БЪЛГАРСКА АКАДЕМИЯ НА НАУКИТЕ
Институт по математика и информатика

Бойко Блажев Банчев

Проблеми на изразната явност в текстове на програми

А В Т О Р Е Ф Е Р А Т

на 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.

Дисертационният труд съдържа 127 стр. и се състои от увод, изложение в пет глави, заключение с резюме на получените резултати, декларация за оригиналност и библиография с 89 заглавия.

Дисертацията е обсъдена и допусната до защита от определеното за провеждането на предзащита научно звено от 12 члена на негово заседание, състояло се на 17 февруари 2014 г.

Защитата на дисертацията ще се състои на от часа в Заседателната зала на ИМИ-БАН на открито заседание на научно жури в състав:

1. проф. д-р Нели Милчева Манева
2. проф. д-р Аврам Моис Ескенази
3. проф. д-р Магдалина Василева Тодорова
4. доц. д-р Трифон Анчев Трифонов
5. проф. дмн Георги Атанасов Тотков

Материалите по защитата са на разположение на интересуващите се в библиотеката на ИМИ-БАН (София, ул. „Акад. Г. Бончев“, бл. 8).

Съдържание

Увод

В дисертационната работа се изследват свойства на езиците за програмиране. По-конкретно, разглеждат се свойства на текстове на програми – някои обусловени пряко от езика за програмиране, други дължащи се на начина на използването му. Като общ знаменател на разглежданията е избрано въведеното в работата понятие явност.

Явността, заедно с няколко други понятия, показва доколко изразните средства на даден език за програмиране са годни да представят желателни за програмиста свойства на програмни обекти и отношения между тях. 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 работата 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 изясним в известна степен как езиците за програмиране се съотнасят с естествените и с този (или тези) на математиката;
- 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;
- 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 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 изложението 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, тези на математиката и езиците за програмиране

Езиците за програмиране – обект на настоящото изследване – са преди всичко езици изобщо. С оглед на пълноценното третиране на темата на работата е уместно да се потърсят паралели между езиците за програмиране и други видове езици. Глава 1 съдържа 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

Сходства и аналогии между ЕП, ГЕ и езика на математиката са търсени и забелязвани неведнъж. Свойствата на ЕП имат значителна близост с тези на говоримите и на математическия езици, а и в ЕП целенасочено се влагат заемки както от говоримите езици, така и от математическия символен стил на изразяване.

И при говоримите езици, и при ЕП е налице взаимно влияние, като то е по-силно изразено при езиците за програмиране.

Съпоставяйки говоримите езици и ЕП, P. Naur [21] обръща внимание на следната асиметрия: 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, лингвисти правят сравнителна оценка на *зрелостта* на говоримите езици от гледна точка на (отново следвайки [21]):

- 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 прос-

ти 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 [18, 27, 15, 25].

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 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 ([19] и др.)

Естествените езици са основен носител на човешката култура и 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.

Заимстването в ЕП на свойства от ГЕ е необходимо и ползотворно, но не може да става механично и няма правила, които диктуват как да става това. В текста на дисертацията се съдържат, без непременно да бъдат изрично посочени, различни конкретни примери за такова заимстване.

1.2. Математиката и езиците за програмиране

Сходства и различия между математиката и програмирането има както между предметите на едното и другото, така и по отношение на техните писмености.

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. При това и двете науки имат свойството да се самообогатяват съдържателно чрез въвеждане на нови определения и връзки между същностите.

Някои същностни разлики между математиката и информатиката са следните.

• Математиката има и конструктивно, и неконструктивно съдържание, и съответно език. Информатиката борави изключително с конструктивни обекти, а и конструктивизмът често се отличава с конкретна по вид и обем ресурсна обусловеност.

• Математиката изучава няколко вида структури, основно наредбени, алгебрични и топологични. В информатиката преобладават наредбените структури, но в известен смисъл неин предмет е и (дискретната) структурност въобще. Информатиката е наука

за строежа и за построяването.

• Редица еднакво именувани понятия в математиката и информатиката – променлива, функция и др. – са съдържателно различни. Например в математиката понятието *променлива* е тясно свързано с понятието функция и служи за изразяване на отношения между множества. В информатиката променливите могат да се създават, изменят явно или неявно, унищожават и др.

• Специфичните за информатиката и основни за нея понятия *памет, достъп, ресурс* нямат аналози в математиката.

• Особена значимост в информатиката има *фòрмата на представяне* на обектите, например конкретният алгоритъм или програма за решаване на задача. Формата на представяне е и сама по себе си предмет на изучаване. В информатиката и програмирането често се работи наведнъж с няколко равнища на представяне, а те по принцип могат да изискват за описването им различни езикови слоеве.

• Програмирането борави с множество специфични парадигми на различни семантични равнища. Тези на най-високо равнище – императивна, апликативна и пр. – определят изобщо възгледа върху процеса на съставяне на програми и качествата, които програмата като формален обект притежава. 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 наука, няма относително завършения и устойчив вид, който притежават много математически дисциплини. С оглед на това методологическите аспекти в нея са и твърде съществени, и слабо разработени. В програмирането динамично възникват голям брой нови понятия, а някои изменят смисъла си. Това влече висока динамика на развитие и усложняване на езиците за програмиране и затруднява систематичното им изследване.

Съществено влияние върху програмирането оказва традиционно използваната в математиката нотация. Математиката има най-силно развита и сравнително устойчива практика в използването на нотация, а заимстването от нея дава възможност да се използва натрупаният чрез обучението и практикуването на тази наука опит. Достойнствата на математическата нотация обаче не я правят непосредствено използваема в езиците за програмиране. Двете главни причини за това са, че тя обикновено не е *нито еднозначна, нито изпълнима.* Контекстът, в който се интерпретира математическият символен запис, е този на естествения говорим и писмен език и като негово продължение този на математиката е съществено несамостоятелен.

Синтактично и семантично приближаване на нотацията в програмирането до тази в математиката се наблюдава в най-голяма степен в апликативни езици като APL, FP и Haskell.

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

Като бързоразвиваща се система информатиката има изменчиво, еклектично и трудноопределимо съдържание. Това личи ясно в областта на езиците за програмиране в тесен и широк смисъл (вкл. командните езици на операционните системи 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 отражение на света в един или друг негов аспект.

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 мотивираност, но също и 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, във все по-голяма степен свързани с текста и езика, на който са написани. Появява се терминът *структурно* или *систематично* програмиране, формулират се и някои други качествени характеристики 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 [17, 20, 13, 24, 23, 9].

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.

Доколкото напоследък се полагат усилия предимно за развиване на езиковите средства на условно по-високото (*large-scale*) равнище на програмиране – свързано с архитектура, модулност, интерфейси и пр. – създава се впечатление за изчерпаност

или маловажност на решенията, отнасящи се до непосредственото, дребномащабно (*in-the-small*) програмиране. Обилният някога поток от научни публикации върху управляващи и други структури в ЕП отшумя, но предизвикалите го реални проблеми с езиците като цяло останаха. А тъй като повседневната дейност на занимаващите се с програмиране е свързана в най-голяма степен с непосредствената му форма, боравеща с понятия като стойност, променлива, операция, израз, команда, съответните езикови средства влияят най-пряко и са определящи за продуктивността на този вид труд.

Широкоизвестните трудове по систематично програмиране на Е. Дейкстра и Ч. Хоър [10, 11], Д. Грис [16], Н. Вирт [26] и др. са посветени именно на непосредственото програмиране, но в тях се слага ударение повече върху математическото третиране на програмите, отколкото собствено програмирането, а на изразните средства и разнообразието от ЕП почти не се обръща внимание.

Емпирично-психологическите изследвания за възприемане на текстове на програми, макар и полезни с възможността да се забележат или обосноват езикови закономерности, също страдат от едностранчивост. Те третират изолирано, и то много малък кръг конструкции в ЕП. При това изследваните конструкции, а и хората, които проиграват съответните тестове, са най-често така или иначе „контекстнообусловени“ от съществуващите, пряко или косвено известни, подобни конструкции. Почти напълно отсъстват опити да се разглеждат общи свойства на текстовете на „малки“ програми – свойства, проявяващи се в светлината на дребномащабното програмиране, които имат значение за възприемането на програмата при четене.

Обзор на именно такива свойства се прави в глава 2 на дисертацията. Те могат да произтичат пряко от използвания език за програмиране или да се дължат на начина на използването му. Част от тях са в малка или в по-голяма степен субективни, защото зависят например от подготвеността на този, който чете текста на програмата.

• **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**

Това е съответност на използваните елементи на дадения ЕП и на съчетанията от тях с *предварителните представи* на четящия за начина на използването им. Добрата или слаба непосредственост може да е характеристика и на самия език.

• **Адекватност**

Адекватността е друг вид съответност: между програмата като съчетание от езикови елементи и абстрактния модел на решението на задача, който програмата представя.

• **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.

• **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.

• **Разход на памет**

В [3] се използва подобно понятие (*memory load*, натовареност) – обем от допълнителна информация за *други* елементи на програмата, необходим за разбиране на даден фрагмент. Тук разбираме „разхода на памет“ в по-широк смисъл, включвайки и натовареността, проявяваща се при четивна интерпретация на *същия* фрагмент.

• **Вербалност/нотационност**

Отнася се до това, дали в записа на програмата преобладава нотацията, т. е. дали за различните действия се използват голям брой специални знаци или преобладава словесният изказ. Очевидно това свойство се предопределя от използвания ЕП.

• **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.

• **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.

Изброените свойства на програмни текстове не изчерпват факторите, обуславящи възприемането на такъв текст. Например при ниска непосредственост то 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

С понятието явност 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.

Явността на изразяване може да се отнася до всяко свойство или отношение в програмата. Явност или неявност може да присъства във всяко предписано от програмата действие, без значение дали то се извършва при изпълняване на програмата или преди това. Например, ако на езика C трябва да определим два масива така, че да имат едни и същи размери, да речем `[100][50]`, можем да запишем

```
int a[100][50];  float b[100][50];
```

Тогава двата масива действително имат еднакви размери, но от текста на програмата не е ясно дали това е преднамерено или е само съвпадение: явна информация за това липсва. Възможни са различни варианти на определението, при които преднамерената на равенството на размерите е явно изразена. В някои от случаите обаче се появяват други нежелателни свойства. Езикът не предвижда средство за явно изразяване на търсеното отношение и в най-добрия случай то се осигурява по заобиколен начин, приближено, и трудно подлежи на обобщаване. Същото важи, дори в по-голяма степен, и за редица други свойства.

Недостигът на средства за явно изразяване далеч не е белег само на езика C, а и не само на „класическите“ ЕП. Благоприятните в това отношение черти на посъвременните езици все още са твърде малко.

В различни случаи желаната явност може да се осигури чрез коментарен текст към програмата. По няколко причини обаче това е недостатъчно удовлетворително. Друго, също непряко и ограничено средство за повишаване на явността е снабдяването на текста с изпълними контролни предикати като командите `assert` в някои езици:

чрез всеки такъв предикат едновременно се *създава* и се *прави явно* дадено свойство на програмата. Благоприятна за постигане на явност е декларативността в някои езици, най-вече функционалните ML, Haskell, Scala, Erlang F# и др. – тя прави възможно наедно да изразяваме собствено програмата и нейни свойства.

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, неявността на нежелателните свойства може да попречи те да бъдат забелязани и евентуално отстранени.

Друг белег на двойственост на понятието явност е това, че в редица случаи се налага да пренебрегваме едно проявление на явността за сметка на друго. 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.

3.2. Явност и близки понятия, свързани с програми

Две важни понятия, близки до понятието явност, са адекватност и нагледност.

Понятието нагледност по начало е термин на психологията, най-вече – на педагогическата психология. 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.

С появяването на богати възможности за визуално взаимодействие на компютърните програми с техните потребители изучаването на нагледността става особено актуално. Наред с изобретяването на различни конкретни форми за онагледяване се правят опити и да се определи същността на нагледността и ролята ѝ в познавателните процеси.

Дори програми с неголям обем вместват голямо количество информация, затова няма съмнение, че нагледността е важна характеристика на текстовете на програми. Нагледността е тясно свързана с явността, по-точно с визуалните аспекти на възприемането на последната. Тя подпомага явността, като я прави по-непосредствено достъпна. При избор на езикови средства (сред съществуващите или като изобретяваме нови), които да подпомагат осигуряване на висока явност в програмите, следва да предпочитаме такива, които могат да осигурят и нагледност.

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. В частност, терминът се употребява често по отношение на *езиковия* модел,

който дадена наука е развила за описване и изследване на област от нейния интерес. В програмирането адекватността съхранява чертите на общото понятие и в частност – невъзможността тя да бъде осигурена по формален (автоматичен) път.

Явността и адекватността в програми могат да се обуславят една друга по различни начини. Например онзи, който чете дадена програма, може 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 тя зависи от явността, ясно е, че колкото повече *явно изразени съществени за алгоритъма* елементи има в програмата, толкова повече *обективно адекватна* ще бъде 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. 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 [14, 12, 2] 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.

Друго разглеждано при измерване на софтуер понятие е *степен на взаимозависимост* (*coupling*) между програмни обекти, подразбирайки такива от ранга на подпрограми. Предполага се, че взаимозависимостта се определя поотделно за двойки „модули“ и на тази основа – обобщено за програмата.

В [22] се предлага интегрална мярка за *взаимосвързаност*, която съвместно отчита различни характеристики – размер, управленски и даннови връзки – на програмата.

Важна отлика на нашия подход от посочените изследвания е, че при него не само се стремим да характеризираме съществуващи *структури* в зададени програми, 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.

4. Явност и неявност в примери: проблеми и решения

В глава 4 на дисертацията са разгледани 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. Преобладават императивните езици и конструкции, но се засягат и апликативни. Изследван е и проблем от областта на метаезиците. В разглежданите области се предлагат оригинални езикови конструкции, които обобщават или другояче усъвършенстват известните. Те могат да бъдат използвани за повишаване на изразителността на езиците за програмиране, като бъдат включени във вече налични езици или при създаване на нови.

Тук привеждаме най-съществените резултати от глава 4.

4.1. Изрази и прости команди

Присвояващи операции

Така наречените „присвояващи операции“ (*assigning operations* или *compound assignments*), съчетават аритметична или друга операция с присвояване. Това дава възможност да се избягват излишни повторни цитирания на имена и същевременно съдейства за повишаване на адекватността и явността на изразяване. Например изразът `a += b` се чете като „увеличи стойността на `a` с тази на `b`“, което описва точно връзката между двете променливи в действието, докато функционално равнозначният израз `a = a+b` цитира двукратно името `a` и се предава с по-неестественото „събери `a` с `b` и постави резултата в `a`“.

В много езици присвояващи операции няма изобщо или такива отговарят само на част от наличните двуместни операции. В нито един език присвояването не може да се съвместява с *едноместна* операция. Последното принуждава за обръщане на аритметичния знак на стойност и за обръщане на булева стойност на променлива да записваме съответно (на C и др.) `a = -a` и `a = !a`, които не са наистина адекватни и съдържат излишни повторения. Допълнителна аномалия в езика C и сродните му е присъствието на едноместните операции `--` и `++`, аналогични на присвояващите двуместни операции, но с префиксна и суфиксна спрямо аргумента си форми на записване, което влече различна стойност на образувания израз.

Двуместните присвояващи операции и едноместните `--` и `++` подсказват обобщение и наред с това по-просто изразяване както следва.

Нека *всяка* операция – едноместна или двуместна – да може да се съвместява с присвояване и нека знакът `:` придава присвояващо действие на операциите. Записваме `:` като префикс или суфикс към знака на операцията за предизвикване съответно на *преди-* и на *последействие*. Например всеки от изразите `a :+ b` и `a +: b` увеличава `a` със стойността на `b`, но стойността на първия израз е *вече измененатa* стойност на `a`, а на втория – тази *отпреди изменението*.

Преди- или последействие на присвояването може да се избира за всяка операция. Например `:-x` обръща знака на `x` и стойността на израза е вече изменената стойност, а при `-:x` стойността на израза е първоначалната стойност на `x`. Наред с другото, сега едноместните операции могат да се записват еднообразно, само префиксно спрямо аргументите им; отпада нуждата от двувариантност като при `--` и `++`.

Постановяваме също, че всяка присвояваща операция има за резултат L-стойност, т. е. обект, който има стойност, но и на който може да се извършва присвояване. Това става, като с всеки израз с такава операция се *асоциира променлива* – тази, на която операцията присвоява стойност. Така става възможно той да бъде субект на ново присвояване, в частност – да бъде аргумент в друга присвояваща операция, включително и в ролята на асоциираната с операцията променлива.

Израз, стойността на който е L-стойност, наричаме L-израз и така се обособява множество от произволно сложни L-изрази, вложено в това на обичайните изрази.

Обичайното 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. (Сама по себе си операцията `=` не образува L-стойност – тя извършва присвояване, но не асоциира променлива.) Тогава например `b = (a =: b)` предизвиква размяна на стойности между `a` и `b`, а `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` и така получената нова стойност става стойност и на израза: изразът образува числа от редицата на Фибоначи.

Условноприсвояващи операции

Начинът и съображенията за въвеждане на L-изразите водят до още една форма за обогатяване на понятието израз: *условноприсвояващите операции*. Получаващите се от тях изрази наричаме Q-изрази. Последните са 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 операцията, като предполагаме, че той има булева стойност или за по-голяма свобода тълкуваме стойността в духа на C и други езици: нулевата стойност е „неис-

тина“, а всички други – „истина“. Тъй като и условноприсвояващите операции биват преди- или последействени, условието по принцип зависи или от текущата, или от условноприсвояваната стойност на съответната променлива.

Така например изразът `a ?+ i < 80` увеличава стойността на `a` с `i`, ако получаващата се при това нова стойност е по-малка от `80`. При това както имената на променливите, така и самата операция се цитират само по веднъж. `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 `i-1` е положително, т. е. ако `i` е по-голямо от `1`. `(a =? b)==0` присвоява на `a` стойността на `b`, ако `a` текущо има нулева стойност.

Съображенията за полезността и начина на използване на условноприсвояващите операции са подобни на тези за присвояващите. И двете обслужват често срещани в практиката на програмирането, типични случаи. Условноприсвояващите операции имат известно приложение и като механизъм за възвратно програмиране (*backtracking*).

Изразите във функционалния език U

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. Оттам и тежестта на съображенията за нагледност при изрази е голяма именно в програмирането. Това се наблюдава особено добре във функционалните езици, тъй като при тях програмите се състоят почти изцяло от изрази.

По различни причини в някои такива езици нагледността и явността на текста са чувствително по-ниски от обичайното. Например ако езикът е динамично типизиран, кое да е име в различни точки и моменти на програмата може да обозначава различни стойности, включително функции. В някои езици това на свой ред води до динамично интерпретиране, а следователно неявност, дори на строежа на изрази. Редица от три лексеми в езика J може да има до десет различни синтактични интерпретации в зависимост от текущите стойности на участващите имена. В REBOL пък функциите могат да имат различен брой аргументи и например изразът `a b c d e` може да се разчете като `a (b c d) e`, `a (b (c d) e)` и по редица други начини.

Оригинално решение за подобряване на нагледността и явността в изрази се съдържа в предложения от автора [8, 1] функционален език за програмиране U. Образуването и пресмятането на изрази в U е подчинено на няколко особености.

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 служат за аргументи и дори по тип да са функции не подлежат на прилагане.

Така понятието *функция*, наред с други като число, низ и пр., се отнася до вида (типа) 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 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.

Всяка функция в U има точно един аргумент, а за да приема две или повече стойности, образуваме от тях редица: аргументът на функцията е тази редица, а нейните членове условно приемаме за отделни аргументи на функцията. Всяко прилагане в $u\ v\ w$ (на v към u и w) е всъщност прилагане на v към редицата $[u;w]$.

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 и в $f\,.\,x$ той е редицата $[f;x]$.) Така всеки прост израз с инфисно прилагане $u\ v\ w$ може да се запише и като $v\,.\,[u;w]$, което условно приемаме за „префиксен запис“, в смисъл на прилагане на v към $[u;w]$. И обратно, всеки „префиксен“ израз от вида $f\,.\,x$, където x е двуелементна редица $[x_1;x_2]$, може да се запише като $x_1\ f\ x_2$. С помощта на операцията `=>`, която е като `.`, но с обърнат ред на аргументите (`f.x` ≡ `x=>f`) може да се имитира и суфиксен запис.

Описаният механизъм за образуване и пресмятане на изрази е прост и напълно регулярен, а наред с това достатъчно гъвкав. Особено важно е, че дори при динамично типизиран език този механизъм не допуска неоднозначност на синтактичното интерпретиране и породената от това неявност.

Съюзи

Някои езици за програмиране предлагат операции, образуващи стойност чрез избор на една от две стойности, без да извършват пресмятане над тези стойности заедно. Пример за такава операция е `,` (запетая) в езика C, която от двата си аргумента e_1 и e_2 дава стойността на втория. APL има две аналогични операции: $\vdash$ и $\dashv$, които дават стойността съответно на e_2 и e_1.

Такава операция има смисъл само ако пресмятането на единия аргуент, 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 втория аргуент.

Понеже от една страна се образува стойност, а от друга – действието на операцията в определен смисъл не зависи от аргументите ѝ, уместно е да имаме особено име за такъв тип операции. Подходящ термин, по аналогия с естествения език, е „съюз“.

Споменатите 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. Подобно нещо се наблюдава в езика Icon при операциите `&` и `|`, аналогични на логическите връзки „и“ и „или“, но използващи сигнала за успех или неуспех, образуван при пресмятането на израз.

Направеното наблюдение подсказва възможността да потърсим и други варианти на съюзно съотнасяне. Такива възникват например във връзка със самоунищожаването на действия в рамките на консумативния модел на пресмятане, разгледан по-нататък.

4.2. Съставни команди

Езиковите средства за управляване на следването на командите, наричани за удобство управляващи команди, са една от най-разискваните теми във връзка с адекватността, явността, нагледността и други черти на възприемане на текста.

Конструкции за цикъл

Сред управляващите конструкции тези за задаване на итеративност (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 на елементите на конструкцията помежду им и с другите части на програмата.

В дисертацията се прави обзор на редица особености и проблемни точки, отнасящи се до езиковото изразяване на итеративност: въпроси, свързани с алтернативно завършване, инициализиране, финализиране, локалност, недетерминираност, абстрактно изразяване и др. Всички те имат пряка връзка с явността на изразяване, адекватността и нагледността на текста. Особено се посочва и се обосновава, че характерен и винаги проявяващ се недостатък на конструкции за цикли в много езици за програмиране е отсъствието на синтактично обособени дялове за инициализиране и реинициализиране (обновяване) като действия от общ вид, което обуславя слаба адекватност и явност на изразяване в програмите.

Консумативни команди и задаване на цикли

В дисертацията се показва чрез примери, че за множество типични случаи на

итеративност решенията в рамките на обичайните средства за задаване на цикли неизбежно съдържат подчертана неадекватност. Причината е, че някои от действията в телата на циклите трябва да се изпълняват еднократно, или до удовлетворяване на някакво условие, или след него, да бъдат заменяни с други и пр., за което традиционните езикови конструкции не предоставят средства.

За по-адекватно и явно изразяване на такива свойства въвеждаме подходящо за случая уточнение на обичайната представа за императивния модел на пресмятане, което наричаме модел на *консумативно пресмятане*, и наред с това – набор от конструкции, които използват този модел. Това е предложено от автора в [4].

Приемаме, че *всяка* команда в програмата се поражда непосредствено преди да бъде изпълнена и се унищожава веднага след това. Условните команди пораждат само един от клоновете си. Командите за цикъл има 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, въвеждаме и *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 основно в цикъл.

Следните консумативни команди се определят неформално, всяка с обща схема и пояснение. 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 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. От друга страна, при наличие на такива команди, за изразяване на итеративност е всъщност достатъчна само една, възможно най-проста команда за цикъл.

Нека командата `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 цикъла – се изпълнява, докато в него има поне една неунищожена 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) T`	цикъл със средусловие

Влагането на консумативни команди една в друга дава възможност в структурата на *всяко* действие да се обособят еднократно или неколкократно изпълнявани части. В следния фрагмент се предполага, че дадени действия се изпълняват многократно, докато условието в `before` не се изпълни. Наред с другото се търси дали при някоя от итерациите се удовлетворява условието `P`. Например може да се обхожда информационен масив и да се проверява дали някой от членовете му изпълнява `P`.

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

Командата `after` осигурява проверката да се извършва само до първото удовлетворяване на `P`, а ако това се случи, да се изпълни присвояването `flag = 1`. На свой

ред, благодарение на `once` последното се случва само веднъж. И така, необходимото присвояване се извършва ако и само ако `P` се окаже поне веднъж вярно, като нито проверката, нито присвояването се изпълняват повече, отколкото е нужно.

Следният фрагмент поражда редицата на Фибоначи $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$. При всяко изпълнение в променливата `t` се появява поредното число от редицата:

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

Командата не зависи от какъвто и да е конкретен повторителен контекст и може да бъде поставена в произволен такъв.

4.3. Агрегиране

Една от най-простите форми за образуване на съвкупности от стойности са n-те (енторки, tuples): къси редици от възможно разнородни стойности. В някои езици за програмиране енторките са един от видовете стойности: такива са например повечето съвременни функционални езици. В други 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 до това да се допуска множествено присвояване и функция да дава като резултат няколко стойности вместо една. Образуването на енторка и боравенето с частите ѝ в тези езици е по-свободно, отколкото ако енторките са вид стойности. От друга страна, 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.

В [5] предложихме 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.

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

• Съвместно присвояване, в конкретния случай – циклична замяна на стойностите 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. Резул-

татът 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. Енторките 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

Разглеждането тук се отнася до език не за програмиране, а имащ косвено отношение към него. Тъй като този език е формален и същевременно предназначен за четене от човек, съображенията за адекватно, явно и нагледно изразяване важат за него така както и за езиците за програмиране.

Предмет на интерес в случая е представянето на редици чрез средствата на метасинтактичен език. Редиците са основен вид синтактична структура и 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. В първоначалния, твърде минималистичен BNF, отсъства конструкция за изразяване на повторност, поради което синтактичните редици от всякакъв вид се представят в него изключително чрез рекурентни определения.

Рекурентните определения имат сериозни недостатъци, свързани с посоката на рекурентността. Непразна редица s от елементи от някакъв (граматичен) тип a може да се определи чрез ляворекурентно или дяснорекурентно правило, именно

$$s \;=\; a \mid s\ a \qquad \text{или} \qquad s \;=\; a \mid a\ s$$

От какво може да е обусловен изборът между двете и какви последствия има той? Видът на рекурентността в правилото интуитивно се асоциира 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 (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 по принцип е възможно, но не е непосредствено и изисква допълнителни действия и ресурси. Така 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 извършването им. 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, 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 схемата „отдолу нагоре“, пораждащият 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.

Следователно изборът между лява и дясна рекурентност за представяне на редици се натъква на 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.

BNF има няколко широкоизползвани понастоящем наследника, например ABNF (Augmented BNF) и EBNF (Extended BNF, ISO/IEC 14977:1996(E)). Всички те в един или друг вид допълват езика с повторителна и условна конструкции и някои други. В частност, така задаването в тях на еднообразни редици може да става не само рекурентно, а и непосредствено. Проста редица s като горната се задава чрез правило с повторител, съответно

$$s \;=\; a* \qquad \text{или} \qquad s \;=\; a\,a*$$

за възможно празна и непременно непразна редица. (Използваме знака * да означава „повтаряне 0 или повече пъти“.) Ако периодично се повтаря не единствен член (синтактична категория), а няколко поредни, достатъчно е да се посочи, че * се отнася за цялата група:

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

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

Много видове редици в езиците за програмиране се задават с редуване на „същински“ членове и разделители. Например в някои езици такива са:

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

В първия и втория примери същинските членове на редиците са съответно определения и изрази, а разделителите – запетаи. В третия членовете са команди, а разделителите – точки и запетаи. Аритметичните и други изрази са на свой ред също редици с членове първични изрази и разделители инфиксни операции. Например вторият аргумент във втория пример е редица от три члена, разделени с + и /. 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. За синтактичното правило за такава редица отново има ляво- и дясноректурентен варианти, съответно

$$s \;=\; a \mid s\,b\,a \qquad \text{и} \qquad s \;=\; a \mid a\,b\,s\,,$$

но те имат същите недостатъци като при простите редици.

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

$$s \;=\; a\,(b\,a)* \qquad \text{или} \qquad s \;=\; (a\,b)*\,a\,,$$

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

Итеративните представяния на редица, подобно на ляво- и дясноректурентните, са несиметрични, докато самата редица е симетрична. Освен това 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 използван за задаване на граматиката на езика U [1]).

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, тъй че има припокриване с функцията на `{` и `}`.

В нашето определение припокриването изчезва, а двете двойки скоби сполучливо взаимодействат за изразяване на различни структури, и по-специално описват редици просто, адекватно, явно, нагледно и без излишни елементи. Примери:

- непразна и възможно празна редица с елементи a: съответно `{` a `}` и `[{` a `}]`;
- редица от циклично повтарящи се елементи: `{(` $a\,b\ldots$ `)}` или `{` $a\,b\ldots$ `{}}`;
- редица с един вид или редувани няколко вида членове и разделител z: `{` $a\ldots z$ `}`;
- подобна редица, но с незадължителни разделители: `{` $a\ldots$ `[` z `]}`.

5. Обща концепция за структурността в програми

Формите, в които се проявяват свойствата на текстове на програми, включително явността и близките до нея, са твърде многообразни за да подлежат на изчерпателно изследване. Многообразни са и изобщо носителите на тези свойства – езиците за програмиране. Тази многоликост подтиква към разглеждане на друго равнище, извън конкретността на езиковите конструкции и понятия: равнище, задаващо обща концептуална основа, върху която да се формулират и обясняват свойствата на езиците и изобщо – да се разглеждат обекти с изчислителна (информатична) природа.

Водещо в това разглеждане е 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 се оформя постепенно от различни източници. Един от тях е структурното програмиране, чиято идеология предписва програмите да се изграждат йерархично от малък брой отнапред избрани конструктивни схеми. Във връзка с това се появяват понятията *даннова* и *управленска структура*, отговарящи на взаимосвързани слоеве в програмата.

Структурни представи възникват и във връзка с формалното измерване на сложност на програми, а също, макар косвено и откъслечно, и при формализмите за конструктивно специфициране на програми като VDM и Z.

Основен принос за оформяне на представи за структурността има еволюцията на данновите и управленски структури в практиката на програмирането и дестилирането на този опит в езиците за програмиране.

5.2. Принципи

При оформяне на общата концепция за структурност в програми следваме някои основни положения като принципни. Един такова положение 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 спецификация за задачата и на абстрактен модел на решение, е процес на *изявяване на обективни структурни свойства и отношения*.

Принципът за обективност на структурността може да се онагледи с различни примери, някои от които се привеждат в текста на дисертацията. От него следва и

че структурата на програма е относително независима от съдържанието ѝ – в този смисъл *структурата е сама по себе си носител на обективно съдържание* – и толкова по-важно е да имаме универсален език за третирането ѝ.

Друг основен принцип е, че се интересуваме от тъкмо *този вид структурност, който е характерен за информатичните обекти*. Във всяка област на знанието се обособява специфичен за обектите в нея вид структурност. В информатиката и конкретно в програмирането предизвикателство е да се облече в понятия нейният (неговият) собствен вид структурност. Наистина, различните парадигми на програмирането и дори отделните езици за програмиране могат да имат твърде различни системи от понятия за описване на еднакви или сходни същности. От друга страна, „информатичност“ съществува и в природата: елементи на информатично поведение има във физични, биологични и изобщо всевъзможни обективно съществуващи системи.

Още един принцип в концепцията за структурност е принципът за *структурно единство и съответствие между статично и динамично*. По-конкретно:

- 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.

В контекста на програмирането посоченият принцип означава, че основните структурни форми и отношения, чрез които описваме данни – следване, избор, повтаряне и пр. – са същите, с които задаваме и структурата на действията. 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.

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. Все пак опитът с конкретни проявления на структурност в програми и езици, обобщен и абстрахиран, дава възможност да формулираме някои независещи от конкретни езици и парадигми и заедно с това намиращи множество проявления в тях понятия.

В [6] 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 [7] предложихме формирането на общоинформатични понятия да става като в речник, като понятията бъдат

групирани в смислови гнезда. Дадохме и примери за такова групиране. Следват някои от тях с кратки описания и конкретни проявления.

- **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, 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 предложен 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: [4, 5, 6, 7, 8].

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

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

[3] 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.

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

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

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

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

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

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

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

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

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

[13] 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.

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

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

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

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

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

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

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

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

[22] 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.

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

[24] 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.]

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

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

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

Issues of Explicit Expression in Program Texts

Boyko Bantchev

(abstract)

In the dissertation we study properties of program texts and ones of the respective program languages. Central to this study is a program text feature that we name *explicitness* and define therein, along with the closely related *adequacy* and *clearness*. From this standpoint, some of the most important types of programming language structures and particular constructs are being discussed. New solutions are proposed that could complement, augment, or replace the commonly used ones, so as to achieve improvement of program text with respect to the properties under consideration.

Generalising on the idea of structure within programs and programming languages, a new approach is outlined for conceptualizing the notion of structuredness as applying to systems of computational nature.