БЪЛГАРСКА АКАДЕМИЯ НА НАУКИТЕ
Институт по математика и информатика

Бойко Блажев Банчев

Проблеми на изразната явност в текстове на програми

ДИСЕРТАЦИЯ

за присъждане на образователна и научна степен

„ДОКТОР“

област на висше образование 4 Природни науки, математика и информатика;
професионално направление 4.6 Информатика и компютърни науки;
научна специалност 01.01.12 Информатика

Научен консултант

проф. д-р Нели Манева

София, 2014 г.

С настоящото декларирам, че представеното тук изследване е извършено от мен самостоятелно и всички резултати, за които не е посочен друг източник, са получени от мен.

Бойко Блажев Банчев

Подпис:

Общоприета истина е, че езикът е не само среда за представяне на мислите, а и инструмент за разсъждаване.

Дж. Бул

Краткото изразяване чрез алгебрични означения на голямо количество смисъл ... усилва способността ни за онзи тип разсъждения, който сме свикнали да извършваме с тяхна помощ.

Ч. Бабидж

Средствата, които използваме, влияят по дълбок (и прикрит!) начин върху мисловните ни навици, а следователно и върху мисловните ни способности.

Е. Дейкстра

Програмите трябва да се пишат за четене от хора, а едва след това – и за изпълняване от машина.

Х. Абелсън, Дж. Съсман

Съдържание

Увод

Предмет на изследване в настоящата работа, общо казано, са свойства на езиците за програмиране. По-конкретно, разгледани са свойства на текстове на програми от гледна точка на непосредственото – в противовес на едромащабното – програмиране. Някои от тях са обусловени пряко от съответния език за програмиране, а други се дължат на начина на използването му. Като общ знаменател на разглежданията е избрано формулираното тук качество явност.

Най-общо казано, явността, заедно с няколко други понятия, показва доколко изразните средства на даден език за програмиране са годни да представят желателни за програмиста свойства на и отношения между програмни обекти. Отнесена към конкретна програма на даден език, явността характеризира степента на изразяване на такива свойства и отношения предвид намеренията на програмиста или адекватното решаване на задачата.

Целта на представената работа е именно да подложи на анализ, с оглед на годността им за явно и адекватно представяне на идеи, някои свойствени на езиците за програмиране структури.

За постигането на тази цел намерихме за необходимо да изпълним следното:

- да уточним някои важни характеристики на текстове на програми, имащи отношение към тяхната съдържателност или към възприемането ѝ;
- сред тях да анализираме по-подробно явността и понятия, които се оказват близко свързани с нея;

- тъй като в частта на горните две задачи разглеждането по необходимост е доста общо, подсказващо по-широк контекст, отколкото този само на езиците за програмиране, намерихме за нужно преди всичко, като начална стъпка, да изясним в известна степен как тези езици се съотнасят с естествените и с този (или тези) на математиката;
- да разгледаме предвид способността им за явно изразяване някои от най-важните видове структури и конкретни конструкции в езиците за програмиране;
- на основата на направения в рамките на горната задача анализ да предложим нови езикови решения, допълващи или заместващи традиционните, които да подобрят разглежданите в темата характеристики на програми;
- като обобщение на представата за структурност в програми да потърсим общ понятиен подход за описване на структурност в системи от информатично естество.

Изследването е извършено с прекъсвания в много продължителен период от време и представя сравнително разнородни резултати, както може да се проследи по съдържанието му. Не са търсени окончателност и завършеност, а и подобна цел едва ли е постижима – според нас темата не предполага достигане на някакъв общ и краен резултат, а позволява проучвания в различни посоки, под различни ъгли и с разнообразни изводи. В този смисъл изложеното тук съдържа възгледи и други резултати от работата на автора по темата без претенция за изчерпателност.

Постигнатите в работата резултати са отчасти с аналитичен и отчасти с „изобретателски“ характер. Под второто се има предвид предлагане на езикови конструкции за решаване на различни свързани с темата конкретни задачи. Разбира се, аналитичният и „изобретателският“ елементи са тясно преплетени и единият предпоставя

другия.

Текстът на работата е построен по следния начин.

Първите три глави са уводни в темата. Тяхната задача е да я поставят в подходящ контекст, да въведат съществени за съдържанието на програмен текст и за възприемането му понятия и да дадат известна представа за духа на по-нататъшните разглеждания.

В глава 1 се прави съпоставяне на езиците за програмиране от една страна с естествените езици, а от друга – с този на математиката, като целта е да се изявят най-съществените сходства, различия и донякъде взаимни влияния между тези три вида средства за изразяване. На този фон може по-успешно да се открои спецификата на езиците за програмиране и впоследствие да се насочи вниманието към съществени за темата черти на текстовете на такива езици.

Такива черти се разглеждат в глава 2, където се прави аналитичен обзор на редица съдържателно и от гледна точка на възприемането на програмен текст важни фактори.

Глава 3 задълбочава започнатото в глава 2 изследване, като го съсредоточава около понятието явност и други близки до него.

Глава 4 разглежда различни видове конструкции в езиците за програмиране, като предлага редица усъвършенствания и нови решения, способстващи за по-адекватно, явно и нагледно изразяване на идеи чрез езика.

В глава 5 се дава по-обща перспектива върху строежа на програми и по-общо – върху структурата на системи с информатична природа. Разглежданията в тази глава имат за цел да поставят основа за оформяне на понятийна рамка за представяне на структурност – рамка, независеща от особеностите на конкретни езици за програмиране и изчислителни парадигми. В този план понятието структура е възможно най-абстрактно, оставайки достатъчно конкретно отнесено именно към програми и текстовете, които ги представят.

1. За говоримите езици, този на математиката и езиците за програмиране

Езиците за програмиране – обект на настоящото изследване – са преди всичко езици изобщо. Затова смятаме за уместно, като прелюдия към същината на темата и за да добием по-широк поглед върху нея, да потърсим паралели между езиците за програмиране и други видове езици. Ще направим кратък анализ на сходствата и отликите между езиците за програмиране (ЕП) от една страна и говоримите езици (ГЕ) и езика на математиката от друга.

1.1. Говоримите езици и езиците за програмиране

Преди всичко следва да отбележим, че сходства и аналогии между ЕП, ГЕ и езика на математиката са търсени и забелязвани неведнъж. Нещо повече, в ЕП целенасочено се влагат заемки както от говоримите езици, така и от математическия символен стил на изразяване. Като добре известен пример за заимстване от говорим език да посочим COBOL, на който отделните конструкции се изразяват чрез къси изречения на ограничен английски. От друга страна, записването на аритметични и други подобни действия в повечето ЕП става по начин, близък до обичайния в математиката. Това отчетливо се наблюдава още при FORTRAN, а отчасти е налице и в по-ранни (предезикови) системи за програмиране.

Всъщност начинът, по който COBOL се стреми да бъде „естес-

твен“, е повърхностен и отдавна осъзнат като нерационален. Две от причините за последното са, че е твърде словесен – програмите изобилстват с английска лексика и са многословни – и че се допуска объркващо разнообразие от форми за изразяване на повечето действия, така че програмистът изпитва несигурност заради това, че трябва да избира между тях. В по-късно създаваните ЕП този подход на COBOL е изоставен, но същевременно се търси заимстване на други, по-съществени за нуждите на програмирането черти и елементи на говоримите езици.

Най-съществената отлика на естествените езици от ЕП е в това, че всеки естествен език е жив организъм, постоянно и плавно самовъзникващ в обществената практика в съответствие с нейните потребности, докато ЕП се създават целенасочено и макар някои от тях също да търпят изменения, те стават на тласъци през известни периоди от време и в повечето случаи не са големи. И при говоримите езици, и при ЕП е налице взаимно влияние, като то е по-силно изразено при езиците за програмиране.

Интересно съпоставяне между говоримите езици и ЕП прави P. Naur [65], чието име е добре известно във връзка с последните. На първо място той обръща внимание на следната асиметрия между двата вида езици. В говоримия език се изказват и възприемат мисли с *размит*, а не точен смисъл и това не е недостатък, а жизненоважно свойство, благодарение на което езикът се развива, добивайки възможност да изразява все нови и нови идеи. Благодарение на това и всеки ЕП е продължение на естествения език – пример за способността му да се саморазширява. В този смисъл ЕП имат подчинена, вторична роля по отношение на говоримите.

Наред с асиметрията се наблюдават и редица сходства, например от прагматично естество. Далеч преди появата на езици за програмиране, видни лингвисти правят сравнителна оценка на *зрелостта*

на говоримите езици от гледна точка на (цитираме отново [65]):

- краткост на изразяването;
- неизлишъчност – неповтаряне на информацията, например за род, число и др. на даден обект, в различните елементи на дадено изречение;
- еднообразие на формообразуването;
- малък брой отклонения от преобладаващите синтактични схеми (особени случаи на синтаксиса);
- възможност за получаване на сложни езикови форми чрез комбиниране на прости и то при различни начини на комбиниране (на всички или на голям брой от различните комбинации на дадени елементи да се приписва смисъл);
- регулярност на словореда и др.

Не е трудно да се забележи, че тези качества се ценят и по отношение на ЕП и обичайно се цитират като желателни достойнства на такива езици в статии и монографии, засягащи проектиране и оценяване на ЕП [42, 85, 35, 81].

Могат да се посочат редица сходства между *елементите* на ГЕ и ЕП:

- глагол/сказуемо в ГЕ и действие (операция, команда) в ЕП;
- изречение в ГЕ и израз или команда в ЕП;
- вмъкнати или подчинени изречения в ГЕ и локални контексти (скоби, блокове и др. под.) в ЕП;
- абзац в писмения говорим език и програмен фрагмент в ЕП и др.

На по-високо езиково равнище, понятието *стил* отнасяме както към употребата на говоримия език, така и, със сходно значение, към програмирането ([47] и др.)

Своеобразен подход при заимстване от ГЕ в ЕП предложи K. Iverson, който, развивайки апликативния стил на създадения от него

още в началото на 60-те г. език APL, въвежда в по-съвременното му издание [44] и по-късно в езика J термините „глагол“, „наречие“, „съюз“ (различавайки няколко вида апликативни действия, нарични в по-раншната терминология общо *оператори*), „съществително“, „местоимение“ и др., употребявани в говоримите езици. Подобна терминология се използва и в езика K, по-късно преименуван на q [21].

Любопитна е и аналогията в [65] между изкуствено създаваните говорими езици, Interlingua, Esperanto, Ido и Novial, и ЕП по повод на това, че и едните, и другите, като изкуствено създавани, се утвърждават или не повече под влиянието на конюнктурни фактори – поддържащите езиците икономически, политически или обществени сили – отколкото поради достойнствата си. Изминалите близо 40 години многократно потвърдиха това наблюдение.

Връзката между естествения език и езиците за програмиране е много по-дълбока и многостранна, отколкото е възможно и уместно да описваме тук. Естествените езици са основен носител на човешката култура и, допълвайки смисъла на първото мото на тази работа – средство за развитието ѝ. Езиците за програмиране, като специфично продължение на говоримите, както и програмирането въобще, се проникват от тази култура, са нейни елементи и частично я създават. Следното мнение на E. W. Dijkstra е красноречиво косвено свидетелство за връзката между мисленето чрез говоримия език и програмирането:

> *„Освен да има математически наклонности, другото жизненоважно за способния програмист качество е изключително доброто владеене на родния език“* [29].

Заимстването в ЕП на свойства от ГЕ (както и от обсъждания по-долу език на математиката) е необходимо и ползотворно, но не може да става механично и няма правила, които диктуват как да

става това. По-нататък в текста се съдържат, без непременно да бъдат изрично посочени, различни конкретни примери за такова заимстване.

1.2. Математиката и езиците за програмиране

Търсейки сходства и различия между писмеността на математиката и програмирането, трябва да си даваме сметка и за сходствата и различията между предметите на едното и другото.

Това, което най-много сближава информатиката с математиката и отличава тях двете от останалите научни дисциплини е, че и двете науки „работят“ на много високо равнище на абстрактност и третират пределно общи и основни явления и зависимости в природата. Това е причина редица специалисти да смятат информатиката, във всеки случай онази част от нея, която пряко или косвено има отношение към програмирането, за специфичен дял на приложната дискретна математика.

Наистина, информатиката е в корените си свързана с математиката. Изследвайки математиката от различни времена и култури още от древността навсякъде се забелязват елементи на алгоритмичност. Например в [50] се посочва, че вавилонската математика от близо 2000 г. пр. н. е. съдържа образци на алгоритмично мислене. Във вавилонските математически текстове прозира и аналог на понятието променлива (и то по-близък до смисъла на този термин в императивните езици за програмиране, отколкото до смисъла на същото понятие в математиката!): дадена стойност може да бъде запомнена (копирана), така че едното копие остава неизменено за по-късна употреба, а другото се променя чрез добавяне, отнемане, умножаване или делене с различни количества. Ясно открояващи се елементи на алгоритмичност има и в математиката на древните египтяни, гърци и др.

Изключителната способност на математиката да отразява света се дължи на богатството на конструктивните възможности, заложени дори в много проста аксиоматична система. При това, въпреки че, формално погледнато, всяко изведено от аксиомите твърдение прави известно само това, което така или иначе се съдържа в тях, развитието на математическите теории е свързано с обективен прираст на информация от психологическа, евристична и прагматична гледна точка. При всяка отделна дисциплина това става за сметка на въвеждането на нови определения и връзки между тях, нарастващи, изменящи и обогатяващи структурата на дисциплината като дял от знанието (вж. напр. [5]).

Тези две свойства – способността за универсално отразяване на природата и за съдържателно самообогатяване – са присъщи и на информатиката, и то не само в гносеологичен план, а и в непосредствено прагматичен, във връзка с конструкциите, които информатиката създава в своята практика. Конкретно в програмирането, съставянето на модел на обект или процес във формата на програма представлява именно натрупване на определения и връзки на взаимодействие между тях. На свой ред понятията и структурите, фиксирани в езиците за програмиране, се обособяват под прякото или косвено въздействие на устойчиво появяващите се създадени ad hoc понятия и структури в практиката на програмирането.

Общо за математиката и информатиката се оказва и разделянето на „чиста“ и приложна наука. От една страна, програмирането се разглежда, в работите на D. Knuth, E. Dijkstra, C. Hoare, N. Wirth, D. Gries и др., като „интелектуално предизвикателство“ (изразът е на E. Dijkstra) и поле за непрестанно усъвършенстване на възможностите на езиците за програмиране и другите езикови средства на информатиката. От друга страна, доколкото програмирането е, за разлика от математиката, също и промишлена дейност, то е и ежед-

невната практика на решаване на различни приложни задачи от хиляди програмисти.

Наред с изброените сходства, математиката и информатиката се различават съдържателно по редица признаци. Да обърнем внимание на някои от тях.

• Математиката има и конструктивно, и неконструктивно съдържание. Съответно двояк е и езикът, на който се извършват математическите разсъждения. Информатиката борави изключително с конструктивни обекти.

Конструктивизмът в информатиката може да бъде както абстрактен, какъвто е в математиката, така и обусловен от реални ресурсни дадености – и при поставянето на проблеми, и при решаването им. Другояче казано, интересуваме се не само дали дадено построение е крайно, но и с какъв *конкретен* (а не само като порядък) ресурсен обем се постига то на дадено реално или мислимо изчислително устройство.

• Математиката изучава няколко вида структури. Разпространена е представата (Бурбаки и др.), че основните сред тях са наредбените, алгебричните (операционните) и топологичните. В информатиката преобладават наредбените структури, но в известен смисъл неин предмет е и (дискретната) структурност въобще, в различните ѝ форми, свойства и аспекти, както и преобразуването на едни структури в други. Информатиката е наука за строежа и за построяването. По израза на R. Milner [37] – „*наука за това как стават нещата*“.

Споменатите строеж и построяване са диалектически свързани елементи – съответно статичен и динамичен – на структурността. Тези понятия визират основната в информатиката дуалност между *данни* и *процеси* от гледна точка именно на структурността в тях.

Предмет на разглеждане в информатиката са и редица конкрет-

ни, специфични за нея структури – даннови, управленски и др.

За отбелязване е, че тъкмо „наредбените“ дялове на математиката са като че ли най-пренебрегнати понастоящем в нея, но има изгледи, че това положение ще претърпи изменение в непосредствено бъдеще, именно под влияние на информатиката, за което става дума и по-нататък.

• Редица еднакво именувани понятия в математиката и информатиката са съдържателно различни. Например в математиката *променлива* е понятие, тясно свързано с понятието функция и бидейки такова то служи за изразяване на отношения между множества. В информатиката, най-вече в императивното програмиране, променливите се създават, изменят явно или неявно, унищожават и др. В математиката почти няма аналог на локална или глобална променлива, разделяема променлива и др. (Означения-имена, които не отговарят на променливи, се използват интензивно и в двете науки.) Понятието *функция* също е като цяло различно в математиката и информатиката.

• Специфичните за информатиката и основни за нея понятия *памет, достъп, ресурс* нямат аналози в математиката. Понятието *тип*, макар да се разглежда в математиката и особено в математическата логика, има значително по-централно място и много различно съдържание в информатиката. Това личи особено в посъвременните езици за програмиране, където има тенденция да се предоставят развити средства за пряко реализиране на различни операции върху типове, последните разглеждани в известен смисъл като данни.

• Особена значимост в информатиката има *фòрмата на представяне* на обектите, например конкретният алгоритъм за решаване на дадена задача или конкретната програма, реализираща даден алгоритъм. Донякъде аналогични в математиката са представянето на

различни доказателства на дадено твърдение и теорията на доказателствата в логиката, но за информатиката е твърде характерно, че формата на представяне е и сама по себе си предмет на изучаване.

Освен това информатиката и програмирането често боравят с йерархични схеми на сводимост на представянията (равнища на абстрактност), при което в рамките на едно разглеждане, например дори една програма, на информатѝка може да се наложи да владее едновременно няколко равнища. Различните равнища на абстрактност по принцип могат да изискват за описването им различни езикови слоеве.

• Програмирането борави с множество специфични парадигми [36, 9] на различни семантични равнища. Парадигмите на най-високо равнище – императивна, апликативна, функционална, комуникационна, съпоставителна, продукционна – определят изобщо възгледа върху процеса на съставяне на програми и качествата, които програмата като формален обект притежава. Според нас в съвременната математика не се наблюдават така съществено различни подходи (и оттам езици) за описване на едни и същи същности.

• Информатиката, като по-късно обособила се научна дисциплина, няма относително завършения и устойчив вид, който притежават много от математически дисциплини. С оглед на това методологическите аспекти в нея са особено съществени и заедно с това – недостатъчно добре разработени. Конкретно в програмирането тези два факта се отнасят както до алгоритмиката, така и до обслужващите програмирането езикови средства от всякакъв вид, а също и до технологичните аспекти на областта.

Тъй като програмирането е бързо развиваща се област, динамиката на развитието му обхваща и понятийната му основа. Обособяват се различни нови понятия, а някои изменят смисъла си. Това влече висока динамика на развитие и усложняване на езиците за

програмиране, но и затруднява систематичното им изследване.

Дотук, говорейки за математическия език, имаме предвид най-вече езика на съдържателните разсъждения в математиката, който се определя в значителна степен от характера на предмета ѝ. По негов адрес следва да отбележим още, че той в редица области става все повече алгоритмичен, т. е. на свой ред търпи влияние от страна на информатиката, в случая – алгоритмиката. Всъщност алгоритмичността, както вече беше споменато, винаги е имала известно място в представянето на доказателства, методи и др. в математиката. Това се наблюдава най-вече в т. нар. конструктивни доказателства.

Понякога приликата на дадено доказателство с алгоритъм е удивително богата на аналогии. По този повод не можем да се въздържим да не споменем доказателството на известната теорема на Wallace-Bolyai-Gerwien за това, че всеки многоъгълник е сводим до всеки друг със същото лице чрез разрязване на краен брой части и подреждането им наново, както е изложено в [49]. Най-напред, даден многоъгълник се разрязва с краен брой стъпки, при това еднообразно извършвани (итерация с броене), на триъгълници. Всеки така получен триъгълник „чрез обръщение към подпрограма“ – посредством два разреза – се превръща в правоъгълник. Всеки правоъгълник се преобразува с краен брой разрези (друга „подпрограма“) в квадрат, като някои правоъгълници се налага първо да се преобразуват в други правоъгълници (това се прави възможно няколко пъти до получаване на подходящ правоъгълник, т. е. отново итерация). Накрая, всеки два квадрата чрез фиксиран брой действия (отново „подпрограма“) се преобразуват в друг квадрат, така че от всички получени квадрати се образува един (итерация), а от него чрез „обръщане на програмата“, за което програмирането разполага с формални, макар и не автоматични методи, се получава другият отнапред даден многоъгълник.

Смята се, че във връзка с развитието на информатиката силно ще расте значението на дискретните дялове в математиката [4, 3] (според втория от посочените източници в най-близко бъдеще ще се наблюдава възраждане по-специално на комбинаториката). Такова развитие предполага все по-засилващо се взаимно влияние между езиците на математиката и на информатиката.

Наред със съдържателните аспекти на математиката, не по-малко съществено влияние върху програмирането оказва традиционно използваната в нея или по-съвременна нотация.

Тъй като именно математиката има най-силно развита и сравнително устойчива практика в използването на нотация – система от символни означения – заимстването от нея в езиците за програмиране както на отделни обозначения, така и по отношение на комбинирането им, дава възможност да се използва натрупаният чрез обучението и практикуването на математиката опит.

Разбира се, използването на символи далеч надхвърля рамките на математиката (вж. напр. [6]). От древни времена културата на повечето народи е тясно свързана с използване на графични символни обозначения, един от резултатите на което е появата на писменост за различните говорими езици и в частност – на азбуки. При това, освен носители на смисъл, знаците често имат естетическа и емоционална натовареност. Нещо повече, в човешкото възприятие визуалните символи придобиват относително самостоятелна спрямо обозначаваните и означавани от тях предмети и идеи роля. Съществува и феноменът на очарованието от символа – съществен психологически фактор при възприемането на знаци и знакови комбинации.

Установяването на символика, разбирана като система от знаци във взаимодействие помежду им, е пределна степен на формализиране при обособяване на понятия и връзки между тях. В руслото на разнообразните символни представяния особеното значение на мате-

матическия символен език е в това, че е системен, т. е. именно език, а не набор от означения, и в това, че съответната му референтна система от понятия и връзки е високоабстрактна.

Достойнствата на математическата нотация не я правят непосредствено използваема в езиците за програмиране. Двете основни причини за това са, че тя обикновено не е *нито еднозначна, нито изпълнима*. По отношение на първото: „нестрогото боравене със символиката е основна черта на математическата култура. То е преднамерен отказ от точното формално означаване на математическите обекти [... тази нестрогост] отваря възможности за *плодотворна двусмисленост* [...] въпреки че в даден контекст смисълът [на даден израз] е напълно определен, често можем да съзрем и друг смисъл, който обогатява тълкуването“ ([58], вж. също [43]).

На математиката е присъща значителна свобода и по отношение на въвеждането на символи, вкл. и графично нови, и по отношение на интерпретирането им. Така например специални обозначения за операции, свойства и отношения могат да се появят в отделна публикация на даден автор или дори в съвсем частен контекст. Изобщо тълкуването на знаците може да зависи и от темата, и от автора, и от непосредствения контекст. Разбирането на смисъла на даден знак в математиката често става полуинтуитивно, чрез употребата му и се подпомага от неформални словесни обяснения. Контекстът, в който се появява и интерпретира математическият символен запис, е този на естествения език. В качеството му на продължение на говоримия език, този на математиката е съществено несамостоятелен.

Обратно, програмистът работи с предопределена от езика за програмиране азбука, а с това начините за образуване на „макрознаци“ – имена на операции и други означения в програмите – е строго ограничен. В повечето езици ограниченията са дори по-големи: небуквените знаци имат отнапред фиксирана употреба и не могат да

бъдат комбинирани за образуване на нови означения. Освен това взаимодействието между различните елементи на езика в програмата е строго фиксирано от синтактичните правила.

Освен, че програмистът не разполага с почти никаква свобода да доопределя езика, на който създава програмите си, не е маловажно и това, че както създаването на средства за обогатяване на писмеността в програмирането (например въвеждане на произволни графични символи или неедномерен начин на записване), така и „прочитането“ ѝ (формално обработване от какъв да е вид, като интерпретиране, превеждане в друга форма и др.) трябва да отговарят на известни ограничения за реална ресурсоемкост.

В [43] се посочва, че благоприятен път за усъвършенстване на писмеността на програмирането е заимстването от математическия понятиен език и символика и дори доколкото е възможно – обединяването на ЕП с този на математиката. Това дава възможност формалният математически запис да стане недвусмислен и изпълним, а ЕП да се обогатят чрез практически доказаната адекватност на математическите понятия и стил на записване. Смята се, че усъвършенстваният чрез „математизиране“ даден език за програмиране (в цитираната публикация илюстрирано с езика APL) трябва да съхрани споменатата по-горе ползотворна двусмисленост на математическата нотация под формата на полиасоциативност (в оригинала *suggestivity*): формата на даден фрагмент, имащ отношение към решаване на дадена задача, да *подсказва* подобни конструкции, приложими в други случаи. За целта е нужна богата елементна база на езика – подходящо голям набор от вградени операции и средства за комбинирането им.

Освен при APL синтактично и семантично приближаване на нотацията в програмирането до тази в математиката се наблюдава в също апликативния език FP [13] и при чисто функционалните езици

като MIRANDA [80] и HASKELL [60], където изпълнението на програмата може да се обясни чрез последователни заместявания в текста ѝ (т. нар. *rewriting systems*).

Формалната достатъчност на дадена нотационна система като базис за решаване на определен кръг проблеми в програмирането не следва да се тълкува като достатъчност на изразните средства, доколкото тя не непременно осигурява *адекватност*. На проблема за изразната адекватност се спираме и по-нататък, тъй като е тясно свързан с основния предмет на разглежданията ни – явността.

Накрая да обърнем внимание на още две съществени разлики между природата на математиката и тази на програмирането. Първо, програмирането в различните му форми постепенно става все по-масова дейност, сравнено със заниманието с математика. Не става дума само за увеличаване на количеството на професионалните програмисти. На практика ползването на повечето съвременни програмни системи или дори на обикновен калкулатор съдържа елементи на програмиране, тъй като изисква характерното за последното планиране на действията.

Споменатата масовост не е и само в смисъл, че програмирането участва по един или друг начин в професионалните занимания на все по-голям кръг от хора. Обемът на формалния материал в писмена форма, създаван, проверяван и другояче анализиран, най-често е по-голям у програмиста, отколкото у математика. Последното, разбира се, не влече същото съотношение и на трудностите на съответните видове работа; степента на новост, на неповтаряемост може да бъде по-голяма в математическия текст.

Второ, в програмирането като производствена дейност се налага разглеждането на въпроси за производителност, ергономичност и др. под., каквито като правило не се засягат в математическата

практика. Можем също да говорим за „красива програма“ точно така, както говорим за „красиво доказателство“, но в първия случай се опитваме и да обясним тази красота чрез особености на алгоритъма и/или езика за програмиране, а понякога и да я измерим. Изглежда, че информатиката е в по-голяма степен от математиката саморефлексивна област на знанието и практиката.

2. Развитие на представата за програмите и езиковите средства за програмиране. Фактори за възприемане на програмен текст

Алгоритмиката и програмирането като научна област и професионална дейност на все по-голям брой хора са едно от изключителните цивилизационни явления на съвременността. Най-напред, информатиката – теоретична и приложна – е уникална с това, че зародишното ѝ състояние, преди да се обособи в самостоятелен организъм, продължава векове. При това тя е примерно поравно рожба на две майки: теоретичната наука, и то от най-абстрактен вид, и технологията. Продължавайки аналогията с жив организъм, зародишът на информатиката преживява интензивен предродов период: няколко мощни тласъка за развитие през 19 в. – станът на Жакар, машините и теоретичните трудове на Бабидж, работата на Менабреа и Лъвлейс, разработките на С. Н. Корсаков [46] – и много бързо развитие през трийсетте години на 20 в., когато за този зародиш съществува повишено хранителна среда от страна и на теорията, и на технологията.

Раждането на същинската информатика, станало в средата на века, се посреща възторжено – белег на натрупано дълго очакване. Новороденáта, станала едновременно и научна, и приложна област, се оказва заобиколена с изобилно внимание и грижи, понякога дори до степен на нездравословност, но заедно с това – и натоварена с

отговорността за сбъдване на множество очаквания към нея, сред които и немалко илюзорни. И едното, и другото са фактори, които могат да стимулират развитието, но и да предизвикват объркване, подтискане на положителни тенденции и избиране на грешни посоки. Много съществен фактор, също двояко влияещ, е това, че в областта на практическата информатика присъстват изключително мощни комерсиални интереси.

Наистина, едва ли има друга наука, износвана тъй дълго, но за която отчетливо може да се посочи времето на раждане и която след това се развива така бързо, при това с огромно влияние върху (и от!) различни области на практиката.

Като бързоразвиваща се система, информатиката има изменчиво, еклектично и трудноопределимо съдържание. Смятаме, че това личи ясно в областта на езиците за програмиране, разбирайки последните както в тесен, така и в широк смисъл, т. е. включвайки и например командните езици на операционните системи, езиците за програмно управление на разнообразни приложни програми и други специализирани езици. Като формализират понятията, с които борави информатиката, заедно с присъщите им свойства и взаимодействия, езиците са концентриран израз на характерното за тази наука научно отражение на света в един или друг негов аспект.

Свидетелство за посочената еклектичност и подвижност е например това, че наред с най-съвременните ЕП, представящи различни смятани за перспективни парадигми в програмирането, остават жизнени и някои от езиците-ветерани с над 50-годишна възраст, като Fortran, COBOL и BASIC.

Съществуващото вече огромно разнообразие от езикови средства на програмирането прави много трудно класифицирането и изследването на техните характеристики и елементи, както и на особеностите на програмирането като процес.

От друга страна, изследването на свойствата на ЕП и на програмите, записвани на тях, се налага не само поради вътрешна за информатиката, отвлечена мотивираност, но също и поради прагматични съображения, свързани с промишленото програмиране: най-общо казано, за да се повишава качеството на труда в програмирането и на резултата от него, както и за да се усъвършенстват анализът и планирането им.

Докато в зората на програмирането обект на внимание във връзка с програмите бива само онова, което е свързано с тяхното правилно и ефективно разполагане и изпълняване, а по-късно – с възможността за пренасяне и изпълняване на различни компютри, постепенно става нужно да се обръща внимание и на други техни качества. Застрашително растящият обем на произвèдения и на предстоящия да бъде произведен софтуер обуславя непознато до момента равнище на сложност на системите. Провалят се някои големи софтуерни проекти. Във връзка с липсата на подход за овладяване на тази сложност става популярен изразът „криза на (производството на) софтуера“ (60-те и 70-те год. на 20 в.).

Това е една от главните причини да се съсредоточат усилията на специалистите към превръщането на програмирането в систематично занятие, а в написаните програми да се забележат нови качества. Появява се терминът (напр. [25]) *структурно* или *систематично* програмиране. Формулират се и някои други качествени характеристики на програмите. Това развитие на свой ред подтиква изследвания и на възможностите да се правят *измервания* на количествени и качествени аспекти на програмите. В литературата по информатика се появяват резултати от изследвания на програмни текстове със средствата на психологията, както и методики за формално измерване на софтуер [40, 61, 33, 76, 73, 24].

Този вид изследвания върху програмите и програмирането е дал

известно отражение върху представите за качествата, които ЕП трябва да притежават, но сравнението между изводите от споменатите изследвания и езиците, с които разполагаме, показва, че влиянието е много по-малко от желателното. В хода на изложението са посочени примери за подобно несъответствие.

Друго следствие от повишеното внимание към систематичното програмиране, а и въобще от развитието на представите за ЕП, е формулирането на някои принципни свойства, които да притежават ЕП, например в класическите [42, 85], а също в повечето монографии по езици за програмиране, като [35, 70, 81]. Това обаче са по същество препоръки за преценка на качествата на един или друг съществуващ език за програмиране; поради общността си те не могат да послужат за *проектиране* на такъв език.

В структурата на даден ЕП обикновено се различават две равнища: макроравнището, на което се визират понятия като архитектура, модулност, интерфейси и други, имащи отношение по-скоро към проектирането на програми (*large-scale programming*), а не собствено програмирането, и равнището на непосредственото програмиране (*programming-in-the-small*), боравещо с понятия като стойност, променлива, операция, израз, команда. Всеки от тези два условни дяла в езика има своя значимост за програмирането. Доколкото напоследък се полагат усилия предимно за развиване на езиковите средства на макроравнището, особено в рамките на обектноориентирания (ОО) стил на програмиране – бяха създадени нови ОО езици, а на редица други беше придадена обектноориентирана надстройка[1] –

[1]Мимоходом да отбележим, че въпреки споменатите усилия за налагане и усвояване на ОО подход в програмирането, неедноkратно е посочено, че той нито е универсално приложим, нито има удовлетворителна методология за прилагането му, а и сам по себе си съдържа съществени недостатъци. При това някои от смятаните за негови достойнства не винаги са налице, тъй като твърде много зависи от решенията, които се вземат във връзка с конкретното му прилагане [74, 41].

създаде се известна тенденция да се пренебрегва другият дял, което пък създава впечатление за изчерпаност на решенията, отнасящи се до него, или за това, че той е маловажен.

От друга страна, повседневната дейност на занимаващите се с програмиране е свързана именно с непосредствената му форма и тя именно преобладава като практика в програмирането. Следователно съответните езикови средства влияят най-пряко и са определящи за продуктивността на този вид труд.

Впрочем, в създаваните или активно развивани напоследък ЕП се срещат нови решения и в областта на непосредствения, „нисък“ етаж на програмирането, но подобренията се постигат чисто емпирично и не са предмет на изследвания. Обилният някога поток от научни публикации върху управляващи и други структури в ЕП отшумя, а предизвикалите го реални проблеми с езиците като цяло останаха.

С оглед на изложеното нашият интерес в тази работа е насочен именно към езиковите средства за непосредствено програмиране, като в тази глава и по-нататък последователно стесняваме и конкретизираме кръга на интересуващите ни въпроси и предлагаме някои решения.

Трябва да се отбележи, че оказалите може би най-голямо влияние върху работната култура на съвременния програмист широкоизвестни трудове по систематично програмиране на E. Dijkstra и C. A. R. Hoare [25, 28], D. Gries [38], N. Wirth [84] и др. са посветени именно на непосредственото програмиране, но страдат от значителна едностранчивост. В тях се слага ударение върху възможностите формализирано (чрез математически по характер методи) да се получи текст на програма по зададени условия, които трябва да бъдат налице преди и след нейното изпълнение, както и върху евристични

методи за разграждане на дадена задача за програмиране до получаване на лесно обозрими части. Макар и полезен, такъв подход е малко самоцелен, подпомагайки все пак повече математическото третиране на програмите, отколкото непосредствено програмирането. Заедно с това в този род изложения се използва или конкретен ЕП, или малък набор от псевдоезикови конструкции, като и в двата случая се подчертава, че демонстрираните методи на програмиране са в голяма степен езиковонезависими.

Тези две страни на споменатите трудове по систематично програмиране дават представа за програмирането от гледната точка на една чиста алгоритмика, като лишават от възможността да се забележи приносът на езиковото богатство и разнообразие при решаване на задачи. Така те влизат в известно противоречие с по-конкретната и по-пъстра представа за структура на програмите, която получаваме чрез различните ЕП.

Емпирично-психологическите изследвания за възприемане на текстове на програми, макар и полезни с възможността да се забележат или обосноват някои езикови закономерности, също страдат от едностранчивост. Първо, те третират изолирано, и то много малък кръг конструкции в ЕП, обикновено за управление на следването на действията на командно равнище – условни, циклични или смесени форми. Второ, изследваните конструкции, а и хората, които проиграват съответните тестове, са най-често така или иначе „контекстнообусловени“ от съществуващите, пряко или косвено известни, подобни конструкции.

Смятаме, че по-ценно (но и по-трудно реализуемо) би било емпирично изследване на езиковия материал на ЕП, при което се експериментира с много и различни по предназначение езикови конструкции, имащи различна семантика, синтаксис и лексика, разглеждани поотделно и в съчетания помежду им. Това се диктува от факта, че

никой аспект на езика не е без значение: семантиката, абстрактният и конкретен синтаксис, прагматиката, идиоматиката — общо и за всяка от съставките на езика.

В оставащата част на тази глава правим преглед на някои свойства на текстовете на „малки“ програми, т. е. свойства, проявими и разглеждани в светлината на дребномащабното програмиране, които имат значение за възприемането на програмата при четене. Такива свойства по принцип могат да произтичат пряко от използвания език за програмиране или да се дължат на начина на използването му. Ясно е, че ако дадено качество на програмата е желателно, то е добре да разполагаме с такъв език, който позволява то да бъде реализирано. Още по-добре е, ако езикът подтиква към това, а в някои случаи – и задължава.

От друга страна, част от свойствата, които описваме, са в малка или в по-голяма степен субективни, защото зависят например от подготвеността на този, който чете текста на програмата.

(Малка) част от свойствата се поддават на формализиране, още по-малко на измерване. За други е неизвестно дали това може да се направи, тъй като (доколкото сме осведомени) не е направено, а смисъла на трети оставяме по-скоро интуитивен. Във всички случаи не се стремим да постигнем съответствие с възможно използваното другаде значение на даден термин.

• **Краткост**

Краткостта се изразява, въобще казано, с обема на текста. Макар и желателна сама по себе си, тя невинаги дава добра представа за лекотата, пълнотата и т. н. на възприемане на текста. Краткият текст може да не бъде *непосредствен* или *адекватен*.

• **Непосредственост**

Под *непосредственост* разбираме съответност на използваните елементи на дадения ЕП и на съчетанията от тях в програмата с *пред-*

варителните представи на четящия за начина на използването им, т. е. най-вече с традиционната прагматика на езика.

Трябва да се отбележи, че самият ЕП може да се характеризира със слаба непосредственост, както е според Ф. Брукс:

> „[. . .] най-добра е онази система, която позволява нещата да се задават с най-голяма простота и непосредственост. Само *простота* не е достатъчна. Разработеният от Муър език TRAC и ALGOL 68 достигат простота, ако се измерва с броя на различните основни понятия, но тези езици не притежават *непосредственост*. За да се изрази с тях това, което се цели, често се налага основните средства да се комбинират по заплетени и неочаквани начини. Не е достатъчно да се изучат само частите и формалните правила за съчетаването им; трябва да се познава и идиоматичната употреба, установените в практиката схеми за съчетаване на частите“ [22].

В случай на ЕП, който в смисъла на горното не е непосредствен, може да се смята, че „предварителните представи“ включват и характерната за езика идиоматика, доколкото тя се съдържа в прагматиката му.

- **Адекватност**

Адекватността е друг вид съответност – между програмата като съчетание от езикови елементи и абстрактния модел на решението на определена задача, който програмата представя. Тя е обективно понятие, макар и в по-малка степен от непосредствеността например. Адекватността е тясно свързана с *явността* и е по-подробно разгледана в следващата глава.

- **Явност**

Говорейки за *явност* или *неявност*, имаме предвид наличие или отсъствие на информация за свойства и отношения на фигуриращите

в програмата обекти. При това наличието на самите свойства и отношения се смята за известно по някакъв начин (от контекста на някакъв абстрактен модел на решението).

Неявността е форма на *неадекватност*, според това което определихме за нея; последната е по-общо свойство, тъй като може да обозначава и други видове несъответност.

Подробно разглеждане на понятието *явност* и съотнасянето му с близки до него понятия разглеждаме в следващата глава.

• Нагледност

Нагледността, в различните ѝ степени, е свойство на всяка визуално представена информация. Кратко определена, тя е достъпност за „непосредствено“ възприемане на основата на навици и изобщо подготвеност у зрителя или четящия. Тъй като също е свързана с *явността*, спираме се отново на нея в следващата глава.

• Разход на памет

В [11] се използва подобно понятие (*memory load* – натовареност) в смисъл на обем от допълнителна информация за *други* елементи на програмата – типове, стойности, изпълнени условия – необходим за разбиране на даден израз, команда или друг малък фрагмент.

Тук определяме „разхода на памет“ в по-широк смисъл, включвайки и възможната натовареност, която се появява в процеса на четивна интерпретация на *същия* фрагмент. Например изразът

```
a+((b+c)+(d+e))
```

пресмята, изобщо казано, същото, което и

```
a+b+c+d+e
```

но по-сложно: чрез запомняне на няколко междинни резултата вместо чрез последователно сумиране (и само един междинен резултат).

• Вербалност/нотационност

Отнася се до това, дали в записа на програмата преобладава нотацията, т. е. дали се използват голям брой специални знаци за раз-

личните действия в нея при съответно малък брой думи или преобладава словесният изказ. (Използването на думата „нотация“ тук в ограничения смисъл само на несловесна система за означаване е преднамерено и само в този контекст.)

Смята се, че удобно въведената система от знаци води до по-бързо и лесно възприемане на текста, тъй като знаците са по-ясно визуално различими от думите. Те могат да бъдат предпочитани и по естетически и други сходни причини (вж. също гл. 1). От друга страна, твърде големият брой знаци, особено при зависеща от контекста смислова натовареност, снижава лекотата на възприемане.

Очевидно това свойство се предопределя от използвания ЕП. Примери за типично вербален и типично нотационен езици са вече коментираните COBOL и APL и J. В последните два *всички* елементи с изключение на избираните от програмиста имена на функции и променливи са изразени нотационно.

• **Мнемоничност**

Това е свойството на отделните лексикални елементи на записа на програмата да предават непосредствено смисъла, заложен в тях, на основата на привичност или асоцииране на обозначенията с други познати обекти. Може да се отнася до ключови думи, символи, както и до имена, избрани от програмиста.

Някои големи софтуерни фирми имат въведени задължителни правила за използване на мнемонични означения от сътрудниците си, с което улесняват възприемането на текстове на програми в рамките на даден колектив.

Чисто визуални страни на представянето на програмния текст също могат да се използват за усилване на мнемоничността (вж. по-долу).

• **Еднообразност**

Под *еднообразност* разбираме смислово близките части на програмата да имат сходна форма. Например намирането на сбора и на произведението на елементите на дадено множество от стойности (представено чрез масив или др.) да са записани подобно. Също така фрагментите, решаващи такава задача върху множества с различни представяния, например масив и списък, да имат сходен запис.

• **Визуални характеристики и атрибути**

По отношение на визуалното представяне в даден програмен текст могат да се открояят различни характеристики: форма и особености на разполагане, наличие на ключови и други характерни думи, характерни знаци, шрифтово оформяне, цвят. Визуалното представяне също може да има отношение към явността, доколкото може, било просто нагледно да *открои*, било във формалния смисъл да *осигури* явност.

Формата и особеностите на разполагане включват ширина на текста (брой знаци), отстъпи за изразяване на вложеност, размер на конструкциите „с тяло“: условни команди, цикли, процедури. В някои езици – OCCAM [66], HASKELL [60], PYTHON и др. отстъпите за влагане са част от синтаксиса.

Колкото до шрифтовото оформяне, има създадени известни традиции в типографското представяне на програмен текст. Програмите на PASCAL, ADA и някои други езици се отпечатват в учебници и монографии по програмиране със смесен шрифт – получер за ключовите думи на езика и курсив за останалите, докато тези на C – с „машинописен“ или „куриер“, а на APL – с куриер-курсив.

Редица програми – редактори на текст използват различни цветове, а и други визуални характеристики, за изобразяване на различните класове от лексеми.

Типографските характеристики на програмния текст се трети-

рат като важен фактор за възприемането му (вж. напр. [14, 7]), но като цяло за визуалните елементи не може да се смята, че е достатъчно добре изучен начинът, по които те участват в това възприемане.

Изброените свойства на програмни текстове не изчерпват факторите, обуславящи възприемането на такъв текст. Например при ниска *непосредственост*, *адекватност* или *нагледност* възприемането на програмния текст може да се подобри чрез коментар или друг вид съпровождаща информация.

3. Факторът „явност на изразяване“ и сродни понятия, отнасящи се до програмен текст

3.1. Общи положения върху понятието явност

Характерна черта на текстовете на естествен език е, че те са в определена степен интуитивни. Предаването чрез тях на информация към четящия, дори когато става дума за „формални“ описания, частично разчита на неговата интуиция, на подразбирането на сведения, които не са предадени явно, а се внушават непряко, долавят се чрез вътрешни или външни аналогии, обобщаване, абстрахиране и други свойствени на мисловния процес механизми. Както вече видяхме, същото е вярно, в различни степени, и за математическите вербални и нотационни текстове. Това е една от основните пречки, ако не и главната, към успешното формализиране и оттам подчиняване на автоматична обработка на редица области на практиката, в които знанията успешно се предават и развиват с помощта на естествения език и „обикновената“ човешка интуиция.

Текстовете на компютърни програми, макар да са най-високо формализираният вид текст, наследяват неявността на естествения език, разбира се в специфична форма и в друг обем. Някои видове подразбирания са неприложими в програмирането, или поне не е прието да се практикуват. Текст, разчитащ на такива подразбирания, прави програмата формално некоректна и такива (не)програми са извън нашето разглеждане.

Друг вид подразбирания – също източник на неявност – са онези, които се приемат в даден ЕП по отношение на описàния, на преобразуване на типа на даннов обект и др. При такива подразбирания информацията не се задава непосредствено в текста на програмата и в този смисъл можем да я смятаме за неявна, но правилата на езика я правят в крайна сметка налична, а поради това този вид неявност също не е основният, който ни занимава. Оставяме настрана въпроса за това в каква степен и какви подразбирания са уместни или не и помагат или пречат на съставянето, проверяването, изменянето и други дейности с програмата.

С понятието явност тук визираме такива страни на текста на дадена програма, които не се отразяват на изпълнението ѝ, а значи – и на коректността на резултатите, които се получават при това, но *обективно повишават нейната информативност*. Явното или неявно изразяване на свойства и връзки на програмните обекти влияе както на предаването на информация за програмата и за реализирания от нея алгоритъм при *четивно възприемане*, така и на възможностите текстът на програмата или преобразувана нейна форма да бъдат *автоматично обработвани* от транслатори и други инструментални програми. Големият брой явно изразени факти за програмата повишава информативността ѝ и така подпомага нейното вярно и пълноценно възприемане, проверяване, поддържане, изменяне, оптимизиране, измерване, извличане на фрагменти за повторно използване и други полезни дейности, извършвани „ръчно“ или автоматично.

Явността на изразяване може да се отнася до всяко свойство или отношение в програмата. Другояче казано, явност или неявност може да присъства във всяко предписано от програмата действие: пресмятане на израз, изпълняване на команда или на подпрограма, формиране на описание на променлива, даннов тип, подпрограма

и др., без значение дали действието се извършва при изпълняване на програмата или преди това.

Да разгледаме следната задача за определяне на масиви чрез езика C. Трябва да определим два масива така, че да имат едни и същи размери, например `[100][50]`.

Най-простото решение е

```
int a[100][50];  int b[100][50];            // 1
```

Двата масива действително имат еднакви размери, но не може да се твърди дали това е преднамерено или е само съвпадение. Явна информация за това липсва.

Възможно подобрение е следното решение, където размерите са определени самостоятелно като константи:

```
#define S1 100                              // 2
#define S2 50
int a[S1][S2];  int b[S1][S2];
```

Тук вече е ясно, т. е. явно изразено е, че размерите на `a` и `b` са *преднамерено* еднакви, но сега пък отсъства явна връзка между двата размера, т. к. те са определени независимо един от друг. Недостатък е и това, че решението изглежда различно според *размерността* на масивите: колкото по-голяма, толкова по-тромаво.

Едно от другите възможни решения е да определим тип, който сетне да се припише на двата масива. Така масивите не само са направени едноразмерни по явен начин, но тази явност е допълнително подчертана чрез невъзможността да променим размера на само едния от масивите:

```
typedef int ATY[100][50];                   // 3
ATY a,b;
```

По този начин обаче масивите се оказват непременно от еднотипни елементи – ограничение, което предишният вариант не поставя и което може да е неприемливо. Следният вариант отстранява този недостатък, като образува смесена стойност:

```
struct {int a; char b;} s[100][50];          // 4
```

И това решение може да е нежелателно, тъй като при него масивите са не самостоятелно образувани и съществуващи обекти в програмата, а части от един. Всъщност тук дори имаме не два масива, а масив от еднообразни смеси. Третирането му като отделни масиви – при посочване, копиране и др. – е невъзможно.

Още едно решение на задачата е следното:

```
#define S [100][50]                          // 5
int a S;
char b S;
```

Тук масивите са самостоятелни, типът на елементите на всеки от тях може да се избира независимо от другия, едноразмерността е явна, непроменима и осигурена чрез само една конструкция (последните две – за разлика от вариант 2).

Последният вариант може да бъде видоизменен, получавайки още едно решение на задачата. Макроопределението `Adecl` служи за описване и образуване на масив по зададени име, тип на елементите и размер(и) – то е синтактична алтернатива на вградената за това средство в езика и всъщност негова „шапка“.

```
#define Adecl(n,T,S) T n S                   // 6
#define S [100][50]
Adecl(a,int,S);
Adecl(b,char,S);
```

За достойнство на това решение може да се смята фактът, че чрез използване на нарочно въведена конструкция допълнително привлича вниманието към особеното, подчинено на някаква обвързаност, задаване на размерите на масиви. Явността на изразяване на нужното ни свойство е допълнително подчертана.

Да разгледаме и друга подобна задача: единият от двата масива трябва да бъде определен така, че да има размерите на другия.

Да подчертаем разликата от вече разгледаната постановка. В нея изискването за общност на размерите се диктува от някакво обстоятелство, на което *двете определения се подчиняват съвместно*, а между самите определения няма (не трябва да има) връзка на подчиненост – те са съотнесени равноправно. Във втората задача едното определение трябва да се формира независимо от другото, а то пък да бъде подчинено на първото.

Всяко решение на предишната задача е приближено решение и на новата (както и обратно), но именно само приближено: то изразява друго отношение в програмата. За решение тъкмо на втората задача можем да приемем следното:

```
#define NumEl(x) (sizeof(x)/sizeof(x[0]))
int a[100];
char b[NumEl(a)];
```

Макроопределението `NumEl` пресмята броя на елементите на масив по зададено име и може да се използва за различни цели. В този случай чрез него осигуряваме, че какъвто и да е размерът на масива `a`, този на `b` е същият, независимо от възможно различните типове на елементите им. (Използваме това, че операцията `sizeof` се изпълнява не в хода на изпълнението на програмата, а предварително, по време на компилирането ѝ.)

Това решение обаче е валидно само за едномерни масиви и не ни

е известен удовлетворителен начин за решаване на задачата в общия случай.

Вижда се, че дори в много прости примери като горните свойството явност се проявява съществено и следва да се разглежда съвместно с други важни свойства. Някое или повече от едно от посочените решения могат да бъдат удовлетворителни според контекста на задачата – например дали предвиждаме евентуални изменения (и какви именно) и желаем да ги улесним, или обратно – да подчертаем, че изменения са малковероятни или нежелателни, като ги затрудним. Съществено може да бъде и това дали е желателно програмата да може да бъде подлагана на автоматичен анализ, а още повече – на преобразуване, дали самата тя се създава ръчно или автоматизирано и др. Не може да има универсални правила за това кой вариант да изберем или дори какъв компромис е допустим.

Например значителна част от предложените по-горе решения съществено използва предпроцесора на езика C. Това по същество е извънезиков механизъм и като такъв затруднява или дори препятства формалното извличане на информация от програмата, а следователно анализа и преобразуването ѝ. Прибягваме до този механизъм по липса на по-адекватни, собствено езикови средства за осигуряване на нужните свойства. За съжаление недостигът на средства за явно изразяване на редица свойства далеч не е белег само на езика C, а и не само на „класическите“ ЕП. Благоприятните в това отношение черти на по-съвременните езици все още са твърде малко.

Желаната явност в различни случаи може да се осигури чрез подходящ коментарен текст към програмата. Това обаче в мнозинството от случаите е по-лошото решение, отколкото ако явността присъства именно в текста на програмата, по следните три причини. Първо, програмата лесно може да се окаже претрупана с коментари, което в крайна сметка затруднява възприемането и другите

действия с нея. Второ, информативността на действителния текст има предимството, че този текст е формално обработваем от различни програми и следователно зададеното в него е достъпно за тях, което влече възможността за различни анализи и преобразования на програмата, както посочихме в началото на тази глава. Трето, създаването и поддържането на съгласуваност между действителния текст и коментарния съпровод изисква допълнителни усилия и лесно може да се наруши неволно.[2]

По-добро от коментирането, но сравнително ограничено по обхват средство за повишаване на явността е снабдяването на текста с *изпълними* контролни предикати, като командите `assert` в някои езици. Освен че се осигуряват прегради срещу различни непредвидени ситуации, в които изпълнението на програмата може да се изроди в нещо погрешно, така се повишава и явността на програмния текст: чрез всеки контролен предикат едновременно се *създава* и се *прави явно* дадено свойство на програмата.

В най-развит вид изпълнимите контролни предикати се срещат в Eiffel [62] и след него в някои други езици. Те се появяват като пред- и следусловия, които ограждат команди за цикъл, процедури и други конструкции. Програмистът може да избере дали тези

[2]Средства за частично автоматизиране на съгласуването на програмата с нейната документация са разработената от D. Knuth система WEB [51], където програмата и съпровождащите я коментари се създават съвместно, както и някои други, следващи същата схема. Програмата се съставя като „плетеница“ (web) от отделни фрагменти на съответния език за програмиране с прави и обратни връзки на цитиране между тях и, възможно, неформални коментари към всеки. В резултат автоматично се получават два документа: цялостният текст на програмата на съответния ЕП в годен за компилиране вид и документация в TEX или друг формат, съдържаща информация за връзките между фрагментите, самите тях, както и евентуалния осигурен от програмиста коментарен текст. Този подход се пропагандира като метод за създаване на софтуер под името „грамотно програмиране“ (literate programming).

Създаваната чрез „плетеницата“ *формална* връзка на съгласуваност между програмата и документацията като цяло е слаба. В крайна сметка съгласуваността зависи преди всичко от автора на програмата.

предикати действително да се изпълняват, като се включат в изпълнимата форма на програмата, или да се игнорират, подобно на коментари. И в двата случая присъствието им в текста повишава явността на изразеното чрез него.

Интересно е да се отбележи също, като особена форма за постигане на явност, че съвременният функционален стил – езиците ML, Haskell, Scala, Erlang, F# и др. – се отличава със силно изразена декларативност, чрез която става възможно едновременно представяне на програмата и на нейни свойства. Една от характерните форми на декларативност в такива езици е задаването на избор и съответни на него пресмятания чрез съпоставяне с шаблони, чрез т. нар. охраняеми (от условия) действия, или и едното, и другото. Да илюстрираме това с един пример на Haskell за пресмятане на н. о. д. по Евклидовия алгоритъм:

```
gcd a b | a > b     = gcd (a-b) b
        | a < b     = gcd a (b-a)
        | otherwise = a
```

Записът, задаващ *пресмятане*, много напомня традиционния математически запис за *определяне* на съответната функция. Горният текст може да бъде разглеждан и като система от правила за пресмятане, и като система от тъждества, които стойността на функцията удовлетворява при различни стойности на аргументите си. Така основни свойства на определяната функция са посочени в явен вид за целите на самото определение.

Друг пример на декларативно изразяване в съвременните функционални езици е описването на даннови типове, където също приликата с тъждества и определения в математиката е силно подчертана. Например следното определение

```
data BinTree a b = Empty | BinCons a (BinTree b a)
                                     (BinTree b a)
```

задава тип „двоично дърво с алтернативно редуващи се по равнища от корена нататък два типа стойности на възлите“. Определението може да се прочете като изречение: „стойност от тип `BinTree a b` е или празно дърво, или е образувана (`BinCons`) от стойност от тип `a` (корен) и две поддървета (рекурентно позоваване на `BinTree b a`)“. (Редуването на типовете се осигурява чрез размяната на `a` и `b`, формални параметри на определението, в рекурентните цитирания.) Характеризиращите свойства на структурата – двоично дърво и алтерниране на типа на възлите – са посочени и явно, и непосредствено, и пределно лаконично, като те и съставят нейното определение.

Явността се проявява двояко според отношението ни към това, което бива явно или неявно. Когато то е свойство на програмата, преднамерено въвеждано в нея, стремим се да го направим явно посредством подходяща конструкция от съответния ЕП.

Обратно, може да се случи дадено *нежелателно* свойство да е неявно. Такова свойство е например възможността в система от паралелни един на друг и взаимно конкуриращи се за даден ресурс или ресурси процеси да възникне изолиране (starvation) или взаимно блокиране (deadlock). Неявността на нежелателните свойства може да попречи те да бъдат забелязани и евентуално отстранени, или поне да сме информирани за възможните отрицателни последствия от тях за поведението на програмата.

Предвид общността, с която определихме понятието явност, почти изпълвайки обема на думата в етествения език, не е странно, че формите на проявление на явността в текстовете на програми са твърде многообразни. Още един белег на двойственост на понятието явност, с който трябва да сме наясно, е това, че съществуват редица случаи, в които пренебрегваме едно проявление на явността за сметка на друго.

На практика това винаги е свързано с прилагане на механизъм за абстрахиране. Същностно свойство на абстрахирането е, че целенасочено и избирателно се скриват, правят се неявни известни свойства и отношения, за да се обособят, да станат явни други, смятани за по-съществени. От гледна точка на йерархията на абстракциите в рамките на дадена система с всяка отделна абстракция явността преминава на по-горно равнище.

Всеки език за програмиране разполага с разнообразни механизми за създаване на абстракции. Една типична и проста форма на абстрахиране е чрез именуване на клас от еднородни в някакъв смисъл същности. Например името на променлива служи за абстрактно цитиране на някакво множество от стойности, давайки възможност свойства на *всяка* от тези стойности да се изразят наведнъж за *всички* стойности, както и да се зададат, също наведнъж за всички стойности, отношения между тях и други обекти в програмата: изрази, условни и циклични команди и др. Лишавайки се от конкретността на явното боравене с отделни стойности, добиваме възможност да боравим еднообразно с коя да е от тях.

Изобщо, присъщата на даден програмен обект абстрактност зависи от това, доколко съставящите го елементи са представени чрез именуване и какви по вид и обхват множества от други програмни обекти стоят зад използваните имена. Следователно и аспектите на явност/неявност в такъв обект са в необходима зависимост от *степента на параметризираност*, реализирана в него.

В тази работа се интересуваме преди всичко от явността в онези случаи, в които тя е „чист прираст на информация“, без да бъде за сметка на явността в други нейни проявления. Предметът на изследването ни включва възможности за постигане на явност чрез подходящи езикови конструкции. Конкретни конструкции биват разгледани в този план по-нататък, където се и предлагат редица такива,

но най-напред да се спрем по-подробно на някои понятия, близки до понятието явност.

3.2. Явност и близки понятия, свързани с програми

Както видяхме в гл. 2, две важни понятия, близки до понятието явност, са адекватност и нагледност. Смятаме за нужно да допълним вече споменатото за тях, с което да се постигне по-добро разбиране на предмета на изследванията ни и мотивировката за него.

Понятието нагледност по начало е термин на психологията, най-вече – на педагогическата психология и понастоящем се изследва интензивно. Познато от педагогическата, а след това и от рекламната и пропагандната практика, то става обект на интерес в науката през 20-те и 30-те год. на 20 в. [1]. Един от подтиците за това е необходимостта да се представят в достъпна и убедителна визуална форма знанията, придобивани от различни научни области и главно тези, които използват математически апарат, за целите на обучението, инженерната практика и др. С развиването на визуалните и особено на масовите средства за комуникация, с усложняването на дейностите в много области на производството и другаде и особено с появяването на богати възможности за визуално взаимодействие на компютърните програми с техните потребители изучаването на нагледността става особено актуално.

По този начин наред с изобретяването на различни конкретни форми за онагледяване – условни означения в математиката, химията, медицината, транспорта и др., графици, диаграми, номограми, таблици в инженерните и някои научни дейности, статични и динамични изображения върху компютърен екран и т.н. – се правят опити и да се определи същността на нагледността и ролята ѝ в познавателните процеси. Съвременното разбиране на нагледността

може да се охарактеризира (резюмирайки [1]) най-общо със следните три факта:

- Няма единно, споделяно от всички *научно* понятие за нагледност.
- Нагледността не се свежда до привичност или достъпност за непосредствено *сетивно* възприемане, нито до сбора на двете. В нея участва по специфичен начин и рационалното познание. Знанията, възникващи като рационални и ненагледни, постепенно се сливат с първоначално нагледните и започват да се възприемат и оценяват като нагледни.
- Има основания да се смята, че количествена оценка на степента на нагледност може да се получи, като се използват емпирично определяеми характеристики като скорост и точност (безгрешност) на разпознаването на представяните визуално обекти.

Характерът и значението на нагледността *в текст*, т. е. без да се прибягва до пряко възприемана символика, могат да бъдат илюстрирани със следния пример. Известното изречение

```
the quick brown fox jumps over the lazy dog
```

съдържа 35 букви, но за разбиращите английски текст е много по-нагледно за проверяване дали правилно се възпроизвеждат (на екран, принтер и т. н.) 26-те букви от латинската азбука, отколкото низа

```
abcdefghijklmnopqrstuvwxyz
```

в който са само тези 26 букви на азбуката, и то последователно разположени. Ако обаче четящият не разбира английски, вторият вариант се оказва по-нагледен. Ако той не знае и последователността на буквите в латиницата, то вероятно всеки друг низ, в който 26-те букви са разбъркани произволно, има същата нагледност.

В примера се вижда същественото участие на предварителните знания във възприемането чрез нагледност. Предварителните знания формират шаблони за разпознаване на структурни образи (както на визуално, така и на абстрактно равнище), в случая – думи и съчетания от думи, усвоени чрез четене.

Няма съмнение, имайки предвид изобилието на информация във формалния материал (типично текстове на програми), дори с неголям обем, с който се борави в програмирането, за важността на нагледността в тази област.

По-конкретно, за нас тя е интересна, тъй като е свързана с явността от гледна точка на визуалните аспекти на възприемането на последната. Най-общо, нагледността подпомага явността, като я прави по-непосредствено достъпна.

Дадено свойство или отношение в програмата може да се окаже явно, без да бъде нагледно в текста. Ако да речем във втория вариант от примера в началото на тази глава описанията на двата масива са на известно разстояние едно от друго в текста на програмата, едноразмерността, макар явна, не е очевидна, тъй като *достъпността* за възприемане е затруднена (вж. също „*разход на памет*“ в гл. 2). Информацията, осигуряваща явността, е *налична*, но не е нагледно *представена*. Изобщо, явността се отнася до наличността, а нагледността – до достъпността за възприемане.

Формално интересувайки се от явността, например когато я извличаме чрез програма, автоматично, нагледността не е съществена. От гледна точка на боравенето с текста на програмата чрез четене обаче е вярно обратното. В контекста на задачата за избиране на езикови средства (сред съществуващите или като изобретяваме нови), които да подпомагат осигуряване на висока явност в програмите, казаното означава, че следва да предпочитаме такива, които могат да осигурят и нагледност.

Да отбележим, че като разглеждаме отношението между явност и нагледност, се приближаваме до онзи вече споменаван тип (не)явност, при който последната се проявява чрез използването или не на правилата за допустими подразбирания в даден език за програмиране. Съдържаща се в програмата информация, която не е изразена изрично, а е резултат от подразбиране, е явна само дотолкова, доколкото разглеждаме текста именно в контекста на възможните подразбирания. Следователно в такъв случай нагледността отсъства, а явността е *условна* – според това дали текстът се разглежда сам по себе си или в контекста на подразбиранията.

Адекватността е другото свойство, което смятаме за особено важно като тясно свързано с явността.

В общия смисъл на думата адекватността означава *съответност в достатъчно висока степен* на едно нещо спрямо друго по отношение на *същностни* черти на второто. При това винаги подразбираме, че първият обект се използва като аналог или модел на втория в някакъв смисъл, т. е. понятието предполага *употребимост* на единия обект в ролята на другия.

В частност, терминът адекватност се употребява много често по отношение на езиковия модел, който дадена наука е развила за описване и изследване на някаква област от нейния интерес. Адекватността на научния език е особено важна, тъй като изразява най-същественото качество на познанието като процес на създаване, използване и усъвършенстване на модели.

В [3] чрез различни примери от математиката и други области се илюстрира важността на намирането и използването на адекватни езици за описване на същностите, съобщаване на информация между хора и формализиране на знанието. Пак там се подчертава, че самото намиране на адекватен език в никакъв случай не е формален процес: „Логиката работи перфектно, след като човечеството

е развило адекватен език. Но логиката е безсмислена, ако тя трябва да развива този адекватен език. Съвсем определено тук участва здравият разум.“

В програмирането адекватността, както я определихме в гл. 2, съхранява чертите на общото понятие и в частност – невъзможността тя да бъде осигурена по формален (автоматичен) път. Тук се налага следното уточнение. Освен за съответност между програма и абстрактен алгоритъм, можем да говорим и за такава на абстрактния алгоритъм към решаваната от него задача. При този друг вид адекватност особено ясно личи верността на казаното по-горе, но в него е заложена и повече условност, доколкото можем да избираме различни алгоритми, от различна гледна точка адекватни на задачата. Причината е, че в програмирането задачите обикновено не се поставят изчерпателно: често се пропускат например „подробности“, отнасящи се до използването на един или друг ресурс – памет, време и др. Всеки алгоритъм, бидейки еднозначно формулиран, практически уточнява решаваната задача и така се поставя в пряко съответствие с уточнената, а само в косвено – и с първоначалната задача. Така за различно уточнени задачи можем да имаме различни адекватни алгоритми, които са по различен начин адекватни на изходната задача.

Представянето на алгоритъм чрез програма на определен език е конкретизиране на алгоритъма, така че и в смисъла на това конкретизиране понятието адекватност получава известна неопределеност и субективност, но ние приемаме, че разстоянието между програмата и абстрактния алгоритъм е по-малко, отколкото това между последния и решаваната задача, и значи следва да смятаме вече определеното понятие *адекватност* за по-достъпно за наблюдаване при анализа на програмен текст.

Следва да направим и друго уточнение. Онзи, който чете дадена

програма, може да е различен от автора ѝ и да не познава абстрактния алгоритъм: последният бива възприеман косвено, именно чрез текста на програмата. В този случай е съществена степента на вярност, с която при четене на текста се възпроизвежда алгоритъмът. Доколкото последното зависи от явността, ясно е, че колкото повече *явно изразени съществени за алгоритъма* елементи има в програмата, толкова повече *обективно адекватна* е тя.

От друга страна, явни могат (и като правило – трябва) да бъдат и други елементи на програмата, които са белег на нейната конкретност, но не са съществени за алгоритъма, което, взето само по себе си, потенциално нарушава адекватността, размивайки границата между съществените за алгоритъма и другите програмни обекти. Във връзка с това да отбележим, че с повишаването на равнището на изразителност в ЕП се смалява разликата между програма и алгоритъм, т. е. участието на вторични спрямо алгоритъма елементи на програмата в текста ѝ намалява относително.

В литературата по измерване на софтуер са познати някои понятия, имащи косвено отношение към явността. Бидейки повече *количествени*, те по-лесно се формализират и оттам по-лесно се поддават на измерване, но също съдържат качествени страни.

Например в [34, 31, 8] се говори за *свързаност* (*cohesion*) на програмен фрагмент: по какъв начин и до каква степен компонентите на дадения фрагмент са съотнесени помежду си от гледна точка на фрагмента като цяло – дали общо изпълняват определено действие, или изпълняват различни действия, но следват една друга във времето, или са само косвено свързани по смисъл и т. н.

Друго разглеждано при измерване на софтуер понятие е *степен на взаимозависимост* (*coupling*) между програмни обекти, подразбирайки такива от ранга на подпрограми, като се уточняват няколко вида и степени на това свойство. Предполага се, че взаимозависи-

мостта се определя поотделно за двойки „модули“ и на тази основа – обобщено за програмата.

В [72] се предлага интегрална мярка за *взаимосвързаност*, която съвместно отчита различни характеристики – размер, управленски и даннови връзки – на програмата.

Свързаността и взаимозависимостта, отнесени към даден фрагмент, третират съответно вътрешния му строеж и връзките му с останалите части на програмата, т. е. те са вътрешна и външна мерки на структурата му. Вижда се, че във връзка с отношението към структурността изобщо има преплитане между тях и разглеждания от нас подход – изследване на явността, но последният се отличава с по-голяма конкретност по отношение на видовете структурност.

Друга важна отлика на нашия подход е, че при него не само се стремим да характеризираме съществуващи структури в зададени програми (без да можем да ги променяме), но и да оценим различни езикови средства – традиционни и други – от гледна точка на явността, която се осигурява чрез тях в програмите, и така да получим възможности за създаване на по-качествени езици и програми.

Посочените две характерни отлики са налице и при съотнасянето на споменатото интегралнометрично изследване [72] и другите подобни на него с това на явността.

4. Явност и неявност в примери: проблеми и решения

В тази глава разглеждаме някои основни видове конструкции в езиците за програмиране от гледна точка на способстването или препятстването на създаване на програми с висока явност и други благоприятни свойства, каквито формулирахме.

Наред с анализа на реализираните в един или друг ЕП или описани в публикации езикови конструкции предлагаме някои нови. Въвеждането им е с оглед на добиване на възможност за изразяване на отношения в програмите, които не могат или е неудобно да бъдат изразени с наличните досега средства.

Предлаганите конструкции не са обвързани с определен език и ако бъдат заимствани в един или друг такъв, конкретният синтаксис на всяка от тях може да бъде адаптиран. За всяка от тях това става ясно от описанието ѝ.

В разглеждането преобладават императивните езици поради това, че те са най-употребяваните от зората на програмирането и понастоящем. Доминирането на императивната парадигма е свързано с два момента. Най-напред, то говори за това, че за мнозинството от решаваните в програмирането задачи императивният стил засега се възприема като най-подходящо изразно средство. Второ, като най-разпространено, то е представено с огромен брой езици за програмиране, от широко използвани до само теоретично известни. Както разнообразната и голяма по обем практика на програмирането сама по себе си, така и, като неин концентриран израз и продължение

– съществуващите езици и научната литература по методология на програмирането и върху усъвършенстване на езиковите конструкции – осигуряват най-добрата възможна основа за изследване на интересуващите ни свойства.

При споменатата традиционност на преобладаване на императивната парадигма в специализираната литература се наброяват хиляди публикации, анализиращи популярните езикови средства – главно тези за управление на следването – и предлагащи нови, за които се предполага, в една или друга степен аргументирано, че подобряват качествата на програмния текст. Това свидетелства за нетривиалността на проблема за намиране на подходящи в един или друг смисъл езикови средства, но е вярно също, че много малко от предлаганото с доказана полезност (понякога потвърдена многократно от различни автори с допълваща се аргументация) е влязло в инструментариума на широкоразпространените ЕП. За последното има различни причини, на които няма да се спираме.

Вярно е и това, че преобладаващият научен интерес – донякъде и мода! – в областта на езиците за програмиране от години се измести към свойствата, поддържащи едромащабно проектиране и програмиране: обектноориентиран модел, модулност, компонентност, разпределеност, типови системи и др. Така езиковите средства на непосредственото програмиране се оказват донякъде загърбени откъм научен интерес. Но за това, че практическата им важност не намалява и че наличните решения не удовлетворяват свидетелства постоянното експериментиране с новопоявяващи се езици и предлагането в тях на нови конструкции.

Като особено съдържателен източник във връзка с търсенето и утвърждаването на подходящи езикови конструкции да споменем класическата днес работа [36], където се посочва необходимостта от *осъзнато прилагане* на възникващите в практиката на програми-

рането парадигми от различно естество и мащаб и *поддържането* на редица основни сред тях от страна на ЕП, както и на възможността да се градят по-сложни парадигми от по-прости. Пак там се посочват и редица конкретни примери: съвместно присвояване, размяна на стойности, цикъл със средусловие и др. (споменатите са разгледани по-долу). И отново със съжаление трябва да се каже, че изразената в [36] неудовлетвореност от ЕП във връзка със слабото поддържане на доказалите полезността си парадигми остава и днес основателна.

Разглежданите по-нататък в тази глава конструкции могат да бъдат причислени към следните езикови области:

- построяване на изрази и прости команди;
- съставни команди и най-вече такива за задаване на повторност (цикли);
- агрегиране на данни и действия;
- специфициране (метаезикови описания).

4.1. Изрази и прости команди

Команди и изрази

Прието е, че на вътрепроцедурно равнище в императивните езици се различават два основни слоя в структурата на действията: изрази и команди. Във връзка с това някои от императивните езици се разглеждат като „изразноориентирани“, с което се има предвид, че имат по-богати възможности за образуване на изрази в сравнение с останалите. Обикновено това означава, че част от средствата, които типично съдържат императивните езици под формата на команди, могат да произвеждат стойност и така да участват в образуване на изрази. С други думи, при изразноориентираните езици намалява относителният дял на „чистите“ команди за сметка на из-

разите. Примери за изразноориентирани езици са ALGOL 68 [83], ICON [39], в по-малка степен C/C++ и част от днешните динамични или сценарни (скриптови) езици.

Командите, които са и изрази, могат да имат, и обикновено имат странични последствия, като изменяне на стойността на променливи. Типично такива команди са присвояването, конструкцията за съставна команда (блок) и условните конструкции, по-рядко и цикличните.

Това, че цикличните конструкции в повечето езици не се разглеждат като изрази не е твърде добре обосновано: например смята се, че ако изпълнението на цикъл може да завърши в различни точки на тялото му, съответно произвежданата стойност би била възможно различна по тип за различните изходи и така би се нарушила типовата правилност за израз, в който е вложен цикълът. На практика обаче е възможно транслаторът да осигури типовата еднаквост за всички изходи на цикъла.

Изразната ориентираност може да бъде използвана в различни случаи за подобряване на текста на програмата откъм краткост, нагледност, адекватност и явност. Тя често се прилага например за верижно присвояване: `a := b := ... := c`. Често е уместно също да се използват изрази като `x := y + (z := 5*w)` , чийто смисъл е аналогичен на изречение на естествен език с подчинено изречение: „`x` е сборът на `y` и `z`, като последното е `5*w`“. Без влагането би трябвало да напишем две последователни присвоявания, при което името `z` се цитира два пъти, а свързаността на едното действие с другото се размива.

Както в последния пример, и по-нататък срещаме случаи на конструкции, чрез които се избягва повтаряне на информация – имена или цели изрази. Това е по начало желателно свойство при четене на програмен текст, тъй като отпада нуждата от съотнасяне на два

или повече програмни фрагмента чрез общо цитираните в тях имена. Освен това, при повторното цитиране съществува възможността някое от имената да бъде неправилно посочено, т. е. да възникне неволна грешка.

Да отбележим, че в изразяването на естествен език също е характерен стремежът към избягване на повторения, за което съществуват различни конструкции за косвено цитиране, например местоимения. Това е пример на свойство на естествените езици, на което много малко се обръща внимание с цел заимстване в ЕП. Последното е за съжаление: ако създателите на езици за програмиране се вглеждат по-внимателно в структурите и тенденциите на използване на естествените езици, те биха намерили там потенциал за плодотворно обогатяване на ЕП.

От друга страна, в горния пример присвояването на `z` изглежда подчинено на това на `x` и би трябвало да използваме този запис, ако между двете действия наистина има такава връзка по смисъла на програмата. Ако е желателно да се придаде симетричност на двете присвоявания (тъй като самите те или двете променливи имат еднаква важност), може да се предпочете обичайният или друг различен запис.

Съществено за средствата, които предоставя изразната ориентираност е, че тя създава възможности за избор между различни конструкции с еднаква функционалност в смисъл на резултата от извършване на действията, но отличаващи се по други качества. Така тя не ограничава обичайния стил на императивното програмиране, а го допълва.

Характерни примери за използване на изразна ориентираност са също

```
a := if P then E else F
```

където `P` е условен израз, а `E` и `F` са произволни изрази от еднакъв

или съвместим тип, съвместим и с този на `a`, и

```
(if P then x else y) := E
```

където или `x`, или `y` получава стойността на `E`, според тази на `P`.

В тези два примера, аналогично на дадения по-горе, се получава кратък, адекватен изказ на съответното действие без повтаряне на информация.

Обмен и съвместно присвояване

Предвид интензивното боравене с променливи, променящи стойностите си, което е характерно за императивния стил, обменът на стойности между две променливи е често срещано действие. Тъй като обаче в повечето езици за него не е предвидена конструкция, използват се записи като следния:

```
t := x;  x := y;  y := t
```

които имат ред недостатъци: такава размяна съдържа три операции вместо една, не е нагледна и в значителна степен е неявна, всяка от разменящите стойностите си променливи се цитира по два пъти, както е и с помощната променлива `t`, която често се и въвежда само заради размяната. Твърде лесно е при записване на такава размяна да се допусне грешка.

Icon е от малкото езици, които разполагат с операция за размяна; в него горният пример се изразява с `x :=: y`.

Друго често срещано, но сравнително рядко предоставяно в езиците действие е съвместното, наричано още „успоредно“ присвояване. Един от ранните езици, където то е реализирано, е Clu [57], а днес успоредното присвояване става все по-разпространено. Предаването на съвместно присвояване чрез няколко единични присвоявания и помощни променливи има същите недостатъци, както при имитирането на размяна, при това колкото повече стойности се присвояват съвместно, толкова повече помощни променливи са

нужни. Например вместо

```
x,y,z := x+y+z, x+y-z, x-y-z
```

се налага да се прибягва до

```
t1 := x+y+z;
t2 := x+y-z;
z := x-y-z;
x := t1;
y := t2;
```

Размяната може да се изрази чрез съвместно присвояване, например `x,y := y,x`, но е по-добре да са налице и двете средства и да се използват всяко в съответния случай.

Присвоявания и синонимия

Синонимията е цитиране на даннов обект с повече от едно имена и/или указатели. С наличието на синонимия се създава възможност дадена стойност да бъде променяна с помощта на един от синонимите, а да се потребява чрез друг.

В различни случаи синонимията се оказва удобно или дори трудно заобиколимо средство. Така е например при работа със сложни даннови съвкупности с много връзки между елементите. Но като правило синонимията създава висока *неявност* в текстовете на програми, още повече, че в много случаи се проявява по време на изпълнение: текстът на програмата се оказва слабо съответен на динамичната ѝ структура.

Синонимията се проявява по няколко начина. Един от тях е на дадена подпрограма да се предаде един и същ аргумент „по адрес“ към два различни нейни формални параметъра. Аналогично, един и същ обект може да се окаже достъпен като параметър „по адрес“ и глобална променлива. Някои езици, напр. EUCLID [54] и OCCAM [66] преднамерено не позволяват синонимия и в частност споменатите

две форми.

Явното използване на указатели в езици като C е очевидно средство за допускане на синонимия, но съществуват и скрити форми на указатели. Такива са изричните синоними (references) в C++ и „цитиращите“ присвоявания в множество съвременни езици, в които при присвояване се копира указател към съответната стойност, а не самата тя. Така две имена на променливи могат да цитират един и същ даннов обект, както и даден обект, освен самостоятелно именуван, да се окаже и цитиран като елемент на агрегатна данна – масив, смес или друго.

Макар че при програмиране на език с такъв вид присвояване се добиват специфични навици, помагащи да се избягват отрицателните последствия на синонимията, данновата структура на програмата остава по принцип в голяма степен неявна.

Присвояващи операции

Така наречените „присвояващи операции“ (*assigning operations* или *compound assignments*), които съчетават аритметична или друга операция с присвояване, са добре познати от широката им употреба в езика C и производните му, но всъщност са въведени още в Algol 68, а някои подобни конструкции се срещат и в COBOL. Популярни езици като Icon също разполагат с такива операции.

На C и подобните му например често се записват изрази като

```
a += b
```

вместо формално еквивалентния израз

```
a = a+b  ,
```

като подобно съвместяване с присвояване се допуска за повечето двуместни операции в езика.

Съществува мнение, че присвояващите операции са въведени и се използват за облекчаване на превеждането в машинен език на

програмата: за да се подскаже поставянето на съответна акумулаторна команда за дадения процесор. Не можем да твърдим дали такъв или не е бил мотивът за *появяването* им, но със сигурност *използването* им подобрява адекватността и явността на текста на програмата.

Горният запис съответства на израза „увеличи стойността на `a` с тази на `b`“, описващ точно връзката между двете променливи в действието, вместо по-неестественото „събери `a` с `b` и постави резултата в `a`“. Освен това, името на променливата, която получава стойност, се цитира само веднъж, което прави записа неизлишъчен, а това, както споменахме и по други поводи, на свой ред намалява възможността да се допускат грешки и при въвеждане на текста, и при четене (което е съществено при дълги и подобни едно на друго имена).

От друга страна, в C и другаде присвояващи операции са допустими не за всички двуместни операции в езика [48]. Този недостатък е избягнат при Icon, но и в единия, и в другия езици присвояването не може да се съвместява с никоя *едноместна* операция. Последното принуждава в програмите да се използват изрази като

```
a = -a
```

за обръщане на знака на стойността на променлива и

```
a = !a
```

за обръщане на булева стойност.

Поради посочените по-горе съображения е очевидно, че и по отношение на едноместните операции се нуждаем от същата адекватност и неизлишъчност.

В C са реализирани и две специални едноместни операции `--` и `++`, съответно за намаляване и за увеличаване с 1 на стойността на променлива, които следователно са аналогични на присвояващите двуместни операции. По отношение на тях обаче `--` и `++` имат

следната особеност: единствени те в езика допускат префиксна и суфиксна спрямо аргумента си форми на записване, което влече различна стойност на образувания израз. Например, ако `i` има стойност `5`, както `++i`, така и `i++` изменят `i` на `6`, но стойността на първия израз е `6`, а на втория – `5`, което е съществено, когато дадената операция участва в по-сложен израз: съответно `5*++i` ще бъде `30`, а `5*i++` – `25`.

Сходствата и разликите между двуместните присвояващи операции и автонарастващата и автонамаляваща едноместни операции подсказват обобщението, което предлагаме тук.

Преди всичко, въвеждаме възможността *всяка* операция – едноместна или двуместна – да се съвместява с присвояване. Наред с това, осигуряваме предидействието или последействието на изменянето на стойността на съответната променлива в съвместена с присвояване операция да може да се избира *за всяка* операция, т. е. то да става независимо от конкретната операция. С други думи, съвместяването с присвояване става „*ортогонално*“ свойство спрямо множеството на операциите, и такова е също преди-/последействието спрямо съвместяването.

Заедно с казаното въвеждаме и еднообразно префиксно записване на всички едноместни операции спрямо аргументите им. Именно, като използваме знака `:` за придаване на присвояващо действие на операциите и записваме `:` като префикс или суфикс към знака на операцията за предизвикване съответно на преди- и на последействие, даденият по-горе пример на C може да се преобразува в

```
a :+ b
```

или в

```
a +: b
```

но стойността на първия израз е *вече изменената* стойност на `a`

(предидействие), а на втория – тази *отпреди изменението* (последействие). В това отношение първият вариант е по-близък до неговия аналог в C.

За едноместните операции получаваме изрази от вида

```
:-x
```

където `x` обръща знака си и стойността на израза е тази вече променена стойност, или

```
-:x
```

което се отличава от горното по това, че стойността на израза е първоначалната стойност на `x`. Следователно

```
r = :-a    и    r = -:a
```

отговарят съответно на

```
r = a = -a    и    r = a,  a = -a
```

Още едно обобщение на присвояващите операции е това, че *всяка* такава операция има за резултат (заимствайки от терминологията на CPL, BCPL и C) L-стойност, т. е. обект, който има стойност, но и на който може да се извършва присвояване (при което стойността се променя). В случая това става, като с всеки такъв израз се *асоциира променлива* – тази, на която присвояващата операция присвоява стойност – и така става възможно той да бъде субект на ново присвояване.

Всяка L-стойност може да се употребява в присвоявания, навсякъде напълно аналогично на променлива, като се подразбира, че всъщност присвояването се отнася до асоциираната с израза променлива. В частност, това означава, че израз с L-стойност може да бъде аргумент в друга присвояваща операция, включително и на мястото на асоциираната с операцията променлива.

Всеки израз, стойността на който е L-стойност, наричаме L-израз. Така се обособява множество от произволно сложни L-изрази, вло-

жено в това на обичайните изрази.

Например

```
a :* 5 :+ b
```

съответства на израза `a *= 5, a += b` в C и означава „да се умножи `a` с `5` и след това да се увеличи със стойността на `b`“.

Да отбележим, че израз като

```
(a *= 5) += b
```

който в C би бил равнозначен на горния, не е възможен, тъй като там присвояващите операции не са L-изрази. Той е възможен в C++, D и Algol 68 – един от предшествениците на C! (В C++ L-стойности произвеждат и операциите `++` и `--` .) Никой от тези езици не предлага вариант с последействие на присвояването.

Едно интересно и полезно следствие на въведеното разширение на понятието присвояваща операция е, че обикновеното присвояване (`=`), също разглеждано като операция, подобно на останалите може да бъде снабдено с префикс или суфикс `:`, като с това се получава съответно предидействие и последействие на присвояването.[3] По този начин можем да съставяме изрази като

```
a := x+y      и      a =: x+y
```

И двата израза, както и `a = x+y`, присвояват `x+y` на `a`, но се различават по *стойността* си – съответно новата и първоначалната стойност на `a`. От втория израз можем да получим

```
b = a =: x+y
```

което е съкращение на еквивалентния C-израз `b = a, a = x+y`. Между другото получаваме и още една, но несиметрична, форма на размяна на стойности (операцията `=` има обичайната за нея дясна съдружителност):

[3]При това предполагаме, че сама по себе си операцията `=`, както всяка друга, *не е* L-израз – тя извършва присвояване, но не асоциира променлива.

```
b = a =: b
```
или
```
a = b =: a
```
.

Посредством L-изрази могат да се записват „механизми“ за пораждане на различни числови и други редици. Например

```
t :+ (s =: t)
```

е израз, в който стойността на `t` се копира в `s`, след което *предишната* стойност на `s` се добавя към тази на `t`. Така получената нова стойност на `t` става стойност и на израза. Ако дадем на двете променливи начални стойности например както следва

```
s = 1, t = 0
```

и подложим този израз на многократно пресмятане, последователно получаваните от него стойности образуват редицата на Фибоначи.

Така въведените присвояващи операции и L-изрази са гъвкаво изразно средство, допълващо обичайния езиков механизъм, включващ изрази и присвоявания на стойности. Полезността му за повишаване на адекватността и явността е подобна на тази на сходните споменати средства в C, Icon и др., но е по-голяма, тъй като нашата конструкция е и по-обща, и семантично по-проста.

Наред с казаното, начинът и степента на използване на такива изрази трябва да се определят във всеки отделен случай с участието на „здравия разум“. Претоварването на даден израз с присвоявания, ако се отнасят до различни променливи, води до голям брой странични за израза последствия и влошава търсените параметри на възприемане на текста.

Условноприсвояващи операции

Начинът и съображенията за въвеждане на L-изразите в горния подраздел ни подсещат за още една форма за обогатяване на понятието израз. Именно, тук описваме *условноприсвояващите операции*. Получаващите се от тях изрази, по аналогия с L-изразите, нарича-

ме Q-изрази. Q-изразите са и синтактично подобни на L-изразите – всяка условноприсвояваща операция се записва подобно на присвояваща, като вместо `:` поставяме `?`.

Условноприсвояващата операция извършва *условно присвояване* – според верността или неверността на дадено условие. Условието се задава от израза, в който е вложена условноприсвояващата операция, като предполагаме, че той има булева стойност или за по-голяма свобода тълкуваме стойността в духа на C и някои други езици: нулевата стойност е „неистина“, а всички други – „истина“.

Тъй като и условноприсвояващите операции биват или предидействени, или последействени, изпълняването на условието е в потенциална зависимост или от текущата стойност на съответната променлива, или от тази, която условно ѝ се присвоява.

Условноприсвояващите операции имат същата общност по отношение на множеството от обичайните операции в езика, каквато имат присвояващите.

Така например

```
a ?+ i < 80
```

увеличава стойността на `a` с `i`, ако получаващата се при това нова стойност (присвояването е с предидействие) е по-малка от `80`, което е приблизително същото по резултат като `if (a+i<80) a += i;` на C, но последното е команда, докато първото е израз. Още едно, и то по-съществено предимство на този Q-израз пред командата на C е, че операцията (както имената на двете променливи, така и знакът `+`) се изписва само веднъж.

Още два примера:

```
i -? 1 > 0
```

намалява стойността на `i` с `1`, ако тя текущо е положителна, а

```
i ?- 1 > 0
```

прави това, ако при текущата стойност на променливата е вярно, че `i-1` е положително, т. е. ако `i` е по-голямо от `1`.

Следващият пример присвоява `b` на `a`, ако `a` текущо има нулева стойност:

```
(a =? b) == 0
```

Същото присвояване прави

```
(a ?= b) == 0 ,
```

но когато стойността на `b` не е нула, т. е. ако `a` след присвояването добива ненулева стойност, независимо от текущата ѝ. Накрая, и двата израза

```
(a =? (b =? a)) != 0     и     (b ?= (a =? b)) != 0
```

разменят стойностите на `a` и `b`, ако `a` текущо не е нула. Разбира се, този начин за размяна не се отличава с добра нагледност и адекватност.

Общите съображения за полезността и начина на използване на условноприсвояващите операции са подобни на тези за присвояващите операции и няма да ги повтаряме. И двете обслужват често срещани в практиката на програмирането, типични случаи.

Да отбележим, че аналог на тук въведените условноприсвояващи операции, но ограничен само до *условно присвояване* (и без предиили последействие), се среща в езика Icon. Това условно присвояване се извършва в зависимост не от булева или аритметична стойност, а от *успешността* на даден израз. Успешността, като успех или неуспех на пресмятането, е присъща на всеки израз в Icon. В съчетание със също характерния за този език *генераторен стил*, условните присвоявания се разглеждат като прост механизъм за възвратно програмиране (*backtracking*) [39]. Условноприсвояващите операции имат ограничено приложение за такава цел, но в други отношения са очевидно по-общи.

Структура на изразите във функционални езици за програмиране

Изразите в програмирането са много богат обект на разглеждане от гледна точка и на синтаксиса, и на семантиката им. Това може да се обясни с обикновено многото, сравнено с математиката, на брой и вид участващи в тях операции, техните различни свойства и взаимодействията между операциите в изрази – правила за предшестване и др. В програмирането изразите са и много по-строго формализирани, отколкото в математиката.

Едно чисто информатично откритие например е обратният полски, или суфиксен запис (докато *правият* или префиксен е изобретен за запис на формули в логиката). Имайки всички достойнства на префиксния запис, суфиксният се оказва много по-удобен за непосредствено реализиране чрез компютър, а и по-нагледен. Нагледността се обяснява чрез следното наблюдение.

Да разгледаме много прост функционален език, в който всяка функция има само един аргумент, а изразите се образуват само от действието „прилагане на функция към аргумент“. Приемаме, че пишем и четем отляво надясно.[4] Приемането означава, че естественият ред на извършване на действията е отляво надясно. Можем да си представяме пресмятането на израз като поток от последователни преобразования на дадена стойност докато тя се превърне в окончателния резултат.

Минималният запис на прилагане на функция `f` към аргумент `a` включва само посочване на функцията и аргумента ѝ и значи има вида `f a` (префиксен) или `a f` (суфиксен). Доколко нагледен е единият и другият запис?

Да разгледаме най-напред префиксния.

Всеки нетривиален израз съдържа вложено прилагане, например

[4] Това самò по себе си е, разбира се, само културно различие – въпрос на навик.

прилагаме функция `g` към `f a`. Това може да се запише във вида

```
g (f a)      или      g f a
```

В първия вариант прибягваме до скоби, и колкото повече влагания на обръщения към функции имаме, толкова повече стават и скобите, което, като усложнява записа на израза, влошава нагледността.

Вторият вариант предполага подразбиране на предимство на най-дясното възможно прилагане в израз. Такова подразбиране обаче е в несъгласие с четенето отляво надясно.

Всъщност тъй или инак и двата варианта на запис изискват да прилагаме функциите в ред *отдясно наляво* – обратен на избраната посока на четене.

Ако решим да удължим израза с прилагане на още една функция, тя също ще трябва да се добави *в началото* вместо, както диктува посоката на четене, в края на израза. А и ако следваме първия вариант на запис, целият остатъчен израз ще трябва да се загради в скоби.

Впечатлението за несъответствие на префиксния запис с посоката на четене се усилва, ако разгледаме и сигнатури на функциите – означения за областта и кообластта на всяка. Нека например $f : A \to B$ – това е общоприетият запис и в него посоката на стрелката отговаря на тази на четене. Но ако заместим в прилагането `f a` функцията `f` със сигнатурата ѝ, а `a` с нейната област A, получаваме сигнатура на израза:

$$(A \to B)\ A.$$

Изразът `f a` е допустим, ако сигнатурата му е правилно образувана, т. е. ако областта на `f` съвпада с тази на аргумента `a`. Полученият сигнатурен израз би бил нагледен, ако в него означенията на тези две области са съседни, по този начин пряко показвайки приложимостта на функцията. В случая това не е така: двете A се разминават по място.

Толкова повече, ако $g : B \to C$, сигнатурата на израза `g(f a)` е

$$(B \to C)\,(A \to B)\;A$$

– тук се разминават не само означенията на областите на `a` и `f`, а и тези на кообластта на `f` и областта на `g`.

Разглеждането на суфиксния запис показва, че никое от посочените нарушаващи нагледността несъответствия не му е присъщо. Именно, вложени прилагания се записват като `a f g`, `a f g h` и т. н. Редът на действията е в съгласие с посоката на четене и то без да се уточнява чрез скоби. Прилагането на нова функция към резултата се задава с естествено добавяне в края на израза. Сигнатурите на изразите `a f`, `a f g` и пр. са

$$A\;(A \to B),$$

$$A\;(A \to B)\;(B \to C)$$

и т. н. – максимално нагледни, вкл. стрелките в тях отговарят на посоката, в която се преобразува стойността в израза.

Понастоящем суфиксният запис се използва в малко на брой езици, например Forth, PostScript и Factor, известни под общото име *конкатенативни езици*.

Да обърнем внимание, че тежестта на съображенията за нагледност, например като тези в горното разглеждане, е голяма именно в програмирането, където изразите са по-сложни, по-разнообразни откъм участващи в тях действия, не се радват на типографско акцентиране като в математиката, а наред с това, отново за разлика от математиката, винаги се подлагат на строго формална интерпретация.

Функционалните езици са особено интересен обект на разглеждане, тъй като при тях програмите се състоят почти изцяло от изрази.

В някои от тях нагледността и явността на текста са чувствително по-ниски от обичайното. Това се случва по различни причини. Добре известен пример е езикът Lisp, в който четенето се затруднява от еднообразието на синтаксиса. Всички конструкции – обръщения към функции, задаване на параметри в определения, разнообразни команди, части от тях, описания на данни и пр. – имат един и същ външен вид: изброяване в скоби.

Друг пример е езикът REBOL. Прилагането на функция в него е префиксно и не е свързано с особен синтаксис, функциите могат да имат различен брой аргументи, а типизирането е динамично, така че кое да е име в различни точки и моменти на програмата може да обозначава различни стойности, включително функции. За да се разчете например изразът `a b c d e`, който може да съдържа няколко обръщения към функции, не е достатъчно да се знае, че предимство в реда на прилагане има най-дясната функция. Самият строеж на израза е негово *динамично* свойство и зависи от текущите стойности на участващите имена. Например `b` може да бъде функция с два аргумента. Те може да се окажат `c` и `d`, но ако `c` е функция, да кажем с един аргумент, и `d` е нефункционна стойност, изразът е равнозначен на `a (b (c d) e)`. При други стойности на променливите тълкуването на израза може да бъде друго. Така дори синтактичният строеж на програмата е донякъде неявен.

Езикът J борави с функции само с един или два аргумента и има удобно правило за различаване дали прилагането на дадено име е от единия или от другия вид. В този език обаче за прилагането на функции от втори ред – функции над функции – има различни от тези за „простите“ функции синтактични правила. Понеже и в J типизирането е динамично, такова се оказва и тълкуването, включително на синтаксиса, на кой да е израз. Редица от три лексеми – а програмите на J са тъкмо редици от интерпретируеми лексеми – може да

има до десет (!) различни синтактични интерпретации в зависимост от стоящите зад лексемите стойности. Това е много висока степен на синтактична неявност.

Повечето съвременни функционални езици предлагат разнообразен и изразителен синтаксис, а и по-богата семантика. Вече беше посочено, че например декларативният стил на изразяване е много благоприятен и за явността, и за нагледността на програмите на такива езици. Определянето на функция чрез съпоставяне с шаблони, като една от формите на декларативност, дава възможност в явен вид да се изрази съответствието между строежа на един или друг конкретен аргумент и пресмятанията, които извършва функцията с него.

Изразите във функционалния език U

Оригинално решение за подобряване на нагледността и явността в изрази се съдържа в предложения от автора [19, 2] функционален език за програмиране U. Подобно на споменатите J, REBOL и др., U е също динамично типизиран и изразите в него в известен смисъл също са синтактично еднообразни. Еднообразието обаче, за разлика от това в Lisp или J, е по-скоро в смисъл на *регулярност*.

Образуването и пресмятането на изрази в U е подчинено на няколко особености, с които този език се различава от останалите.

Всеки израз е поредица от елементи. Пресмятането на израз се състои в добиване на стойностите на елементите и извършване с тях на действия. Във всяко действие участват три последователни стойности $u\ v\ w$, а действието се състои в прилагане на функцията v към u и w. Следователно v трябва да бъде функционна стойност и тъй като в случая тя бива прилагана, наричаме я *операция*. Стойностите u и w също може да са функции, но те не се прилагат, а служат за аргументи; това дали са функции е безразлично.

Така понятието *функция*, наред с други като число, низ и пр., се отнася до вида на една или друга стойност сама по себе си, а понятията *операция* и *аргумент* – до ролята на стойност в израз. При това ролята се определя строго от поредното място на стойността в израза. Например в израза `f+g` стойността `+` се прилага като операция, а в `*h+` и `+`, и `*`, макар и функционни стойности, са аргументи.

Стойностите на елементите на израз се пресмятат строго последователно. За всяка така добита тройка стойности $u\ v\ w$ се извършва прилагане на операцията и получената от това стойност се разглежда като стойност на нов елемент на израза на мястото на изходните три: пресмятането на израза продължава по-нататък с нейно участие. Например в израза `a b c d e` най-напред се пресмята `a b c`, а получената стойност образува нова тройка, все едно че имаме `(a b c) d e`.

Подизразите в скоби се пресмятат като самостоятелни изрази, така че с тяхна помощ можем да променим реда на пресмятане, например за горния израз – на `a (b c d) e` или `a b (c d e)`.

Поради описаното правило различаването между операции и аргументи става чрез просто синтактично правило – поредност, а всяко прилагане на операция е инфиксно.

Въпреки инфиксния вид на прилагането на операции, всяка функция в U има всъщност точно един аргумент. За да приема дадена функция две или повече стойности, образуваме от тях редица: аргументът на функцията е тази редица, а нейните членове условно приемаме за отделни аргументи на функцията.

Основният начин да пресметнем функция f за каква да е стойност x е да приложим операцията `.` към f и x, записвайки $f\,.\,x$. Функцията `.` също има единствен аргумент, а горното всъщност означава, че този аргумент е редицата $[f\,;x]$.

В смисъла на казаното всяко прилагане $u\ v\ w$ (на v към u и

w) е всъщност прилагане на v към редицата $[u;w]$, т. е. $u\ v\ w$ е равнозначно на $v\ .\ [u;w]$.

Така на разположение са два стила на запис. При първия операцията е `.`, като в $f\,.\,x$, и се разглежда като „префиксен“, в смисъл на прилагане на f към x. При другия стил функцията се поставя между „два аргумента“ (всъщност два члена на редица-аргумент) и затова го възприемаме като инфиксен. В действителност инфиксни по форма са и двата стила, а разграничаването им е само условно. Изборът между единия и другия вид запис се прави според това, кой за даден случай смятаме за по-удобен.

С помощта на операцията `=>`, която е като `.`, но с обърнат ред на аргументите (`f.x` $\equiv$ `x=>f`) може да се имитира и суфиксен запис.

Описаният механизъм за образуване и пресмятане на изрази съдържа достойнствата на практически всички налични в езиците за програмиране записни системи. Той е възможно най-прост и в частност напълно регулярен, подчинен на само едно правило. Наред с това, той позволява придържане към, или имитиране на, всяка от основните форми на запис – инфиксна, префиксна и суфиксна, както намери за удобно програмистът и с минимална лексикална излишъчност.

Простотата и гъвкавостта осигуряват отлична нагледност, а поради регулярността на синтаксиса, неявността, за разлика от J, REBOL и др., е много по-малка: тя се обуславя от динамиката на съответствията между имена и стойности и полисемичността на много от вградените в езика U функции, но не се проявява в нееднозначност на синтактичното интерпретиране.

Съюзи

Някои езици за програмиране предлагат операции, образуващи стойност чрез избор на една от две стойности, без да извършват

пресмятане над тези стойности и изобщо без да ги използват. Пример за такава операция е `,` (запетая) в езика C: изразът e_1`,`e_2 има стойността на e_2, но при това се пресмятат и e_1, и e_2, именно в този ред. Характерно е и, че това е една от много малкото на брой операции в езика, за която редът на пресмятане на аргументите е определен.

Такава операция има смисъл само ако e_1, наред с произвеждането на стойност, има един или друг вид странично последствие – най-често присвояване, при което така променените величина или величини се използват за пресмятането на e_2. Това, че операцията фиксира поредността на пресмятане на двата си аргумента, може да се използва и другояче, като страничното последствие на e_1 е външно спрямо програмата, например визуално (печат, рисунка), звук или движение. Във всички случаи съществено е и образуването на стойност (e_2).

Операция със същия смисъл е `before` в езика ML. APL има две аналогични операции: $\vdash$ и $\dashv$, които дават стойността съответно на десния и левия си аргумент. Същите операции в J са `]` и `[`.

Понеже при операция като тези от една страна се образува стойност, а от друга – това не става чрез каквото и да е пресмятане и значи действието на операцията фактически не зависи от аргументите ѝ, уместно е да имаме особено име за такъв тип операции. Подходящ термин, по аналогия с естествения език, е „съюз“.[5]

Споменатите досега съюзи не обръщат внимание не само на стойностите на аргументите си, но и на каквото и да е друго, свързано с пресмятането им, освен неговия завършек. При наличие на подходящи странични последствия от пресмятането на аргументите обаче съюзът може да се възползва от тях. В езика Icon, където произвеждането на стойност от израз може да е съпроводено с различни

[5] Думата *съюз* се използва в езика J с различен смисъл.

допълнителни свойства, в частност присъства понятието успешност: при пресмятането си даден израз успява или не. В този език операциите `&` и `|` са аналогични на логическите връзки „и“ и „или“, но използват именно сигнала за успех (а не стойността) на израз: например e_2 в израза e_1`&`e_2 се пресмята само ако преди това пресмятането на e_1 успее. Благодарение на това `&` и `|`, макар да не са наистина съюзи, могат и често се използват именно като такива.

Направеното наблюдение подсказва възможността да потърсим и други варианти на съюзно съотнасяне. В [15] е предложено множество от действия, за което става дума и в следващия раздел, които имат свойството да се „самоунищожават“ в смисъл, че при повторно изпълнение могат да се окажат неактивни, все едно, че не съществуват повече. Сигналът за продължаване или прекратяване на съществуване е странично (по отношение на произвеждането на стойност) последствие, на основата на което можем да образуваме нови съюзи, например със следното действие [16] (e_1 и e_2 са изрази – аргументите на съюз):

- пресмята се e_1 и ако той остава неунищожен, пресмята се и e_2, като за резултат се взима стойността на e_1;
- като горното, но за резултат се взима стойността на e_2;
- пресмята се e_1; ако при това той остава съхранен, неговата стойност е и стойността на целия израз, а ако се окаже унищожен, пресмята се и e_2 и за резултат се взима неговата стойност.

Във всички случаи при повторно пресмятане остават или e_1 и e_2 със съответния съюз, или само e_2, или само e_1, или никое от тях.

Използването на два и повече такива или от различен вид съюзи в един израз съдържа потенциал за изразяване на много разнообразни пресмятания. Тук обаче няма да се спираме повече на това, а по-общите *консумативни команди* разглеждаме по-нататък.

4.2. Съставни команди

Езиковите средства за управляване на следването на командите, наричани за краткост управляващи команди, са една от най-разисканите теми във връзка с адекватността, явността, нагледността и други черти на възприемане на текста. Особено спорни се оказват средствата за изразяване на циклично изпълнение.

По отношение на управляващите структури често се пренебрегва разликата между *формална достатъчност* и *практическа адекватност*. Това е една от причините за очевидното разминаване на наличните в съществуващите ЕП средства с доказано необходимите. Например в последователните варианти на определение на езика Oberon [86, 87, 63, 88] някои команди за циклично изпълнение се добавят, а после премахват, други такива отначало се премахват (спрямо по-ранния език Modula-2), а по-късно се добавят отново, в очевидно нестихващо колебание кое да се избере. При това всяко отстраняване е със съображенията за функционална достатъчност на наличните средства, без изобщо да се взима предвид тяхната практическа адекватност и фактическа изразителност.

Много изследвания са посветени на класифициране на програмите според съдържащите се в тях управляващи структури и на формални методи за преобразуване на едни програми в други, „по-структурни“ (вж. напр. [69, 56]). Известно е също огромното влияние на [26][6] и изобщо работите на E. Dijkstra, C. Hoare и N. Wirth и други пионери на информатиката върху оформяне на представите за структурна програма.

При всички положителни страни на утвърдилото се разбиране за необходимостта да се избягва безразборното, хаотично подреждане на действията в програмата, като вместо това се използват малко на брой лесно обозрими конструкции, нерядко това разбиране е из-

[6] Малко известен е фактът, че подобна тема е третирана и по-рано, в [64].

мествано от безсмислената сама за себе си доктрина за избягване на прословутата команда `goto` и от също така безсмисленото стриктно придържане към точно определени управляващи команди.

Междувременно е многократно показано, че сведеният до функционален минимум набор от изразни средства за управление е неадекватен на реалните задачи, тъй като често води до изкуствени построения, с голям брой променливи – въведени само за обслужване на управлението, но нямащи нищо общо със същината на решаваната задача – или повтаряне на части от програмата.

Да разгледаме двата основни типа управляващи конструкции: тези за условно изпълнение и избор и конструкциите за цикъл.

Условно изпълнение и избор

Повечето (императивни) езици за програмиране предоставят условна команда (`if`), в която има една или две подчинени на условието части: незадължителната втора част се изпълнява при стойност „неистина“ на условието.

По същество обаче това са по-скоро две различни действия. Условната команда с един клон съответства на *условно изпълнение* на съответното действие, т. е. тя задава изпълнение в зависимост от верността на даден предикат, докато командата с два клона е *избор* на една от две алтернативи. Втората форма е следователно по-близка по смисъл до по-общото действие „избор на една от няколко възможности“, отколкото до условно изпълнение.

Принципното различие между двете действия изпъква още повече, като разгледаме контекстната им приложимост. Именно, нека командата се изпълнява в среда, в която от нея изрично се очаква определено последствие, например произвеждане на стойност както в изразноориентиран език. Командата за условен избор може да посочи две (или повече) различни последствия, при това тяхната

еднотипност може да бъде осигурена чрез подходящо задължаващо правило в определението на езика. Командата за условно изпълнение обаче е неприложима, защото ако условието се окаже неизпълнено тя не може да осигури последствие, тъй като и не изпълнява никакво действие.

Следователно представянето на двете различни условни действия като варианти на една конструкция, макар и много широко практикувано в езиците, е концептуално погрешно. Стремежът към адекватно изразяване изисква те да се обособят в различни езикови форми.

Посоченото разделяне не само обогатява и прецизира използвания език, а заедно с това отстранява известния проблем в някои езици с влагането на условни команди – този за съответствието между условията и подчинените им действия (вж. напр. [11]) – възникващ именно когато някоя от вложените условни команди има само един клон.

В някои езици въведеното тук разделяне е частично реализирано. Повечето варианти на Lisp [77] предлагат фòрмата `when`, която е именно условно изпълнение (не избор), но заедно с това фòрмата `if` може да има както два, така и само един клон, така че езикът само прави възможен, но не налага избора на адекватната конструкция – той зависи от програмиста. Подобно е и решението в езика BCPL [71].

В други езици, например конкатенативните Forth и PostScript, разделянето между условни изпълнение и избор е налице, защото обединеното представяне с една команда би довело до невъзможност за еднозначно интерпретиране на програмата. Подобно е и при REBOL, функционален език с почти отсъстващ синтаксис, тъй като там всъщност и условните, и други действия се задават чрез функции, а всяка функция трябва да има фиксиран брой аргументи.

Конструкции за цикъл

Сред управляващите конструкции тези за задаване на цикли са исторически най-значително изменяли се. Те са и най-много обсъждани в литературата. Някои от причините за това са потенциално високата семантична сложност на този вид конструкции, разнообразието от задачи, за решаването на които се налага те да се използват, и понякога сложното взаимодействие между елементите на такава конструкция и другите части на програмата.

Интересно е да се проследи еволюцията на развитието на конструкциите за цикъл по присъствието им в основните ЕП.

Командата за „цикъл с брояч“ – изброител на аритметична прогресия – с какъвто единствено разполага FORTRAN, в ALGOL 60 преминава в команда, съчетаваща в управлението на повторенията използването на брояч и на (пред)условие, при това задаването на условие е незадължително, но броячът е неотменен елемент.

В PL/I и ALGOL 68 командата за цикъл също е единствена, с по няколко форми, но вече използването на брояч не е задължително, а само се *допуска*. Всъщност с това цикълът се разпада на две основни, съществено *различни форми* – с брояч и с условие – макар да се допуска съвместното им използване. Окончателното обособяване на тези основни форми като *различни команди* става в ALGOL W, PASCAL и други следващи езици, където те се появяват като отделни, *несъвместими* помежду си конструкции. Тази традиция следват и повечето разпространени днес езици.

Циклите с условие са популярни в ЕП в две форми: с пред- и следусловие, като практиката показва, че втората се използва значително по-рядко.

Циклите с брояч, както са представени в някои ЕП, например PASCAL [68], ADA [10] и подобни на тях, имат характерни ограничения: те са твърде тясно обвързани именно с аритметично броене;

често типът на управляващата променлива е задължително да бъде цял или изброим; начинът на изменянето ѝ най-често се свежда до нарастване или намаляване (или дори само до нарастване) с 1 и др. Тези ограничения нерядко се проявяват като недостиг на изразни средства при програмиране на съответния език.

От друга страна, традиционните команди за цикли, управлявани от условие, имат следния съществен недостатък: в тях са обособени само две части – условието и подчиненото нему тяло. Ако съпоставим структурно този вид команди с тези за цикли „с броене“ забелязваме, че последните притежават и други две, или всичко четири обособени части:

- инициализиращо действие,
- управляващо условие,
- тяло и
- реинициализиращо действие.

Инициализиращата част има за цел да подготви начална стойност – включително, ако е нужно, да определи съответната променлива – както за пресмятане на условието, така и за изпълнение на тялото. Реинициализиращата част обновява стойността на управляващата променлива преди всяко следващо пресмятане на условието и евентуално изпълнение на тялото. Условието в случая е сравнение за достигане от управляващата променлива на определена гранична стойност.

Не е трудно да съобразим, че инициализиращо и реинициализиращо действия са на практика присъщи на всеки итеративен процес, без той да е непременно свързан с броене. Следователно отсъствието на такива дялове при формите за т. нар. условни цикли е езиков недостатък, водещ до много съществено снижаване тъкмо на явността на изразяване в програмите.

За илюстрация на това да разгледаме как се изразяват две сход-

ни действия – обхождане на масив `a[]` и на свързан списък с начало `s` – на езика PASCAL (подобно би било на ADA, OBERON и др.).

```
{ масив }
for i := 1 to n do begin
  ...a[i]...
end
```

```
{ списък }
p := s;
while p<>nil do begin
  ...p^...
  p := p^.next
end;
```

Преди всичко, да обърнем внимание, че и двете действия са типични за употребата на езика и че те наистина са съдържателно много близки: и двете са последователно обхождане на линейна структура. Реализацията им обаче е твърде различна и това е принудително: цикълът с брояч не може да се използва за изброяване на списък, а и в цикъла с предусловие не могат да се обособят явно инициализиращото и реинициализиращото действия. Тази различност сама по себе си е признак за неадекватна изразителност на езика: не е налице еднообразност в смисъла, в който въведохме това понятие в гл. 2.

По-нататък, при обхождането на списък инициализиращото за цикъла действие, присвояването `p := s` е извън цикъла и формално, т. е. синтактично, не е свързано с него. Всъщност то може да се намира където и да е преди същинския цикъл. Реинициализиращото действие `p := p^.next` пък е, също от формална гледна точка, неразличимо сред другите действия в тялото на цикъла. Типично то се поставя в края му, но може да се намира и навсякъде другаде.

Така и инициализиращото, и реинициализиращото действия в примера са неявни, формално необособени.[7] Да подчертаем още веднъж, че става дума за типичен език за програмиране и типичен при-

[7] Неявно, но само визуално, е реинициализирането и в цикъла `for` – там то се подразбира от смисъла на конструкцията в езика.

мер в него. При това следва да отбележим и че всяко от двете неявни действия може да бъде сложно, съставено от известен брой отделни, а с това и всяко от тях може да се окаже пръснато из текста на програмата. Това прави програмата още по-трудно разбираема и уязвима от невнимателни промени.

С оглед на горното наблюдение, сполучливо решение е избрано в езика C, а оттам то е наследено в множеството произлезли от него езици. Командата `for` в такива езици притежава синтактично обособени дялове, отговарящи на инициализиране на цикъла, проверка на условието за завършване/продължаване, реинициализиране и тяло. Всеки от трите първи дяла може да бъде произволен израз, а може и да отсъства. С това тази команда е обобщение и на цикъл с брояч, и на цикъл с предусловие. В частност, тя не е обвързана с каквато и да е форма на броене, но удобно се приспособява и за такава цел и това е една от типичните ѝ употреби.

С помощта на такава команда `for` двата примера от по-горе се изразяват адекватно, с явно изразена структура и еднообразно един с друг, както следва да бъде:

```
/* масив */                  /* списък */
for (i=0; i<n; ++i) {        for (p=s; p!=NULL; p=p->next) {
  ...a[i]...                   ...p...
}                            }
```

За отбелязване е, че обособяването на инициализираща част в цикъла по принцип позволява тя да съдържа и описателни действия, така че променливи, използвани само в рамките на цикъла, изрично да се направят локални в него.

Едно непубликувано преброяване, направено от нас върху свои (∼2500 реда) и чужди (от операционната система SunOS/Solaris, ∼11000 реда) програми на C показва, че около 1/3 от циклите с пред- и следусловие и около 2/3 от всички имат, явно изразено или

не, различими при четене инициализиращи части. (Циклите от вида `for` са приблизително колкото тези с пред- и следусловие общо и практически всички имат инициализираща част.) Примерно такъв е и делът на циклите с реинициализиращи части, независимо дали са явно изразени или не. И отново приблизително същият брой цикли имат по същество локални променливи (даденият вариант на езика не позволява те да се посочат като такива в явен вид). Наблюденията ни върху множество други програми не само потвърждават тези сведения, а дори често показват доста по-високи дялове.

И така, адекватна конструкция за явно изразяване при описване на типичен прост итеративен процес е такава, която позволява синтактично обособено задаване на четирите посочени съставни действия.

Веднага обаче трябва да отбележим, че споменатата конструкция `for` в C и пр. не е наистина пълноценна в това отношение: липсва ѝ общност.

Инициализиращата и реинициализираща части не могат да бъдат произволни действия, а само изрази, т. е. те могат да съдържат само аритметични и други операции и присвояване (съюзът `,` в случая служи за формално обединяване на няколко присвоявания в един израз), но не команди.

Възможно е в инициализиращата част да се постави определение на променлива, за да стане тя локална в цикъла, но се допуска само едно определение, а и присъствието на такова действие изключва възможността за други присвоявания. Така, запис като следния

```
for (int i=0; i<n; ++i) ...
```

е възможен, но не и някой като

```
for (int i=0, int j=n-1; i<j; ++i,--j) ...
```

или

```
for (int i=0, j=n-1; i<j; ++i,--j) ...
```

или

```
for (i=0, int j=n-1; i<j; ++i,--j) ...
```

макар че потребността от него е точно толкова основателна, колкото и за горния.

Различни ситуации от практиката на програмирането подсказват разнообразни допълнителни очаквания към изразителността на конструкциите за цикъл. Например в някои случаи е желателно да направим една или повече променливи локални не за всичките четири части общо, а за някои от тях поотделно, или за няколко заедно – да кажем, за инициализиращата, реинициализиращата части и управляващото условие, или поотделно за инициализацията и за условието.

Свързан с употребата на локални стойности въпрос е този за назначаването на действия, които да се изпълняват при завършване на цикъл. Такива действия трябва да имат достъп до локални за цикъла променливи, а наред с това да произвеждат последствия – стойности, свързвания – в съдържащия цикъла контекст, извън неговата област на локалност. Прологови, епилогови и друг вид помощни действия може да има не само цикълът като цяло, а и отделни негови части. Пример за това е управляващото условие, ако неговото пресмятане е сложно. Друг пример е действието за еднократно инициализиране на тялото на цикъл, след общата инициализация и пресмятането на условието и следователно – само ако то е истинно. Някои такива варианти на организацията на циклични конструкции са изследвани например в [75].

Простият итеративен процес може да бъде видоизменян и по други начини. Многократно е посочвано в различни публикации [52, 36, 73, 24, 23], че проверяването на управляващо условие и съ-

ответно възможното напускане на цикъл е по-удобно да става *в тялото му* вместо преди или след него. За това свидетелстват както експерименти с различни циклични конструкции, така и някои формални разглеждания.

Очевидно такъв цикъл – със *средусловие* – обобщава тези с пред- и следусловие. При това той се съгласува с направения по-горе извод за необходимост на цикъл с обособени дялове в следния смисъл. В редица случаи адекватността на цикъл със средусловие произтича от необходимостта да се зададе реинициализираща част в тялото на цикъл с предусловие непосредствено преди условието, или да се зададе такава част в тялото на цикъл със следусловие непосредствено след условието. И в двата случая тялото се разделя на две, т. е. получава се цикъл със средусловие.

Според вече споменатото направено от нас преброяване повече от 1/4 от всички цикли или са били действително написани като цикли със средусловие (с изрична команда за напускане, `break` или друга), или очевидно се подобряват (като например с това се спестят повтаряния на еднакви части в програмата), ако бъдат пренаписани като такива. За някои видове програми посоченото отношение се оказва значително по-високо: при друго преброяване, разглеждащо реализации на комбинаторни алгоритми, „средусловните“ команди за цикъл се оказаха чувствително над половината!

Същевременно следва да имаме предвид, че команди от типа на `break` не осигуряват същинска конструкция за цикъл със средусловие. Тя не е синтактична част от цикъла, а и винаги се влага не непосредствено в него, а на поне едно равнище по-дълбоко. Това намалява явността на изразяване и най-вече адекватността, наглед-ността и формалната анализируемост на текста.

Съществуват още редица други моменти по отношение на структурата на цикличното изпълнение. В различни публикации като

[55, 89, 23, 82, 53, 59] се застъпват гледища, в които изпъкват различни негови аспекти. Едно от често обсъжданите като желателни свойства на цикли е възможността тялото да се напуска в няколко различни точки, а понякога – да се напускат няколко вложени един в друг цикъла наведнъж. Този въпрос е свързан и със споменатите вече осигуряване на епилог и проблема с локалността.

В някои изследвания [32, 45] се търси обобщено третиране на управлението в програми, при което разнообразни итеративни и условни форми се разглеждат единно.

Важен във връзка с отношението му към адекватността и явността въпрос е този за недетерминистичното съотнасяне на няколко части в команда за избор или за цикъл [27, 28], както и в комбинирани форми [33, 12, 67]. Недетерминираните форми на управление позволяват изрично да се посочи отсъствието на определеност във връзка с един или друг вид отношение в множество от действия – наредба, съвместяване или обособеност и др. – т. е. явно да се изрази *произволността на избор* от известен брой възможности. Според нас семантичната изразителност, свързана с участието на недетерминизъм в команди е все още слабо изучена, а и за съжаление такъв вид команди се срещат крайно рядко в използваните езици за програмиране.

Част от изброените желателни, с цел по-адекватна изразителност, свойства на конструкциите за цикъл са осигурени в отделни езици за програмиране. Например командата за цикъл в езици като Ada и Eiffel позволява управляващото условие да се постави в тялото на цикъл, да има няколко различни условия и дори да се прекратява повече от един цикъл наведнъж.

В езика Java към конструкцията за цикъл може да се припише т. нар. финализатор, който играе ролята на епилог към цикъла. За напускането на цикъл и в този език, и другаде, често е удобно да се

използва механизмът на изключенията (exceptions), но това решение е косвено, няма общо собствено с конструкцията за цикъл, по природата си е в известна степен неявно и по други причини може да бъде неподходящо.

В други езици адекватно, явно и нагледно изразяване се постига чрез абстрактна конструкция за цикъл и механизма на т. нар. итератори. Абстрактната конструкция обобщава „цикъла с брояч“, като скрива подробностите по инициализиране, реинициализиране и проверка на края на изброяването. Последните се реализират чрез самостоятелен модул – итератор. Различните итератори осигуряват еднообразен интерфейс за достъп до споменатите действия. Така една и съща, абстрактна конструкция за цикъл се използва за изброяване на всякакви съвкупности и структури от данни – числови интервали, масиви, списъци и друг вид редици, множества, дървета, таблици – и то по различни начини, стига за целта да е осигурен итератор.

Този подход се появи в експерименталните езици Euclid [54], Clu [57] и Alphard [78], а едва в последните години вече се възприе в редица широкоупотребявани езици като Lua, JavaScript, C++, Java, C# и ред други. Но, макар и изключително сполучливо да се справя с един доста широк клас от случаи на употреба на цикъл, той не е предназначен за изразяване на итеративност от всякакъв вид.

Консумативни команди и задаване на цикли

В този подраздел разглеждаме един различен от известните подход към задаване на итеративност, предложен от автора [15]. Основната негова особеност е, че различни части на тялото на цикъла могат да се окажат изпълнявани различен от този на останалите брой пъти. По същество структурата на повторното изпълнение се

задава *вътре* в тялото на цикъла, а не извън него, както е обичайно в познатите конструкции.

Да покажем с някои конкретни примери използването на този подход за повишаване на явността и адекватността в цикли.

Най-напред да разгледаме следната типична програма на C:

```
compute(x[0],y[0]);
move(x[0],y[0]);
for (i=1; i<n; ++i)  {
  compute(x[i],y[i]);
  draw(x[i],y[i]);
}
```

за която се предполага, че рисува начупена линия, добивайки координатите на върховете ѝ чрез `compute()` всеки път непосредствено преди съответният връх да бъде използван за чертане. Тъй като действието с първия връх (преместване) е различно от това с останалите (чертане), това, което би съответствало на `i=0` в цикъла остава извън него, въпреки че е сходно и частично се покрива със съдържанието на цикъла.

Друг вариант, в който се предпочита обединеното обработване на всички върхове в тялото на цикъла, е:

```
for (i=0; i<n; ++i)  {
  compute(x[i],y[i]);
  if (i==0)  move(x[i],y[i]);
  else       draw(x[i],y[i]);
}
```

но при него се налага да се проверява стойността на `i` при всяка итерация, което е по същество излишно, а няма как да се избегне.

Нито една от двете конструкции не е удовлетворителна. Това, което би трябвало да изразим е „да се пресмята всяка координатна

двойка и тогава, ако тя е първата, да се използва `move()`, а *винаги* след това, без изрично да проверяваме – `draw()`“.

Друг пример представя типична задача за търсене. Обхожда се даден информационен масив, като наред с другите дейности се търси наличието на конкретна данна или друга разпознаваема ситуация. Ако търсенето е успешно, установява се булев признак `flag`. Първото решение е грубо: условието `P` за наличие се проверява ненужно много пъти и съответно се извършва много пъти присвояване на `flag`.

```
flag = 0;
while (...)  {
  ...
  if (P)  flag = 1;
  ...
}
```

В следващия подобрен вариант пресмятането на `P` и съответното присвояване се възпират от допълнителна проверка за установката на признака, но тази проверка все пак се прави при всяка итерация:

```
flag = 0;
while (...)  {
  ...
  if (flag == 0)
    if (P)  flag = 1;
  ...
}
```

Отново никой от вариантите не задоволява, защото не отговаря точно на това, което трябва да се прави по същество, а именно „да се установи признакът при първото удовлетворяване на `P` и след това и проверката, и присвояването да се пропускат в тялото на цикъла“.

Да разгледаме изменение на горната задача, при което признакът `flag` предварително има някаква стойност, която не знаем и не бива да променяме, ако търсенето е неуспешно. Това означава, че инициализацията на признака е недопустима, а и проверката във външния `if` на подобрения вариант на програмата е безсмислена – с нея програмата би била грешна. Налага се да се въведе нова променлива (`t`), която дублира основния признак:

```
t = 0;
while (...)  {
  ...
  if (t == 0)
    if (P)  t = flag = 1
  ...
}
```

Трябва да се отбележи, че промяната от втория към третия вариант на програмата се налага не защото се изменя *по същество* извършваното в цикъла, а за да сменим охраняващата проверка, която е невалидна без инициализацията на `flag`. Проверката на свой ред е въведена, за да предотврати ненужно изпълняване на по-сложни действия. Така програмата „обраства“ с несъществени за основното ѝ действие елементи.

Като последен пример да разгледаме частен случай на предишния: даден признак се установява *безусловно*, ако и само ако изпълнението достига (поне веднъж) тялото на даден цикъл.

```
flag = 0;
while (P)  {
  flag = 1;
  ...
}
```

В горния вариант на решението се повтаря присвояването при всяка итерация, което можем да избегнем с:

```
flag = P;
while (P)  {
  ...
}
```

но тук пък `P` се пресмята с един път повече от необходимото, а и установяването на признака става извън цикъла, за който се отнася, т. е. формално несвързано с него: връзката между установяването на признака и цикъла става неявна.

Примери като горните нямат удовлетворителни решения в рамките на обичайните средства за задаване на цикли. Причината е, че някои от действията в телата на циклите трябва да се изпълняват еднократно, или до удовлетворяване на някакво условие, или след него и др. под., за което не разполагаме с подходящи средства. В резултат програмите често се претрупват с допълнителни, не по същество, променливи и действия.

На основата на горните наблюдения, като средство за по-адекватно и явно изразяване на отношения като в цитираните примери, въвеждаме едно подходящо за случая уточнение на обичайната представа за императивния модел на пресмятане, което наричаме модел на *консумативно пресмятане*, и набор от конструкции, които използват този модел.

Именно, да приемем, че всяка команда в програмата се поражда непосредствено преди изпълняването си и се унищожава веднага след него. Условните команди пораждат само един от клоновете си. Командите за цикъл имат допълнителното свойство да *възстановяват по принцип* всички команди в тялото си за всяка следваща итерация.

С такова уточнение познатите ни команди и съставените от тях програми се държат по обичайния начин. За да се възползваме от модела на консумативното пресмятане, като допълнителни команди – или по-скоро командни конструктори, тъй като те, подобно на `if`, `for`, `while` и други познати, служат за *организиране* на изпълнението на команди – въвеждаме *консумативни команди.*

Консумативните команди могат да потискат възстановяването си, т. е. да се самоунищожават, или да заменят едно действие с друго. Тъй като и потискането, и замяната имат смисъл само в повторителен контекст, консумативните команди са предназначени за използване в цикъл.[8]

Да определим следните консумативни команди, заедно с кратки неформални пояснения за всяка. Навсякъде в определенията `S`, `T` и `U` са команди, а `P` – условие.

```
once  S  [ then  T ]
```

Изпълнява се `S` веднъж. На всяка следваща итерация се изпълнява `T` (или нищо, ако втората част не е зададена).

```
oncewhen (P)  S
onceif (P)  S  else T
```

Проверява се `P` веднъж; ако е „истина“, изпълнява се `S`. Ако `P` не е „истина“, във втория случай се изпълнява `T`. При всяка следваща итерация се изпълнява отново `S` или `T`, или нищо, както е определено на първата итерация, но вече без да се прави проверка.

```
before (P)  S  [ then  T ]
```

При всяка итерация се проверява `P` и ако не е „истина“, изпълнява

[8] Заслужава да отбележим, че повторителен контекст може да възниква и в друг смисъл, например при повторно изпълнение на подпрограма. В тези случаи също може да е желателно дадени действия да се изпълняват еднократно или да имат други подобни особености. Това подсказва, че консумативният модел и команди допускат по-широка приложимост от обсъжданата тук, но на нея няма да се спираме.

се S. След като P стане „истина“, при текущата и всички следващи итерации се изпълнява само T, ако е зададено.

```
after (P)  S
```

Проверява се P на всяка итерация, докато добие стойност „истина“. След това, при текущата и всички следващи итерации се изпълнява (безусловно) S.

```
do  S  while (P)  T [ then U ]
```

При всяка итерация се изпълнява S и се проверява P. Ако P има стойност „истина“, изпълнява се и T. Когато P стане „неистина“, U се изпълнява вместо T и при всяка следваща итерация се изпълнява само U (ако е зададено).

Консумативните команди могат да се влагат една в друга. Следващата команда

```
again [ L ]
```

възстановява консумативната команда, в която се съдържа, ако не е зададено L. В противен случай възстановява посочената чрез етикета L команда (и също всички вложени в нея). Действието на again се проявява на *следващата* итерация.

Изброените команди или аналогични на тях могат да бъдат добавени към съществуващите в кой да е императивен език за програмиране като негово усъвършенстване. За да подчертаем изразителната им сила, да забележим, че при наличие на такива команди, за изразяване на итеративност всъщност е достатъчна само една, възможно най-проста команда за цикъл.

Нека командата (или команден конструктор) loop задава „безкраен“ цикъл – тя осигурява само повтарянето на дадена проста или съставна команда, заедно с приписаното от консумативния модел потенциално възстановяване. Аргументът на loop – тялото на ци-

къла – се изпълнява, докато в него има поне една несамоунищожила се команда.

С помощта на `loop` и консумативни команди можем както да имитираме популярните форми на циклично изпълнение, така и да съставим различни други. Например:

`loop before (P) S;`	цикъл с предусловие
`loop do S; while (P) ;`	цикъл със следусловие[9]
`loop do S; while (P) T;`	цикъл със средусловие
`loop {once S; before (P) T; }`	цикъл с инициализираща част `S`, тяло `T` и предусловие `P`

Първият пример от началото на този подраздел може да се програмира сега така:

```
loop {
  once i = 0;
  before (i==n) {
    compute(x[i],y[i]);
    once move(x[i],y[i]);
    then draw(x[i],y[i]);
    i :+ 1
  }
}
```

(`:+` е присвояваща операция, както бе въведено по-горе). Вторият – със следния фрагмент:

```
loop {
  once flag = 0;
  before (...) {
```

[9] Частта от тялото след условието в случая е празна команда

```
    ...
    after (P)
      once flag = 1;
    ...
  }
}
```

За да реализираме *изменения вариант* на втория пример, достатъчно е да премахнем първата команда от тялото на `loop`. Остатъкът от тялото на цикъла не зависи от това дали признакът има начална стойност или не, както беше преди това.

Накрая, третият пример се решава с:

```
loop  {
  once flag = 0;
  before (P)  {
    once flag = 1;
    ...
  }
}
```

Консумативните команди дават възможност да се зададат прологова и епилогова части към даден цикъл, като прологът може да бъде изпълнен и условно (според това, дали тялото на цикъла ще бъде изпълнено или др.). При това във всеки от случаите е възможно прологът и/или епилогът да се поставят в тялото на цикъла, с което в тях стават достъпни локалните за тялото променливи.

Освен това, влагането на консумативни команди една в друга дава възможност в структурата на *всяко* действие да се обособят еднократно или неколкократно изпълнявани части.

Една възможност, благоприятно илюстрираща използването на консумативни команди, е програмирането на генератори за реку-

рентноопределени числови редици, стига рекурентността да може да бъде сведена до итеративност, както в случая с редицата на Фибоначи $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$.

```
once  {s = 1;  t = 0}  then  t :+ (s =: t);
```

Горният фрагмент може да бъде поставен в тялото на произволен цикъл: тъй като не използва променлива-брояч на цикъла, нито зависи по друг начин от обкръжението си, такъв фрагмент е *преносим*. След всяка итерация той дава в променливата `t` поредното число от редицата, а контекстът определя например колко числа да бъдат породени в дадения цикъл.

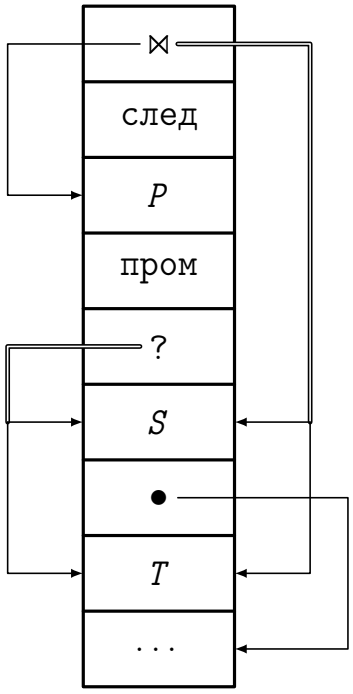
Същият фрагмент дава повод и да направим следната забележка във връзка с консумативните команди. Като форма на изразяване на *самоотнасяне и съотнасяне* в програмен текст те се родеят с рекурсивността, но последната, основавайки се на *именуване* на програмните обекти, чието действие се възпроизвежда, обикновено се отнася до *подпрограмното* равнище, докато чрез консумативните команди се управлява именно командното равнище на следване на действията. Самоотнасянето и съотнасянето в програмен текст по принцип дават възможност за по-пряко явно изразяване на свойства и връзки на или между обектите в програмата, избягвайки посредничеството на допълнителни променливи, както и повторното изписване на едни и същи действия.

В смисъла на горното, съпоставяйки двете традиционни в програмирането форми на задаване на повторност – итеративната, с подчертано императивен стил, и рекурсивната, по-близка до функционалната парадигма – можем да кажем, че консумативните команди, усилвайки и без друго характерната за итеративната форма по-голяма универсалност, я и съчетават с по-високата адекватност и явност, която в много случаи е налице при рекурсивното изразяване на действия.

Последният пример показва изброяване на нерегулярна редица от няколко, в случая три стойности, която е превърната в безкрайна периодична редица.

```
sw:  once  t = A;
     then  once  t = B;
           then  {t = C;  again sw}
```

Въпреки че това няма пряко отношение към явността и другите разглеждани тук свойства, практически важно е консумативните команди да могат да бъдат ефективно реализирани на компютър с традиционна архитектура. Да покажем как може да стане това, като разгледаме схема за реализиране на конкретна команда:
`onceif (`P`)` S `else` T .



Първоначално полето ⋈ съдържа команда за преход към пресмятане на условието P. След като P бъде пресметнато процедурата `пром` променя полето ⋈, така че при следващо изпълнение преходът да бъде към някоя от командите S или T – според истинността на P. Непосредствено след това, също според истинността на P, командата за условен преход `?` предава управлението към S или T.

При всяко следващо изпълнение на командата `onceif (`P`)` S `else` T благодарение на ⋈ се преминава направо към изпълнение на S или T.

Полето `след` съдържа указател към начало на фрагмент, отговарящ на следващата консумативна команда. В частност, такива команди може да се съдържат в S или T и тогава указателят сочи към първата от тях.

Чрез полетата `след` се образува свързан списък от всички консумативни команди в даден участък, по който да могат да се възстановят стойностите на полетата ⋈ в целия участък или част от него. (Възстановяване се прави при завършване на изпълнението на цикъл или при команда `again`.) За всяка консумативна команда независимо от вида ѝ възстановяването се свежда до поставане на едно и също относително отместване като адрес в командата за преход в ⋈.

За реализиране на въведената по-горе команда `loop` е необходимо ефективно да се проверява дали всички части на тялото ѝ са унищожени. Това е възможно чрез допълнително усъвършенстване на реализационната схема на консумативните команди, което не обсъждаме тук.

Без съмнение интересна е и възможността за реализиране на консумативни команди чрез подходящо специализиран за консумативния модел на изпълнение хардуерен процесор.

4.3. Агрегиране

Програмирането борави с различни форми за образуване на съвкупности от стойности. Една от най-простите сред тях са n-торките (енторки, tuples): къси редици с фиксирана дължина – двойки, тройки и т. н. – от възможно разнородни стойности. Странно е, че това е и една от най-очевидните форми за агрегиране на данни и въпреки

това бе пренебрегвана много дълго, дори донякъде все още е.

В някои езици енторките са един от видовете стойности. Такива са повечето съвременни функционални езици като ML, Haskell и Scala, но не само те. В езика C++ съвсем доскоро този вид стойности бяха представени само от двойки, но обновяването на определението на езика въведе енторки с произволна дължина.

Енторките-стойности могат да бъдат полезни навсякъде, където е нужно няколко стойности да бъдат третирани като една. Например при наличие на енторки-стойности в даден език за процедурите в него е достатъчно да имат само по един аргумент; където има нужда от няколко, образуваме енторка. Това опростява езика без да намалява изразителността му.

В други езици енторките не са стойности, а само средство да се представят повече от една стойности заедно. Такова третиране има в Lisp, Clu и Lua, където употребата на енторки се ограничава с това да се допуска множествено присвояване и функция да дава като резултат няколко стойности вместо една. Например по този начин функция за целочислено делене може да дава получените частно и остатък. Тъй като обаче не са стойности, такива енторки не могат да се съхраняват под едно име, да бъдат елемент на друга стойност и в частност друга енторка или да бъдат предавани като аргумент при обръщение към функция.

Тук предлагаме понятие за енторка, което се различава и от двете споменати по това, че е чисто синтактично средство за агрегиране и като такова може да бъде добавено на практика към всеки език за програмиране без да го изменя по същество [16].

Да разгледаме няколко примера, като бележим енторките с квадратни скоби.

• Следното е споменатото в друга форма по-горе съвместно присвояване, което в конкретния случай е циклична замяна на стойнос-

тите на три променливи:

```
[a, b, c] = [b, c, a]
```

• Намиране на сбор, разлика, целочислено делене и остатък от такова делене на `a` с `b`. Резултатите се записват съответно в `s`, `d`, `q` и `r` (образуваме енторка от операции и такава от имена на променливи):

```
[s, d, q, r] = a [+, -, /, %] b
```

• Променливите `x`, `y` и `z` получават една от две тройки съответни стойности според верността на посочено условие:

```
[x, y, z] = if (...) [e1, e2, e3] else [e4, e5, e6]
```

• Прилагане на функция `f` „успоредно“ към няколко набора от аргументи. Резултатът е енторка от съответния брой резултати от повикванията на `f`:

```
f[(...),(...),...]
```

• Повикване на три функции с един и същ набор от аргументи. Резултатът е тройка от резултатите от повикванията:

```
[f, g, h] (...)
```

И така, „синтактичните“ енторки могат да се използват за съвместяване на цитирането на аргументи за няколко операции или функции, или обратното – цитирането на операция или функция за няколко набора от аргументи, и др. под. случаи, където трябва да се изрази родството на няколко действия. Те могат да бъдат и вложени една в друга. Освен да направи явна една или друга връзка в програмата, употребата на енторки може да подпомогне по-ефективното ѝ или паралелно изпълнение.

4.4. Специфициране

Накрая привеждаме пример с език не за програмиране, а имащ косвено отношение към него. Тъй като става дума за формален и

същевременно предназначен за четене от човек език, съображенията за адекватно, явно и нагледно изразяване по подобен начин важат и за него.

Разглеждаме представянето на редици чрез средствата на мета-синтактичен език.

Редиците са основен вид синтактична структура. По-конкретно, редици от (синтактично) еднотипни елементи се срещат много нашироко – например в езиците за програмиране такива са редиците от цифри в числа, от букви в думи, от лексеми в програмата, от команди и много други. Такива редици са и най-простият вид нетривиална синтактична структура – нетривиална в смисъл че за описването ѝ е нужно да се прибегне до една или друга форма на повторност.

Най-широко използваното средство за описване на синтаксиса на езици за програмиране е (мета)езикът BNF и различни негови производни. Първоначалният BNF има по същество само две конструкции: за задаване на следване и на вариантност. Отсъства например дори конструкция за условно (незадължително) включване, но най-съществената отсъстваща конструкция е такава за изразяване на повторност. Поради това синтактичните редици от всякакъв вид се представят в този език изключително чрез рекурентни определения.

Рекурентните определения имат сериозни недостатъци, свързани с посоката на рекурентността. Да разгледаме непразна редица s от елементи от някакъв (граматичен) тип a. Тя може да се определи чрез ляворекурентно:

$$s \quad = \quad a \mid s\ a$$

или дяснорекурентно правило:

$$s \quad = \quad a \mid a\ s$$

От какво може да е обусловен изборът между двете и какви последствия има той?

Преди всичко да отбележим, че строежът на правилото и по-конкретно видът на рекурентността в него интуитивно се асоциират с поведението на рекурсивна програма с аналогичен строеж. Двата вида рекурентност в структурата на програма водят до различен тип изпълнение в действителност. Програма с „дяснорекурентна“ структура отговаря на остатъчно (опашково, tail) рекурсивно изпълнение, което непосредствено се превежда в итеративно – такава рекурсия не е същинска. Ляворекурентно определение води до наистина рекурсивно изпълнение; превеждането на такъв вид рекурсивност в итеративност, макар по принцип възможно, не е непосредствено, а и така или иначе програмата е по-ресурсоемка от итеративната.

Следователно двата вида рекурентност на интуитивно равнище се асоциират с различно ефективно поведение.

Освен това, ако редицата представя някаква последователност от действия, видът рекурентност би трябвало да е съгласуван с посоката на извършване на действията. Именно, лявата рекурентност подсказва извършване на действията отляво надясно, а дясната – в обратен ред. Така например за поредица от числови изваждания или деления, където обичайният ред е отляво надясно, естественото представяне е ляворекурентно, а за редица от повдигания на степен или присвоявания, за които съдружаването е дясно, естествено е дяснорекурентно представяне.

От друга страна, задаването на граматични правила най-често е свързано с използването им от автоматичен граматичен анализатор, а видът на анализатора диктува съвсем друго правило за избор между лява и дясна граматична рекурентност.

Анализаторите биват основно два вида. Тези, които работят по схемата „отгоре надолу“, пораждат разпознавателен процес, който, ако на свой ред се опише рекурентно, има същия вид рекурентност като съответното правило. При анализаторите, които прилагат схе-

мата „отдолу нагоре“, порожданият процес е обърнат по вид рекурентност спрямо правилото. Така едно и също рекурентно граматично правило е благоприятно за единия вид анализатори и точно обратното за другия.

И така, изборът между лява и дясна рекурентност за представяне на редици се натъква на няколко противоречиви изисквания, а резултатът от този избор съдържа неявни допускания и решения и може да бъде заблуждаващ. Дори само фактът, че такива граматични правила не са с универсална, извън конкретен вид анализатор полезност, или че ги съобразяваме с анализаторите вместо със същността на описваните обекти, е достатъчен да заключим, че като описателно средство за редици те са неадекватни.[10]

BNF има няколко широкоизползвани понастоящем наследника. Два от тях са ABNF (Augmented BNF) и EBNF (Extended BNF), като вторият е официално стандартизиран (ISO/IEC 14977:1996(E)). Всички наследници в един или друг вид допълват езика с повторителна и условна конструкции и някои други. В частност, така задаването в тях на еднообразни последователности може да става не само рекурентно, а и непосредствено.

Проста редица s като горната се задава с просто синтактично правило, непосредствено използващо повторител:

$$s \quad = \quad a*$$

Тук използваме знака $*$ да означава „повтаряне 0 или повече пъти“; ако е нужно редицата да бъде непразна, записваме

$$s \quad = \quad a\,a*$$

Ако периодично се повтаря не единствен член (синтактична категория), а няколко поредни, достатъчно е да се посочи, че $*$ се отнася за цялата група:

[10] Въпреки това, за съжаление продължават да бъдат широко използвани.

$$s \quad = \quad (a\, b\, \ldots)*$$

Този вид задаване на редица не е обвързан с и не подсказва съображения нито за посока на съдружителност, нито за такава на граматичния анализ, нито за неговата ефективност. Следователно описването чрез повторител следва да се предпочита пред рекурентното.

Много видове редици в езиците за програмиране обаче се задават с редуване на „същински“ членове и разделители. Например в някои езици такива са (редиците могат да имат и ограничители за начало и край, но това не е съществено тук):

- списък от параметри като `(int a, float x, const char c)`
- списък от аргументи като `(4*i,d+(pi-beta)/r,'w')`
- инициализатори като `{5,2,12+a}`
- редица от команди като `x = 12; f(&t); if (...) ...`

В първия случай същинските членове на редиците са определения, а разделителите – запетаи. Такива са разделителите и в следващите два случая, където същинските членове са изрази. В четвъртия пример членовете са команди, а разделителите – точки и запетаи.

Изразите, членове на два от видовете редици по-горе, са на свой ред също такива редици: членове на тези редици са първични изрази, а разделители са инфиксните операции. Например вторият аргумент в списъка по-горе е редица от три члена, разделени с `+` и `/`. Да забележим, че в случая разделителите не само не са един и същ навсякъде, а в зависимост от избраното граматично представяне могат да принадлежат на различни граматични категории.

В общ вид редица с разделители, ако не е празна, има вида $a\ b\ a\ b\ \ldots\ b\ a$ или само a, където a е същински член, а b е разделител. С какво синтактично правило или правила се задава такава редица?

Отново са възможни ляво- и дясноректурентни решения, съответно

$$s \quad = \quad a \mid s\ b\ a$$

и

$$s \quad = \quad a \mid a\ b\ s$$

но те имат същите недостатъци като при простите редици. Имаме на разположение и трета, симетричнорекурентна възможност за определяне:

$$s \quad = \quad a \mid s\ b\ s$$

Тук проблемите с посоката отсъстват, но се появява друг. Разпознаването на частите на редица, т. е. коя част отговаря на първата и коя на втората поява на s отдясно на знака = с такова правило по начало е нееднозначно. Граматичен анализатор от вида „отдолу нагоре“ допуска работа с такива правила, като нееднозначността се разрешава чрез задаване на допълнителен вид правила, определящи лява или дясна съдружителност за разделителите b на съответните редици. Това е много удобно за редици, в които „разделителите“ са аритметични и други операции.

Наред с другото, такова решение позволява да се въведе само една граматична категория за израз вместо няколко за различни (под)видове изрази, отразяващи в неявен вид различните предимства на операциите. Решението обаче не е приложимо за анализатори от вида „отгоре надолу“.

Представено чрез правило с повторител, описанието на редица с разделители може да бъде такова:

$$s \quad = \quad a\ (b\ a)*$$

или такова:

$$s \quad = \quad (a\ b)*\ a$$

като вторият вариант е донякъде по-неестествен и не се използва на практика.

Да обърнем внимание, че итеративните представяния на редица, подобно на ляво- и дясноректурентните, са несиметрични, докато самата редица е симетрична. Освен това, всички тези представяния съдържат известен излишък – двукратното цитиране на категорията a във всяко. И двете наблюдения говорят за известна неадекватност на представянията и подсказват да потърсим друг вариант.

Тук предлагаме следния. (За пръв път той е използван за задаване на граматиката на езика U [19, 2].)

Нека фигурните скоби `{` и `}` в метаезика за описване на синтаксис означават повтаряне на съдържанието между тях *един или повече пъти*. Нека при това, ако елементите между скобите са повече от един, последният от тях се появява *между* всеки две повторения на редицата от останалите елементи. Така този последен и неединствен в редицата елемент играе ролята на разделител между останалите.

Предложеното определение е усъвършенстване на езика BNF и други разширения на BNF, без при това да вкарва в употреба нови конструкции. В EBNF и другаде понастоящем фигурните скоби задават повтаряне на *цялото съдържание между тях нула или повече пъти* – това, което досега бележехме със знака $*$. Квадратните скоби `[` и `]` задават незадължително, т. е. нула или едно възпроизвеждане на съдържанието между тях. Налице е известно припокриване между функцията на `{` и `}` и тази на `[` и `]`.

В нашето определение припокриването изчезва, а двете двойки скоби сполучливо взаимодействат за изразяване на различни структури. Получаващите се изрази на така променения BNF могат да описват редици адекватно, явно и нагледно, при това просто и без излишни елементи.

Да разгледаме няколко примера.

• Проста непразна редица с елементи a се задава с `{`a`}`. За да допуснем и празна редица, записваме `[{`a`}]`.

• Редица от известен брой циклично повтарящи се елементи се представя с `{(`$a\,b\ldots$`)}` или `{`$a\,b\ldots${`}`.

• Редица с еднотипен или с редуване на няколко вида елемента и разделител z: `{`$a\ldots z$`}`.

• Подобна редица, но с незадължителни разделители: `{`$a\ldots$`[`z`]}`.

Понеже на така описваните структури на редици не се придава посока на съдружаване на елементите, ако това е нужно например в граматичен анализатор, такава посока може да се зададе за едно или друго правило отделно от него, както бе споменато че се прави при анализаторите от вида „отдолу нагоре“.

5. Обща концепция за структурността в програми

Разглежданите свойства на текстове на програми и в частност понятието явност и близките до него, както и начините и формите, в които тези свойства се проявяват, са твърде многообразни за да подлежат на изчерпателно изследване. Изключително многообразие е присъщо и на носителите на тези свойства – езиците за програмиране, а и изобщо на концептуалните модели от всякакъв мащаб за и в програмирането.

Тъкмо тази многоликост подтиква към разглеждане на друго равнище, отстранено от конкретността на конструкции и понятия на отделни езици и парадигми на програмирането. Такова равнище е свързано с изграждане на обща концептуална основа, на система от понятия, чрез които да се формулират и обясняват самите свойства на езиците и изобщо – да се разглеждат обекти с изчислителна (информатична) природа. Началата на такъв подход са набелязани в настоящата глава, а по-подробно са изложени в [17, 18].

В основата на подхода стои разбирането, че търсените общи понятия имат структурна природа: те описват *строежа* на програми и други информатични обекти.

5.1. Източници

Представата за структурност в програмирането възникна с появата на идеологията на т. нар. структурно програмиране. Тя се определя от схващането, че програмите (трябва да) се изграждат от

малък брой предварително посочени конструкции, което прави възможно техните свойства да бъдат извличани, вкл. чрез формални средства, от тези на съставките им. С появата на понятието структурно програмиране започва да се говори за даннови и управленски структури и се осъзнава, че в програмата те са взаимосвързани, двойствени един на друг слоеве. Впрочем по отношение на природата на тази двойственост не се отива далеч.

Структурни представи възникват и в контекста на опитите за формализиране на измерването на сложност на програми, но тези представи са в голяма степен едностранчиви, бидейки подчинени на наложен с определена цел опростяващ модел. Това се отнася например за графовия модел като в [61].

Развитието на езиците за конструктивно специфициране като VDM [20] и Z [30] също има отношение към представите за структурност в програми, доколкото чрез тези езици се описват свойства на програми, вкл. и отнасящи се до тяхната структура. Отношението към структурността обаче е косвено и фрагментирано. Например конструкциите за управление се представят не чрез техни вътрешни отношения, а като преобразователи на данни, а структурите от данни се описват отделно и на практика без връзка с първите.

Най-ценен източник на конструктивни представи за програмите разбира се е еволюцията на даннови и управленски структури в практиката на програмирането и дестилирането на този опит в езиците за програмиране. По този път се развива и натрупва конкретно, предимно емпирично знание за форми на структурност. В някои случаи се достига и до обобщения във вид на правила, схеми или принципи, например структурните по същество принципи за съдържанието на език в [79].

5.2. Принципи

Оформянето на концепцията за структурност в програми изисква да формулираме и обосновем някои основни положения като принципни.

На първо място това е *принципът за обективност на структурността*. Структурата на програма е носител на обективен смисъл, на информация за решаваната задача, тя никога не е напълно произволна.

Реализирането на програма за решаване на определена задача, вкл. изготвянето на описание или спецификация за задачата и на абстрактен модел на решение, е процес на *изявяване на обективни структурни свойства и отношения*.

Като прост пример за валидността на този принцип да разгледаме задачата за подреждане чрез сравнения и размени. От комбинаторни съображения е известно, че тази задача се решава с брой елементарни действия, чийто порядък е не по-малък от $n \log n$, където n е дължината на подрежданата редица. Тогава всяка програма за подреждане трябва да има поне два вложени един в друг итеративни процеса – ако това не е така, програмата би извършвала не повече от $\sim n$ сравнения и размени, което е невъзможно. Сложността на задачата диктува, макар частично, структурата на програмата за решаването ѝ и в този смисъл структурата е поне отчасти обективна.

Друг пример показва не само обективността на структурата в програми, а и доколко всъщност неизследвана е тя понастоящем.

Известно е, че в сравнение с по-простите, но бавни, скоростните алгоритми за подреждане разменят елементи на редицата, които се намират на голямо разстояние един от друг в нея. В общи линии, това е така, защото тогава с една размяна могат да се елиминират голям брой инверсии. По същество обаче не знаем каква точно е

връзката между скоростта на алгоритъма и тези разстояния в общия случай. Познаваме само конкретни алгоритми за подреждане, а и тяхната ефективност, чрез което косвено и структура, се измерва поотделно в скорост и в обем използвано пространство, но интегрална мярка за двете липсва. Така ни убягва обективна структурна информация за програмата, която отразява определено свойство на решаваната задача.

От принципа за обективност на структурността следва и че структурата на програма е относително независима от съдържанието ѝ. Казано другояче, *структурата е сама по себе си носител на обективно съдържание*, и толкова по-важно е да имаме универсален език за работа с нея.

Очевидно изискване, което все пак следва да формулираме като принцип е, че се интересуваме от тъкмо *този вид структурност, който е характерен за информатичните обекти* като такива.

Във всяка област на знанието се обособява специфичен за обектите в нея вид структурност, която им приписваме – един за математиката, друг за физиката, химията и т. н. В случая на информатиката и дори конкретно на програмирането предизвикателство е да се облече в понятия нейният (неговият) собствен вид структурност.

За да си дадем сметка за това, да обърнем внимание, че различните парадигми на програмирането излагат представата за програма, програмиране и всички свързани с тях понятия на почти напълно чужди един на друг езици. Удивително е доколко различни представи дори за понятието програма внушават например императивната и функционалната парадигми: все едно, че става дума за напълно различни неща. Дори отделните езици за програмиране понякога прибягват до твърде различни системи от понятия за описване на сходни същности.

Откриването на адекватните именно на информатичните обекти

структурни понятия е важно и заради това че „информатичността“ е свойствена не само на обектите, които самата информатика създава. Тя съществува и в природата: елементи на информатично поведение има във физични, биологични и изобщо всевъзможни обективно съществуващи системи.

Обособяването на система от понятия, които описват информатичния вид структурност би помогнало, от една страна, да се намери „общият знаменател“ на езиците на информатиката, а от друга, да се открои по-добре информатичният аспект в съдържателно разнородни системи.

Друг принцип в концепцията за структурност е принципът за *структурно единство и съответствие между статично и динамично.*

Уточнявайки смисъла на думите *единство* и *съответствие*, принципът означава, че:

- едни и същи структурни форми описват свойствата и отношенията и в статичните, и в динамичните аспекти на структурността в информатиката;
- там, където динамичното и статичното взаимодействат, те се характеризират с еднаква или съответна структурност.

По-конкретно в контекста на програмирането принципът за структурно единство и съответствие означава, че основните структурни форми и отношения, чрез които описваме данни – следване, избор, повтаряне и пр. – са същите, с които задаваме и структурата на управлението в програма. При това, ако в представянето на данните има вариране, то има разклоняване (избор) и в действията с тях; ако данните са последователни, такива са и действията с тях и пр.

Изкушаващо е съответствието между статично и динамично да бъде разглеждано като двойственост, както например в проективно-геометричен смисъл двойствени са точки и прави или както върхове

и ребра са двойствени понятия за графи. За съжаление, не ни е известно този въпрос да е изследван в информатиката, освен в някои повърхностни аспекти.

5.3. Понятия

Какви конкретно понятия и форми на структурност са свойствени на информатичните обекти? Едва ли е възможно да се посочи изчерпателен списък, а и преди всичко едва ли е възможна такава формализация на въпроса, че да позволи да преценим дали подобен списък е пълен. Така или инак обаче опитът от наблюдаване на множество конкретни проявления на структурност в програми и езици, преминал през филтъра на обобщаване и абстрахиране, дава възможност да формулираме редица общи понятия, независещи от конкретни езици и парадигми и заедно с това намиращи множество проявления в тях. Във всички случаи това е сложен и продължителен процес, в който има нужда да вземат участие множество заинтересовани.

В [17] предложихме и дадохме описания на общи понятия за свойства на информатични обекти и отношения между тях. Такива са **хомогенност**/**хетерогенност** (за съставен обект), **статичност**/**динамичност**, **крайност**/**безкрайност** (на фòрмата), **самоотнасяне** (при рекурентно и др. под. задаване) и др.

В [18] предложихме формирането на общоинформатични понятия да става като в речник, като понятията бъдат групирани в смислови гнезда. Пак там дадохме примери за такова групиране. Тук правим същото, като привеждаме известно множесто понятия с кратки пояснения. Разчитаме на интуицията на читателя да допълни с примери от собствената практика разбирането си за тях.

- **Действие**

Това е основно, неопределимо понятие, с което изразяваме, че програма или друг информатичен обект има наблюдаемо поведение. Действието се извършва от определен обект, потребявайки **ресурси** и предизвиквайки **последствия**. Потребяването на ресурс се характеризира с **достъп**, а **видът** на последния може да се уточни по различни начини от различни гледни точки – напр. потребленски или ефекторен, пряк или косвен, едноличен или споделян (и във втория случай – разделено или съвместено във времето), детерминиран или не и др.

Действието може да се състои от други действия, а според вида на последствията може да бъде причислено към различни категории.

- **Създаване и унищожаване**

Конкретни варианти на създаване са **произвеждане на екземпляр**, **възпроизвеждане** и някои други.

- **Съединяване** (или по-общо: **агрегиране**)

В тази група попадат множество понятия с различна степен на общност, някои от които са **предшестване**, **влагане**, **поставяне в редица**, **разклоняване**, **итериране**, **подчиняване** и **съподчиняване**.

- **Преобразуване**

Отнася се до структурата на обект като цяло и може да включва различни конкретни видове като **приспособяване**, **съхраняване**, **допълване** и др.

- **Взаимодействие**

Например **активиране** (иницииране), **възобновяване** и **прекъсване**.

Някои понятия се отнасят не пряко до информатични обекти, а до *представянето* им чрез език за програмиране или др. под. Такива

са например следните две.

- **Привързване**

  Това е отнасяне на ресурс към определен контекст или контексти. Близки понятия могат да отразяват взаимното съотнасяне между ресурси.

- **Създаване на изглед**

  Включва проектиране, различни форми на абстрахиране и други, отнасящи се до тълкуване на информатични същности в езика.

Създаването на понятийна основа за информатични разглеждания, каквато набелязахме тук, може да се използва, наред с друго, за методологически единно третиране на различни проявления на явността в текстове на програми. То е и предпоставка за създаване на средства за откриване на потенциална неявност в текстове на програми и за метрично оценяване на явността.

Заключение

Постигнатите в работата резултати могат да бъдат резюмирани както следва.

- Направена е обща сравнителна съпоставка на езиците за програмиране като изразно средство с говоримите (преди всичко естествените) езици и този на математиката (гл. 1).
- Формулирани и съотнесени са ред фактори за възприемане на програмен текст (гл. 2). По-подробно са изяснени понятията „явност на изразяване“ и някои близки по смисъл до него (гл. 3). Понятието явност е изцяло ново. Такова е в една или друга степен и съдържанието на другите понятия в контекста на програмирането.
- От гледна точка на свойството явност са разгледани примери с основни видове езикови конструкции: изрази, прости и съставни команди и др. Във връзка с тези разглеждания са предложени оригинални езикови конструкции (гл. 4). По-конкретно:
  - предложени са конструкции за универсално съвместяване на операции с присвояване и с условно присвояване, с преди- и последействие;
  - предложена е проста, универсална и еднообразна схема за построяване на изрази, особено подходяща за функционални езици;
  - предложен е нов модел на пресмятане, наречен консумативен и разширяващ обичайния императивен модел от гледна точка на изразяването на повторност;
  - на основата на консумативния модел на пресмятане е пред-

ложен набор от специфични за него команди – консумативни команди; показано е, че с помощта на последните се изразяват, освен широкоизползваните итеративни схеми, и такива от нов, значително по-общ клас;
  - предложено е обособяване на клас от операции, наречени съюзи, както и конкретни съюзи, произтичащи от консумативния модел на пресмятане;
  - предложено е семейство конструкции (енторки) за агрегиране на действия, на техните аргументи и резултати.
- Разгледана е една типична, обща употреба на метаезика BNF и на тази основа е предложена конструкция за усъвършенстването му (гл. 4);
- Като обобщение на разглежданията около понятието явност е формулирана обща концепция за структурността в програми и други информатични обекти (гл. 5). В частност, формулирани са принципи на такава структурност и са посочени примерни понятия, с които тя се описва.

Посочените резултати са публикувани в следните статии на автора: [15, 16, 17, 18, 19].

Възможности за бъдещо развитие

Смятаме, че работата по темата може да бъде продължена поне в следните няколко посоки:

- Провеждане на тестове, които да покажат доколко успешно програмисти или обучавани в програмиране биха използвали различните предложени тук конструкции.
- Разглеждане от гледна точка на явност, адекватност и пр. на други конструкции в езиците за програмиране, например такива на подпрограмно равнище или свързани с управление на видимостта.

– Разглеждане от същата гледна точка на други формални езици.
– Развиване на понятийната система, въведена в гл. 5, вкл. за унифицирано представяне на езици за програмиране.
– Уточняване на проявленията на принципа за съответствие между статично и динамично в програми.
– Развиване на методология за автоматизирано откриване на потенциални неявности в текстове на програми.
– На основата на горното, създаване на софтуер за анализ на текстове на програми с цел откриване на потенциални неявности и евентуално преобразуване на такива програми.

Библиография

[1] Антонов А. В. *Информация: восприятие и понимание.* К., Наукова думка, 1988.

[2] Банчев Б. Б. *Езикът за програмиране U.* `http://www.math.bas.bg/bantchev/place/u/u.pdf`.

[3] Гелфанд И. М. *Два архетипа в психологията на човека.* Физикоматематическо списание, т. 67 (1993), кн. 1, 3-15.

[4] Кодряну И. Г. *ЭВМ и математика.* Кш., Штиинца, 1984.

[5] Перминов В. Я. *Развитие представлений о надежности математического доказательства.* М., Изд-во Моск. ун-та, 1986.

[6] Симеонов В. *Символите.* Унив. изд. „Св. Кл. Охридски“. С., 1991.

---

[7] Abrahams P. W. *Typographical extensions for programming languages: breaking out of the ASCII straightjacket.* ACM SIGPLAN Notices, Vol. 28 (1993), No. 2, 61-68.

[8] Abran A. *Software metrics and software metrology.* John Wiley & Sons, 2010.

[9] Abelson H., Sussman G. J. with Sussman J. *Structure and interpretation of computer programs.* MIT Press, 1985.

[10] *Programming language Ada. ISO/IEC 8652:2012*, 2012.

[11] Amit N. *A different solution for improving the readability of deeply nested IF-THEN-ELSE control structures.* ACM SIGPLAN Notices, Vol. 19 (1984), No. 1, 24-30.

[12] Anson E. *A generalized iterative construct and its semantics.* ACM Trans. Progr. Lang. Syst., Vol. 9 (1987), No. 4, 567-581.

[13] Backus J. *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs.* (1977 ACM Turing award lecture) Comm. ACM, Vol. 21 (1978), No. 8, 613-641.

[14] Baecker R. M., Marcus A. *Human factors and typography for more readable programs.* Addison-Wesley, 1990.

[15] Bantchev B. B. *Terminable statements and destructive computation.* ACM SIGPLAN Notices, Vol. 29 (1994), No. 2, 33-38.

[16] Bantchev B. B. *Putting more meaning in expressions.* ACM SIGPLAN Notices, Vol. 33 (1998), No. 9, 77-83.

[17] Bantchev B. B. *Towards a framework for comprehending structure in programs.* Proc. 27$^{\text{th}}$ Spring Conf. of the Union of Bulgarian Mathematicians, 1998, 210-215.

[18] Bantchev B. B. *Understanding computing through defining its lexicon.* Proc. 39$^{\text{th}}$ Spring Conf. of the Union of Bulgarian Mathematicians, 2010, 171-177.

[19] Bantchev B. B. *A language for compositional programming: a rationale and design.* Proc. 40$^{\text{th}}$ Spring Conf. of the Union of Bulgarian Mathematicians, 2011, 236-243.

[20] Bjørner D. et al., ed. VDM'87, VDM – A formal method at work. Lecture Notes in Computer Science **252**, Springer-Verlag, 1987.

[21] Borror J. A. *Q for mortals.* CreateSpace, 2008.

[22] Brooks Fr.P., jr. *The mythical man-month. Essays on software engineering*, 2nd ed. Addison-Wesley, 1995.

[23] Buhr P. A. *A case for teaching multiexit loops to beginning programmers.* ACM SIGPLAN Notices, Vol. 20 (1985), No. 11, 14-22.

[24] Covington M. A. *A pedagogical disadvantage of repeat and while.* ACM SIGPLAN Notices, Vol. 19 (1984), No. 8, 85-86.

[25] Dahl O.-J., Dijkstra E. W., Hoare C. A. R. *Structured programming.* Academic Press, 1972.

[26] Dijkstra E. W. *Goto statement considered harmful.* A letter to the Editor. Comm. ACM, Vol. 11 (1968), No. 3, 147-148.

[27] Dijkstra E. W. *Guarded commands, nondeterminacy and formal derivation of programs.* Comm. ACM, Vol. 18 (1975), No. 8, 453-457.

[28] Dijkstra E. W. *A discipline of programming.* Prentice-Hall, 1976.

[29] Dijkstra E. W. *How do we tell truths that might hurt?* ACM SIGPLAN Notices, Vol. 17 (1982), No. 5, 13-15. (Memo EWD 498)

[30] Diller A. *Z: an introduction to formal methods*, 2nd ed. John Wiley & Sons, 1994.

[31] Ebert C., Dumke R. *Software Measurement.* Springer-Verlag, 2007.

[32] Elliott I. B. *The EPN and ESN notations.* ACM SIGPLAN Notices, Vol. 19 (1984), No. 7, 9-17.

[33] Embley D. W. *Empirical and formal language design applied to a unified control construct for interactive computing.* Intern. J. of Man-Machine Studies, Vol. 10 (1978), No. 2, 197-216.

[34] Fenton N. E., Pfleeger S. L. *Software metrics. A rigorous & practical approach*, 2nd ed. PWS Publishing Company, 1997.

[35] Fischer A. E., Grodzinsky F. S. *The Anatomy of Programming Languages.* Prentice-Hall, 1992.

[36] Floyd R. W. *The paradigms of programming* (1978 ACM Turing Award Lecture). Comm. ACM, Vol. 22 (1979), No. 8, 455-460.

[37] Frenkel K. A., Milner R. *An interview with Robin Milner.* Comm. ACM, Vol. 36 (1993), No. 1, 90-97.

[38] Gries D. *The Science of Programming.* Springer-Verlag, 1981.

[39] Griswold R. E., Griswold M. T. *The Icon programming language*, 3rd ed. Coriolis Group Books, 2000.

[40] Halstead M. H. *Elements of software science.* North Holland, 1975.

[41] Hatton L. *Does OO really match the way we think?* IEEE Software, V. 15 (1998), #3, 46-54.

[42] Hoare C. A. R. *Hints on programming language design.* Stanford A. I. Lab. Memo AIM-224, 1973.

[43] Iverson K. E. *Notation as a tool of thought.* (1979 ACM Turing award lecture) Comm. ACM, Vol. 23 (1980), No. 8, 444-465.

[44] Iverson K. E. *A dictionary of APL.* APL Quote-Quad, Vol. 18 (1987), No. 1, 5-40.

[45] Jonsson D. *The flow of control notations Pancode and Boxcharts.* ACM SIGPLAN Notices, Vol. 25 (1990), No. 8, 106-119.

[46] Karsakof S. *Apercu d`un procédé nouveau d`investigation au moyen de machines à comparer les idées.* St. Petersbourg, 1832.

[47] Kernighan B. W., Plauger P. J. *The elements of programming style.* McGraw-Hill, 1978.

[48] Kernighan B. W., Ritchie D. M. *The C programming language*, 2nd ed. Prentice-Hall, 1988.

[49] Klee V., Wagon S. *Old and new unsolved problems in plane geometry and number theory.* The Math. Assoc. of America, 1991.

[50] Knuth D. E. *Ancient Babylonian algorithms.* Comm. ACM, Vol. 15 (1972), No. 7, 671-677.

[51] Knuth D. E., Levi S. *The CWEB system of structured documentation.* Addison-Wesley, 1991.

[52] Knuth D. E. *Structured programming with* `goto` *statements.* ACM Computing Surveys, Vol. 6 (1974), No. 4, 261-301.

[53] Kovats T. A. *A conservative alternative to Pancode.* ACM SIGPLAN Notices, Vol. 25 (1990), No. 11, 80-84.

[54] Lampson B. W. et al. *Report on the programming language EUCLID.* ACM SIGPLAN Notices, Vol. 12 (1977), No. 2, 1-79.

[55] Laski J. W. *On readability of programs with loops.* ACM SIGPLAN Notices, Vol. 14 (1979), No. 11, 73-83.

[56] Ledgard H. F., Marcotty M. *A genealogy of control structures.* Comm. ACM, Vol. 18 (1975), No. 11, 629-639.

[57] Liskov B. et al. *CLU Reference Manual.* Lecture Notes in Computer Science **114**, Springer-Verlag, 1981.

[58] Mac an Airchinnigh M. *Tutorial lecture notes on the Irish School of the VDM.* Trinity College, Dublin, Ireland, 1991.

[59] Magel K. *A theory of small program complexity.* ACM SIGPLAN Notices, Vol. 17 (1982), No. 3, 37-45.

[60] Marlow S., ed. *Haskell-2010 language report*, 2010. `http://haskell.org`.

[61] McCabe T. J. *A complexity measure.* IEEE Trans. on Software Engineering, Vol. SE-2 (1976), No. 4, 308-320.

[62] Meyer B. *Eiffel: a language and environment for software engineering.* The Journal of Systems and Software, Vol. 8 (1988), No. 3, 199-246.

[63] Mössenböck H., Wirth N. *The Programming Language Oberon-2.* Structured Programming 12 (1991), 179-195.

[64] Naur P. *Goto statements and good Algol style.* BIT, Vol. 3 (1963), No. 3, 204-208.

[65] Naur P. *Programming languages, natural languages, and mathematics.* Comm. ACM, Vol. 18 (1975), No. 12, 676-683.

[66] *Occam 2 reference manual.* Inmos Ltd., Prentice-Hall, 1988.

[67] Parnas D. L. *A generalized control structure and its formal definition.* Comm. ACM, Vol. 26 (1983), No. 8, 572-581.

[68] *Pascal ISO/IEC 7185:1990(E)*, 1990.

[69] Peterson W. W., Kasami T., Tokura N. *On the capabilities of While, Repeat, and Exit Statements.* Comm. ACM, Vol. 16 (1973), No. 8, 503-512.

[70] Pratt T. W., Zelkowitz M. W. *Programming languages: design and implementation*, 4th ed. Prentice-Hall, 2000.

[71] Richards M., Whitby-Strevens C. *BCPL: the language and its compiler.* Cambridge University Press, 1980.

[72] Robillard P. N., Boloix G. *The interconnectivity metrics: a new metric showing how a program is organized.* The Journal of Systems and Software, Vol. 10 (1989), No. 1, 29-39.

[73] Soloway E., Bonar J., Ehrlich K. *Cognitive strategies and looping constructs: an empirical study.* Comm. ACM, Vol. 26 (1983), No. 11, 853-860.

[74] Scholtz J. et al. *Object-oriented programming: the promise and the reality.* The Journal of Systems and Software, Vol. 23 (1993), 199-204.

[75] Shivers O. *The anatomy of a loop.* Proc. ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'05), 2005, 2-14.

[76] Shneiderman B. *Software psychology. Human factors in computer and information systems.* Winthrop Publishers, 1980. [превод на руски: Шнейдерман Б. *Психология программирования: Человеческие факторы в вычислительных и информационных системах.* М., Радио и связь, 1984.]

[77] Steele G. L., jr. et al. *Common LISP. The language*, 2nd ed. Digital Press and Prentice-Hall, 1990.

[78] Shaw M., Wulf W. A., London R. L. *Abstraction and verification in Alphard: defining and specifying iteration and generators.* Comm. ACM, Vol. 20 (1977), No. 8, 553-564.

[79] Tennent R. D. *Language design methods based on semantic principles.* Acta Informatica, **8** (1977), 97-112.

[80] Turner D. A. *Miranda: a non-strict functional language with polymorphic types.* Proc. Conf. Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science **201**, Springer-Verlag, 1985.

[81] Watt D. A. *Programming language design concepts.* John Wiley & Sons, 2004.

[82] Whitelaw M. W. *Some ramifications of the exit statement in loop control.* ACM SIGPLAN Notices, Vol. 20 (1985), No. 8, 99-106.

[83] van Wijngaarden A. et al. *Revised report on the algorithmic language Algol 68.* Acta Informatica **5** (1975), 1-236.

[84] Wirth N. *Systematic programming: an introduction.* Prentice-Hall, 1973.

[85] Wirth N. *On the design of programming languages.* Proc. IFIP Congress 74, 386-393, North Holland, 1974.

[86] Wirth N. *From Modula to Oberon.* Software–Practice and Experience, Vol. 18 (1988), No. 7, 661-670.

[87] Wirth N. *The Programming Language Oberon.* `http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon.Report.pdf`, 1990.

[88] Wirth N. *The Programming Language Oberon. (Oberon-07).* `http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon07.Report.pdf`, 2013.

[89] Zahn C. T., jr. *A control statement for natural top-down structured programming.* Programming Symp. Proc., Lecture Notes in Computer Science **19**, 170-180, 1974.