Глава 5

СТИЛОВЕ НА ПРОГРАМИРАНЕ

Дейностите в програмирането не се подчиняват на единни водещи принципи. Стремежът към развитие, усъвършенстване и разнообразие в реализациите на програмни системи са наложили различни стилове в програмирането, които могат да се класифицират така:

• Процедурно програмиране (procedure based programming);

• Логическо програмиране (logical programming);

• Функционално програмиране (functional programming).

Процедурно ориентирано програмиране: Красноречиво пояснение на този стил представлява заглавието на книгата на Никлаус Вирт [6]

алгоритми + структури от данни = програми

Програмистът съставя програма и указва на компютъра *как* на базата на избран алгоритъм и възприети структури от данни да се намери решение на поставената задача. Типични процедурни програмни езици са Fortran, Algol, PL/1, Basic, Pascal, C/C++, Java.

Логическо програмиране: Водещият принцип з този стил гласи

правила + факти - програми

Изпълнението на една програма от логическото програмиране се свежда до *доказателство* или *извод* на определен факт, правило или твърдение въз основа на последователно обхождане на предварително изградена база данни, съдържаща множество правила и факти от определена предметна област. Алгоритмичният въпрос от процедурното програмиране как да се реши една задача тук не е централен [13]. Той се измества от въпроса *какво (каква задача)* има да се решава и как да се опише задачата, която предстои да се решава. Този стил е известен още и като декларативно програмиране. Типичен пример на език за логическо програмиране е Пролог (Prolog - Programming in logic)[45].

Функционално програмиране: Водещият принцип в този стил [29] гласи

програма = Композиция от функции

Наименованието на стила подсказава важността на обекта *функция.* Системите за функционално програмиране предоставят на програмиста богато множество от базови функции, както и средства за дефиниране на нови потребителски функции от базовите, с което се постига решаване на поставения проблем. Този стил е известен още и като апликативно програмиране. Типичен пример на език за функционално програмиране е Лисп (Lisp - List programming) [40].

5.1. ПРОЦЕДУРНО ПРОГРАМИРАНЕ

Ще повторим образеца на този програмен стил

алгоритми + структури от данни = програми

Това е определящ принцип в областта на практическото програмиране. Доминиращо количество комерсиални софтуерни продукти се създават с помощта на инструментални процедурни програмни езици и програмни среди като Object Pascal, Visual Basic, Visual C/C++, Visual Java, Borland C/C++.

Ключовите моменти в развитието на процедурното програмиране са:

1. Възможност за работа с подпрограми;

2. Появата и развитието на програмните езици от високо ниво;

3. Установяване на принципите на структурното програмиране;

4. Установяване на принципите на обектно ориентираното програмиране и неговите разновидности - компонентно програмиране, събитийно ориентирано програмиране и др.

5. Установяване на принципите на унгарската нотация.

5.1.1. РАБОТА С ПОДПРОГРАМИ

Подробности за организацията на работа, правилата за съставяне, предимствата, недостатъците и еволюцията на концепцията подпрограма са разгледани в *гл. 2. Подпрограми.*

Тук допълнително ще поясним следните две характеристики на подпрограмите:

• От гледна точка на практическото програмиране подпрограмите се развиват като синтактична категория в програмните езици, която служи за обособяване в отделно самостоятелно цяло на повтарящи се програмни фрагменти. Така общият програмен текст става по-компактен и се пести програмистки труд. По тази причина в [50] подпрограмите са наречени средства, спестяващи труд (labor saving devices);

• Не по-маловажна е и още една гледна точка относно подпрограмите. Тя се налага с развитието на информационните технологии и е ориентирана към етапите програмиране и проектиране в жизнения цикъл на програмните системи. Подпрограмите се третират като абстрактна категория. Нарича се софтуерна или процедурна абстракция (software or procedural abstraction), която се явява звено посредник между проблемната задача и компютъра. Такава постановка облекчава проектаната и му позволява да се абстрахира от детайлите на конкретната реализация. Кулминация на тази тенденция е структурното програмиране и по-нататъшната еволюция на подпрограмите като методи на класове в обектно ориентираното програмиране.

5.1.2. РАБОТА С ПРОГРАМНИ ЕЗИЦИ ОТ ВИСОКО НИВО

Предимствата при програмиране на програмни езици от високо ниво в сравнение с програмиране на асемблерен език и/или машинен език са очевидни и са извън всякакво съмнение. Известна е следната класификация за еволюцията на програмните езици [50]:

• Първо поколение програмни езици -1954-1958. FORTRAN I, ALGOL 58. Основна структурна единица за съставяне на програми е подпрограмата в своя примитивен вид без параметри. Реализираните приложения на тези езици се характеризират с относително проста структура и състав, формиран от глобални данни и примитивни подпрограми. Основен недостатък е възможността за достъп до общите данни от различни подпрограми, което крие опасност от странични ефекти и грешки при изпълнение;

• Второ поколение програмни езици -1959-1961. FORTRAN П, ALGOL 60, COBOL, Lisp. Концепцията подпрограма се разширява като абстракция за формулиране и описание на проблеми. Развиват се различни механизми за обмен на данни между подпрограми чрез параметри. Разработват се въпроси, свързани с блокова структура, влагане на подпрограми, обсег и видимост на дефиниции и декларации. Залагат се основите на структурното проектиране и програмиране на програмни системи;

• Трето поколение програмни езици - 1962-1970. PL/1, ALGOL 68, Pascal, Simula. Увеличаването на обема на програмните проекти налага разработването на отделни компоненти на програмната система да се извършва от отделни програмисти Въвежда се структурата модул (module, unit) и практиката за разделна компилация на модули (separately compiled modules). В началото модулът е механичен сбор (контейнер) от произволни данни и подпрограми. В последствие модулът включва подпрограми, обединени по определени признаци. В програмните езици липсва развита концепция за структури данни и това затруднява реализацията на междумодулните връзки;

• Съвременни програмни езици - 1970-199x. Smalltalk, Object Pascal, C++, Ada, Eiffel, Java, Perl. Развитието на структурите и типовете данни, структурното програмиране и принципите на обектно ориентираното програмиране наложиха програмни езици, за които се казва, че са обектно базирани (object based) и обектно ориентирани (object oriented). Основен градивен компонент и в тези езици е структурата модул, която се разглежда като логическа съвкупност от класове и обекти, вместо като колекция от данни и подпрограми. В [50] се цитира следната аналогия. Ако подпрограмите се смятат като глаголи, а данните като съществителни, то центърът на една процедурно ориентирана програма се строи около глаголите, докато центърът на една обектно ориентирана програма се изгражда около съществителните. Допълнително в обектните програми изобщо липсват или рядко се срещат глобални данни. Данните и операциите над тях се унифицират в класове и обекти, които по значимост изместват алгоритмите и структурите данни до такава степен, че водещият принцип на този програмен стил от алгоритми + структури данни = програми следва да се промени в образеца: *класове + об екти*
=
програми

5.1.3. СТРУКТУРНО ПРОГРАМИРАНЕ

Различни методи, техники, методики за проектиране и програмиране бяха изложени в *т. 4.2. Жизнен цикъл на програмните системи.*

Какво е структурното програмиране (СП)? То е и метод, и техника, и методика [12, 17]. Най-общо казано,
структурното програмиране
е
концепция
със значително влияние върху принципите на разработка на програмно осигуряване. СП се третира от историците като крупен принос в технологията на програмиране, равностоен по значимост с концепцията подпрограма и работата с езици от високо ниво.

Общоприето строго и ясно определение на структурно програмиране няма. Най-широко известната, но в същност тясна и повърхностна представа за СП е като програмиране без използване на оператор за безусловен преход goto. Предпоставката за тази оценка са две работи на Дийкстра. През 1965 г. на конгрес на IFIP той изнася доклад със заглавие „Programming Considered a Human Activity". В доклада се изтъква, че квалификацията на един програмист е обратнопропорционална на броя на операторите goto в неговите програми. Изложената теза на автора не привлича общественото внимание. През 1968 г. на страниците на сп. Comm of the ACM Дийкстра публикува дописка до редактора със заглавие „GOTO Statement Considered Harmful". В материала авторът явно обявява оператора за безусловен преход като вреден и развива някои идеи на низходящото проектиране/програмиране. По-точно Дийкстра посочва, че между текста на една програма и потока на нейното управление (реда на изпълнение на операторите ѝ) трябва да съществува просто съответствие. Програмата следва да бъде така структурирана, че да се чете и възприема по низходящ (top-down) принцип отгоре надолу. Неограниченото и неконтролирано използване на оператори за безусловен преход нарушава това съответствие. По тази причина структурното програмиране е известно като програмиране без оператор за безусловен преход (goto-less programming). Това е само една частична и непълна представа, тъй като винаги ще могат да се цитират програми - изключения:

1. Програми без goto, но структурирани така, че са трудни за възприемане или са напълно непонятни;

2. Програми с goto, но с достатъчно ясна логика, така че присъствието на оператор goto не пречи на тяхната четимост.

По този повод през 1974 и Д. Кнут в своята статия „Structured Programming with GOTO Statements" изтъква, че наличието или отсъствието на оператор goto само по себе си не е достатъчен показател за оценяване качествата на една програма.

Ето защо добива гражданственост следното определение за структурно програмиране по [17]: *Структурното програмиране е ориентирано към общуване на програми с хора, а не към общуване между програми и компютри.* В същия източник изискванията към една структурна програма са формулирани в 7 пункта:

1. Програмният текст се съставя от три основни елемента. Това са последователност (следване, sequence), избор на условие (разклонение, selection) и повторение (цикъл, repetition);

2. Всеки програмен модул се организира така, че да има един вход и един изход;

3. Да се избягва употребата на оператор goto навсякъде, където е възможно;

4. Програмният текст да е написан в добър и приемлив стил;

5. Програмният текст да се структурира с изместване и вмъкване на празни позиции в редовете с цел леко да изпъкват операторите от тялото на оператор за цикъл, както и ясно да се открояват групата оператори от клаузите then и else в условните оператори;

6.1 Програмният текст физически да се разбива на части така, че изпълнимите оператори от един модул да заемат обем една печатна страница;

7. Програмата сама по себе си представлява просто и ясно решение на конкретната задача.

Цитирахме пълния списък на изискванията за структурна програма по [17]. Съществено значими са първите 4 пункта, които ще обсъдим.

Точка 1: Изискването програмният текст да се съставя от елементите *последователност , избор и повторение*
следва от теоретични изследвания, които се основават на доказаната теорема от Bohm и Jacopini и публикувана през 1965 на италиански и през 1966 на английски език в статия Flow Diagrams, Turing Machines and Languages with Only Two Formulation Rules. Теоремата гласи:
Всяка реална програма може да се състави като се използват само две управляващи структури - действие и двоична проверка (разклонение).
Не е трудно да се обобщи следното.

Елементът *последователност е* верига от две или повече действия.

Елементът *избор* е познат в три варианта if... then ..., if... then ... else, switch... case... default..., но реализацията му се базира на двоична проверка.

Елементът *повторение* е известен във варианти с предусловие и с постусловие. Реализацията му се основава на управляващи структури действие и двоична проверка.

Очевидна е рекурсивната природа на структурата действие. В съдържанието на едно действие могат да бъдат включени вложени елементи и от трите вида.

Точка 2: Изискването всеки програмен модул да има един вход и един изход се свързва с понятието функционален блок, представен на фиг. 5.1. [14].

Функционалният блок има един вход и един изход. Той може да представлява отделен изчислителен оператор (дори команда на машинен език) или друга последователност от изчисления с един вход и един изход. Описаните по-горе елементи последователност, избор и повторение след обобщаваща трансформация могат да се представят като единствен функционален блок. Може да се направи следният извод. Всяка проста програма, съставена от функционални блокове (т.е. от последователност, проверка и повторение, които се трансформират във функционални блокове), се поддава на последователна трансформация до единствен функционален блок. В цитирания извод

се говори за проста програма. В [15] се дава следното определние за проста програма: *Това е програма, чиято блоксхема или управляваща структура има първо един вход и един изход и второ през всеки възел минава път (поток на управление) от входа към изхода.* Второто условие по теоретичен път определя като недопустими управляващите структури, които са проектирани да съдържат недостижими при изпълнение възли или безкрайни повторения (цикли). Това не означава, че в програмата като краен продукт няма да се получат неблагополучия като изброените, но е гаранция, че на етап проектиране те няма да се заложат като задача за програмиране. Разглеждат се още проста подпрограма, елементарна програма, неелемнетарна програма, съставна програма, структурирана програма, за които заинтересуваният читател може да намери пояснение в [15].

Точка 3: Изискването да се работи без оператор goto не бива да бъде абсолютизирано като задължително правило. То е препоръка и насока да се следи безконтролното използване на този оператор. Тук се отнасят и операторите break и continue в езиците C/C++. Те не са структурни оператори, но употребата им е оправдана и не влияе на яснотата на програмните текстове.

Точка 4: Изискването за добър и приемлив стил на програмиране също е централно в структурното програмиране. Под стил в този контекст се разбира маниерът и културата на програмиста, с които той ползва програмния език така, както един писател се цени според степента на владеене и ползване на естествения език за неговите литературни произведения. В тази насока програмистът следва да възприеме като правило в своята работа определен набор практически препоръки, бележки, насоки и принципи, спазването на които ще направи ясни и четливи за другите програмисти текстовете, които той създава. Една богата гама от такива препоръки предлагат [16, 17, 36]. Вече бе изтъкната предпоставката на структурното програмиране, че задачата на програмиста е да съставя програми, предназначени за хора, а не за компютри. По тази причина се налага доминиращото изискване за *яснота в програмата, простота, достъпност и прозрачност* на програмните текстове пред оценяваните като по-маловажни критерии за краткост или машинна ефективност. Ето някои препоръки:

1.Да се използват смислени, мнемонични имена за означаване на идентификатори, именуващи променливи, функции, класове, обекти и др. Съвременните подходи изискват интензивното ползване на мнемоника с малки и големи букви при наименованията, като голяма буква служи за начало на всяка дума, както и символ за подчертаване _ (underscore). Например BufferSize, StringPtr, ArraySize, Left_Corner_X. Тази тенденция се развива с прилагането на т.нар. *унгарска нотация,* която е пояснена по-долу;

2. Да се избягва използването на служебни думи за потребителски идентификатори дори ако езикът допуска това;

3. При кодирането на сложни изрази да се избягва въвеждането на временни променливи за съхраняване стойностите на подизрази като междинни резултати;

4. При съмнение за приоритета на операции в изрази винаги да се ползват балансирани скоби за явно означаване на подизрази, които следва да се пресмятат в ред, избран от програмиста;

5. Да се разполага по един оператор на отделен ред;

6. Да се ползва интервал за подобряване на четливостта на програмата;

7. Да не се променя явно от програмиста стойността на променлива, брояч на цикъл;

8. Да се познават синтаксисът и изразните възможности на езика;

9. Да се познават добре ситемните библиотечни функции, които се предоставят, за да се избегне излишното писане на собствени функции;

10. С цел постигане преносимост да се избягва употребата на специфични програмни възможности, които са зависими от конкретната реализация;

11. Да не се игнорират предупредителните съобщения на компилатора;

12. Излишно е да се търси ефективност чрез подобрения като замяна на операция степенуване с умножение, замяна на операции умножение/деление на 2, 4, 8 с операции за преместване наляво или надясно на 1, 2, 4 бита, вмъкване на асемблерски текст или текст на машинен език в първичния текст на език от високо ниво, търсене на най-ефективна операция за нулиране на регистър и др;

13. Да се използва фазата оптимизация на компилатора;

14. Да не се жертва четливост на програмата, за да се постигне ефективност на изпълнимия код;

15. Да се включват коментарни текстове в програмите;

16. Явно да се описват всички променливи. Да не се ползват конвенциите по подразбиране. Те са стара концепция и крият рискове от странични ефекти;

17. Да се знае разликата между дефиниция и декларация;

18. Да се избягва употребата на явни константи в програмните текстове и те да бъдат заменяни със символни константи или константни променливи, които не могат да се променят по време на изпълнение;

19. Да се избягва употребата на специфични екзотични възможности на програмните езици;

20. Да не се използва една и съща променлива за повече от една цел;

21. Да се установи и следва собствен стил в записа на програмните текстове;

22. Да не се съставят програми, които по време на изпълнение модифицират собствения си текст.

В заключение на темата структурно програмиране ще обърнем внимание, че в практиката се срещат програми, чиито реализации не отговарят на условието за структурни програми. Поставя се въпросът могат ли неструктурни програми да се трансформират в стуктурни? В общия случай отговорът е отрицателен [14]. *Произволна програма не*
може да се преобразува в структурна, реализираща същия алгоритъм, съставена от същите елементи и не използваща допълнителни променливи.
Изводът е, че привеждането на една програма в структуриран вид изисква усилия и има цена, тъй като се повтарят определени програмни конструкции, въвеждат се флагови променливи. Известни подходи за привеждане в структурен вид на програмни текстове са методът с дублиране на кодовете, приложим за програмни структури от тип мрежи и решетки, но неприложим за програми с цикли; метод с въвеждане променлива на състоянието; метод на булевия признак и др.

Идеите на структурното програмиране са заложени и вътрешноприсъщи на съвременните процедурни програмни езици от високо ниво като Pascal, C/C++, Java. В езиците са налице оператори, чрез които в различни варианти е възможно да се програмират разклонения Pascal:

IF THEN IF THEN ELSE C/C++, Java:

if() if() else и циклични повторения Pascal:

FOR = TO DO FOR = DOWNTO DO WHILE DO REPEAT UNTIL C/C++, Java:

for (;;) while () do while ( )

Наред с цитирания синтаксис на операторите за разклонение и цикъл друга предпоставка за лека реализация на идеите на структурното програмиране се свързва с концепцията съставен оператор и блокова структура на програмния език. Чрез тях програмистът групира няколко оператора в едно логическо цяло и след това към него се обръщат като към отделен оператор. Обединението на група оператори в един съставен оператор се постига чрез включване на операторите в операторки скоби от следния вид:

при Pascal: BEGIN END
при C/C++, Java: { }

5.1.4. ОБЕКТНО ОРИЕНТИРАНО ПРОГРАМИРАНЕ

При традиционното процедурно ориентирано програмиране алгоритмичният въпрос как да се реши даден проблем води до съставяне на програма, представляваща последователност от инструкции (оператори), които компютърът трябва да изпълни, за да реши поставената задача.

При обектно ориентираното програмиране (ООП) се развива концепцията за класове и обекти, при които въпросът как да се реши даден проблем се измества така, че програмистът описва обект от даден клас и определя какво да се прави (какви действия да се извършат) с този обект. Първоначално тези идеи са лансирани в ранните обектни езици Simula 67 и Smalltalk 80. Своя разцвет те получават с реализацията и масовото разпространение на комерсиалните версии на езиците C++, Object Pascal, Java, Object Oriented Lisp и други [21, 23, 35, 38].

Три са основните характеристики на ООП:

1. Пакетиране (encapsulation) на данни и подпрограми. Основната идея е да се обединят в едно цяло (unit, entity) данните и подпрограмите, които оперират върху тези данни. За целта в ООП се борави с обекти, които са представители (инстанции) на класове (потребителски дефинирани типове). Обектът може да олицетворява фигура, структура

данни (стек, масив, списък), прозорец от екрана и дори програма - приложение. Той обединява в себе си елементи данни (data elements) и подпрограми. Наличието на данни наподобява структури в C/C++ или записи в Pascal. Подпрограмите се наричат методи (methods) и служат за обработка на елементите с данни. Така се постига пакетиране или още капсулиране на данните в рамките на отделен обект. Достъпът до техните стойности не е произволен, а се канализира чрез методите, които са предназначени за тази цел. Така данните „се скриват" и достъпът до тях се регламентира посредством методите;

2. Наследяване (inheritance). Освен работа със собствени независими класове в ООП се допуска програмистът да зададе свои класове, които са породени от или наследяват предварително разработени настроени и работоспособни класове. Програмистът може да модифицира съществуващи класове, като добавя нови елементи данни и/или методи за тяхната обработка. Така той получава възможност да създава приложения, като не започва от нула. За основа служи определено множество базови класове, на които се придава нова специфична функционалност, насочена към обработките на конкретния случай. Базовите класове се запазват непроменени. Тази техника е известна като наследяване (inheritance) и позволява многократно използване на определени програмни текстове (code reusability);

3 Полиморфизъм (polymorphism). Буквалният смисъл на думата полиморфизъм изразява възможността едно нещо да съществува или да се ползва в няколко различни форми. В термините на ООП това значи, че в множество свързани класове се дефинира отделна функция (метод) с едно и също име и различно тяло (реализация). По време на компилация (compile time) не е необходимо да се знае извикваната функция към обект на кой от свързаните класове принадлежи (static binding, early binding). Едва по време на изпълнение се определя принадлежността на обекта към съответния клас и се активира методът на точно този клас. Тази техника е известна като отложено, динамично или късно свързване (dynamic binding, late binding), а функциите (методите), с които се оперира по гореописания начин, се означават като виртуални (virtual).

5.1.5. УНГАРСКА НОТАЦИЯ

Унгарската нотация е конвенция, въведнена от програмиста на Microsoft Чарлз Симоняй (американец с унгарски произход), която се посреща неоднозначно в професионалните среди. Някои автори считат принципите на тази нотация като важен момент в еволюцията на процедурното програмиране. Други автори я игнорират като несъществена. Основната идея се състои в наличието на подобрена система на

мнемоничност за именуване на идентификатори. Тя включва представка, съставена от 1 до 2 малки букви, която се разполага преди името на идентификатора. Представката посочва атрибутите на обекта, именуван с въпросния идентификатор Така за всеки идентификатор, означаващ скаларна променлива, указател, функция и други в началото на низа, съставящ името му, се включва съкратен запис на атрибутите на съответния идентификатор. Ще поясним казаното със следния пример. Нека пресмятаме функция факториел с аргумент от тип unsigned int и приемем конвенцията:

- Представката *ui* се използва за данни от тип unsigned int;

• Представката *lg* се използва за данни от тип long int. Тъй като 8! = 40320 е най-голямата стойност, която не препълва диапазона на типа unsigned int, удачно е типът на връщаната от функцията стойност да се обяви от тип long. Функцията се вика два пъти и резултатът от нейната работа се присвоява на две променливи от различен тип: int resultl и long result2. Съобразителният програмист ще се досети, че присвояването

result2=fact(arg);

е напълно коректно, докато присвояването resultl=fact(arg); крие опасност от загуба на значещи разряди на резултата при стойности по-големи от 8! поради разликата в типовете. За по-несъобразителните е предназначено присвояването

uiResult1 =lgFact(uiArg);

То е съставено по унгарската конвенция и съдържа в себе си предупреждаващите подсещащи представки пред имената на променливите и функциите. Представките напомнят, че се присвояват данни от различни типове заедно с произтичащите от това опасности за загуба на значещи разряди. Съответствието между типовете данни на пресмятащата функция и приемника на резултата явно се вижда в присвояването lgResult2=lgFact(uiArg);

Изложените съображения са обобщени в табл. 5.1.

Ефектът от прилагането на унгарската нотация е, че използваните име-на - идентификатори изглеждат странно и не са четливи. Те са трудни за произнасяне, но са пълни с информация за тяхната цел на употреба. В [16, 17] се казва, че е по-важно имената да са информативни от съдържателна гледна точка, отколкото четливи в буквален смисъл. Основната поука е, че по време на програмиране програмистът получава възможност за самоконтрол при боравене с променливи от различни типове.

В табл. 5.2 са описани типични представки, използвани при програмиране на C/C++ под Windows [9]:

Описаната методика на унгарската нотация може да бъде въведена и дисциплинирано спазвана и при работа с потребителски дефинирани типове, структури, класове и обекти.

5.2. ЛОГИЧЕСКО ПРОГРАМИРАНЕ

В своята същност логическото програмиране се основава на принципите на формалната логика и предикатното смятане. Съществен смисъл в този аспект имат понятията за предикат, клаузна форма, резолвента, метод на резолюцията [52].

Предикат (съждение, твърдение, predicate, proposition) е логическа категория, която може да бъде истина или да не бъде истина. За описание на предикатите служат обекти и отношения между тях. Обектите в предикатите се представят като прости, примитивни терми - константи и променливи. Константата е символ, който представя точно един обект. Променливата е символ, който може да представя различни обекти по различно време. Най-простите предикати, наричани атомарни предикати (atomic propositions) се състоят от единствен съставен терм (compound term). Съставният терм се представя във функционален запис и се състои от две компоненти: а/ функтор (functor), който задава име на функцията и изразява отношението между обектите; б/ списък от параметри, разделени със запетаи и обградени в скоби. Пример за примитивен предикат, изразен като съставен терм, е твърдението, че Петров е офицер.

офицер(петров)

В означения предикат участват функтор *офицер* и константа *петров.*

Два или повече атомарни предиката (съставни терми) се композират с логически връзки/операции (logical connectors, logical operators) и формират

съставни предикати (compound propositions) в съответствие с правилата за конструиране на логически изрази в процедурните програмни езици. Пет логически връзки намират приложение в предикатното смятане. Те са показани в табл. 5.3 в намаляващ приоритет заедно със знаците за означаване.

Пример за съставен предикат е твърдението, че Петров е офицер и работи в Бургас.

офицер(петров) & работи(петров,бургас)

В предикатното смятане се ползват и два специални символа, наречени квантор за общност и квантор за съществуване (табл. 5.4). С тяхна помощ в предикатите се означават и променливи. Двата квантора са с по-висок приоритет в сравнение с логическите връзки. В означенията на табл. 5.4 X има смисъла на променлива, а P - на предикат.

В предикатното смятане се допуска възможността един и същ съставен предикат да се запише в различни тъждествени форми. Известни са

преобразувания, при които например импликация и еквивалентност се изразяват с отрицание, дизюнкция и конюнкция.

С цел опростяване, унификация и постигане на еднозначност при записване на твърденията в предикатното смятане се въвежда стандартизирана форма за означаване на съставни предикати, известна като *клаузна форма,* която се представя в следния общ вид [52]:

Bl U B2 U... U Bn C A1 П A2 П...П Am, където означенията Ai и Bj са прости или съставни терми, а клаузната форма за представяне на предикати има следния смисъл. Ако всички терми Al, A2 ... Am са верни, то най-малко един от термите Bl, B2 ... Bп е верен. За клаузната форма е характерно, че се прилагат само дизюнкция, импликация и конюнкция в посочения ред: отляво последователно дизюнкция, отдясно последователно конюнкция, в средата една импликация. Дясната страна на клаузната форма се нарича антецедент (antecedent). Лявата страна на клаузната форма се нарича консеквент (consequent). Консеквентът е следствие (последица) от верността на антецедента. Доказано е, че всички твърдения в предикатното смятане могат алгоритмично да се приведат в клаузна форма [52].

Представянето на множество предикати в унифицирана клаузна форма позволява да се развие процес на извеждане (доказване) на нови твърдения (факти) от наличните предикати. Този процес е известен като метод на резолюцията. Датира от 60-те години на XX век и е дал силен тласък за развитие на направлението автоматично доказателство на теореми.

Резолюцията е правило за извод, приложимо над предикати, записани в клаузна форма. Като елементарна илюстрация са показани два предиката.

A C B1

B2 C A

Представените предикати означават, че от верността на B1 следва верността на A и че от верността на A следва верността на B2. Очевидно, от веригата предикати може да бъде направено заключението, че от верността на B1 ще следва и верността на B2, т.е. може да се формулира ново твърдение (факт), записано като предикат B2 C B1

Клаузата (новият факт), която се получава като резултат, се нарича *резолвента.*

Обобщението на горното правило по същество представлява методът на резолюцията. Според него клаузата резолвента се получава чрез обединяване на левите и десните страни на наличните клаузи, след което

общите терми в лявата и дясната част на новополучената клауза се елиминират. Следният пример илюстрира казаното в по-общ вид.

Налични клаузи:

A1 U B U A2 C C1 П C2

Dl U D2 C E1 П B П E2

След обединяване на левите и десните страни и елиминиране на общия терм B се получава клаузата резолвента в следния вид:

Al U A2 U D1 U D2 C C1 П C2 П E1 П E2

Резолюцията се усложнява при условие, че термите в клаузите съдържат променливи. В този случай еднаквостта на изключваните терми може и да не е буквална, както бе разгледано дотук. Резолюцията следва да включва определяне (замяна, субституция) на променливи с такива стойности (терми), че да се постигне успешно съпоставяне и последващо елиминиране. Този процес на замяна се нарича *унификация.*

Подробности по изложените дотук, както и по други въпроси на формалната логика и предикатното смятане могат да се видят в [13,45, 52].

5.2.1. ВЪВЕДЕНИЕ В PROLOG

В увода на тази глава бяха изложени правилата на този стил [13,45].

правила + факти = програми

изпълнение на логическа програма =доказателство

Същината на логическото програмиране е във възможността за деклариране (задаване) на факти и правила, които описват определена предметна област. Този маниер е известен като *декларативен* стил на програмиране, а Prolog е класическият пример на език за логическо или декларативно програмиране. Фактите и правилата формират, изграждат база данни. Към базата данни се задават въпроси, верността на които се проверява от системата за логическо програмиране чрез вградените средства за доказателство (извод). По тази причина тук няма програмиране с избор на алгоритъм и подходящи структури данни. Една Prolog-програма е съвкупност от данни (факти) за обекти и отношения между обекти. Изпълнението на една програма Prolog се състои в това потребителят да формулира цел (goal) - въпрос относно фактите и правилата в базата данни. Целта се задава и системата се опитва да я удовлетвори. Доколкото програмите тук не съдържат алгоритми и структури данни, специалистите по логическо програмиране не са класически програмисти. Те са специалисти по знания в определена област (knowledge engineers), които проектират и реализират практически системи за логическо програмиране. В заключение ще обобщим, че съставянето на една програма на Prolog включва задаване на факти за обекти и отношения, както и дефиниране на правила за множество факти и отношения. Изпълнението на една програма Prolog включва задаване на въпроси относно описаните факти и дефинираните правила.

5.2.2. ФАКТИ

Фактите в Prolog се представят във формализиран вид. Приема се, че те описват

(задават) отношения между обекти, които са безусловно верни в изгражданата по този начин база данни. Така се формират знания в една област [26]. Ето пример за един факт и неговия запис в термините на Prolog. *Книга1 е интересна публикация.* Обявеният факт е твърдение (предикат), което описва отношение -
интересна
и обект -
книга1
,
за който се отнася отношението. Същият факт в Prolog се нарича клауза и се записва така:

```
интересна(книга1).
```

В горния запис на Prolog думата *книга1* е обект, а думата *интересна* е отношение. Отношението е наименование, което описва обща характеристика, отнасяща се до обекти. В този смисъл отношението
интересна
може да бъде записано и за други обекти като книга2, книга3 и т.н.

Допустимо е да се обявяват отношения с повече от един обект и отношения без обекти. *Гришам е автор на книгата Фирмата*
е пример за факт с едно отношение и два обекта. Той се записва на Prolog по следния начин:

```
автор(гришам, фирмата).
```

Записът на един факт изразява знание за определено свойство или отношение между обекти. Думата преди скобите означава отношението. То именува факта (предиката). Елементите вътре в скобите се наричат аргументи и могат да бъдат обекти или променливи. Когато се записват фкти, отношенията и обектите се записват с малки букви. Променливите се записват с начална голяма буква. Отношенията се записват първо. Следват обектите, заградени в скоби и разделени със запетая. Допуска се произволен брой аргументи. Всеки факт завършва с точка.

Ще изградим база данни за книги и учебници с помощта на 9 факта. Книгите са интересни, нови и търсени, а учебниците са евтини и важни.

интересна(книга 1).

интересна(книга2).

нова(книга1).

нова(книга2).

търсена(книга1).

интересна(книга1).

интересна(книга2).

нова(книга1).

нова(книга2).

търсена(книга 1).

търсена(книга2).

евтин(учебник2).

евтин(учебник 1).

важен(учебник1).

5.2.3. ЦЕЛИ (ВЪПРОСИ)

Изпълнението на една логическа програма представлява запитване към изградената база данни. Въпросът се задава във вид на факт. Например, въпросите Важен ли е Учебник1? и Нова ли е Книга5?, се представят така.

goal: важен(учебник1)

goal: нова(книга5)

Възможни са положителни (да) и отрицателни (не) отговори. Отговор се получава след сканиране на базата данни. По време на сканирането се извършва съпоставяне и проверка за съответствие между всеки факт от базата данни и въпроса. Един факт и един въпрос съвпадат, ако:

1. Имената на отношенията (предикатите) съвпадат;

2. Броят и имената на аргументите съвпадат.

В конкретния пример резултатът от сканиране на базата данни показва, че първият въпрос съвпада с факт No 8 и отговорът е положителен, а вторият въпрос няма съвпадение и отговорът е отрицателен, goal: важен(учебник1)

true goal: нова(книга5)

false goal:

5.2.4. ПРОМЕНЛИВИ В PROLOG

В разгледаните примери се описват факти и се задават въпроси, които съдържат именуване на конкретни обекти. Има случаи, когато в Prolog не се именуват конкретни обекти. При такива ситуации се използват променливи. Те се записват с начална главна буква и биват свободни и свързани. Употребата на променливи предоставя възможност за задаване на по-интерсни въпроси:

goal: нова(XX)

XX = книга1

XX = книга2 goal:

За да се получи отговор, и тук се налага сканиране на базата данни. Провежда се съпоставяне и проверка за съответствие между въпроса и фактите. В този случай един факт и един въпрос съвпадат, ако:

1. Има съвпадение в имената на отношенията (предикатите);

2. Има съвпадение в броя и имената на аргументите или един от неидентичните аргументи е свободна променлива.

В резултат от сканиране на базата данни, името на отношението *нова* от въпроса съвпада с името на отношението от факт No 3. В този момент свободната променлива XX се обвързава с обекта
книга1
.
Казва се, че целта се удовлетворява (успява). Мястото на този факт в БД се маркира и като отговор вместо true или false на екрана се извежда XX=книга1. Сега процесът на сканиране може да приключи, ако отговорът ни задоволява, или да продължи с търсене на съвпадение с други факти в БД от точката на маркиране по-нататък. Ако търсенето се активира отново, променливата XX се освобождава за следващи обвързвания, като се прави опит за ново удовлетворяване на въпроса. В примера има съвпадение и с факт No 4 от базата данни. По тази причина на екрана се извежда и резултатът от новото свързване на променливата XX с обекта книга2.

5.2.5. КОНЮНКЦИИ И ДИЗЮНКЦИИ

Въпросите от горните примери илюстрираха проверка на едно условие. Възможно е да се задават въпроси, с които се цели проверка на повече условия. В такъв случай отделните условия във въпроса могат се свързват със знак за конюнкция - символ',' или служебна дума *and* и знак за дизюнкция - символ ';' или служебна дума *or.* Ще илюстрираме приложение на конюнкция и дизюнкция с три примера на сложни въпроси. Евтин и важен ли е учебник1? Интересна или нова или търсена е книга2? Има ли важни и евтини учебници? Записите на въпросите и отговорите на системата са следните:

goal: евтин(учебник1), важен(учебник1) true

goal: интересна(книга2) or нова(книга2) or търсена(книга2) true

goal: евтин(XX) and важен(XX) XX=учебник1

goal:

Отговорът на въпрос конюнкция е резултат от последователните отговори на подвъпросите операнди на конюнкцията, които се проверяват за удовлетворяване отляво на дясно. Отговорът е положителен, ако отговорите на всички подвъпроси успеят. В противен случай, конюнкцията пропада.

Ще анализираме отговора на третия въпрос. Първият подвъпрос, който се проверява, е: Има ли евтини учебници? - евтин(XX)? Първият факт, който съвпада, е евтин(учебник2). Мястото на съвпадение се маркира. Променливата XX се обвързва с учебник2, поради което вторият подвъпрос се модифицира от „Има ли важни учебници?" във „Важен ли е учебник2?"

Няма факт, който да съвпада с такъв въпрос. Втората цел пропада. Сега става връщане назад и се търси ново удовлетворяване (преудовлетворяване) на първия подвъпрос - евтин(XX)?. Сканирането продължава от маркираното място, като променливата XX се освобождава. Търси се съвпадение между подвъпроса евтин(XX) и факти от базата след маркираното място. Вижда се, че новото успешно съвпадение е с факта евтин(учебник1). Първата подцел е преудовлетворена. XX се обвързва този път с обекта учебник! и следва опит за удовлетворяване на втората подцел. Тя звучи така: Важен ли е учебик!?-важен(учебик1)? Сега търсенето започва от началото на БД. Този подвъпрос успява и по такъв начин след успех на двете компоненти на конюнкцията следва, че цялата конюнкция успява. Отговорът е XX=учебник1. Процесът на преудовлетворяване на подцели, който се налага при случаи като гореописания, се нарича *възврат (backtracking).* Всяка цел (подцел) има специфичен маркер за място на съвпадение в БД. Той указва точката в БД, откъдето търсенето продължава при опит за преудовлетвораване на съответната цел (подцел). Изобщо при удовлетворяване на една цел БД се сканира отначало, а при преудовлетворяване на една цел БД се сканира от точката (маркера), която е достигната при предишен опит за удовлетворяване (преудовлетворяване) на цели (подцели).

5.2.6. ПРАВИЛА

Правилата са следващият елемент след фактите, с който се изгражда БД в логическото

програмиране. Те показват, че истинността на един факт зависи от истинността на един или няколко други факта. Правилата се използват за съкратено записване на въпрос, включващ конюнкция от цели Правилата позволяват да се правят изводи от факти, които са налице в БД. Едно правило се състои от две части - заглавие и тяло. Двете части се разделят от низа ':-' и правилото има следната форма:

заглавие: - тяло.

Заглавието описва отношението, което се задава с правилото, а тялото съдържа конюнкция от въпроси (цели), които следва да се удовлетворяват, за да се приеме, че правилото е достоверно (истинно). Ще обогатим въведената БД за книгите и учебниците с две правила, които ще представят отношението *книга бестселър* и *продаваем учебник.* Нека в рамките на БД една книга се счита за бестселър, ако тя е интересна, нова и търсена, а един учебник е продаваем, ако той е евтин и важен. Съответните правила могат да бъдат записвани в следните алтернативни форми.

бестселър(XX) if интересна(XX) and нова(XX) and търсена(XX).

бестселър(XX) : - интересна(XX) , нова(XX) , търсена(XX).

продаваем(XX) if евтин(XX) and важен(XX).

продаваем(XX) : - евтин(XX) , важен(XX).

Присъствието на горните правила в БД позволява да променим формата на задаваните към базата данни въпроси.

Можем да питаме в съкратен вид „Има ли в БД книги бестселъри?" -бестселър(XX) вместо да формулираме в явен вид пълния въпрос към БД от вида „Има ли интересни,

нови и търсени книги?" -интересна(XX),нова(XX),търсена(XX)?

Можем да питаме в съкратен вид „Има ли в БД продаваеми учебници?" - продаваем(XX) вместо да формулираме в явен вид пълния въпрос към БД от вида „Има ли важни и евтини учебници?" - евтин(XX),важен(XX)?

Можем да питаме в съкратен вид „Продаваем ли е учебник1?" -продаваем(учебник!) вместо да формулираме в явен вид пълния въпрос към БД от вида „Евтин и важен ли е учебник1?" -евтин(учебник1),важен(учебник 1)?

Рекурсията намира приложение и в логическото, подобно на процедурното програмиране. Пример за рекурсия е случай, при който отношението от заглавието на правилото се използва и в тялто на правилото.

Комерсиалните версии на системи за логическо програмиране [45] на основата на езика Prolog съдържат множество вградени предикати, познаването и използването на които прави гъвкаво и ефективно програмирането на Prolog. Такива са управляващият предикат fail, задаващ цел, която винаги пропада, и управляващият предикат cut, който винаги успява и блокира механизъм за възврат отвъд точката на срещането му. Такива са предикатите за въвеждане на данни (readln, readchar, readint, readreal, inkey, keypressed), предикатите за извеждане на данни (write, writef), предикатите за работа с файлове, предикатите за работа с екрани и прозорци, предикатите за работа със символни низове, предикатите за преобразуване на типове, предикатите за работа на системно ниво.

5.3. ФУНКЦИОНАЛНО ПРОГРАМИРАНЕ

Образецът на функционалното програмиране като програмен стил е:

програма - Композиция от функции

А. П. Ершов описва особеностите на този програмен стил в предговора към [29] по следния начин. Функционалното програмиране (ФП) е специфична техника на програмиране, при която съставянето на програми е свързано само с дефиниране на функции. Единственото действие, което се извършва при изпълнение на една функционална програма, е обръщение към функции. И още: ФП е програмиране без разпределение на памет, без присвояване, без цикли, без блоксхеми и без предаване на управлението. Изброените действия се поемат от практическите реализации на системите за функционално програмиране.

5.3.1. ПРЕДВАРИТЕЛНИ ПОНЯТИЯ

Функционалното програмиране е програмиране с функции. Функцията е фундаментално математическо понятие. Това е правило, по което на елемент от едно множество се съпоставя елемент от друго множество. Функцията е отношение (релация) f(x) = y. Функцията е изображение f:X -> Y. Съществуват различни начини за описание на функции - чрез изброяване на съответни двойки, таблично описание, чрез запис във вид на равенства, описание с определящо правило или определение (дефиниция).

Основните принципи, на които се основава ФП и които трябва да осигурява една система за функционално програмиране, са следните:

1. Наличие на достатъчно мощно и богато множество базови примитивни функции;

2. Наличие на възможност за дефиниране на нови функции от наличните базови функции;

3. Възможност за извикване на функции и изпълнение на активираните функции.

Условие 1 е напълно естествено. За да бъде богат един език, той трябва да разполага с достатъчен репертоар от изразни средства в дадена предметна област. Допълнителни изисквания, които се налагат над множеството базови функции, са:

1. Те да са с високо ефективна реализация в изчислителната среда на даден компютър;

2. Да се допуска леко формулиране на различни проблеми в термините на базовите функции [29].

Например нормално е за всяка предметна област да се извършват аритметични операции, т.е. сред базовите функции трябва да има възможност за задаване и изпълнение на аритметични операции. За целта двуместните операции събиране, изваждане, умножение, деление могат се третират като функции с два аргумента, които връщат една стойност.

add(p 1 ,p2)	е равностойно на	p 1 +p2
sub(p 1 ,p2)	е равностойно на	p 1 -p2
mul(pl,p2)	е равностойно на	pl*p2
div(pl,p2)	е равностойно на	((p20) -> pl/p2)

Наличието на тези функции позволява произволен аритметичен израз да се запише във функционална нотация като следния пример:

a*b + c/d	е равностойно на	add(mul(a,b),div(a,d))

Представянето на изрази във функционален запис се нарича още апликативна структура на изрази в програмните езици [29, 39]. Един израз, записан по такъв начин,

се разбива на съставни компоненти, които са или операция, или операнд. Операндът задава стойност, а операцията указва функция. Апликативната структура на израза се изразява в това, че той се състои от операции, които се прилагат над операнди. Език, в който описаната структура на апликативните изрази се запазва за всички изрази, които могат да се изразят в него, се нарича *апликативен* или *строго функционален* език [29].

Условие 2 е свързано с изискването да се създаде възможност за определяне на нови функции от стари известни и съществуващи функции при конструиране на по-сложни. Този подход, метод или принцип е известен като *композиция на функции.* Следният пример служи като илюстрация. Нека сред базовото множество функции има функция, която определя по-малката от две стойности - min(pl,p2). Целим да съставим функция least(pl,p2,p3), която определя най-малката от три стойности като се ползва базовата функция min(). Решението е очевидно.

least(pl,p2,p3) = min(pl,min(p2,p3))

То илюстрира принципа композиция на функции, т.е. построяване на функции от стари известни функции. Вмъкването на обръщение към функцията min() на мястото на аргумент в друго извикване на същата функция min() изгражда композиция на функцията min() със себе си. Така се получава нова функция [29]. Дълбочината на влагане е без ограничение, с което се създава възможност за създаване на произволни композиции. Например функция, определяща най-малката от 6 стойности, може да бъде конструирана по много начини:

min(min(p 1 ,p2),min(min(p3,p4),min(p5,p6)))

min(least(p 1 ,p2,p3),least(p4,p5,p6))

least(min(pl,p2),min(p3,p4),min(p5,p6))

Условие 3 се отнася до извикването и изпълнението на функции в реалните системи за функционално програмиране. ФП се свързва с езика Lisp (LISt Programming), считан

като най-представителен език за функционално програмиране. Затова изложението по тази точка ще бъде илюстрирано в термините на този език, по-специално диалекта SCHEME Lisp [1]. Без да разглеждаме синтаксиса на Lisp (атоми, S-изрази, базово множество функции) ще спрем вниманието на читателя върху структурата *списък.* В Lisp тя е основна и служи за задаване както на програми (композиции от функции), така и на данни. Неформално, списъкът е конструкт, който се формира като отделни обекти се заградят в скоби (). Обектите, от които се строят списъци, са или отново списъци, или представители на основната неделима единица в езика - атома. Изразите, записани във функционална нотация при работа на Lisp, могат да бъдат представени още и чрез списъчна нотация, както е показано в табл. 5.5.

5.3.2. ОЦЕНКА НА КОМБИНАЦИИ

Изрази, записани в списъчен вид, се наричат комбинации. Една комбинация съдържа най-ляв елемент- операция, следвана от операнди. Този запис е известен още и като префиксна нотация. При нея първо се записва операцията, а след това и операндите. Префиксната нотация допуска задаване на произволен брой аргументи и влагане на изрази в същата префиксна нотация на мястото на операндите. Действието пресмятане на изрази във ФП се нарича оценка на комбинации. Всяка система за функционално програмиране (в случая наричана машина Lisp) има вградена система за оценка на комбинации. Алгоритъмът на работа е на интерпретативен принцип и изпълнява следния основен цикъл от действия:

Прочети!

Оцени!

Отпечатай!

Процесът започва с въвеждане на израз. Машината Lisp чете винаги изрази, записани в списъчна нотация с изключение на случаите, когато на вход се подава израз в най-прост вид - стойност или променлива.

Изразът се пресмята след въвеждането му.

За целта се активира процедурата оценка на комбинации.

Стойността на израза се извежда безусловно в третата стъпка, без да е необходимо явно издаване на команда за печат.

Следват примери за диалог с машина Lisp, която интерпретира (чете, оценява и отпечатва) изрази по описания алгоритъм.

На входа се подава израз - стойност. Оценката на такава комбинация е самата стойност и интерпретаторът я извежда.

=>312

312

=>76

76

=>34

34

На входа се подава комбинация - израз в списъчен вид, който включва операция, следвана от операнди. Стойността на комбинацията (нейната оценка) се получава като се приложи функцията, определена от операцията с аргументи, дефинирани от фактическите данни (операндите). По-точно [1] първо рекурсивно се оценяват подизразите на комбинацията и второ прилага се процедурата, която е зададена като най-ляв подизраз (операция) с аргументи, които са стойности на останалите подизрази. Резултатът от пресмятането на израза се извежда.

```
=>(+ 22 44 33 11)

110

=>(- 100 34)

66

=>(* 15 4)

60

=>(/ 200 5)

40

=>(+ (* 3 5) (- 10 6))
```

19

Преди да илюстрираме работа на машина Lisp с по-сложни обекти, следва да се поясни понятието *именуване в изчислителна среда.* Имат се предвид средствата, които програмните езици предоставят за описание на изчислителни обекти посредством имена - идентификатори. Казва се, че *имет о идентифицира променлива, чиято стойност е самият обект* [1].

В процедурните програмни езици средствата за именуване в изчислителна среда са операторите за дефиниране, деклариране и инициализация на променливи с определени стойности.

Във функционалните програмни езици средствата за именуване в изчислителна среда са функции, чието обръщение в списъчен запис изглежда по следния начин. Използва се конкретно име на функция *define.* Тя е валидна за SCHEME Lisp. В други версии на Lisp за същите цели се ползват функции SETQ, DEFUN и др.

=>(define obem 200)

obem

Интерпретаторът чете и оценява израза (define obem 200), като извежда за стойност на израза името obem. Казва се, че в резултат от оценката на израза машината Lisp асоциира стойността 200 с името obem. Напомняме, че в Lisp всеки израз има стойност. В списъчния израз (+358) най-естествено е стойността на израза да бъде сумата на операндите 3, 5, 8. В списъчния израз (define obem 200) не е очевиден признакът за определяне стойността на израза. В такъв случай стойността се определя по уговорка, което силно зависи от конкретната реализация. Така се създава контекст [1], който е полезен за позоваване при по-нататъшната работа.

=>(define length 20)

length

=>(define width 30)

width

=>(* length width)

600

=>(define-area (* length width))

area

=>area

600

Оценяването на комбинации, в които участва define, се различава от разгледаното досега оценяване на комбинации, което се свежда до пресмятане на изрази в префиксна нотация. Безпредметно и невъзможно е да се търси аналогична интерпретация (пресмятане) на изрази като (define xx 33). Изрази от този вид имат друга цел. Предназначението на define е да асоциира името xx със стойността 33. Този процес се нарича *свързване* [1]. Подобни изключения от общото правило за оценяване се наричат *специални форми.* Функцията define е специална форма, а всяка специална форма има свое собствено правило за оценяване на комбинации.

Правилото за оценяване на специалната форма define гласи, че стойността на израза с define е името на първия аргумент. Затова интерпретаторът оценява израза (define xx 33) и извежда името xx.

```
=>(define xx 33)

xx
```

Казаното тук пояснява диалога с машината Lisp, представен по-горе за именуване на имената length, width и area.

Разгледаното асоцииране (свързване) на имена със стойности е примитивна и ограничена възможност за абстракция. По-мощно средство за абстракция е възможността за дефиниране на функции. Тя позволява именуване на сложна операция и обработка на формални параметри. Общият формат за дефиниране на функция в списъчен запис е:

```
(define ( ) )
```

Следват конкретни примери за дефиниране на функции за определяне площ и обиколка на правоъгълник.

```
=>(define (rectarea x y) (* x y))

rectarea

=>(define (rectlen x y) (+ (* 2 x) (* 2 y)))
```

rectlen

Обръщението към новодефинираните функции се извършва в традиционен списъчен запис.

=>(rectarea 20 30)

600

=>(rectlen 20 30)

100

Примерите дотук илюстрират дефиниране на функции, при изпълнението на които се извършва само линейна последователност от действия. За изразяване на по-сложни действия, в машината Lisp е предвидена възможност за запис на условни изрази в следния списъчен вид:

(if)

Оценката на предиката определя следващите действия на интерпретатора. Ако предикатът има стойност истина, машината Lisp оценява и връща стойността на следствието. В противен случай, когато предикатът върне стойност лъжа, машината Lisp оценява и връща стойността на алтернативата. Ще дефинираме следната функция:

0, n = 0

F(n) = F(n/2), n - четна стойност

1 + F(n-l), n - нечетна стойност

За да опишем функцията F(n), ще формулираме предикат за определяне на четна или нечетна стойност. Ползва се примитивна функция за пресмятане на остатък от деление или дефинирана от нас функция remainder за намиране остатък от деление по формулата m mod n = m - m/n*n. (define (remainder m n) (- m (* (/ m n) n))

)

(define (F n)

(if(=n0) 0

(if (= (remainder n 2) 0) (F(/n2)) (+l(F(-nl))))))

Условни изрази могат да бъдат записани по два други начина. Те се базират на записа на условен израз по Макарти. Като специални символи служат cond и else [1]. (cond ( ) ()

()

(**))**

Двойките изрази ( ) са аргументи и се наричат клаузи. Ако

някой от предикатните изрази е истина, тогава cond връща стойността на съответния израз . Ако никой от предикатните изрази не е верен, тогава cond връща . (cond ( ) ( )

( ) (else))

В новия вариант специалният символ else се използва вместо предикатния израз . При тази постановка, ако никой от изразите не е верен, cond връща стойността на израза . Ще приложим втората версия на записа за условен израз cond, за да предефинираме функцията F(n):

```
(define (F n)

(cond ( (= n 0)                                                  0)

(  (= (remainder n 2) 0)  (F (/ n 2)))
 ( else                    (+ 1 (F (- n 1))))
 )
 )
```

В Lisp се поддържа възможност за безименно дефиниране на функции с помощта на т.нар. ламбда-нотация. Вместо дефиницията (define (decr x) (-x 1)) може да се запише израз в ламбда-запис (lambda (x) (- x 1))

Една функция може да се дефинира по два начина: чрез define и с ламбда-нотация. Асоциирането на името decr с тялото на функцията за намаляване на стойност с единица може да се зададе със следните два еквивалентни записа

[1].

```
(define (decr x) (-x 1))
```

```
(define decr (lambda (x) (- x 1)))
```

Обръщението към функция decr (за намаляване на стойност с единица) може да се укаже по два различни начина в списъчен запис:

```
=>(decr 50)
```

49 '

```
=>( (lambda (x) (- x 1)) 40)
```

39

5.3.3. ОСНОВНИ ЕЛЕМЕНТИ НА LISP

Lisp е език за списъчно програмиране и е естествено да разполага с примитивни функции за обработка на списъци. Ще поясним последователно елементите на езика *атом, S-израз, списък, примитивни функции* [40].

Атом. Атомът е основна неделима единица в езика.

S-израз. Означението S-израз е съкращение от symbolic expression. Рекурсивната дефиниция на S-изрази се определят така [8]:

1. Всеки атом е S-израз. ::= атом

2. Ако A и B са S-изрази, то и записът (A . B) е S-израз. ::= (.)

3. S-изрази се задават само по правила 1. и 2.

Записът (A. B) се нарича точкова двойка. Дадените определения за S-израз са рекурсивни, т.е. може да се обобщи, че всеки S-израз е или атом, или точкова двойка.

Списък. Това е специален S-израз, за който се отнася следното определение. Нека A1, A2, A3,..., AN са S-изрази. Тогава S-изразът, записан като

(A1 .(A2.(A3. (...(AN. NIL)...))))

се нарича списък и съкратено се записва като

(Al A2 A3 ... AN)

NIL има смисъл на специален атом, който означава край на списък.

В [2] се дава следното словесно рекурсивно определение на списък: *Списъкът е или празен, или се състои от начало, което е атом или списък и от остатък, който е списък.*

Примитивни функции *cons, car, cdr, atom, null.*

Функция cons. Има два аргумента и служи за създаване на точкова двойка. Аргументите могат да бъдат атоми и/или S-изрази.

(cons A B) ще създаде (A . B)

Извикване: (cons A B)
(cons (A . B) C) ще създаде ((A . B). C)

Извикване: (cons (cons A B) C)

(cons (A. B) (C . D)) ще създаде ((A.B).(C.D))

Извикване: (cons (cons A B) (cons C D))

Функция car. Има един аргумент от тип S-израз. Връща началото (главата, първата част) на своя аргумент.

(car (A . B)) ще върне A

Извикване: (car (cons A B))

(car (A. (C . D))) **ще върне** **A**

Извикване: (car (cons A (cons C D)))

(car ((A . B). C)) **ще върне** **(A . B)**

Извикване: (car (cons (cons A B) C))

Функция cdr. Има един аргумент от тип S-израз. Връща остатъка (опашката, втората част) на своя аргумент.

(cdr (A . B)) **ще върне** **B**

Извикване: (cdr (cons A B))

(cdr (A . (C . D))) **ще върне** **(C . D)**

Извикване: (cdr (cons A (cons C D)))

(cdr ((A . B). C)) **ще върне** **C**

Извикване: (cdr (cons (cons A B) C))

Функция atom. Функция - предикат. Има 1 аргумент. Връща истина, ако аргументът е атом.

Функция null. Функция - предикат. Има 1 аргумент. Връща истина, ако аргументът е празен списък. Използва се при обхождане елементите на списък и за формулиране на условие за изчерпване елементите на списъка. Като пример за дефиниране на нови функции за работа със списъци от описаните примитиви ще представим дефиницията на функция, определяща дължината на списък. Името на функцията е length, а аргументът е x. (define

(length x) (if (null x) O

(+ 1 (length (cdr x)))))