Увод в езика REXX

Бойко Банчев

Общи сведения за езика

Езикът за програмиране Rexx се появява през 1979 г. Създаден е във фирмата IBM като „език за командни процедури“ на знаменитата операционна система VM/SP, но заедно с това и като самостоятелно използваем, пълноценен език за програмиране. Програми на друг език могат да повикват интерпретатора на Rexx за изпълнение на фрагменти на Rexx; по този начин се придава динамично поведение на по начало „статични“, т. е. написани на предимно статичен, компилиран език програми. Тези черти причисляват Rexx към класа езици, които днес е прието да се наричат „сценарни“ (scripting).

От момента на създаването му насетне Rexx е част от всички операционни системи на фирмата IBM, а днес реализации на езика съществуват и изобщо за всички значителни о. с.

Като език за програмиране Rexx е чувствително по-богат от типичните командни езици като sh (или bash, csh, ksh и т. н.). От друга страна, сравнен със силно развитите „сценарни“, или по-вярно – *динамични* езици като Perl, Ruby, Python и др. под. Rexx е много по-прост за овладяване и лесен за ползване, макар също донякъде и по-скромен по възможности. Това прави Rexx език почти без съперници в неговата област – „лесно и бързо“ създаване на неголеми програми или прототипи на програми, най вече такива с преобладаване на текстови манипулации и без големи изисквания за бързодействие.

Rexx се използва успешно и за управление на малки бази от данни, за написване на интерпретатори за несложни езици, за съставяне на командни процедури и като макрокоманден език за текстови редактори, асемблери и други програми. Съществуват и някои големи – с обем стотици хиляди редове – програми, написани на Rexx. Да отбележим, че разпространението и развитието на езика се дължат отчасти на съвместното му използване със семейството широко използвани текстови редактори xedit, kedit и the.

Сценарните и изобщо динамичните езици, някога рядкост, днес вече заемат обширна територия в света на програмирането и тенденцията е тяхната роля да нараства. Тези езици все повече добиват универсална приложимост и наистина се използват за създаване на всякакви по вид, сложност и големина програми. В тази област Rexx е не само далновиден първенец, далеч във времето изпреварил подобните нему, но и език, и днес използваем и удобен за решаване на множество задачи.

Повечето използвани днес реализации на Rexx са много близки до първоначалния вариант на езика и общо биват обозначавани като „класически Rexx“. Някои от тях съдържат допълнения към езика. Най-популярен и достъпен за всички о. с. е интерпретаторът Regina.

Известни са и два езика, създадени на основата на Rexx: Object Rexx и NetRexx. И двата, както и самият Rexx, са дело на фирмата IBM. Първият е разширение на Rexx със средства за обектноориентирано програмиране, а вторият е близък до Rexx, но различен от него език, който се транслира на езика на т. нар. виртуална машина на Java и областта му на приложение е сходна с тази на Java.

Object Rexx има свободно достъпна реализация под името Open Object Rexx.

Основните отличителни черти на Rexx са следните:

- Изпълняващата система на Rexx е интерпретатор. Операторите в програмата се прочитат и изпълняват *отделно един от друг*, без междинно преобразуване на програмата.
- Както операторите в програмата, така и стойностите на данновите обекти в нея са *литерни низове*. Според контекста различни низове или части от тях могат да бъдат интерпретирани като служебни думи на езика, числа и др. Низовете, стойности на обекти в програмата, могат да бъдат произволно дълги.

  Редица вградени в езика функции изпълняват различни операции върху низове. Това, заедно с оператора `PARSE` за разбор на низове, прави езика много удобен за анализ и преработване на текст.

  В множество случаи – при прилагането на някои вградени функции, на операторите `PARSE`, `ARG` и др. – като особен вид низове се разглеждат „*думи*“. Дума е несъдържащ интервали низ, обикновено подниз на друг низ.
- Езикът не разполага със средства за изграждане на структури от данни и работа с тях. От друга страна, съществува прост механизъм за работа със *съставни имена на променливи*, чрез който могат да се имитират различни даннови структури.
- Части от програмата могат да бъдат пораждани по време на изпълнението ѝ.
- В програмата на Rexx лесно се вграждат *обръщения към команди на друга изпълнителна среда*, например операционната система.

Структура на програмата

Програмата на Rexx се записва като последователност от оператори. По отношение на основните управляващи конструкции езикът е сходен с повечето процедурни езици. Осигуряват се условно и циклично изпълнение в стил, близък до този на езика PL/I.

Подпрограмите могат да бъдат *вътрешни от отворен тип* или *външни* за програмата, а обръщения към тях могат да се правят както в процедурен, така и във функционен стил. Механизмите за формиране на подпрограми и за предаване на аргументи при обръщения към тях отличават съществено езика от повечето други.

Всеки оператор е от един от следните видове: оператор за присвояване, оператор-команда, етикет, празен оператор и несобствен оператор. Операторите-команди започват всеки с определена служебна дума, обозначаваща съответното специфично за оператора действие. Заедно с операторите за присвояване те задават основните действия в повечето програми.

Интерпретаторът подлага всеки оператор в програмата на лексикален разбор, при което даденият низ се разбива на служебни думи, имена на променливи, непосредствено представени низове, знакове на операции и други характерни лексеми. Ако полученият низ от лексеми задава *собствен* за езика оператор, т. е. от някой от първите четири вида, той се изпълнява. В противен случай операторът се смята за

несобствен. Такъв оператор се интерпретира като израз и полученият в резултат на пресмятането низ се предава за изпълнение на установената по подразбиране външна изпълнителна среда. Последната обикновено може да бъде избирана измежду няколко възможни среди, предварително фиксирани чрез задаване на конфигурация от интегрирани с интерпретатора на Rexx програми.

Обичайно е операторите да се записват всеки на отделен ред. Допуска се два и повече последователни оператора да се записват на един ред, като се отделят един от друг със знака `;`. Един оператор може да се запише на няколко реда: като знак за пренос в края на пренасяния ред се поставя запетая (`,`), която задава конкатениране със следващия ред с един междинен интервал.

Поредовото интерпретиране на програмата дава възможност в нея да се вмества *свободен текст*, който не подлежи на изпълнение и по същество не е част от програмата. Такъв текст може да съдържа коментари, сведения или произволни бележки от автора на програмата. Той може да се тълкува и като данни, които програмата да прочете при изпълнението си. Програмистът осигурява недостигането от интерпретатора на съответните текстови редове като разполага по подходящ начин операторите за управление в програмата.

Най-удобно е свободен текст да се поставя или в края, или в началото на програмата, като в последния случай той се предхожда от оператор за преход към първия изпълним ред, който е същинското начало.

Освен по споменатия начин, коментарен текст може да се задава и като се огради в коментарни скоби `/*` и `*/`. Тогава той може да заема част от ред или няколко последователни реда и се пропуска автоматично от интерпретатора.

Текстът на произволен ред от програмата се дава от вградената функция `sourceline()`, чийто аргумент е номерът на реда. Обръщение към същата функция без аргумент дава броя редове във файла, съдържащ програмата. Като използва тези две форми на обръщение към функцията, програмата може сама да анализира или извежда текста си, а също и да го променя, така че при следващото нейно изпълнение да има възможно друго поведение.

В частност, чрез `sourceline()` програмата може да прочете или измени данни, записани в самата нея като свободен текст. Един пример за това е в програмата да се съхранява актуална информация, свързана с последното ѝ повикване: продължителност на изпълнението, дата и час на започване и др.

Разпространена практика е дадена програма да съдържа няколко реда с *помощна информация* за потребителя, която се извлича с помощта на `sourceline()` и се отпечатва при неправилно повикване на програмата или когато тя е повикана по специален начин именно за да даде помощна информация. Същият текст служи заедно с това и като вътрешна документация на програмата.

Променливи, имена и структури от данни

Променливите в програмата се цитират посредством имена, съставени от букви от латиницата, цифри и някои други знакове. Интерпретаторът автоматично преобразува всички малки букви на имената в големи, така че *действителното име* на всяка променлива съдържа само главни букви. (Същото важи и за служебните думи в езика.) Така например `form`, `Form` и `FORM` цитират една и съща променлива с име `FORM`.

Променливите не се обявяват, не се създават и не им се приписват типове. Смята се, че за всяко допустимо име съществува променлива, именувана от него. На всяка променлива се приписва като начална стойност низът, съдържащ името на променливата.

Предопределената ситуация `NOVALUE`, ако бъде установена реакция на нея, възниква при използване на стойността на неинициализирана променлива, т. е. когато в някой израз се използва променлива, за която не е извършено явно присвояване. Ако за ситуацията `NOVALUE` не се допуска потребителска реакция, тя се игнорира, така че използването на подразбиращите се стойности на променливите се приема за нормално действие. Това е установеното по подразбиране поведение на всяка програма.

Чрез оператора `DROP` една или повече променливи могат да бъдат приведени в неинициализирано състояние, все едно, че в програмата не им е присвоявана стойност. Всяка такава променлива получава отново началната си стойност по подразбиране: низ, съдържащ името ѝ.

Посредством вградената функция `symbol()` в програмата може да се определи дали даден низ отговаря на допустимо име на променлива и ако е така, дали тази променлива е инициализирана. Употребява се най-вече за второто.

Имена, които не съдържат в записа си точка, се наричат *прости*. Допуска се употребата и на *съставни* имена. Съставните имена се образуват от *основа*, която е просто име, и един или повече *индекса*, разделени един от друг и от основата с точка (`.`). Самите индекси са прости имена или цели числа.

Както простите имена, всяко съставно име също съответства на променлива, но съответствието е косвено чрез индексите. Когато за цитиране на променлива се използва съставно име, действителното име на променливата се образува като всеки индекс в съставното име се замести със стойността, която представя: ако индексът е цяло число, то самò представя себе си; ако е име – стойността на променливата с това име.

Началната стойност на променлива със съставно име се образува от името на основата и стойностите на индексите.

Например записът `a.12.ref` цитира променливата `A.12.REF`, ако променливата `REF` не е инициализирана явно. Ако на `REF` е присвоена стойност `101`, то същият запис цитира променливата `A.12.101`.

Допускат се и съставни имена, които завършват с точка или съдържат две или повече последователни точки, така че `x.hoy.`, `v.w..` и `f..1` са валидни цитати на променливи.

Важен частен случай са имената, образувани само от основа и точка. Те могат да се нарекат *родови имена*. Присвояването на стойност на променлива с родово име автоматично присвоява същата стойност и на всички променливи, чиито имена имат дадената основа. Например присвояването

```
t. = 'Cicero'
```

задава стойност не само за променливата с име `T.`, но и за променливите с имена `T.U`, `T..`, `T.7.SJ.4.5` и т.н. Стойността на променливата `T` обаче остава непроменена. Променливата `T` е различна както от `T.`, така и от всички, чиито имена имат тази основа.

Ако като аргумент на оператора `DROP` се цитира родово име, действието се отнася и за всички променливи, на които имената имат за основа това родово име.

Правилата за именуване на променливи дават възможност да се имитират различни структури от данни, като масиви, смеси и асоциативни масиви. Всяко просто или съставно име задава *потенциално безкрайна асоциативна таблица*, чиито елементи имат имена, получени от изходното с добавяне на по един индекс. На свой ред всеки такъв елемент поражда друга таблица и т. н.

В следния пример се пресмята сборът от елементите на едномерен „масив“:

```
summa = 0
  DO  i = 1  TO 20
  summa = summa + x.i
  END
```

Фактически образуването на структури от данни се реализира чрез образуване на *структури от имена*. Макар стойността на всяка отделна променлива да е скаларна, неструктурна, задаването на йерархични или други връзки между имената на променливи се използва на задаване на съответната организация за стойностите им. Гъвкавостта на този механизъм се усилва и от възможността имената, като всички низове, да се образуват в хода на програмата като стойности на изрази, например да бъдат получавани чрез конкатениране.

Недостатък на подхода е, че с изключение на родовото присвояване не съществуват средства за работа със структури от данни като цяло. Няма непосредствен начин дадена структура да бъде извлечена, копирана или унищожена. Всъщност всяка структура от данни е *неявна*, тъй като броят и имената на елементите ѝ и връзките между тях са само неформално известни – като замисъл на програмиста.

Друг недостатък, отнасящ се до свойствата на променливите изобщо, е невъзможността стойността на дадена променлива да се цитира или променя по непряк начин, без употреба на името ѝ. В езика няма указатели, образуване на синоними и изобщо никаква форма на косвено цитиране на стойността на променлива. Наистина, косвено цитиране на самата променлива може да се постигне чрез третиране като стойност на името ѝ: низ, резултат от пресмятането на израз (който в типичния и най-прост случай е просто друга променлива), тълкуван като име на променлива може да се използва за получаване на достъп до променливата. Това обаче е косвеност по отношение на името, а не на обозначаваната от него стойност. В крайна сметка единственото средство за достъп до стойността на променлива е нейното име.

Липсата на възможност за косвено цитиране се отразява и на използването на подпрограми. За това ще стане дума по-нататък, в раздела „Подпрограми“.

Изрази

Изразите се образуват чрез прилагане на операции и обръщения към функции – вградени или определени от програмиста. Стойностите както на аргументите, така и на резултата на всяка операция или функция са низове. Когато низ-аргумент се задава непосредствено, той се загражда в единични или двойни кавички. Такъв низ е самопредставена стойност.

Особен вид низове са числовите. Числовият низ съдържа десетичен запис на цяло или реално число и когато бива аргумент на аритметична операция стойността му се тълкува като числова. Резултатът на аритметична операция е също числов низ. С помощта на това допълнително свойство, което имат числовите низове в повече от останалите, се имитира присъствие в езика на числов даннов тип.

За удобство непосредствено представените числови низове могат да се записват без кавички. Например изразите `'19.016'+'8.27'` и `19.016+8.27` са еквивалентни.

Чрез оператора `NUMERIC` може да се избере произволно голяма точност на числените пресмятания, както и точност за закръгляне на аргументите при извършване на числени сравнения. В частност, това дава възможност за работа с неограничено големи числа.

Операциите в Rexx могат да бъдат разделени по смисъл на няколко групи.

Аритметичните включват, освен обичайните действия, два вида делене – с цял и реален резултат, както и остатък по модул, включително за реални числа, и повдигане в цяла степен.

Операциите за сравняване са традиционните шест операции за установяване на наредбено отношение, като върху числови аргументи добиват смисъл на числови сравнения, а в общия случай – лексикографски. При последните не се отчитат водещите и остатъчните интервали, ако има такива, на аргументите-низове. Има и операции, проверяващи точно съвпадане или несъвпадане на низове.

Булевите операции са дизюнкция, конюнкция, изключващо или и отрицание. За аргументите им се допуска да имат само стойностите `1` и `0`, за които се приема, че съответстват на булевите *истина* и *неистина*. Тези две стойности са и възможните резултати на операциите за сравняване.

Операциите върху низове са *плътно* и *неплътно конкатениране*. И двете са много характерни за повечето програми на Rexx. Първата операция се бележи с `||` и задава точно слепване на два низа. Втората се обозначава с разделяне на аргументите ѝ един от друг с поне един интервал, а резултатът е слепване на двата низа с точно един междинен интервал.

Плътно слепване може да се зададе и непосредствено, като двата операнда се запишат плътно един след друг, стига това да не води до изменяне на смисъла. За непосредствено задаване на плътно слепване е достатъчно например единият от аргументите да е низ в кавички или левият аргумент да е обръщение към функция; то не може да стане за аргументи, които са имена на променливи, тъй като тогава вместо конкатениране се образува име на друга променлива.

Управление на изпълнението

За задаване на условно изпълнение са предвидени операторите `IF` и `SELECT`: `IF` – за условно изпълнение на оператор или за избор на един от два, а `SELECT` – за избор на една от няколко алтернативи.

Когато за дадена алтернатива не трябва да се изпълняват действия, може да се използва операторът `NOP`. Когато пък трябва да се изпълнят повече от един оператора се прибягва до оператора `DO`.

Операторът `DO` групира следващите го оператори в последователност, която се тълкува като един оператор и следователно може да бъде в ролята на изпълняван при избрана алтернатива оператор в `IF` или `SELECT`. Групата `DO` завършва с оператор `END`.

Освен за групиране, операторът `DO` се използва и за организиране на циклично изпълнение на един или повече оператора. Това става, като след думата `DO` се запише *уточнител на повторенията* на групата `DO`, който може да бъде четири вида. Три от тях задават цикли с пред- или следусловие, със зададен брой повторения

или с управляваща променлива, равномерно нарастваща или намаляваща в посочен интервал.

Допуска се съвместно използване на различни видове управление в един цикъл, например на зададен брой повторения и предусловие, както и съвместно задаване на предусловие и следусловие. Пример за използване на фиксиран повторител в цикъл с управляваща променлива е следният:

```
DO  t = 0.001  BY 0.001  FOR 1000
```

Целта е променливата `t` да се изменя с равномерна стъпка между `0` и `1.0`, но ако вместо това запишем

```
DO  t = 0.001  BY 0.001  TO 1.0
```

резултатът може и да не бъде същият поради неточното представяне на числата в компютъра, което може да причини недостигане на граничната стойност.

Уточнителят `FOREVER` задава безкраен цикъл. Изпълнението му може да бъде прекратено само посредством някой от операторите `LEAVE` или `SIGNAL` в тялото на цикъла. `DO FOREVER`, заедно с условно изпълняван оператор за завършване, се прилага за имитиране на цикъл със средусловие.

Операторът `LEAVE` прекратява изпълнението на цикъл, а операторът `ITERATE` – само на текущата итерация в даден цикъл. За посочване на външен, обхващащ цикъл, като аргумент на `LEAVE` или `ITERATE` може да се зададе име на съответна управляваща променлива. Оператор `LEAVE` или `ITERATE` без аргумент, ако се появява в проста група `DO`, се отнася не до нея, а до най-близкия цикъл `DO`, който обхваща този оператор.

Операторът `SIGNAL` може да се използва за предаване на управлението към точка в програмата със зададен етикет, аргумент на `SIGNAL`. Ако след `SIGNAL` се запише служебната дума `VALUE`, аргументът на оператора се тълкува не като непосредствено зададен етикет, а като израз, който се пресмята и получената стойност е име на етикет за извършване на прехода.

След изпълняване на оператор `SIGNAL` специалната променлива `SIGL` получава като стойност номера на реда в програмата, в който е изпълнен операторът.

Операторът `SIGNAL` се използва също за задаване или отменяне на *възможност за програмирано реагиране* при възникване в програмата на някои предопределени *ситуации на грешка*. В тази си форма операторът не извършва преход, а позволява или забранява извършването му при дадена ситуация от текущия момент нататък. Ако за даден тип грешка е установена потребителска реакция, при възникване на съответната ситуация се извършва преход към етикета, съвпадащ с името на ситуацията (такъв етикет трябва да е предвиден в програмата). Например `SIGNAL ON SYNTAX` позволява реагиране на синтактични грешки, а `SIGNAL OFF SYNTAX` го забранява. Възникването на ситуация, за която не е разрешена потребителска реакция, или води до прекратяване на програмата, или се игнорира.

Тъй като ситуациите възникват неявно, за да може след възникването на дадена ситуация, при разрешена реакция за нея програмата да установи мястото и причината на възникването ѝ, специалните променливи `SIGL` и `RC` съдържат съответно номера на реда в програмата, при изпълнението на който е възникнала ситуацията, и код на грешката.

Вградената функция `errortext()` дава текста на стандартно в системата съобщение за грешка по зададен код на грешка. Тази функция, както и функцията

`sourceline()`, може да се използва за потребителска реакция на ситуация, като за аргументи на двете служат съответно `RC` и `SIGL`.

`SIGL` може да се използва също и по следния начин. След изпълняване на реакцията за дадена ситуация може да се осигури връщане на управлението (чрез `SIGNAL VALUE`) към подходяща точка преди или след мястото на възникване на грешката.

Входно-изходен обмен на програмата с потребителя

Операторът `SAY` е основното средство за извеждане на информация. Нейният аргумент е израз, чиято стойност е извежданият низ.

За въвеждане на информация интерпретаторът поддържа буфер от низове, съответни на въвежданите от потребителя в програмата текстови редове.

Посредством оператора `PULL` от входния буфер се извлича низ и се разбива на лексеми, които биват присвоявани на посочени в оператора променливи. Действието на оператора `PULL` е еквивалентно на това на `PARSE UPPER PULL` със същите аргументи.

Операторът `QUEUE` вмъква низ във входния буфер, все едно, че низът се въвежда от потребителя. Операторът `PUSH` също вмъква низ във входния буфер, но така, че все едно низът е въведен първи сред намиращите се в буфера, т. е. именно той би бил прочетен от `PULL` непосредствено след `PUSH`.

Вградената функция `queued()` дава броя низове, намиращи се във входния буфер. Нейният резултат може например да се използва за изпразване на буфера, като се изпълни операторът `PULL` необходимия брой пъти.

Чрез `PULL`, `QUEUE`, `PUSH` и `queued()` може да се редактира съдържанието на входния буфер не само за вътрешни за програмата на Rexx цели, а и за да се имитира *подаване на входна информация* за външни подпрограми и програми, изпълнявани като несобствени команди в програмата на Rexx. Такъв похват е удобен, тъй като чрез него се избягва създаването на отделен „*файл с отговори*“ при недиалогово задействане на диалогова програма. На всяка подпрограма или несобствена команда отговаря определен отрязък от буфера. За да не възникват конфликти между различните програми при съвместно използване на буфера, всяка от тях трябва да оставя непроменени отрязъците на активираните преди нея и още незавършили изпълнението си програми.

Различните реализации на Rexx могат да предоставят вградени функции и процедури като допълнителни средства за въвеждане и извеждане на информация, например на отделни литери, в и от файлове и др.

Разбор на низове чрез извличане по шаблон

Езикът Rexx дава възможност да се прави несложен анализ на низ посредством съпоставянето му с шаблон. Низът се разбива на лексеми и те могат да бъдат присвоявани на променливи.

Лексикалният разбор се извършва посредством оператора `PARSE`, чийто общ вид е следният:

`PARSE [ UPPER ]` *източник* *шаблон*

Незадължителната дума `UPPER` предизвиква преобразуване в анализирания низ

на всички малки букви в големи преди да се изпълнят останалите действия, предписвани от оператора.

Два от възможните източници са `PULL` и `ARG`, като в първия случай анализираният низ постъпва от входния буфер, а във втория се предполага, че действието се извършва в подпрограма и се анализират аргументите на обръщението към нея.

Друг източник се задава чрез думата `VAR`, следвана от име на променлива, чиято стойност е подлаганият на разбор низ. За по-голяма общност може да се използва източника `VALUE` *израз* `WITH`, който позволява низът да се зададе като стойност на произволен израз.

Източниците `VERSION` и `SOURCE` дават низове с предопределен формат, съдържащи информация за използваната версия на Rexx, името на операционната система, начина на повикване на текущо изпълняваната подпрограма, име на файла, от който е заредена за изпълнение, име на външната изпълнителна среда и др. По същество `PARSE VERSION` и `PARSE SOURCE` изпълняват ролята на вградени системни функции.

Независимо от източника разборът се извършва според шаблона по едни и същи правила. Единствено изключение е това, че при `ARG` се анализират *няколко низа* – толкова, колкото са аргументите, за което се задават същият брой шаблони, разделени помежду си със запетаи. Разборът на всеки от аргументите се извършва по съответния му шаблон, последователно от първия към последния.

Шаблонът за съпоставяне е редица от имена на променливи и разделители, произволно наредени. На всяка цитирана в шаблона променлива се присвоява част от анализирания низ, като разбиването на части се управлява от разделителите.

Съпоставянето на шаблона с източника става на стъпки по следния начин. От шаблона се извличат един по един разделителите. За всеки разделител се търси съответствие в низа-източник, който бива обхождан от първата към последната му литера. За първия разделител търсенето започва от началото на низа, а за всеки следващ – от текущата позиция върху него, установена от предишното търсене. На всяка стъпка съпоставянето на разделителя с низа-източник придвижва текущата позиция. Ако непосредствено преди разделителя в шаблона стои променлива, на нея се присвоява определен според вида на разделителя подниз на низа-източник.

Разделителите биват разделители на думи, позиционни и литерни.

Разделителят на думи се бележи с интервал. Интервалът е разделител само ако стои между две променливи в шаблона. Ако такива интервали са няколко последователни, те задават единствен разделител. На съответната променлива се присвоява първата дума в низа-източник, разположена след текущата позиция, ако тя сочи интервал, а в противен случай – думата от текущата позиция до първия интервал. Новата текуща позиция сочи след непосредствено следващия присвояваната дума интервал, или края на низа, ако той завършва с тази дума.

Позиционните разделители са цели числа. Такъв разделител задава преместване на текущата позиция на посоченото място в низа-източник. Ако се предхожда от променлива, на нея се присвоява поднизът от предишната текуща позиция до новата, без тя да се включва. Допустимо е да се задава позиция, предхождаща текущата, което дава възможност за *повторно съпоставяне* на части от низа. Когато посочената позиция предхожда текущата или съвпада с нея, възможно присвояваният подниз е остатъкът от низа от текущата му позиция до края. Като позиционни разделители числата без знак или предхождани от знак `=` задават абсолютна позиция, а тези със знак `+` или `-` – относително преместване спрямо текущата позиция.

Освен чрез константа, позиционен разделител може да се зададе и като стойност на заградена в скоби променлива. За целта непосредствено пред лявата скоба се поставя някой от трите знака `=`, `+` или `-`, като `=` означава задаване на абсолютна позиция, а другите два знака – относителна. Стойността на променливата трябва да бъде цяла и положителна.

Литерният разделител е низ, представен или непосредствено, или като стойност на променлива, цитирана в шаблона чрез заградено в скоби име. Такъв разделител се съпоставя със съвпадащ с него подниз на низа-източник. Ако се предхожда от променлива, на нея се присвоява поднизът от текущата позиция до позицията, непосредствено предхождаща съвпадащия с разделителя подниз. Новата текуща позиция е тази непосредствено след съвпадащия низ. Ако не се намери съвпадение, текущата позиция се премества след края на низа-източник и на предхождащата променлива се присвоява празният низ.

Може да се смята, че всеки шаблон завършва с неявно приписан позиционен разделител, чиято стойност задава позиция непосредствено след края на низа-източник. Вследствие на това, ако последният действителен елемент в шаблона е променлива, тя получава като стойност *целия остатък от низа* от текущата позиция до края му.

Ако в процеса на съпоставянето низът-източник се изчерпи преди шаблона, оставащите в последния променливи добиват стойност, равна на празния низ. Ако шаблонът се изчерпи преди източника, част от последния остава непрочетен.

Знакът точка (`.`) в шаблон има ролята на *фиктивна променлива*: съответният на такава променлива подниз на източника се извлича без да се присвоява.

В следните примери се анализира променливата `S`, чиято стойност е низът `'  Ледът се пука тчк  парада ще командвам аз '`.

```
PARSE VAR  s  a b c
```

присвоява на `A` низа `'Ледът'`, на `B` – `'се'`, а на `C` – `'пука тчк  парада ще командвам аз '`.

```
PARSE VAR  s  'т' a . b 'к ' c
```

задава стойностите `A='се'` , `B='тч'` и `C=' парада ще командвам аз '` . При изпълнение на оператора

```
PARSE VAR  s  a 18 b +2 +18 c
```

се получава `A='  Ледът се пука т'`, `B='чк'` и `C='вам аз '`, а операторът

```
PARSE VAR  s  . . . a b 26 c . d
```

дава `A='тчк'` , `B=' пара'` , `C='да'` и `D='командвам аз '`.

Косвеното задаване на позиционни разделители дава възможност в анализирания низ да се поставят табулиращи стойности, управляващи анализа. Ако `X` има стойност

```
'7: ^^^лимонада\  24:  #оранжада\  42:  ==сайдер\   бутилки ... '
```

при изпълнение на оператора

```
PARSE VAR  x  i ':' =(i) a '\' i ':' =(i) b '\' i ':' =(i) c '\'  x
```

се получава `A='лимонада'`, `B='оранжада'`, `C='сайдер'` и `X='  бутилки ... '`. Тъй като присвояванията се извършват отляво надясно, променливата `I` получава

последователно стойностите 7, 24 и 42 и всеки път се използва като индекс за началото на следващата дума, прочитана съответно в A, B и C. Накрая променливата X, съдържаща анализирания до момента низ, също променя стойността си.

Последното е често използван похват: от даден низ се извлича част от записаното в началото му, след което непрочетеният остатък заменя първоначалното съдържание на низа и може да бъде обработен по-късно.

Вградени функции

Вградените функции на Rexx са за работа с низове, за числови операции и няколко други. Най-много и съществени са функциите за работа с низове. Те са няколко десетки, подходящо подбрани, така че в хармония с останалите средства на езика да осигуряват лесно съставяне на програми за различни видове обработване на текст. Могат да се обособят следните подгрупи:

- Функции за търсене, извличане, вмъкване, замяна и изтриване на подниз в даден низ.
- Функции за броене, търсене, извличане, вмъкване и изтриване на думи в даден низ.
- Функции за форматни преобразования на низ като редица от думи: сгъстяване, запълване, центриране, подравняване наляво, надясно и двустранно и др.
- Функции за търсене и замяна в низ, разглеждан като литерно множество.
- Функции за пораждане на низ по зададен друг низ или друг вид входна информация, например създаване на многократно копие на низ.
- Функции за форматни преобразования на числов низ.
- Функции за преобразуване на числов низ от една бройна система в друга.
- Функции за преобразуване на низ от литерен вид в числови представяния и обратно на основата на съответствието между литери и техните номера в използваното кодово множество.

Функцията `datatype()` разпознава дали даден низ представлява запис на число, т. е. е числов и ако е така, дали представя цяло или реално число и др.

Подпрограми

Подпрограмите на Rexx могат да бъдат *вградени*, *вътрешни* и *външни*. Обръщенията към всеки от трите вида стават по еднакъв с останалите начин. Допускат се рекурсивни обръщения.

Вградените подпрограми имат предопределени имена, брой и вид параметри и действие и се предоставят от интерпретатора на системата на разположение на програмиста. Другите два вида подпрограми се задават чрез програмен текст на Rexx и се изпълняват чрез интерпретиране.

Вътрешните подпрограми са от *отворен* тип: нямат синтактично обособено тяло. Началото на всяка подпрограма се задава посредством някой от етикетите в програмата, приет за нейно име. Изпълнението ѝ започва от посоченото място, а завършва с някой от операторите `RETURN` или `EXIT`.

Съответната група от оператори може да бъде приписвана на подпрограмата само условно. Същите оператори могат да бъдат изпълнени не само чрез обръщение към подпрограмата, а също и при достигане на началото ѝ по нормалния ред на следване на операторите или при извършване на преход към началния етикет. Две или повече подпрограми могат да имат общи оператори.

По начало променливите, с които работи всяка вътрешна подпрограма при дадено обръщение към нея, са тези на (под)програмата, от която е извършено обръщението: подпрограмите имат *динамичен контекст* за променливите си.

Локална контекстна среда за вътрешна подпрограма може да се зададе посредством оператора `PROCEDURE`, записан в началото ѝ: всички променливи, които използва подпрограмата, се смятат за различни от тези на главната програма, дори когато имат еднакви имена с тях.

Екраниране на имена на променливи чрез `PROCEDURE` може да бъде отслабено, като се добави след `PROCEDURE` думата `EXPOSE`, следвана от имената на онези променливи, които не трябва да бъдат локални в подпрограмата, а да принадлежат на главната програма. Получава се съчетаване на *динамично образуван локален контекст* със *статично зададен достъп* от подпрограмата до тази, която я повиква.

Чрез цитиране на родово име се задава глобалност на всички променливи с имена, образувани от същата основа.

При извършване на обръщение към подпрограма всеки от аргументите може да бъде произволен израз; неговата стойност се пресмята и се предава като низ в подпрограмата. Там тя може да бъде извлечена и анализирана посредством оператора `ARG`, чието действие е като на `PARSE UPPER ARG` със съответните аргументи.

Посредством функцията `arg()` аргументите на обръщението могат да бъдат извличани в подпрограмата и по поредните им номера: номерът се задава като аргумент на `arg()`. Това е удобно, когато не се налага да се дават имена на съответните формални параметри.

Обръщение към `arg()` без аргументи дава броя на предадените аргументи за съответното обръщение към подпрограмата, в която се изпълнява `arg()`.

Характерно за подпрограмите на Rexx е, че извличането на стойностите на аргументите на обръщението *не става по подразбиране* и то само в началото на изпълнението им, а се извършва *явно*, *на произволно място* и е възможно да става *неколкократно* и *по различни начини*.

Явното извличане на стойности от аргументите на обръщението позволява последните да варират по брой. От друга страна, за формални параметри на коя да е подпрограма не може да се говори в обичайния смисъл, тъй като на аргументите на обръщението не съответстват еднозначно имена на променливи-параметри. Извличането на стойности от аргументите и присвояването им на променливи става в резултат на *лексикален разбор на всеки от аргументите*. Той може да бъде различно зададен и да инициализира различни множества от променливи по различен начин.

Операторът `RETURN` завършва изпълнението на подпрограма и може да посочва израз, чиято стойност се предава от нея в главната програма.

Операторът `EXIT` завършва изпълнението на цялата програма и също може да предава стойност, която в случая е достъпна в изпълнителната среда, от която е станало обръщението към интерпретатора на Rexx. Когато в главната програма на Rexx се използва `RETURN`, той действа като `EXIT`.

Обръщението към подпрограма в процедурен стил става с оператора `CALL`, в който се задават името на подпрограмата и аргументите ѝ, разделени помежду си със запетаи.

Предаването на стойност чрез `RETURN` дава възможност обръщението към подпрограмата да става като към функция, чиято стойност е предадената. Обръщението към всяка функция, независимо от кой от трите вида е, може да бъде както в процедурен, така и във функционен стил, както е по-удобно. И в двата случая специалната променлива `RESULT` съхранява стойността, предадена чрез `RETURN`, след обръщението към подпрограмата.

За повечето от вградените подпрограми на Rexx е обичайно да се използват във функционен стил.

Външна подпрограма се нарича записана в отделен файл програма на Rexx. Името, което ѝ се приписва за обръщение към нея, е това на файла. Обръщението към външна подпрограма, предаването на аргументи и връщането на стойност стават както за вътрешни подпрограми. Единствената разлика между едните и другите е, че променливите на главната програма са недостъпни във външна подпрограма, все едно че за последната е неявно зададен оператор `PROCEDURE`.

Когато програма на Rexx се стартира самостоятелно от външната среда, могат да ѝ се предават аргументи, които се обработват както в подпрограма.

Всяка програма може да се използва и самостоятелно, и като подпрограма на друга програма. Като правило, това не изисква при написването ѝ възможният двойствен начин на повикване да бъде предвиден, програмата да различава по кой от двата начина е активирана и да избира различно поведение за всеки от тях. От друга страна, ако последното е нужно, то може да се постигне, като необходимата информация се извлича чрез оператора `PARSE` с аргумент `SOURCE`.

При всяко обръщение към подпрограма посоченото име се търси най-напред сред етикетите, т. е. като вътрешна подпрограма, след това сред вградените подпрограми и накрая – сред външните. Изпълнява се първата намерена в този ред подпрограма. Такъв механизъм дава възможност да се „скрие“ вградена подпрограма, като вместо нея се употребява специално подготвена потребителска.

Ако при обръщението името на подпрограмата се огради в кавички, търси се само сред вградените и външните подпрограми, което ускорява изпълнението.

Тъй като е невъзможно стойността на променлива да бъде задавана другояче, освен чрез името ѝ, за да може дадена подпрограма да промени стойността на външна за нея променлива, в тази подпрограма съответното (единствено за променливата) име трябва да обозначава същата променлива, а не локална променлива със същото име. Последното е възможно *само ако подпрограмата е вътрешна*, а и при това се налага или името на променливата да се цитира явно в оператор `PROCEDURE EXPOSE` като глобално, или подпрограмата изобщо да няма локални променливи (като не се използва `PROCEDURE`). Недостатък на първия вариант е, че името на изменяемата променлива е фиксирано в подпрограмата – то не може да бъде избирано по различен начин при различни повиквания към нея. При втория вариант пък се дава достъп

до всички глобални променливи, което може да бъде нежелателно, защото допуска промяна на *всяка* от тях.

Компромисно решение на проблема дава механизмът на родовите имена. Променливите, подлежащи на предаване към подпрограми, могат да се именуват от сложни имена, при което родовото име играе ролята на *фамилно* и задава принадлежност по смисъл на променливата към някаква група, а във всяка група променливите се различават по индексите си, аналогични на *собствени* имена. За да се осигури достъп в подпрограма до дадена променлива, може в `EXPOSE` да се цитира нейното родово име, чрез което стават достъпни и всички променливи от съответната група.

По този начин нито се налага собственото име на променливата да се цитира явно, нито се дава достъп до всички глобални променливи. Освен това, собственото име на необходимата ни променлива може да бъде предадено като стойност на аргумент и следователно да е различно при различните обръщения към подпрограмата. Следният фрагмент пояснява прилагането на този метод:

```
        a.x = '12-те стола';  a.y = 'Воробянинов'
        SAY a.x',' a.y      /* отпечатва '12-те стола, Воробянинов' */
        CALL alt 'X', 'Златният телец'
        CALL alt 'Y', 'Паниковски'
        SAY a.x',' a.y      /* отпечатва 'Златният телец, Паниковски' */
        EXIT

alt:    PROCEDURE EXPOSE a.
        ARG p,q
        a.p = q
        RETURN
```

Косвено задавани действия

Пресмятането на израз може да дава като резултат текст, който да се тълкува като един или повече оператора на Rexx. Операторът `INTERPRET` дава възможност текст, получен по такъв начин, да се изпълни като част от програмата. Той е основното средство на езика за изпълняване на програмен фрагмент, чийто текст се получава *в хода на изпълнението на същата програма*.

Използването на `INTERPRET` позволява например пресмятане на израз, в който знакът на дадена операция се задава *косвено* като стойност на променлива, както в оператора:

```
        INTERPRET 'SAY A' P 'B'
```

който отпечатва сумата, разликата, конкатенацията и т. н. на стойностите на променливите `A` и `B` според текущата стойност на `P`, съответно `'+'`, `'-'`, `'||'` или друго.

`INTERPRET` може да се използва също за *косвено цитиране на променлива*, като името ѝ е стойност на друга променлива (в общия случай – на израз). Ако променливите `A` и `I` имат стойности `'B'` и `4`, операторът

```
        INTERPRET 't='a'.'i
```

присвоява на `T` стойността на `B.4` .

В някои случаи за косвено цитиране вместо INTERPRET е по-удобно да се използва вградената функция value(). Резултатът от прилагането на тази функция е такъв, че все едно вместо обръщението към value() в програмата стои променлива, чието име е стойност на израза-аргумент на value(). Ако X има стойност 'y' и Y има стойност 17.6, всеки от операторите

```
INTERPRET 'z = 'x'+3.2'
```

и

```
z = value(x)+3.2
```

присвоява на Z стойността 20.8.

В случаи като горния се предпочита value() пред INTERPRET като по-ограничено по възможности, но по-удобно за целта средство за задаване на косвено цитиране.

В следния пример чрез косвено цитиране се избира променлива, обект на присвояване. В случая се присвоява на променливата, чието име е стойност на x:

```
interpret x '=' 2.718
```

Такъв похват може да се използва и при предаване на име на променлива като аргумент на подпрограма, която искаме да присвоява стойност на неизвестната променлива. Трябва да имаме предвид обаче, че за да променяме нужната променлива, освен да получим името ѝ като стойност трябва и да осигурим достъп на подпрограмата до самата променлива, чрез PROCEDURE EXPOSE или изобщо без PROCEDURE, както бе описано в предния раздел.

Косвено зададени действия могат да бъдат изпълнени и чрез обръщение към външна процедура на Rexx, чийто текст е породен в хода на програмата.

Изпълняване на несобствени команди

При предаване на команди към външна изпълнителна среда, за да се подтисне пресмятането и преобразуването на съответния низ или негови части, те се ограждат в кавички. Например командата

```
dir *rex
```

за MS DOS (или аналогичната на нея ls *rex за Unix) не може да бъде зададена в този вид като несобствен оператор на Rexx. Интерпретирането ѝ би предизвикало „объркване“ у интерпретатора, една от причините за което е, че той разпознава знака * като знак на операцията умножение и се опитва да пресметне аритметичен израз. Най-добре е цялата команда да се огради в кавички:

```
'dir *rex'
```

Удобно е аргументите на несобствена команда да бъдат задавани като стойности на променливи или изрази. Горната команда може да се изпълни и така:

```
a = '*rex'
...
'dir' a
```

В общия случай като самопредставени низове следва да се задават онези части

на несобствените команди, които трябва да бъдат предадени неизменени за изпълнение, докато други части могат да бъдат стойности на променливи и дори на изрази.

След завършване на изпълнението на несобствена команда специалната променлива `RC` получава целочислена стойност, съдържаща информация за това, как е завършила командата: успешно или не, ако не – причина за неуспеха и др.

Чрез оператора `ADDRESS` се избира *изпълнителна среда* за несобствени команди. По-точно, или се посочва среда за изпълнение на отделна команда, или се установява подразбираща се среда за всички команди до следващия оператор `ADDRESS` или до края на програмата. Типични външни среди за Rexx са операционната система и текстов редактор.

Може да се направи така, че изходът на дадена несобствена команда да се изпрати във входния буфер на Rexx или направо в променливи на програмата. Например текстът, извеждан от командата `dir` по-горе може ред по ред да се присвои като стойности на семейство променливи с обща основа.

Примери

Първата от следващите програми форматира текстов файл по абзаци, като във всеки абзац всички редове освен последния са двустранно подравнени, първият ред започва с ляв или десен отстъп спрямо останалите, а последният ред е подравнен наляво.

Първият непразен ред в текста дава начало на първия абзац. Всеки път при срещане на празен ред или на непразен, който започва с един или повече интервала, текущият абзац завършва, като във втория случай новият ред започва нов абзац. За начало на нов абзац служи и непразен ред след един или повече празни. Празните редове се пренасят в изходния файл.

Освен името на входния файл, при стартиране на програмата могат да се зададат стойности за параметрите за подравняване на абзаците като номера на позиции върху реда: лява и дясна граница на реда и отстъп за първия ред на абзац. Подразбират се стойностите съответно 1, 60 и 5.

Входният файл се чете ред по ред в променливата `s`. Целият текст на даден абзац се натрупва последователно в променливата `par`, след което подпрограмата `do_paragraph` извършва форматирането му. Всеки изходен ред се образува чрез извличане на последователни думи от `par`, докато не може да побира повече думи или докато `par` се изчерпи. В първия случай вградената функция `justify()` разпределя думите в реда така, че да го запълват, като между тях се поставя необходимия брой интервали. След това тези думи се премахват от `par` и се преминава към следващия изходен ред. Във втория случай полученият ред е последен за абзаца и се подравнява по лявата граница, като чрез обръщение към функцията `space()` думите се сгъстяват така, че между всеки две да остава по точно един интервал.

Променливата `front` показва броя водещи интервали за текущия изходен ред, а `fs` е низ, съдържащ точно толкова интервала.

```
ARG fileid lmg rmg pin
IF  pin = ''  THEN  DO;  pin = 5
  IF  rmg = ''  THEN  DO;  rmg = 60
    IF  lmg = ''  THEN  DO;  lmg = 1
      IF  fileid = ''  THEN
```

```
        DO;  SAY 'Не е зададено име на файл';  EXIT;  END
    END
  END
END

par = ''
DO  WHILE lines(fileid) > 0
  s = linein(fileid)
  IF  wordindex(s,1) \= 1  THEN  DO
    IF  par \== ''  THEN  CALL do_paragraph
    par = ''
    IF  s = ''  THEN  DO;  SAY;  ITERATE;  END
  END
  par = par||s' '
END

do_paragraph:
front = pin-1;  fs = copies(' ',front)
DO  FOREVER
  d = -1
  DO  i = 1  TO words(par)  UNTIL  d > rmg-front
    d = d+1+wordlength(par,i)
  END
  IF  d <= rmg-front  THEN  DO;  SAY fs || space(par);  RETURN;  END
  SAY fs || justify(substr(par,1,wordindex(par,i)-1),rmg-front)
  par = delword(par,1,i-1)
  front = lmg-1;  fs = copies(' ',front)
END
```

Програмата лесно може да бъде усъвършенствана така, че да разпознава и изпълнява вмъкнати в текста команди за задаване на различни режими на форматиране или за отменяне на форматирането, за разбиване по страници с автоматично поставяне на номера и др.

Следващата програма служи за улесняване на работата с команди в т. нар. „конзола“ или команден прозорец на операционната система. Тя създава на екрана меню от една до девет команди и ги изпълнява при избиране от клавиатурата на съответен номер. След като дадена команда бъде изпълнена, менюто отново се появява на екрана и процесът се повтаря до завършване на работата с програмата. Потребителят може да добавя и изтрива команди в/от менюто, а също да съхранява съдържанието му за следващата употреба на програмата.

При неактивна програма съдържанието на командното меню се съхранява в същия файл, в който е записана самата тя, след текста ѝ. В началото на изпълнението си програмата прочита последните редове на файла, в който е записана, и ги присвоява на променливи с родово име `cmd.`, представляващи текущото съдържание на командното меню. Извличането на командите започва от последния ред и продължава до достигане на празен ред, но в менюто се записват не повече от девет реда-команди. Останалата част от файла – собствено текстът на програмата – се прочита и записва ред по ред в променливи с родово име `pgm.` . При завършване на програмата, ако текущото съдържание на менюто трябва да бъде запомнено, прог-

рамният файл се създава наново, като в него се записват всички променливи `pgm.` и `cmd.` . Името на файла се узнава чрез оператора `PARSE` с аргумент `SOURCE`.

Функцията `action` анализира въведеното чрез клавиатурата от потребителя и като своя стойност дава числов код на избраното действие, който се използва в главната програма. За избор на команда се въвежда число от `1` до `9`, а за изтриване на команда от менюто – число от `-1` до `-9`. В тези случаи `action` има за резултат съответното число. За добавяне на команда в менюто се въвежда буквата `a`, за завършване на програмата със запомняне на съдържанието на менюто – буквата `s`, а за завършване без запомняне – буквата `q`. Числовият код, който `action` дава в тези случаи е съответно `101`, `102` и `103`. За всички останали случаи функцията дава стойност `0`, която за главната програма означава да не извършва никакво действие.

```
cmd. = ''

n = sourceline()
DO  i = 1  WHILE  i < 10
  IF  sourceline(n) = ''  THEN  LEAVE
  cmd.i = sourceline(n);  n = n-1
END
DO  WHILE  sourceline(n) \= '';  n = n-1;  END
DO  i = 1  TO n;  pgm.i = sourceline(i);  END

CALL  display

DO  forever
  k = action()
  SELECT
    WHEN  k = 0             THEN  ITERATE
    WHEN  1 <= k & k < 10   THEN  IF  cmd.k \= ''  THEN  cmd.k;  ELSE  ITERATE

    WHEN  -9 <= k & k < 0   THEN  DO;  k = -k;  cmd.k = '';  END
    WHEN  k = 101           THEN  DO
                                    SAY
                                    DO  i = 1  TO 9  WHILE  cmd.i \= '';  END
                                    IF  i = 10  THEN  SAY 'Няма място за повече'
                                    ELSE  DO
                                      CALL charout , 'Въведи команда: '
                                      PARSE PULL cmd.i
                                    END
                                  END
    WHEN  k <= 103          THEN  DO
                                    IF  k = 102  THEN  DO
                                      PARSE SOURCE . . x .;  'del' x '>nul'
                                      DO  i = 1  TO n;  CALL lineout x, pgm.i;  END

                                      DO  i = 10  TO 1  BY -1
                                        IF  cmd.i \= ''  THEN  CALL lineout x, cmd.i

                                      END
```

```
                              END
                              EXIT
                            END
  END
  CALL  display
END

display:  PROCEDURE EXPOSE cmd.

SAY
SAY '------------------------'
DO  i = 1  TO 9;  SAY right(i':',9) cmd.i;  END
SAY
SAY '1...9: избери   -1...-9: изтрий   a: добави   s: съхр+край   q: край'

RETURN

action:  PROCEDURE

s = strip(linein())
IF  length(s)=1  THEN  DO
  IF  datatype(s,'N')  THEN  DO
    IF  1 <= s & s < 10  THEN  RETURN  s
  END;  ELSE  DO
    s = pos(s,'asq')
    IF  s>0  THEN  RETURN  100+s
  END
END;  ELSE  IF  length(s)=2  THEN
  IF  datatype(s,'N')  THEN
    IF  -9 <= s & s < 0  THEN  RETURN  s
RETURN  0

!-!-!   Следващият ред трябва да бъде празен, а след него може да има
!-!-!   команди, по една на ред
```