ПРОГРАМИРАНЕТО КАТО МЕТАСМЯТАНЕ

Бойко Банчев

Институт по математика и информатика – БАН
boykobb@gmail.com

***Резюме**: Описват се елементи на подход за обучение по програмиране в училище, основан на функционалния стил. Подходът хармонично съчетава програмиране, смятане и елементи на алгебрата, осигурявайки естествена връзка с математиката. Смятаме, че прилагането му би довело до плавно, свободно от изкуствени понятия и конструкции навлизане в програмирането и заедно с това – до разширяване и задълбочаване на материала, изучаван по математика в училище.*

***Ключови думи**: смятане, алгебра, програмиране*

Програмиране и смятане

Ефективността на обучението по програмиране много зависи от използвания общ модел на, или възглед за, програмирането. Той определя що е програма, от какво се състои и по какъв начин тя се образува от частите си. От съществено значение е и конкретният език за програмиране, чрез който се избират и налагат уточняващи общия модел правила.

Не е трудно да се забележи, че най-простият общ модел е функционалният. Според него програмата е преобразовател на информация: по зададена стойност тя намира друга. Казано иначе, програмата е функция. И обратно – всяка функция може да се разглежда като програма. Аргументи и резултати на функциите могат да бъдат всякакви по състав и сложност стойности.

Изпълнението на програма се състои в пресмятане на стойността на съответната функция по зададен аргумент. В повечето случаи пресмятането е сложно действие; тогава то се изразява в извършване на редица „по-малки" пресмятания на други функции, те също се пресмятат чрез по-прости и т.н. до достигане на такива, които са представени непосредствено в езика за програмиране. Всяко извършвано от програмата действие, просто или сложно, е пресмятане на функция.

Съставянето и пресмятането на изрази, наред с извършване на различни преобразования, са отлично познати от училищния курс по алгебра. Тъй като именно до това се свежда и функционалното програмиране, тъкмо то предоставя най-плавния – с *най-малко на брой необходими нови понятия* – път за навлизане в програмирането. То има и предимството на отлична пригодност за непосредствено, диалогово взаимодействие с програмиращия: задаването на израз и получаването на резултат от пресмятането му е естествена и адаптивна по предметен обхват и сложност комуникационна единица. За сравнение,

непосредствеността изчезва, когато се налага натрупване на определения, образуване на главна програма, компилиране и свързване – всичко това преди и за да може да се задейства изпълнението на програмата – какъвто е случаят с езици като C/C++, Pascal, Java и др.

Разбира се, в сравнение с училищната алгебра, програмирането борави с по-разнообразни по вид стойности, с повече операции, като при това множеството и на видовете стойности, и на операциите може да бъде разширявано в самите програми. Близостта между двете области обаче е очевидна и значителна. Затова удивително и твърде жалко е, че в практиката тя е напълно пренебрегната.

Цел на тази статия е чрез примери от избрани предметни области да даде представа за възможностите за сближаване на преподаването на програмиране, в рамките на функционалния модел, с това на „училищна“ математика. Този подход благоприятства обучението не само по програмиране, но и по математика, като разширява и обогатява предметното съдържание на второто.

Смятане и алгебра

Боравенето с аритметични и други изрази е свързано с две равнища на знание за участващите в тях операции. За *пресмятане* на изрази трябва да умеем да извършваме операциите. За *съставяне и преобразуване* на изрази е нужно да познаваме *свойствата* на операциите. Вторият вид действия превръща „обикновеното“ смятане в алгебра. Обект на внимание за алгебрата са повече самите изрази, отколкото стойностите, които те образуват. В този смисъл тя е „метасмятане“: смятане с изрази (образуване и преобразуване) вместо пряко със стойности.

Отликата на манипулирането на изрази от непосредственото пресмятане се проявява и като отлика на програмирането от изпълнението на програми. Съставянето на програма е съставяне на израз, а изпълнението ѝ – пресмятане на израза.

Изразите се записват в една или друга конкретна форма, в която за реда на извършване на действията и други подробности съществуват определени подразбирания. Правилата от този вид са също важни за боравенето с изрази и тази важност се откроява особено в програмирането, където броят на различните операции е много по-голям от този в „училищната“ алгебра. Затова е добре тези правила да бъдат малко на брой, естествени и прости. Но тъй като те са не същностни свойства на операциите, а само на представянето им в езика, тук не им обръщаме повече внимание.

Манипулирането на изрази се основава на свойствата на операциите. Преобладаващите свойства са само няколко на брой и се притежават от различни операции. Някои свойства съотнасят действието на една операция с това на друга. Съвкупността от притежавани свойства характеризира операцията сред

другите, а съчетаването на операции в израз поражда интересни възможности като откриване на зависимости между обекти или между начините на построяването им и в частност – начини за построяване на едни обекти (стойности на изрази) чрез други.

Да припомним най-често срещаните алгебрични свойства на операции. От училище те са познати във връзка с операциите над числа, но в следващите части ще разгледаме съвсем други видове обекти и други операции, където се проявяват все същите свойства. Така различни предметни области се оказват родствени чрез наличието в тях на операции с едни и същи свойства. Това добре известно откритие на *абстрактната алгебра* подчертава естествеността на функционалния подход в програмирането. То е и предпоставка както за обогатяване на преподаваното по математика в училище, така и за сближаване между него и това по програмиране.

Едно свойство, така привично, че рядко му се обръща внимание, е *запазващото*. Такова свойство има операция, резултатът от прилагането на която е същият вид стойност като този на аргумента или аргументите ѝ. Някои операции с числа са запазващи, но не всички са такива, а някои – само условно. Например сравняването по големина не е запазващо действие. Ако разглеждаме само цели числа, то и деленето не е, а ако разглеждаме само цели положителни – и изваждането не е.

Друго привично и малко забелязвано свойство е съществуването на *„пасивен“ елемент* – стойност, която при действието на операцията оставя неизменна която и да е друга стойност. Такава е 0 за събирането и 1 за умножението. 0 и 1 са пасивни и съответно за изваждането и деленето, но само като десен аргумент – не всяка операция има двустранно пасивен елемент. За степенуването 1 е дясна пасивна стойност и ляв „поглъщател“: стойност, която при действието на операцията превръща която и да е друга стойност в себе си (напр. $0\cdot n\equiv 0$, $0^n\equiv 0$ и $1^n\equiv 1$). По-нататък срещаме операция, за която двустранно пасивна стойност е ∞ (безкрайност)!

Разместителното, съдружителното и разпределителното свойства също се прилагат широко. Разместителни и съдружителни са събирането и умножението ($a+b\equiv b+a$, $a\times b\equiv b\times a$; $(a+b)+c\equiv a+(b+c)$, $(a\times b)\times c\equiv a\times(b\times c)$), а също логическите действия „и“ и „или“. Изваждането, деленето и степенуването не са разместителни. Разпределително свойство има умножението спрямо събирането и изваждането – $a\times(b\pm c)\equiv(a\times b)\pm(a\times c)$, $(a\pm b)\times c\equiv(a\times c)\pm(b\times c)$. Имат го и логическото „и“ спрямо „или“, и логическото „или“ спрямо „и“. Деленето е ляво-, но не дясноразпределително спрямо събирането и изваждането: $(a\pm b):c\equiv(a:c)\pm(b:c)$, но не $a:(b\pm c)\equiv(a:b)\pm(a:c)$.

Алгебричните свойства съвсем не са безразлични на информатиката. Например благодарение на съдружителното свойство прилагания на една и съща операция могат да се извършват в произволен ред, а значи и паралелно едно

на друго. Да речем, $a+b+c+d+e+f+g$ може да се пресметне, като едновременно намерим $a+b$, $c+d$ и $e+f$, после, също едновременно, $(a+b)+(c+d)$ и $(e+f)+g$ и накрая общия сбор, с всичко три стъпки вместо шест последователни събирания. Аналогично, събиране или друго съдружително действие над 100 стойности може да се извърши с всичко седем стъпки. Такъв вид разглеждания определено заслужават място в училищната информатика.

Редици

Структурата редица, или наредено множество (за всеки два елемента на редицата единият предхожда другия), е една от основните за информатиката. Като структура от данни в различните езици за програмиране тя е различно пълноценно представена, а в училищното програмиране, като правило – зле. Вместо редици изобщо се разглежда тяхно конкретно, ограничено въплъщение – почти винаги масиви, понякога списъци – и то по начин, специфичен за даден език. Съдържанието на масивите се третира поелементно вместо цялостно; вместо общи свойства и действия с редици обект на внимание са имена, размери и индекси на масиви, устройството на цикли и други езикови конструкции, които нямат пряко отношение към редиците като вид структура.

Показателно е, че работата с текстови низове – частен случай на редици – се разглежда като нещо напълно отделно, без връзка с масиви.

Съвсем излишна за училищното програмиране, а и обърквща е постановката за масиви като за *поначало празни вместилища*, които биват запълвани със стойности-елементи, вместо самите масиви да бъдат цялостни стойности, и като другите стойности образувани с подходящи действия. Упражненията по въвеждане и извеждане – също поелементно – на масиви изобилстват, но те са и безинтересни за изпълняване, и без обучителна ценност.

За илюстрация на по-съдържателно третиране на редици да разгледаме две операции с тях. Едната е *съединяване*: образуване на редица чрез разполагане на две дадени една след друга. По-точно, като бележим тази операция с | и A и B са две редици, $A|B$ е редицата, образувана от членовете на A, следвани от тези на B. Другата операция е *обръщане* на редица, което тук бележим с ¬: за коя да е редица A обратната на нея е $\neg A$.

Лесно се съобразява, че съединяването на редици е запазваща и съдружителна операция с двустранно пасивен елемент. Последният, разбира се, е празната редица. Налице е прилика със събирането на числа, но разместително свойство съединяването няма. Обръщането е подобно на обръщане на аритметичен знак на число, намиране на реципрочно число, логическо отрицание (откъдето заимстваме знака ¬) и др., и също като тях е самообратна операция: $\neg\neg X = X$.

Редиците са проста, лесно разбираема структура и удобно се онагледяват чрез примери с букви, думи, числа, геометрични точки, картини и др., включи-

телно и редици от разнородни стойности. Също прости и нагледни, а същевременно и абстрактни, не зависещи от вида на елементите на редиците са и посочените операции с тях. Това е отлична възможност за усвояване на абстрактно мислене в информатиката, подсилвана и от подобието между операциите с редици и тези над числа.

Да разгледаме някои прости, но важни следствия от съвместната употреба на операциите ¬ и |. В изрази, където се появяват и двете операции и предимството не е изрично посочено със скоби, приемаме, че ¬ се прилага преди |.

Налице е следното свойство. За кои да е две редици L и R: $\neg(L|R) \equiv \neg R|\neg L$.

Казано другояче, както и да разделим дадена редица на две части L и R, обърнатата редица е съединение на обратните на редиците R и L – именно в този ред. Това свойство прилича на разпределителното, за ¬ спрямо |, но понеже L и R разменят местата си, да го наречем „антиразпределително“.

Как можем да се уверим, че свойството винаги е изпълнено? Понеже то се изразява във вид на тъждество – което впрочем е вярно и за редица други свойства, например съдружителното – то е в сила точно тогава, когато редиците от двете страни на знака ≡ съдържат едни и същи елементи в един и същи ред. Последното е еквивалентно на това всеки два съседни в едната редица елемента да са съседни и то в същия ред и в другата. Така имаме общ подход за доказване на свойства на действия с редици.

Ако разглеждаме ¬ и | не само като алгебрични операции, а и като програмно изпълнявани действия, забелязваме, че тъждеството $\neg(L|R) \equiv \neg R|\neg L$ посочва как работата по обръщане на редица може да се разпредели между двама изпълнители, а като делим всяка от редиците L и R също на две и т.н. – между много изпълнители. Антиразпределителното свойство на ¬, заедно със съдружителното на |, осигурява възможността ¬ да се изпълнява паралелно над части от редицата, които после също паралелно да бъдат съединени.

Същото свойство непосредствено води до тъждеството $R|L \equiv \neg(\neg L|\neg R)$, което пък ни дава алгоритъм за *размяна на две съседни части на редица* чрез обръщане: по какъвто и начин редицата да е разделена на части L и R, местата им могат да се разменят, като обърнем поотделно L и R, а после обърнем цялата редица! Алгоритъмът е прост за изпълнение и извършва малко на брой действия, следователно и бърз. Освен това, тъй като обръщането може да се извърши на място, без използване на допълнителна памет, това е вярно и за размяната. Да отбележим, че намирането на решение с тези свойства изобщо не е a priori очевидно. Те правят алгоритъма не само теоретично интересен, но и ценен на практика.

Същият подход може да се приложи за решаване и на по-общата задача за размяна на *несъседни* части от редица – също просто, бързо и на място. Именно, прилагането на антиразпределителното и съдружителното свойства дава $\neg(\neg L|\neg M|\neg R) = \neg((\neg L|\neg M)|\neg R) = \neg(\neg(M|L)|\neg R) = \neg(\neg(R|(M|L))) = \neg(\neg(R|M|L)) =$

R|*M*|*L*. Това значи, че за получаване на подредбата *RML* от редицата от три части *LMR* можем да обърнем поотделно всяка от частите *L*, *M* и *R* и после да обърнем цялата редица.

Естествено е оттук да потърсим и по-нататъшно обобщение. Не е трудно да се докаже, че ако дадена редица е разбита на *какъв да е* брой части, *всяко разместване* на тези части може да бъде направено без допълнителна памет, просто и с неголям брой операции. Отворен остава въпросът до *какъв най-малък брой* могат да се сведат тези операции.

Освен разгледаните дотук действия с редици, има много други, които заслужават внимание – толкова много, че е добре да се разделят на групи. Например едни действия служат за пораждане на редици, други за преобразуване на елементите им, за подбор на подредици по зададени условия, за извличане на части или отделни елементи и т.н. Твърде полезно е действията да се разглеждат именно по този начин, тъй като така стигаме до общи, удобно параметризирани определения.

Така например обхождането на редица отляво надясно с прилагане на „натрупваща“ операция може да се представи чрез израз от вида $(\ldots((x_1 \odot x_2) \odot x_3) \odot \ldots \odot x_{n-1}) \odot x_n$, където $\odot$ е тази операция. Следователно обхождането е операция (функция) с два аргумента: $\odot$ и редицата.

Да бележим тази операция със знака >, както в езика за програмиране U [1]: ако *X* е редицата $x_1, x_2, \ldots, x_n$, изразът $\odot$>*X* е еквивалентен на горния. Така ако *X* е числова редица, +>*X* дава сбора на елементите ѝ, *>*X* – произведението и пр. Ако редицата съдържа коефициентите на полином, изразът `@{[v;c]::v*t+c}`>*X* пресмята по правилото на Хорнър стойността на полинома за `t`. Безименната двуаргументна функция `@{[v;c]::v*t+c}`, в ролята на $\odot$ тук, извършва повтарящото се при пресмятането по правилото действие – умножаване с `t` и добавяне на поредния коефициент.

Примерът с операцията > (в различните езици за програмиране наричана *reduce*, *fold* и пр.) показва, че операциите от общ вид често са „свръхфункции“ – функции, имащи за аргументи други функции. Така всяка от функциите/операциите на различните равнища на прилагане съсредоточава смисъла на конкретно, добре обособено действие, а йерархията на изразите с участието на подобни функции отразява точно йерархията на действията.

Освен това, такива общи операции над редици отново подчертават важността на (фактически – необходимостта от!) третиране на редиците като цялостни обекти. Именно то позволява да формулираме нужните ни действия във вид на добре обособени операции.

Листове и листови фигури

Да отбележим, че ако елементите на редица са равнинни фигури, обръщането ѝ може да се представи като въртене в равнината на ъгъл 180°. (При това, разбира се, фигурите не само сменят подредбата си по хоризонтала, а всяка от тях се обръща „с главата надолу". За да е пълно съответствието с обръщане на редица, последното или трябва да се пренебрегне, или всяка от фигурите поотделно също да се завърти на 180°.) Оттук и всяко разместване на части на редицата с помощта на обръщане може да се представи като последователност от равнинни завъртания на частите.

Вместо какви да е равнинни фигури в посоченото съответствие можем да вземем правоъгълници или правоъгълни таблици (все едно от числа, букви или друго) с хоризонтални и вертикални страни. Да наречем такъв обект *лист*. Казаното за обръщане и разместване на редици от фигури важи в частност и за редици от листове.

Листовете представляват интерес не само във връзка с редици, а преди всичко сами по себе си. Действията с тях, особено с таблици, се онагледяват лесно чрез печатан текст и така са достъпни във всяка компютърна среда. Конкретен вид листове, а по-точно конкретен вид таблици, са обичайно разглежданите в училище двумерни масиви.

Да изследваме някои действия и задачи, възникващи във връзка с листове, които отново ни връщат към важността на алгебричния подход.

По колко начина може да се разположи лист, като се запазва характеристичното му свойство – страните му да са хоризонтални и вертикални? Като отчитаме и възможността листът да се обръща откъм другата си страница, а не само да се върти в равнината, получаваме осем различни положения. (Всеки от четирите ръба може да бъде „горен" и като такъв да бъде ориентиран по два начина – с „лице" и „гръб" на листа.)

Как от някое положение на лист може да се стигне до кое да е от останалите? Лесно установяваме, че всички нужни трансформации са следните:

- ↻ завъртане на 90° по посока на часовника;
- ↺ завъртане на 90° срещу часовника;
- ⌒ завъртане на 180°;
- \ обръщане около левия диагонал;
- / обръщане около десния диагонал;
- — обръщане около хоризонтална ос;
- | обръщане около вертикална ос;
- • идентитет (не променя листа).

Наистина, изброените са осем и всяка води до различно разполагане на листа, т.е. имаме по една трансформация за всяко възможно разполагане.

Последното означава, че всеки от изброените знаци може да бележи както самата трансформация, така и резултата от нея – положението, което при действието ѝ заема листът спрямо някое изходно.

Ако приложим последователно две такива трансформации, листът заема все някое от осемте си положения, следователно резултатът е като да сме приложили вместо двете само една трансформация. Следната таблица описва всички такива случаи – всяка клетка съдържа трансформацията, равнозначна на последователно прилагане на трансформациите, с които са белязани съответните на клетката ред и стълб:

	•	\	/	—	|	↳	↲	⌒
•	•	\	/	—	|	↳	↲	⌒
\	\	•	⌒	↳	↲	—	|	/
/	/	⌒	•	↲	↳	|	—	\
—	—	↲	↳	•	⌒	/	\	|
|	|	↳	↲	⌒	•	\	/	—
↳	↳	|	—	\	/	⌒	•	↲
↲	↲	—	|	/	\	•	⌒	↳
⌒	⌒	/	\	|	—	↲	↳	•

Например ↳, последвано от —, дава \.

Понеже нито операциите, нито строежа на горната таблица зависят от листа или изходното му положение, последователното прилагане или композицията на трансформации е операция *над тях*, която има запазващо свойство. Таблицата описва действието на тази операция – тя е един вид „таблица за умножение“ на трансформации.

Забелязваме също, че за композицията • е пасивен елемент и че тя не е разместителна, но е съдружителна. Освен това спрямо нея всяка трансформация има обратна: такава, че композицията на дадената и обратната ѝ трансформации е пасивната трансформация. (Всъщност повечето трансформации са обратни сами на себе си.)

Наличието на обратни трансформации означава, че наред с „умножението“ горната таблица задава и „делене“: ако последователното прилагане на трансформации T_1 и T_2 записваме с T_1T_2, а обратната на трансформация T – с T', то за кои да е две трансформации A и B трансформацията, която трябва да се приложи *преди* A, за да получим B, е BA', а онази, която приложена *след* A води до същото е $A'B$. Поради неразместителността се налага да различаваме леви и десни обратни трансформации и ляво и дясно „делене“.

Предвид описаните свойства, на езика на алгебрата казваме, че трансформациите на лист образуват *група относно композицията* между тях.

Всички трансформации могат да се образуват чрез последователно прилагане на само две подходящо избрани от тях. Такива *образуващи* могат да бъдат кои да е две от тройката \ — ↳, т.е. \ и —, \ и ↳ или — и ↳, както и всяка двойка, получена от тези три чрез замяна на единия или и двата ѝ члена, \ с /, — с | и ↳ с ↲: всичко 12 възможности. Изборът на друга двойка, да речем \ /, — | или ↳ ↲, не води до образуване на всички трансформации, а само на *подгрупа*.

За заниманията по програмиране интерес представляват и двата аспекта на трансформациите – и отношенията между тях самите, и въздействието им върху листове. Например, ако листът е правоъгълна таблица и е представен като редица *S* от редици, трансформациите също се свеждат до действия с редици, самата *S* или нейните членове: — е обръщане на *S*, а \ е обхождане отляво надясно на всички членове (редове) на *S* заедно, като на всяка стъпка съответните елементи (стълб в дадената таблица) образуват редица в таблицата-резултат. Другите трансформации са малко по-сложни за непосредствено реализиране, но то може да се избегне, като ги изразим чрез споменатите две.

Както при редици, изразяването на тези и други преобразования на таблици съществено печели от третирането на таблиците като цялостни обекти.

Разглеждането на таблици и свеждането на действията с тях до такива с редици подсказва възможността за разглеждане и на „кутии“ (паралелепипеди) и дори на подобни конструкции с още по-голяма размерност.

Геометричното третиране на листове също поражда интересни задачи. Да наречем отношението на ширината към височината на лист негов *формат*. Форматът не е свързан с големина (площ) и изобщо тук смятаме, че листът няма големина, а само формат. Два такива листа могат да се съединят, образувайки нов, също такъв, като ги поставим *един до друг* с еднакви височини (запазвайки формàта на всеки). Съединяване може да стане и като поставим листовете *един над друг* с еднакви ширини. Ако форматите на листовете са r_1 и r_2, в първия случай резултатът има формат r_1+r_2, а във втория – $1/(1/r_1+1/r_2)$.

За удобство на цитирането да означим втората операция над r_1 и r_2 с $\oplus$ и да я наречем *реципрочно събиране*. Операцията $\oplus$ е запазваща, разместителна, съдружителна и неин пасивен елемент е ∞ (безкрайност) – това е добър пример за ползата от ∞ като пълноправна стойност в пресмятанията (каквато ∞ не е нито в математиката, нито в повечето езици за програмиране). $\oplus$ е подобна на т.нар. *средно хармонично*, което обаче не е съдружително и няма пасивен елемент, а името „реципрочно събиране“ е оправдано и от разместителността и съдружителността, и от взаимната реципрочност на пасивните елементи 0 на + и ∞ на $\oplus$.

Операциите + и $\oplus$ могат да бъдат отнесени не само към форматите на листове, а и към самите тях – тогава пасивен елемент за + е фиктивен лист с нулева ширина, а за $\oplus$ – такъв с нулева височина. Ако при това листовете имат графично съдържание, те представляват мащабируеми чрез + и $\oplus$ една

спрямо друга безразмерни фигури. Такъв вид обекти има в т.нар. функционална (композиционна) геометрия и в езиците за програмиране GeomLab [2] и U [1], където хоризонталното и вертикално съединения служат за построяване във функционален стил на разнообразни сложни картини.

Любопитно е, че + и ⊕ дават също сумарните стойности съответно на последователно и успоредно свързани електрически съпротивления. Съдружителността на реципрочното събиране, доказуема, но неочевидна от определението на ⊕, е направо нагледна при съединяването на листа и особено при успоредното свързване на съпротивления.

Допълнителни бележки

Функционалният модел на програмиране предоставя изобилен материал за изследване както на даннови, така и на програмни структури. По-горе е засегната само малка част от първото. За да дадем макар и бегла и силно ограничена представа за второто да разгледаме функция w, „удвоител“, която прилага дадена функция към две други, еднакви помежду си стойности. На езика U можем да определим w така: `w == @{f::@{x::x f x}}`.

Макар и проста, w е достатъчно обща, че да може редица други полезни функции да се зададат чрез нейни подходящи прилагания: `w.+` удвоява число или редица, `w.*` повдига число на квадрат, `w.⊕` намира *половината* на число, а `w.>>` прилага двукратно функция (>> е композиция на функции в U). Като именуваме втората от тези функции `square`, а четвъртата `twice`, получаваме още например функциите `twice.square`, която двукратно повдига на квадрат, и `fourtimes==twice.twice`, която прилага функция четирикратно. Оттук `fourtimes.square.3` дава $(((3^2)^2)^2)^2 = 43046721$ и т.н.

Благодарности

Работата е частично финансирана по договор ДТК 02-69/2009 с ФНИ.

Литература

1. Б. Банчев. Езикът за програмиране U. http://www.math.bas.bg/bantchev/place/u/u.pdf.
2. M. Spivey. GeomLab—exploring computer science. http://www.cs.ox.ac.uk/geomlab.

PROGRAMMING AS METACALCULATION

Boyko Bantchev

Elements of an approach to teaching programming in school are described, based on the functional paradigm and harmonically blending programming, calculation, and algebra. We believe that it ensures a smooth introduction to programming, free of contrived concepts and constructs, and also fosters the teaching of a wider and deeper mathematics in school.