U – ФУНКЦИОНАЛЕН ЕЗИК ЗА ИЗУЧАВАНЕ НА ИНФОРМАТИКА

Бойко Банчев

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

Резюме. *В статията се разглежда възможността за изучаване на информатика чрез функционално програмиране като обобщение на съставянето и пресмятането на аритметични изрази. Дават се сведения за функционалния език U и идеи за използването му в това отношение.*

Ключови думи: *информатика, функционално програмиране*

Програмирането в обучението по информатика

Изучаването на информатика е неделимо от изучаването и практикуването на програмиране. Когато става дума за училище, това поставя двойно предизвикателство. От една страна, овладяването на умения за програмиране следва да се съчетава с усвояването на знания от други предметни области – нали тогава се изучава не само информатика; а и заниманията с програмиране, ако се възприемат като самоцелни, едва ли са интересни и полезни.

От друга страна, конкретността на заниманията изисква използване на език за програмиране – най-често универсален, но възможно и специализиран. Езикът трябва да спомага възможно най-пълноценно за навлизане в света на информатичните понятия, като сам той не поставя допълнителни бариери пред изучаващи и преподаватели, а напротив, е колкото може по-ненатрапчив.

Примери за гротескна неадекватност в това отношение са езиците C++ и Java. Широкото им използване за други цели и отсъствието на добре обмислен избор са довели до неоправдано масовата им употреба и за начално обучение по информатика.

В предишна статия [1] обърнах внимание на някои от недостатъците на споменатите и други езици в тази им употреба. Също там, чрез примери на езика Хаскел, посочих много по-добрата пригодност за обучение на функционалното програмиране. Тук продължавам темата, като давам сведения за новия функционален език за програмиране U [2], замислен като средство именно за учебно и експериментално програмиране.

Преди да се спра на езика U, смятам за нужно да изтъкна естествената връзка на функционалния възглед за програмирането с усвоени от обучението по математика понятия и да уточня посоките на разширяване на тези понятия за целите на програмирането. Според мен училищно или друго начално обуче-

ние по информатика може успешно да се построи, следвайки именно тази линия на приемственост и развитие. Тя осигурява, от една страна, плавно, с много малко на брой нови понятия, навлизане в програмирането, а от друга – поддържане на понятийна и предметна близост между заниманията с информатика и математика.

Функционалният възглед за програмирането

Според функционалния възглед програмата е функция, а изпълнението ѝ – пресмятане на тази функция. Обикновено функцията-програма е образувана от обръщения към други функции, а те на свой ред – от обръщения към други и т.н., докато се стигне до някои основни, дадени непосредствено в езика. Пресмятането на основната функция се свежда до това на тези „по-малки“ функции. В крайна сметка всяко извършвано от програмата действие, просто или сложно, е пресмятане на функция.

Привична, използвана далеч не само в езиците за програмиране форма на задаване на пресмятания са аритметичните, логически и други подобни изрази. Основно свойство на такива изрази е, че те като правило са йерархично построени. Освен това в тях могат да участват именовани величини. Дадено име може да замества подизраз, който за яснота като цяло и краткост на основния израз се пресмята отделно. Използването на имена по този начин прави възможно съвкупност от изрази (формули) да се разглежда единно, като обобщен израз. Други имена отговарят на свободни променливи: те обозначават произволно избрани стойности – параметри на израз или „глобални“ за няколко израза променливи. Понякога имената от първия вид означават не просто подизрази, а функции със свои параметри – имена от втория вид.

Всичко това срещаме разбира се и в програмирането; по-точно, то е основата на функционалния стил в него. Функционалните езици за програмиране са продължение и обобщение именно на този модел на пресмятане. Всеки, който е съставял аритметичен израз за решаване на задача, фактически е съставял функционална програма. Уменията за програмиране, които могат да се добият в рамките на функционалния подход, са естествено продължение на уменията за решаване на задачи чрез съставяне и пресмятане на изрази.

Така функционалният модел дава възможност за плавно навлизане в програмирането от вече позната област. В това отношение той има неоспоримо предимство пред другите подходи, които изискват овладяване на напълно нови, непознати от други учебни дисциплини или дейности системи от понятия.

Какво обаче отличава същинските програми от аритметичните изрази?

Съществена черта на програмите е по-голямото разнообразие от видове стойности и съответно – от действия с тях. Програмирането борави не само с числа, но и например с булеви стойности, литери, текстови низове и абстрактни (представящи сами себе си като стойности) символи. За всеки от тези

типове стойности съществуват и подходящи действия, съставящи „аритметиката“ на типа.

Образуването на изрази от множество разнотипни стойности прави и самите изрази различни по тип, което на свой ред води до необходимост от по-разнообразни и добре уточнени правила за комбинирането им.

Много важна е възможността за агрегиране – образуване на съставни стойности – включително от други такива стойности. Функционалното програмиране благоприятно изпъква сред другите подходи с по-голяма непосредственост както на самото агрегиране, така и на третирането впоследствие на образуваните стойности. Първото означава пряко, без въвличане на помощни понятия и конструкции построяване на съставни стойности, а второто – работа с тези стойности като цяло вместо само с отделните им части.

Например в езика U можем да образуваме стойност от тип редица непосредствено чрез изброяване на елементите ѝ или чрез подходяща операция. Редиците винаги имат определено съдържание, а действията с тях ги третират като единни стойности. В език като C++, обратно, действията с масиви или вектори са най-често поелементни, т.е. смислово откъснати от съответната структура, а създаването ѝ е непряко – образуваме „празен“ масив с определен размер, който после „запълваме“ и т.н.

Характерно за функционалното програмиране е разглеждането на функциите не само като носител на действие, но и като стойности. Така функция може да се окаже елемент на структура от данни, да бъде аргумент на друга функция, а и сама да бъде образувана чрез действието на друга функция. Това е и едно от най-ценните открития на информатиката – благодарение на него програмата може да настройва и разширява извършваните от нея действия в хода на собственото си изпълнение, а понятията, свързани с данни, носят не само пасивен, а и действен смисъл. Особено ценно е обособяването на функции – шаблони за различни общи схеми на поведение свързано с пораждане, вариантност, избор, обхождане, извличане и натрупване. Използването на такива функции помага за постигане на добро смислово структуриране на програмата и в частност намалява необходимостта от изразяване чрез рекурсия, замествайки я с по-непосредствен, чисто апликативен стил.

Въпреки че такова боравене с функциите няма аналог, а напротив, силно се различава от това, което (понастоящем) предлага училищната математика, то в никакъв случай не е непреодолима концептуална трудност за овладяване на основите на програмирането. Едно от последните свидетелства за казаното е успешното преподаване на и чрез функционни абстракции в рамките на системата BYOB [3] – разширение на визуалния език Scratch със средства за действително програмиране, особено във функционален стил.

Езикът U

U е нов функционален език, предназначен най-вече за обучение и опити по програмиране. Работата по него започна с експерименти по използване и видоизменяне на езика ГеомЛаб [4], създаден за подобни цели. На даден етап натрупаните идеи за усъвършенстване се оказаха толкова много и така значително отклоняващи се от ГеомЛаб, че фактически преминах към създаването на нов, напълно различен език. Полученият резултат е съзнателно и несъзнато повлиян от множество езици, а съдържа и ред напълно оригинални конструктивни решения.

В езика U има 7 вида стойности: числа, булеви, низове, символи, редици, фигури и функции.

Редиците имат особена роля. Те са единственото, но универсално средство за агрегиране на произволни стойности, включително при образуване на аргумент на функция – всяка функция има точно един аргумент и той е редица. Редица може непосредствено да се използва като динамичен масив, асоциативен масив, стек, дек, опашка, n-ка, смес, дърво и др. Съвкупности от редици могат да поделят елементи и да образуват произволно сложни структури. Разностранната употреба на редиците се осигурява чрез подходящи операции.

В определенията на функции строежът на редица може да бъде анализиран чрез структурно съпоставяне с шаблон. Съпоставителното разграждане може да става във всеки от двата края на редицата. Например функцията

```
@{(a\(b\(_/z))) a<z<b ::
 ^              b<z<a :: "t
 ^              "t    :: "f}
```

има булев резултат, показващ дали последният елемент на редица с поне три елемента е по големина между първия и втория.

Налице е голямо множество от вградени в езика операции и библиотечни функции за работа с редици: за образуване на редици, за извличане на части или отделни елементи на редица, за подреждане, пренареждане, групиране или друг вид преобразуване, за извличане на обобщена информация от членовете и за обхождане с прилагане на какво да е действие.

Действията с редици са приложими и върху низове, макар че низовете имат и някои специфични свойства. Символите служат за означаване на предопределени от езика константи или са абстрактни стойности, представящи сами себе си. Върху последните могат да се прилагат и действия с низове, а значи и действия над редици.

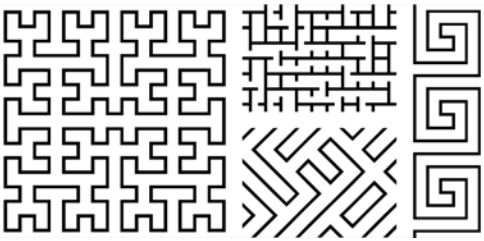
Важна особеност на U е боравенето с фигури като вграден в езика тип стойности. Фигурите биват прости и съставни. Простите са предопределени от езика или образувани в програмата, а съставните се образуват от простите и от

други съставни чрез хоризонтално и вертикално композиране, а също чрез въртене, осева симетрия и припокриване.

Идеята за подобна „композиционна геометрия“ и то в контекста на функционалното програмиране е описана в [5], а за пръв път е програмно реализирана в езика ГеомЛаб. Авторът на последния смята, че композирането на фигури е богат източник на аналогии със структурите в информатиката и затова програмирането с такива обекти е добър начин за навлизане в нея.

Трябва да подчертаем, че в U, в сравнение с [4,5], фигурите са функционално по-богати. Основна разлика е, че докато в [4,5] съставните фигури, веднъж образувани, се оказват атомарни стойности, в U те остават наистина съставни: строежът на фигура може да бъде изследван чрез структурно съпоставяне, подобно на това за редица, така че да се извличат частите и връзките между тях.

По-долу са показани примери на съставни фигури, образувани чрез програми на U. Лявата и дясна фигури са построени от отсечки и прави ъгли, горната средна от Т-образни фигури, а долната – от двойки успоредни отсечки със скосени краища. Наред с подреждането по двете координатни оси е приложено огледално обръщане по осите и въртене на прав и изправен ъгли.



Водещ стремеж при създаването на U беше получаването на език, от една страна, достатъчно изразителен и практически удобен за решаване на тематично разнообразни задачи, а от друга – възможно най-икономичен на понятия, форми на запис и правила, за да бъде лесно усвояван.

Понеже е функционален език, основна структура в U са изразите. За да бъде изразителен и при това лаконичен, U разполага със сравнително много и разнообразни операции и стандартни функции. Свързаното с това потенциално усложняване на езика, например чрез правила за относителни предимства между операциите и за ляво/дясно съдружаване, е избягнато. Образуването и пресмятането на изрази е подчинено на няколко особености, с които U се различава от останалите езици и чрез които се постига възможно най-голяма еднообразност, а чрез нея – простота.

Всеки израз, който не е непосредствено зададена стойност, има вида *x f y*, където *x*, *f* и *y* са подизрази, а стойността на *f* трябва да бъде функция. Тази функция се прилага към стойностите на *x* и *y*. Самата тя може да бъде зада-

дена непосредствено чрез знак за операция, чрез име, или да се получава посредством пресмятане. Между операциите и функциите няма предимства (като умножението да бъде преди събирането и пр.), но където е нужно дадено действие да предхожда друго могат да се използват (кръгли) скоби. Без скоби пресмятането се извършва строго последователно отляво надясно.

В математическото писане и другаде е прието функцията да се записва преди аргументите си, например *f*(*a*) и *g*(*u,v*). В U на такъв запис отговарят **f.[a]** и **g.[u;v]**, където знакът **.** е операцията „прилагане на функция към аргумент“. Аргументът на функция е само един и е редица (**[**…**]**), но редицата може да съдържа различен брой членове, а те да се тълкуват като отделни аргументи.

Да обърнем внимание, че обръщението към функция по този начин спазва общия инфиксен вид на изразите, както беше посочен по-горе. Израз от вида **f.z**, макар да се тълкува като прилагане на **f** към **z**, всъщност е прилагане на операцията . към аргументи **f** и **z**. При това **f** може да бъде име, знак на операция или произволен израз, стига той да произвежда функционна стойност.

Няма разлика между действия, зададени чрез знаци на операции, имена на функции и изрази с функционна стойност. Например **x+y** може да се запише и като **+.[x;y]** (прилагане на **+** към редица от две числа). Обратно, всеки израз от вида **f.[x;y;**…**]** (предполагаме, че редицата-аргумент има поне два члена) може да бъде записан като **x f [y;**…**]**.

Известно опростяване в конструкцията на езика се постига и чрез съвместяване на много от знаковете на операции за повече от едно действия. Съвместяват се като правило сродни или подобни действия, а означенията са подбрани така, че да бъдат мнемонични. Различаването между действията става по вида на аргумента. Например **|** е абсолютна стойност на число, дизюнкция на булеви стойности, вертикалноосев огледален образ на фигура или хоризонтална композиция на две фигури, а **.** (точка)– прилагане на функция към аргумент или извличане на елемент на редица по пореден номер, по асоциирана стойност или по условие, което елементът изпълнява (последните три могат да се разглеждат общо като разновидности на „извличане по индекс“ в редица, а то пък – като частен случай на прилагане на функция: „индексът“ избира елемент на редица точно както аргументът определя стойността на функция).

Функционните стойности в U се образуват чрез изрази-функции, чрез преобразуване на функции и чрез частично прилагане на функции. Израз-функция е особен вид израз, чрез който се задава определение на функция. Определението се състои от една или повече клаузи, във всяка от които се посочва поведението на функцията според съответствието на аргумента ѝ на даден шаблон и удовлетворяването на едно или друго условие.

Например изразът по-долу задава функция, вмъкваща число (**x**) в подредена по големина редица. Шаблонът в първата клауза отговаря на празна реди-

ца, а този във втората и по подразбиране в третата – на непразна (операцията **** удължава редица чрез добавяне на елемент, а в шаблон „разгражда“ редицата на челен елемент и остатък). Условията **x=<y** и **"t** (стойността „истина“) уточняват, че втората клауза се прилага при **x≤y**, а третата – в обратния случай. Знакът **@** в израза за стойността на функцията в третата клауза е служебно име за обозначаване на определяната функция, така че там то отговаря на рекурсивно повикване.

```
@{[x;[]]        :: x
 ^[x;y\ys] x=<y :: x\(y\ys)
 ^         "t   :: y\(x@ys)}
```

Следната функция образува редица от резултатите на последователното прилагане на функция **f** към стойност **x**. По-точно, ако прилагането на **f** дава двойка стойности, първата от тях става член на редицата-резултат, а втората се използва като „ново **x**“, към което се прилага **f**. Когато прилагането на **f** даде празна редица, окончателният резултат е получен:

```
@{[f;x] :: g.(f.x)
 <-- g == @{[a;y] :: a\(f@@y)
           ^[]    :: []}}
```

За анализ на резултата от **f** определението използва локално (чрез **<--**) зададена помощна функция **g**. В нея **@@** обозначава външната спрямо **g** функция, т.е. основното определение, към което по този начин се извършва рекурсивно повикване. (Изобщо, служебните имена **@**, **@@** и пр. дават възможност за пряко- или косвенорекурсивни повиквания дори на неименувани функции.)

Вместо израза **g.(f.x)** по-горе можем да запишем **g<<f.x** или **f>>g.x**, правейки композицията на **f** и **g** явна, съответно отдясно наляво (**g<<f**) или отляво надясно (**f>>g**). Чрез частично прилагане, композиция и ред други определени в езика действия от дадени функции могат удобно да се образуват разнообразни нови. В това отношение U следва традициите на съвременните функционални езици, като освен това се стреми чрез използване предимно на небуквени означения за действията да приближи съставянето и изпълнението на програми възможно най-много до съставяне и пресмятане на аритметични изрази.

Тъй като компактният символен запис може да бъде затруднителен за начинаещи, всяка от операциите може да се именува и имената да се използват равнозначно със знаците на операции. Например, ако в програмата определим **преди == >>**, **след == <<** и **към == .**, вместо **g<<f.x** и **f>>g.x** можем да пишем съответно **g след f към x** и **f преди g към x**: изразът добива по-описателен вид, въпреки че строежът му се запазва. При това дори можем да използваме различни имена за различните операции, означавани с един знак.

В духа на програмирането чрез изрази, в U много действия се задават непосредствено във вид на израз, използвайки само наличните функции, без определяне на нови. Например 0,1/*n*,…,(*n*-1)/*n* получаваме чрез израза **_:n!(0..n)**:

операцията **..** образува редицата 0,…,*n*-1 и към нея чрез **!** се прилага **_:n**, „функцията, която дели на *n*“ – частично прилагане на операцията **:** (делене).

Приложения

Изучаването на информатика чрез U е благоприятно за усвояване и на понятия, структури и похвати, характерни именно за програмирането, и на ред такива от математиката. Нещо повече, функционалността на езика и разнообразието от операции над различни видове стойности са добра предпоставка за единно разбиране на двете области, което е само по себе си твърде ценно.

Един възможен слой на разглеждане е този на числовата и булевата аритметики, както и аритметиката на редиците и по-специално на низовете. Също много подходяща област за обучително програмиране е тази на текста – смислово и структурно продължение и обобщение на низовете. Свойствата на функциите – видове, частично прилагане, начини на композиране и формални преобразования на изрази, следователно на програми – са също ценен, малко по-абстрактен предмет на разглеждане в обучението. Всяко от изброените предоставя изобилие от възможни теми на различни равнища на сложност.

Нова, много благодатна, при това и нагледна област е споменатата композиционна геометрия. Например, ако разглеждаме ротации на кратни на прав ъгли и симетрии относно координатните оси и ъглополовящите им (всички тези трансформации са операции в U), можем да анализираме начините на получаване чрез тях на осемте възможни разполагания на правоъгълник успоредно на координатните оси. С това достигаме до пример на алгебрична група (на трансформациите) и съответно можем да разглеждаме представимост на едни нейни елементи чрез други, базис, подгрупа и други важни понятия в конкретна и нагледна форма.

Богата на интересни и донякъде неочаквани свойства, композиционната геометрия заслужава подробно разглеждане в отделна публикация.

Литература

[1] Б. Банчев. *Преподаването на информатика: специфика и алтернативи.* Нац. конф. „Образованието в информационното общество“, Пловдив, 27-28 май 2010, стр. 266-274.

[2] B. Bantchev. *A language for compositional programming: a rationale and design.* Proc. 40th Spring Conf. of the UBM, April 2011, 236-243.

[3] *BYOB—Build Your Own Blocks.* http://byob.berkeley.edu.

[4] *GeomLab–exploring computer science*. http://web2.comlab.ox.ac.uk/geomlab

[5] P. Henderson. *Functional geometry.* Proc. 1982 Symp. on Lisp and functional programming, 179-187, 1982.