ОРДЕНА ЛЕНИНА
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
АКАДЕМИИ НАУК СССР

В.Ф. Турчин

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ РЕФАЛ

III. Программирование на базисном рефале

Препринт № 44 за 1971 г

Москва

ОРДЕНА ЛЕНИНА ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ

АКАДЕМИИ НАУК СССР

В.Ф.ТУРЧИН

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ РЕФАЛ

Ш. ПРОГРАММИРОВАНИЕ НА БАЗИСНОМ РЕФАЛЕ

Москва, 1971 г.

I. Базисный рефал

Введя некоторые ограничения на средства, предусмотренные Формальным описанием рефала, мы получим язык, который, с одной стороны, допускает достаточно простую реализацию, а с другой стороны, достаточно богат для компактной и прозрачной записи алгоритмов языковых преобразований. Мы назовем этот язык (подъязык рефала) базисным рефалом. Описание рефала в работах [3] и [6] относится, с точки зрения нынешней терминологии, к базисному рефалу (с точностью до небольших модификаций). Рефал-трансляторы, описанные в работах [4,8] также являются трансляторами для базисного рефала[x].

В базисном рефале вводятся следующие ограничения.

I. Запрещается использование рекурсивных свободных переменных, иначе говоря, после указателя переменной S, W или E должен следовать объектный знак.

2. Запрещается использование символов обмена '§' и '∋', но разрешается использование машинных процедур закапывания и выкапывания ('ЗК' и 'ВК'), как они описаны в Главе 3. Таким образом, обмен информацией между полем зрения и полем памяти допускается лишь в той степени, в какой это возможно с помощью процедур 'ЗК' и 'ВК'. Лучше всего описать эту ситуацию как полностью фиксированное поле памяти при дополнительной возможности присваивания именам значений.

---

x) В настоящее время (апрель 1971 г.) заканчивается отладка компилятора с полного рефала. Список литературы см. в 5-ом выпуске данной серии

Открытые и закрытые переменные. Форматы функций.

Наличие открытой переменной выражения в левой части предложения порождает необходимость просмотра отождествляемого выражения. Поэтому открытыми переменными следует пользоваться только в тех случаях, когда просмотр необходим по смыслу алгоритма. Мы приводили в предыдущей главе пример простой процедуры просмотра, заменяющей все знаки + (на верхнем уровне скобочной структуры) знаками - (См.предложения § 8 на стр. 2/28). Приведем еще один пример. Пусть надо просмотреть данное выражение, и каждую группу из нескольких подряд идущих пробелов заменить на один пробел. Пробел мы рассматриваем как объектный знак, изображаемый на письме "корытом" ␣. Говоря о просмотре, мы подразумеваем и всегда будем подразумевать в дальнейшем просмотр без захода в скобки, то есть только на верхнем уровне скобочной структурн. Такое понятие просмотра мы будем считать первичным, а имея в виду просмотр с заходом в скобки, будем это специально оговаривать и называть такой просмотр сквозным. Задача решается следующими двумя предложениями:

§ I4.I Ликвидация повторных пробелов

К 'ЛИКППР' Е1 ␣ ␣ Е2 = Е1 К 'ЛИКППР' ␣ Е2 .

§ I4.2 К 'ЛИКППР' Е1 = Е1

Принцип, исходя из которого написаны эти предложения, можно сформулировать так: не должно быть ни одной пары пробелов, идущих подряд: ␣ ␣ Свободную переменную Е1 мы выносим за область действия знака К так как она не содержит ни одной пары ␣ ␣ и,следовательно, не нуждается в просмотре. Один

из пробелов стирается, другой должен остаться в аргументе функции 'ЛИКПР' чтобы обеспечить стирание следующего за ним пробела, если таковой окажется.

Задача 4.I. Описать на рефале алгоритм: всюду, где встречаются два подряд идущих одинаковых терма разделить их косой чертой /

Часто бывает необходимо как-то разметить объект работы, разделить его на части, играющие различные роли в процессе выполнения алгоритма. Пусть, например, задано некоторое преобразование (функция) α и надо построить преобразование β состоящее в том, что α применяется столько раз, сколько термов входит в преобразуемое выражение, причем сначала α применяется к первому терму, а затем - к уже преобразованной части, удлиненной на очередной терм. Сначала попробуем решить эту задачу, используя для отделения преобразованной части от непреобразованной символ P (флажок), в предположении, что такой символ не встречается в обрабатываемом выражении ни до, ни в процессе преобразования. Так как в начале работы преобразованная часть пуста, будем предполагать, что обращение к функции β имеет вид:

$$K\,\beta\,P\,\langle \mathcal{E} \rangle .$$

где $\langle \mathcal{E} \rangle$ - выражение, подлежащее преобразованию. И потребуем, чтобы после окончания работы функция β сама удалила ставший теперь ненужным флажок. Этим требованиям отвечает следующая группа предложений:

§ I5.I $K\,\beta\,E1\,P\,W\,A\,E2 \geq K\,\beta\,K\,\alpha\,E1\,W\,A\,.\,P\,E2.$

§ I5.2 $K\,\beta\,E1\,P \geq E1$

Кроме того неудобства, что на объект работы и преобразование α накладывается определенное ограничение (отсутствие символа Γ это решение имеет гораздо более серьезный недостаток. Предложение § I5.I содержит открытую переменную $\underline{E}1$, поэтому на каждом шаге своей работы рефал-машина будет совершать просмотр, ища флажок. Это совершенно лишний просмотр, ибо по смыслу алгоритма машина должна на каждом шаге только отделить очередной терм и вместе с преобразованной частью отдать его функции α

Поэтому формат функции β надо определить таким образом, чтобы сделать $\underline{E}1$ закрытой переменной. Для этого в качестве разделителя надо использовать не символ, а скобки. Итак, будем преобразованную часть заключать в скобки, обращение к функции β примет вид

$$\underline{K}\,\beta\,(\,)\,\langle e\rangle\,.$$

а ее описание -

§ I6.I $\underline{K}\,\beta(\underline{E}1)\,\underline{W}A\,\underline{E}2 \;=\; \underline{K}\,\beta\,(\underline{K}\alpha\;\underline{E}1\;\underline{W}A\;.)\,\underline{E}2\;.$

§ I6.2 $\underline{K}\,\beta\,(\underline{E}1) \;=\; \underline{E}1$

Теперь рефал-транслятор, не просматривая выражения $\underline{E}1$ перескакивает его по скобкам, выделяет очередной терм $\underline{W}A$ и совершает перестройку, диктуемую правой частью предложения. Здесь нет лишних просмотров и нет никаких ограничений на объекты работы.

Можно сформулировать следующее правило: когда какие-либо выражения надо оставлять в аргументе функции, но просматривать при конкретизации этой функции не надо, их следует заключать в

скобки. Таким образом, в рефале в качестве разделителей служат, в основном, скобки. Разделитель-символ можно использовать в тех случаях, когда это не приводит к дополнительным просмотрам. Например, если некоторое выражение так или иначе должно просматриваться терм за термом, то признаком окончания просмотра вполне может служить символ. Используются символы - разделители и в тех случаях, когда отделяемые объекты не велики, и потерей времени на их дополнительные просмотры пренебрегают. Может возникнуть вопрос: зачем, вообще, использовать в качестве разделителей символы, когда есть скобки? Ответ таков: во-первых, символ занимает одно звено, в то время как пара скобок - два звена; во-вторых, иногда удобно ввести разделитель-символ, чтобы избежать нагромождения скобок.

Пару скобок, служащих для отделения части аргумента, мы будем называть сумкой. Так, в приведенном выше примере мы помещаем в сумку преобразованную часть выражения. Сумок может быть и несколько. Наличие сумок, и вообще, разделителей, придает аргументу функции определенную структуру, которая повторяется во всех предложениях, относящихся к данной функции. Структуру аргумента функции мы будем называть ее форматом. В группе § I6 аргумент имеет структуру с одной сумкой:

$$(\langle e_1 \rangle)\ \langle e_2 \rangle$$

Два предложения этой группы распознают два частных вида выражения $\langle e_2 \rangle$ когда оно содержит хотя бы один терм (то есть имеет вид W A E2), и когда оно пусто. О выражениях $\langle e_1 \rangle$ и $\langle e_2 \rangle$ можно говорить, как о двух аргументах функции β Понятие о числе аргументов, имеющее важное значение и четкий

смысл для обычных функций математики, становится бессодержательным для функций, определенных на множестве выражений, ибо любое число выражений может быть соединено - с участием или без участия разделителей - в одно выражение. Более того, как бы мы ни записали комбинацию из нескольких выражений, она с точки зрения рефала снова будет выражением. Поэтому во избежание путаницы мы всегда считаем, что формально все функции рефала суть функции от одного аргумента, имеющего, быть может ту или иную структуру. Но в тех случаях, когда структура аргумента точно указана, мы будем называть отдельные элементы этой структуры также аргументами, апеллируя к содержанию алгоритма и надеясь, что это не вызовет недоразумений.

Аргумент функции, состоящий из двух частей, можно записать и в таком виде:

$$\langle e_1 \rangle \, (\langle e_2 \rangle)$$

а также - ради симметрии - и в виде структуры с двумя сумками

$$(\langle e_1 \rangle) (\langle e_2 \rangle)$$

хотя с точки зрения эффективности вторая сумка - излишняя.
Более сложные аргументы могут иметь такие структуры как

$$(\langle e_1 \rangle) (\langle e_2 \rangle) \langle e_3 \rangle$$

$$\langle e_1 \rangle \, (\langle e_2 \rangle) (\langle e_3 \rangle)$$

$$(\, (\langle e_1 \rangle) \langle e_2 \rangle \,) \langle e_3 \rangle$$

$$(\langle e_1 \rangle) (\langle e_2 \rangle) (\langle e_3 \rangle) \langle e_4 \rangle$$

и т.п. Можно также вставлять между сумками мнемонические симво-

лы для удобства программиста, например:

§ К 'ПРЕОБР' (Е1) 'ОПЕРАТ' (Е2) 'РЕЗУЛЬТ' (Е3) ≡

Приведем пример функции со сложным аргументом. Пусть надо сравнить два выражения и составить список тех термов, которые имеют один и тот же порядковый номер в обоих выражениях и тождественны друг другу. Оба исходных выражения надо сохранить, заключив их в скобки, и приписать к ним полученный список термов, также заключенный в скобки. Аргумент этой функции будет иметь структуру

$$((\langle \mathcal{E}_1 \rangle)\langle \mathcal{E}_2 \rangle)\,(\,(\langle \mathcal{E}_3 \rangle)\langle \mathcal{E}_4 \rangle\,)\langle \mathcal{E}_5 \rangle$$

где $\langle \mathcal{E}_1 \rangle$ и $\langle \mathcal{E}_2 \rangle$ – просмотренная и не просмотренная части первого выражения, $\langle \mathcal{E}_3 \rangle$ и $\langle \mathcal{E}_4 \rangle$ – то же для второго выражения, а $\langle \mathcal{E}_5 \rangle$ – накапливаемый список термов. Функция описывается группой предложений:

§ 17.1 К φ((Е1)ТАЕ2)((Е3)ТАЕ4)Е5 ≡

К φ((Е1ТА)Е2)((Е3ТА)Е4)Е5ТА .

§ 17.2 К φ((Е1)ТАЕ2)((Е3)ТВЕ4)Е5 ≡

К φ((Е1ТА)Е2)((Е3ТВ)Е4)Е5 .

§ 17.3 К φ((Е1)Е2)((Е3)Е4)Е5 ≡ (Е1Е2)(Е3Е4)(Е5)

Задачи

4.2 Подвергнуть следующие два выражения

1 (x+y) B

A A B B C C D D

действию процедуры φ (§ I7). Написать выражение, которое надо поместить в поле зрения (обращение к процедуре) и проследить за его трансформацией в процессе работы рефал-машины.

4.3. Описать процедуру ψ, просматривающую выражение терм за термом, и применяющую к очередному терму процедуру α если этот терм встречается в обработанной части выражения (на верхнем уровне скобочной структуры) и процедуру β - в противном случае. Испробовать два формата: $(\varepsilon 1)\,\varepsilon 2$ и $\varepsilon 2\,(\varepsilon 1)$ где $\varepsilon 1$ обработанная часть, и сравнить работу рефал-машины в обоих случаях.

3. Размерность просмотра.

Если левая часть предложения содержит лишь одну открытую переменную, то такой просмотр можно назвать простым или линейным. Число элементарных операций, необходимых в общем случае для выполнения такого просмотра, пропорционально числу термов в просматриваемом выражении. Но левая часть предложения может содержать и несколько открытых переменных. Это порождает более сложные просмотры. Число открытых переменных в левой части предложения назовем размерностью просмотра, порождаемого этим предложением. Просмотр ν-ой размерности требует, вообще говоря, $\mathrm{const}\ N^{\nu}$ элементарных операций, где N - число термов в конкретизируемом выражении.

Приведем простой пример квадратичного просмотра. Пусть надо обнаружить в объектном выражении наличие двух одинаковых термов. Это может быть достигнуто предложением с левой частью

$$K\,\varphi\,\varepsilon 1\,W\,A\,\varepsilon 2\,W\,A\,\varepsilon 3 =$$

Здесь две открытых переменных: Е1 и Е2, поэтому размерность просмотра $\nu = 2$. Посмотрим, какие действия будет совершать рефал-машина. Сначала она придает переменной Е1 пустое значение, то есть в качестве терма WA выберет первый терм выражения. Затем пустое значение будет придано переменной Е2, и если второй терм не совпадает с первым, переменная Е2 будет удлиняться в поисках такого терма. Просмотрев всё выражение и не найдя нужного терма, машина приступит к удлинению переменной Е1 то есть выберет в качестве WA второй терм и снова станет искать тождественный ему терм и т.д. Таким образом, рефал-машина совершает именно те, и только те действия, которые необходимы по содержанию алгоритма, так что работа рефал-транслятора будет столь же эффективна, как если бы была написана специальная программа.

Пусть в заданной последовательности символов надо найти две тождественные цепочки, начинающиеся символом A и кончающиеся символом Z. Запишем соответствующую левую часть:

К φ Е1 A ЕХ Z Е2 A ЕХ Z Е3 ≥

Она порождает просмотр третьей размерности. Первая открытая переменная - Е1, вторая - ЕХ (первое вхождение), третья - Е2 Переменная Е3 - закрытая. Проанализировав работу рефал-машины, мы найдем, что в тех случаях, когда отождествление возможно, рефал-машина, как и в предыдущем примере, совершит лишь абсолютно необходимые действия, но если отождествление невозможно, может случиться, что машина будет совершать лишние действия. Мы покажем это, несколько упростив наш пример. Пусть левая часть предложения имеет вид:

К φ Е1 A ЕХ Z Е2 ≥

Допустим, что в просматриваемом выражении встречается много символов A , но нет ни одного символа Z . Найдя первый символ A , рефал-машина станет удлинять E.X, ища Z Из того, что символ Z не найден, уже можно сделать вывод о невозможности отождествления. Однако глупая рефал-машина, как она определена в Формальном описании, будет удлинять E.1 и каждый раз искать несуществующий символ Z . Только перебрав все символы A , она объявит о невозможности отождествления. Этот недостаток абстрактной рефал-машины может быть, конечно, исправлен на уровне реализации. Хороший рефал-транслятор должен распознавать такие ситуации и не делать лишних просмотров.

Однако, можно и не полагаясь на транслятор написать программу в таком виде, чтобы исключить лишние просмотры. Для этого надо просто более детально описать процесс поиска нужных объектов и вовремя прекратить поиск, ставший бессмысленным. Пусть левая часть, которую мы приводили выше, используется в процедуре:

§ I8.I Выделение цепочки, начинающейся символом A
и кончающейся символом Z

K 'выдцAZ' E.1 A E.X Z E.2 = A E.X Z

§ I8.2 Если нужной цепочки нет

K 'выдцAZ' E.1 = 'увы'

Разделив алгоритм поиска на две части и введя вспомогательную функцию, дадим следующее описание той же процедуры:

§ I9 A.I Сначала находим A :

K 'выдцAZ' E.1 A E.2 = K 'иск Z' (E.1 A) E.2 .

§ I9 A.2 K 'выдцAZ' E.1 = 'увы'

§ I9 Z.I Теперь ищем Z :

```
К 'искZ' (Е1 A) ЕX Z Е2 = A ЕX Z
§ 19Z.2  К 'искZ' Е1 = 'увы'
```

Использование предложений с несколькими открытыми переменными приводит, вообще говоря, к просмотрам высших размерностей. Поэтому оно полностью оправдано лишь в случае, когда такой просмотр необходим по смыслу алгоритма. Если же его можно избежать, то решение вопроса зависит от используемого рефал-транслятора. Если он автоматически устраняет ненужные просмотры, можно спокойно пользоваться краткой записью. Если же он выполняет все действия, предписываемые формальным описанием, придется либо пожертвовать эффективностью, либо описать алгоритм более детально путем введения вспомогательных функций. Впрочем, используя рекурсивные переменные, можно сделать краткую запись абсолютно эффективной.

Задача 4.4. Описать одним предложением рекурсивную функцию φ выделяющую из заданной последовательности символов последовательность, ограниченную звездочками * , и содержащую хотя бы один символ из заданного списка символов. Использовать форматы (Е С) Е П и Е П (Е С) , где Е С - список символов, а Е П - объектная последовательность. Определить размерность просмотра, сравнить алгоритм поиска при том и другом формате. Описать на базисном рефале эффективный алгоритм, решающий ту же задачу.

4. Устранение открытых переменных

В левой части предложения § 18.1 две открытые переменные. Предложения группы § 19 содержат в левых частях не более чем по одной открытой переменной. Размерность просмотра, необходимого для отождествления, снижается с двух до единицы. Естественно

поставить вопрос: нельзя ли вообще обойтись без открытых переменных? Ответ на этот вопрос утвердительный. Конечно, устранение открытых переменных не устраняет просмотра, если он необходим по смыслу алгоритма, но просмотр этот осуществляется не в процессе отождествления, а путем многократного обращения к одной и той же рекурсивной функции или к ряду рекурсивных функций, вызывающих одна другую. Общий принцип устранения открытой переменной таков: с помощью введения вспомогательной функции описать процедуру набора (удлинения) данной открытой переменной, выполняемую при отождествлении . При этом можно ликвидировать ненужные просмотры, что и было продемонстрировано в приведенном выше примере.

Устраним открытую переменную $\underline{\in}1$ в § I9 А.I. Нам придется ввести вспомогательную функцию α , содержащую сумку, куда будет набираться терм за термом значение переменной $\underline{\in}1$

§ 20 Начальное значение пусто:

$\underline{К}$ 'ВЫДЦАТ' $\underline{\in}1 \supseteq \underline{\leq} \alpha ( ) \underline{\in}1 .$

§ 20 А.I Когда дошли до символа А

$\underline{К} \alpha (\underline{\in}1) A \underline{\in}2 \supseteq \underline{\leq}$ 'ИСКАТ' $(\underline{\in}1 A) \underline{\in}2 .$

§ 20 А.2 Пока до него не дошли:

$\underline{К} \alpha (\underline{\in}1) \underline{\vee} X \underline{\in}2 \supseteq \underline{\leq} \alpha (\underline{\in}1 \underline{\vee} X) \underline{\in}2 .$

§ 20 А. 3 Когда его нет:

$\underline{К} \alpha (\underline{\in}1) \supseteq$ 'УВЫ'

Предложения § 20 А.I и § 20 А.3 почти совпадают с предложениями § I9 А.I и § I9 А.2, соответственно. Отличие состоит только в том, что открытая переменная $\underline{\in}1$ заключается в скобки и становится,тем самым,закрытой. Удлинение $\underline{\in}1$ описывается рекурсивно предложением § 20 А.2. Так как оно является более общим, чем

§ 20 A.I, оно должно быть помещено после него.

В простейших случаях устранение открытой переменной возможно без введения вспомогательных функций. Классический пример замены знаков + на – (§ 8,) можно описать как просмотр терма за термом:

§ 2I.I К φ+ Е2 = – К φ Е2 .

§ 2I.2 К φ W A Е2 = W A К φ Е2 .

§ 2I.3 К φ =

Это описание хуже, чем § 8 оно содержит больше предложений и выполняется медленнее, так как требует рекурсивного обращения к функции φ для каждого терма. В тех случаях, когда наличие открытой переменной не требует лишних просмотров, оно приводит к ускорению работы машины.

Так как в рефале в качестве основного направления просмотра выражений принято направление слева направо, возникает определенная асимметрия по отношению к понятиям первый и последний, то-есть к возможностям обработки выражений с того и другого конца. Так, отщепление подвыражения до первого заданного терма, например, знака +, задается просто левой частью:

К φ Е1 + Е2 =

в то время, как распознавание последнего терма заданного вида с помощью открытой переменной невозможно. В полном рефале эта асимметрия может быть в значительной степени преодолена путем использования специальных рекурсивных переменных (см. выпуск 4). В базисном рефале просмотр справа налево описывается рекурсивно

как просмотр терма за термом, подобно тому как описывается просмотр слева направо при устранении открытых переменных.

Процедура φ отщепляющая участок за последним знаком +, описывается предложениями:

§ 22 К φ Е1 = К ψ Е1 ( )=

§ 22 А.1 К ψ Е1 + (Е2) = Е2

§ 22 А.2 К ψ Е1 WA (Е2) = К ψ Е1 (WA Е2)=

§ 22 А.3 К ψ (Е2) = 'увы'

Задачи:

4.5. Устранить открытую переменную в описании процедуры 'иск z' , § 19 Z .

4.6. Описать алгоритм нахождения последней пары примыкающих друг к другу одинаковых термов. Процедура должна оставить только эту пару, стерев остальную часть выражения.

5. Размножение переменных

Если некоторая свободная переменная встречается в правой части предложения в большем числе экземпляров, чем в левой части, то мы говорим, что эта переменная размножается. Размножение переменной требует от рефал-транслятора физической переписи ее проекции в поле зрения, снятия с нее копии, в то время как переменные, которые не размножаются, требуют только перешивки концов их проекций. Поэтому, например, время, которое транслятор затрачивает, применяя предложение

§ К φ (Е1) Е2 = (Е2) Е2

может в сотни раз превышать время, затрачиваемое при применении предложения

§ К φ (Е1)Е2 = (Е2)Е1

Когда мы употребляем переменную в правой части не в большем числе, чем она встречается в левой части, мы даем рефал-транслятору указание переставить те или иные объекты или удалить их. Если же число вхождений переменной в правую часть хотя бы на единицу больше чем в левую часть, то это указание транслятору размножить один из объектов, то-есть изготовить один или несколько тождественных объектов. Наряду с соображениями об открытых и закрытых переменных, эти соображения необходимо учитывать при составлении на рефале эффективных программ. Вообще говоря, следует использовать размножение лишь в тех случаях, когда оно необходимо по смыслу алгоритма. В то же время, иногда бывает удобно размножить объект с тем, чтобы затем его уничтожить (обычно, по частям). С точки зрения эффективности такая процедура может быть вполне допустимой, если размножаемые объекты не слишком велики или размножение происходит не слишком часто.

Аналогичные соображения применимы к размерам правой части предложения. Необходимо учитывать, что если правая часть содержит много символов, которых нет в левой части, то транслятору понадобиться заметное время, чтобы вписать эти символы в поле зрения. И если эти символы на следующем этапе становятся ненужными и удаляются из поля зрения, а затем снова оказываются нужны и снова вписываются и удаляются, и так далее, то такой режим работы нельзя признать эффективным. Вместо этого следует в начале работы закопать нужный список символов в поле зрения и выкапывать его, когда в нем появляется потребность, не забывая, конечно, снова закапывать его после использования. Процедуры закапывания и выкапывания

выполняются, как известно, весьма быстро, независимо от размеров списка.

Рассмотрим такой пример. Пусть надо описать процедуру 'ПОРЯДОК' , которая будучи применена к двум русским буквам, заключает их в скобки, если они расположены в том порядке, в котором они входят в алфавит или совпадают друг с другом, и приписывает спереди знак отрицания ¬ , если они расположены в обратном порядке. Простейший вариант описания таков:

```
§ 23. I  K 'ПОРЯДОК' SA SA = (SA SA)

§ 23.2  Необходима дополнительная информация - алфавитный
        порядок букв

    K 'ПОРЯДОК' E1 = K∝ E1 АБВГ    эюя .

§ 23 А.I  K∝ SA SB E1 SA E2 SB E3 = (SA SB)

§ 23 А.2  K∝ SA SB E1 SB E2 SA E3 = ¬ SA SB
```

Это решение обладает тем недостатком, о котором мы говорили выше: каждый раз при использовании предложения § 23.2 рефал-транслятор будет заново формировать в поле зрения список из 32 символов, а на следующем шаге - при использовании предложений § 23 А.I или § 23 А.2 - будет его уничтожать.

Решение с использованием процедур 'ЗК' и 'ВК' таково. В начале работы необходимо выполнить конкретизацию:

```
K 'ЗК' АЛФАВИТ = АБВГ    эюя
```

Функция 'ПОРЯДОК' описывается предложениями

```
§ 24.I  K 'ПОРЯДОК' S A S A = (S A S A)
```

```
§ 24.2 К 'Порядок' Е1 = Кα Е1 К 'ЗК' АЛФАВИТ .
§ 24 А.1 К α S A S B Е1 S A Е2 S B Е3 = (S A S B)
                К 'ЗК' АЛФАВИТ = Е1 S A Е2 S B Е3 .
§ 24 А.2 К α S A S B Е1 S B Е2 S A Е3 = ¬ S A S B
                К 'ЗК' АЛФАВИТ = Е1 S B Е2 S A Е3 .
```

Заметим, что хотя правые части в предложениях § 24 А записываются довольно длинно, время, требуемое для замены, невелико. Ведь комбинация пяти переменных Е1 S A Е2 S B Е3 входит в правую часть точно так же, как и в левую, поэтому никакой перешивки здесь не требуется; алфавит закапывается целиком и без просмотров.

Задача 4.7. Записать предложение § 24 А.2 короче, не нарушая алгоритма.

6. Рефал-предикаты

Предикаты - это функции, принимающие значения из множества, содержащего всего два элемента, которые называются истинностными значениями и изображаются словами ИСТИНА и ЛОЖЬ или буквами И и Л, F и T, и т.п. Мы примем в качестве истинностных значений составные символы 'ДА' и 'НЕТ', так как они и короче, и выразительнее, чем 'ИСТИНА' и 'ЛОЖЬ' Предикаты широко используются в языках программирования как средство для создания разветвлений в алгоритме. С помощью предикатов формируются условные операторы и выражения. На рефале нетрудно описать условное выражение в том классическом виде, в котором оно используется например, в алголе.

§ 25.1 Условное выражение. Первые два предложения работают тогда, когда уже выполнена конкретизация предиката до истинностного значения.

К 'ЕСЛИ' ('ДА') 'ТО' (Е1) 'ИНАЧЕ' Е2 = Е1

§ 25.2 К 'ЕСЛИ' ('НЕТ') 'ТО' (Е1) 'ИНАЧЕ' Е2 = Е2

§ 25.3 Это предложение приводит к конкретизации предиката, если программист забыл или не смог заключить предикат в функциональные скобки. Его можно истолковать так: чтобы конкретизировать условное выражение, сначала конкретизируют предикат.

К 'ЕСЛИ' (Е Р) Е1 = К 'ЕСЛИ' (К Е Р .) Е1

Понятие оператора не является с точки зрения рефала основным, это лишь определенный способ использования или функционирования выражений - определенная семантика выражений. Поэтому то, что в языках программирования называют условными операторами, может быть интерпретировано в рефале через условные выражения.

Нетрудно описать на рефале и логические связки, например:

§ 26.1 К 'НЕ' 'ДА' = НЕТ'

§ 26.2 К 'НЕ' 'НЕТ' = 'ДА'

§ 26.3 К 'НЕ' Е1 = К 'НЕ' К Е1

Задача 4.8. Описать логические связки 'И' и 'ИЛИ' в формате (Е1)(Е2) , соблюдая тот же принцип, что и выше: предложения должны работать как в том случае, когда аргументы записываются с явным введением знаков К, так и в том случае, когда знаки К отсутствуют.

При такой системе записи логических выражений понятие старшинства связок отсутствует, поэтому она требует слишком много скобок. Можно было бы слегка модифицировать систему, введя старшинство связок, и тогда мы научим машину воспринимать логические выражения в обычной форме.

Однако эта система - мы имеем в виду не столько систему записи логических выражений, сколько систему использования предикатов в условных выражениях, интерпретируемых согласно § 25 - имеет другой, гораздо более серьезный недостаток.

Допустим, что у нас определено некоторое отношение (двуместный предикат) следования, которое мы записываем в формате

$$К\,'\text{СЛЕДУЕТ}'\,(\underline{Е}1)\,'\text{ЗА}'\,\underline{Е}2\,.$$

Это отношение широко используется при проведении аналитических выкладок для упорядочения выражений относительно коммутативных операций. Аргументы здесь могут быть весьма громоздки.

И пусть надо определить процедуру упорядочения двух выражений, используя это отношение таким образом, чтобы второе выражение "следовало" за первым. Мы записываем:

§ 27 $$К\,'\text{УПОРЯД}'\,(\underline{Е}1)(\underline{Е}2)\,\supseteq$$

$$К\,'\text{ЕСЛИ}'\,(\text{'СЛЕДУЕТ'}\,(\underline{Е}2)\,'\text{ЗА}'\,\underline{Е}1)$$

$$'\text{ТО}'\,((\underline{Е}1)(\underline{Е}2))\,'\text{ИНАЧЕ}'\,(\underline{Е}2)(\underline{Е}1)$$

Это описание приятно для глаза (ибо копирует формы естественного языка) и оно будет правильно работать. Его недостаток - что оно приводит к неэффективному использованию машины, ибо требует размножения переменных, которое совершенно бессмысленно с точки зрения

алгоритма. Каждая из переменных $\underline{E}1$ и $\underline{E}2$ левой части входит в правую часть трижды и следовательно воспроизводится дважды, причем без всякой необходимости, ибо обе копии тут же уничтожаются: первая копия уничтожается при конкретизации предиката (когда аргумент исчезает и остается только истинностное значение), вторая копия - при конкретизации условного выражения в соответствии с §§ 25.

Не следует думать, что эта неэффективность отражает какой-то недостаток языка рефал. Скорее наоборот: описав условное выражение на рефале, мы увидели воочию тот источник неэффективности, который коренится в самом принципе использования классических предикатов, однако приводит к неприятностям только в случае операций с громоздкими аргументами. Действительно, когда мы описываем алгоритмы на обычном операторном языке, например, на языке вычислительной машины, мы не думаем об аргументах как о физических объектах, которые переносятся, воспроизводятся, уничтожаются и т.п. Все эти заботы мы оставляем на долю тех частей системы, которые отвечают за реализацию языка. Как же будет происходить реализация условного оператора, подобного рассмотренному выше, если бы он был написан на машинном языке? Пусть упорядочиваемые объекты - числа, которые хранятся в ячейках оперативной памяти, а отношение следования - это отношение больше-меньше. Чтобы сравнить два числа машина переписывает в специальные регистры сначала одно из них, а затем другое. Это и есть первое размножение. Затем машина производит вычитание, в результате чего вырабатывается истинностное значение предиката, например, сигнал ω а обе копии аргументов уничтожаются. В зависимости от значения предиката управление передается в ту или

иную ячейку, и выполняется программа переписи чисел в нужном порядке. Это - второе размножение.

По сравнению с вычислительной машиной абстрактная рефал-машина чрезвычайно проста. Правда, она обладает важной способностью распознавания типовых ситуаций (синтаксическое отождествление), но в отношении преобразования объектов её способности ограничены элементарными подстановками, которые все на виду у программиста, все должны быть явно описаны. Всякие неявные переписи отсутствуют. Свободные переменные в предложениях - не абстрактные символы, а по существу, физические объекты, которые только и умеет перемещать рефал-машина. Поэтому наивно описав семантику условных выражений, мы и получили в явном виде те снятия копий, которые в ней подразумевались. Чтобы описать на рефале алгоритм конкретизации условных выражений, не приводящий к лишним действиям, надо специально проследить за перемещениями объектов в поле зрения рефал-машины. Надо учесть, что если какая-то функция уничтожает свободную переменную, которая еще понадобится,то её обязательно придется размножать во всех тех предложениях, которые эту функцию используют. Коротко говоря, чтобы не размножать, не надо уничтожать.

В первую очередь приложим этот принцип к предикатам. Вместо классических предикатов, которые аргумент заменяют на истинностное значение введем рефал-предикаты, которые аргумент сохраняют и только приписывают к нему истинностное значение или какие-либо иные знаки, выполняющие функцию истинностных значений. Например, можно определить функцию 'СЛЕДУЕТ' таким образом, что конкретизация

К 'СЛЕДУЕТ' (Е1) ЗА Е2 .

даст либо

'ДА' (Е1) Е2

либо

'НЕТ' (Е1) Е2

Другой способ определения рефал-предикатов - заключать аргумент в скобки, когда он удовлетворяет условию (эквивалент истинностного значения 'ДА') и приписывать спереди знак отрицания ¬ , когда он не удовлетворяет условию (эквивалент 'НЕТ'). В этом случае приведенное выше обращение к функции 'СЛЕДУЕТ' даст результат

((Е1) Е2)

или

¬ (Е1) Е2

соответственно. Определенные так рефал-предикаты мы будем называть стандартными. Они имеют то преимущество, что могут одновременно служить для образования рекурсивных переменных (см. выпуск 4). Мы будем пользоваться преимущественно стандартными предикатами. В то же время мы подчеркиваем, что использование той или иной формы рефал-предикатов - это вопрос, решение которого целиком зависит от программиста, ибо он пишет как предложения, определяющие предикат, так и предложения где он используется. Программист может, например, если это ему покажется удобным, определить отношение 'СЛЕДУЕТ' таким образом, что результатом конкретизации будет либо

'СЛЕДУЕТ' (Е1) 'ЗА' Е2

либо

'ПРЕДШЕСТВУЕТ' (Е1) Е2

Предложения, использующие подобным образом определенные предикаты, приобретают сходство с предложениями естественного языка, что имеет свои преимущества.

Второй источник размножения переменных при использовании классического условного выражения (§ 25) - наличие в левой части двух вариантов результата подстановки (после 'ТО' и после 'ИНАЧЕ'), из которых в правую часть переходит только один. Уничтожение свободной переменной в этих предложениях порождает размножение при их использовании. Чтобы устранить эту особенность надо разнести два варианта подставляемого выражения по разным предложениям: одно предложение будет описывать случай, когда предикат дал значение 'ДА' , другое - значение 'НЕТ' . Процедура упорядочивания, рассчитанная на стандартный рефал-предикат 'СЛЕДУЕТ' будет описываться двумя предложениями:

§ 28.1 К 'УПОРЯД' ((Е1)Е2) = (Е2)(Е1)

§ 28.2 К 'УПОРЯД' ¬(Е1)Е2 = (Е1)(Е2)

Такое описание предполагает, что при обращении к функции 'УПОРЯД' аргумент уже обработан функцией 'СЛЕДУЕТ', то-есть для того, чтобы действительно упорядочить пару выражений $\langle e_1 \rangle$ и $\langle e_2 \rangle$, надо выполнить конкретизацию

К 'УПОРЯД' К 'СЛЕДУЕТ' ($\langle e_1 \rangle$) 'ЗА' $\langle e_2 \rangle$ . .

Это экономно, но не всегда удобно, поэтому функцию 'УПОРЯД' определенную предложениями § 28, лучше считать вспомогательной функцией, а основную функцию (которую мы назовем 'УПОР' чтобы не переименовывать функцию § 28) определить предложением:

§ 29. К 'УПОР' (Е1) Е2 =

К 'УПОРЯД' К 'СЛЕДУЕТ' (Е1) 'ЗА' Е2 .

Можно обойтись и без введения вспомогательной функции, если ставить вслед за детерминативом метку 'УСЛ' - условное выражение - означающую, что конкретизация предиката уже произведена и осталось только выбрать один из двух вариантов. В описании функции эти два предложения должны стоять обязательно в начале, перед предложением, отсылающим к конкретизации предиката, иначе при некоторых форматах аргумента может произойти зацикливание. Формально это вытекает из того, что первые два предложения описывают более частный случай (аргумент начинается с метки 'УСЛ'), чем третье.

§ 30. I К 'УПОР' 'УСЛ' ((Е1)Е2) = (Е2)(Е1)

§ 30.2 К 'УПОР' 'УСЛ' ¬ (Е1)Е2 = (Е1)(Е2)

§ 30.3 К 'УПОР' (Е1)Е2 =

К 'УПОР' 'УСЛ' К 'СЛЕДУЕТ' (Е1) 'ЗА' Е2 . .

Это описание по своей структуре и по приемам программирования, отразившимся в нем, ничем не отличается от примеров, рассмотренных нами выше. Разветвление алгоритмического процесса осуществляется путем исследования аргумента на предмет вхождения в него в определенных местах определенных символов ('УСЛ' , ¬). Но наличие в группе предложений более чем одного предложения всегда порождает разветвление процесса, регулируемое синтаксическим анализом аргумента. Поэтому в рефале предикаты теряют то исключительное положение, которое они имеют в операторных языках как единственное средство управления ветвлением процесса. Специфика рефал-предикатов

по сравнению с другими функциями состоит в том, что они не преобразуют аргумент, а только анализируют его, или, если говорить строго формально, преобразуют его таким образом, что приписывают к нему те или иные синтаксические указатели ('ДА' , 'НЕТ', ¬ , скобки и т.п.) в зависимости от результата анализа. Разделение функций на те, которые только анализируют и те, которые только преобразовывают, принятое в операторных языках, при программировании на рефале оказывается чаще всего ненужным, ибо в описание каждой функции можно включить синтаксический анализ аргумента. Обычно рефал-функция, преобразуя аргумент, оставляет тем самым информацию в виде простых синтаксических признаков для следующей функции, которая будет использовать эти признаки для управления ветвлением. Это делает функции более емкими, а описание - более компактным В тех случаях, когда результат преобразования не обладает сам по себе такими признаками, можно создать их искусственно, приписывая тот или иной синтаксический указатель и объединив,опять-таки,предикат и преобразователь в одной функции. Например, для процедуры упорядочения многих термов можно в качестве основы принять не предикат следования для двух термов, а процедуру упорядочивания двух термов. Если мы ожидаем, что нам понадобится информация о том, были ли первоначально термы в нормальном порядке, или их пришлось переставить, эта процедура должна быть определена так, чтобы давать результат

'НОРМ' W1 W2

в первом случае, и результат

'ПЕРЕСТ' W2 W1

- во втором.

Итак, при программировании на рефале предикаты обычно растворяются в общей массе функций, за исключением тех случаев, когда они используются для создания рекурсивных переменных. Однако программист, привыкший к программированию в обычном операторном стиле, или имеющий уже готовые алгоритмы, описанные таким образом, может пожелать пользоваться условными выражениями, но так, чтобы это не приводило к ненужному размножению объектов. Мы увидим в выпуске что этого можно достичь с помощью определенных мета-процедур.

Задачи

4.9. Описать стандартный предикат: выражение содержит не менее трех термов.

4.10. Описать стандартный предикат 'СИММ' симметричности цепочки символов. Определение: пустая цепочка и цепочка из одного символа симметричны; цепочка симметрична, если она может быть получена приписыванием слева и справа одного и того же символа к симметричной цепочке.

4.11. Опираясь на предикат 'ПОРЯДОК' § 23, описать процедуру, ставящую данный символ на нужное место в данном упорядоченном списке символов. Формат:

```
§ К ВСТАВ' (Е⊔)§А =
```

где Е⊔ - список, §А - символ.

7. Сквозные просмотры

Чтобы описать сквозной просмотр выражения, надо в явном виде дать указание о вхождении в скобки.

Пусть процедура φ должна заменить все знаки + на знаки -, причем не только на верхнем уровне скобочной структуры, а с заходом в скобки. Простейшее решение задачи - это вставить в группу предложений § 8 описывающую простой просмотр, предложение, описывающее вхождение в скобки:

§ 31.1 $\underline{К}\ \varphi\ \underline{Е}1+\underline{Е}2 \geq \underline{К}\varphi\ \underline{Е}1.-\underline{К}\varphi\ \underline{Е}2.$

§ 31.2 $\underline{К}\ \varphi\ \underline{Е}1(\underline{Е}2)\underline{Е}3 \quad \underline{Е}1\ (\underline{К}\varphi\ \underline{Е}2.)\underline{К}\varphi\ \underline{Е}3.$

§ 31.3 $\underline{К}\ \varphi\ \underline{Е}1 \geq \underline{Е}1$

Здесь мы вынуждены были изменить также и первое предложение, заключив в функциональные скобки переменную $\underline{Е}1$ в правой части, ибо она должна быть еще просмотрена на предмет вхождения в скобки. Переменную $\underline{Е}1$ в правой части § 31.2 мы выносим из конкретизационных скобок, так как она уже не содержит ни знаков +, ни скобок.

При таком описании просмотр выражения выполняется не подряд, а сначала выражение разбивается на отрезки от плюса до плюса (верхнего скобочного уровня), и уже затем осуществляется вхождение в скобки. Этот порядок можно изменить, если переменить порядок предложений:

§ 32.1 $\underline{К}\ \varphi\ \underline{Е}1(\underline{Е}2)\underline{Е}3 \geq \underline{К}\varphi\ \underline{Е}1.\ (\underline{К}\varphi\ \underline{Е}2.)\underline{К}\varphi\ \underline{Е}3.$

§ 32.2 $\underline{К}\ \varphi\ \underline{Е}1+\underline{Е}2 \geq \underline{Е}1 \quad \underline{К}\varphi\ \underline{Е}2.$

§ 32.3 $\underline{К}\ \varphi\ \underline{Е}1 \geq \underline{Е}1$

Можно построить и такую процедуру, чтобы перебор символов производился в строгой последовательности слева направо. Для этого надо устранить открытые переменные и описать обработку каждого терма:

§ 33.1 K φ + E1 = K φ E1 .

§ 33.2 K φ (E1)E2 = (K φ E1 .) K φ E2 .

§ 33.3 K φ S1 E2 = S1 K φ E2 .

§ 33.4 K φ =

Задача 4.12. Распространить на все скобочные уровни процедуру 'ЛИКПР' , § 14, в двух вариантах: с открытыми переменными и без них.

Превращение простого просмотра в сквозной может иногда доставить хлопоты, связанные с необходимостью перехода с одного уровня скобочной структуры на другой. Скобочная структура - это структура дерева, и те операции, которые представляются естественными с точки зрения структуры дерева, описываются на рефале наглядно и компактно; операции же, которые тем или иным образом игнорируют структуру дерева, требуют для записи на рефале определенных усилий. Естественным с точки зрения структуры дерева является, прежде всего, движение по горизонтали, когда каждая ветвь предстает как единый объект. Это - простой просмотр, описываемый на рефале наиболее непринужденно. Естественным является также движение по вертикали, то-есть на одну ступеньку вниз по ветви дерева. Этот процесс описывается на рефале тоже непринужденно (вхождение в скобки), ибо встретив скобку, рефал-машина тут же находит и парную ей скобку, получая тем самым всё основание ветви в свое распоряжение для последующего анализа. Сквозной просмотр, когда он не предполагает передачи информации от ветви к ветви, также представляется достаточно естественным с точки зрения структуры дерева, ибо он разлагается на два независимых движения - по горизонтали и по вертикали. Сквозная замена плюсов на минусы относится как раз к этому типу. Рассматривая предложения

§ 31.2, § 32.I и § 33.2 мы видим, что появление скобки порождает два (в случае § 32.I - три) независимых процесса конкретизации: вход в скобки и продолжение просмотра по горизонтали.

Но если требуется обмен информацией между различными ветвями дерева, то задача усложняется, ибо обеспечение такого обмена не входит в задачи структуры дерева оно чуждо принципу таких структур. Рассмотрим следующий пример. Пусть надо из всех вхождений каждого символа оставить только самое первое (левое) вхождение, а все остальные уничтожить. Эта процедура фактически игнорирует скобочную структуру, рассматривает скобки как досадную помеху, и только. В то же время она требует полноценного обмена информацией между всеми подвыражениями на каких бы удаленных друг от друга ветвях они не находились.

Приступим к описанию нашей процедуры, которую обозначим через φ . Очевидно, на каждом этапе будет необходим список уже встреченных символов[x]. Поместим его в сумку, которую расположим в конце аргумента. Следовательно, необходимо ввести вспомогательную функцию α

§ 34* К φ Е1 ≥ К α Е1 ( ) .

Если бы не было скобок, функция α описывалась предложениями:

§ 34 ж.I К α ≤ A Е B (Е1 ≤ A Е2) ≥ К α Е B (Е1 ≤ A Е2) .

§ 34 ж.2 К α ≤ A Е B (Е L) ≥ ≤ A К α Е B (Е L ≤ A) .

§ 34 ж.3 К α W L ≥

---

[x] Просмотренная часть не может сама служить этим списком, так как не является выражением; ср. задачу 4.8.

(Просмотренную часть α оставляет вне функциональных скобок).

Что же должна делать функция α, встречая скобки? Попробуем, по аналогии с приведенным выше примером, написать:

$$\S\ \underline{K}\,\alpha\,(\underline{E}1)\,\underline{E}2\,\underline{W}\,\underline{L} \supseteq (\underline{K}\,\alpha\,\underline{E}1\,\underline{W}\,\underline{L}\,\underline{\cdot})\,\underline{K}\,\alpha\,\underline{E}2\,\underline{W}\,\underline{L}\,\underline{\cdot}$$

Легко видеть, что это предложение не обеспечивает правильной работы. В самом деле, когда отработает конкретизация, введенная в скобки, список $\underline{W}\,\underline{L}$ будет уничтожен в соответствии с предложением § 34 ж.3, и конкретизация функции α будет продолжаться со старым списком $\underline{W}\,\underline{L}$, не учитывающим символов, появившихся в скобках. Здесь мы и встречаемся с проблемой обмена информацией между различными ветвями дерева. Ее можно решить, "поднимая" информацию из той ветви, где она получена, на следующий уровень скобочной структуры. В нашем случае это означает, во-первых, что мы не должны уничтожать список $\underline{W}\,\underline{L}$ после конца просмотра. Для этого § 34ж.3 надо переписать следующим образом:

$$\S\ \underline{K}\,\alpha\,\underline{W}\,\underline{L} \supseteq \underline{W}\,\underline{L}$$

Во-вторых, войдя в скобки, мы должны оставить на верхнем уровне функциональные скобки, которые обеспечат извлечение информации из объектных скобок и продолжение работы с этой информацией. Обозначим через β вспомогательную функцию, выполняющую эту задачу. Итак,

$$\S\ \underline{K}\,\alpha\,(\underline{E}1)\,\underline{E}2\,\underline{W}\,\underline{L} \supseteq \underline{K}\,\beta\ \ \underline{K}\,\alpha\,\underline{E}1\,\underline{W}\,\underline{L}\,\underline{\cdot})\,\underline{E}2\,\underline{\cdot}$$

$$\S\ \underline{K}\,\beta\,(\underline{E}1\,\underline{W}\,\underline{L})\,\underline{E}2 \supseteq (\underline{E}1)\,\underline{K}\,\alpha\,\underline{E}2\,\underline{W}\,\underline{L}\,\underline{\cdot}$$

Учитывая, что функция α теперь оставляет в конце список $\underline{W}\,\underline{L}$ и для его уничтожения надо ввести еще одну вспомогательную функцию, получаем следующее описание:

§ 34 К φ Е1 ≥ К γ К α Е1 .

§ 34А.1. К α ≤ А Е В (Е1 ≤ А Е2) ≥ К α Е В (Е1 ≤ А Е2) .

§ 34А.2. К α ≤ А Е В (Е L) ≥ ≤ А К α Е В (Е L ≤ А) .

§ 34А.3 К α (Е1) Е2 W L ≥ К β (К α Е1 W L .) Е2 .

§ 34А.4 К α W L ≥ W L

§ 34 Б К β (Е1 W L) Е2 ≥ (Е1) К α Е2 W L .

§ 34 В К γ Е1 W L ≥ Е1

Как видим, оно получилось довольно громоздким. Другой способ передавать информацию, минуя скобочные структуры - это закапывая и выкапывая информацию при переходе из одной ветви дерева в другую. Чтобы применить этот способ к данному случаю, введем имя СПИС и определим функцию α так, чтобы она в конце каждого просмотра закапывала бы под этим именем терм W L содержащий список встречавшихся символов. Тогда при каждом обращении к α (за исключением первого) надо выкапывать этот список и ставить его на должное место в аргументе. В конце работы надо выкопать и уничтожить список, чтобы он не занимал понапрасну память. Получаем следующее описание:

§ 35. К φ Е1 ≥ К α Е1 ( ) . К 'УНИЧТ' К 'ВК' СПИС . .

§ 35А.1 совпадает с 34А.1

§ 35А.2 совпадает с 34А.2.

§ 35А.3 К α (Е1) Е2 W . . (К α Е1 W L)

К α Е2 К 'ВК' СПИС . .

§ 35А.4 К α WL ≥ К 'ЗК' спис = WL

§ 35 Б К 'уничт' Е1 ≥

Это описание содержит на одну вспомогательную функцию и на одно предложение меньше, чем предыдущее. Кроме того, передача информации с помощью команд 'ЗК' и 'ВК' - более универсальное средство. Мы отсоединяем на время информацию от аргумента, когда она не нужна, и снова присоединяем её, когда она становится нужной. Путем такого отсоединения и присоединения мы можем, в частности, миновать любые скобочные преграды.

Наконец, последний - и самый радикальный - способ выполнения сквозных операций состоит в том, чтобы на время выполнения операции вообще ликвидировать скобочную структуру, заменив скобки объектными знаками L и R . Эта процедура, которую мы назовем разъединением скобок, может быть описана предложениями:

§ 36.1 К 'разск' Е1 (Е2) Е3 ≥ Е1 L К 'разск' Е2 R Е3 .

§ 36.2 К 'разск' Е1 ≥ Е1

Легко видеть, что эта процедура требует однократного просмотра выражения. Противоположная процедура - спаривание скобок с детерминативом 'СПАРСК' превращает символы L и R в должным образом связанные скобки. Она несколько сложнее, чем разъединение скобок, и будет списана в конце настоящей главы. Однако и она требует лишь однократного просмотра выражения.

Процедуру φ , требующую сквозного просмотра, можно описать таким образом:

§ 37. К ψ Е1 = К 'СПАРСК' К ψ К 'РАЗСК' Е1 . . .

где ψ - процедура, рассчитанная на цепочку символов. Эта процедура, конечно, может совершать какие-то действия и над скобками, учитывая, что они предстают для нее в виде объектных знаков L и R На нее накладывается только то требование, чтобы в результате ее работы символы L и R могли снова быть превращены в скобки с соблюдением правильного синтаксиса.

Хотя такой способ требует двукратного дополнительного просмотра, он часто, а именно в случае сложных сквозных просмотров, оказывается наиболее экономичным, благодаря упрощению процесса обработки. И, конечно, он самый простой для программиста, особенно, если учесть, что процедуры 'РАЗСК' и 'СПАРСК' описаны раз и навсегда. Заметим, в заключение, что иногда может оказаться полезным провести специальную предварительную обработку объекта, которая некоторые пары скобок превращает в объектные знаки, в то время как другие оставляет нетронутыми.

8. Разделение алгоритма на функции

Невозможно дать какие-либо формальные предписания относительно того, как выбирать вспомогательные функции для конструирования функции с требуемыми свойствами; это видно из того, что вспомогательные функции могут быть выбраны по-разному, при сохранении благопристойного вида описания. Однако, можно указать несколько оснований или поводов для введения новой вспомогательной функции, иначе говоря несколько задач, для решения которых обычно требуется завести вспомогательную функцию. Этот перечень может служить руководством при разделении алгоритма, который надо описать на рефале,

на отдельные процедуры (функции).

Обычные основания для введения новой функции таковы.

I. Членение объекта согласно определенной схеме, задаваемой типовым выражением в левой части предложения. Например, функция γ, § 34В, отделяет последний терм $\underline{W}\,L$, чтобы его уничтожить.

2. Создание разветвления путем описания различных частных случаев отдельными предложениями с одним и тем же детерминативом. Классическим примером служат функции, использующие рефал-предикаты, такие как 'УПОРЯД', § 28, которая является вспомогательной функцией для функции 'УПОР', § 29.

3. Изменение формата, диктуемое, как правило, необходимостью включить в рекурсивный процесс новый объект (аргумент), или несколько объектов. Иллюстрацией может служить переход от функции φ к функции ψ в предложениях § 22 и § 22А, а также от φ к α (§ 34 и § 34 А) и др.

4. Предварительная обработка аргумента для эффективной работы основной функции или нескольких функций. В качестве примера укажем на § 37.

Отметим еще ряд особенностей системы функций, создаваемой при программировании на рефале.

Все функции можно разделить на три категории: глагольные, именные и глагольно-именные функции. Глагольной мы называем такую функцию φ, что конкретизация выражения

$$\leq \varphi < \mathcal{E} > \doteq$$

где $< \mathcal{E} >$ – произвольное объектное выражение, порождает в конце концов (после выполнения некоторого числа шагов) пустое выражение в поле зрения, но в процессе выполнения конкретизации происходит

изменение состояния поля памяти или внешних сред (последнее - с помощью машинных процедур). Примерами глагольных функций являются процедура 'ЗК' ,которую можно трактовать как описанную на рефале функцию, и машинная процедура вывода 'ВЫВ' .Само выражение,которое в результате предельной конкретизации дает пусто назовем глагольным выражением,или просто глаголом. Приведенное выше выражение является глаголом. Еще пример глагола:

К 'ЗК' A = К φ 123 . . К 'ВЫВ' ␣␣ . К 'ВПТ' К 'ВК' A . .

где функция 'ВПТ' описана предложениями:

§ 38.I Вывод по термам

К 'ВПТ' W1 E2 = К 'ВЫВ' W1 . К 'ВПТ' E2 .

§ 38.2 К 'ВПТ' =

В противоположность глагольному выражению, мы называем именным выражением или именем существительным такое выражение, которое будучи помещено в поле зрения рефал-машины вызывает серию конкретизаций, после выполнения которой не меняется состояние ни поля памяти, ни внешних сред. Функцию φ , такую, что

К φ < E > .

есть именное выражение при любом объектном выражении < E > , мы называем именной функцией. Пример имени существительного (функция 'ВЫДЦАZ' определена предложениями § I8):

К 'ВЫДЦАZ' A B C A B C X Y Z X Y Z .

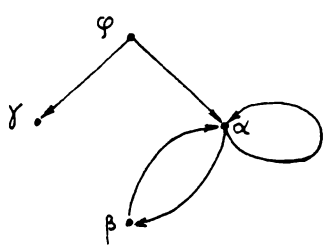
Наконец, глагольно-именной функцией назовем функцию, которая не является ни чисто глагольной, ни чисто именной. Выражение в общем случае является глагольно-именным выражением.

Если в правую часть предложения входит подвыражение вида

К φ <ℰ>

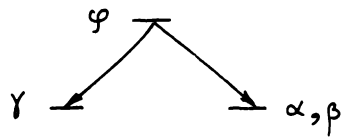
где <ℰ> любое рабочее выражение (содержащее, возможно, знаки конкретизации), то будем говорить, что данное предложение вызывает функцию φ Если хотя бы одно предложение с детерминативом α вызывает функцию β , будем говорить, что функция α вызывает функцию β Не всякое вхождение знака К в правую часть предложения является вызывом определенной функции. За знаком К может следовать свободная переменная символа или выражения, тогда вызываемая функция определится только динамически, в процессе выполнения программы. Такие ситуации мы рассмотрим в главе о мета-процедурах, а пока предположим, что они отсутствуют, так что с каждой функцией можно связать список (возможно, пустой) функций, которые она вызывает, и никаких других функций, кроме входящих в этот список, она вызвать не может.

Отношения вызова между функциями могут быть изображены в виде ориентированного графа, где вершины соответствуют функциям и помечаются их детерминативами, а вызов функцией α функции β изображается дугой (стрелкой), направленной от α к β . Например, система функций, описанная предложениями §§ 34, 34А, 34Б, 34В, образует граф, представленный на Фиг. 4.I.



Фиг. 4.I. Граф вызовов системы функций.

Петли на таком графе наглядно изображают рекурсию. На рисунке 4.I мы видим две петли: одна образуется функцией α, которая вызывает сама себя, другая - той же функцией вместе с функцией β Граф вызовов можно превратить в граф без петель, изображающий иерархию функций, если в одну вершину объединить все петли, имеющие хотя бы одну общую точку (см. Фиг. 4.2). Оба типа графов весьма полезны, когда надо разобраться в сложном алгоритме.

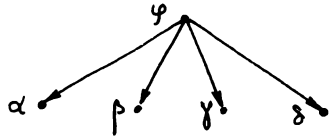


Фиг. 4.2 Иерархия функций.

Если некоторый объект надо подвергнуть последовательной обработке двумя или бо́льшим числом функций α, β, γ ... и т.д., то это можно осуществить двумя способами. Во-первых, можно определить функцию φ, которая к аргументу последовательно применяет независимо друг от друга описанные функции α, β, γ ..., например:

$$\S\, \underline{K}\, \varphi\, \underline{\in} 1 \supseteq \underline{K}\, \delta\, \underline{K}\, \gamma\, \underline{K}\, \beta\, \underline{K}\, \alpha\, \underline{\in} 1 \,.\,.\, .$$

В более сложных случаях в правую часть могут входить и другие объекты: пустые сумки и т.п. Такой способ порождает граф вызовов, изображенный на фигуре 4.3, поэтому мы будем называть его



Фиг. 4.3 Горизонтальная последовательность функций.

горизонтальным соединением функций в последовательность.

Во-вторых, можно определить функцию α таким образом, чтобы окончив работу, она вызывала функцию β, передав ей свое значение в качестве аргумента. Аналогично, β вызывает γ и т.д. Соответствующий граф вызовов имеет вертикальное строение (см. Фиг.4.4),

Фиг. 4.4 Вертикальная последовательность функций.

поэтому назовем такое соединение вертикальным. Достоинством горизонтального соединения является независимое определение функций. С другой стороны, вертикальное соединение часто оказывается удобнее, особенно, когда при передаче аргумента его надо разбивать на части.

Пусть две функции - α и β - заданы наборами из n и из m предложений:

$$\S\ \underline{К}\ \alpha\langle \mathcal{E}_{a1}\rangle \supseteq$$
$$\dots$$
$$\S\ \underline{К}\ \alpha\langle \mathcal{E}_{an}\rangle \supseteq$$
$$\S\ \underline{К}\ \beta\langle \mathcal{E}_{\beta 1}\rangle \supseteq$$
$$\S\ \underline{К}\ \beta\langle \mathcal{E}_{\beta m}\rangle \supseteq$$

Эти две функции можно слить в одну функцию φ, определив её $n+m$ предложениями

§ К φα < $\mathfrak{E}_{a1}$ > ≥

§ К φα < $\mathfrak{E}_{an}$ > ≥

§ К φβ < $\mathfrak{E}_{b1}$ > ≥

§ К φβ < $\mathfrak{E}_{bm}$ > ≥

где в правых частях вызовы функций α и β К α < $\mathfrak{E}$ > . и К β < $\mathfrak{E}$ > . , заменены на К φα < $\mathfrak{E}$ > . и К φβ < $\mathfrak{E}$ > . , соответственно.

Слияние функций - процедура чисто формальная и для практики бесполезная. Она показывает только, что теоретический минимум числа функций, которые нужно ввести для описания произвольного алгоритма, равен единице. Противоположная процедура - расщепление функции, описанной некоторым числом предложений, на ряд функций, каждая из которых требует меньшего числа предложений для своего описания - оказывается полезной на практике для оптимизации выполнения алгоритма. Каково минимальное число предложений, необходимое для описания любой функции? Ясно, что если каждая из используемых функций описывается одним предложением, то с их помощью можно описать лишь алгорим, не содержащий разветвлений. Но двух предложений для каждой функции уже достаточно, и мы это сейчас покажем. Более того, одно из этих предложений всегда может иметь простую стандартную форму.

Назовем однопробной такую функцию φ , которая описывается не более чем двумя предложениями, причем второе предложение, если оно присутствует, имеет в качестве аргумента свободную пере-

менную выражения:

§ ... К φ Е1 ≥

Фактически, однопробная функция определяется первым, основным, предложением. Второе предложение утверждает только, что если основное предложение не подошло, надо с аргументом, рассматриваемым как целое, сделать то-то и то-то: обычно, передать его другой функции или выдать в качестве результата.

Покажем, что каждую функцию можно заменить на вертикальную последовательность однопробных функций. Пусть функция φ описывается n предложениями:

§ X.1 К φ < $\mathcal{E}_1$ > ≥ < $\mathcal{E}_1'$ >

§ X.2 К φ < $\mathcal{E}_2$ > ≥ < $\mathcal{E}_2'$ >

§ X.n К φ < $\mathcal{E}_n$ > ≥ < $\mathcal{E}_n'$ >

Введем n -1 новых детерминативов: α_2 , α_3 , α_n , и определим n однопробных функций следующими наборами предложений:

§ XI.1 К φ < $\mathcal{E}_1$ > = < $\mathcal{E}_1'$ >

§ XI.2 К φ Е1 ≥ К α_2 Е1 .

§ X2.1 К α_2 < $\mathcal{E}_2$ > ≥ < $\mathcal{E}_2$ >

§ X2.2 К α_2 Е1 ≥ К α_3 Е1 .

§ Xn.1 К α_n < $\mathcal{E}_n$ > ≥ < $\mathcal{E}_n'$ >

Легко видеть, что новая функция φ эквивалентна старой. Сначала будет проектироваться на аргумент левая часть $\langle \mathcal{E}_1 \rangle$. Если она не подходит, вызывается функция α_2, которая пробует левую часть $\langle \mathcal{E}_2 \rangle$ и т.д. Когда обнаруживается подходящая левая часть, производится такая же замена, как и при старом описании φ. Последняя функция α_n описывается одним предложением. Если оно не подходит, отождествление невозможно как при старом, так и при новом описании.

При расщеплении функции на однопробные можно путем модификации правых частей добиться значительной оптимизации процесса, устранения лишних просмотров. Рассмотрим, например, процедуру сквозной замены плюсов на минусы определенную набором предложений § 3I. Расщепляем её, как описано выше:

§ 39.I $\underline{К}\ \varphi\ \underline{Е}1 + \underline{Е}2 \geq \underline{К}\ \varphi\ \underline{Е}1\ .\ -\ \underline{К}\ \varphi\ \underline{Е}2\ .$

§ 39.2 $\underline{К}\ \varphi\ \underline{Е}1 \geq \underline{К}\ \psi\ \underline{Е}1\ .$

§ 40.I $\underline{К}\ \psi\ \underline{Е}1\ (\underline{Е}2)\ \underline{Е}3 \geq \underline{Е}1\ (\underline{К}\ \varphi\ \underline{Е}2\ .)\ \underline{К}\ \varphi\ \underline{Е}3\ .$

§ 40.2 $\underline{К}\ \psi\ \underline{Е}1 \geq \underline{К}\ \chi\ \underline{Е}1\ .$

§ 4I.I $\underline{К}\ \chi\ \underline{Е}1 \geq \underline{Е}1$

Прежде всего, можно ликвидировать функцию χ, заменив в соответствии с § 4I.I в правой части предложения § 40.2 $\underline{К}\ \chi\ \underline{Е}1\ .$ на $\underline{Е}1$. Это следствие простого вида предложения § 3I.3. Но не в этом, конечно, состоит оптимизация. Рассмотрим правую часть § 39.1. Так как $\underline{Е}1$ заведомо не содержит знаков + на высшем уровне скобочной структуры, мы можем процедуру φ, обрабатывающую

$\underline{E}1$, заменить на ψ , избавив рефал-транслятор от необходимости снова просматривать содержимое $\underline{E}1$ в соответствии с § 39.I и не найдя знака + вызывать ψ в соответствии с § 39.2. По тем же причинам в правой части § 40.I заменяем $\underline{K}\ \psi\ \underline{E}3\ .$ на $\underline{K}\ \psi\ \underline{E}3\ .$ Теперь получаем описание:

§ 42.I Замена + на - на высшем уровне скобочной структуры и передача управления на вхождение в скобки

$$\underline{K}\ \varphi\ \underline{E}1 + \underline{E}2 \geq \underline{K}\ \psi\ \underline{E}1\ .\quad \underline{K}\ \varphi\ \underline{E}2\ .$$

§ 42.2 $\underline{K}\ \varphi\ \underline{E}1 \geq \underline{K}\ \psi\ \underline{E}1\ .$

§ 42A.I Внесение в скобки процедуры φ

$$\underline{K}\ \psi\ \underline{E}1\ (\underline{E}2)\ \underline{E}3 \geq \underline{E}1\ (\underline{K}\ \varphi\ \underline{E}2\ .)\ \underline{K}\ \psi\ \underline{E}3\ .$$

§ 42A.2 $\underline{K}\ \psi\ \underline{E}1 \geq \underline{E}1$

Легко убедиться, что описанный так алгоритм больше не содержит лишних просмотров.

Задача 4.I3. Доказать, что любая функция может быть заменена на эквивалентную горизонтальную последовательность однопробных функций.

9. Прогонка

В правой части предложения § 40.2 мы заменили $\underline{K}\ f\ \underline{E}1\ .$ на $\underline{E}1$. Основанием для этой замены является то, что независимо от значения, которое принимает свободная переменная $\underline{E}1$ на следующем шаге после использования § 40.2 рефал-машина обязательно использует § 4I.I и произведет эту замену. Таким образом, мы

совершаем за рефал-машину часть её работы, заглядывая на один шаг вперед и соединяя два шага в один. Эта операция называется прогонной. Она представляет собой эквивалентное преобразование программы, которое производится с целью улучшить программу, ускорить её выполнение, а иногда и объем: например, в нашем случае § 41.1 становится в результате прогонки предложения § 40.2 ненужным для конкретизации функции φ и может быть выброшен из программы.

В каких случаях может производиться прогонка? Очевидно, если некая функция φ описывается одним единственным предложением, причем предложением вида

$$\S\ \underline{K}\ \varphi\ \underline{E}1\ \geq\ <\mathcal{E}>$$

то все предложения, содержащие в правой части вызов этой функции, допускают прогонку, после которой описание функции φ становится ненужным (подразумевается, что $<\mathcal{E}>$ не содержит вызова φ, иначе конкретизация φ никогда не закончится).

Но это только самый тривиальный случай. Рассмотрим более сложный пример. Пусть функция α определяется предложением:

$$\S\ 43\ \ \underline{K}\ \alpha\ \underline{E}1\ \geq\ \underline{K}\ \varphi\ (\underline{K}\ \beta\ \underline{E}1.)\ .$$

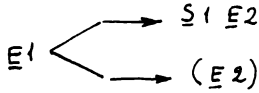
а функция β имеет описание

$$\S\ 43A.1\ \ \underline{K}\ \beta\ \underline{S}1\ \underline{E}2\ \geq\ \underline{E}2$$

$$\S\ 43A.2\ \ \underline{K}\ \beta\ (\underline{E}2)\ \geq\ \underline{E}2$$

Конкретизация функции β зависит от вида её аргумента: начинается ли он с символа, или со скобки. Чтобы прогнать на один шаг предложение § 43, мы должны расщепить его на два,

согласно тем двум частным случаям аргумента $\underline{\Sigma}1$, различать которые необходимо для конкретизации функции β



Чтобы изобразить процесс прогонки, будем отделять знаком $\trianglerighteq$ стадии преобразования правой части:

§ 43.I $\underline{K}\alpha\,\underline{\Sigma}1\,\underline{\Sigma}2 \trianglerighteq \underline{K}\varphi\,(\underline{K}\beta\,\underline{\Sigma}1\,\underline{\Sigma}2\,\underline{.})\,\underline{.} \trianglerighteq \underline{K}\varphi\,(\underline{\Sigma}2)\,\underline{.}$

§ 43.2 $\underline{K}\alpha\,(\underline{\Sigma}2) \trianglerighteq \underline{K}\varphi\,(\underline{K}\beta\,(\underline{\Sigma}2)\,\underline{.})\,\underline{.} \trianglerighteq \underline{K}\varphi\,(\underline{\Sigma}2)\,\underline{.}$

Итак, вместо трех предложений, необходимых для описания конкретизации функции α (не считая описания функции φ) - § 43, §§ 43А.I-2, получаем два предложения для α

§ 43.I $\underline{K}\alpha\,\underline{\Sigma}1\,\underline{\Sigma}2 \trianglerighteq \underline{K}\varphi\,\underline{\Sigma}2$

§ 43.2 $\underline{K}\alpha\,\underline{\Sigma}2 \trianglerighteq \underline{K}\varphi\,(\underline{\Sigma}2)\,\underline{.}$

а функция β оказывается исключенной.

Прогонка - важнейшее из формальных преобразований рефал-программ. В следующей главе мы дадим примеры его эффективного использования.

IO. Стиль программирования на рефале

Программирование на рефале заметно отличается по стилю работы от программирования на обычных операторных языках программирования. Оно ближе к деятельности математика классического образца. Как и математик, программист на рефале имеет дело прежде всего

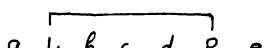
с формальными языковыми объектами, и лишь во вторую очередь - с действиями над ними. Приступая к программированию на рефале, надо начать с того, что четко определить систему формальных объектов, над которыми будут совершаться преобразования. И сам способ выбора объектов, и способ их записи на бумаге, должны ясно отображать особенности решаемой задачи, фиксировать понятия, присущие данной задаче. Если эта часть работы сделана хорошо, то действия над объектами удается описать в виде фундаментальных рекурсивных соотношений, ясных по форме и содержанию. Поэтому программирование на рефале требует гораздо большей работы по конструированию и исследованию языковых объектов, выяснению их свойств и отношений, чем это требуется в обычном программировании.

Обычное операторное программирование делает упор на действия над объектами, которые (объекты) сами по себе весьма незамысловаты. Динамика, временная связь событий приобретает решающее значение. Приходится следить за передачей управления на метку (в ячейку), за тем, когда было последний раз обращение к данному блоку и т.п. Это ставит перед человеческим мозгом трудные задачи, для решения которых он плохо приспособлен. Человеку легче мыслить в терминах статических соотношений между объектами, такой подход непосредственно опирается на способность мозга к ассоциации представлений, лежащую в основе мышления. Он создает больше возможностей расчленить сложную задачу на простые, построить иерархию понятий и объектов и охватить умственным взором задачу в целом. Это резко уменьшает число ошибок при программировании с помощью рекурсивных функций. Само понятие понятия (во всяком случае, логического, то-есть языкового понятия) будучи формализовано, превращается в понятие рекурсивной функции.

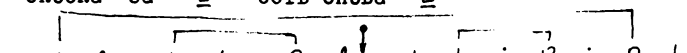
Чтобы проиллюстрировать метод работы на рефале, дадим подробный разбор процесса написания программы спаривания скобок. Пусть мы имеем последовательность символов, среди которых есть символы $\underline{L}$ и $\underline{R}$, и надо заменить эти символы на левую и правую скобки. Мы не можем просто просматривать аргумент, заменяя $\underline{L}$ на левую, а $\underline{R}$ - на правую скобку. Формально - с точки зрения абстрактного рефала - это невозможно потому, что требует предложений, правые части которых не являются выражениями. С точки зрения реализации рефала это невозможно потому, что превращая символ в скобку, мы должны в поле A_1 поместить адрес парной скобки. Итак, мы можем вставить только две скобки (...) сразу, причем это должны быть скобки, образующие пару в смысле скобочной структуры выражения. Следовательно, прежде чем производить замену, надо для символа $\underline{L}$ найти парный ему символ $\underline{R}$

Как подойти к этой задаче?

Начнем с исследования простейших примеров. Если между "скобкой" $\underline{L}$ и "скобкой" $\underline{R}$ нет никаких "скобок", то они, очевидно, образуют пару:



Следовательно, просматривая аргумент слева направо, и обнаружив, что следующая за $\underline{L}$ "скобка" есть $\underline{R}$, мы можем связать их, заменив их на пару скобок (без кавычек). Допустим теперь, что следующая "скобка" за $\underline{L}$ есть снова $\underline{L}$



Тогда мы откладываем спаривание первой скобки, и встретив $\underline{R}$, спариваем её со второй. Идя дальше и снова (после h) встречая

$\underline{L}$, мы ставим её на очередь и спариваем с первой же $\underline{R}$ И лишь встречая еще одну $\underline{R}$, мы спариваем её с первой $\underline{L}$, которая стала теперь очередной.

Таким образом, мы пропускаем все $\underline{L}$, а встретив $\underline{R}$, спариваем её с последней неспаренной $\underline{L}$. Просмотренная часть аргумента представляет из себя объект, который может быть назван левой мультискобкой. Он содержит некоторое число неспаренных символов $\underline{L}$, но не содержит $\underline{R}$ Спаренные $\underline{L}$ и $\underline{R}$ уже заменены на настоящие скобки. Например, когда стрелка просмотра находится в приведенном выше выражении между символами f и g, левая мультискобка есть

$$a\ \underline{L}\ b\ c\ (d\ e)\ f$$

Число символов $\underline{L}$ определяет глубину, на которой находится стрелка в скобочной структуре.

Теперь нетрудно описать процесс спаривания как действия над левой мультискобкой. Сначала сделаем это на нашем примере. Если за мультискобкой следует обычный символ или "скобка" $\underline{L}$, мы просто приписываем их к мультискобке:

$$a\ \underline{L}\ b\ c\ (d\ e)\ f \rightarrow a\ \underline{L}\ b\ c\ (d\ e)\ f\ g$$

$$a\ \underline{L}\ b\ c\ (d\ e)\ f\ g\ h \rightarrow a\ \underline{L}\ b\ c\ (d\ e)\ f\ g\ h\ \underline{L}$$

Встречая символ $\underline{R}$, мы спариваем его с последним в мультискобке знаком $\underline{L}$

$$a\ \underline{L}\ b\ c\ (d\ e)\ f\ g\ h\ \underline{L}\ i \rightarrow a\ \underline{L}\ b\ c\ (d\ e)\ f\ g\ h\ (i)$$

Теперь обсудим вопрос о синтаксисе объектов для написания рефал-программы. Мы могли бы хранить мультискобку в своем нату-

ральном виде, помещая её в сумку. Однако нетрудно видеть, что такой способ потребует многократного просмотра аргумента. Первый просмотр происходит, когда мы удлиняем мультискобку, и он, разумеется, неизбежен. Остальные просмотры производятся (справа налево) при поиске последнего в мультискобке символа ⊔ Как их избежать? Заключить все, что не надо просматривать - то-есть отрезки от ⊔ до ⊔ - в скобки! Подновлять эту структуру мы сможем во время первого просмотра.

Итак, мы приходим к необходимости несколько изменить определение мультискобки. Мультискобкой мы будем называть объект вида

$$(\langle \mathcal{E}_1 \rangle) \sqcup (\langle \mathcal{E}_2 \rangle) \sqcup \quad \sqcup (\langle \mathcal{E}_n \rangle)$$

где $\langle \mathcal{E}_1 \rangle \ldots \langle \mathcal{E}_n \rangle$ - выражения, не содержащие символов ⊔ и R . Приведенная выше мультискобка занишется теперь в виде

$$(a) \sqcup (bc(de)f)$$

Для единообразия записи мы потребуем, чтобы мультискобка всегда начиналась и кончалась термом $(\langle \mathcal{E}_i \rangle)$, хотя $\langle \mathcal{E}_i \rangle$ может быть и пустым. Число символов ⊔ может быть, в частности, равно нулю. Например пустая мультискобка имеет вид ( ).

Теперь можно приступить к описанию процедуры 'СПАРСК' на рефале. Первым делом необходимо форматное преобразование:

```
§ 44 К 'СПАРСК' Е1 = К 'СПАР1' (()) Е1 .
```

Процедура 'СПАРI' имеет одну сумку, содержащую мультискобку. Начальный вид мультискобки - пустая мультискобка.

Предложения, описывающие 'СПАРI' , получаются непосредственно из тех примеров преобразования мультискобок, которые мы

приводили выше. Надо только учесть новый формат мультискобки и обратить внимание на порядок предложений . Первые два случая совсем просты, третий - спаривание - сложнее, и чтобы не ошибиться, мы сначала выполним преобразование на примере. Если бы мультискобка записывалась в натуральном виде, преобразование аргумента функции 'СПАРI' можно было бы проиллюстрировать таким примером:

$$(a\,b \sqcup c\,d \sqcup e\,f)\ \underline{R}\ g\,h \rightarrow (a\,b \sqcup c\,d\,(e\,f))\ g\,h$$

Теперь преобразуем формат записи мультискобки в соответствии с определением данным выше:

$$((a\,b) \sqcup (c\,d) \sqcup (e\,f))\ \underline{R}\ g\,h \rightarrow$$

$$((a\,b) \sqcup (c\,d\,(e\,f)))\ g\,h$$

и заметим конкретные выражения на свободные переменные:

$$(\underline{\in} M\ (\underline{\in} Y) \sqcup (\underline{\in} Z))\ \underline{R}\ \underline{\in} 1 \rightarrow (\underline{\in} M\ (\underline{\in} Y\ (\underline{\in} Z)))\ \underline{\in} 1$$

Первый символ $\sqcup$ мы включили в свободную переменную $\underline{\in} M$: ведь его может и не быть ($\underline{\in} M$ - пусто). Для преобразования существенен лишь отрезок, примыкающий слева к последнему символу $\sqcup$.

Итак, получаем следующую программу:

§ 45.I $\underline{K}$ 'СПАР1' $(\underline{\in} M) \sqcup \underline{\in} 1 \equiv \underline{K}$ 'СПАР1' $(\underline{\in} M \sqcup ()) \underline{\in} 1$.

§ 45.2 $\underline{K}$ 'СПАР1' $(\underline{\in} M\ (\underline{\in} Y) \sqcup (\underline{\in} Z))\ \underline{R}\ \underline{\in} 1 \equiv$

$\underline{K}$ 'СПАР1' $(\underline{\in} M\ (\underline{\in} Y\ (\underline{\in} Z)))\ \underline{\in} 1$.

```
§ 45.3  K 'СПАР1' (EM (EZ)) SA E1 =
                K 'СПАР1' (EM (EZ SA)) E1 .
§ 45.4  K 'СПАР1' ((EM)) = EM
```

Последнее предложение требует, чтобы дойдя до конца исходного выражения, мы получили мультискобку, состоящую из одного отрезка, то-есть не содержащую символов L . Если это не так, отождествление окажется невозможным и рефал-машина испытает аварийную остановку.

Задача 4.14. Так модифицировать описание § 45, чтобы в случае, когда "скобки" L и R не образуют правильной скобочной структуры процесс обрывался и печаталось сообщение об ошибке.

С О Д Е Р Ж А Н И Е

№ Т-12615 от "22" VII 1971 г. Заказ № 732 Тираж 250 экз.

Ордена Ленина институт прикладной математики
Москва, Миусская пл., 4