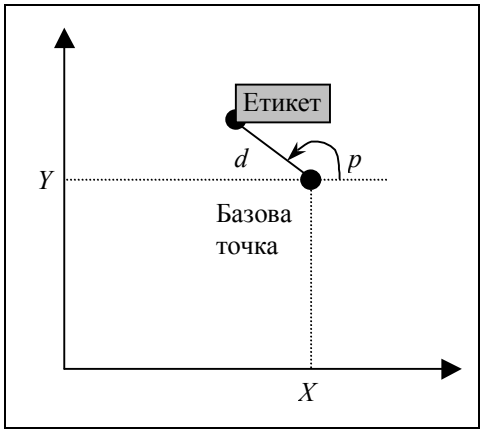
Точката спрямо ъгъл $A(X,Y,R,\alpha_1,\alpha_2)$ се задава с: $(X,Y)+(R,0)\circ(\alpha_1-p)$

III.7.2.4 Етикети на обектите

Етикетите на геометричните обекти в библиотека Geomland са независими обекти, които с правила са вързани към основните обекти. По този начин промяната на обект променя и етикета, свързан с него.

По подразбиране геометричните обектите са без етикети. Когато за обект е създаден етикет, неговите параметри могат да се променят.

Фигура III-51 Етикет на точка



Понеже етикетите са самостоятелни обекти, техните базови характеристики могат да се променят свободно. В тези характеристики се включват видът, цветът и размерът на буквите, начинът на оцветяване и дори самият текст на етикета.

Връзката с основния обект дефинира еднозначно позицията на етикета спрямо него. Етикетите имат и допълнителни характеристики, които ги доопределят. Това са позицията *p* и отклонението *d*.

За етикет на точка, долният ляв ъгъл се позиционира така, че да е на разстояние *d* от базовата точка в посока, определена от ъгъла *p*. Изразът, с който се задава координатите на етикета, е:

(III.7-36h) $(X,Y)+(d,0)\circ p$

За линейните етикети позицията се определя по друг начин. Позицията *p* определя точка, спрямо която долният ляв ъгъл на етикета се намира на разстояние *d* по нормалата на линейния обект в тази точка. Знакът на *d* определя в коя полуравнина ще е етикетът.

(III.7-37h) $(X_1,Y_1)+p\vec{n}+d\left(\vec{n}\circ\frac{\pi}{2}\right),$

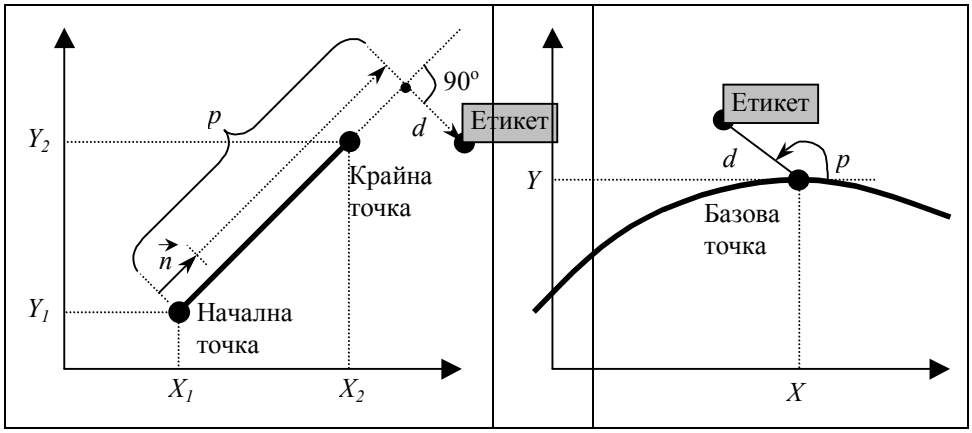
Подобно на PointOn функцията, положението на етикета се пресмята чрез използване на точкова и ъглова аритметика. Формулата, която се изчислява, използва единичния вектор върху фигурата и единичният вектор по нормалата. Нормалният вектор се получава по следния начин:

(III.7-38h) $\vec{n}=\frac{\overrightarrow{(X_1,Y_1)(X_2,Y_2)}}{\left|\overrightarrow{(X_1,Y_1)(X_2,Y_2)}\right|}$

Позиционирането на етикет на окръжност, елипса, парабола или хипербола става подобно на това при позициониране на етикет на точка. Чрез функцията PointOn се определя точка от кривата и тя се използва като базова точка, спрямо която се определя позицията на етикета.

Фигура III-52 Етикет на линия, отсечка и лъч

Фигура III-53 Етикет на крива от втора степен



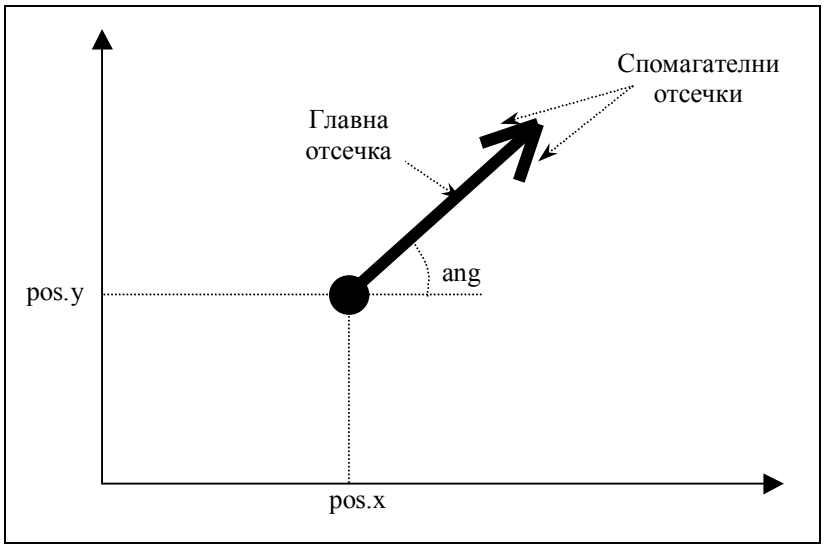
Особеност при кривите е, че позицията на етикета не зависи от нормалата към кривата в избраната базова точка. Единствената причина за това е , че пресмятанията на вектор по нормалата би изисквала повече изчисления и би забавила работата с етикети.

III.7.3 Библиотека за костенуркова графика

Библиотеката Turtle е реализирана изцяло на Elica Logo, без пряко да се използват външни библиотеки. Целта на библиотеката е да предостави костенуркова графика –

средство, което е характерно за езика Logo още от самото му създаване. Костенурката в Elica е реализирана като дефиниция на обект, т.е. потребителя може да използва неограничен брой костенурки, като всяка от тях ще е независима от останалите.

Фигура III-54 Костенуркова графика



Основата на реализацията на обекта е векторната аритметика така както е дефинирана в библиотеката Geomland. Образът (и същината) на обекта костенурка се състои от три отсечки – главна и две спомагателни. Главната отсечка определя посоката и положението на костенурката, а двете спомагателни са само за доопределяне на посоката.

Управлението на костенурката става по елементарен начин. Завъртането наляво и надясно става като се промени ъгълът `ang` и се преизчисли положението на главната и спомагателните отсечки. Движението напред и назад става като се приложи транслация с вектор, който има същата насоченост както главната посока и дължина, определена от големината на движението.

Преместването напред с `dist` стъпки се реализира с една единствена команда:

```
make local "newpos :pos + (point :dist 0) ° :ang
```

Смисълът на командата е, че новата позиция на костенурката се получава като към текущата се прибави вектор, който е с дължина `:dist` и завъртян на `:ang` градуса обратно на часовниковата стрелка. Събирането на точка и вектор в Geomland е равносилно на транслирането на точката спрямо вектора.

Цветът се контролира чрез промяна на цвета на отсечките. Следата на костенурката се променя с методите PenUp и PenDown – ако следа трябва да се остава, това става като се използва режимът `trace`, приложен към копие от главната отсечка.

Скриването на костенурката става като се изтрият отсечките. По този начин движенията на костенурката няма да оставят следа и няма да се изобразяват на екрана. Показването на костенурката създава отново тези отсечки.

ЗАКЛЮЧЕНИЕ

Разработването на система Elica не е еднократен процес. Фактът, че тя има не само академична, но и практическа стойност създават необходимостта от нейното постоянно развитие. Независимо от това, че нови версии на системата излизат почти всеки месец, задачите, които си е поставил докторантът са изпълнени. А именно: създадена е софтуерна система за образованието, чрез която да могат да се създават конкретни модули по отделните дисциплини и модели на процеси и устройства. Разработен е и конкретен модул за използване на геометрични обекти и връзки, а също и редица други модули.

Както вече бе споменато разработването на система Elica не е приключило. Напротив, то все повече се ускорява. Ползата от подобна система е забелязана от редица чуждестранни фирми и институции като Best Practices in Education, която спонсорира десетмесечен проект във връзка с усъвършенстването на Elica и реализирането на Elica 4.0 и Elica 5.0, и Stevens Institute of Technology, който ще предостави условия за разработването на бъдещите версии на системата.

Основните насоки, към които е насочена бъдещото развитие на системата е тя да се направи достъпна за потребителите на Internet. Това ще изисква разделянето на системата на сървер и на клиент и разработването на нови технологии и начини за връзка между отделните модули. Основно предизвикателство ще е симулирането в реално време на процеси, явления и механизми чрез използването на най-модерните web-технологии.

Популярността на системата тепърва ще се разраства. Представянето й на редица международни форуми относно образованието и образователния софтуер създава подходяща база за бъдещото й разпространение. В резултат на тези форуми преподаватели от България и САЩ са запознати с възможностите на системата и някои от тях вече са започнали създаването на курсове по геометрия, базирани на Elica.

АВТОРСКА СПРАВКА

Приносът на дисертанта при разработването на системата е съществен, понеже той е проектирал и създал почти цялата система самостоятелно.

Относно езика за програмиране Elica докторантът е проектирал и реализирал изцяло структурата на езика Elica Logo, начинът на унифицирано представяне на данните, пълна унификация на метаобектите, начин на компилиране на изрази с неконвенционални оператори, компилатора и транслатора на системата. По време на проектирането на езика докторантът е провел множество консултации с научния ръководител доц. Божидар Сендов за по-точно формулиране на необходимите свойства и идеологическите параметри на езика, а също и за начина на дефиниране и използване на връзките между обектите. Реализацията на ядрото на системата и на концепцията на унификацията е извършена изцяло от дисертанта.

В идеологическия предшественик на Elica – Система Планиметрия (Геомландия, PGS) дисертантът има минимално участие в разработването на някои функции за работа с костенуркова графика. Езикът за програмиране TopLogo++ е създаден заедно с екип от софтуерни специалисти. Докторантът е проектирал и реализирал управлението на паметта, компилирането на изрази и графичните библиотеки. Всички следващи системи: TGS, LGS, LGSW, RLS и Elica са реализирани самостоятелно от дисертанта.

Относно разширенията към системата докторантът е проектирал и реализирал методи на представяне и работа с двумерни и тримерни графични обекти, множество библиотеки и богат набор от прости и сложни примери. При графичните обекти приноса на докторанта е в начина на йерархичното им представяне и декомпозиране до примитиви, които могат да се рендират с OpenGL. Базовата графична система OpenGL е единствената софтуерна подсистема (ако не се брои и операционната система Windows), която не е разработена от докторанта. Докторантът самостоятелно е реализирал всички библиотеки към Elica, като сътрудничеството с научния ръководител се е отнасяло главно за билбиотеките Logo и Geomland, с цел постигането на по-пълна съвместимост със Система Планиметрия. Част от примерите, демонстриращи новите възможности на Elica са реализирани изцяло от докторанта. Другите примери, с които се показва съвместимостта със Система Планиметрия, са адаптирани.

Работна среда за работа, включваща създаване на програми, трасиране, изпълняване, проследяване на резултата и конфигуриране е проектирана и реализирана самостоятелно от докторанта, а също и помощната програма за създаване на web страници с описание на Elica.

Описанието на ядрото на системата, публикувано на сайта на Elica е написан от докторанта. Други части от описанието (главно на библиотеките) са написани от други, под ръководството на докторанта.

ИЗПОЛЗВАНА ЛИТЕРАТУРА

1. Billstein R., Libeskind Sh., Lott J. W. (1985) *Logo - MIT Logo for the Apple*, University of Montana, Missoula, Montana, 196-208
2. Blaho A., Kalas I. (1995) *Playing, Developing and Computing With Images in Comenius Logo for Windows*, EuroLogo Proceedings 95, 15-19
3. Burke M. E. (1987) *Logo and Models of Computations*, San Jose state University
4. *Cabri-géomètre: User's Manual* (1990), Laboratoire de Structures Discrètes et de Didactique, Institut d'Informatique et de Mathématiques Appliquées de Grenoble, Université Joseph Fourier, Centre National de la Recherche Scientifique
5. Hain St. (1990) *ObjectLogo for the Apple Macintosh*, Paradigm Software, Cambridge, Massachusetts
6. Jacobs J. Q. (1998) *Delphi Developer's Guide to OpenGL*, Wordware Publishing Inc, Plano, Texas
7. Nikolova I., Georgiev I. (1992) *Programming with Logo*, Publishing House "Technica", Sofia
8. *LogoWriter Reference Guide*, (1986), Logo Computer Systems Inc.
9. *Math Connections: Algebra II*, (1992), Wings for learning
10. Pratt T. W. (1975) *Programming Languages - Design and Implementation*, Department of CS, University of Texas as Austin, 342-553
11. Rogers D., F. (1985) *Procedural Elements for Computer Graphics*, McGraw-Hill Book Company
12. Sendov B, Boytcheva S (1997) *Geomland*, University of Sofia
13. Sendova E., Azalov P., Muirhead J. (1995) *Informatics in the Secondary School - Today and Tomorrow*, UNESCO International Workshop, Sofia, Bulgaria, 99-109
14. *The Geometer's Sketchpad: User Guide and Reference Manual*, (1995), Key Curriculum Press
15. *Visualizing Algebra: The Function Analyzer*, (1988), SUNBURST Communications Inc., Educaional Development Center, Newton, MA
16. Wal R. V. D. (1995) *Trees and Objects*, Eindhoven, 18-39
17. Weizenbaum J. (1993) *Computer Power and Human Reason*, Penguin Books, London, England, 132-153
18. Wolfram S. (1991), *Mathematica: A system for Doing mathematics by Computer*, Addison-Wesley Publishing Company, Inc.
19. Гелерт В., Кестнер Х., Нойбер З. (1983) *Математически енциклопедичен речник*, Държавно издателство "Наука и изкуство"
20. Ковачев М. (1995) *Графичен интерфейс на новата Microsoft Windows реализация на Система Планиметрия*, дипломна работа, СУ "св. Кл. Охридски", ФМИ, София
21. Николова И. (1986) ЛОГО – *Ръководство за програмиране*, ЦСМТА Проблемна група по образованието при БАН и МНП, София
22. *Система Планиметрия*, (1988), ComseD, Sofia
23. Станилов Г., Борисов А. (1988) *Нови срещи с коничните сечения*, Народна Просвета, София

24. Stanilov G. (2000) *Cubic sections by moving plane and applications in the finite art.* Proceedings of the "Internaional Congress C. Caratheodory", Hadronic Press, Vissa - Orestiada, Greece, 2000

25. Boytchev P. (1997) *Overview of Research Logo System*, Proceedings of 8th PEG'97 Conference, Sozopol

26. Boytchev P. (1999) *Elica Logo and Objects*, Proceedings of 7th European Logo Conference EUROLOGO'99, Sofia

27. Boytchev P. (2000) *Programming as Poetry (The Power of the Simplicity in Programming)*, Proceedings of 29th Spring Conference of the Union of Bulgarian Mathematicians, Lovetch

28. Papert S. (1980) Mindstorm. Children, Computers and Powerful Ideas, Harverster Press, BasicBooks Inc.

WEB АДРЕСИ ЗА ДОПЪЛНИТЕЛНА ИНФОРМАЦИЯ

1. ftp://anarres.cs.berkeley.edu/pub/ucblogo/
2. ftp://cher.media.mit.edu/pub/logo/
3. ftp://cher.media.mit.edu/pub/logo/FAQ
4. http://bestpraceduc.org/Technology/JennySendovaPractice.html
5. http://caesar.elte.hu/%7Eeurologo/comlogo/
6. http://caesar.elte.hu/~eurologo/lectures/hecht.htm
7. http://caesar.elte.hu/~eurologo/prog.htm
8. http://ccl.sesp.northwestern.edu/netlogo/
9. http://centres.xtec.es/logo/ponencia/darina1.htm
10. http://gise.org/JISE/Vol1-5/LOGOASAL.htm
11. http://iea.fmi.uni-sofia.bg/PGS/
12. http://indy.fmi.uni-sofia.bg/~iliana/
13. http://library.thinkquest.org/18446/eindex.shtml
14. http://logofoundation.org
15. http://members.nbci.com/iderf/
16. http://phywww1.ncssm.edu/goebel/geomland/index.html
17. http://www.angelfire.com/pa/mswlogoinfo/
18. http://www.asap.um.maine.edu/starlogo/
19. http://www.atarimagazines.com/v2n6/logo.html
20. http://www.atlantic.net/~caggiano/logo/index.html
21. http://www.atlantic.net/~caggiano/logo/index.html
22. http://www.awbruna.nl/infosl/index.htm
23. http://www.cc.gatech.edu/~jonp/6410/6410.html
24. http://www.ccl.tufts.edu/cm/
25. http://www.ccs.neu.edu/home/arthur/logo.page.html

26. http://www.cnotinfor.com.br/megalogo.html
27. http://www.cnotinfor.pt/megalogo.htm
28. http://www.cs.earlham.edu/%7Ebickiia/logo-scheme/
29. http://www.cs.pitt.edu/~bigrigg/cs1520/logo.html
30. http://www.edi.fmph.uniba.sk/logo/
31. http://www.elica.net
32. http://www.embry.com/rLogo/
33. http://www.eurologo.org/
34. http://www.eurologo.org/aduc01.html#sendov
35. http://www.idiom.com/free-compilers/LANG/Logo-1.html
36. http://www.lcsi.ca
37. http://www.lego.com
38. http://www.legomindstorms.com/
39. http://www.lgl.lu/Departements/Informatique/Logo_home/LOGOlanguage.htm
40. http://www.logo.co.jp/
41. http://www.logo.com/catalogue/titles/superlogo/index.html
42. http://www.machturtles.com/
43. http://www.media.mit.edu/starlogo/
44. http://www.messiah.edu/hpages/facstaff/barrett/logo_hlp.htm
45. http://www.nalejandria.com/fundaustral/
46. http://www.netlogo.org/
47. http://www.palmspot.com/software/detail/ps3104a_98232.html
48. http://www.ph-ludwigsburg.de/nutzer/klaudt_dieter/logo.htm
49. http://www.pitsco-legodacta.com/
50. http://www.primenet.com/pcai/New_Home_Page/ai_info/pcai_logo.html
51. http://www.siue.edu/~jandris/HTMLDocuments/ANDRIS/logow.html
52. http://www.softronix.com/
53. http://www.terrapinlogo.com
54. http://www.ugcs.caltech.edu/~dazuma/turtle/
55. http://www.vobs.at/WfS/sponsors/plt/mland/index.htm

СПИСЪК ПУБЛИКАЦИИ

на Павел Христов Бойчев

свързани с тематиката на дисертацията

1. Boytchev P., *Fast Drawing of Circles, Ellipses, Parabolas and Hyperbolas in RLS*, Technical Conference, Shoumen, 1996
2. Boytchev P., *Overview of Research Logo System*, Proceedings of 8th PEG'97 Conference, Sozopol, 1997
3. Boytchev P., *Elica Logo and Objects**, Proceedings of 7th European Logo Conference EUROLOGO'99, Sofia, 1999
4. Boytchev P., *Programming as Poetry (The Power of the Simplicity in Programming)***, Proceedings of 29th Spring Conference of the Union of Bulgarian Mathematicians, Lovetch, 2000

* *Докладът е получил наградата за "Най-добър доклад и представяне на конференцията".*

** *Докладът е по покана на СМБ.*

Други публикации, свързани с тематиката на дисертацията:

1. Бойчев П., *Питагорово дърво*, в-к ComputerNews, бр. 7, 1993
2. Бойчев П., *Стандартът IEEE за числа с плаваща запетая*, в-к ComputerNews, броеве 15, 16, 17, 18, 19, 20 и 21 , 1993
3. Бойчев П., *Проблеми при работата с числа с плаваща запетая*, в-к ComputerNews, бр. 22, 1993
4. Бойчев П., *Използване на перспектива като начин на генериране на фотореалистична графика*, ComputerNews, бр. 25, 1993
5. Бойчев П., *Алгоритми за практическото използване на три вида перспектива*, в-к ComputerNews, бр. 26, 1993

КРАТКА АВТОБИОГРАФИЯ

Павел Христов Бойчев е роден през 1969 година. Завършва гимназия с преподаване на западни езици "Р. Ролан" в гр. Стара Загора. Получава магистърска степен по компютърни науки във Факултета по математика и информатика към СУ "св. Климент Охридски". В периода на гимназиалното и висшето си обучение е участвал в множество състезания и олимпиади по информатика и математика от които е спечелил:

- 10 награди от национални и международни състезания и олимпиади
- 5 награди от национални научни конференции и олимпиади
- 3 награди от национални математически състезания и олимпиади
- 3 награди от национални музикални и литературни състезания

По време на отбиването на редовната войнишка служба работи като програмист и ДБ администратор във Военното окръжие в гр. Стара Загора и Областният център в гр. Хасково. Есента на 1990, когато започва да следва, е назначен като системен програмист в ComseD със задача разработването на бързи графични процедури за костенуркова графика. В периода от 1991 до 1993 е на полущат към Факултета по математика и информатика на СУ "св. Климент Охридски", където се занимава с разработването на Logo-базирани езици за обучение.

Става съосновател на компанията TopTeam и като такъв участва в разработването на езика за програмиране и компилатора на TopLogo++. Когато компанията започва да издава вестниците ComputerNews, AppleNews и списанието Byte Bulgaria, се занимава с предпечатна изработка, дизайн, оформяне и изработване на материали и реклами. Паралелно с това има над 60 публикация в тези издания на разнообразни теми от информационните технологии.

През 1996 постъпва на работа в отдела за софтуерен развой в централата на Обединена Българска Банка, където участва в разработването на MIS система и средства на базата на Oracle. Участва в проектирането на базовата релационна база от данни на банката, в проектирането, създаването и поддържането на универсален генератор на банкови отчети, в разработването система за кеш мониторинг и за пилотна имплементация на система за обработване на многомерна OLAP-базирана банкова информация.

В началото на 1999 постъпва на работа в отдел Oracle на фирма TechnoLogica където участва в проекти за проектирането и реализирането на информационни система за БНБ. Участва в дизайн и реализация на web-страници.

През лятото става консултант по информационни технологии към австралийската компания HIC.

Есента на 1999 преминава към фирма MPS, където ръководи разработването на подсистема към един от проектите извършван съвместно със Siemens ElectroCom, Германия.

В резултат на участия на международни форуми получава Grand Award от американската компания Best Practices in Education, който спонсорира неговата дейност относно разработването на Elica. През 2000 година получава предложение за работа като гостуващ учен към Stevens Institute of Technology, с цел усъвършенстване на система Elica.