2.4. МНОГОСВЯЗНЫЕ СТРУКТУРЫ

Теперь, после подробного исследования линейных списков и древовидных структур, принципы представления структурной информации внутри компьютера должны быть очевидны. В настоящем разделе будет рассмотрено еще одно приложение таких методов, на этот раз — для типичного случая с несколько более сложной структурной информацией. В приложениях более высокого уровня обычно используется сразу несколько типов структур.

"Многосвязная структура" состоит из узлов с несколькими полями связи в каждом узле, а не только с одним или двумя в структурах из предыдущих примеров. Примеры использования множественных связей приводились выше, например, при моделировании работы лифта в разделе 2.2.5 и работе с полиномами по многим переменным в разделе 2.3.3.

Как мы увидим далее, присутствие множества различных типов связей в одном узле *не* обязательно сопровождается усложнением их создания или восприятия по сравнению с уже рассмотренными алгоритмами. Кроме того, мы ответим на еще один важный вопрос: *В каком объеме структурная информация должна быть представлена в памяти в явном виде?*

Рассматриваемая здесь задача возникает в связи с созданием программы-компилятора для трансляции программ на языке COBOL и других сходных с ним языков. При работе с языком COBOL программист может присваивать символьные имена переменным программы на нескольких уровнях. Например, программа может иметь дело с файлами данных о продажах и покупках со следующей структурой.

```
1 SALES                    1 PURCHASES
  2 DATE                     2 DATE
    3 MONTH                    3 DAY
    3 DAY                      3 MONTH
    3 YEAR                     3 YEAR
  2 TRANSACTION              2 TRANSACTION
    3 ITEM                     3 ITEM
    3 QUANTITY                 3 QUANTITY
    3 PRICE                    3 PRICE
    3 TAX                      3 TAX
    3 BUYER                    3 SHIPPER
      4 NAME                     4 NAME
      4 ADDRESS                  4 ADDRESS
```
(1)

На этой схеме некоторой конфигурации данных показано, что каждый элемент файла SALES (продажи) состоит из двух частей: DATE (дата) и TRANSACTION (транзакция). Причем DATE подразделяется на три части, а TRANSACTION — на пять частей. Аналогичные замечания относятся и к файлу PURCHASES (покупки). Относительный порядок имен указывает порядок, в котором эти величины предстают во внешних представлениях файла (например, на магнитной ленте или распечатанных формах). Обратите внимание на то, что DAY и MONTH в этих двух файлах представлены в разном порядке. Программист приводит и другую не показанную здесь информацию, которая сообщает о том, какое пространство в памяти занимает каждый элемент данных и в каком формате эти данные представлены. Подобные соображения несущественны для темы данного раздела, а потому не будут рассматриваться.

Программист при работе с языком COBOL описывает сначала формат файла и другие переменные программы, а затем — алгоритмы, которые оперируют этими величинами. Для ссылки на отдельную переменную в приведенном выше примере было бы недостаточно просто указать имя DAY, так как не существует способа указания, в каком файле она находится: в SALES или PURCHASES. Следовательно, при работе с языком COBOL можно с помощью выражения DAY OF SALES указать, что элемент DAY является частью элемента SALES. Программист мог бы также записать в более полной форме, что

DAY OF DATE OF SALES,

но, вообще-то, не следует придавать величинам больше квалификаций (описаний), чем это действительно необходимо, во избежание неоднозначности. Таким образом, выражение

NAME OF SHIPPER OF TRANSACTION OF PURCHASES

можно сократить до

NAME OF SHIPPER,

поскольку существует только одна часть данных с именем SHIPPER.

Эти правила языка COBOL могут быть выражены более точно в следующей форме.

a) Каждому имени предшествует некоторое связанное с ним положительное целое число, которое называется *номером уровня*. Имя относится либо к *простейшему элементу*, либо к *группе* из одного или нескольких элементов со своими именами. В последнем случае все элементы группы должны иметь один номер уровня, который должен быть выше, чем номер уровня для имени группы. (Например, элементы DATE и TRANSACTION в приведенном примере имеют уровень 2, который выше уровня 1 для элемента SALES.)

b) Для ссылки на простейший элемент или группу элементов с именем A_0 используется общая форма

$$A_0 \text{ OF } A_1 \text{ OF } \ldots \text{ OF } A_n,$$

где $n \geq 0$, а A_j является именем элемента, который прямо или косвенно содержится внутри группы с именем A_{j+1} для $0 \leq j < n$. Должен существовать только один элемент A_0, который удовлетворяет этому условию.

c) Если одно и то же имя A_0 появляется в нескольких местах, должен существовать способ ссылки на каждый случай использования такого имени с помощью квалификации (описания).

Например, согласно правилу (c) конфигурация данных

```
1 AA
  2 BB
    3 CC
    3 DD
  2 CC
```
(2)

недопустима, так как при втором появлении элемента CC не существует однозначного способа ссылки на элементы с таким именем (см. упр. 4).

COBOL обладает еще одним свойством, которое может оказать влияние на процесс создания компилятора и работу рассматриваемых приложений, а именно — возможностью ссылаться сразу на несколько элементов. В таком случае программист может записать

MOVE CORRESPONDING α TO β,

что приведет к перемещению всех элементов с соответствующими именами из области данных α в область данных β. Например, команда языка COBOL

```
MOVE CORRESPONDING DATE OF SALES TO DATE OF PURCHASES
```

означает, что значениями переменных MONTH, DAY и YEAR из файла SALES нужно заменить значения переменных DAY, MONTH, YEAR в файле PURCHASES. (Относительный порядок DAY и MONTH при этом изменяется.)

В данном разделе будут рассмотрены три алгоритма, которые можно использовать в компиляторе COBOL и которые предназначены для выполнения перечисленных ниже действий.

Операция 1. Обработать описания имен и номеров уровней, подобных показанным на схеме (1), разместив соответствующую информацию в таблицах внутри компилятора для использования в операциях 2 и 3.

Операция 2. Определить, справедлива ли заданная ссылка, квалифицированная согласно правилу (b), и, если справедлива, найти соответствующий элемент данных.

Операция 3. Найти все соответствующие пары элементов, которые указаны в команде CORRESPONDING.

Допустим, что наш компилятор уже имеет "подпрограмму таблиц символов", которая может преобразовать символьное имя в связь, указывающую на позицию таблицы, соответствующую этому имени. (Более подробно методы построения алгоритмов для обработки таблиц символов обсуждаются в главе 6.) Помимо таблицы символов, имеется таблица большего размера, *таблица данных*, содержащая по одной позиции для каждого элемента данных из исходной программы на языке COBOL, которую нужно откомпилировать.

Очевидно, что нельзя создать алгоритм выполнения операции 1 до тех пор, пока неизвестно, какого рода информацию предполагается хранить в таблице данных. Причем форма таблицы данных зависит от типа информации, необходимой для выполнения операций 2 и 3. Таким образом, прежде всего следует обратить внимание на операции 2 и 3.

Для определения значения ссылки на языке COBOL

$$A_0 \text{ OF } A_1 \text{ OF } \ldots \text{ OF } A_n, \qquad n \geq 0, \tag{3}$$

сначала необходимо найти имя A_0 в таблице символов. Причем должен существовать ряд связей от позиции таблицы символов ко всем позициям таблицы данных, которые относятся к этому имени. Затем для каждой позиции таблицы данных потребуется установить связь с элементом-группой, в которую он входит. Тогда, если существует поле связи от позиций таблицы данных к таблице символов, нетрудно сообразить, как следует организовать обработку ссылок наподобие (3). Более того, чтобы найти пары, заданные в команде MOVE CORRESPONDING, потребуется установить некоторые связи от позиций таблицы данных для каждого элемента-группы к отдельным элементам этой группы.

Таким образом, для каждой позиции таблицы данных необходимо создать дополнительно пять полей связи:

PREV (связь с предыдущей позицией с тем же именем, если таковая имеется);

PARENT (связь с наименьшей группой, если таковая имеется, содержащей элемент);

NAME (связь с позицией таблицы символов элемента);

CHILD (связь с первым подэлементом группы);

SIB (связь со следующим подэлементом группы, содержащей элемент).

Ясно, что структуры данных в языке COBOL, подобные приведенным выше структурам SALES и PURCHASES, являются деревьями, а связи наподобие PARENT, CHILD и SIB уже знакомы нам из предыдущего материала. (Представление дерева в виде обычного бинарного дерева основано на связях CHILD и SIB, а при добавлении связи PARENT получим "трижды связанное дерево". Пять упомянутых выше связей состоят из этих трех связей вместе со связями PREV и NAME, которые несут дополнительную информацию о данной древовидной структуре.)

Вероятно, не все пять связей являются необходимыми, или достаточными, но попробуем создать алгоритм с исходным предположением о том, что элементы таблицы данных содержат все пять полей (и дополнительную информацию, которая не имеет отношения к данной проблеме). В качестве примера множественного связывания рассмотрим такие две структуры данных языка COBOL:

```
1 A          1 H
  3 B          5 F
    7 C          8 G
    7 D        5 B
  3 E          5 C
  3 F            9 E
    4 G          9 D
                 9 G
```
(4)

Их следует представить в виде (5) (со связями, указанными в символьной форме). Поле LINK в каждой позиции таблицы символов указывает на последнюю из встреченных позиций таблицы данных с символьным именем из позиции таблицы символов.

Сначала потребуется создать алгоритм для построения таблицы данных такого типа. Обратите внимание на то, что в языке COBOL предусмотрена гибкость выбора номеров уровней. Левая структура (4) полностью эквивалентна структуре

```
1 A
  2 B
    3 C
    3 D  ,
  2 E
  2 F
    3 G
```

поскольку номера уровней необязательно должны быть последовательными числами.

Таблица символов

	LINK
A:	A1
B:	B5
C:	C5
D:	D9
E:	E9
F:	F5
G:	G9
H:	H1

В пустых клетках содержится информация, не имеющая отношения к данной задаче

Таблица данных

	PREV	PARENT	NAME	CHILD	SIB	
A1:	Λ	Λ	A	B3	H1	
B3:	Λ	A1	B	C7	E3	
C7:	Λ	B3	C	Λ	D7	
D7:	Λ	B3	D	Λ	Λ	
E3:	Λ	A1	E	Λ	F3	
F3:	Λ	A1	F	G4	Λ	
G4:	Λ	F3	G	Λ	Λ	
H1:	Λ	Λ	H	F5	Λ	
F5:	F3	H1	F	G8	B5	
G8:	G4	F5	G	Λ	Λ	
B5:	B3	H1	B	Λ	C5	
C5:	C7	H1	C	E9	Λ	
E9:	E3	C5	E	Λ	D9	
D9:	D7	C5	D	Λ	G9	
G9:	G8	C5	G	Λ	Λ	

(5)

Однако некоторые последовательности номеров уровней недопустимы. Например, если номер уровня для элемента D в (4) был бы заменен номером 6 (в любом месте), была бы получена бессмысленная конфигурация данных, нарушающая правило, в соответствии с которым все элементы группы должны иметь одинаковые номера. Поэтому в следующем алгоритме выполняется проверка, соблюдается ли правило (а) языка COBOL.

Алгоритм А (*Построение таблицы данных*). Этот алгоритм позволяет получить последовательность пар (L, P), где L — положительное целое число, обозначающее номер уровня, а P — позиция таблицы символов, соответствующая таким структурам данных COBOL, как (4). Данный алгоритм создает таблицу данных, подобную приведенной выше, в примере (5). Когда P указывает на позицию таблицы символов, которая прежде не встречалась, связь LINK(P) становится равной Λ. В этом алгоритме используется вспомогательный стек, который обрабатывается, как обычный стек (на основе последовательного распределения памяти, как в разделе 2.2.2, или на основе связанного распределения памяти, как в разделе 2.2.3).

A1. [Инициализация.] Ввести в стек элемент $(0, \Lambda)$. (В этом алгоритме стек будет содержать пары (L, P), где L — целое число, а P — указатель. В ходе работы алгоритма стек содержит номер уровня и указатели на последние позиции данных на всех уровнях данного дерева, которые располагаются выше текущего уровня. Например, в приведенном примере до появления пары 3 F стек будет содержать пары

$$(0, \Lambda) \qquad (1, \text{A1}) \qquad (3, \text{E3})$$

в направлении снизу вверх.)

A2. [Следующий элемент.] Пусть (L, P) — это следующий элемент данных, взятый из входного потока. После исчерпания входного потока выполнение алгоритма прекращается. Установить Q ⇐ AVAIL (т. е. пусть Q — адрес нового узла, в котором можно разместить следующую позицию таблицы данных).

A3. [Установка связей для символьных имен.] Установить

$$\texttt{PREV(Q)} \leftarrow \texttt{LINK(P)}, \qquad \texttt{LINK(P)} \leftarrow \texttt{Q}, \qquad \texttt{NAME(Q)} \leftarrow \texttt{P}.$$

(Так будут заданы значения для двух связей из пяти в узле NODE(Q). Теперь нужно соответствующим образом установить значения связей PARENT, CHILD и SIB.)

A4. [Сравнение уровней.] Пусть пара (L1, P1) является верхним элементом стека. Если L1 < L, установить CHILD(P1) ← Q (или, если P1 = Λ, установить FIRST ← Q, где FIRST — переменная, которая будет указывать на первый элемент таблицы данных) и перейти к шагу A6.

A5. [Удаление верхнего элемента.] Если L1 > L, то удалить верхний элемент стека. Пусть, например, (L1, P1) — новый элемент, который только что был удален из верхней части стека. Затем повторить шаг A5. Если L1 < L (т. е. на одном уровне обнаружены разные номера), выдать сообщение об ошибке. В противном случае, а именно — при L1 = L, установить SIB(P1) ← Q и удалить верхний элемент стека. Пусть, например, пара (L1, P1) является парой, которая только что была удалена из верхней части стека.

A6. [Установка связей семьи.] Установить PARENT(Q) ← P1, CHILD(Q) ← Λ, SIB(Q) ← Λ.

A7. [Ввести элемент в стек.] Поместить пару (L, Q) в верхнюю часть стека и вернуться к шагу A2. ▮

Благодаря введению вспомогательного стека, который описан на шаге A1, данный алгоритм настолько упрощается, что не требует дополнительных разъяснений.

Следующая задача заключается в поиске позиции таблицы данных, соответствующей ссылке

$$A_0 \ \texttt{OF} \ A_1 \ \texttt{OF} \ \ldots \ \texttt{OF} \ A_n, \qquad n \geq 0. \tag{6}$$

В хорошем компиляторе следует также предусмотреть проверку недвусмысленности такой ссылки. В этом случае сразу же напрашивается следующий алгоритм (рис. 40). Все, что теперь необходимо сделать, — просмотреть список позиций таблицы данных для имени A_0 и убедиться в том, что в точности одна из них соответствует квалификации $A_1, \ldots, A_n$.

Алгоритм B (*Проверка квалифицированной ссылки*). В соответствии со ссылкой (6) программа таблицы символов найдет указатели $P_0, P_1, \ldots, P_n$ на позиции таблицы символов $A_0, A_1, \ldots, A_n$ соответственно.

Назначение данного алгоритма заключается в проверке $P_0, P_1, \ldots, P_n$ и либо определении того, что ссылка (6) ошибочна, либо в установлении для значения переменной Q адреса позиции таблицы данных для элемента, на который ссылается (6).

B1. [Инициализация.] Установить Q ← Λ, P ← LINK(P_0).

B2. [Готово?] Если P = Λ, то выполнение алгоритма прекращается; в этот момент Q равно Λ, если (6) не соответствует никакой позиции таблицы данных. Но если

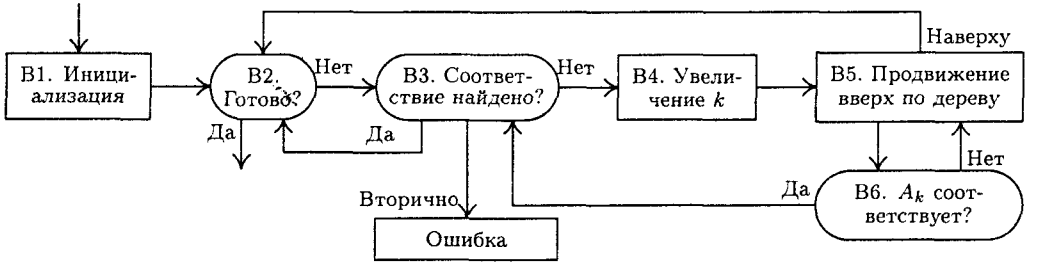

Рис. 40. Алгоритм для проверки ссылок в языке COBOL.

$\texttt{P} \neq \Lambda$, установить $\texttt{S} \leftarrow \texttt{P}$ и $k \leftarrow 0$. ($\texttt{S}$ — переменная-указатель, значения которой меняются от $\texttt{P}$ и ведут вверх по дереву по связям PARENT; k — целочисленная переменная, которая принимает значения от 0 к n. На практике указатели $P_0, \ldots, P_n$ часто содержатся в связанном списке, и тогда вместо k используется переменная-указатель, которая совершает обход этого списка; см. упр. 5.)

B3. [Соответствие найдено?] Если $k < n$, то перейти к шагу B4. В противном случае найдена соответствующая позиция таблицы данных. Если $\texttt{Q} \neq \Lambda$, то найдена вторая такая позиция, и поэтому нужно отослать сообщение об ошибке. Установить $\texttt{Q} \leftarrow \texttt{P}$, $\texttt{P} \leftarrow \texttt{PREV(P)}$ и перейти к шагу B2.

B4. [Увеличение k.] Установить $k \leftarrow k + 1$.

B5. [Продвижение вверх по дереву.] Установить $\texttt{S} \leftarrow \texttt{PARENT(S)}$. Если $\texttt{S} = \Lambda$, то соответствие найти не удалось; установить $\texttt{P} = \texttt{PREV(P)}$ и перейти к шагу B2.

B6. [A_k соответствует?] Если $\texttt{NAME(S)} = P_k$, перейти к шагу B3; в противном случае перейти к шагу B5. ▮

Обратите внимание, что связи CHILD и SIB в этом алгоритме не использовались.

Третий, и последний, из нужных нам алгоритмов имеет отношение к команде MOVE CORRESPONDING. Прежде чем приступить к созданию алгоритма, нужно четко сформулировать его назначение. В языке COBOL выражение

$$\texttt{MOVE CORRESPONDING}\ \alpha\ \texttt{TO}\ \beta, \tag{7}$$

где α и β — ссылки наподобие (6) на элементы данных, обозначает сокращенную запись множества всех выражений типа

$$\texttt{MOVE}\ \alpha'\ \texttt{TO}\ \beta',$$

для которых существует такое целое число $n \geq 0$ и такие n имен $A_0, A_1, \ldots, A_{n-1}$, что

$$\begin{aligned} \alpha' &= A_0\ \texttt{OF}\ A_1\ \texttt{OF}\ \ldots\ \texttt{OF}\ A_{n-1}\ \texttt{OF}\ \alpha, \\ \beta' &= A_0\ \texttt{OF}\ A_1\ \texttt{OF}\ \ldots\ \texttt{OF}\ A_{n-1}\ \texttt{OF}\ \beta \end{aligned} \tag{8}$$

и либо α', либо β' является простейшим элементом (а не группой элементов). Более того, необходимо, чтобы в (8) была указана *полная* квалификация первых уровней, а именно — что A_{j+1} является родителем A_j для $0 \leq j < n$. α' и β' должны располагаться в дереве на n уровней ниже, чем α и β.

Для рассматриваемого здесь примера (4) выражение

```
MOVE CORRESPONDING A TO H
```

является сокращенной формой записи выражений

```
MOVE B OF A TO B OF H
MOVE G OF F OF A TO G OF F OF H
```

Алгоритм для распознавания соответствующих пар α', β' несмотря на свою простоту довольно интересен, т. е. необходимо совершить обход дерева с корнем α в прямом порядке, одновременно выискивая в дереве β совпадающие имена и пропуская поддеревья, в которых появление соответствующих элементов невозможно. Имена $A_0, \ldots, A_{n-1}$ из (8) располагаются в обратном порядке: $A_{n-1}, \ldots, A_0$.

Алгоритм С (*Поиск соответствующих пар*). Для заданных PO и QO, которые указывают на позиции таблицы данных для α и β соответственно, этот алгоритм последовательно находит все пары указателей (P, Q) на элементы (α', β'), удовлетворяющие упомянутым выше требованиям.

С1. [Инициализация.] Установить P $\leftarrow$ PO, Q $\leftarrow$ QO. (В оставшейся части этого алгоритма указательные переменные P и Q совершают обход деревьев с корнями α и β соответственно.)

С2. [Простейший элемент?] Если CHILD(P) $= \Lambda$ или CHILD(Q) $= \Lambda$, то вывести (P, Q) как одну из искомых пар и перейти к шагу С5. В противном случае установить P $\leftarrow$ CHILD(P), Q $\leftarrow$ CHILD(Q). (На этом шаге P и Q указывают на элементы α' и β', удовлетворяющие (8), а команду MOVE α' TO β' необходимо выполнять тогда и только тогда, когда либо α', либо β' (или оба сразу) — простейший элемент.)

С3. [Сравнение имен.] (P и Q указывают на элементы данных, которые соответственно имеют квалификацию в виде

$$A_0 \texttt{ OF } A_1 \texttt{ OF } \ldots \texttt{ OF } A_{n-1} \texttt{ OF } \alpha$$

и

$$B_0 \texttt{ OF } A_1 \texttt{ OF } \ldots \texttt{ OF } A_{n-1} \texttt{ OF } \beta.$$

Теперь задача заключается в том, чтобы узнать, можно ли сделать так, чтобы $B_0 = A_0$, проверяя все имена в группе A_1 OF ... OF A_{n-1} OF β.) Если NAME(P) $=$ NAME(Q), то перейти к шагу С2 (найдено совпадение). В противном случае, если SIB(Q) $\neq \Lambda$, установить Q $\leftarrow$ SIB(Q) и повторить шаг С3. (Если SIB(Q) $= \Lambda$, значит, в этой группе нет соответствующего имени и следует перейти к шагу С4.)

С4. [Продвижение.] Если SIB(P) $\neq \Lambda$, то установить значения P $\leftarrow$ SIB(P), Q $\leftarrow$ CHILD(PARENT(Q)) и вернуться к шагу С3. Если SIB(P) $= \Lambda$, то установить P $\leftarrow$ PARENT(P) и Q $\leftarrow$ PARENT(Q).

С5. [Готово?] Если P $=$ PO, то прекратить выполнение алгоритма; в противном случае перейти к шагу С4. ▮

Блок-схема этого алгоритма показана на рис. 41. Доказательство корректности алгоритма можно легко получить с помощью метода индукции по размеру обрабатываемых деревьев (см. упр. 9).

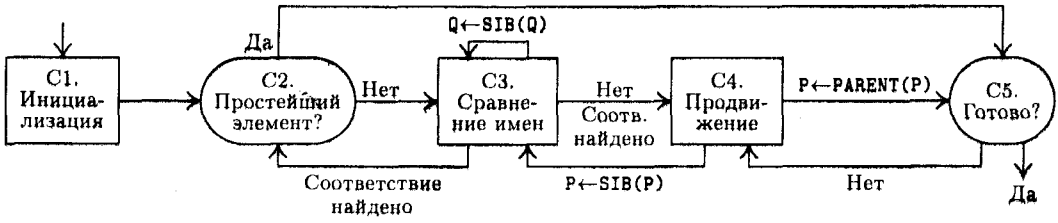

Рис. 41. Алгоритм для выполнения команды MOVE CORRESPONDING.

Теперь рассмотрим способы применения пяти полей связи (PREV, PARENT, NAME, CHILD и SIB) в алгоритмах B и C. Замечательно то, что эти связи образуют "полный набор" в том смысле, что алгоритмы B и C действительно выполняют минимальный объем работы при продвижении по таблице данных. И всякий раз, когда нужно сослаться на другую позицию таблицы данных, ее адрес сразу же становится доступным, поэтому нет необходимости проводить дополнительный поиск. Трудно представить, как можно было бы сделать алгоритмы B и C более быстрыми, включив в таблицу дополнительную информацию о связях (см., однако, упр. 11).

Каждое поле связи может рассматриваться как *ключ* к этой программе, который используется для ускорения работы алгоритмов. (Конечно, алгоритм для построения таблиц, т. е. алгоритм A, выполняется медленнее, поскольку он имеет больше связей, которые необходимо заполнить. Но таблица строится лишь один раз.) С другой стороны, ясно, что в построенной выше таблице данных содержится гораздо больше избыточной информации. Посмотрим, что произойдет, если *удалить* некоторые поля связи.

Связь PREV, хотя она и не используется в алгоритме C, имеет большое значение в алгоритме B и, похоже, является существенной частью любого компилятора языка COBOL, за исключением случаев, когда требуется выполнять длительные операции поиска. Следовательно, поле, которое связывает все элементы с одинаковыми именами, имеет большое значение для эффективной работы. Поэтому стратегию действий в такой ситуации можно слегка модифицировать и вместо связи Λ в конце каждого списка применить циклическое связывание. Но этого не стоит делать, когда поля связи не изменяются и не удаляются.

Связь PARENT используется в алгоритмах B и C, хотя можно обойтись и без нее, если в алгоритме C применить вспомогательный стек или добавить связь SIB так, чтобы задействовать связи-нити (как в разделе 2.3.2). Таким образом, становится ясно, что связь PARENT используется только в алгоритме B. Если бы поле SIB было связью-нитью, чтобы элементы со значением поля связи SIB $= \Lambda$ содержали вместо него значение SIB = PARENT, можно было бы найти родителя любого элемента данных, следуя по связям SIB. Добавленные связи-нити можно отличить либо с помощью нового поля TAG в каждом узле, в котором указывалось бы, что поле SIB содержит связь-нить, либо с помощью условия SIB(P) < P, если позиции таблицы данных хранятся последовательно в памяти в порядке появления. Это значит, что на шаге B5 придется выполнить короткий поиск, а сам алгоритм соответственно станет работать медленнее.

Связь NAME используется в этих алгоритмах только на шагах B6 и C3. В обоих случаях можно было бы выполнить проверку NAME(S) $= P_k$ и NAME(P) = NAME(Q)

иначе, если бы связь NAME отсутствовала (см. упр. 10), но это существенно замедлило бы выполнение внутренних циклов в алгоритмах B и C. Ясно, что здесь снова имеет место компромисс между экономией пространства для связи и скоростью выполнения алгоритмов. (Скорость работы алгоритма C не так уж важна для компиляторов языка COBOL, если рассматривать типичные способы употребления команд MOVE CORRESPONDING. Однако алгоритм B должен работать быстро.) Известно, что связи NAME используются и для выполнения других важных задач в компиляторе языка COBOL, особенно при выводе диагностической информации.

Алгоритм A постепенно создает таблицу данных, причем никогда не приходится возвращать узлы в область свободной памяти. Поэтому позиции таблицы данных обычно располагаются в последовательных ячейках памяти в порядке появления элементов данных в исходной программе на языке COBOL. Таким образом, в нашем примере (5) ячейки A1, B3, ... будут располагаться одна за другой. Такой последовательный характер размещения ячеек в таблице данных позволяет существенно упростить работу. Например, связь CHILD каждого узла либо равна Λ, либо указывает на узел, который следует сразу же за ним, поэтому поле CHILD можно сократить до размера 1 бит. Или поле CHILD можно удалить и вместо него проверить условие PARENT$(P + c) = P$, где c — размер узла в таблице данных.

Таким образом, необязательно использовать сразу все пять полей связи, хотя они очень полезны для ускорения работы алгоритмов B и C. Эта ситуация весьма типична для большинства многосвязных структур.

Интересно отметить, что по крайней мере полдюжины разработчиков компиляторов COBOL в начале 60-х годов независимо пришли к одному способу организации таблицы данных с помощью пяти связей (или четырех из пяти, так как связь CHILD обычно не используется). Впервые описание такого метода было опубликовано Г. В. Лоусоном (мл.) (H. W. Lawson, Jr.) [см. *ACM National Conference Digest* (Syracuse, N.Y.: 1962]. Однако Дэвид Дам (David Dahm) в 1965 году предложил оригинальный метод, который позволяет получить тот же результат, что и при работе с алгоритмами B и C, но с использованием двух полей связи и последовательного распределения позиций в таблице данных без существенного снижения скорости выполнения (см. упр. 12–14).

УПРАЖНЕНИЯ

1. [*00*] Если конфигурации данных в языке COBOL рассматривать как древовидные структуры, то в каком порядке они записаны: в прямом, обратном или ни в одном из них?

2. [*10*] Оцените время выполнения алгоритма A.

3. [*22*] Структуры данных языка PL/I подобны структурам языка COBOL, но в них допустима любая последовательность номеров уровней. Например, последовательность

1 A		1 A
3 B		2 B
5 C	эквивалентна последовательности	3 C
4 D		3 D
2 E		2 E

В итоге правило (a) изменяется таким образом: "Элементы группы должны иметь невозрастающую последовательность номеров уровней, причем все они должны быть больше номера уровня группы данной группы". Как необходимо изменить алгоритм A, чтобы

выполнить переход от соглашений, принятых для языка COBOL, к соглашениям, принятым для языка PL/I?

▸ **4.** [*26*] Алгоритм A не позволит обнаружить ошибку, если программист в COBOL-программе нарушит упомянутое в этом разделе правило (c). Как следует изменить алгоритм A, чтобы принимались только те структуры, которые удовлетворяют правилу (c)?

5. [*20*] На практике алгоритм B может получать из входного потока связанный список ссылок на таблицу символов, а не то, что прежде называлось "$P_0, P_1, \ldots, P_n$". Пусть T является такой указательной переменной, что

$$\texttt{INFO(T)} \equiv P_0,\ \texttt{INFO(RLINK(T))} \equiv P_1,\ \ldots,\ \texttt{INFO(RLINK}^{[n]}\texttt{(T))} \equiv P_n,\ \texttt{RLINK}^{[n+1]}\texttt{(T)} = \Lambda.$$

Покажите, как изменить алгоритм B, чтобы в нем в качестве входного потока можно было использовать связанный список.

6. [*23*] В языке PL/I допускается использование структур данных, которые во многом подобны структурам языка COBOL, но без применения ограничения (c). Вместо этого получим правило, по которому квалифицированная ссылка (3) является однозначной, если указана "полная" квалификация, т. е. если A_{j+1} — родитель A_j для $0 \le j < n$ и если A_n не имеет родителя. Ограничение (c) теперь сведено к простому условию, согласно которому никакие два элемента данных группы не могут иметь одно и то же имя. На второе появление CC в (2) можно было бы недвусмысленно сослаться с помощью ссылки "CC OF AA", а на три элемента данных

```
1 A
  2 A
    3 A
```

можно было бы в соответствии с соглашениями, принятыми в языке PL/I, сослаться в такой форме: "A", "A OF A", "A OF A OF A". [*Замечание.* На самом деле "OF" заменяется точкой в языке PL/I, а порядок является реверсивным. Так, вместо "CC OF AA" в языке PL/I используется обозначение "AA.CC", но для данного упражнения это несущественно.] Покажите, как можно модифицировать алгоритм B, чтобы он удовлетворял соглашениям, принятым для языка PL/I.

7. [*15*] Что означает в языке COBOL команда MOVE CORRESPONDING SALES TO PURCHASES по отношению к структуре данных (1)?

8. [*10*] При каких условиях команда MOVE CORRESPONDING α TO β означает то же самое, что и команда MOVE α TO β, в соответствии с данным в этом разделе определением?

9. [*M23*] Докажите корректность алгоритма C.

10. [*23*] (a) Как можно было бы выполнить проверку условия NAME(S) $= P_k$ на шаге B6, если бы в узлах таблицы данных не было связи NAME? (b) Как можно было бы выполнить проверку NAME(P) = NAME(Q) на шаге C3, если бы в узлах таблицы данных не было связи NAME? (Предположим, что все другие связи присутствуют, как описано в этом разделе.)

▸ **11.** [*23*] Какие дополнительные связи или изменения в стратегии создания алгоритмов могли бы ускорить работу алгоритма B или C?

12. [*25*] (Задача Д. М. Дама (D. M. Dahm).) Рассмотрим представление таблицы данных в последовательных ячейках памяти с помощью всего двух связей для каждого элемента данных:

PREV (так же, как в данном разделе);

SCOPE (связь с последним простейшим элементом в данной группе).

Получим SCOPE(P) = P тогда и только тогда, когда NODE(P) представляет собой простейший элемент. Например, таблица данных (5) будет заменена таким представлением.

	PREV	SCOPE		PREV	SCOPE		PREV	SCOPE
A1:	Λ	G4	F3:	Λ	G4	B5:	B3	B5
B3:	Λ	D7	G4:	Λ	G4	C5:	C7	G9
C7:	Λ	C7	H1:	Λ	G9	E9:	E3	E9
D7:	Λ	D7	F5:	F3	G8	D9:	D7	D9
E3:	Λ	E3	G8:	G4	G8	G9:	G8	G9

(Ср. с (5) из раздела 2.3.3.) Обратите внимание, что NODE(P) является частью дерева под узлом NODE(Q) тогда и только тогда, когда $\texttt{Q} < \texttt{P} \le \texttt{SCOPE(Q)}$. Создайте алгоритм, который выполняет функции алгоритма B, если таблица данных имеет такой формат.

▶ **13.** [*24*] Предложите вместо алгоритма A другой алгоритм для работы с таблицей данных с форматом, описанным в упр. 12.

▶ **14.** [*28*] Предложите вместо алгоритма C другой алгоритм для работы с таблицей данных с форматом, описанным в упр. 12.

15. [*25*] (Задача Дэвида С. Вайса (David S. Wise).) Измените алгоритм A так, чтобы для стека не использовалось никакое дополнительное пространство. [*Указание.* Поля SIB всех узлов с указателями в стеке в этой формулировке равны Λ.]