Увод в езика Snobol

Бойко Банчев

Общи сведения за езика

Snobol е създаден през 60-те години и е един от първите езици, предназначени главно за символно програмиране и работа с текстова информация. Загърбвайки преобладаващата дълго време ориентация изключително към числени пресмятания, този език въвежда в програмирането редица конструкции, които и днес са интересни и важни.

Макар и поизместен от по-късно създадени езици, Snobol продължава да бъде активно употребяван. Въпреки някои свои недостатъци, представяният от него съпоставително-продукционен стил на програмиране се оказа жизнен и много удобен в посочената проблемна област. С ефективния си и лаконичен подход към боравене с текстова информация Snobol и днес е ненадминат.

Главната черта, отличаваща програмирането на Snobol, е наличието на средства за *търсене на подниз в текстов низ*. Намереният подниз може да бъде *заменен с друг низ*, като по този начин първоначалният низ се променя. Търсенето в низ се извършва чрез *съпоставяне на шаблон* с части от низа. Низът, в който се извършва търсенето, се нарича *предметен низ*.

С помощта на търсенето в низ и на търсенето със замяна се реализират сложни процедури за анализ и синтез на текстова информация. Тъй като това са двете основни действия в програмите на Snobol, той е език от марковски тип.

Snobol се използва за форматиране и други преобразования на текстова информация, за анализ на текстове на естествен език и други задачи, изискващи сложна обработка на текст. Може да бъде и действително е успешно прилаган за решаване на задачи от областта на изкуствения интелект. Удобен е като целеви език за реализиране на специализирани езици за работа с текст.

Съществуват няколко реализации на интерпретатор на Snobol, включително и свободно достъпни. Да отбележим и компилатора Spitbol, който, освен че реализира вариант на езика с някои допълнения, произвежда програми с извънредно високо бързодействие.

Структура на програмата

Програмите на Snobol се състоят от оператори, биващи само четири вида: за присвояване, за съпоставяне, за замяна и за завършване на програмата.

Операторите за присвояване, съпоставяне и замяна имат съответно вида:

```
етикет  предмет = заместител          :преход
етикет  предмет шаблон                :преход
етикет  предмет шаблон = заместител   :преход
```

като и в трите вида оператори полетата за етикет и преход са незадължителни.

Операторът за завършване на програмата се поставя винаги в края ѝ, след всички други оператори. Той има етикет `END`, който може да бъде последван от име

на друг етикет в програмата, задаващ мястото, от където да започне изпълнението ѝ. За начало на програмата се подразбира първият оператор в нея.

Обикновено всеки оператор се записва на отделен ред, но се допуска два и повече последователни оператора да се записват на един ред, като се разделят със знака `;`, както и един оператор да се записва на няколко реда, като се използва знак за пренос на мястото на етикета.

Операторите за присвояване и съпоставяне могат да се смятат за частни случаи на оператора за замяна: все едно, че в оператора за присвояване е извършено успешно съпоставяне на фиктивен шаблон с целия предметен низ, а в оператора за съпоставяне се извършва фиктивна замяна, при което предметният низ не се променя.

В операторите за присвояване и за замяна заместителят може да отсъства, като вместо него се подразбира празен низ. Това съответства на изтриване на съпоставения с шаблона низ.

Допуска се и т. нар. *изроден оператор*, в който участва само предметният низ и, възможно, етикет и/или преход.

Предметният низ, шаблонът и заместителят могат да бъдат представени чрез изрази, чиито стойности отговарят на съответния контекст. Например израз със стойност шаблон може да бъде шаблон в оператор за съпоставяне или замяна, а също и заместител в оператор за присвояване.

Изрази се образуват с помощта на операции и обръщения към функции. Обозначенията на операции са предопределени в езика, но действието им може да бъде променяно в хода на програмата посредством обръщения към вградената функция `OPSYN`.

Потребителят може да определя в хода на програмата функции, които да използва наравно с вградените в езика. Определянето става с помощта на специална вградена функция, а телата на функциите представляват *синтактично отворени и динамично изменяеми процедури*. Предаването на аргументи към функция при обръщение към нея се извършва чрез присвояване на стойностите им на съответните им параметри.

Изпълнението на оператор в общ вид се състои в последователно извършване на следните действия: (1) пресмятане на предметния низ; (2) пресмятане на шаблона; (3) извършване на съпоставянето; (4) пресмятане на заместителя; (5) извършване на замяната; (6) пресмятане на прехода и (7) извършване на прехода. Когато даден елемент на оператора отсъства, действията с него се пропускат.

Програмистът има на разположение определен брой *ключови думи*: променливи, чиито имена започват с `&`. Те задават различни режими на изпълнение на програмата, предоставят информация, полезна за трасиране и др. под. Стойностите на някои ключови думи могат да бъдат променяни, а на други – само потребявани. Последните се наричат *защитени ключови думи*.

Формално се смята, че `&` е едноместна операция, чрез която от даден идентификатор се получава ключова дума, но тази операция е приложима само за предопределените за целта в езика имена.

Има няколко променливи, на които са приписани определени начални стойности. По това те приличат на ключови думи, но иначе не се различават от останалите променливи. Условно могат да бъдат наречени *системни променливи*.

Двете специални променливи `INPUT` и `OUTPUT` съответстват на стандартния входен и стандартния изходен канали. `INPUT` може да се използва като заместител или в израз, който задава заместител, а `OUTPUT` – като предметен низ, обикновено в оператор за присвояване. Например

```
OUTPUT = INPUT
```

е програма, която копира входния поток ред по ред в изходния.

Присвояванията в SNOBOL са от цитиращ тип или, както се казва, езикът има цитираща семантика. „Получаването на стойност“ от страна на променлива води до цитиране, а не на образуване на отделно копие на стойността.

Освен в оператор за присвояване или замяна, в процеса на съпоставяне също могат да бъдат извършвани присвоявания. Те могат да зависят или не от успешността на съпоставянето и се задават чрез специални операции при формиране на шаблона.

Присвояванията могат да бъдат и страничен ефект от пресмятането на предметния низ, шаблона, заместителя или прехода, доколкото всеки от тези изрази може да съдържа обръщения към функции, имащи такива последствия. В крайна сметка всяко присвояване в програмата се извършва или в оператор за присвояване, или при замяна, или при съпоставяне, ако в шаблона това е зададено.

Даннови типове

В SNOBOL има прости и съставни даннови типове. Обектите и от едните, и от другите типове се създават динамично, в хода на програмата, като стойности в резултат на пресмятане на изрази, в частност – на специални за целта операции и функции.

Числата – цели и реални – и низовете са прости даннови типове. Числата могат да се разглеждат като частен случай на низове в следния смисъл: всяка числова константа може да се разглежда като низ, съдържащ точно писменото представяне на числото, с което то е цитирано в програмата. От друга страна, всеки низ, съдържащ запис на число, може да бъде операнд на числова операция и участва в нея със стойността на това число. Освен това празният низ се тълкува като целочислена нула.

За формиране на изрази езикът разполага с традиционните аритметични операции, на които, предвид посоченото съответствие между числа и низове, може да се гледа и като на операции върху низове. Единствената операция върху низове, различна от числовите, е операцията конкатениране, за която няма специален знак, а се бележи с разделяне на двата ѝ операнда с един или повече интервала. Конкатенирането може да участва в изрази заедно с аритметичните операции и има по-нисък приоритет от тях.

Има различни вградени функции за работа с низове.

Шаблоните са друг тип стойности, подобен на низовете, но различен от тях. Шаблоните са пълноценни даннови обекти: могат да се образуват изрази, чиито стойности са шаблони; тези стойности могат да бъдат присвоявани на променливи в оператори за присвояване; шаблони могат да бъдат аргументи на някои операции и на функции и да бъдат резултат на обръщение към функция.

Всеки низ може да служи като шаблон, но шаблонът в общия случай не е низ. В езика са предвидени специални операции и функции за образуване на шаблони, които не могат да участват в образуване на низове.

Програмистът може да създава масиви, таблици и потребителски типове с именувани полета.

Особени даннови типове са идентификаторите и компилатите. Първите са подобни на указатели: те са стойности, служещи за косвен достъп до други стойности. Компилатите са програмни фрагменти, получавани чрез формиране на програмен текст в хода на самата програма и преобразуването му във вътрешно за системата представяне, така че да бъдат годни за изпълнение. Създават се чрез вградената функция `CODE`.

На променливите не се приписват типове. На всяка променлива могат да бъдат присвоявани стойности от различни типове. Носители на типовете са стойностите, а променливите имат динамично изменящи се типове, според текущата стойност на всяка.

Допускат се различни неявни и явни преобразования на обектите в програмата от един тип към друг. Типът на всеки обект може да бъде установен посредством вградената функция `DATATYPE`.

Успешност на действията и управление на изпълнението на програмата

В програмите на SNOBOL има три вида действия: пресмятане на израз, съпоставяне с шаблон и изпълняване на оператор. Структурата на управлението и в трите вида действия се основава на понятието *успешност*: всяко от тях завършва с успех или неуспех според стойностите на участващите в него величини. Успешността на дадено действие определя избора на следващото и така се формира последователността на действията в програмата.

От трите вида носители на действие само операторът има фиксирана структура и съответно такава е структурата на управлението му. Изразите могат да бъдат произволно сложни. Те се построяват рекурсивно от по-прости изрази, а в крайна сметка – от операции и обръщения към функции. Аналогично шаблоните се построяват от по-прости шаблони и т. н. до елементарните шаблони. Между операторите съществува единствено отношение на следване, зададено отчасти като фиксирано отношение, отчасти управляемо чрез успешността на изпълнението им.

При успешно завършване на пресмятането си всяка операция и всеки израз дават стойност, която участва в пресмятане на друг израз (обхващащия) или служи като аргумент при изпълнението на даден оператор. Успешно завършилият оператор обикновено има за последствие промяна на стойностите на някои променливи.

Неуспешното завършване на пресмятането на израз влече прекратяване на пресмятането с неуспех за всеки обхващащ го израз и като следствие – прекратяване с неуспех на изпълнението на оператора, на който този израз е аргумент. Неуспешното завършване на оператор, също както успешното, може да бъде съпроводено с промяна на стойностите на променливи. Макар и по начало нетипично, това понякога е необходимо, например за установяване на причината и източника на неуспеха или за съхраняване на информация за хода на изпълнението и неговата частична успешност.

Съпоставянето е сложна операция, ходът на изпълнението на която се определя както от шаблона сам по себе си, така и от съответствието на предметния низ с

него. В процеса на съпоставянето може да възниква успех или неуспех на дадено частично съпоставяне. За разлика от пресмятането на израз, това не води автоматично до прекратяване на цялото съпоставяне, а предизвиква избор на следваща алтернатива, ако има такава. Крайният резултат от съпоставянето е успех или неуспех, определящи по-нататъшното изпълнение и успешността на съответния оператор.

Средствата за задаване на шаблони предвиждат разнообразни възможности за управляване на процеса на съпоставяне.

Фактът за успешността или неуспешността на всеки завършил изпълнението си оператор може да бъде използван за посочване на онзи оператор в програмата, откъдето да продължи изпълнението ѝ. Това става в незадължително присъстващото поле за преход на оператора, където могат да се посочат адреси за преход при успех и при неуспех или адрес за безусловен преход. Определянето на наследник при успех, неуспех или безусловно за всеки оператор е единственият начин да се измени последователността на изпълнението на операторите, предписана от текстовото им следване в програмата.

Адресът за преход се огражда в кръгли скоби и е или етикет на някой от операторите в програмата, или произволен израз, чиято стойност е низ, съвпадащ с такъв етикет. Преходът при успех се бележи с `S` (от англ. success) пред скобите, преходът при неуспех – с `F` (от англ. failure), а безусловният преход не се отбелязва специално.

Като частни случаи, преходът може да цитира етикета `END` за завършване на програмата или някой от *системните етикети* `RETURN`, `FRETURN` и `NRETURN`. Последните не са действителни етикети на оператори, а обозначават завършване на действието на функция.

Както се вижда от изложеното, средствата за управление на междуоператорно равнище в SNOBOL са доста ограничени, вероятно поради представата на авторите на езика, че не тази му страна е характерна за стила на изразяване в него, а съпоставянето, за управлението на което наистина са предвидени значително по-богати средства.

Механизмът на управление чрез успешност заменя използването на булеви стойности и оператори за избор, цикъл и др. под., обичайни за други езици.

Така например аналогът на операция с булев резултат в SNOBOL е операция, която завършва с успех или неуспех, като в първия случай резултатът е някаква отнапред фиксирана стойност. Вградените в езика функции-предикати са подобни на точно такива операции, като при успех имат винаги стойност празен низ. Това се използва идиоматично за условно пресмятане на израз по следния начин: обръщението към предиката се поставя като първи аргумент в операция конкатениране, в която условно изпълняваният израз е втори аргумент. Ако предикатът завърши с успех, пресмята се изразът и в резултат се получава неговата стойност, предшествана от празен низ, т. е. непроменена. Ако предикатът не успее, изразът не се пресмята и самата операция конкатениране завършва с неуспех.

Даденият по-долу оператор е типичен за формиране на циклично управление. Ако `N` е положително, неговата стойност се намалява с `1` и се преминава към следващия в текста оператор, а в противен случай се прави преход по зададен етикет:

```
N = GT(N,0)  N - 1       :F(END_LOOP)
```

Конюнкция на предикати може да се изрази чрез конкатенирането им. При

успех на всички предикати се получава празен низ, а неуспяването на някой поред отляво надясно влече неуспех на целия израз, при това останалите предикати не се пресмятат.

Едноместната операция `~` (знакът тилда) се използва като отрицание. Тя образува успешно пресметнат израз със стойност празен низ, ако зададеният като неин аргумент израз е завършил с неуспех, и обратно: ако аргументът е пресметнат успешно, получава се неуспех. Това е единствената операция, чрез която е възможно пресмятането на израз да бъде продължено след неуспех на пресметнат негов подизраз. Прилага се обикновено за обръщане на действието на предикат.

Операцията `?` (проверка) също е едноместна. Нейната успешност съвпада с тази на аргумента ѝ. При успех резултатът на операцията `?` е празен низ. Следователно, приложена към даден израз, тази операция не отменя възможния му неуспех, но при успех подтиска получаването на стойност от израза. Прилага се за преобразуване на произволен израз в предикатен и в частност за превръщане на произволна функция в предикат, когато само успешността е от значение, а резултатът се пренебрегва.

Следният пример показва реализиране на дизюнкция с помощта на конюнкция и отрицание. Изразът `~(~(P1) ~(P2))` е успешен тогава и само тогава, когато успява поне един от предикатните изрази `P1` и `P2`. При това, ако `P1` успее, изразът като цяло се оказва успешен без изобщо да се премята `P2`. В противен случай успешността на израза зависи от тази на `P2`.

Условно разклоняване, управляван от брояч или условие цикъл и други управляващи структури на надоператорно равнище се имитират в SNOBOL по разнообразни начини с използване на успешността на действията, включително на съпоставянията, посочване на адреси за преход и вариране на адресите, задавайки ги като стойности на изрази. Последното следва да се използва умерено, тъй като по принцип прави структурата на управлението неявна.

В един от следващите раздели ще покажем друг подход за условно изпълнение чрез съпоставяне с предикатни изрази-шаблони.

Основни средства за образуване на шаблони

Както другите видове стойности, и задаването на шаблон може да става чрез непосредствено посочена стойност или чрез израз, в който подходящи операции и функции построяват шаблона от други, по-прости. Пример за елементарен, непосредствено зададен шаблон, е кой да е низ: такъв шаблон се съпоставя с еквивалентен нему подниз на предметния низ. Елементарни шаблони образуват също някои системни променливи и вградени функции.

Основните операции за построяване на изрази-шаблони са конкатенирането и алтернирането. Конкатенирането на шаблони е подобно на това при низове, като обаче се отчита и възможното присъствие на алтерниране във всяка от частите. Алтернирането се бележи със знака `|` и задава шаблон, разпознаващ всеки от низовете, съпоставяни с някой от двата алтернирани шаблона.

От гледна точка на процеса на съпоставяне с даден шаблон конкатенирането и алтернирането задават съответно конюнкция и дизюнкция на шаблони по отношение на успешността на съпоставянето им. Успехът на съпоставянето с шаблона `P1 P2` зависи от съвместното успяване на съпоставянията на последователни поднизове на предметния низ с `P1` и `P2`, при това съпоставянето с `P2` следва това с `P1` и не се

извършва, ако то не успее. Съпоставянето с шаблона `P1 | P2` е успешно при успех на поне едно от съпоставянията с `P1` или `P2`, като съпоставянето с `P2` следва това с `P1` и се извършва само ако то не успее.

Чрез посочените две операции шаблонът се формира като *редица от алтернативни образци*, последователно съпоставяни с предметния низ до достигане на успех или изчерпване. Всяка от алтернативите представлява конкатениране на низове, т. е. в крайна сметка – един определен низ. За всеки шаблон редицата от алтернативи съдържа един или повече члена, възможно безкрайно много, които се съпоставят в еднозначно определен ред. Така всеки шаблон задава не само множеството от разпознавани от него низове, но и броя и реда на последователно съпоставяните с предметния низ алтернативни един на друг подшаблони, т. е. шаблонът определя еднозначно алгоритъма на извършване на съпоставянето.

Например изразът `('A' | 'B') ('C' | 'D')` задава шаблон за разпознаване на всеки от низовете `'AC'`, `'AD'`, `'BC'` и `'BD'`. Тези четири алтернативи се съпоставят с предметния низ именно в този ред.

Стойността на ключовата дума `&ANCHOR` определя дали в операциите за съпоставяне да се търси съответствие с начален подниз на предметния низ или с произволен негов подниз. Първото се нарича режим на *фиксирано съпоставяне* и се задава с ненулева стойност на думата, а второто – *свободно*, нефиксирано съпоставяне. По подразбиране стойността на `&ANCHOR` е `0`.

Конкатенирането и алтернирането, заедно с функцията `ARBNO`, чрез която се задава *конкатениращa граница* на шаблон (вж. по-долу определението на `ARBNO`), задават множеството на т. нар. *регулярни изрази*. Регулярните изрази са шаблони, чрез които се описват формални езици от трети тип по класификацията на Н. Чомски, самите те наричани регулярни езици. В действителност Snobol, предоставяйки голямо разнообразие от вградени функции и системни променливи за образуване на шаблони, дава възможност за разпознаване на по-широк клас от езици. При това програмистът и сам може да определя такива функции.

Друг съдържателен аспект на конкатенирането, алтернирането и конкатениращата граница е, че те в най-общ смисъл съответстват на трите основни управляващи конструкции: следване, условно разклонение и цикъл. Съответствието не е точно, тъй като алтернирането по принцип предвижда локално възстановяване на контекста при неуспех. Например в израза-шаблон `(P1 | P2) P3` с предметния низ се съпоставя отначало шаблонът `P1`; ако се намери съответствие, шаблонът `P3` се съпоставя с непосредствено следващия подниз на предметния низ. При неуспех на съпоставянето с `P3` процесът на съпоставяне започва наново с `P2` и след него отново `P3`, като последствията от частично успялото съпоставяне (с `P1`), изобщо казано, биват отстранени. Изключение са безусловните присвоявания в процеса на съпоставянето, ако има такива (вж. за тях по-долу).

Основните функции, които задават елементарни шаблони и разпознаваните от тях низове са следните:

`LEN(n)` – всеки низ с дължина `n`;

`SPAN(s)`, `BREAK(s)` – максимално дълъг низ само от елементи на `s` или само от елементи не от `s`;

`ANY(s)`, `NOTANY(s)` – коя да е литера от `s`, съответно не от `s`;

`TAB(n)`, `RTAB(n)` – низ от текущата позиция до `n`, като `n` се определя броено съответно от началото на предметния низ надясно или от края му наляво;

`POS(n)`, `RPOS(n)` – празен низ, ако текущата позиция е `n`, съответно отляво надясно или обратно;

`ARBNO(p)` – низ, съдържащ нула, един или повече подниза, всеки от които се съпоставя с шаблона `p`. Редицата от алтернативи на шаблона съдържа последователно празния низ, низ, съпоставящ се с `p`, конкатенация на два низа, съпоставящи се с `p` и т. н.

Основните системни променливи, чиито стойности са шаблони, и разпознаваните от тях низове са следните:

`REM` – еквивалентно на `RTAB(0)`, т. е. остатъкът от низ, броено от текущата позиция;

`ARB` – низ от нула, една или повече литери;

`BAL` – низ, в който присъствието на скоби `(` и `)` е равновесно, т. е. левите и десните скоби са по равен брой и са правилно вложени.

Редиците от алтернативи за шаблоните на `ARB` и `BAL` са безкрайни и се образуват от низове с нарастваща дължина, аналогично на `ARBNO`.

Специалният шаблон `EMPTY` се съпоставя успешно с всеки низ, включително с празния.

Някои системни променливи формално задават шаблони, а всъщност са средства за *управление на процеса на съпоставяне* в операторите за съпоставяне и замяна.

`FAIL` се съпоставя с всеки низ неуспешно. Предизвиква отхвърляне на текущата алтернатива на шаблона и преминаване към следващата, ако такава има, или завършване на съпоставянето с неуспех. Дава възможност да се потърси повече от едно успешно съпоставяне на определен шаблон с предметния низ. Например шаблонът `(P1 | P2 | P3 | P4) FAIL` изпитва всички алтернативи подред, независимо от успеха им.

`SUCCEED` е шаблон с безкрайно много алтернативи, всяка от които отговаря на успешно съпоставяне с празния низ. Употребата на тази променлива има смисъл в шаблон, конструиран така, че при неговото съпоставяне `SUCCEED` да участва като подшаблон в две или повече алтернативи. Осигурявайки успех винаги, при това без да променя текущата позиция върху предметния низ, `SUCCEED` предотвратява пораждането на алтернативи наляво от себе си в шаблона.

Чрез `SUCCEED` и `FAIL` може да бъде определен шаблон за незавършващо (нито с успех, нито с неуспех) съпоставяне с безбройно много циклично повтаряни алтернативи. Шаблонът `SUCCEED (LEN(1) ARB) $ OUTPUT FAIL` задава последователно отпечатване на началните поднизове с дължини 1, 2 и т. н. на предметния низ, включително накрая и целия низ, след което същото се повтаря отначало безкрайно много пъти.

`FENCE` се съпоставя с празния низ. Ако частичен неуспех на съпоставянето предизвика отхвърляне на алтернатива, съдържаща `FENCE`, съпоставянето в съответния оператор завършва с неуспех. Това обуславя използването на `FENCE` за извънредно прекъсване на процеса на съпоставяне в случай, че е разпознато началото на даден низ, но не продължението му и е желателно да се предотврати по-нататъшното разглеждане на алтернативи.

Например при търсене с фиксирано начало, ако съпоставянето показва, че началото на предметния низ отговаря на някакъв шаблон, но по-късно възниква неуспех, чрез `FENCE` може да се избегне изпробването на други алтернативи за началото на низа, предвид тяхната нереализуемост.

`FENCE` се използва и за локално фиксиране в по начало нефиксиран режим

на съпоставяне. За целта в съответния оператор за съпоставяне или замяна пред шаблона се добавя FENCE. Съпоставяйки се с празния низ, FENCE привързва съпоставянето на същинския шаблон към началото на предметния низ и не допуска друга възможност.

ABORT предизвиква завършване с неуспех на оператора за съпоставяне или замяна. Използва се за задаване на условни шаблони. Типична употреба е да се построи шаблон от вида A ABORT | P, съпоставим с всеки низ, с които е съпоставим P, но не A. (Първо се извършва съпоставянето с A и ако то е успешно, задейства се ABORT, а ако не – извършва се съпоставяне с P.)

Програмата може да променя стойностите на изброените системни променливи, като по този начин влияе на поведението на съпоставящите операции. Заедно с това, сред защитените ключови думи има такива, чиито имена, с изключение на префикса &, съвпадат с имената на системните променливи REM, ARB, BAL, FAIL, SUCCEED, FENCE и ABORT. Стойността на всяка от тях съвпада с началната стойност на едноименната системна променлива. Това дава възможност както да се възстановяват стойностите на изменяните променливи, така и да се пишат програми, независещи от такива изменения, като се цитират само ключови думи вместо променливи.

Шаблони със съпровождащи присвоявания

В операторите за съпоставяне и за замяна разпознат от шаблона или от част от него подниз на предметния низ може да бъде присвоен на дадена променлива. За целта се построява специален шаблон с желания страничен ефект. Това става с помощта на двуместните операции . (точка) или $. Всяка от тях има за аргументи шаблон и променлива и образува шаблон, еквивалентен на зададения по отношение на съпоставянето, но такъв, че при успешно съпоставяне посочената променлива получава стойността на съответния подниз на предметния низ.

Чрез . се задава т. нар. *условно присвояване*: изразът P . V е шаблон, който при съпоставяне е еквивалентен на P, но присвоява на V частта от предметния низ, разпознавана от P, ако съпоставящата операция завърши успешно като цяло. Ако P бъде успешно съпоставено, но е част от шаблон, който като цяло не намира съответствие в предметния низ, присвояване не се извършва. Например операторът

```
P1 = ('AB' | 'CD') . OUTPUT
```

образува шаблон P1, който може да бъде съпоставен с всеки от низовете 'AB' и 'CD' и при успех на съпоставянето отпечатва намерения низ. Операторът

```
P2 = 'AB' | 'CD' . OUTPUT
```

образува шаблон P2, съпоставим също с 'AB' или 'CD', но само при съпоставяне с 'CD' отпечатва този низ.

Всички условни присвоявания в даден шаблон се извършват след съпоставянето, но преди пресмятането на заменящия низ, което дава възможност стойностите на съответните променливи да бъдат използвани при задаването му.

Операцията $ задава *безусловно* (непосредствено) присвояване: изразът P $ V е шаблон, който, ако бъде успешно съпоставен, присвоява на V съответната част от предметния низ, независимо от това, дали съпоставящата операция завършва успешно като цяло.

С един шаблон може да се свърже повече от една променлива. Например в израза `P $ V . W` променливата `V` е свързана безусловно с шаблона `P`, а променливата `W` – условно.

Едноместната операция `@` образува шаблон, който се съпоставя с празния низ и присвоява на дадена променлива текущата позиция в предметния низ. Присвояването е непосредствено, както при операцията `$`.

Отложени и косвено зададени пресмятания

Отложен шаблон се получава чрез едноместната операция `*`. Изразът от вида `*E` е шаблон, чиято стойност се получава като резултат от пресмятане на израза `E` непосредствено в хода на съпоставянето, в съответствие с реда на съпоставяне на компонентите на шаблона с части от предметния низ в даден оператор за съпоставяне или замяна. Ако `E` бъде успешно пресметнато, неговият резултат се използва като съставна част на шаблона в дадения оператор и съпоставянето продължава с тази част. При неуспех на `E` става връщане към друга алтернатива в шаблона.

В пресмятането на отложен израз може да се използва стойността на променлива, на която преди това в хода на съпоставянето е извършено безусловно присвояване. Това не може да се направи с условно присвоена стойност, тъй като тя е недостъпна до края на съпоставянето.

Например шаблонът `LEN(1) $ X *X` се съпоставя с двойки еднакви елементи в низ, а операторът за разпознаване

```
X (SPAN('0123456789') $ Y) 'A' *Y
```

завършва успешно, ако `X` съдържа два еднакви цифрови низа, разделени от `'A'`.

Шаблон за съпоставяне с най-дългия подниз, съпоставим с някоя от алтернативите на шаблона `P`, се задава така:

```
(*P $T *GT(SIZE(T),SIZE(S))) $ S FAIL
```

(където преди съпоставянето `S` трябва да бъде празен низ).

Отложените шаблони дават удобна възможност за *рекурсивно определяне* на шаблони. Например операторът

```
P = *P 'B' | 'A'
```

определя шаблон `P` за разпознаване на низове от вида `A`, `AB`, `ABB`, `ABBB` и т. н.

Стойността на системната променлива `ARB` (вж. по-горе) може да се зададе рекурсивно така:

```
ARB = EMPTY | LEN(1) *ARB
```

Резултатът `A` на функцията `ARBNO` (също от по-горе) може да се зададе чрез следното рекурсивно определение, в което `P` е аргументът на функцията:

```
A = EMPTY | P *A
```

Използването на отложен шаблон може, подобно на операциите `$` и `@`, да предизвика извършване на присвоявания като странично последствие от операцията съпоставяне.

Освен в операция за съпоставяне, отложен шаблон може да се пресметне и посредством функцията `EVAL`. Тя пресмята зададен като низ произволен израз или отложен шаблон.

Вградената функция `APPLY` прави обръщение към функция, при което и името на функцията, и аргументите ѝ се задават като низове – стойности на аргументите на `APPLY`.

Чрез `EVAL` или `APPLY` може не само да се пресмятат изрази и функции, но и да се изпълни програмно зададен оператор. За целта предметният низ, шаблонът и заместителят се представят с подходящи обръщения например към `EVAL`. Където е нужно, получаваната чрез `EVAL` стойност може да бъде празен низ.

В езика е предвидено и по-пряко средство за изпълняване на един или повече програмно зададени оператора. Вградената функция `CODE` създава *компилат* по зададен низ, съдържащ текста на фрагмент от програма. Компилатът става част от програмата, все едно че даденият фрагмент е записан на неопределено място в изходния ѝ текст. Единствената разлика между операторите в даден компилат и останалите е, че първият оператор в компилата няма предходник в текста, а последният няма наследник. Това обуславя наличието на специално средство за преход към началото на компилата и на специфично правило за преход след последния му оператор, което се обяснява по-долу.

Динамично определяните чрез `CODE` оператори могат, както останалите, да имат етикети. Тези етикети могат да бъдат цитирани като адреси на преход не само от оператори в самия него, а от всеки оператор в програмата. Ако етикет на оператор в компилат съвпадне с друг етикет в програмата, последният става недостъпен за сметка на първия: всеки преход към етикет със съответното име се отнася до оператора, създаден чрез `CODE`.

В частност, когато в полето за преход на някой оператор извън компилата е цитиран етикет, който бива припокрит след създаването на компилата, този преход изменя смисъла си. Операторът ще има различни наследници според това, дали се изпълнява преди или след създаването на компилата. Това ни кара да отбележим, че компилатите предоставят изключителна гъвкавост и динамизъм по отношение на формирането на структурата на програмата, но те са и средство тази структура да се направи много неявна, неочевидна, което е нежелателно. Друга потенциална опасност е това, при образуване на компилат даден етикет може да бъде припокрит непреднамерено, още повече, че текстът на компилата и в частност името на припокриващия етикет сами могат да бъдат създадени по неявен начин, в резултат на пресмятания.

Предаване на управлението към първия оператор в компилат може да стане чрез цитиране като адрес за преход в даден оператор на стойността на компилата. Такъв преход се нарича пряк и е синтактично обособен, като адресът се загражда в < и > вместо в кръгли скоби. Стойността на компилата се задава или непосредствено чрез обръщението към `CODE`, в което той бива създаден, или чрез променлива, на която е присвоен резултатът от това обръщение.

Ако след изпълнението си последният оператор в даден компилат не предизвика явен преход, изпълнението на програмата завършва.

Интерпретаторът на SNOBOL може да прилага определени евристики за ускоряване на съпоставянето с даден шаблон чрез пропускане на някои негови алтернативи. Това се нарича *опростено съпоставяне* и всъщност е подразбиращият се режим на

работа. Той може да се избере и изрично, чрез установяване на стойност `0` на ключовата дума `&FULLSCAN`. Ако на `&FULLSCAN` се зададе различна от `0` стойност, установява се режим на *пълно съпоставяне*, при който то протича в точно съответствие с определените от езика правила.

Последствията от извършване на съпоставяне в опростен режим не винаги са предвидими. Ако в шаблона на дадена операция за съпоставяне отсъстват непосредствени присвоявания под каквато и да било форма – чрез операции `$`, `@` или отложени подшаблони – опростеният режим е външно еквивалентен на режима на пълно съпоставяне: извършването на съпоставянето във всеки от двата режима води до едно и също състояние на програмата. В общия случай еквивалентност няма и поведението на програмата може да се различава от това, което предопределят правилата за извършване на операцията съпоставяне.

Шаблони без непосредствени присвоявания се използват най-вече в случаите, когато е важна *само успешността* на съпоставянето и не е нужно то да се съпровожда от допълнителни действия. В такива случаи използването на опростен режим е надеждно и ефективно.

Условни разклонения чрез фиктивни съпоставяния

Шаблонът `''` (празен низ) се съпоставя винаги успешно без да променя текущата позиция върху предметния низ. Ако в даден оператор за съпоставяне шаблонът се свежда до поредица от елементарни алтернативи, всяка от които или се пресмята неуспешно, или има за резултат празния низ, съпоставянето е *фиктивно*, тъй като предметният низ фактически не участва в него (и вследствие на това може да бъде какъвто и да е). Фиктивните съпоставяния се извършват заради пресмятането на шаблона, който в случая е и предикат в описания по-рано смисъл, т. е. играе ролята на булев израз. Успехът или неуспехът на шаблона при фиктивно съпоставяне непосредствено определя и този на целия оператор, така че в резултат се получава оператор за условно разклонение в програмата. В частност, при фиктивно съпоставяне конкатенирането и алтернирането играят ролите на конюнкция и дизюнкция над предикатни изрази-шаблони.

По-горе вече показахме, че булеви изрази могат да бъдат имитирани чрез какви да е предикатни изрази, не непременно шаблони. Това е универсален подход, тъй като дава възможност да се моделира условно пресмятане на какъв да е израз, т. е. на всяка от частите – предметния низ, шаблона, заменителя и полето за преход – на кой да е оператор. Подходът на фиктивното съпоставяне има по-ограничено приложение, защото позволява да се въвежда условност само спрямо част от изразите – шаблоните. Той обаче позволява по-непосредствено реализиране на някои булеви връзки, причина за което е възможността дизюнкцията да се изразява пряко чрез алтерниране (срв. с израза от по-горе).

Например при изпълнението на оператора

```
''      GT(N,0) | ~IDENT(S)      :S(A)F(B)
```

се извършва преход към етикета `A`, ако `N` е по-голямо от `0` или низът `S` е непразен, а в противен случай – към етикета `B`.

Фиктивните съпоставяния не изключват съпровождащи присвоявания и други странични последствия.

Променливи и идентификатори

Идентификаторите са стойности-препратки, които се използват за достъп до други стойности. Всяка променлива има уникален идентификатор, различен от тези на останалите променливи. В този смисъл идентификаторите са абстракция на реални адреси на данни в паметта на компютъра.[1]

Някои изрази имат за стойности идентификатори: това са цитати на променливи, цитати на масиви или техни елементи, на таблици или техни елементи, на полета на потребителски типове и др. Когато израз, чиято стойност е идентификатор, се цитира в потребленски контекст, идентификаторът се *разименува* и стойността на израза се оказва тази на съответната променлива. Във въздействен контекст това не се извършва.

Смята се, че за всеки непразен низ съществува променлива, чийто идентификатор е този низ. Всички променливи имат начална стойност, която е, с изключение на системните променливи, празния низ. Идентификаторите-низове се наричат *явни* идентификатори.

Синтаксисът на езика определя множество от допустими имена на променливи. Имената служат за непосредствено цитиране на променливите в програмата. Всяко име обозначава онази променлива, чийто идентификатор е низът, съдържащ името. Името на променлива като израз има стойност, равна на идентификатора ѝ.

Някои идентификатори не са низове и се наричат *неявни идентификатори*. Такива са идентификаторите на т. нар. *породени променливи*: елементите на масиви и на таблици.

Променлива, чийто идентификатор е неявен или не съответства на име, може да бъде цитирана *само косвено*.

`$` като едноместна операция се нарича операция *стойност* и има за резултат стойността на аргумента си, която трябва да бъде идентификатор, т. е. тя извършва разименуване на аргумента си.

В частност, всяка променлива, чийто идентификатор е низ, може да бъде цитирана чрез израз от вида `$'...'`. По-специално, за всеки низ, който съдържа допустимо име, горният израз е еквивалентен на израз, в който участва само името, например `$'SPY'` е еквивалентно на `SPY`. От друга страна, изразът `$'A<5>'` цитира променлива, чийто идентификатор е низът `'A<5>'`, а не се отнася до елемент на масива `A`.

Операцията `$` може да се тълкува като *косвено цитиране*. Например операторите

```
REF = 'VAR'
$REF = 1.6
```

правят същото, което и

```
VAR = 1.6
```

Операторите

```
V = 'ABC'
$(V 4) = $(V 4) + 1
```

[1] В други езици думата *идентификатор* обикновено се употребява в смисъл на име, с което в текста на програмата се обозначава променлива или нещо друго. Употребата ѝ тук е малко по-обща.

увеличават с 1 стойността на променливата ABC4, а операторът

```
N = F()          :($('LAB' N))
```

е преход-превключвател към един от множество етикети, избирани според стойността на N.

Обратната по смисъл на $ е едноместната операция *идентифициране*, която се бележи с . (точка). Тя има за аргумент израз, чиято стойност е идентификатор и дава тази стойност, като в потребленски контекст подтиска разименуването. Тази операция обобщава за произволни идентификатори действието на кавичките върху низове. За низове двете съвпадат, например .ABCD е същото като 'ABCD'.

Операцията . е необходима за получаване на (неявни) идентификатори на породени променливи. Например идентификаторът на елемента A<2,7> на масива A може да бъде зададен само с израза .A<2,7>, но не с 'A<2,7>', което е идентификатор на друга променлива.

Операцията . е нужна преди всичко за предаване на променлива към функция, така че стойността ѝ да може да бъде променяна в нея, и за получаване на идентификатор като стойност на функция. Когато е нужно стойността на променлива да бъде променена от дадена функция, като аргумент на функцията се предава идентификаторът на променливата, а към съответния параметър се прилага косвено цитиране.

Операциите $ и . не са операции в инстинския смисъл на думата, тъй като действието на всяка от тях не се обяснява само като получаване на стойност от стойността на даден аргумент. Те са по-скоро операции на синтактично, отколкото на изпълнително равнище, подобно на операциите * и & в езика C. Заедно с това и идентификаторите са особен вид стойности, тъй като изразите, имащи такива стойности, са твърде ограничени по вид.

Масиви и таблици

Масиви и таблици се създават чрез вградените функции ARRAY и TABLE.

При създаването на масив се посочват размерност, граници и, незадължително, начална стойност, обща за всички елементи на масива.

Всеки елемент на масив може да има всяка възможна стойност от кой да е тип. В един и същи масив може да има стойности от различни типове.

Синтактичното цитиране на елемент на масив става чрез името на масива и съответните индекси в скоби < и >, например Q<4,7,2>.

Цитирането на елемент на масив с невалиден индекс предизвиква неуспех. Това се използва за формиране на цикъл в програми за обхождане на масиви, когато горната граница на индекса е неизвестна.

Функцията PROTOTYPE по зададен масив дава низ, съдържащ границите на индексите му. Резултатът може да се използва например като аргумент на ARRAY за създаване на масив. Така могат да бъдат създавани масиви, чиито размери съвпадат с тези на масив-оригинал, за който те не са непосредствено известни.

Таблиците са асоциативни масиви. Цитирането на елемент на таблица е подобно на цитиране на елемент на масив, но вместо един или няколко целочислени индекса се използва произволно избрана стойност. С други думи, таблицата е мно-

жество от наредени двойки от стойности, като във всяка двойка първият елемент, наричан *ключ*, служи за идентифициране на втория.

Таблиците се отличават от масивите и по това, че броят на елементите в тях е променлив. Всяка новосъздадена таблица е празна и бива попълвана чрез присвоявания: нов елемент се образува, когато на този елемент за първи път бъде дадена стойност. Ако при създаването на някоя таблица е посочен размер, той се тълкува като указание за обема на първоначално заделяната за нея памет, но нито задава фактическия ѝ размер, нито ограничава възможния растеж на таблицата. Действителният размер на таблица е равен на броя на елементите, които са получили стойност и е по принцип неограничен и неявен.

Цитирането на елемент на масив чрез индексиране на името му не винаги е възможно, защото масивът може да е зададен посредством неявен идентификатор, получаван като стойност на израз. Синтаксисът на езика допуска индексиране само на непосредствено зададено име на масив, т. е. на явен идентификатор. Същият проблем съществува и при таблиците. И в двата случая той се решава с помощта на функцията `ITEM`.

Функцията `ITEM` дава универсално средство за достъп до елемент на масив или на таблица. Като нейни аргументи се задават изрази, чиито стойности са идентификаторът на масива или таблицата и съответните индекси.

С помощта на функцията `CONVERT` всяка таблица може да бъде преобразувана в масив с размерност 2. Броят на елементите на масива по първия индекс е равен на броя на непразните и различни от празния низ елементи в таблицата, а по втория индекс – 2. За всяка таблица `T` и съответния ѝ масив `A` елементът `A<I,2>` е равен на `T<A<I,1>>` за всяко `I`, не по-голямо от броя на елементите в таблицата. Всеки непразен елемент на `T` е представен по този начин в `A` за някое `I`.

Обратно, всеки масив с такава структура може да бъде преобразуван в таблица.

Преобразуването на таблица в масив създава „фиксиран“ екземпляр на таблицата и е удобно, когато идентифициращите елементи на таблицата са неизвестни. Табличното индексиране чрез неизвестни стойности се заменя с целочислено индексиране в масив с известни стойности на индексите.

Определени от програмиста типове

Чрез функцията `DATA` програмистът може да определя собствени типове – агрегати с именувани полета. При обръщението към `DATA` се посочват името на създавания тип и имената на полетата му. Изпълняващата система автоматично предоставя функция за създаване на обекти от типа с име, съвпадащо с името на типа, както и функции за достъп до полетата с имена като на самите полета. Всяка функция за достъп до поле има за резултат идентификатор, така че може да се използва и за получаване, и за промяна на съдържанието на това поле у променлива от съответния тип.

При изграждане на списъчни и други подобни структури в ролята на указатели се използват идентификатори на обекти.

Фрагментът

```
DATA('LISTEL(INFO,LINK)')
P = LISTEL('A',)
P = LISTEL('B',P)
P = LISTEL('C',P)
```

```
*   ... други оператори ...
        T = INFO(P) ;  P = LINK(P)
        INFO(LINK(P)) = 'Z'
```

създава даннов тип LISTEL с полета INFO и LINK, след това – елемент от тип LISTEL с 'A' в полето INFO и празно поле LINK, след него – нов елемент с 'B' в полето INFO, свързан чрез полето си LINK към другия елемент и накрая – елемент с 'C', свързан към този с 'B'. Така е получен списък с три елемента, наредени в него в ред, обратен на реда на създаването им. В предпоследния оператор съдържанието на полето INFO на първия в списъка елемент се присвоява на променливата T и този елемент се отстранява от списъка, като главата на списъка (P) получава за стойност идентификатора на следващия елемент. В последния оператор се променя съдържанието на полето INFO на втория в списъка елемент.

Функции

С помощта на функцията DEFINE могат да бъдат определяни потребителски функции. Наред с името на функцията и имената на параметрите ѝ се задават имена на локални за функцията променливи и етикет на оператор, към който да се извършва преход при обръщенията към функцията. Този етикет се нарича *входна точка* на функцията, по-точно – на *реализиращата дадената функция процедура*. Входната точка може да не се задава, ако името ѝ съвпада с това на определяната функция, което най-често се и прави.

Реализиращата процедура е *синтактично отворена*. Тя не е по никакъв начин синтактично обособена, а е просто част от програмата. Обхваща всички потенциално достижими от входната точка оператори и може да има общи части с реализиращите процедури на други функции. Текстът ѝ не е непременно последователно разположен и се намира не непременно след входната точка.

При обръщение към функция най-напред параметрите ѝ получават стойности от съответните им аргументи, след това управлението се предава във входната точка, откъдето изпълнението на програмата продължава според предписаното действие на последователно изпълняваните оператори. Пресмятането на функцията завършва, когато от някой оператор се изпълни преход, в който се цитира някой от етикетите END, RETURN, NRETURN или FRETURN. При END изпълнението на програмата завършва. FRETURN е фиктивен етикет за завършване на функцията с неуспех, а другите два такива етикета са за успех. Те се отличават един от друг по това, че при NRETURN като резултат от функцията се получава идентификатор, а при RETURN – обикновена стойност.

Имената на функцията и на нейните параметри представят локални за функцията променливи по време на изпълнението ѝ. Стойността, която има променливата с името на функцията, когато последната завършва с RETURN или NRETURN се предава като резултат на функцията в точката на обръщение.

Обръщение към функция, имаща за резултат идентификатор, може да бъде използвано аналогично на променлива в оператор за присвояване, както и като предметен низ в оператор за съпоставяне или замяна.

Допуска се рекурсивно определяне на функции.

И аргументите, и резултатът на функция могат да имат всеки възможен даннов тип.

Входната точка на дадена функция, както и останалите параметри на обръщението към DEFINE, може да бъде *преопределяна* чрез ново обръщение към DEFINE, възможно дори при изпълнението на същата функция. По този начин реализиращата процедура на функцията може да бъде изцяло или частично променяна по време на изпълнението на програмата и дори на (текущата реализираща процедура на) самата функция.

Едно възможно приложение на това е да се опише функция, част от тялото на която се изпълнява еднократно, което е удобно за инициализиране на стойности на променливи, с които работи функцията. Еднократно изпълняваната част се поставя непосредствено след първоначално посочената входна точка на функцията и наред с другите действия съдържа обръщение към DEFINE, където входната точка се променя на подходящ етикет след тази част.

Вградената функция OPSYN дава възможност на всяко име да бъде приписан смисъл на функция, чието пресмятане е еквивалентно на пресмятането на друга функция или операция, дори и ако името вече е определено като име на функция. OPSYN позволява също на знак на операция да бъде придаден смисъл на функция или друга операция.

Множеството от знакове на операции е предопределено в езика и не може да бъде изменяно. Според първоначално зададеното им назначение някои от знаковете обозначават едноместни операции, други – двуместни, трети обозначават както едноместни, така и двуместни операции, а има и знакове, на които не съответства операция. Освен това на всички знакове са приписани стойности за приоритетност и асоциативност, значими, когато тези знакове биват употребявани като двуместни операции.

Допустимо е да се задава или променя назначението на всеки от знаковете във всеки от двата му контекстни варианта – и като едноместна, и като двуместна операция, така че използването на знака в програмата в съответния контекст да предизвиква обръщение към функция или пресмятане на операция. При това обаче стойностите за приоритетност и асоциативност са неизменяеми за всеки знак.

Примери

Следните примери са адаптирани от учебника [GPP71]. Първата програма извежда броя появи на всяка дума във въвеждания текст. Текстът се прочита ред по ред в променливата LINE. Думите се извличат от LINE една по една, като за всяка се увеличава съответният ѝ брояч в таблицата WCNT, а самата тя се изтрива от LINE. За дума се смята подниз, който не съдържа никоя от литерите в променливата DELIMITER.

За отпечатване на резултатите таблицата WCNT се преобразува в масив.

```
        &ANCHOR = 1
        DELIMITER = ' .,-:;?!'
        WCNT = TABLE()
*
READ    LINE = INPUT                                :F(PRINT)
NEXT.R  LINE SPAN(DELIMITER) =
        LINE BREAK(DELIMITER) . WORD =              :F(READ)
        WCNT<WORD> = WCNT<WORD> + 1                 :(NEXT.R)
*
```

```
PRINT   OUTPUT =
        WCNT = CONVERT(WCNT,'ARRAY')              :F(END)
        I = 1
NEXT.P  OUTPUT = WCNT<I,1> ':' WCNT<I,2>          :F(END)
        I = I + 1                                 :(NEXT.P)
END
```

Следващата програма определя чрез алгоритъма на Х. Ван дали дадена съждителна формула е тъждествено истинна. Формулите включват логическите връзки отрицание, конюнкция, дизюнкция, импликация и еквивалентност, записвани като обръщения към функции, съответно едноместната `NOT` и двуместните `AND`, `OR`, `IMP` и `EQU`. Атомарните формули са произволни низове, несъдържащи интервали и играят ролята на аргументи във формулата. Формулата е тъждествено истинна, ако е вярна за всички комбинации от истинностни стойности на аргументите си.

Неатомарните формули се разпознават чрез шаблона `FORMULA`, като на едноместните функции отговаря подшаблонът `UNOP.FORMULA`, а на двуместните – `BINOP.FORMULA`. На променливите `PHI` и `PSI` се присвояват аргументите на разпознатата функция, а името ѝ се запомня в променливата `OP`. Шаблонът `ATOM` се съпоставя с атомарна формула, която при това се записва в променливата `A`. Определянето на шаблоните става еднократно, при първото обръщение към функцията `WANG`, след което нейната входна точка се променя по подходящ начин.

В главната програма се въвежда низ, който се оценява от функцията `WANG` като вярна или невярна формула, след което същото се повтаря с друг низ и т. н. Ще опишем кратко действието на функцията, без да обосноваваме валидността на алгоритъма, реализиран от нея.

Определянето на истинността на формула става чрез редица от преобразования на секвенции, всяка с предпоставка `ANTECEDENT` и следствие `CONSEQUENT`. Започва се с празна предпоставка и следствие, съвпадащо с подлежащата на доказване формула.

Всяка секвенция бива преобразувана в една или две други чрез рекурсивни обръщения към функцията `WANG`: ако `ANTECEDENT` съдържа формула, извършват се едни преобразования – операторите след етикета `WANG.A` до етикета `WANG.C`; ако горното не е вярно, но `CONSEQUENT` съдържа формула, извършват се други преобразования – операторите след етикета `WANG.C` до етикета `WANG.E`. И в двата случая нужното действие се избира, като в полето за преход на оператора след етикета `WANG.A` или `WANG.C` етикетът за успех се получава чрез израз, в който участва стойността на променливата `OP` – името на разпознатата функция.

В крайна сметка във всеки рекурсивен клон се достига до секвенция, на която двете части са редици от атоми (всяка от които може да бъде празна). Истинността на такава секвенция се определя непосредствено в цикъла след етикета `WANG.E`: тя се смята за истинна, ако предпоставката и следствието имат поне един общ атом. Окончателно, формулата е тъждествено вярна, ако са верни всички секвенции, получени от изходната при рекурсивното ѝ привеждане.

Например формулата `IMP(AND(NOT(P),NOT(Q)),EQU(P,Q))` дава последователно получаваните една от друга секвенции

```
                    ==> IMP(AND(NOT(P),NOT(Q)),EQU(P,Q))
AND(NOT(P),NOT(Q))  ==> EQU(P,Q)
     NOT(P) NOT(Q)  ==> EQU(P,Q)
            NOT(Q)  ==> EQU(P,Q) P
```

```
                    ==> EQU(P,Q) P Q
```

От последната се получават по двата рекурсивни клона съответно `P ==> P Q Q` и `Q ==> P Q P`. И двете атомарни секвенции са истинни и следователно формулата е вярна. Формулата `IMP(IMP(OR(P,Q),OR(P,R)),AND(P,IMP(Q,R)))` води до секвенцията `IMP(OR(P,Q),OR(P,R)) ==> AND(P,IMP(Q,R))`, от която се получават два рекурсивни клона за

```
          OR(P,R) ==> AND(P,IMP(Q,R))
```

и

```
                    ==> AND(P,IMP(Q,R)) OR(P,Q)
```

Първата от тези две секвенции при по-нататъшното си рекурсивно преобразуване дава `P ==> AND(P,IMP(Q,R))` и `R ==> AND(P,IMP(Q,R))`, а първата от тях на свой ред води до `P ==> P` и `P ==> IMP(Q,R)`. От последните две секвенции първата очевидно е истинна, но втората се трансформира в `P Q ==> R`, което е неистина и влече резултат неистина за цялата формула. Останалите рекурсивни клонове не се изпълняват.

```
        DEFINE('WANG(ANTECEDENT,CONSEQUENT)PHI,PSI','WANG.I')
*
        &ANCHOR = 0
        &FULLSCAN = 1
        &TRIM  = 1
*
READ
        EXPRESSION = INPUT                              :F(END)
        OUTPUT =
        OUTPUT = 'Формула:  ' EXPRESSION
        OUTPUT = WANG(, ' ' EXPRESSION) 'вярна'         :S(READ)
        OUTPUT = 'невярна'                              :(READ)
*
*
WANG.I
        UNOP = 'NOT'
        BINOP  = 'AND' | 'OR' | 'IMP' | 'EQU'
        UNOP.FORMULA = ' ' (UNOP . OP) '(' (BAL . PHI) ')'
        BINOP.FORMULA = ' ' (BINOP . OP) '(' (BAL . PHI) ',' (BAL . PSI) ')'
        FORMULA = UNOP.FORMULA | BINOP.FORMULA
        ATOM = (NOTANY(' ') (BREAK(' ') | REM)) . A
*
        DEFINE('WANG(ANTECEDENT,CONSEQUENT)PHI,PSI','WANG.A')
*
WANG.A
        ANTECEDENT FORMULA =                    :F(WANG.C)S( $('WANG.A.' OP) )
WANG.A.NOT
        WANG(ANTECEDENT, CONSEQUENT ' ' PHI)            :S(RETURN)F(FRETURN)
WANG.A.AND
        WANG(ANTECEDENT ' ' PHI ' ' PSI, CONSEQUENT)    :S(RETURN)F(FRETURN)
WANG.A.OR
        WANG(ANTECEDENT ' ' PHI, CONSEQUENT)            :F(FRETURN)
```

```
        WANG(ANTECEDENT ' ' PSI, CONSEQUENT)            :S(RETURN)F(FRETURN)
WANG.A.IMP
        WANG(ANTECEDENT ' ' PSI, CONSEQUENT)            :F(FRETURN)
        WANG(ANTECEDENT, CONSEQUENT ' ' PHI)            :S(RETURN)F(FRETURN)
WANG.A.EQU
        WANG(ANTECEDENT ' ' PHI ' ' PSI, CONSEQUENT)    :F(FRETURN)
        WANG(ANTECEDENT, CONSEQUENT ' ' PHI ' ' PSI)    :S(RETURN)F(FRETURN)
*
WANG.C
        CONSEQUENT FORMULA =                  :F(WANG.E)S( $('WANG.C.' OP) )
WANG.C.NOT
        WANG(ANTECEDENT ' ' PHI, CONSEQUENT)            :S(RETURN)F(FRETURN)
WANG.C.AND
        WANG(ANTECEDENT, CONSEQUENT ' ' PHI)            :F(FRETURN)
        WANG(ANTECEDENT, CONSEQUENT ' ' PSI)            :S(RETURN)F(FRETURN)
WANG.C.OR
        WANG(ANTECEDENT, CONSEQUENT ' ' PHI ' ' PSI)    :S(RETURN)F(FRETURN)
WANG.C.IMP
        WANG(ANTECEDENT ' ' PHI, CONSEQUENT ' ' PSI)    :S(RETURN)F(FRETURN)
WANG.C.EQU
        WANG(ANTECEDENT ' ' PHI, CONSEQUENT ' ' PSI)    :F(FRETURN)
        WANG(ANTECEDENT ' ' PSI, CONSEQUENT ' ' PHI)    :S(RETURN)F(FRETURN)
*
WANG.E
        ANTECEDENT ATOM =                               :F(FRETURN)
        CONSEQUENT A                                    :S(RETURN)F(WANG.E)
*
END
```

Да отбележим, че десет от петнадесетте рекурсивни обръщения в тази програма представляват остатъчна рекурсия. С цел повишаване на ефективността на изпълнението те могат да бъдат заменени с преходи към `WANG.A`, предхождани от съответните за всеки от случаите присвоявания, но с това програмата става значително по-малко нагледна. По-подходящо е такива преобразования да се извършват автоматично от транслатора.