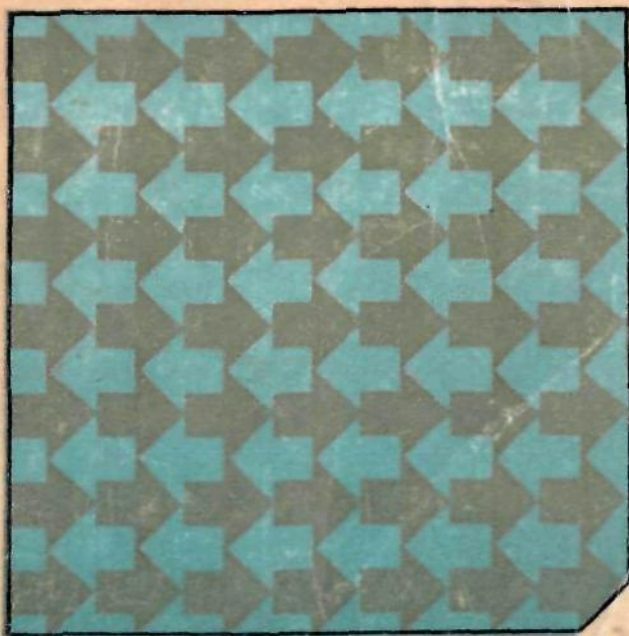


**МАТЕМАТИЧЕСКОЕ  
ОБЕСПЕЧЕНИЕ  
ЭВМ**

**Дж.Фостер**  
**ОБРАБОТКА  
СПИСКОВ**





COMPUTER MONOGRAPHS  
General Editor: Stanley Gill  
Associate Editor: J. J. Florentin

## LIST PROCESSING

J. M. FOSTER

Senior Research Fellow  
Department of Engineering  
Aberdeen University

MACDONALD : LONDON

1968

## **МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ЭВМ**

---

**Дж.Фостер**

## **ОБРАБОТКА СПИСКОВ**

Перевод с английского  
В. В. Мартынюка

Под редакцией  
Э. З. Любимского

ИЗДАТЕЛЬСТВО  
«МИР»  
Москва 1974

## ОТ ИЗДАТЕЛЬСТВА

Книга содержит систематическое описание методов обработки списков, необходимых каждому программисту при работе с задачами со сложными данными. В ней описывается применение методов обработки списков в процессе программирования, а также представление списков в машинной памяти. Проводится сравнение некоторых языков программирования, специально предназначенных для обработки списков.

Книга представляет несомненный интерес для широкого круга читателей, занимающихся программированием. Она доступна читателям с небольшими познаниями в программировании и будет полезна специалистам, работающим в области автоматизированных систем управления (АСУ).

*Редакция литературы по математическим наукам*

Дж. Фостер

## ОБРАБОТКА СПИСКОВ

Редактор Л. Бабынина. Художник В. Студиков. Художественный редактор В. Шаповалов.

Технический редактор Н. Малохина. Корректор О. Румянцева

Сдано в набор 6/II 1974 г. Подписано к печати 16/V 1974 г.

Бумага тип. № 3 60×90<sup>1</sup>/<sub>16</sub>. 2,25 бум. л. 4,5 печ. л. Уч.-изд. л. 3,58. Изд. № 1/727

Цена 26 коп. Зак. 75

ИЗДАТЕЛЬСТВО «МИР»

Москва, 1-й Рижский пер., 2

Ордена Трудового Красного Знамени Ленинградская типография № 2  
имени Евгения Соколовой Союзполиграфпрома при Государственном комитете  
Совета Министров СССР по делам издательств, полиграфии и книжной торговли  
198052, Ленинград, Л-52, Измайловский проспект, 29

Этой книгой издательство «Мир» начинает новую серию переводов «Математическое обеспечение ЭВМ». Ранее книги по этой тематике публиковались в серии «Библиотека «Кибернетического сборника». В 1972—73 гг. были опубликованы книги:

Дж. Айлиф, Принципы построения базовой машины,

Дж. Каттл, П. Робинсон (ред.), Супервизоры и операционные системы,

Ч. Линдси, С. ван дер Мюйлен, Неформальное введение в АЛГОЛ 68,

М. Уилкс, Системы с разделением времени,

Ф. Хопгуд, Методы компиляции.

Выделение новой серии вызвано возросшим интересом к этому важнейшему разделу современной прикладной математики. В ближайшее время в этой серии будут опубликованы следующие книги:

Д. Баррон, Ассемблеры и загрузчики,

Д. Баррон, Рекурсивные методы в программировании,

Л. Коддингтон, Ускоренный курс КОБОЛа,

Дж. Хенли, Автоматизированная библиотека и информационные системы,

Б. Хигман, Сравнительное изучение языков программирования.

Издательство надеется, что книги серии «Математическое обеспечение ЭВМ» окажутся полезными читателям.

Сильная сторона вычислительных машин состоит в их арифметических возможностях. Простые численные программы легко могут быть составлены из таких машинных команд, как сложение, вычитание, умножение или деление. Однако поиск чисел может оказаться затруднительным, поскольку запоминание данных в машине организовано специальным образом. Все хорошо, когда нужную информацию можно расположить таким образом, чтобы порядок ее использования соответствовал структуре памяти. Все слова в памяти нумеруются, и содержимое слова может использоваться или изменяться только путем ссылки на соответствующий номер (адрес слова). Машина способна вычислять номера; поэтому данные должны быть упорядочены так, чтобы облегчить вычисление последующих адресов. В случае усложненного порядка использования машинных слов поиск адресов может оказаться основной частью программы.

Если вычислительная машина должна, например, переводить с русского языка на английский, управлять системой дорожных светофоров, доказывать теоремы или составлять школьные расписания, то ей необходимо манипулировать данными со сложными взаимосвязями. Такие взаимосвязи могут означать необходимость сложной и запутанной организации обращений к словам в памяти. При обработке списков мы занимаемся таким моделированием объектов внутри машины, при котором становятся явными их взаимосвязи, что облегчает поиск соответствующих адресов. Термин «обработка списков» указывает на то, что основные идеи относятся к упорядочиванию наборов. Если группу элементов нужно просматривать в определенном порядке, то ее можно упорядочить. Один и тот же элемент может принадлежать нескольким упорядоченным наборам. Можно привести много различных примеров списков. Очередь является упорядоченным набором участников; предложение — упорядоченный набор слов; матрица — упорядоченный набор векторов, каждый из которых представляет собой упорядоченный набор чисел; программа — это упорядоченный набор действий; служащим фирмы соответствует упорядоченный набор фамилий, а инвентарный каталог фирмы является упорядоченным набором

элементов. В других случаях порядок внутри набора может не иметь значения. Чертеж является объединением линий; электрическая схема — это объединение наборов соединений; железнодорожная система представляет собой набор путей, которыми соединяются элементы набора станций. Эта идея наборов настолько фундаментальна, что соответствующие приемы использовались в программировании с самого начала. Обработка списков является не новым методом, а системой широко распространенных практических приемов.

Для обработки списков желательны несколько иные базисные операции, чем для арифметических расчетов. Предположим, что нужно представить в вычислительной машине железнодорожную систему, задав набор названий станций, причем каждому названию соответствует список станций, непосредственно связанных с данной, с указанием дополнительной информации о связях. В этом случае, по-видимому, были бы желательны такие операции, как «найти место хранения (в памяти) названия следующей станции», или «добавить новое число к данному списку», или «найти первый элемент данного списка». В принципе программист мог бы описать эти операции в машинном коде; в рабочих программах они именно так и будут представлены. Однако обычно программирование обработки списков выполняется с помощью компилятора. При этом программист пользуется языком, на котором легко описываются нужные действия, а компилятор переводит программу с этого языка в код машины.

Мы будем заниматься языками, которые явно предназначены для обработки списков. Другими словами, в них предусмотрены элементарные операции над списками. В книге не уделяется особого внимания языкам, в которых выполняется обработка списков, но элементарные операции спрятаны от пользователя, и он может оперировать, например, целыми фразами. Это объясняется не превосходством языков первого типа, а просто тем, что основное содержание данной книги сводится к описанию простейших форм обработки списков и их применения. Простейшие формы обработки списков являются в некотором смысле машинными командами обработки списков. Их применению сопутствуют некоторые преимущества и неудобства, присущие работе с машинным кодом.

Языки для обработки списков возникали двумя способами: либо они явно предназначались для работы со списками и включали более или менее удобные дополнительные средства описания численных действий, либо возникали на основе языков, предназначенных первоначально для численного анализа, к которым впоследствии подключались операции обработки списков. Второй способ имеет то преимущество, что такие языки

понятны широкому кругу пользователей и что на этих языках уже написано много программ численного анализа. Программы часто содержат одновременно как численные выкладки, так и обработку списков; поэтому возможность доступа к библиотеке имеющихся подпрограмм может оказаться существенным преимуществом. С другой стороны, применение специальных языков обработки списков может оказаться более удобным. В данной книге мы будем описывать программы обработки списков на АЛГОЛе (с некоторыми модификациями), за исключением тех случаев, когда будут специально рассматриваться другие языки обработки списков. Этот выбор обусловлен широкой известностью АЛГОЛа.

### ПРЕДПОСЫЛКИ ОБРАБОТКИ СПИСКОВ

Обработка списков приобретает значение тогда, когда простейшие методы использования памяти оказываются неприменимыми. В простых программах машинная память часто разделяется на отдельные части, соответствующие отдельным объектам: векторам, матрицам, таблицам декодирования, строкам символов и т. п. Размеры этих блоков данных известны заранее, и машинные адреса присваиваются им до начала выполнения программы. Этим часто можно удовлетвориться, однако могут возникнуть трудности. Типичным примером таких трудностей является необходимость чтения в память набора чисел, когда размер этого набора не известен до окончания чтения. При этом возникает вопрос: сколько памяти следует зарезервировать для данного вектора и как быть, если таких векторов несколько? Методы обработки списков возникали при решении такого рода проблем. Эти проблемы встречаются довольно часто при обработке нечисловой информации в реальном масштабе времени, когда размер памяти, необходимый программе для данных, может оставаться неизвестным до тех пор, пока программа не станет выполняться. Однако обработка списков — это не только удобный метод использования памяти. Методы, применяемые для распределения памяти, могут породить сложную структуру памяти и, таким образом, позволяют программисту решать проблему обработки и представления данных с нестандартной структурой. Следует учитывать, что эти две довольно самостоятельные области применения существуют независимо и используются программистами в различной степени в зависимости от конкретных потребностей.

Первоначально языки обработки списков разрабатывались для выполнения специальных исследований не численного характера. Ряд языков под общим названием IPL [21, 22, 26] возник как средство обеспечения удобных методов организации

информации для автоматического доказательства теорем и решения проблем. Для задач из этой области характерно, что количество нужной памяти значительно изменяется и не может быть заранее предсказано, а сама структура списков несет в себе важную информацию. Язык ЛИСП [16, 17, 18] был разработан отчасти для использования в рамках проекта «Прием советов» («Advice Taker»), предназначенного для того, чтобы управлять автоматической системой, сообщая ей информацию о сложных ситуациях в виде предложений на английском языке. Язык под названием FLPL [11], полученный расширением ФОРТРАНА, предназначался для записи программ доказательства геометрических теорем. Язык КОМИТ [7, 31] был разработан для лингвистических исследований.

Во всех этих случаях практическая потребность в сложной организации данных привела к появлению языков, которые оказались полезными для многих приложений. Задача этой книги состоит не в том, чтобы дать детальное описание какого-либо языка обработки списков. Такие описания уже имеются в литературе, и в первую очередь в руководствах по языкам программирования. Вместо этого мы будем рассматривать применяемые принципы и некоторые общие методы реализации таких языков. При этом мы будем пытаться выделить универсальные приемы из тех конкретных приложений, в которых они проявляются.

Объекты и отношения, с которыми производятся действия при обработке списков, представляются на двух уровнях. В вычислительной машине списки представляются в виде содержимого памяти, тогда как программист представляет их знаками на бумаге. Сделанная программистом запись может быть полным описанием содержимого памяти, или же она может выражать только те свойства, которые кажутся относящимися к делу. В этой главе мы опишем элементарные свойства представления списков в вычислительной машине, а также обозначения, которыми программист может пользоваться на бумаге.

### СПИСКИ В ВЫЧИСЛИТЕЛЬНОЙ МАШИНЕ

Рассмотрим программу, которая предназначена для того, чтобы прочесть предложение, затем упорядочить слова в алфавитном порядке и, наконец, выдать их в этом порядке на печать. Если предположить, что слова при чтении упаковываются в памяти как можно более плотно, например по шесть символов в ячейке, то прочтенное предложение может быть представлено следующим образом:

```
100 ЭТА + ЗИ
101 МА + БЫЛ
102 А + НА + Р
103 ЕДК ОСТ
104 Ъ + МАЛО
105 СНЕЖНО
106 И + ---
```

Буквы представляются малыми целыми числами, упакованными в ячейки, номера которых указаны слева. Пробелы между словами отмечаются знаком плюс. После упорядочивания этих слов они будут представлены следующим образом:

200	Б	Ы	Л	А	+	З
201	И	М	А	+	М	А
202	Л	О	С	Н	Е	Ж
203	Н	О	Й	+	Н	А
204	+	Р	Е	Д	К	О
205	С	Т	Ь	+	Э	Т
206	А	+	—	—	—	—

Очевидно, что, хотя нетрудно запрограммировать это преобразование, оно все же излишне громоздко. Было бы значительно удобнее упаковывать символы менее плотно и начинать каждое слово предложения с нового адреса. При этом гораздо легче получить упорядоченную последовательность слов, так как перемещаемые слова занимают теперь по одной или по несколько машинных ячеек, а не расщепляют ячейки.

100	Э	Т	А	+	—	—
101	З	И	М	А	+	—
102	Б	Ы	Л	А	+	—
103	Н	А	+	—	—	—
104	Р	Е	Д	К	О	С
105	Т	Ь	+	—	—	—
106	М	А	Л	О	С	Н
107	Е	Ж	Н	О	Й	+

Содержимое этих ячеек можно переупорядочить, получив следующий результат:

200	Б	Ы	Л	А	+	—
201	З	И	М	А	+	—
202	М	А	Л	О	С	Н
203	Е	Ж	Н	О	Й	+
204	Н	А	+	—	—	—
205	Р	Е	Д	К	О	С
206	Т	Ь	+	—	—	—
207	Э	Т	А	+	—	—

Программирование может быть облегчено, если к представлению предложения добавить вектор начальных адресов всех слов предложения.

300	100	100	Э	Т	А	+	—	—
301	101	101	З	И	М	А	+	—
302	102	102	Б	Ы	Л	А	+	—
303	103	103	Н	А	+	—	—	—
304	104	104	Р	Е	Д	К	О	С
305	106	105	Т	Ь	+	—	—	—
		106	М	А	Л	О	С	Н
		107	Е	Ж	Н	О	Й	+

Мы можем теперь получить аналогичный вектор начальных адресов всех слов в алфавитном порядке, вовсе не перемещая упакованных слов.

400	102
401	101
402	106
403	103
404	104
405	100

Это легко программируется. Очевидно, можно получить много различных векторов начальных адресов, представляющих различные порядки слов предложения, не изменяя содержимого ячеек, содержащих символы. К этой системе можно добавлять новые слова, не трогая имеющиеся слова или существующие векторы. Важная особенность этого довольно тривиального примера состоит в том, что имеется удобная возможность оперировать адресами величин, а не самими величинами, благодаря чему облегчается программирование и экономится память. Этот стандартный прием программирования является одной из основ обработки списков.

Другой простой пример применения адресных вычислений связан с матрицами. Рассмотрим находящуюся в памяти двумерную матрицу и соответствующий вектор начальных адресов всех столбцов. Предположим, что эта матрица используется при решении системы уравнений; целесообразно на каждом этапе исключения переменных выбирать наибольший главный элемент, чтобы сохранить точность. Необходимая последовательная перестановка столбцов может быть обеспечена перемещением элементов вектора адресов, а не целых столбцов, если выигрыш времени при отказе от перемещения чисел компенсирует потерю, связанную с их косвенной адресацией.



Второй основной прием обработки списков состоит в использовании цепочки элементов, связь между которыми обеспечивается тем, что каждый элемент содержит адрес следующего элемента цепочки. Рассмотрим опять представление предложения в памяти вычислительной машины, на этот раз несколько другим способом. Слова снова упаковываются, но каждая ячейка памяти содержит еще и адрес ячейки, в которой находится следующая часть предложения.

100	ТОТ	101
101	ПЕС	102
102	УКУ	103
103	СИЛ	104
104	ЕГО	0

Числа слева — это адреса ячеек. Числа справа являются частью содержимого ячеек. Они задают адреса следующих ячеек для предложения. Нуль означает, что следующего элемента нет. Если какие-то слова нужно добавить к этому предложению или исключить из него, то это может быть выполнено без перемещения имеющихся слов, так как не требуется, чтобы последовательные слова встречались в последовательных ячейках.

100	ТОТ	105
101	ПЕС	109
102	УКУ	103
103	СИЛ	104
104	ЕГО	0
105	ВЗБ	106
106	ЕСИ	107
107	ВШИ	108
108	ИСЯ	101
109	СЕЙ	110
110	ЧАС	102

Очевидно, что цепочка является упорядоченным набором элементов.

Третий и последний шаг к сути обработки списков читатель сделает, поняв, что элементы цепочки могут содержать адреса

либо объектов типа групп букв, как это было в примере, либо начальных элементов других цепочек такого же типа.

100	РОБИНС	200	100, 201
101	ОН+---	201	103, 0
102	ДЖОН+--	202	204, 203
103	СМИТ+--	203	206, 0
104	ДЖО+---	204	100, 205
		205	102, 0
		206	103, 207
		207	104, 0

В ячейках, показанных справа, содержатся четыре цепочки. Первая начинается с адреса 200 и состоит из двух элементов, один из которых является указателем слова РОБИНСОН, а другой — указателем слова СМИТ. В ячейке 202 начинается цепочка, состоящая из двух элементов, причем первый является цепочкой и включает РОБИНСОН и ДЖОН, а второй — тоже цепочкой, включающей СМИТ и ДЖО. Далее вместо термина «цепочка» будем употреблять общепринятый термин «список».

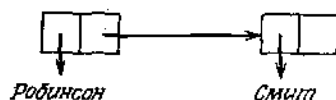
Казалось бы, нет необходимости делать различие между двумя адресами, хранящимися в каждой ячейке, но на практике это признано удобным, и мы по-прежнему будем считать, что первый указатель относится к элементам списка, а второй — к следующей ячейке списка.

При обработке списков внутри вычислительной машины данные обычно состоят из двух частей. Некоторые ячейки памяти содержат обрабатываемые объекты типа слов в предложении, которые не перемещаются. Другие ячейки принадлежат спискам и содержат адреса, указывающие на другие списки или объекты.

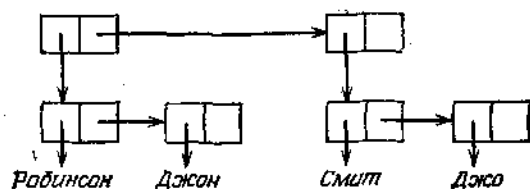
Объекты, не состоящие из двух указателей, называются атомами, так как они неделимы. В дальнейшем мы не будем уделять особого внимания фактическим видам атомов, хотя схема обработки списков, разумеется, предусматривает стандартные типы атомов, представляющих стандартные объекты. Примеры, с которыми мы будем иметь дело в этой книге, как правило, сводятся к операциям вида «изменить порядок элементов в списке на обратный», а не к таким операциям над содержимым списков, как «напечатать имя первого человека из этого списка». Это объясняется тем, что нас интересуют общие задачи обработки списков, а не конкретные представления конкретных работ.

## СПИСКИ НА БУМАГЕ

Применявшаяся нами до сих пор запись списков точно отражает содержимое памяти вычислительной машины, но она слишком громоздка для того, чтобы пользоваться ею на бумаге. Поскольку нас не интересуют конкретные ячейки, в которых хранится информация, можно отказаться от адресов и представлять ячейки в виде прямоугольников, состоящих из двух частей. Из каждой части могут выходить указатели, которыми определяются ссылки на другие прямоугольники. Как и ранее, правый указатель будет соединять ячейки списка, а левый будет указывать соответствующий элемент. Для атомов допускаются произвольные обозначения. Вот изображение списка, начинающегося с адреса 200, из предыдущего примера:



Список, начинающийся с адреса 202, изображается так:



Такая запись содержит всю информацию, которая обычно представляет интерес, но во многих случаях она все еще оказывается излишне усложненной.

Можно предложить такую запись, которая проще для пользователя и в которой элементы списка объединяются скобками и разделяются запятыми:

((РОБИНСОН, ДЖОН), (СМИТ, ДЖО))

Эта запись значительно ближе к интуитивному понятию списков. Заметим, что для предложения

(ЭТА, ЗИМА, БЫЛА, НА, РЕДКОСТЬ, МАЛОСНЕЖНОЙ)

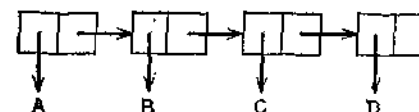
левый указатель первой ячейки показывает на слово

ЭТА

а правый показывает на список

(ЗИМА, БЫЛА, НА, РЕДКОСТЬ, МАЛОСНЕЖНОЙ)

Далее в книге будет использоваться такая запись, если в ней будет содержаться достаточно информации. Запись с прямоугольниками будет применяться только в тех случаях, когда возникнет потребность в более точном представлении содержимого машинной памяти. Ситуация, когда недостаточно простой скобочной записи, может быть проиллюстрирована на следующем примере:



С первого слова начинается список

(A, B, C, D)

а со второго слова — список

(B, C, D)

Эти скобочные выражения не отражают того факта, что данные два списка частично состоят из одних и тех же машинных ячеек, так что изменение одного списка повлечет за собой изменение другого, хотя на это не указывают никакие явные ссылки. Такое скобочное выражение пригодно для случая, когда списки состоят из различных машинных ячеек и изменение одного списка не влияет на другой.

## ПРОСТЫЕ ОПЕРАЦИИ НАД СПИСКАМИ

Теперь мы можем рассмотреть некоторые элементарные операции над списками и составленную из них программу. В первую очередь введем пару функций (операторов)  $hd(x)$  и  $tl(x)$ , обозначающих начало списка  $x$  и остаток списка  $x$ ; значениями этих функций являются соответственно левый и правый указатели от  $x$ . Если  $l$  — список.

(ЭТА, ЗИМА, БЫЛА, НА, РЕДКОСТЬ, МАЛОСНЕЖНОЙ)

то  $hd(l)$  принимает значение

ЭТА

а  $tl(l)$  принимает значение

(ЗИМА, БЫЛА, НА, РЕДКОСТЬ, МАЛОСНЕЖНОЙ)

Если  $n$  — список

((РОБИНСОН, ДЖОН), (СМИТ, ДЖО))

то  $hd(n)$  принимает значение

(РОБИНСОН, ДЖОН)

а  $tl(n)$  принимает значение

((СМИТ, ДЖО))

Рассмотрим более строго результат выполнения этих операторов над данными списками. Если применяются такие представления списков, на которых показаны адреса, то можно видеть, как возникает несимметричность. По определению будем считать, что функция  $tl$  от списка из одного элемента принимает значение нуль.

$$m = (\text{СМИТ}) \\ tl(m) = 0$$

Рассматриваемые операторы не определены для объектов, являющихся атомами. Суперпозиция разрешается, например, в случае, когда список  $p$  имеет вид

$$\begin{aligned} &(\text{РОБИНСОН, ДЖОН}) \\ &hd(p) = \text{РОБИНСОН} \\ &tl(p) = (\text{ДЖОН}) \\ &hd(tl(p)) = \text{ДЖОН} \\ &tl(tl(p)) = 0 \end{aligned}$$

Аналогично для списка

((РОБИНСОН, ДЖОН), (СМИТ, ДЖО))

имеем

$$hd(tl(hd(tl(n)))) = \text{ДЖО}$$

Заметим, что последовательные элементы списка  $x$  задаются функциями

$$hd(x), hd(tl(x)), hd(tl(tl(x))), hd(tl(tl(tl(x)))), \dots$$

Операции  $hd$  и  $tl$  расчленяют список. Для составления списка применяется оператор  $cons$ . Если  $x$  — атом или список, а  $y$  — список, то значением функции  $cons(x, y)$  является новая начальная ячейка списка, в которой левый указатель направлен на  $x$ , а правый указатель обозначает  $y$ . Другими словами,  $cons(x, y)$  — это список, состоящий из элемента  $x$ , за которым следует имеющийся список  $y$ . Если  $s$  — атом

ЭТА

а  $t$  — список

(ЗИМА, БЫЛА, НА, РЕДКОСТЬ, МАЛОСНЕЖНОЙ)

то  $cons(s, t)$  принимает значение

(ЭТА, ЗИМА, БЫЛА, НА, РЕДКОСТЬ, МАЛОСНЕЖНОЙ)

Аналогично, если  $u$  — список

(РОБИНСОН, ДЖОН)

и  $v$  — список

((СМИТ, ДЖО))

то  $cons(u, v)$  принимает значение

((РОБИНСОН, ДЖОН), (СМИТ, ДЖО))

По определению список из одного элемента формируется путем указания в операторе  $cons$  в качестве второго параметра значения нуль.

$$k = \text{СМИТ} \\ cons(k, 0) = (\text{СМИТ})$$

В принципе можно было бы разрешить, чтобы второй параметр оператора  $cons$  был атомом; однако мы никогда не будем этим пользоваться. Будем следовать правилу, что элементы списка указываются левым указателем (первым параметром), а правый указатель (второй параметр) подключает списковую ин-формацию.

Допускаются суперпозиции оператора  $cons$ :

$$\begin{aligned} h &= \text{СМИТ} \\ j &= \text{ДЖО} \\ cons(cons(h, cons(j, 0)), 0) &= ((\text{СМИТ, ДЖО})) \end{aligned}$$

Список, имеющий вид

$$(a, b, c, d, \dots, z)$$

строится по формуле

$$cons(a, cons(b, cons(c, cons(d, \dots cons(z, 0)))) \dots$$

Очевидно, что операции  $hd$  и  $tl$  являются обратными по отношению к операции  $cons$ , и в любом случае справедливы следующие формулы:

$$\begin{aligned} hd(cons(a, b)) &= a \\ tl(cons(a, b)) &= b \end{aligned}$$

Однако обратные правила не всегда выполняются. Значением

$$cons(hd(x), tl(x))$$

является новая ячейка в машинной памяти, содержащая такие же указатели, как и исходная ячейка  $x$ . Другими словами, результатом является копия ячейки  $x$ , но не сама ячейка  $x$ .

Перечисленные операции позволяют нам написать короткую программу обработки списка, находящегося в машинной памяти. Предположим, что задан список пар фамилий и имен вида

((РОБИНСОН, ДЖОН), (СМИТ, ДЖО))

и что нужно получить список, состоящий только из фамилий. Длина списка неизвестна, порядок появления фамилий в списке несуществен. Обозначим через  $l$  исходный список и через  $m$  — тот список, который нужно построить. Первоначальное состояние списка  $l$  не сохранится.

```

m := 0;
цикл: if l ≠ 0 then
  begin m := cons(hd(hd(l)), m);
        l := tl(l);
        go to цикл
  end;

```

Рассмотрим применение этой процедуры к предыдущему списку. При первом попадании на метку *цикл*<sup>1)</sup> имеем

$l = ((\text{РОБИНСОН}, \text{ДЖОН}), (\text{СМИТ}, \text{ДЖО}))$   
 $m = 0$

После выполнения составного оператора и нового перехода к метке *цикл* имеем

$l = ((\text{СМИТ}, \text{ДЖО}))$   
 $m = (\text{РОБИНСОН})$

При третьем попадании на метку *цикл* получаем

$l = 0$   
 $m = (\text{СМИТ}, \text{РОБИНСОН})$

и программа прекращает работу. В полученном списке фамилии следуют в обратном порядке по сравнению с исходным списком.

<sup>1)</sup> При переводе примеров используется вариант языка АЛГОЛ-60, в котором допускаются идентификаторы, содержащие буквы русского алфавита. — Прим. перев.

Мы начнем эту главу с рассмотрения двух методов программирования: метода рекурсии и метода итерации. При этом будут использоваться введенные ранее операции, а также две новые операции *atom* и *eq*. Процедура *atom(x)* принимает значение **true** (истина), если  $x$  — атом; в противном случае она принимает значение **false** (ложь). Процедура *eq(x, y)* принимает значение **true**, если атомы  $x$  и  $y$  одинаковы, и значение **false** в противном случае. Эта процедура определена только тогда, когда оба ее аргумента являются атомами.

Под итерацией понимается обычный способ программирования, примером применения которого является последняя программа из предыдущей главы. Повторное выполнение операций организуется передачами управления назад и возобновлением выполнения последовательности команд до тех пор, пока не будет произведено достаточного количества повторений. Ниже приводится простая программа перевертывания списка  $l$ , т. е. перестановки в обратном порядке элементов списка  $l$ ; результатом является список  $m$ .

```

m := 0;
цикл: if l ≠ 0 then
  begin m := cons(hd(l), m);
        l := tl(l);
        go to цикл;
  end;

```

Работа этой программы не нуждается в пояснении, так как аналогична примеру, разобранным в предыдущей главе. Изменяется порядок элементов только на верхнем уровне списка, так что если применить эту программу к списку

((РОБИНСОН, ДЖОН), (СМИТ, ДЖО))

то результатом будет список

((СМИТ, ДЖО), (РОБИНСОН, ДЖОН))

Следующая программа присваивает величине  $m$  значение того атома, который встречается в списке  $l$  на первом месте без учета скобок:

```

цикл: if atom(l) then m := l
      else
      begin l := hd(l);
            go to цикл
      end;

```

Список  $l$  мог бы принимать последовательно значения

```

((РОБИНСОН, ДЖОН), (СМИТ, ДЖО))
(РОБИНСОН, ДЖОН)
РОБИНСОН

```

причем последнее значение было бы присвоено величине  $m$ .

### РЕКУРСИЯ

Предыдущие программы были очень просты, и их работа сводилась к тому, что они осуществляли последовательный спуск по начальным или по конечным указателям списка. Часто возникает ситуация, когда при спуске внутрь списка нужно выполнить то же самое действие над подписанием, причем после завершения обработки подписка следует вернуться к основному списку. В таком случае полезным способом программирования может оказаться рекурсия.

Под рекурсией здесь понимается в точности то же самое, что и в обычной математике; приводимое ниже определение является рекурсивным потому, что значение факториала ищется путем нового применения функции *factorial*:

```

integer procedure factorial(n); integer n;
factorial := if n = 0 then 1 else n × factorial(n - 1);

```

Вернемся к проблеме отыскания первого атома из списка и на этот раз воспользуемся рекурсивной программой. Если список представляет собой атом, то он уже является искомым результатом. В противном случае необходимо найти первый атом подписка, являющегося началом данного списка. Над подписанием нужно выполнить в точности такое же действие, как и над исходным списком. Можно описать соответствующую процедуру, которую мы назовем *первый атом* (*firstatom*, [17]);

значением этой процедуры является первый атом ее параметра <sup>1)</sup>.

```

list procedure первый атом (x); list x;
первый атом := if atom (x) then x else первый атом (hd(x));

```

Аналогично, можно написать рекурсивную программу перевертывания верхнего уровня списка.

```

list procedure rev (x, y); list x, y;
rev := if x = 0 then y else rev(tl(x), cons(hd(x), y));

```

Если требуется получить список  $m$  перевертыванием списка  $l$ , то достаточно записать

```
m := rev(l, 0);
```

Рассмотрим работу этой программы над списком

```
(A, (B, C), D)
```

выписывая значения параметров при каждом обращении к процедуре *rev*.

Первый параметр	Второй параметр
(A, (B, C), D)	0
((B, C), D)	(A)
(D)	((B, C), A)
0	(D, (B, C), A)

При достижении первым параметром значения 0 производится присваивание функции *rev* значения второго параметра и выполнение процедуры завершается. Производится возврат к предыдущему обращению к процедуре *rev* и при этом найденное значение присваивается функции *rev* с предыдущего уровня. Тем самым соответствующее выполнение процедуры завершается, и такой процесс продолжается до тех пор, пока не завершится исходное обращение к процедуре, причем результатом будет все тот же список.

Следующая процедура, также рекурсивная, просматривает все атомы списка в поиске какого-нибудь атома, равного заданному (независимо от структуры списка). И в этом случае нужно выполнять над любым подписанием такой же процесс, как и над исходным списком.

```

boolean procedure mem(a, l); list a, l;
mem := if atom(l) then eq(a, l)
      else if mem(a, hd(l)) then true else mem(a, tl(l));

```

<sup>1)</sup> Описатель *list* означает тип списковой структуры. — Прим. перев.

Если список является атомом, то его можно сразу проверить на равенство атому  $a$ . В противном случае тот же процесс применяется к началу списка. Если этот подсписок содержит атом  $a$ , то на исходный вопрос дается положительный ответ и больше ничего не нужно делать. Если же начало не содержит атома  $a$ , то необходимо проверить остаток.

Написание рекурсивных процедур часто кажется неестественным при отсутствии соответствующей практики. В этом отношении типичен рассматриваемый выше процесс. Предположим, что процедура  $mem(a, l)$  правильно работает для всех списков, содержащихся в списке  $l$ . Тогда можно определить ее для самого списка  $l$  условным выражением

**if**  $mem(a, hd(l))$  **then** **true** **else**  $mem(a, tl(l))$

так как процедура  $mem$  применима и к началу, и к остатку списка  $l$ . Это справедливо, если  $l$  не атом. Если же  $l$  — атом, то он не содержит внутренних списков и проблема решается операцией  $eq(a, l)$ . Объединяя эти рассуждения, мы получаем описанную выше процедуру.

Эти процедуры довольно просты, но рекурсивные программы бывают и логически сложными. Следующая программа не является простой, хотя рассуждения такого же типа объясняют ее работу. Задается произвольный список  $l$  и требуется получить новый список из тех же атомов в том же порядке, но с одним уровнем атомов. Пусть список  $l$  имеет вид

(A, (B, (C, D)), E)

Вывороченный вариант этого списка приобретает вид

(A, B, C, D, E)

**list procedure**  $flat(a, b)$ ; **list**  $a, b$ ;

$flat :=$  **if**  $a = 0$

**then**  $b$

**else if**  $atom(a)$  **then**  $cons(a, b)$

**else**  $flat(hd(a), flat(tl(a), b))$ ;

В результате выполнения функции  $flat(a, b)$  вывороченный вариант списка  $a$  помещается перед списком  $b$ . Поэтому функция  $flat(l, 0)$  определяет искомый список. Предположим, что  $a$  — список и что процедура  $flat$  правильно работает для всех меньших списков. Тогда процедура

$flat(tl(a), b)$

помещает вывороченный вариант списка  $tl(a)$  перед списком  $b$ . Процедура

$flat(hd(a), flat(tl(a), b))$

помещает вывороченный вариант списка  $hd(a)$  перед вывороченным вариантом списка  $tl(a)$  перед списком  $b$ . Таким образом, процедура применима к списку  $a$ . Если же список  $a$  пуст, то ответом должен быть список  $b$ . Если  $a$  — атом, то можно поместить  $a$  непосредственно перед  $b$ , выполнив операцию  $cons(a, b)$ . Объединяя эти три возможных варианта, получаем данное выше определение. Обработка списка процедурой  $flat$  выглядит следующим образом:

$flat[(A, (B, (C, D)), E), 0]$

$flat[A, flat[[(B, (C, D)), E), 0]]$

$flat[A, flat[(B, (C, D)), flat[E, 0]]]$

$flat[A, flat[(B, (C, D)), (E)]]$

$flat[A, flat[B, flat[[(C, D)], (E)]]]$

$flat[A, flat[B, flat[(C, D), flat[(0, (E))]]]]$

$flat[A, flat[B, flat[(C, D), (E)]]]$

$flat[A, flat[B, flat[C, flat[(D), (E)]]]]$

$flat[A, flat[B, flat[C, flat[D, flat[0, (E)]]]]]$

$flat[A, flat[B, flat[C, flat[D, (E)]]]]$

$flat[A, flat[B, flat[C, (D, E)]]]$

$flat[A, flat[B, (C, D, E)]]$

$flat[A, (B, C, D, E)]$

(A, B, C, D, E)

Очень наглядным примером удобства применения рекурсии может служить программа вычисления арифметического выражения. Пусть список  $l$  имеет одну из следующих структур:

(+, (a, b))    (−, (a, b))    (×, (a, b))    (/, (a, b))

Здесь  $a$  и  $b$  — либо атомы с численными значениями, либо списки такого же вида, как список  $l$ . Типичным является список

(/, (+, (A, (−, (B, C))))), (×, (/, (D, E)), F))

представляющий выражение

$(A + (B - C)) / ((D/E) \times F)$

Рекурсия применяется потому, что при вычислении выражения нужно сначала вычислить параметры, а затем выполнить над

ними операцию, причем вычисления выполняются одинаково на обоих уровнях.

```

real procedure вычислить(l); list l;
begin list m, n;
    if atom(l) then вычислить := l
    else begin m := hd(hd(tl(l))); n := hd(tl(hd(tl(l)));
        вычислить := if hd(l) = '+' then вычислить(m) +
                                вычислить(n)
                    else if hd(l) = '-' then вычислить(m) -
                                вычислить(n)
                    else if hd(l) = 'X' then вычислить(m) X
                                вычислить(n)
                    else вычислить(m)/вычислить(n)
    end
end;

```

### ЭФФЕКТИВНОСТЬ РЕКУРСИИ

В некоторых языках программирования рекурсия влечет за собой значительные затраты памяти или машинного времени, или и того и другого. Рассмотрим вопрос, при каких условиях следует ожидать, что рекурсия окажется лучшим способом программирования, чем итерация, и приведем примеры неэффективной и эффективной рекурсий.

Недостатки и преимущества рекурсии связаны с тем фактом, что при каждом попадании на более глубокий уровень обращения возникает необходимость запоминания состояния рабочих ячеек и ячеек связи для того, чтобы иметь к ним доступ при возврате на данный уровень. Если рекурсия очень глубока, то для этого могут потребоваться большие затраты памяти. Они могут быть оправданы, если запоминаемая информация является существенной и будет использоваться при возврате на данный уровень. Однако они могут оказаться совершенно напрасными, если эту информацию не предполагается использовать. Рассмотрим рекурсивную программу перевертывания списка.

```

list procedure rev(x, y); list x, y;
rev := if x = 0 then y else rev(tl(x), cons(hd(x), y));

```

При работе этой программы результат постепенно формируется во втором параметре. Как только первый параметр свелся к нулю, производится по очереди выход из всех уровней

рекурсии без выполнения какой-либо дополнительной работы. Если список содержал тысячу подписков, то процедура прошла бы тысячу уровней вглубь, а затем последовательно выходила бы из них вхолостую. Выполнение процедуры было бы медленным, и так как каждый уровень рекурсии может использовать несколько рабочих ячеек, то для программы могло бы потребоваться несколько тысяч бесполезных ячеек. Итеративная программа перевертывания списка не требует излишней рабочей памяти и не производит ненужной работы. В этом случае применение рекурсии оказалось бы чрезвычайно обременительным.

Аналогично, рекурсивная программа отыскания первого атома в списке неэкономна по сравнению с итеративной программой. С другой стороны, рекурсивная программа вычисления выражения экономна по следующей причине. Пусть нужно вычислить выражение по списку, который начинается с символа «плюс». Тогда необходимо вычислить один параметр, запомнить его, вычислить второй параметр и затем сложить полученные два числа. В точности это и выполняется при рекурсии, так что в данном случае попытка обойтись без рекурсии привела бы к трудностям без повышения эффективности. То же самое справедливо для процедуры выравнивания списка и для процедуры проверки того, входит ли некоторый атом в некоторый список.

Можно продемонстрировать эффективность рекурсии на очень изящном и простом примере рекурсивной программы копирования списка. (Этот пример заимствован из работы [29].) Копией списка называется список, состоящий из других ячеек машинной памяти, но с такими же связями между частями и со ссылками на те же атомы. Если употребляется формула

$$m := l;$$

то копирование списка не происходит; *m* — это тот же самый список *l*.

```

list procedure копия(l); list l;
копия := if atom(l) then l else cons(копия(hd(l)), копия(tl(l)));

```

Предположим, что процедура *копия* применима ко всем под спискам списка *l*. Тогда при помощи операции

$$\text{cons}(\text{копия}(\text{hd}(l)), \text{копия}(\text{tl}(l)))$$

получаем новую ячейку со ссылками на копии начала и остатка списка *l*. Если *l* — атом, то в соответствующей ячейке содержится само значение. Приведенная ниже итеративная

программа получения копии  $l$  списка  $b$  неудобна и не имеет преимуществ перед рекурсивной программой.

```

 $l := b;$ 
if atom ( $l$ ) then go to выход;
 $p := m := q := 0;$ 
r2: if  $l = 0$  then
begin r1: if  $p \neq 0$  then
begin  $l := \text{cons}(\text{hd}(p), l); p := \text{tl}(p);$  go to r1 end;
if  $q = 0$  then go to выход;
 $p := \text{cons}(l, \text{hd}(q));$ 
 $q := \text{tl}(q);$ 
 $l := \text{hd}(m);$ 
 $m := \text{tl}(m);$ 
go to r2
end
end
else if atom ( $\text{hd}(l)$ ) then
begin  $p := \text{cons}(\text{hd}(l), p);$ 
 $l := \text{tl}(l);$ 
go to r2
end
else
begin  $m := \text{cons}(\text{tl}(l), m);$ 
 $q := \text{cons}(p, q);$ 
 $p := 0;$ 
 $l := \text{hd}(l);$ 
go to r2
end;
end;
```

Разумеется, языки, в которых нет рекурсии, обеспечивают другие средства выполнения аналогичной работы, например наборы процедур последовательного прохождения списков различными способами.

Может оказаться целесообразным избегать чрезмерной глубины рекурсии, возникающей от остатка списка. Отсутствие симметрии между началами и остатками проявляется в том, что глубина рекурсии относительно начал часто оказывается значительно меньше, чем глубина рекурсии относительно остатков. Иногда лучше выполнять рекурсию над началами и итерацию над остатками. Приведенная ниже процедура  $\text{mem}(a, l)$  может

использовать меньше памяти, чем полностью рекурсивный вариант той же процедуры.

```

boolean procedure mem ( $a, l$ ); list  $a, l$ ;
 $n$ : if atom ( $l$ ) then  $\text{mem} := \text{eq}(a, l)$ 
else if mem ( $a, \text{hd}(l)$ ) then  $\text{mem} := \text{true}$ 
else if  $\text{tl}(l) = 0$  then  $\text{mem} := \text{false}$ 
else begin  $l := \text{tl}(l);$ 
go to  $n$ 
end;
```

Вот два правила, которыми можно руководствоваться, решая вопрос о целесообразности применения рекурсии:

1. Если задачу легко решить итеративно, то пользуйтесь итерацией.
2. Если рекурсивный вариант процедуры содержит два обращения к собственному наименованию, то он может оказаться полезным.

### МАГАЗИННЫЕ СПИСКИ

Магазинные списки тесно связаны с рекурсией при обработке списков. Магазинный список представляет собой такой вид очереди, при котором должны обрабатываться первыми те элементы, которые появились в очереди последними. Такая структура называется также стеком, погребом или очередью типа LIFO (начальные буквы английской фразы «last — in — first — out», означающей «последним пришел — первым ушел»). Программа, которая выполняет копирование списка без применения рекурсии, работает с тремя магазинными списками  $p$ ,  $q$  и  $m$ . Хорошо известно применение магазинных списков при вычислении выражений в обратной польской записи. Эта запись очень удобна для работы с вычислительными машинами благодаря тому, что она бессклобочная, а также благодаря простому способу вычисления выражений. Если выражение вида

$$a + b \times (c + d/e)/f$$

перестроить в функциональную запись, поместив названия функций после аргументов

$$(a, (b, (c, (d, e)/f) +) \times, f)/f +$$

и исключить скобки и запятые

$$abcde/f + \times /f +$$



то результатом будет обратная польская запись исходного выражения. Эта запись может быть преобразована снова в скобочную форму единственным способом, так как для каждого оператора известно число аргументов. Продвигаясь по обратной польской записи слева направо, можно утверждать следующее. Значение  $a$  должно быть аргументом некоторого оператора; запомним его. Так же поступим со значениями  $b$ ,  $c$ ,  $d$  и  $e$ . Обои́ми аргументами деления должны быть последние два запомненных объекта  $d$  и  $e$ ; можно объединить их, получив в результате одно число, частное. Далее оператору плюс нужны два аргумента; первым аргументом должно быть значение  $c$ , а вторым — полученное только что частное. Аналогично можно узнать структуру всего выражения. Если требуется вычислить такое выражение, представленное в виде списка  $l$ , то можно воспользоваться следующей программой, в которой с помощью магазинного списка реализуется алгоритм, весьма похожий на описанный выше способ отыскания аргументов <sup>1)</sup>:

```

rep: t := hd(l);
l := tl(l);
if буква (t) then x := cons(t, x)
else x := cons(применение(t, hd(x), hd(tl(x))), tl(tl(x)));
go to rep;

```

Процедура *применение* выдает в качестве своего результата значение, которое получается применением оператора, являющегося ее первым аргументом, к параметрам, заданным во втором и третьем аргументах. Магазинный список  $x$  пополняется, когда встречается какая-то переменная или когда получается промежуточный результат, а выборка (с исключением) из него элементов производится для формирования аргументов арифметических операторов. Эти операции добавления и исключения элементов в магазинных списках встречаются так часто, что в некоторых языках для их выполнения предусмотрены специальные функции.

### ОПЕРАЦИИ ИЗМЕНЕНИЯ СПИСКОВ

До сих пор использовались только операции *cons*, *hd*, *tl*, *atom* и *eq*. Эти операции служат либо для исследования существующего списка, либо для создания совершенно нового списка. Они не изменяют списков. Возможность изменения списка является весьма эффективной, но и очень опасной.

<sup>1)</sup> Функция *буква* ( $t$ ) принимает значение true, если  $t$  — буква. — Прим. перев.

Чтобы изменить список, необходимо заменить либо целую ячейку, либо некоторую часть ячейки. Мы вводим три новых оператора *set*, *sethd* и *settl*. Результатом операции *set*( $x, y$ ) является замена всего содержимого ячейки  $x$  на значение  $y$ . Результатом операций *sethd*( $x, y$ ) и *settl*( $x, y$ ) является замена соответственно либо начала  $x$ , либо остатка  $x$  на значение  $y$ .

Опасность применения этих операций возникает из-за существования общих подсписков, о которых программист может забыть. Если какой-то список изменяется и является частью другого списка, то тот список тоже изменится. Часто избегают пользоваться такими операциями, потому что к концу сложного участка счета может оказаться затруднительным помнить об источниках возникновения списков. Эффективность этих операций обусловлена двумя причинами. Первая причина является как бы обратной стороной описанной трудности. Когда список изменяется, то для изменения его применений не требуется дополнительной работы. Вторая причина состоит в том, что может отпасть необходимость выполнять много копирований списков.

Рассмотрим случай, когда новые элементы нужно добавлять в конец списка, а не в начало. Поскольку нужно изменить конец списка, мы могли бы получить копию всего списка с измененным последним элементом, если нет возможности изменить последний элемент там, где он находится. Лучшим вариантом, если это приемлемо, было бы построение списка в обратном порядке с последующим перевертыванием. Однако так нельзя работать, если список используется одновременно с пополнением. Данную задачу можно решить совсем просто, воспользовавшись операцией *settl*. В следующей программе обрабатываемая очередь имеет наименование  $l$ .

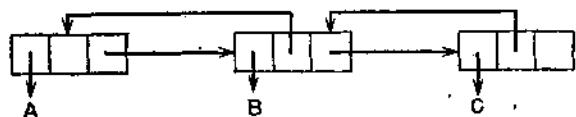
```

цикл: x := следующий атом;
if l = 0 then l := t := cons(x, 0)
else begin settl(t, cons(x, 0));
          t := tl(t)
        end;

```

Различные языки обработки списков отличаются размерами ячеек, допустимыми операциями и стандартными представлениями простых списков. Казалось бы, лучше предоставить свободу в этих вопросах пользователю, чтобы он смог выбирать себе параметры, наиболее подходящие для его задачи. Однако слишком большая свобода приводит к неудобству машинного кодирования, а установившийся и привычный способ мышления может оказаться предпочтительнее, чем беспрестанная оптимизация, ухудшающая возможность обмена алгоритмами. Поэтому языки налагают специальные стандартные ограничения, некоторые из которых рассматриваются и поясняются в этой главе.

Если отказаться от ограничения, что каждая ячейка содержит ровно два указателя, и ввести соответствующие операции проверки содержимого различных частей ячейки, заполнения новых ячеек и изменения содержимого имеющихся ячеек, то представление объектов в машине становится очень гибким. Нет необходимости придерживаться правила, что указатель начала показывает на элементы списка, а указатель остатка содержит адрес следующей ячейки списка. Свойства более гибкого машинного представления могут быть в точности воспроизведены на бумаге посредством записи, содержащей прямоугольники и стрелки. Например, обычный список мог бы быть представлен на схеме следующим образом:

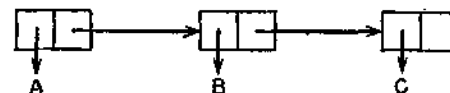


Однако если этот список описан в виде

(A, B, C)

то это подразумевает некоторые предварительные соглашения. На верхней схеме такой список представляется внутри машины совсем не так, как тот же список в применявшейся ранее форме,

который был бы представлен следующим образом:



Итак, одна и та же простая запись может соответствовать различным представлениям в различных языках обработки списков; возможные операции также будут существенно отличаться.

Например, очевидно, что приведенная выше схема с тремя указателями в каждой ячейке предназначена для того, чтобы сделать легко осуществимым продвижение по списку в обоих направлениях. Такое продвижение может оказаться затруднительным при описанном ранее представлении. Простейшим примером применения этой возможности может служить следующая задача. Предположим, что требуется отыскать в списке атомов такой атом, который встречается непосредственно перед заданным атомом  $x$ . Если предполагается наличие указателей только одного направления, придется запоминать атомы, непосредственно предшествующие каждому атому из списка, на случай, если именно он окажется нужным.

```
rep: a := hd (l);
      l := tl (l);
      if hd (l) ≠ x then go to rep;
```

При структуре с указателями обоих направлений можно получить искомый результат сдвигом назад по списку на одну позицию после обнаружения совпадения очередного атома с атомом  $x$ .

```
rep: if первый (l) ≠ x then begin l := вниз (l); go to rep end;
      a := первый (верх (l));
```

В этом примере<sup>1)</sup> отличие тривиальное, но оно могло бы оказаться существенным, если бы проверка совпадения производилась совсем в другой части программы.

Возможен также вариант, при котором каждая ячейка по-прежнему содержит два указателя, но изменяются соглашения о форме окончания списков. Если рекурсия не допускается и нужны операции типа копия (неудобные для программ без рекурсии), то можно иметь в конце каждого подсписка дополнительный указатель, показывающий то место в основном списке, которому соответствует данный подсписок [24]. До некоторой

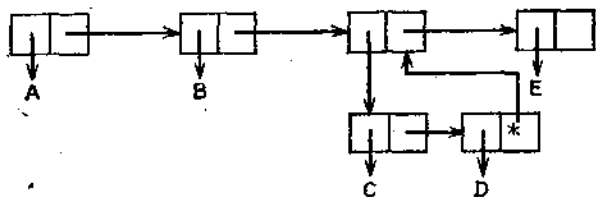
<sup>1)</sup> Здесь *первый* — первый элемент списка, *вниз* — нижний указатель, *верх* — верхний указатель. — Прим. перев.

степени это соответствует запоминанию места возврата, обеспечивающему простоту рекурсивных программ. Необходимо заметить, что последняя ячейка подписка не только продолжает общий список, но и является концом подписка, и поэтому в ней должны быть предусмотрены дополнительные разряды для указания соответствующего признака. Если процедура обработки списка доходит до точки, снабженной признаком конца подписка, то производится переход к рассмотрению того места в основном списке, которому соответствует данный подписание (вместо продвижения на один уровень рекурсии вверх). Существенное неудобство этой схемы состоит в том, что каждый список может быть подписком только одного списка и, если он нужен как часть другого списка, его нужно копировать. Если общие списки не допускаются, то при решении некоторых задач возникают тяжелые затруднения из-за нехватки памяти.

При этой схеме список

(A, B, (C, D), E)

получил бы машинное представление



(звездочка означает признак конца подписка). Заметим, что подписание (C, D) содержит указатель соответствующего ему места в основном списке и поэтому не может нигде больше служить подписанием.

## АТОМЫ

Различные языки существенно отличаются также представлением атомов внутри вычислительной машины. Атомы — это такие части информации, которые должны рассматриваться как неделимые в процессе обработки списка. Атомами могут быть числа, а также строки символов, представляющие имена, под которыми известны какие-то объекты, или, быть может, информацию об объектах. Существенно, что структура атомов не имеет отношения к специфике программы, выполняющей обработку списка, в котором они встречаются. Атомами одного списка могут служить другие списки при условии, что некоторый признак указывает системе, что данные списки являются

атомами. Атом может иметь и собственную структуру, но он должен быть снабжен указанием, что его структура не рассматривается как часть структуры содержащих его списков. Если система способна опознать такое указание, то этого достаточно.

## ВЕКТОРЫ

Весьма полезна возможность включения в список вектора информации. Под вектором понимается набор ячеек, имеющих последовательные машинные адреса. Поэтому при работе с вектором удобно пользоваться машинными индексными операциями и можно вычислять адреса нужных элементов. Элементами вектора могут быть числа, символы или другие списки.

Если программе потребуется доступ к элементам списка не в последовательном порядке (который легко обеспечивается списковой структурой), а в более или менее случайном порядке (например, к четвертому, двенадцатому и двадцатому элементам списка), то возникнет значительная потеря эффективности из-за того, что элементы списка просматриваются по очереди в поисках нужных элементов. Очевидно, что лучше вместо этого воспользоваться имеющейся во всех машинах возможностью доступа к элементам с помощью базы и вычисляемого индекса. Кроме того, можно получить выигрыш в экономии памяти. Если величины организованы сложным способом и структура содержит много информации, то большой расход памяти на указатели является оправданным. Однако если некоторые данные легко доступны и хранятся в последовательных ячейках, то нецелесообразно затрачивать чуть ли не половину памяти на представление весьма простой структуры. Использование векторов может вызвать значительные затруднения, которые будут рассмотрены далее. Вот некоторые примеры применения векторов.

Очевидным примером является обыкновенный вектор чисел. Обычно в машинах предусматриваются команды эффективного вычисления скалярного произведения двух таких векторов. При этом предполагается, что каждый вектор хранится в последовательных ячейках.

Вектор списков находит очень важное применение в одном из методов «перемешанного кодирования». Задача состоит в том, чтобы отыскать нечто связанное с объектом, если задан сам объект. Например, может потребоваться, прочтя фамилию человека, найти в памяти информацию о нем. Если хранится информация об очень большом количестве людей, то попытка просмотра списка всех фамилий привела бы к недопустимому замедлению. При данном методе кодирования среднее время поиска делится на любое заранее заданное целое значение  $n$

ценой дополнительной затраты памяти, занимаемой вектором из  $n$  элементов. Вместо того чтобы хранить один список из фамилий всех людей, хранятся  $n$  списков, каждый из которых содержит примерно  $1/n$  количества всех фамилий. Если понадобится какая-то фамилия, то выбирается и просматривается соответствующий список. Если удобно выбрать  $n=26$  (по числу букв в латинском алфавите) и если пренебречь неравномерностью распределения начальных букв фамилий, то эти начальные буквы могли бы служить для индексации списков. Можно было бы организовать вектор из 26 списков и выбирать нужный список путем выполнения одной индексной операции при чтении фамилии. Поскольку индекс вычисляется, применение вектора оправдано. Можно было бы использовать любую другую функцию от фамилии, принимающую целые значения из диапазона от 1 до  $n$  и достаточно равномерно распределенную. Приводим фрагмент программы, в котором выполняется отыскание значения, соответствующего заданной фамилии.

```

l := прочесть фамилию;
i := функция (l);
x := вектор списков [i];
цикл: if x ≠ 0 then
begin y := hd (x);
      x := tl (x);
      if eq (hd (y), l) then
begin значение := hd (tl (y));
      go to выход
end;
      go to цикл
end;
выход:

```

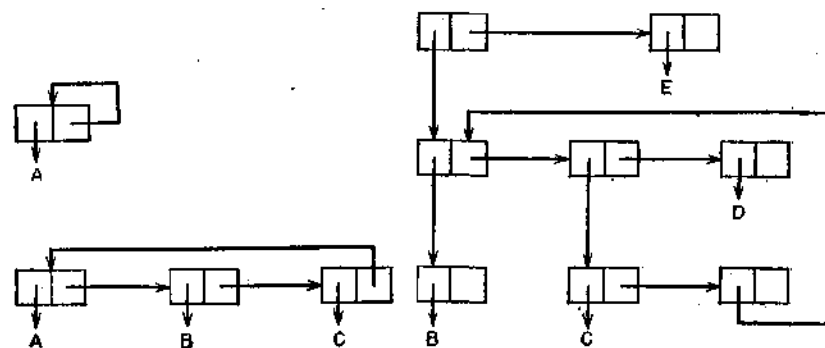
Если требуется всегда наилучшим образом использовать память, то может возникнуть трудность, состоящая в необходимости перемещать векторы в машинной памяти; процесс такого перемещения является крайне медленным и требует сложных методов обновления ссылок. Мы рассмотрим этот вопрос в гл. 6. Можно сократить количество необходимых перемещений данных, наложив некоторые ограничения на представления, связанные с использованием векторов (см. [3, 6]).

Если программа начинает работать, располагая некоторой областью памяти, предназначенной для размещения векторов, то не представляет труда получать последовательно новые блоки до заполнения этой области. К этому моменту может отпасть

нужда в некоторых векторах. Если нужны новые векторы, то они могут занять освободившиеся места в памяти при условии, что они там поместятся. Если не предполагается перемещать блоки в памяти для размещения новой информации, то следует объединять соседние области в памяти, занятые ненужной информацией; благодаря этому может открыться возможность размещения больших векторов. Если векторы нужны для экономии памяти, а не для использования быстроты индексных операций, то можно размещать их частями в имеющихся блоках освободившейся памяти при условии, что отдельные части объединяются цепочками ссылок для указания структуры. Безусловно, таким способом достигается экономия памяти, поскольку всегда будет использоваться не больше указателей, чем в случае одноэлементных векторов, к которому сводится дело при работе с обыкновенными списками, а на начальных этапах выполнения программы будет достигнуто очень большое сокращение числа указателей.

## ЦИКЛИЧЕСКИЕ СПИСКИ

Имеется возможность организовать в машинной памяти циклические списки. Они не допускают непосредственного представления в скобочной записи, но могут быть изображены с помощью прямоугольников. Под циклическим списком понимается список, в котором указатель из некоторой ячейки направлен на такое место из списка, откуда данная ячейка может быть достигнута снова. Например, следующие списки являются циклическими:



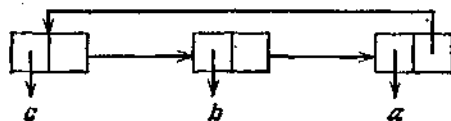
Такие списки не могут быть построены, если единственным способом формирования новых начальных ячеек является только процедура *cons*. В самом деле, процедура *cons* создает для списка ячейку, содержащую два указателя на элементы, которые заведомо были сформированы раньше этой ячейки и

поэтому не могут показывать на нее. Цикличность можно ввести только с помощью процедур, изменяющих существующий список.

Операторы

```
l := cons (a, 0);
m := cons (c, cons (b, l));
settle (l, m);
```

порождают список



Цикличность не вносит в программирование новых проблем, но если она разрешается, то это может сказаться на структуре системы.

Применявшаяся раньше скобочная запись непригодна для представления существа циклических списков или для выражения того факта, что один список может быть подсписком в нескольких местах. Можно выразить точную структуру, если разрешить записывать в качестве элементов списков и имена других списков. Третий из приведенных выше списков можно было бы представить в следующей записи:

```
(x, E)
где x = ((B), (C, x), D)
```

#### СПИСКИ ВО ВСПОМОГАТЕЛЬНОЙ ПАМЯТИ

Если емкость машинной памяти недостаточна для решения нужной задачи или если требуется запоминать результаты на некоторое время между просчетами, то может понадобиться помещать списки во внешнюю память. Представление списка во внешней памяти может совсем отличаться как от представления в основной машинной памяти, так и от записи, применяемой программистом. Обычное машинное представление удобно только до тех пор, пока элементы списка занимают постоянные места в памяти, отраженные в указателях. Но оно порождает значительные трудности при считывании списков в основную память, когда нельзя использовать старые адреса. То представление, с которым имеет дело пользователь, тоже не подходит, так как оно часто оказывается недостаточно детальным для того, чтобы дать возможность в точности восстановить список. Если список, включающий общий подсписок, помещается во

внешнюю память, а потом считывается оттуда, то обычно возникает необходимость восстановить точную структуру общих подсписков вместо того, чтобы просто копировать их много раз. Поэтому во внешней памяти должна быть представлена детальная структура списков. Впрочем представление во внешней памяти имеет то преимущество, что оно предназначено для машинного чтения, а не для человеческого восприятия; поэтому не требуется, чтобы оно было понятно для людей. В частности, можно пользоваться той формой записи, которая описана выше для циклических списков, уплотнив ее так, чтобы она стала удобной для машинного применения. Для атомов также необходимо найти некоторую форму представления.

Чтобы проиллюстрировать применение обработки списков, мы рассмотрим построение простой программы, работающей со списком. Эта программа будет читать предложение и решать, соответствует ли оно введенному заранее набору грамматических правил. На применении этой программы основан метод, разработанный Куно и Оттингером [33] и известный под названием прогнозирующий анализ.

Применительно к этой программе будем считать, что грамматика языка состоит из набора грамматических правил. Каждое правило описывает одну из структур некоторого грамматического класса, такого, как именная группа, определение или предложение. Например, следующие правила могли бы входить в число правил, определяющих возможные структуры классов именной группы и предложения:

(именная группа) → *the* (существительное)  
 (именная группа) → *the* (прилагательное) (существительное)  
 (именная группа) → (именная группа) (именная группа) (глагол)  
 (предложение) → (именная группа) *is* (прилагательное)

Название класса помещается слева, а справа помещается список из английских слов и названий классов<sup>1)</sup>. Каждое правило интерпретируется так, что правая часть является примером класса, название которого помещено слева. Применяв второе правило, мы могли бы обнаружить, что «*the black cat*» является примером именной группы, если известно, что «*black*» является примером прилагательного, а «*cat*» — примером существительного. Аналогично можно показать, что «*the man the black cat likes*» является именной группой. Предполагается, что программа читает набор таких грамматических правил, а затем читает предложения и пытается решить, правильно ли они построены.

Метод прогнозирующего анализа будет работать, если в каждом правиле список слов и названий классов начинается со

<sup>1)</sup> Для удобства чтения входящие в список названия классов при переводе заключены в скобки (и). — Прим. перев.

слова. Первые два описания, данные для именной группы, удовлетворили бы этому требованию, так как они начинаются со слова «*the*». Третье правило не удовлетворило бы этому требованию, потому что оно начинается с названия класса (именная группа). Можно переписать правила для именной группы и привести их в соответствие с этим ограничением.

(именная группа) → *the* (существительное)  
 (именная группа) → *the* (прилагательное) (существительное)  
 (именная группа) → *the* (существительное) (именная группа) (глагол)  
 (именная группа) → *the* (прилагательное) (существительное) (именная группа) (глагол)

Этими правилами будут разрешаться в точности те же самые предложения. Можно показать, что любая такая грамматика может быть переписана так, чтобы охватывались все правильные предложения и чтобы при этом удовлетворялось указанное ограничение.

Предположим, что имеется список, состоящий из слов и названий классов и представляющий одну из возможных структур предложения. Мы можем предположить, что этот список начинается со слова. Первое слово проверяемого предложения читается и сравнивается с первым словом из списка. Если они отличаются, то эта структура не может соответствовать данному предложению. Если же они одинаковы, то возможен дальнейший анализ. Первое слово из списка уже проверялось, поэтому оно отбрасывается. Если второй элемент является словом, то повторяется ситуация, рассмотренная только что. Предположим, что второй элемент является названием класса. Каждое из правил, описывающих соответствующий класс, начинается со слова. Очередное слово из предложения читается и сравнивается со всеми этими начальными словами. Те правила, которые начинаются с этого слова, могут оказаться соответствующими структуре предложения; остальные правила заведомо не соответствуют. Поэтому мы конструируем из исходного списка несколько списков, по одному для каждого правила, начинающегося с прочитанного слова. Каждый из этих новых списков получается путем помещения остатка одного из правил перед остатком исходного списка. Затем все эти списки обрабатываются параллельно таким же способом, как и исходный список. Если какой-то список оказывается пустым, то анализ выполнен.

Рассмотрим попытку анализа предложения «*the man the black cat likes is ill*»<sup>1)</sup>. Программа начинает работу при наличии

<sup>1)</sup> Человек, которого любит черный кот, болен (англ.).



Если программа доходит до конца условного оператора с меткой *rl*, то это значит, что список *l* содержит все возможные структуры, подготовленные для сопоставления со следующим словом<sup>1)</sup>.

```

t := предложение;
цикл: l := 0; c := следующее слово;
rl: if t ≠ 0 then
  begin s := hd (l); t := tl (l);
    if atom (hd (s)) then
      begin if одинаковые слова (hd (s), c) then
        begin if tl (s) = 0 then go to правильное;
          l := cons (tl (s), l)
        end;
      end
    else
      begin u := hd (s); s := tl (s);
        r2: if u ≠ 0 then
          begin t := cons (присоединить (hd (u), s), t);
            u := tl (u);
            go to r2
          end
        end;
      end;
    go to rl
  end;
if l = 0 then go to неправильное;
t := l;
go to цикл;

```

<sup>1)</sup> Выход по метке *правильное* означает, что предложение правильное, а по метке *неправильное* — что предложение неправильное; предполагается, что описана логическая функция *одинаковые слова*. — Прим. перев.

Одной из важнейших особенностей системы обработки списков является применяемый в ней метод размещения списковой структуры в машинной памяти. Очень прост последовательный способ отведения машинной памяти нуждающимся в ней программам, когда ячейки берутся от начала свободной области, а затем это начало сдвигается. Однако большинство задач обработки списков требует больше памяти, чем можно получить при таком прямолинейном способе, и нужно предпринимать что-то для того, чтобы обеспечить возможность повторного использования той части памяти, содержимое которой больше не понадобится. Экономичность обработки списков зависит от возможности такого гибкого использования памяти, чтобы хранились только те данные, которые могут понадобиться. Все языки обработки списков обеспечивают возможность повторного использования памяти посредством либо явных указаний на языке, либо алгоритмов, заложенных в систему.

Простейшим является метод, который применяется в некоторых программах, работающих в режиме реального времени, и в других случаях, когда важна скорость перераспределения памяти, а структуры просты. Вся память, предназначенная для использования под списки, организуется перед началом работы в единый список, называемый свободным списком. Если нужно новое место в памяти, то оно берется в начале свободного списка, а указатель начала свободного списка продвигается. Если пользователь знает, что он завершил работу с некоторыми ячейками, то он информирует об этом систему, которая присоединяет эти ячейки к свободному списку, подготовив их к повторному использованию.

Предположим, что исходный свободный список размещен в памяти следующим образом. (Слово СВОБОДНЫЙ указывает начало свободного списка.)

100	0,0	
101	0,100	
102	0,101	
103	0,102	
104	0,103	
105	0,104	
106	0,105	СВОБОДНЫЙ



Если выполнить операции

$cons(cons(cons(a, 0), cons(b, 0)), 0)$

то память будет заполнена следующим образом:

100	0,0	
101	0,100	
102	0,101	СВОБОДНЫЙ
103	104,0	
104	106,105	
105	b,0	
106	a,0	

Первая операция сформировала  $cons(a, 0)$ ; нужная для этого ячейка берется из начала свободного списка. Следующая операция сформировала  $cons(b, 0)$ , используя ячейку из нового начала свободного списка. Ячейка 104 содержит результат применения функции  $cons$  к содержимому ячеек 106 и 105. Ячейка 103 содержит ответ. Теперь свободный список начинается в ячейке 102.

Если составные части сформированного списка возвращаются в свободный список в некотором произвольном порядке, то содержимое памяти может стать таким:

100	0,0	
101	0,100	
102	0,101	
103	0,104	СВОБОДНЫЙ
104	0,105	
105	0,106	
106	0,102	

Таким образом, свободный список скоро становится запутанным. Это не имеет ни малейшего значения с точки зрения машинной реализации, но из этого следует, что печать содержимого памяти обычно не оказывает практической помощи при поиске ошибки в программе.

В системе с режимом реального времени ячейка (элемент списка) обычно занимает несколько машинных слов. Память организуется в свободный список, состоящий из блоков соответствующего размера, причем каждый блок содержит указатель на следующий блок. Эти блоки будут содержать как числовые данные, так и указатели.

Для того чтобы эта простая схема работала, должны выполняться два требования. Блоки памяти должны иметь постоянный размер, и пользователь должен иметь возможность сообщать системе, что определенная ячейка больше никогда не понадобится. Ни одно из этих ограничений нельзя считать полностью приемлемым. С первым требованием, по-видимому, легче примириться, так как можно составлять ячейки различных размеров, используя для этого списки, состоящие из ячеек одного размера. Однако при этом теряется удобство индексации в пределах одного блока и требуются значительные затраты памяти для размещения указателей, связывающих ячейки при формировании блоков. Данное ограничение налагается во многих схемах обработки списков.

Второе ограничение менее удобно для пользователей. Требуется сообщать системе, что не будет использоваться определенная ячейка, а не определенный список. Если мы запрограммировали

$l := cons(cons(a, b), 0);$

$m := l;$

и через некоторое время решаем, что список  $l$  больше никогда не будет использоваться, то нет возможности вернуть системе те ячейки, из которых он состоит, если будет использоваться содержащий их новый список  $m$ . Прежде чем вернуть ячейку в свободный список, необходимо удостовериться в том, что она не используется ни в одном из тех списков, которые могут еще понадобиться. Гораздо более трудные ситуации могут возникать в языках, разрешающих применение общих подсписков. Если в рассмотренном примере  $b$  — список и если в некоторый момент выясняется, что список  $b$  больше не требуется, то нельзя возвращать его в свободный список, если еще может понадобиться какой-нибудь из списков  $l$  и  $m$ , частью которых он является. Необходимость в такой предосторожности может оказаться весьма затруднительной для программистов, предпочитающих не знать, как их списки организуются и используются внутри машины. Для того чтобы облегчить пользователю запоминание нужных ячеек, принимается соглашение объявлять список собственным только в одном месте. Если он используется в других местах, то считается, что он заимствуется. Список-собственник отвечает за то, чтобы вовремя объявлять об освобождении своих ячеек.

В оставшейся части этой главы описываются некоторые способы частичного или полного устранения двух описанных недостатков простой и быстрой системы ценой некоторой потери скорости или введения дополнительных соглашений.

Два основных способа избежать описания ненужных ячеек состоят в том, чтобы заставить пользователя описывать либо ненужные списки, либо нужные списки. При первом способе пользователь должен, как и раньше, сообщать системе, какие списки не нужны, но это означает, что ячейки, из которых они состоят, могут быть возвращены в свободный список только тогда, когда система установит, что все списки, в которых используются эти ячейки, описаны как ненужные. При втором способе система выясняет, какие ячейки принадлежат спискам, которые объявлены нужными и поэтому могли бы быть использованы. Остальные ячейки могут быть помещены в свободный список. Первый вариант системы может работать постепенно<sup>1)</sup>. Второй вариант обычно работает в особые моменты, так как ненужные ячейки возможно обнаружить только путем перебора всех нужных ячеек, а этот процесс очень медлен. В первом случае все-таки требуется, чтобы пользователь думал о ненужных списках; во втором случае необходимая информация может поступать автоматически (описания нужных списков могут подразумеваться неявно).

#### ОПИСАНИЕ НУЖНЫХ СПИСКОВ

Если система больше не может поставлять новые ячейки из свободного списка, то она должна попытаться получить обратно часть памяти, занятую ячейками, которые не понадобятся в дальнейшем. Описываемый ниже способ применяется в системе ЛИСП. Все ячейки, к которым может обратиться пользователь, должны находиться в списках, определяемых известными заголовками списков; в заголовках списков система получает описание всех действующих списков. Ячейка, на которую не ссылаются действующие списки, может быть возвращена в свободный список. Однако выявить ячейки, не относящиеся к действующим спискам, возможно только путем перебора всех ячеек из этих списков в предположении, что остальные ячейки не нужны. Поэтому такой процесс сборки мусора должен выполняться в два этапа. На первом этапе просматриваются и отмечаются как нужные все списки, определяемые заголовками списков. Затем на втором этапе просматривается все поле, предназначенное для списков, и неотмеченные ячейки возвращаются в свободный список; одновременно отмеченные ячейки становятся неотмеченными, т. е. подготавливаются для последующей сборки мусора. Разумеется, если нет ни одной неотмеченной ячейки, то это означает, что система переполнила память и программа должна прекратить работу. Процесс отыска-

ния нужных ячеек достаточно медлен из-за многочисленности ячеек, содержащихся обычно в нужных списках. Поэтому желательно производить сборку мусора как можно более редко, только когда к этому вынуждает нехватка памяти.

Ячейка может быть отмечена как нужная с помощью бита из машинного слова, занятого этой ячейкой, или же соответствующий признак может храниться в памяти отдельно. В обоих случаях процесс просмотра списков и проставления отметок выполняется по следующей процедуре. Для каждого указателя рассматривается указываемая им ячейка, и если она уже отмечена или содержит атом, то над ней не производится никаких действий; в противном же случае она отмечается, а затем рассматриваются таким же способом все содержащиеся в ней указатели. Этот процесс представляет собой обычную рекурсивную процедуру<sup>1)</sup>.

```
begin integer j; list h;
  procedure пройти (l); list l;
    fl: if not atom (l) and not отмечен (l)
      then begin отметить (l); пройти (hd (l)); l := tl (l);
            go to fl
          end;
    rl: if оглавление ≠ 0 then
      begin h := hd (оглавление); оглавление := tl (оглавление);
            go to rl end;
  свободный список := 0;
  for j := 1 step 1 until размер поля списков do
    if отмечен (поле списков [j])
      then переметить (поле списков [j])
    else свободный список := добавить (свободный список, поле
                                         списков [j])
  end;
```

Благодаря тому что предусмотрено отсутствие проверки тек списков, которые уже отмечены, этот метод без осложнений применим к циклическим спискам. Заметим, что мы исключили из рассмотрения действия, которые должны выполняться над атомами.

<sup>1)</sup> В описании процедуры *пройти* функция *отмечен (l)* принимает значение true, если список *l* отмеченный; процедура *отметить (l)* делает список *l* отмеченным; процедура *переметить (l)* делает список *l* неотмеченным; *оглавление* — список, состоящий из заголовков всех действующих списков; процедура *добавить* применяется для добавления к свободному списку очередной ячейки из поля списков; *and* означает алгольный символ  $\wedge$ , *not* — символ  $\neg$ . — *Прим. перев.*

<sup>1)</sup> То есть параллельно основному процессу счета. — *Прим. перев.*

Недостатком этого алгоритма является то, что из-за своей рекурсивности он использует память для запоминания промежуточных результатов. Поскольку нет возможности предсказать заранее, сколько для этого потребуется памяти, и нельзя брать для этого память из области списков, то могут возникнуть трудности с обеспечением соответствующего рабочего поля. Существует другой алгоритм, который осуществляет сборку мусора без применения дополнительной памяти, но использует часть проверяемых ячеек для запоминания необходимой текущей информации. При этом для каждой ячейки требуется столько же признаков, сколько указателей содержится в одной ячейке, тогда как в описанной системе требовался единственный признак. Читатель увидит, что эта программа несколько напоминает итеративный вариант программы копирования списков, и копирование могло бы производиться по такому способу, если бы допускались отмеченные списки. Следующая программа рассчитана на то, что каждая ячейка содержит два указателя<sup>1)</sup>.

```

t: if not atom (l) and not отмечен (l, 1) and not отмечен (l, 2) then
begin отметить (l, 1);
  mem := hd (l);
  sethd (l, n);
  n := l;
  l := mem;
  go to t
end
else
s: if отмечен (n, 1) and not
отмечен (n, 2) then
begin mem := tl (n);
  settl (n, hd (n));
  sethd (n, l);
  l := mem;
  go to t
end
else
if n = 0 then go to выход
else begin mem := tl (n);
  settl (n, l);
  l := n;
  n := mem;
  go to s
end;
выход:

```

<sup>1)</sup> Процедура отметить (*l*, 1) присваивает ячейке *l* признак 1; функция отмечен (*l*, 1) принимает значение true, если ячейка *l* имеет признак 1. — Прим. перев.

При просмотре списка нужно помнить указатель предыдущего элемента. Для каждой данной точки списка переменная *n* содержит указатель предыдущего элемента, а переменная *l* содержит указатель текущего элемента. Если очередная ячейка не отмечена, то необходимо просмотреть список от начала элемента *l* и поэтому переменной *l* присваивается значение *hd* (*l*), а переменной *n* — прежнее значение *l*. Так как мы собираемся запомнить в *l* указатель прежнего начала *l*, то прежнее значение *n* может быть запомнено в начале ячейки *l*. Аналогично после просмотра начала ячейки, которая оказалась отмеченной, необходимо просмотреть продолжение, и продолжение может быть использовано для хранения текущей информации. Программа сборки мусора в отличие от обычных программ может отмечать и изменять содержимое ячеек, так как все процессы прерываются до тех пор, пока не завершится сборка мусора.

Этот метод работает значительно медленнее, чем простая рекурсивная программа, но к нему можно прибегать в тех случаях, когда для простой программы возникают проблемы нехватки памяти.

Другой способ накопления освободившейся памяти позволяет обойтись без применения реального списка свободных элементов. После того как все ненужные элементы стали отмеченными, не строится никакого списка свободных ячеек. Вместо этого в некоторый индекс заносится адрес нижней ячейки поля списков. Если требуется новая ячейка (например, для подпрограммы *cons*), то производится поиск начиная с того места, которое указано в индексе, до обнаружения такой ячейки, которая отмечена как ненужная; затем эта ячейка используется. Значение индекса продвигается таким образом, чтобы в следующий раз бралась ячейка, находящаяся над найденной. Когда значение индекса становится равным адресу самой верхней ячейки, выполняется новый процесс опознавания ненужных элементов. Этот метод не короче и не эффективнее, чем предыдущий; он только иначе распределяет затрачиваемое время. В самом деле, время, которое уходило на формирование свободного списка путем поиска области ненужных элементов, затрачивается теперь постепенно по мере возникновения надобности в новых элементах. Этим частично устраняется один из недостатков данной группы методов, состоящий в необходимости прерывания процессов на время выполнения сборки мусора. Такая необходимость не создает трудностей для машин, используемых в автономном режиме работы, но если машина применяется в оперативном режиме, то может понадобиться быстро получать места в памяти в моменты, не предсказуемые заранее. Возникновение пауз на время сборки мусора (возможно, порядка одной десятой секунды) исключает такой быстрый захват памяти.

Следует помнить, что системе должны быть известны все заголовки списков. Некоторые из них могут быть описаны для системы пользователем, а другие могут неявно подразумеваться в программе. Например, оператор

`cons(cons(a, b), cons(c, d))`

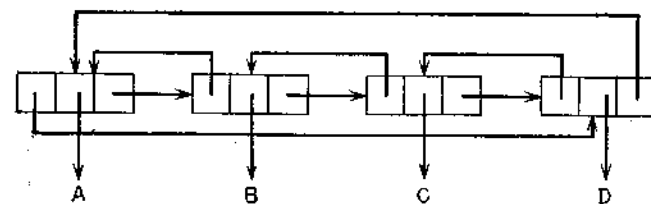
требует, чтобы подпрограмма `cons` использовала три новые ячейки. Вычисление может происходить в следующем порядке. Сначала формируется и запоминается `cons(a, b)`, затем формируется `cons(c, d)`, и, наконец, из этих двух промежуточных результатов формируется окончательный результат. Если в памяти имеется место только для выполнения первой промежуточной операции, но не для второй, то при втором обращении к подпрограмме `cons` будет работать программа сборки мусора. При этом система не должна занимать память, использованную под `cons(a, b)`. По сути дела, промежуточный результат операции должен быть известен системе как заголовок списка. Это относится не только к процедурам типа `cons`, которые являются частью системной библиотеки, но также и к процедурам, которые написаны самим пользователем, если описание параметров этих процедур может потребоваться системе в моменты, не контролируемые пользователем. К сожалению, очень трудно бывает обнаружить любое упущение, касающееся защиты информации от программы сборки мусора.

Если система может различать, какие величины в машинной памяти представляют числовые данные, а какие — списки, то система способна выяснить, какие элементы являются заголовками списков. В этом случае программист не должен описывать нужные списки. Такая автоматическая система может оказаться очень удобной. Она могла бы быть реализована с помощью некоторого аппаратного устройства, хранящего дополнительные разряды для каждого машинного слова, позволяющие определять тип информации, содержащейся в этих словах. Однако обычно эти цели достигаются только программной интерпретацией ценой соответствующей потери в скорости.

## ОПИСАНИЕ НЕНУЖНЫХ СПИСКОВ

Другой метод сборки мусора применяется в системе СЛИП, которая включена в ФОРТРАН как набор подпрограмм [28]. В этом случае в каждой ячейке используются три указателя: один обозначает атом или подсписок, другой показывает продолжение списка, а третий показывает на предыдущую ячейку списка. С каждым набором элементов списка связаны некоторые дополнительные ячейки, которые используются для реги-

страции. Список может быть представлен следующим образом:



Поскольку в списке расставлены указатели в обоих направлениях, каждая ячейка может быть частью только одного списка. В ячейках регистрации формируется имя списка, и через них проходят все обращения к данному списку. Предназначены они для того, чтобы обеспечивать доступ к обоим концам списка, а также для подсчета числа ссылок на этот список. Если список превращается в подсписок другого списка, счетчик автоматически продвигается.

Из памяти организуется свободный список. Когда программист решает, что он больше не собирается использовать какой-то список, он сообщает об этом системе. При этом список не обязательно включается в свободное пространство, а выполняется проверка, не нужен ли еще список где-нибудь как подсписок, т. е. равен или не равен нулю счетчик ссылок. Если счетчик не равен нулю, то список еще нужен; поэтому он не ликвидируется, а из счетчика вычитается единица. Если же счетчик равен нулю, то список больше нигде не используется и его ячейки могут быть возвращены в свободный список. Свободный список присоединяется к одному концу ненужного списка, а другой конец становится новым началом свободного списка. Поскольку оба конца списка доступны через управляющие слова, эта операция может быть выполнена быстро.

После того как с помощью этой процедуры были возвращены ненужные ячейки, свободный список содержит указатели на списки, которые были когда-то подсписками ликвидированного списка. Можно было бы теперь уменьшить их счетчики ссылок на единицу и, если они окажутся ненужными, вернуть их в свободный список. Но вместо этого такие списки сохраняются до тех пор, пока ячейки со ссылками на них не будут затребованы из свободного списка. На этом этапе производится проверка указанных списков. Преимущества этого изящного метода состоят в следующем. Производится некоторая экономия времени за счет того, что ненужный список просматривается не более одного раза. Сборка мусора выполняется только по необходимости, когда требуется место в памяти. Процесс сборки мусора протекает постепенно, параллельно с работой

основной программы, не требуя обязательного прерывания остальных процессов.

Циклические списки не могут быть ликвидированы.

### СБОРКА МУСОРА И ВЕКТОРЫ

Основная трудность применения блоков памяти переменной длины состоит в том, что это может вынудить систему перемещать информацию внутри машины, если все время требуется использовать память оптимальным образом. Такой процесс может оказаться очень медленным и неудобным.

Предположим, что вначале поле списков пусто и что по мере того, как требуются новые блоки памяти, они берутся от начала пустого поля. Когда это поле заполнено и требуется размещать новые блоки, приходится использовать память, освободившуюся из-под блоков, которые уже не нужны. Определение нужных и ненужных ячеек может быть выполнено с помощью процесса, аналогичного любому из методов, описанных только что для случая блоков постоянной длины. Возможность узнать, какие блоки могут быть использованы заново, имеется, однако, хотя область доступной освободившейся памяти в целом может оказаться достаточной для создания новых нужных блоков, она может не содержать непрерывных участков достаточной длины. Если это так и если тем не менее не удастся отказаться от программ, вызывающих такие затруднения, то нет другого выхода, кроме перемещения некоторых нужных данных, чтобы ненужные части памяти оказались собранными вместе.

Перемещение информации является медленным процессом. В сущности одна из причин введения обработки списков состоит в том, что она позволяет избегать такого перемещения. Этот процесс должен выполняться без прерываний, и поэтому он выводит систему из действия до своего завершения. Кроме того, он влечет за собой обновление всей информации, содержащей ссылки на перемещенные блоки.

Последнее неудобство может быть сведено к минимуму, если воспользоваться следующим способом, также не свободным от недостатков. В памяти имеются два поля. На одном поле размещаются блоки переменной длины. Другое поле разбито на ячейки одинакового размера с применением свободного списка. Если все ссылки на блоки переменной длины осуществляются через особые элементы поля с постоянным размером ячеек, то можно ограничиться обновлением только этих одиночных элементов. Тем не менее это не решает проблемы неэффективности перемещения блоков.

В этой главе кратко рассматриваются языки IPL-V [21, 22, 23], ЛИСП [17, 18], СЛИП [28], FLPL [11] и КОМИТ [7, 31]. Более подробную информацию можно извлечь из указанных в библиографии руководств и статей. Сравнение языков ЛИСП, СЛИП и КОМИТ произведено в статье [4], содержащей краткую таблицу основных свойств этих языков.

Для каждого из этих языков мы ответим по мере возможности на следующие вопросы:

1. Какую структуру имеет сам язык? Она может варьироваться от подобия машинного кода до обычного языка программирования вроде АЛГОЛа или ФОРТРАНа. Структура языка очень важна для пользователя, так как она влияет на его способ мышления больше, чем что-либо другое.
2. Каков размер имеющейся библиотеки подпрограмм?
3. Доступны ли пользователю простейшие операции над списками?
4. Фиксировано ли в системе представление списков или же пользователь имеет возможность строить новые представления?
5. Какой метод сборки мусора применяется и в какой степени ответственность за сборку мусора возлагается на пользователя? Возможно требование, чтобы пользователь описывал нужные или ненужные списки. Могут быть и другие ограничения.
6. Легко ли пользоваться рекурсией?
7. Насколько легко описываются арифметические действия? Имеется ли большая библиотека стандартных подпрограмм численного анализа?
8. Какие стандартные обозначения списков используются программистом? Имеются ли подпрограммы чтения и печати списков в явном виде?
9. Предусмотрены ли стандартные методы представления списков во вспомогательной памяти?
10. Насколько быстро получается программа?
11. Используется ли интерпретатор или компилятор?
12. В каком виде программы хранятся в машинной памяти? Они сами могут быть представлены в форме списков. В таком

случае возможно изменять программы, а также генерировать и подключать новые программы с помощью операций над самой программой.

13. Допускаются ли общие подписки?

14. Допускаются ли циклические списки?

## IPL-V

Система IPL-V является интерпретатором. Она включает ведущую программу, которая последовательно рассматривает инструкции из программы пользователя и выполняет для каждой инструкции указанные в ней операции. Машинная аппаратура не производит непосредственного выполнения инструкций. Инструкции не хранятся в последовательных ячейках машинной памяти, а образуют списки, каждый из которых рассматривается как подпрограмма. Интерпретатор выполняет последовательно инструкции из списка, каждая из которых может указывать одну из основных операций системы или же имя другой подпрограммы, которую нужно выполнить. Поскольку элементами программы являются списки, они могут обрабатываться этой же программой; таким образом, возможно самоизменение.

Ячейки системы IPL-V содержат по два указателя, а также некоторую дополнительную информацию, описывающую указатели. Для данных и для программы применяется одинаковый формат. Два указателя обозначают элемент и следующую ячейку из списка и называются Symb и Link соответственно<sup>1)</sup>. Описательная информация содержится в двух частях ячейки, называемых P и Q. Числа и символы хранятся в ячейках с особыми значениями P и Q.

Принятая запись на бумаге отличается от всего описанного ранее в этой книге. Каждый список имеет имя, которым он обозначается, и состоит из последовательности имен. Эти имена обозначают либо другие списки, либо атомы. Так было бы, если бы вместо записи

(A, B ((C, D), E, F), G)

мы писали

$l = (A, B, x, G)$

$x = (y, E, F)$

$y = (C, D)$

представляя каждый уровень отдельно. Эта запись может дать достаточно информации о структуре внутри машинной памяти, для того чтобы выявить общие подписки, не слишком затемняя

<sup>1)</sup> Символ и связь. — Прим. перев.

представление списков. На самом деле применяемая запись несколько отличается от приведенной выше. Предусмотрены специальные соглашения относительно формирования имен, которые могут применяться. Имена, начинающиеся с цифры 9, за которой следует какое-нибудь число, являются локальными и определены только в пределах отдельного списка. Имена, начинающиеся с буквы, за которой следует число, являются региональными и могут быть использованы в любом месте. Типичный список мог бы выглядеть следующим образом:

Имя	PQ	Symb	Link
L2		9—3	
		9—1	
		G300	0
9—3		0	
		A1	
		V1	
		A2	0
9—1		0	
		S22	
		9—2	0
9—2	1		12

Имя описываемого списка находится в первой колонке. Для элементов списка колонки P и Q пусты. В колонке Symb указывается элемент списка, а колонка Link предназначена для указания следующей ячейки, если она не описана непосредственно за данной. Значение нуль в колонке Link является признаком конца списка. Список с именем L2 содержит элементы 9—1 (локально определенный список) и G300 (региональная величина). Локальный список 9—1 содержит региональную величину S22, а также локальную величину 9—2. Значением величины 9—2 является число 12. Список 9—3, являющийся первым элементом в L2, несет несколько иную функциональную нагрузку. Любому списку может соответствовать описательный список, который запоминается в качестве его первого элемента. Список 9—3 является описательным списком для L2. Для списка 9—1 описательный список пуст. Описательный список состоит из некоторого числа пар свойств и соответствующих им значений. Например, свойству «Имя» могло бы соответствовать значение «Джон». Предусмотрены подпрограммы для отыскания значения, соответствующего заданному свойству, для изменения значений, а также для добавления и исключения пар.

В каждой инструкции колонка Symb содержит имя, являющееся аргументом этой инструкции. Колонка Q из той же ячейки содержит информацию о том, как следует интерпретировать содержимое колонки Symb при вычислении имени S. Если  $Q = 0$ , то S — содержимое Symb, если  $Q = 1$ , то S — имя из той ячейки, имя которой указано в Symb, а если  $Q = 2$ , то S — имя из той ячейки, имя которой указано в ячейке с именем из Symb. Признак P указывает, какую из восьми возможных инструкций нужно выполнять в соответствии со следующей таблицей:

0	ВЫПОЛНИТЬ S	S — имя одной из основных подпрограмм, предусмотренных в системе, или имя одной из собственных подпрограмм пользователя.
1	ВВЕСТИ S	Копия S помещается на вершину магазинного списка, называемого H0.
2	ВЫДАТЬ В S	Содержимое вершины списка H0 помещается в S, и список H0 продвигается вверх.
3	ВОССТАНОВИТЬ S	Магазинный список для S продвигается вверх <sup>1)</sup> .
4	СОХРАНИТЬ S	Копия символа из S помещается в магазинный список для S.
5	ЗАМЕНИТЬ H0 НА S	Копия S помещается на место списка H0.
6	СКОПИРОВАТЬ H0 В S	Копия H0 помещается на место S.
7	УХОД НА S ЕСЛИ H5 ЕСТЬ «—»	Если H5 есть «—», то S указывает следующую ячейку, подлежащую интерпретации; если H5 есть «+», то следующая ячейка указана в столбце Link.

Организация IPL основывается на применении магазинных списков. Адрес ячейки, содержащей текущую инструкцию, хранится в H1. Если эта инструкция указывает, что нужно выполнить подпрограмму, то необходимо запомнить адрес для возврата после завершения выполнения подпрограммы. Это организуется с помощью магазинного списка для H1. Перед тем как инструкция передаст управление на указанную подпрограмму, производится продвижение текущего содержимого H1 в магазинный список для H1. После окончания работы подпрограммы

<sup>1)</sup> С занесением прежнего верхнего элемента на место. — Прим. перев.

верхний элемент списка выдвигается обратно в H1. Это означает, что подпрограмма может использоваться рекурсивно без затруднений в отношении места возврата. Обычно подпрограмме требуются параметры. Они сообщаются ей через магазинный список H0. Каждая подпрограмма выбирает свои аргументы из H0 и оставляет свои результаты также в H0. Для запоминания промежуточных результатов при каждой программе предусматривается группа из десяти ячеек. Для них записываются списки W0, ..., W9, и каждый процесс, использующий эти рабочие ячейки, должен предварительно продвинуть их текущие состояния в магазинные списки, а после завершения своей работы восстановить их.

Система IPL-V включает примерно 150 основных процессов. Этот набор не является минимальным, а содержит много операций, которые часто бывают нужны. Соответствующие подпрограммы разделяются на восемь классов:

1. Общие. Такие операции, как останов, засылка в H5 знака «+».
2. Ввод-вывод.
3. Списковые. Сюда относятся такие процессы, как включение символа в список, копирование списка.
4. Организация общей рабочей памяти. Для облегчения использования списка H0.
5. Процессы над описательными списками.
6. Арифметические операции.
7. Обработка предварительной информации. Сюда относятся операции проверки типа имен.
8. Работа со вспомогательной памятью. Операции использования внешней памяти.

Для организации повторяющихся процессов предусмотрен класс операций, называемых генераторами. Они применяют заданный процесс к каждому аргументу из некоторого набора.

Программист отвечает за своевременное объявление того, что какой-либо список ему больше не понадобится. Соответствующий список возвращается в свободный список, из которого получают новые ячейки. Чтобы облегчить пользователю прослеживание нужных ему списков, последние рассматриваются как собственность определенных списковых структур, которые вправе уничтожить их. Задержка в уничтожении списков приводит к потере памяти.

## ЛИСП

Система ЛИСП включает как компилятор, так и интерпретатор. Программы, изготовленные компилятором, выполняются в 10—100 раз быстрее, чем при интерпретации, и используют меньше памяти.

Обращение ко всем данным производится как к S-выражениям (символическим выражениям). S-выражения представляются на бумаге в той записи, которая применялась во всех предыдущих главах этой книги. Атом представляется в виде строки не более чем из тридцати прописных букв и цифр, начинающейся с буквы. Скобки и запятые применяются для указания структуры, причем запятые могут отбрасываться. Следующие два S-выражения представляют один и тот же список:

((РОБИНСОН, ДЖОН), (СМИТ, ДЖО))

((РОБИНСОН ДЖОН) (СМИТ ДЖО))

Внутри машинной памяти данные представляются в виде ячеек, содержащих по два указателя. В принципе один из них предназначен для указания элементов списка, а другой — для указания адреса следующей ячейки из того же списка, но указатели атомов могут помещаться в обеих частях. Для обозначения конца списка используется специальный атом ПУСТО<sup>1)</sup>, который может, впрочем, применяться и для других целей.

Программы записываются на бумаге в виде M-выражений (метавыражений), которые переводятся программистом в S-выражения до того, как вводятся в машину. Интерпретатор использует внутреннее представление программ, согласующееся с форматом данных, так что можно писать самоизменяемые программы.

Программа имеет функциональную форму. Она не состоит из набора инструкций или операторов, подлежащих последовательному выполнению, а является функциональным выражением, которое должно быть вычислено. Выражением является, например, следующая запись:

cons[cons[A; B]; ПУСТО]

Для того чтобы конструировать интересные программы, программист должен иметь возможность описывать для себя функции, выражать условия и некоторым способом повторять операции. Повторение осуществляется с помощью рекурсии, которая является основным способом программирования на языке ЛИСП.

На метаязыке условные выражения формируются так:

$[p_1 \rightarrow e_1; p_2 \rightarrow e_2; \dots; p_n \rightarrow e_n]$

Каждая переменная  $p$  принимает значение *истина* или *ложь*, а каждый символ  $e$  означает выражение, которое является одним из возможных значений всего условного выражения. Вычисляется значение первого  $p$ , и если оно — истина, то первое

<sup>1)</sup> В оригинале NIL. — Прим. перев.

$e$  является значением всего выражения. В противном случае вычисляется второе  $p$ , и если оно — истина, то берется соответствующее выражение. Если ни одно значение  $p$  не оказывается истиной, то выражение не определено. Значения  $e_i$ , соответствующие тем  $p_i$ , которые находятся за первым истинным  $p$ , не вычисляются, как и те  $e_i$ , которые соответствуют ложным  $p_i$ . Для описания функции, аналогичной *mem*, которая проверяет, встречается ли где-нибудь в списке определенный атом, мы применяем рекурсивную запись, идентичную по значению записи из гл. 3.

$mem[a; l] = [atom[l] \rightarrow eq[a; l]; mem[a; car[l]] \rightarrow T;$

$T \rightarrow mem[a; cdr[l]]$

Описание *mem* является условным выражением с тремя  $p_i$  и тремя  $e_i$ . Функции *car* и *cdr* соответствуют функциям *hd* и *tl* (эти названия образованы от наименований частей слов для машины IBM 709). Атом  $T$  имеет значение *истина*. Заметим, что одно из трех значений  $p_i$  обязательно окажется истиной, поскольку последнее  $p$  само является  $T$ .

В действительности эта запись не применяется для указания того, какие переменные в выражении являются параметрами. Для этого используется запись, основанная на  $\lambda$ -исчислении Чёрча. Вместо того чтобы давать приведенное выше описание *mem*, мы указываем параметры  $a$  и  $l$  следующим образом:

$mem = \lambda[a; l]; [atom[l] \rightarrow eq[a; l]; mem[a; car[l]] \rightarrow T;$

$T \rightarrow mem[a; cdr[l]]$

Необходимо также указать, что слово *mem* внутри описания означает то же самое, что и описываемое слово *mem*. Для этого в аналогичной записи применяется слово *label*. Правильное описание функции *mem* в виде M-выражения имеет следующий вид:

$label[mem; \lambda[a; l]; atom[l] \rightarrow eq[a; l]; mem[a, car[l]] \rightarrow T;$

$T \rightarrow mem[a; cdr[l]]$

Предусмотрено много встроенных в систему функций, обеспечивающих выполнение общеупотребительных операций, а также примитивных операций. В их число входят некоторые арифметические операции. Числа рассматриваются как некоторый вид атомов и могут употребляться в S-выражениях.

Рекурсия не является единственным способом организации повторений в программах. Если рекурсия оказывается неудобной или нежелательной, то можно пользоваться методом представления программ в виде последовательности операторов.

Если программист пишет программу в M-выражениях, то он должен перевести ее в S-выражения прежде, чем пропускать.



ее на машине. Перевод порождает неудобочитаемое обилие скобок. Ниже описывается процесс перевода.

1. Имя функции, имя переменной или атом записываются прописными буквами.
2. Константа  $x$  переводится в вид (QUOTE  $X$ ).
3. Форма  $f[a; b; \dots; n]$  переводится в вид (FAB ...  $N$ ).
4. Из условного выражения получаем (COND (P1 E1) (P2 E2) ... (PN EN)).
5.  $\lambda[x; y; z]; e$  переводится в вид (LAMBDA (XYZ) E).
6.  $label[n, e]$  переводится в вид (LABEL NE).

Приведенное выше M-выражение для функции *mem* переводится в вид

```
(LABEL MEM (LAMBDA (A L) (COND ((ATOM L) (EQ AL))
  ((MEM A (CARL (QUOTE T)) ((QUOTE T)) (MEM A (CDRL)))))))
```

Описанные функции вычисляются только для получения их значений и не изменяют существующих списков. Существуют функции, которые изменяют списки и вычисляются для воздействия на объекты. Функции *rplaca* $[x; y]$  и *rplacd* $[x; y]$  соответствуют функциям, названным нами *sethd* $(x, y)$  и *settl* $(x, y)$ . Эти названия образованы от частей слова для машины IBM 709.

Атомы представляют собой специально отмеченные списки, которые называются списками собственности. Они состоят из пар указателей и значений. Например, указателю PNAME соответствует в качестве значения имя данного атома, которое выводится на печать.

Использование внешней памяти не предусматривается.

Сборка мусора является автоматической. Система сама знает заголовки всех нужных списков, так как они представлены в базовых регистрах, содержащих список всех атомных символов, в регистрах, содержащих промежуточные результаты, и в других регистрах.

## СЛИП

Система СЛИП состоит из набора подпрограмм, включенных в систему ФОРТРАН. Это означает, что доступны все арифметические возможности ФОРТРАНа наряду со всем огромным множеством существующих программ на языке ФОРТРАН. Несмотря на то что СЛИП — компилятор, а не интерпретатор, обработка списков производится не очень быстро, потому что операции выполняются путем вызова подпрограмм, а не с помощью включения в программу скомпилированных групп кодов.

Доступны фортрановские программы ввода и вывода, а следовательно, и принятое в ФОРТРАНе использование внешней памяти.

Структуры программ и данных в системе СЛИП являются совсем разными, и поэтому нет возможности организовать в программах самоизменение. В этой системе предусмотрено много специальных подпрограмм, в некоторых реализациях их около ста двадцати.

Каждая ячейка состоит из двух машинных слов для IBM 7090 и содержит три элемента. Два из них всегда являются указателями и показывают предыдущую и следующую ячейки списка. Третий элемент может быть числом или еще одним указателем, на этот раз показывающим элемент списка. Поскольку в ячейках указываются оба направления по списку, любая ячейка может содержаться только в одном списке. Однако разрешается, чтобы один подсписок использовался во многих различных списках. У каждого списка имеется заглавная ячейка, через которую с ним осуществляется связь. Все указатели на список, по сути дела, указывают его заголовок. Заголовок содержит в качестве трех элементов информации указатели начала и конца списка, а также число, являющееся счетчиком использований данного списка как подсписка. (Счетчик нужен для программы сборки мусора.) Каждый элемент списка — либо число, либо указатель заглавного слова какого-то списка.

Пользователю доступны простые операции. Самыми элементарными из этих операций являются следующие:

ID(C)	Результатом является характеристика ячейки C, занимающая два бита и показывающая вид соответствующего элемента.
LNKL(C)	Результатом является левый указатель из ячейки C.
LNKR(C)	Результатом является правый указатель из ячейки C.
SETDIR(I, L, R, C)	I заносится в поле характеристики, L — на место левого указателя и R — на место правого указателя в ячейке C.
STRDIR(D, C)	Данные D заносятся в ячейку C.
CONT(A)	Результатом является содержимое ячейки с машинным адресом A.
INHALT(A)	Результатом является содержимое машинного слова с адресом A.

Введение некоторых из этих операций потребовалось для того, чтобы преодолеть трудности, связанные с нежелательными свойствами ФОРТРАНа. Например, функции CONT и INHALT

применяются для того, чтобы избежать условностей, связанных с фиксированной и плавающей запятой в числах.

В качестве свободного списка используется список, называемый LAVS (список доступной памяти). Метод сборки мусора описан в гл. 6.

Используя простые операции, можно создавать циклические списки, но нельзя вернуть циклические списки в LAVS, не нарушив предварительно цикличность.

Применение рекурсии затруднено и требует от программиста специальной обработки параметров и результатов.

### FLPL

Система обработки списков FLPL состоит из набора подпрограмм на ФОРТРАНе. Ее название образовано из первых букв английской фразы «ФОРТРАН — язык обработки списков». Структуры списков напоминают те структуры, которые приняты в IPL-V. Каждая ячейка содержит два указателя или представляет собой слово данных. Тип ячейки указывается в описательном поле, состоящем из двух разрядов. Один указатель показывает элемент, а другой — следующую ячейку из того же списка. Метод сборки мусора основан на том, что пользователь указывает ненужные ячейки. Чтобы облегчить пользователю запоминание нужных ячеек, в каждой ячейке используется один разряд, в котором может быть указано, что данный элемент принадлежит этому списку<sup>1)</sup>. Если какой-то элемент не принадлежит списку, то он не должен стираться этим списком.

Приведем примеры подпрограмм.

XCDRF(J)	Результатом является один из указателей из слова J.
XSTORDF(J, K)	Устанавливает один из указателей из ячейки J равным K.
XLASLCF(J)	Находит последнюю ячейку в списке J.
XWORDF(J)	J запоминается в новой ячейке, извлеченной из свободного списка.
XTOERAF(J)	Стирает весь список J, включая любые подсписки или слова данных, принадлежащие списку J.

### КОМИТ

КОМИТ представляет собой совсем другой вид языка. От программиста не требуется, чтобы он мыслил в терминах таких списков, которые рассматривались нами ранее. Этот язык пер-

<sup>1)</sup> То есть списку ненужных ячеек. — *Прим. перев.*

воначально разрабатывался для того, чтобы облегчить программирование алгоритмов машинного перевода. КОМИТ оказывается весьма полезным в очень широкой области приложений, но он специально предназначен для обработки информации, представленной в виде символов. Арифметические возможности ограничены. Система КОМИТ является интерпретирующей.

Данные для программы находятся на рабочем поле и в архивах. Предусмотрены усложненные операции для исследования рабочего поля в поисках разделов, удовлетворяющих сложным критериям, а также для преобразований рабочего поля. Автоматически реализуется быстрый метод поиска по словарю элементов информации.

Внутри машинной памяти информация хранится в виде стандартных списков. Контроль над тем, какие списки нужны, а какие могут быть возвращены в свободную часть памяти, полностью осуществляется системой и недоступен программисту.

Данные состоят из рабочих выражений. Эти выражения представляют собой последовательности элементов, называемых рабочими составляющими. Каждая составляющая может иметь некоторую структуру, но рабочее выражение не может оказаться частью рабочей составляющей. Вот пример составляющей:

РОБИНСОН/23, ПОЛ МУЖСКОЙ, ИНТЕРЕСЫ ВИНО  
ЖЕНЩИНЫ ПЕНИЕ

Каждая составляющая содержит один символ (в нашем примере это строка РОБИНСОН) и несколько индексов. Один из индексов может оказаться числом, и допускается любое количество логических индексов, каждый из которых может содержать дальнейшие подиндексы. Этим ограничивается глубина структуры рабочей составляющей.

Программа на языке КОМИТ состоит из набора правил. Они определяют способ размещения и обработки данных, а также способ отыскания следующего правила. Правило может содержать пять частей, причем не обязательно, чтобы все эти части присутствовали в каждом случае. Правило содержит свое имя, по которому к нему могут происходить обращения из других правил той же программы. Оно включает часть, в которой задается имя следующего правила. В нем содержатся левая часть и правая часть, определяющие поиск и действия, и, наконец, в нем имеется программная часть, которая связана с вводом и выводом, а также с использованием архивов.

Левая часть инструкции определяет поиск на рабочем поле. Она содержит выражение, аналогичное тому выражению, которое желательно найти. Рабочее поле просматривается слева направо в поиске первого вхождения выражения, соответствующего

заданному. Однако заданное выражение не обязательно является точной копией искомого. Например, можно написать только символ, и тогда будет найден первый элемент, включающий этот символ, независимо от того, каковы его индексы. Или же можно было бы задать вместе с символом какой-то индекс, и тогда был бы найден первый элемент, включающий по крайней мере такой же символ и такой же индекс. Если написать

**МУЖЧИНА/ФАМИЛИЯ СМИТ**

то можно было бы найти элемент

**МУЖЧИНА/23, ФАМИЛИЯ СМИТ, ИНТЕРЕСЫ ВИНО  
ЖЕНЩИНЫ ПЕНИЕ**

В языке КОМИТ существует такой символ, который может заменять что угодно, например, \$1 означает любой единичный элемент. Таким образом, запись

**\$1/ФАМИЛИЯ РОБИНСОН**

означала бы запрос первого элемента из рабочего поля, содержащего указанный индекс. Символ \$ в выражении из левой части означает поиск любого числа объектов, так что запись

**A + \$ + B**

означает поиск первого вхождения A, за которым следует любая строка элементов, за которой следует B. Возможно также управление поиском с помощью численных значений. Можно запросить первый элемент с индексом, превосходящим определенное число.

Выражение, записанное в правой части, является инструкцией относительно тех действий, которые должны быть выполнены после отыскания на рабочем поле аналога левой части. Записанная в правой части составляющая заносится на рабочее поле. Если запись содержит число n, то на рабочее поле копируется та составляющая из найденного участка рабочего поля, которая соответствует n-й составляющей из левой части. Таким образом, запись

**A + \$1 + B = 2 + C**

привела бы к отысканию первого элемента с символом A, за которым следует один элемент, за которым в свою очередь следует элемент с символом B. Элементы с A и B будут отброшены, а C занесется на рабочее поле вслед за промежуточным элементом.

Выражения из правой части позволяют использовать и изменять индексы и производить численные выкладки.

Система КОМИТ допускает использование вспомогательной памяти.

Практика показала, что обработка списков успешно применяется в компиляторах, в супервизорных программах, в области обработки символьной информации, а также при решении сложных невычислительных задач. Чаще всего ею пользуются высококвалифицированные программисты или же те, кого вынуждает к этому специфика решаемых задач. Если бы обработка списков была более легкой и более доступной для среднего программиста, то она оказалась бы, по-видимому, еще более полезной. Во многих программах содержатся части, для реализации которых удобны методы обработки списков. Такими частями являются, например, ввод данных в форме слов и печать переменных сообщений. Сомнительно, чтобы широкому распространению этих методов способствовали специализированные языки обработки списков. Обычный пользователь не станет изучать специализированный язык, если этот язык не окажет ему существенной помощи при написании всей его программы. А в любом случае такой язык может оказаться совсем неподходящим для выполнения какой-то части программирования. Обработка списков должна стать одним из многих технических средств, находящихся в распоряжении программистов и предназначенных для использования в тех частях программ, в которых они нужны. Необходима будет разносторонняя языковая система, включающая списки.

Можно указать некоторые черты такого языка. Любой из существующих в настоящее время языков фиксируется на одном или двух методах представления программ. Каждый из этих методов обладает некоторыми преимуществами, но для любого метода найдутся такие задачи, которые легче решаются другими методами. Новый язык будет включать последовательные операторы, как языки ассемблеров, ФОРТРАН и АЛГОЛ; он будет допускать функциональные описания, как ЛИСП и АЛГОЛ; он будет включать описательные инструкции, как КОМИТ. Для удобства работы малоквалифицированных программистов потребуется усовершенствование системы обозначений. В частности, запись элементарных операций во многих случаях является не наглядной, а они часто применяются в сложных

суперпозициях. Рассмотрим типичное списковое выражение в функциональной записи:

$\text{cons}(a, \text{cons}(b, \text{cons}(c, \text{cons}(\text{cons}(\text{cons}(d, 0), 0), \text{cons}(f, 0))))))$

Его трудно понять. В арифметических выражениях такая невразумительность исключается благодаря применению нефиксированных операторов типа плюса и минуса. Запись с использованием нефиксированного оператора «запятая» делает приведенное выше выражение вполне понятным:

$(a, b, c, ((d)), f)$

Другим недостатком языков обработки списков является отсутствие хороших описаний. АЛГОЛ позволяет сказать, что идентификатор принадлежит классу действительных, целых или логических переменных. При обработке списков имеется гораздо больше возможностей, так как может быть описана структура списка. Очень часто случается, что при написании новой программы много усилий уходит на выбор представления объектов. Программирование не начинается до тех пор, пока такой выбор не будет завершен. Природа большинства языков обработки списков заставляет встраивать такой выбор в программу. Часто, когда программа уже частично написана, выясняется, что выбранное представление было не совсем удачным и что можно было бы применить лучшее представление, которое существенно облегчило бы программирование или способствовало бы большей эффективности программы. Изменение программы может оказаться настолько трудным, что придется от него отказаться, если представление объектов встроено во все части программы. Задача программирования состоит в том, чтобы отделять те свойства программ, которые могут реализовываться логически независимо. Это позволит сделать программы более наглядными и облегчить внесение в них изменений. Таким образом, обработка списков упростилась бы, если бы пользователь имел возможность с помощью понятных системе описаний отделить ту часть своей программы, в которой определяется структура информации, от той части, в которой описывается процесс.

Рассмотрим следующий функциональный оператор, запутанный еще больше, чем предыдущий:

$t := \text{cons}(\text{hd}(\text{tl}(\text{tl}(\text{tl}(l)))), \text{cons}(\text{hd}(\text{tl}(\text{hd}(\text{tl}(\text{tl}(l)))))),$   
 $\text{cons}(\text{cons}(\text{hd}(l), \text{cons}(\text{hd}(\text{tl}(l)), 0)), 0))$

Совершенно невозможно понять, что здесь происходит. Однако если описать структуру и применить нефиксированные опера-

торы, то оператор становится простым.

**список l со структурой**  $(a, b, (c, d));$

$t := (c, d, (a, b))$

Эта программа является наглядной; другое ее преимущество состоит в том, что она может быть изменена заменой только описания структуры.

Естественная тенденция, которая четко проявляется в существующих языках, состоит в том, чтобы каждый новый язык включал лучшие свойства своих предшественников с постоянным усовершенствованием способов написания программ. Можно ожидать, что эта тенденция приведет к тому, что обработка списков станет в близком будущем общедоступной и вполне удобной в применении.

## СПИСОК ЛИТЕРАТУРЫ

В списке употребляются следующие сокращения: *CACM* — Communications of the Association for Computing machinery; *JACM* — Journal of the Association for Computing machinery.

1. Baeker H. D., Mapped list structures, *CACM*, 6 (Aug. 1963), 435.
2. Bailey M. J., Barnett M. P., Burleson P. B., Symbol manipulation in FORTRAN-SASP 1 routines, *CACM*, 7 (June 1964), 339.
3. Berztiss A. T., A note on the storage of strings, *CACM*, 8 (Aug. 1965), 512.
4. Bobrow D. G., Raphael B., A comparison of list processing languages, *CACM*, 7 (Apr. 1964), 231.
5. Carr III J. W., Recursive subscripting compilers and listtype memories, *CACM*, 2 (February 1959), 4.
6. Comfort W. T., Multi-word list items, *CACM*, 7 (June 1964), 357.
7. COMIT Programmers reference manual, MIT Press 1961.
8. Conway R. W., Delfausse J. J., Maxwell W. L., Walker W. E., CLP — the Cornell list processor, *CACM*, 8 (Apr. 1965), 215.
9. Evans A., Perlis A. J., Van Voeran H., The use of threaded lists in constructing a combined ALGOL and machine like assembly processor, *CACM*, 4 (Jan. 1961), 36.
10. Fredkin E., Trie memory, *CACM*, 3 (Sep. 1960), 490.
11. Gelernter N., A FORTRAN compiled processing language, *JACM*, 7 (Apr. 1960), 87.
12. Gelernter H., Hansen J. R., Coveland D. W., Empirical explorations of the geometry theorem machine, Proc. Western Joint Computer Conference (May 1960), 143.
13. Green B. F., Computer languages for symbol manipulation, *I. R. E. Trans., HFE-2* (Mar. 1961), 2.
14. Iliffe J. K., Jodeit J. G., A dynamic storage allocation scheme, *Computer Journal*, 5 (Oct. 1962), 200.
15. Jenkins D. P., List programming in: Introduction to system programming, Academic Press, 1964.
16. McCarthy J., Programs with commonsense, Proc. Symposium on the mechanization of thought processes, N. P. L. HMSO, 1959.
17. McCarthy J., Recursive functions of symbolic expressions and their computation by machine, Pt. 1, *CACM*, 3 (Apr. 1960), 184.
18. McCarthy J. et al., LISP 1.5 Programmers manual, MIT Press, 1962.
19. Newell A., Simon H. A., The logic theory machine, *I. R. E. Trans., IT-2* (Sep. 1956), 61.
20. Newell A., Shaw J. S., Simon H. A., Chess playing programs and the problem of complexity, *IBM Journal of research and development*, 2 (Oct. 1958), 320.
21. Newell A. et al., Information Processing Language V, Manual Section I and II, Rand Corp. P 1918 (Mar. 1960).
22. Newell A., Tonge F., An introduction to Information Processing Language V, *CACM*, 3 (Apr. 1960), 205.

23. Newell A., Documentation of IPL-V, *CACM*, 6 (Mar. 1963), 86.
24. Perlis A. J., Thornton C., Symbol manipulation by threaded lists, *CACM*, 3 (April 1960), 195.
25. Research Lab. of Electronics and MIT computation center, Introduction to COMIT programming, MIT Press, 1961.
26. Shaw J. C., Newell A., Simon H. A., A command structure for complex information processing, Proc. Western Joint Computer Conference (1958), 119.
27. Weizenbaum J., Knotted list structures, *CACM*, 5 (Mar. 1962), 161.
28. Weizenbaum J., Symmetric list processor, *CACM*, 6 (Sep. 1963), 524.
29. Woodward P. M., List processing in: Advances in programming and non-numerical computation, Pergamon, 1966.
30. Woodward P. M., Jenkins D. P., Atoms and lists, *Computer Journal*, 4 (Apr. 1961), 47.
31. Yngve V. H., A programming language for mechanical translation, *Mech. Translation*, 5 (July 1958), 25.
32. Yngve V. H., COMIT, *CACM*, 6 (Mar. 1963), 83.
33. Kuno S., Oettinger A., Multiple path syntactic analyser in: Information Processing 62, North Holland, Amsterdam, 1963.

## ОГЛАВЛЕНИЕ

От издательства . . . . .	5
1 Введение . . . . .	7
Предпосылки обработки списков . . . . .	9
2. Представление списков . . . . .	11
Списки в вычислительной машине . . . . .	11
Списки на бумаге . . . . .	16
Простые операции над списками . . . . .	17
3. Операции над списками . . . . .	21
Рекурсия . . . . .	22
Эффективность рекурсии . . . . .	26
Магазинные списки . . . . .	29
Операции изменения списков . . . . .	30
4. Более сложные свойства . . . . .	32
Атомы . . . . .	34
Векторы . . . . .	35
Циклические списки . . . . .	37
Списки во вспомогательной памяти . . . . .	38
5. Пример обработки списков . . . . .	40
6. Сборка мусора . . . . .	45
Описание нужных списков . . . . .	48
Описание ненужных списков . . . . .	52
Сборка мусора и векторы . . . . .	54
7. Некоторые типичные языки обработки списков . . . . .	55
IPL-V . . . . .	56
ЛИСП . . . . .	59
СЛИП . . . . .	62
FLPL . . . . .	64
КОМИТ . . . . .	64
8. Будущее обработки списков . . . . .	67
Список литературы . . . . .	70

### УВАЖАЕМЫЙ ЧИТАТЕЛЬ!

Ваши замечания о содержании книги, ее оформлении, качестве перевода и другие просим сообщать по адресу: 129820, Москва, И-110 ГСП, 1-й Рижский пер., д. 2, издательство «Мир».