ТАРТУСКИЙ
ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ

ТРУДЫ

ВЫЧИСЛИТЕЛЬНОГО ЦЕНТРА

45

ТАРТУ
1980

ТАРТУСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ

ВОПРОСЫ ОБРАБОТКИ ДАННЫХ СО СЛОЖНОЙ СТРУКТУРОЙ

ТРУДЫ ВЫЧИСЛИТЕЛЬНОГО
ЦЕНТРА

Выпуск 45

ТАРТУ 1980

Утверждено на заседании совета математического факультета 24 октября 1980 года.

ТЕХНИКА РАСПОЗНАВАНИЯ СОСТАВНЫХ ИМЕН

А. Изотамм

Иерархическая структура данных широко используется как в языках программирования (начиная с языков Кобол и ПЛ/I), так и в системах баз данных. В программах, обрабатывающих структурные данные, возникает необходимость идентифицирования элементов структуры. В общем случае это делается при помощи составных имен. Каждое такое имя состоит из имени ссылаемого элемента, которому прибавлено нужное количество имен корней групп более высоких уровней; при этом может иметь место избыточность составного имени. Разработчики транслятора с языка, допускающего структуры, должны решить следующие задачи:

1. определение множества допустимых ссылок;
2. программирование модуля для проверки корректности имен в структурах, определенных пользователем;
3. программирование модуля для проверки допустимости заданной ссылки и поиска ссылаемого элемента.

Известно несколько методов обработки составных имен, например алгоритм Гриса [2] и два решения, выдвинутые Гейтсом и Поплавским [1]. В настоящей статье описывается решение, принятое при реализации СУБД РАМА.

1. Допустимость ссылок

Пусть T является отмеченным корневым деревом, в котором v_0 – неотмеченный корень, а V – множество отмеченных вершин. Метками или именами вершин $v_s \in V$ $(s = 1, 2, \ldots, p = |V|)$ служат элементы из конечного множества имен $\mathcal{A}$. Корни поддеревьев в T будем называть отцами, а их непосредственные подчиненные – сыновьями отца или братьями между собой. Единственным ограничением для меток вершин v_s является требование, что братья должны иметь различные метки. На рис. 1 в качестве примера представлено дерево (взаимствованное из [1]), для которого $p = 8$ и $\mathcal{A} = \{a, b, c, d\}$.

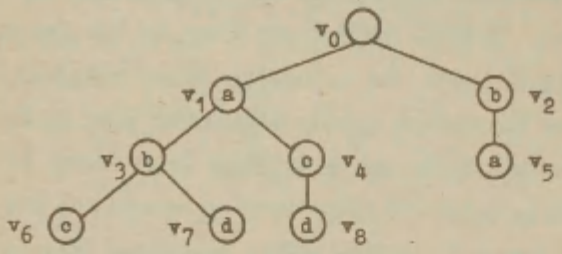


Рис. 1.

Если v_s является вершиной n-ого уровня, то ее идентификатором (или полным составным именем) называется кортеж

$$I_s = \langle a_1, a_2, \ldots, a_n \rangle ,$$

где $a_i \in \mathcal{A}$ $(i = 1, 2, \ldots, n)$ является меткой вершины i-того уровня на пути от корня v_0 до вершины v_s. Множество всех идентификаторов данного дерева обозначаем через $\mathcal{I}$:

$$\mathcal{I} = \{I_1, I_2, \ldots, I_p\} .$$

Поскольку все сыновья любого отца имеют несовпадающиеся имена, то элементы множества $\mathcal{I}$ являются попарно несовпадающими.

В дальнейшем нам иногда удобно истолковывать идентификатор I_s как пару $I_s = F_s a_n$, где $F_s = \langle a_1, a_2, \ldots, a_{n-1} \rangle$ является идентификатором отца рассматриваемой вершины v_s. Для вершин первого уровня при этом считаем, что $F_s = \Lambda$ (пустой кортеж).

Будем еще предполагать, что в каждой вершине дерева T зафиксировано упорядочение ее сыновей, т.е. считаем, что братья всюду снабжены порядковыми номерами. Тогда любой идентификатор $I_s \in J$ можно перекодировать в виде вектора $K_s = (c_1, c_2, \ldots, c_n)$, где значение компоненты c_i ($i = 1, 2, \ldots, n$) является порядковым номером вершины с меткой a_i среди ее братьев (ср. [4], стр. 4). Вектор K_s будем называть вектором координат вершины v_s.

В таблице 1 приведены все характеристики вершин дерева, представленного на рис. 1 (векторы координат составлены в предположении, что братья всюду пронумерованы в порядке их расположения на рисунке слева направо).

Таблица 1.

v_s	уровень	I_s	K_s	F_s	a_n
v_1	1	⟨a⟩	(1)	Λ	a
v_2	1	⟨b⟩	(2)	Λ	b
v_3	2	⟨a,b⟩	(1,1)	⟨a⟩	b
v_4	2	⟨a,c⟩	(1,2)	⟨a⟩	c
v_5	2	⟨b,a⟩	(2,1)	⟨b⟩	a
v_6	3	⟨a,b,c⟩	(1,1,1)	⟨a,b⟩	c
v_7	3	⟨a,b,d⟩	(1,1,2)	⟨a,b⟩	d
v_8	3	⟨a,c,d⟩	(1,2,1)	⟨a,c⟩	d

Каждый идентификатор $I_s \in \mathfrak{I}$ однозначно определяет соответствующую вершину v_s и может, таким образом, быть использован в качестве ссылки на v_s. В языках, предназначенных для работы с деревообразными структурами, обычно допускаются еще и ссылки в виде более коротких кортежей, т.е. множество $\mathcal{f}(v_s)$ допустимых ссылок на v_s содержит, как правило, и другие элементы кроме I_s. Элементы множества $\mathcal{f}(v_s)$ должны, конечно, быть подкортежами кортежа I_s (т.н. неполные составные имена вершины v_s), но допустимыми считаются далеко не все такие подкортежи. Далее уточняется, какие ссылки считаются допустимыми в реализации СУБД РАМА, т.е. даются конкретные правила образования множеств $\mathcal{f}(v_s)$.

Сопоставим каждому имени $a \in \mathfrak{A}$ множество $V(a)$, состоящее из всех тех вершин дерева, для которых a служит меткой. Случай $V(a) = \emptyset$ (имя a не является меткой ни одной вершины дерева T) можно при этом не рассматривать, считая, что такие имена исключены из множества $\mathfrak{A}$.

Если $|V(a)| = 1$, то a является уникальным именем, т.е. дерево содержит в точности одну вершину v с меткой a. Кроме соответствующего идентификатора $I = \langle a_1, a_2, \cdots, a_{n-1}, a\rangle$ допустимыми теперь считаются все ссылки на v в виде $X = Za$, где Z является любым подкортежом (включая Λ) идентификатора $P = \langle a_1, a_2, \cdots, a_{n-1}\rangle$, т.е. идентификатора отца рассматриваемой вершины v. Если множество всех образуемых таким образом сокращений (включая I) обозначить через $\mathcal{P}(v)$, то очевидно $|\mathcal{P}(v)| = 2^{n-1}$. Итак, в данном случае множество допустимых ссылок совпадает с множеством всех сокращений:

$$\mathcal{f}(v) = \mathcal{P}(v).$$

В общем случае имени a соответствуют несколько вершин, т.е. $|V(a)| = k > 1$. На любую вершину $v_i \in V(a)$ можно ссылаться ее идентификатором $I_i = F_i a$, но дополнительные возможности построения ссылок на такие вершины ограничены. Для более точного определения множества допустимых ссылок на вершину $v_i \in V(a)$ исходим из предположения упорядоченности элементов множества V(a).

Будем предполагать, что вершины в множестве V(a) упорядочены в порядке возрастания номеров их уровней. В случае же вершин одного уровня упорядочим их в порядке лексикографического возрастания соответствующих векторов координат. Пусть элементами множества V(a) в порядке такого упорядочения являются вершины $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$.

Метку a будем теперь считать уникальным именем вершины v_{i_1} и множество $f(v_{i_1})$ допустимых ссылок на нее строим так же, как и в случае $|V(a)| = 1$, т.е.

$$f(v_{i_1}) = \mathcal{P}(v_{i_1}).$$

Далее, из всевозможных сокращений идентификатора I_{i_2} допускаются все те, которые не совпадают с любой ссылкой из $f(v_{i_1})$ и т.д.; т.е. допустимыми ссылками на $v_{i_j} \in V(a)$ считаются все те ссылки, которые не входят в множества допустимых ссылок на вершины $v_{i_1}, v_{i_2}, \ldots, v_{i_{j-1}}$. Таким образом, для каждого $j = 2,\ldots,k$ получаем

$$f(v_{i_j}) = \mathcal{P}(v_{i_j}) \setminus \Big(\bigcup_{r=1}^{j-1} f(v_{i_r}) \Big).$$

Например, для дерева рисунка 1 допустимыми являются ссылки, приведенные в таблице 2.

Таблица 2.

Имя	v(имя)	j	f
a	v_1	1	<a>
	v_5	2	<b,a>
b	v_2	1	<b>
	v_3	2	<a,b>
c	v_4	1	<c>, <a,c>
	v_6	2	<b,c>, <a,b,c>
d	v_7	1	<d>, <a,d>, <b,d>, <a,b,d>
	v_8	2	<c,d>, <a,c,d>

В качестве второго примера таблица 3 приводит допустимые ссылки на вершины дерева, представленного на рисунке 2 (где братья опять пронумерованы в порядке их расположения на рисунке слева направо).

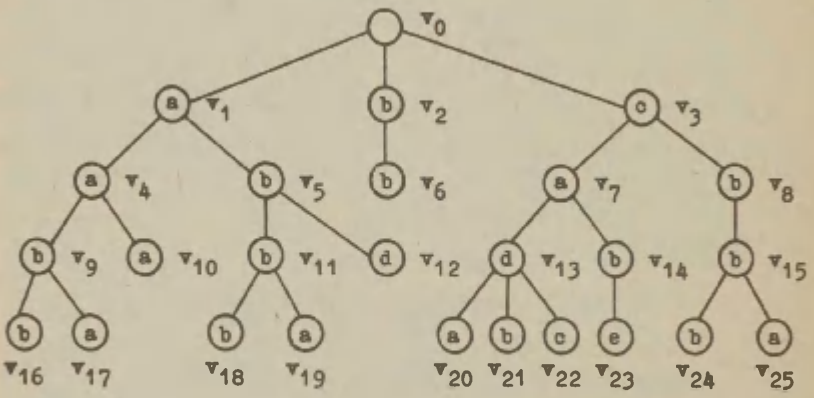


Рис. 2.

Таблица 3.

Имя	v(имя)	j	$\mathcal{F}$
a	v_1	1	⟨a⟩
	v_4	2	⟨a,a⟩
	v_7	3	⟨c,a⟩
	v_{10}	4	⟨a,a,a⟩
	v_{17}	5	⟨b,a⟩, ⟨a,b,a⟩, ⟨a,a,b,a⟩
	v_{19}	6	⟨b,b,a⟩, ⟨a,b,b,a⟩
	v_{20}	7	⟨d,a⟩, ⟨a,d,a⟩, ⟨c,d,a⟩, ⟨c,a,d,a⟩
	v_{25}	8	⟨c,b,a⟩, ⟨c,b,b,a⟩
b	v_2	1	⟨b⟩
	v_5	2	⟨a,b⟩
	v_6	3	⟨b,b⟩
	v_8	4	⟨c,b⟩
	v_9	5	⟨a,a,b⟩
	v_{11}	6	⟨a,b,b⟩
	v_{14}	7	⟨c,b⟩, ⟨c,a,b⟩
	v_{15}	8	⟨c,b,b⟩
	v_{16}	9	⟨a,a,b,b⟩
	v_{18}	10	⟨b,b,b⟩, ⟨a,b,b,b⟩
	v_{21}	11	⟨d,b⟩, ⟨a,d,b⟩, ⟨c,d,b⟩, ⟨c,a,d,b⟩
	v_{24}	12	⟨c,b,b,b⟩
c	v_3	1	⟨c⟩
	v_{22}	2	⟨d,c⟩, ⟨a,c⟩, ⟨c,c⟩, ⟨a,d,c⟩, ⟨c,a,c⟩, ⟨c,d,c⟩, ⟨c,a,d,c⟩
d	v_{12}	1	⟨d⟩, ⟨b,d⟩, ⟨a,d⟩, ⟨a,b,d⟩
	v_{13}	2	⟨c,d⟩, ⟨c,a,d⟩
e	v_{23}	1	⟨e⟩, ⟨b,e⟩, ⟨a,e⟩, ⟨c,e⟩, ⟨a,b,e⟩, ⟨c,b,e⟩, ⟨c,a,e⟩, ⟨c,a,b,e⟩

2. Обработка ссылок

Задачу обработки составных имен можно разбить на две подзадачи. Первая подзадача состоит в проверке, является ли данное составное имя подкортежем какого-то идентификатора. Если таких идентификаторов более одного, то вторая подзадача состоит в выборе (по каким-то критериям) точно одного из них. Известны следующие решения этих задач.

В монографии Гриса ([2], стр. 268-274) описывается восходящий метод проверки: если задана подлежащая обработке ссылка

$$X = \langle b_1, b_2, \ldots, b_m \rangle ,$$

то в качестве кандидатов проверяются все идентификаторы вершин $v \in V(b_m)$, а X считается допустимой ссылкой, если она является подкортежем точно одного из этих идентификаторов. Если такого идентификатора не найдется, то X является ошибочной, если же подходящих идентификаторов больше одного, то ссылка считается двусмысленной.

В статье [1] представлены два варианта нисходящего просмотра, где исходной точкой является компонента b_1 ссылки X. Ссылка X сравнивается со всеми (под)кортежами, которые начинаются с именем b_1, а принятие решения в отношении допустимости фактически совпадает с соответствующим решением восходящего метода проверки.

Недостатком указанных методов является то, что они значительно ограничивают множество допустимых ссылок, воздвигая тем самым некоторые дополнительные ограничения для выбора меток вершин (какие именно, остается неопределенным). Напри-

мер, в случае дерева на рис.1 Гэйтс и Поплавский показывают, что ссылки <a, c>, <a> и <b> являются недопустимыми; тот же результат получается при восходящем методе проверки. Таким образом, возможны случаи, где любые ссылки на некоторые вершины являются недопустимыми.

В системе РАМА используется метод обработки ссылок, который можно называть восстановлением идентификатора. Составное имя является, в общем случае, сокращением некоторого идентификатора: если идентификатором рассматриваемой вершины служит $I = \langle a_1, a_2, \ldots, a_n \rangle$, то ее неполное составное имя $X = \langle b_1, b_2, \ldots, b_m \rangle$, где $b_m = a_n$, зафиксирует m точек на пути, определенным идентификатором. Распознавание ссылки можно рассматривать как попытку восстановления полного пути с корня v_0 в вершину, т.е. восстановления идентификатора I.

Такое восстановление возможно осуществить как сверху вниз, так и снизу вверх. Как показал Грис [2], восходящий просмотр имеет то преимущество, что уже в начале проверки определяется множество проверяемых идентификаторов, а следующие шаги не могут увеличить это множество. Количество шагов проверки определяется длиной m заданного составного имени X.

При восстановлении идентификатора по заданной ссылке будем пользоваться следующей операцией. Если на пути от корня v_0 до рассматриваемой вершины v_s зафиксировать какую-нибудь вершину v, то идентификатор $I_s = \langle a_1, a_2, \ldots, a_n \rangle$ можно рассматривать как пару $I_s = BE$, где начало

$$B = \langle a_1, a_2, \ldots, a_i \rangle$$

является идентификатором вершины v, а конец

$$E = \langle a_{i+1}, a_{i+2}, \ldots, a_n \rangle$$

представляет путь от v до v_s. Если v_s является вершиной n-ого уровня, то имеется n различных возможностей представления I_s в виде таких пар (в одной из них $E = \Lambda$).

Пусть задана ссылка $X = \langle b_1, b_2, \ldots, b_m \rangle$, для которой требуется проверить ее допустимость и в случае положительного ответа восстановить соответствующий идентификатор. В качестве первого шага построим множество идентификаторов $\mathfrak{I}(b_m)$ для всех вершин множества $V(b_m)$. Очевидно, из них кандидатами на место искомого идентификатора могут служить лишь те, длина которых не меньше чем m. Таким образом получается исходное множество кандидатов

$$\mathcal{L}_m = \{ I : I \in \mathfrak{I}(b_m) \;\&\; |I| \geq m \}.$$

На следующих шагах постараемся для каждого идентификатора-кандидата I выбрать некоторое его представление в виде пары $I = B_j E_j$ так, чтобы при каждом $j = m-1, \ldots, 2, 1$ было $j \leq |B_j| < |B_{j+1}|$. Если для какого-то идентификатора такого представления не существует, то этот идентификатор выбывает из дальнейшего рассмотрения.

Последовательность множеств идентификаторов-кандидатов

$$\mathcal{L}_m \supset \mathcal{L}_{m-1} \supset \ldots \supset \mathcal{L}_2 \supset \mathcal{L}_1$$

строится теперь по следующей формуле:

$$\mathcal{L}_j = \{ I : I \in \mathcal{L}_{j+1} \;\&\; I = B_j E_j \;\&\; B_j \in \mathfrak{I}(b_j) \}.$$

Ведь если ссылка X является корректной, то на пути от корня v_0 до искомой вершины должна находиться вершина с таким идентификатором B_j.

Если на j-том шагу проверки оказывается, что $\mathfrak{I}(b_j) = \emptyset$ или $\mathcal{L}_j = \emptyset$, то ссылка X является некорректной. После завершения последнего шага (j=1) искомым идентификатором является тот, который в полученном множестве $\mathcal{L}_1$ окажется первым по упорядоченности, определенной для множества $V(b_m)$.

Этот идентификатор можно формально обозначить как одноэлементное множество $\mathcal{L}_0$. В целях общности обозначений можно его также рассматривать как пару $I = B_0E_0$: если компонента b_1 ссылки X является меткой вершины первого уровня, то $B_0 = \Lambda$; если же b_1 служит меткой вершины i-того ($i > 1$) уровня, то B_0 представляет собой идентификатор вершины $(i - 1)$-го уровня.

Приведем несколько примеров восстановления идентификаторов вершин дерева, представленного на рис. 1. Проверки ссылок представляются в виде таблиц, содержащие элементы множества кандидатов после каждого шага, а также разложение каждого кандидата на очередном шагу, номер шага j, компоненту b_j ссылки X и множество $V(b_j)$.

В качестве первого примера таблица 4 воспроизводит ход проверки ссылки $X = \langle b, c\rangle$, для которой устанавливается допустимость и определяется соответствующая вершина v_6.

Проверку ссылки $X = \langle a, c\rangle$ иллюстрирует таблица 5. Здесь следует обратить внимание на то, что множество $\mathcal{L}_1$ содержит два идентификатора и искомая вершина v_4 определяется по упорядоченности. Ход проверки недопустимой ссылки $X = \langle b, b, c\rangle$ показан в таблице 6.

Последний пример опирается на дерево рисунка 2. В таблице 7 показана проверка ссылки $X = \langle b, b, b\rangle$.

Таблица 4.

j	b_j	$V(b_j)$	B_j	E_j	$\mathcal{L}_j$
2	o	I_4	⟨a,c⟩	Λ	⟨a,c⟩
		I_6	⟨a,b,c⟩	Λ	⟨a,b,c⟩
1	b	I_2	-	-	-
		I_3	⟨a,b⟩	⟨c⟩	⟨a,b,c⟩
0	-	-	⟨a⟩	⟨b,c⟩	⟨a,b,c⟩

Таблица 5.

j	b_j	$V(b_j)$	B_j	E_j	$\mathcal{L}_j$
2	o	I_4	⟨a,c⟩	Λ	⟨a,c⟩
		I_6	⟨a,b,c⟩	Λ	⟨a,b,c⟩
1	a	I_1	⟨a⟩	⟨c⟩	⟨a,c⟩
		I_5	⟨a⟩	⟨b,c⟩	⟨a,b,c⟩
0	-	-	Λ	⟨a,c⟩	⟨a,c⟩

Таблица 6.

j	b_j	$V(b_j)$	B_j	E_j	$\mathcal{L}_j$
3	o	I_4	⟨a,c⟩	Λ	-
		I_6	⟨a,b,c⟩	Λ	⟨a,b,c⟩
2	b	I_2	-	-	-
		I_3	⟨a,b⟩	⟨c⟩	⟨a,b,c⟩
1	b	I_2	-	-	-
		I_3	-	-	-

Таблица 7.

j	b_j	$V(b_j)$	B_j	E_j	$\mathcal{L}_j$
3	b	I_9	⟨a,a,b⟩	$\wedge$	⟨a,a,b⟩
		I_{11}	⟨a,b,b⟩	$\wedge$	⟨a,b,b⟩
		I_{14}	⟨c,a,b⟩	$\wedge$	⟨c,a,b⟩
		I_{15}	⟨c,b,b⟩	$\wedge$	⟨c,b,b⟩
		I_{16}	⟨a,a,b,b⟩	$\wedge$	⟨a,a,b,b⟩
		I_{18}	⟨a,b,b,b⟩	$\wedge$	⟨a,b,b,b⟩
		I_{21}	⟨c,a,d,b⟩	$\wedge$	⟨c,a,d,b⟩
		I_{24}	⟨c,b,b,b⟩	$\wedge$	⟨c,b,b,b⟩
2	b	I_9	-	-	-
		I_{11}	⟨a,b⟩	⟨b⟩	⟨a,b,b⟩
		I_{14}	-	-	-
		I_{15}	⟨c,b⟩	⟨b⟩	⟨c,b,b⟩
		I_{16}	⟨a,a,b⟩	⟨b⟩	⟨a,a,b,b⟩
		I_{18}	⟨a,b,b⟩	⟨b⟩	⟨a,b,b,b⟩
		I_{21}	-	-	-
		I_{24}	⟨c,b,b⟩	⟨b⟩	⟨c,b,b,b⟩
1	b	I_{11}	-	-	-
		I_{15}	-	-	-
		I_{16}	-	-	-
		I_{18}	⟨a,b⟩	⟨b,b⟩	⟨a,b,b,b⟩
		I_{24}	⟨c,b⟩	⟨b,b⟩	⟨c,b,b,b⟩
0	-	-	⟨a⟩	⟨b,b,b⟩	⟨a,b,b,b⟩

3. Реализация

Рассмотрим теперь программное решение задачи обработки составных имен в СУБД РАМА. Среда данной задачи описывается в статьях [4], [5], [6], [7] и [8].

Таблица имен создается транслятором с языка описания данных RDL в виде хэш-таблицы с внешними цепочками. Формат звена такой цепочки изображен на рис. 3. Подполе name предусмотрено для помещения имени $a \in \mathcal{A}$, подполе col содержит ссылку в случае коллизии, а подполе list ссылает на цепочку вершин дерева T с меткой a: звенья этой цепочки представляют элементы множества V(a).

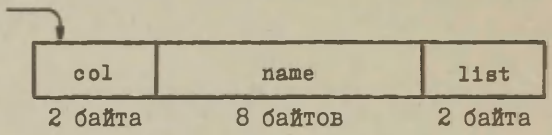


Рис. 3.

Каждое звено указанной цепочки содержит разную информацию о вершине, но с точки зрения данной статьи интерес представляют лишь те поля, которые показаны на рис. 4. Подполе KV предназначено для ссылки на вектор координат вершины $v \in V(a)$, который представляется как последовательность двухбайтовых полей. Значением подполя l является длина вектора

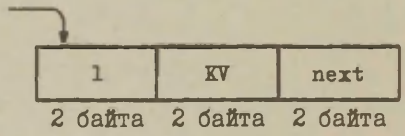


Рис. 4.

координат в байтах (сокращенная на единицу). Подполе **next** содержит ссылку на следующее звено данной цепочки; признаком конца звена служит пустая ссылка.

Пример таблицы имен для дерева, приведенного на рис. 1, представлен на рис. 5. Здесь предполагается, что хэш-функция вырабатывает совпадающие адреса для имен "a" и "c".

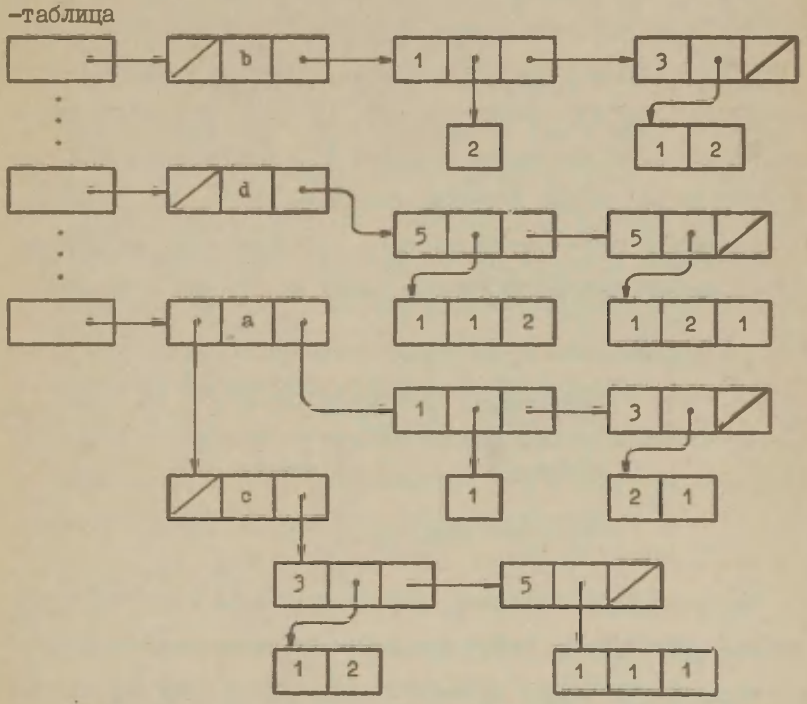


Рис. 5.

Построение таблицы имен происходит следующим образом. При трансляции очередной вершины легенды по ее имени a определяется адрес цепочки для V(a); новое звено включается в

цепочку так, чтобы соблюдалась описанная выше упорядоченность элементов множества V(a). Поиск подходящего места для звена сопровождается проверкой корректности имени: вектор координат отца новой вершины должен отличаться от всех векторов координат отцов тех вершин, которые уже включены в данную цепочку.

Поиск из таблицы имен по составному имени интерпретируется как определение адреса подходящего звена той цепочки, которой представляется множество $V(b_m)$, где b_m является последней компонентой ссылки $X = \langle b_1, b_2, \ldots, b_m \rangle$. Если длина ссылки $m > 1$, то множество кандидатов представляется как цепочка звеньев такого формата, как показано на рис. 6. Выбытие кандидата на каком-нибудь шагу проверки осуществляется как исключение соответствующего звена из цепочки кандидатов.

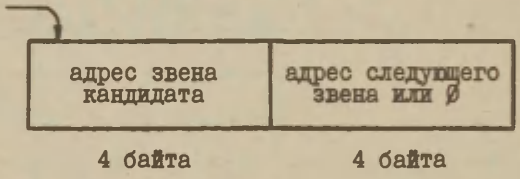


Рис. 6.

Для построения таблицы имен и поиска из этой таблицы в трансляторах системы РАМА используется реентерабельная макрокоманда NNAME. Ниже приводится формальное описание ее параметров в форме, принятой в документации математического обеспечения ЕС ЭВМ (см. [9], стр. 14-16). Чтобы включить новое звено цепочки V(a) в таблицу имен, макрокоманду следует записывать в следующем формате:

[символ] WNAME W, HT = адрес таблицы имен, NAME = адрес имени вершины, U = адрес нового звена [,ERROR = символ блока обработки ошибок]

Для обработки составного имени WNAME кодируется таким образом:

[символ] WNAME R, HT = адрес таблицы имен, NAME = адрес составного имени, L = адрес длины составного имени, U = адрес звена искомой вершины [,ERROR = символ блока обработки ошибок]

Каждый адрес как операнда макрокоманды может кодироваться как RX-тип, A-тип или номер общего регистра (в скобках) из промежутка 2-11 (см. [9], стр. 251). Код завершения работы WNAME содержится в регистре 15: об успешной работе сигнализирует значение 0. Если при создании таблицы выясняется, что имя вершины является недопустимым, то код возврата равен 4. Аварийное завершение поиска кодируется таким образом: код 4 значит, что какая-то компонента ссылки не входит в множество имен 0, а код 8 вырабатывается, если заданная ссылка не является подкортежем ни одного идентификатора.

Литература

1. Gates, G.W., Poplawski, D.A., A Simple Technique for Structured Variable Lookup. CACM, September 1973, Vol. 16, Numb. 9, 561-565.
2. Грис Д., Конструирование компиляторов для цифровых вычислительных машин. "Мир", Москва, 1975.

3. Кнут Д., Искусство программирования для ЭВМ. т. 1, "Мир", Москва, 1976.

4. Изотамм А., Дерево описания записи в системе РАМА. Труды ВЦ ТГУ, 1980, № 43, 3-35.

5. Изотамм А., Каазик Ю., Томбак М., Язык определения записи. Труды ВЦ ТГУ, 1978, № 41, 7-64.

6. Изотамм А., Физическое представление записи в системе РАМА. Труды ВЦ ТГУ, 1980, № 43, 36-54.

7. Каазик Ю., Рауп А., Язык манипулирования данными. Труды ВЦ ТГУ, 1978, № 41, 97-140.

8. Каазик Ю., Ээльма П., Ввод и корректировка данных. Труды ВЦ ТГУ, 1978, № 41, 75-96.

9. Единая система электронных вычислительных машин. Операционная система. Макрокоманды супервизора и управления данными. Руководство программиста. Ц51.804.002 Д5, 1977.

РЕАЛИЗАЦИЯ ЯЗЫКА ВВОДА ДАННЫХ

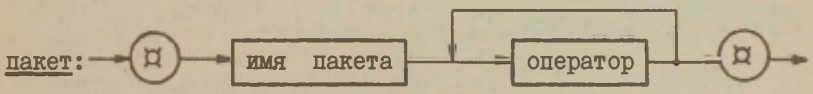
Ю.Каазик, А.Толпин

Одной из составных частей СУБД РАМА является язык DIL, предназначенный для ввода и корректировки данных. Синтаксис и семантика этого языка описаны в статье [2]. Настоящая статья посвящена проблемам, возникающим при построении соответствующего компилятора.

1. Изменения синтаксиса языка DIL

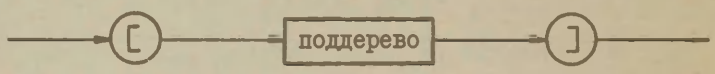
В ходе реализации проекта РАМА пришлось по разным причинам ввести в синтаксис языка DIL некоторые изменения. Из-за этих изменений необходимы следующие исправления в определениях статьи [2].

По техническим причинам из языка DIL была удалена конструкция "разделители", а также изменен признак начала и конца DIL-программы. Конструкция "пакет" определяется теперь следующим образом:



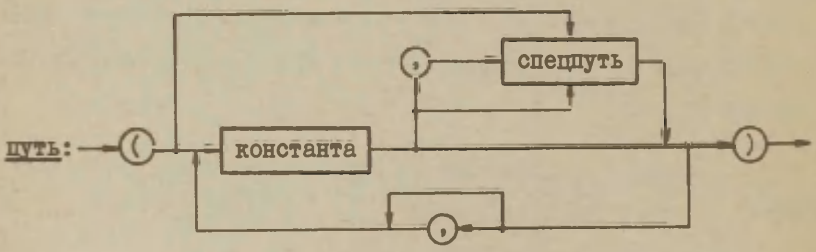
Разделители α, β, γ и δ получили в реализации конкретные значения. В качестве разделителя α можно использовать как запятую, так и пробел (причем запятую в некоторых случаях можно опускать), а в качестве разделителей β, γ и δ используются соответственно точка с запятой, круглая открывающая скобка и круглая закрывающая скобка. Такое упрощение сделано из предположения, что на используемых средствах ввода имеются упомянутые разделители. В противном случае их легко можно заменить на любые другие подходящие разделители.

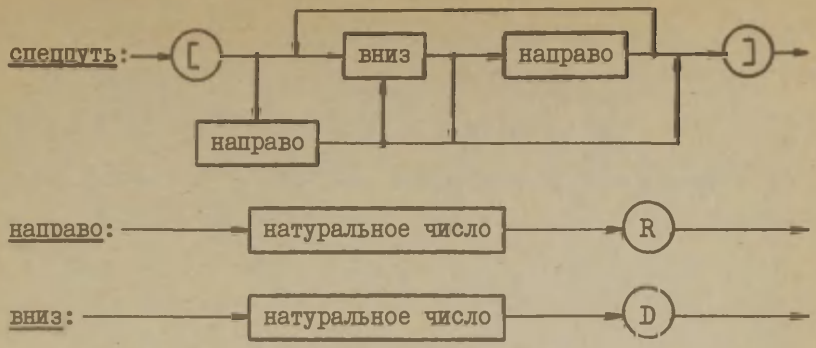
Для упрощения анализа DIL-программы в конструкции "дерево" вместо круглых скобок (т.е. разделителей γ и δ) используются теперь квадратные:



Опускать квадратные скобки при этом запрещается.

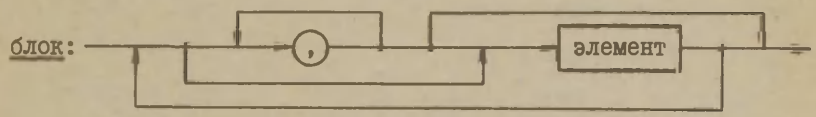
В конструкции "путь" для разделения констант можно использовать как запятую так и пробел. Кроме того, в этой конструкции появилась новая возможность определить движение в повторяющейся группе со списочной организацией - можно указать последовательность шагов по указателям "направо" и "вниз". Формально конструкция "путь" определяется теперь так:





Например, отрывок пути [1D 2R] указывает на второй элемент в подсписке первого элемента в соответствующей LIST-группе.

В конструкции "блок" разрешается в случае отсутствия элемента данных опускать символ * , но при этом нельзя опускать запятую в качестве разделителя, так как запятая, стоящая за запятой или в начале блока, считается признаком отсутствия данных. Во всех остальных случаях элементы данных можно разделить как запятой, так и пробелом. Формальное определение конструкции "блок" выглядит теперь так:



Если среди данных или ключей встречается атом типа TEXT или HEX, то соответствующее значение должно быть в DIL-программе оформлено в виде стринга. Исключение составляют лишь те случаи, когда значение атома типа TEXT является словом (идентификатором длиной не более восьми символов), или значение атома типа HEX содержит только десятичные цифры.

2. Семантика операций

При реализации несколько уточнялась и семантика операций языка DIL, по сравнению с ее описанием в [2]. По существу все соответствующие изменения сводятся к уточнению понятий наличия или отсутствия данных в банке. Такие уточнения необходимы как для однозначного установления выполнимости различных операций, так и для более экономного генерирования вводимых или модифицируемых экземпляров.

Каждому атому можно при помощи средств языка DIL присвоить или конкретное значение, или же значение * (т.е. признак отсутствия конкретного значения). Если имело место одно из этих присвоений, то можно сказать, что рассматриваемый атом создан. Если же атому еще ничего не присвоили, то атом не создан в том смысле, что в физической записи нет соответствующего кодослова (см. [4]).

Аналогично подлежат созданию и все остальные вершины - корни групп. При этом любую вершину можно создать лишь тогда, когда ее отец уже создан или создается в этом же операторе. Корень записи можно создать всегда, если только создан (открыт) соответствующий файл.

Так как носителями самих данных являются только атомы, то корню группы нельзя присвоить конкретное значение. Эквивалентной операцией, создающей корень группы, является присвоение конкретного значения хотя бы одному элементу соответствующей группы. При этом следует учитывать, что присвоение конкретных значений только некоторым элементам группы автоматически вызывает присвоение значения * всем остальным элементам той же группы, оставшимся без конкретного значения.

Элементами группы могут, конечно, быть и корни других групп. Поэтому в языке DIL допускается присвоение значения * корню группы. Такое присвоение может быть явно написано в DIL-программу, а может также возникнуть косвенно в результате других присваиваний. Семантика указанного присвоения зависит от типа соответствующей группы.

Присвоение значения * корню простой группы можно истолковывать*) как создание корня и присвоение признака отсутствия значения всем вершинам ветки (см. [1] стр. 13) этой группы.

Семантика присвоения значения * корню альтернативной группы зависит от значения соответствующего ключа выбора (см. [1] стр. 27). Если ключ выбора имеет значение *, то рассматриваемое присвоение оставляет корень группы не созданным. Если же ключ выбора имеет конкретное значение, то корень альтернативной группы создается и происходит еще присвоение значения * соответствующему одному элементу этой группы.

Если значение * присваивается (явно или косвенно) корню повторяющейся группы, то это никаких других присваиваний не порождает. Единственным результатом такого присвоения является создание корня этой группы. Однако, корень не создается тогда, когда в этой группе количество повторений или верхняя граница индекса задается в виде ссылки на атом со свойством CONST, а этот атом не имеет конкретного значения.

---

*) В конкретной реализации описываемые присвоения не всегда физически проводятся, но подробности такой "экономии" для пользователя не существенны.

Экземпляры можно ввести только в такую повторяющуюся группу, корень которой создан (или создается во время этого ввода). Для создания экземпляра необходимо присвоить конкретное значение хотя бы одному атому в ветке этого экземпляра - если группа имеет доступ, то конкретные значения должны получить все ключевые атомы. Только в случае массива допускается и создание "пустого" экземпляра, т.е. присвоение значения * всему экземпляру. Даже более, если хотя бы один экземпляр массива получает конкретное значение, то все остальные экземпляры получают и физически значение * .

Для ввода и корректировки данных в языке DIL имеется четыре операции: добавление (код операции A), замена (R), удаление (D) и включение (S). Семантика этих операций приведена уже в [2] (стр. 87-88). Указанные выше уточнения иллюстрируются конкретными примерами. Первая группа примеров опирается на следующую легенду (см. [2] стр. 78-79):

```
LEG КЛАСС KEY=НОМЕР TEXT PICT=15
  *1 НОМЕР PICT=3
  *1 КЛАССРУК
   *2 ФАМИЛИЯ
   *2 ИМЯ
   *2 ОТЧЕСТВО
  *1 ПРЕДМЕТЫ REP
  *1 КОЛПРЕД CONST
  *1 УЧЕНИК REP KEY=ФАМИЛИЯ,ИМЯ SORT
   *2 ИМЯ
   *2 ФАМИЛИЯ
   *2 СРОЦ PSEUDO REAL
   *2 ДАТАРОЖД DATE
   *2 ОЦЕНКИ ARRAY[4,КОЛПРЕД] MAT MAX=5
END
```

Например, оператор

```
## S  КЛАСС '1А')
  # '1А' *  (РУССКИЙ;'АРИФМЕТИКА') 1Ø
    (ИВАН ПЕТРОВ 171Ø73 * ;
     ИЛЬЯ ИВАНОВ 1ØØ873 * )
```

создает запись о 1А классе: если такая запись уже имелась, то вводимая заменяет ее. В связи с этим примером следует обратить внимание на то, что ключи записи всегда должны быть указаны в операторе до начала порций - даже тогда, когда они являются локальными (см. [2] стр. 80) и повторяются в данных. Такое требование ставилось во время реализации в целях упрощения работы с внешними накопителями.

Если в DIL-программе теперь имеется оператор

```
## A  КЛАСС '1А'
  # .КЛАССРУК.ИМЯ) НИКОЛАЙ
  # .УЧЕНИК)  ПЕТР СИДОРОВ * *
  # .ПРЕДМЕТЫ  5)  ПЕНИЕ
  # .ОЦЕНКИ ИВАН ИВАНОВ  1  2) 4
```

то он работает с той же записью - вводит имя классного руководителя и еще одного ученика. Последние две порции оператора, однако, окажутся невыполнимыми. Третья порция потому, что в группе ПРЕДМЕТЫ нет еще пяти экземпляров и нельзя добавить новый экземпляр за пятым. Четвертая порция невыполнима потому, что еще не создан экземпляр ученика Ивана Иванова и тем самым в этом экземпляре не создан корень группы ОЦЕНКИ (сама эта порция не создает экземпляра, так как конструкция "путь" не является носителем вводимой информации).

Если рассматриваемая DIL-программа содержит еще оператор

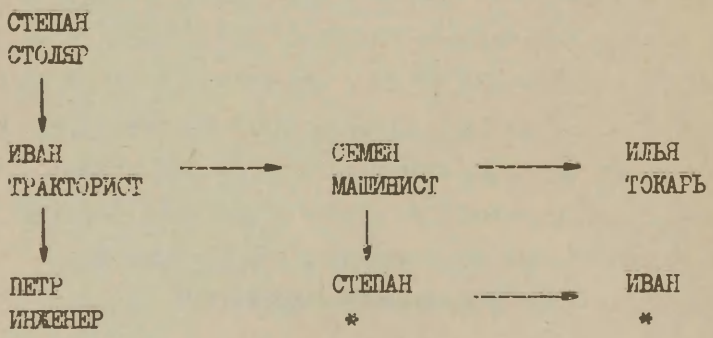
```
** R  КЛАСС '1А'
  * .КЛАССРУК.ФАМИЛИЯ) ПЕТРОВ
  * .КЛАССРУК) ПЕТРОВ ИВАН ИВАНОВИЧ
```

то в этом операторе первая порция окажется невыполнимой, а вторая заменит данные о классном руководителе на новые. Если же в последнем операторе код операции R заменить на A, то невыполнимой окажется вторая порция, а первая вводит фамилию классного руководителя.

Вторая группа примеров будет опираться на следующую легенду (в этой легенде учтены те изменения языка RDL, которые введены в статье [3]):

```
LEG ДИНАСТИЯ KEY=НОМЕР
  *1 НОМЕР INDEX
  *1 ЧЛЕН LIST TEXT PICT=15
   *2 ИМЯ
   *2 ПРОФЕССИЯ
END
```

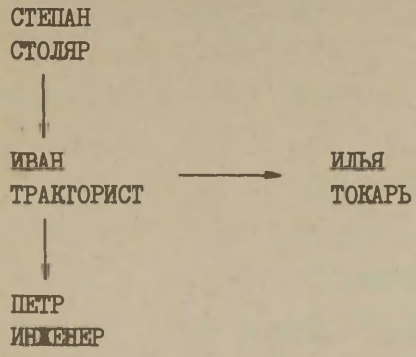
Допустим, что запись о 3-ей династии находится в следующем состоянии:



Применение к этой записи оператора

```
* * D  ДИНАСТИЯ.ЧЛЕН З [1D 1R]
```

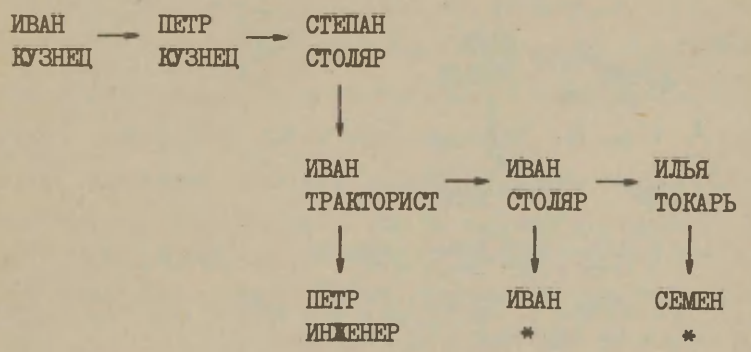
преобразует ее к виду:



(так как элемент списка модифицируется вместе со всеми подчиненными ему подсписками). Последующее применение оператора

```
* * A  ДИНАСТИЯ.ЧЛЕН З
  *) [ИВАН КУЗНЕЦ; ПЕТР КУЗНЕЦ ]
  *  [2R 1D 1R])[ИВАН СТОЛЯР [ИВАН * ]]
  *  [2R 1D 2R 1D]) [СЕМЕН * ]
```

преобразует запись к виду

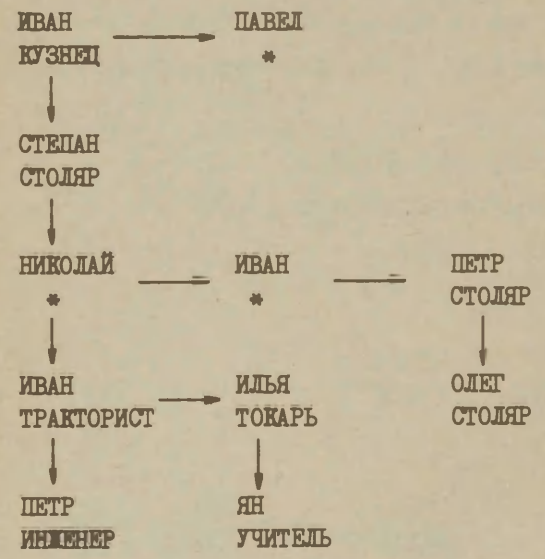


Особый случай имеет место тогда, когда при операции A конструкция "спецпуть" заканчивается шагом ØR (в остальных операциях такой шаг игнорируется). Если указанный экземпляр отсутствует, то создается новый подсписок; действия в противном случае описываются следующим примером.

Если вместо предыдущего оператора применить оператор

```
* * A  ДИНАСТИЯ.ЧЛЕН 3
  * [ØR]) [ИВАН КУЗНЕЦ; ПАВЕЛ * ]
  * [2D ØR]) [НИКОЛАЙ * ; ИВАН * ; ПЕТР * [ОЛЕГ СТОЛЯР]]
  * [3D 1R 1D ØR]) [ЯН УЧИТЕЛЬ ]
```

то рассматриваемая запись будет выглядеть так:



Если путь в LIST-группе выходит за пределы списка более чем на один шаг, то соответствующий оператор (порция) пропускается и на АЦПУ выдается сообщение.

3. Интерпретирование DIL-программы

При реализации языка DIL использовалась система построения трансляторов WIRTH, созданная в ВЦ ТГУ. Если DIL-программа содержит синтаксические ошибки, то СДТ WIRTH их обнаруживает; в противном случае она вырабатывает разреженное дерево анализа DIL-программы и передает управление DIL-интерпретатору.

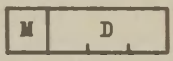
DIL-интерпретатор обходит разреженное дерево анализа DIL-программы "слева-направо, сверху-вниз", совершая при этом в каждой вершине действия, соответствующие ее семантике. Единицей выполнения в языке DIL является порция (или оператор, не содержащий порций). Двигаясь по дереву анализа DIL-интерпретатор собирает информацию, необходимую для выполнения порции. Закончив обход поддерева, соответствующего одной порции, DIL-интерпретатор обращается к системным программам, выполняющим действия с записью. После их выполнения интерпретатор анализирует результаты работы с записью (если нужно, печатает необходимые сообщения) и переходит затем к обработке следующей порции.

Если в новом операторе должна обрабатываться новая запись (не та, что в предыдущем), то старая запись закрывается и открывается новая. Эти действия связаны с внешними накопителями и выполняются сравнительно медленно. По этой причине в целях повышения скорости обработки данных рекомендуется провести всю работу с одной и той же записью в соседних операторах. Кроме того, группы операторов, работающих с записями одной легенды, должны в DIL-программе располагаться друг за другом.

Все программы DIL-интерпретатора являются реентерабельными, что позволяет эксплуатировать его в системе мультидоступа. Кроме того, некоторые программы являются рекурсивными, что существенно упростило обработку структур данных.

В разреженном дереве анализа DIL-программы данные представлены в виде поддерева. Такое представление, однако, не приемлемо системным программам и содержит избыточную информацию. Поэтому данные преобразуются в специальное промежуточное представление, которое во всей системе PAMA используется для обмена данными с системными программами.

Промежуточное представление генерируется DIL-интерпретатором, а системным программам передается ссылка на него. Эта четырехбайтная ссылка имеет вид



,

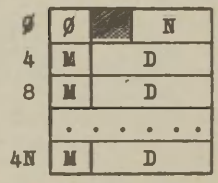
где поля M и D интерпретируются следующим образом:

если M ≠ Ø, то D является указателем на M-байтный атом;

если M = Ø и D = Ø, то это признак отсутствия данных;

если M = Ø и D ≠ Ø, то D является указателем на данные в промежуточном представлении.

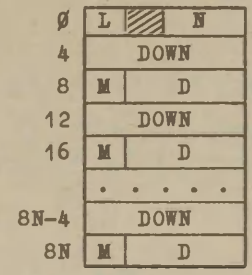
В промежуточном представлении, как и в языке DIL, структура данных бывает двух типов - блочного и спискового. Данные блочной структуры представляются T-блоком, который может быть связан с другими T-блоками. Каждый T-блок имеет вид:



,

где N - число элементов в группе, а поля M и D интерпретируются также, как и в вышеупомянутой ссылке (в случае M = Ø, D ≠ Ø поле D является указателем на другой T-блок).

Данные списковой структуры представлены L-блоком, который может быть связан с другими L-блоками или же T-блоками. Каждый L-блок имеет следующий вид:



где N - число элементов-братьев в списке. Каждому элементу соответствуют в L-блоке два слова. Из них DOWN является указателем на L-блок подсписка этого элемента (если DOWN = Ø, то подсписок пуст), а следующее слово дает информацию о значении элемента, причем поля M и D интерпретируются также, как в блочной структуре.

Например, если данные (вводимые по приведенной выше легенде КЛАСС) представлены в DIL-программе блочной структурой

```
'8_Б' (ИВАНОВ ИВАН ИВАНОВИЧ) (РУССКИЙ;МАТЕМАТИКА) 2
('ПЕТР' 'РОМАНОВ' 181265 (3  3; 4  4; 4  3; 3  * );
 'РОМАН' 'ПЕТРОВ' 15Ø565  * )
```

то соответствующее промежуточное представление схематически изображено на рис. 1. В этом примере следует обратить внимание на разницу между представлением данных типа TEXT в виде стринга и идентификатора.

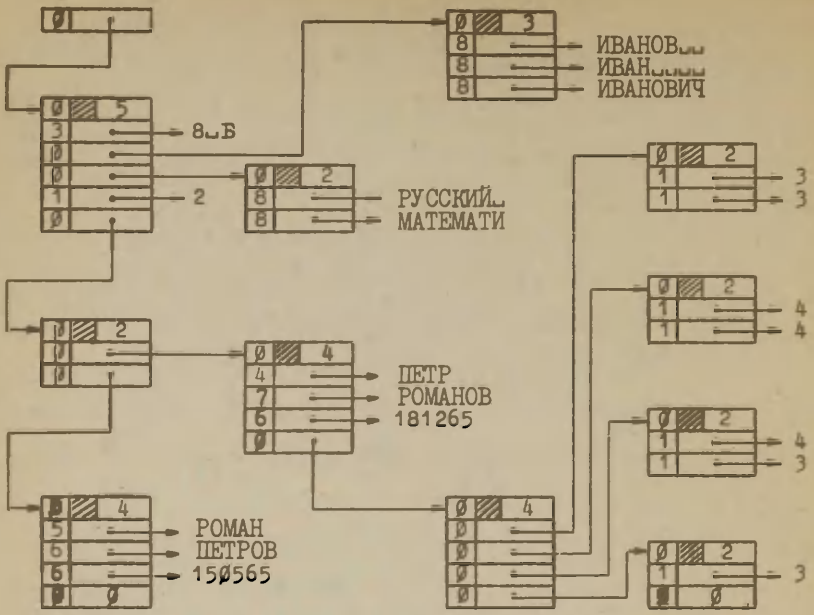

ИВАНОВ
ИВАН
ИВАНОВИЧ
8 Б
2
РУССКИЙ
МАТЕМАТИ
3
3
4
4
4
3
3
ПЕТР
РОМАНОВ
181265
РОМАН
ПЕТРОВ
150565

Рис. 1.

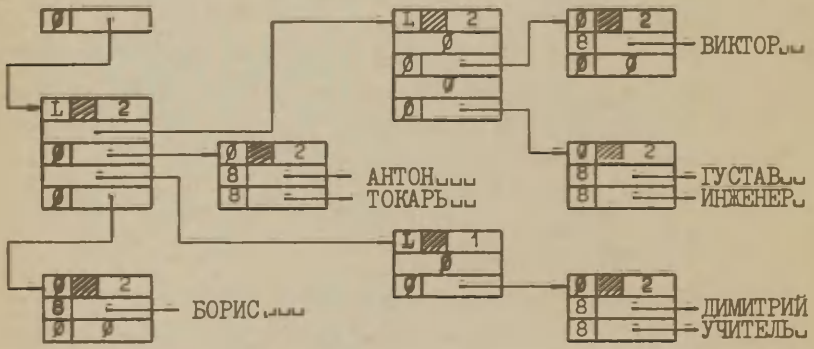

ВИКТОР
АНТОН
ТОКАРЬ
ГУСТАВ
ИНЖЕНЕР
БОРИС
ДИМИТРИЙ
УЧИТЕЛЬ

Рис. 2.

Если по легенде ДИНАСТИЯ вводится списковая структура

```
[АНТОН ТОКАРЬ [ВИКТОР * ; ГУСТАВ ИНЖЕНЕР];
 БОРИС * [ДИМИТРИЙ УЧИТЕЛЬ]]
```

то промежуточное представление принимает вид, изображенный на рис. 2.

Память для генерирования промежуточного представления данных распределяется так, что размеры вводимых в одной порции данных ограничены только размерами памяти машины. Разумное написание DIL-программы позволяет создавать структуры данных практически любых размеров.

Литература

1. Изотамм А., Каазик Ю., Томбак М., Язык определения записи. Труды ВЦ ТГУ, 1978, № 41, 7–64.
2. Каазик Ю., Ээльма П., Ввод и корректировка данных. Труды ВЦ ТГУ, 1978, № 41, 75–96.
3. Изотамм А., Дерево описания записи в системе РАМА. Труды ВЦ ТГУ, 1980, № 43, 3–35.
4. Изотамм А., Физическое представление записи в системе РАМА. Труды ВЦ ТГУ, 1980, № 43, 36–54.

РЕАЛИЗАЦИЯ ЯЗЫКА МАНИПУЛИРОВАНИЯ ДАННЫМИ

А. Рауп

Настоящая статья является продолжением статьи [1] и от читателя предполагается знакомство с последней. Здесь рассматриваются вопросы реализации языка DML, предназначенного для манипулирования данными в системе РАМА. Основное внимание при этом уделяется объяснению семантики некоторых конструкций языка, а также приводятся изменения и дополнения языка, сделанные во время реализации.

1. Транслятор и интерпретатор языка DML

При выборе подходящего метода реализации языка исходными являлись следующие соображения. Язык DML предназначен в основном для поиска и пересылки данных, а не для вычислений. Поэтому, естественно ожидать, что бо́льшая часть времени выполнения DML-программы уходит на работу разных системных программ, таких как программы ввода-вывода, локализации места в записи, динамического распределения памяти и т.д. Отсюда и вытекает, что наиболее удобным средством для реализации языка DML является интерпретатор.

Выполнение DML-программ проходит в два этапа. На первом этапе исходная программа, написанная на языке DML, транслируется в специальное внутреннее представление. Системную программу, выполняющую эту работу, будем называть DML-транслятором. Второй этап состоит в выполнении протранслированной программы путем интерпретирования внутреннего представления программы. Эту работу выполняет т.н. DML-интерпретатор.

При написании DML-транслятора использована система построения трансляторов WIRTH, созданная на ВЦ ТГУ. Система WIRTH выполняет синтаксический анализ исходной DML-программы и вырабатывает разреженное дерево анализа программы. Это дерево является исходной информацией для DML-транслятора, который проводит семантический анализ программы и преобразует дерево анализа.

Внутреннее представление программы получается путем некоторого преобразования дерева анализа, которое при этом связывается с разными необходимыми таблицами (таблицы констант, составных имен и т.п.). Каждая вершина полученного дерева представляет собой 16-байтовое поле, схематическая структура которого приведена на рисунке 1.

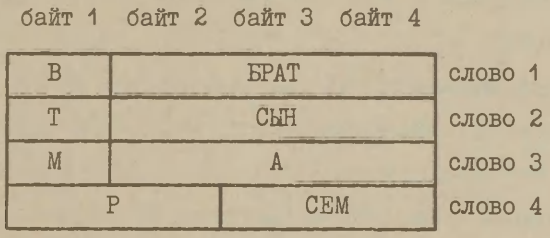


Рис. 1.

10

Поле БРАТ содержит при В = 0 ссылку на соседнюю вершину, а при В = 1 ссылку на отца. Поле СЫН содержит ссылку на первую подчиненную вершину, или же нуль в случае висящей вершины. Поле СЕМ предназначено для размещения т.н. семантического кода вершины, от конкретного значения которого зависит точное назначение полей Т, М, А и Р.

Подробный разбор всех возможных значений этих полей здесь можно не приводить. Отметим лишь, что обычно поле Т используется для хранения числа подчиненных вершин, а поле А содержит ссылку еще на некоторую вершину дерева, семантически связанную с данной вершиной. При этом поле М (точно таким же образом как поле В) уточняет, следует ли движение по ссылке А истолковывать как движение вверх, или же как движение направо.

В качестве примера приводим внутреннее представление условного оператора

IF C THEN S1 ELSE S2 FI

где через С обозначено некоторое условие, а через S1 и S2 операторы, входящие в состав условного оператора. Результат транслирования такого условного оператора схематически показана на рисунке 2.

На этом рисунке символ ОР условно обозначает семантический код оператора, а IF - семантический код условия. Следует учитывать, что верхняя вершина приведенного рисунка является корнем поддерева рассматриваемого условного оператора, а остальные вершины - корнями (опущенных на этом рисунке) поддеревьев условия С и операторов S1 и S2.

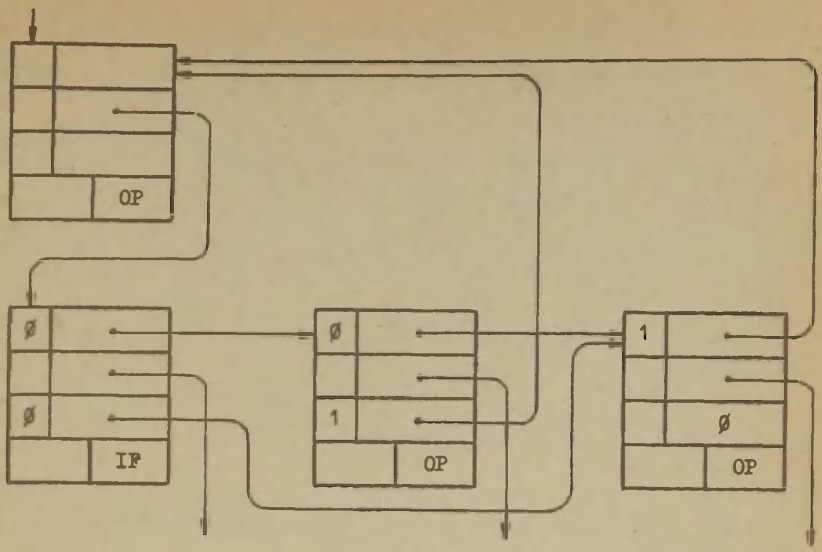


Рис. 2.

DML-интерпретатор начинает свою работу с анализа заказа, в котором указываются конкретные имена файлов, а также определяется соответствие между файлами и именами комплектов. В заказе можно еще присвоить конкретные значения переключателям программы.

DML-интерпретатор вводит необходимые FDL- и RDL-легенды и начинает проход дерева программы в прямом порядке. При этом движение вниз или направо не вызывает никаких действий. Но находясь в висящей вершине, или же попадая в вершину движением вверх, интерпретатор будет обращаться к соответствующим подпрограммам, определяемым семантическим кодом вершины. Обычный порядок прохождения дерева может иногда быть нарушен, так как некоторые семантические подпрограммы требуют в конце своей работы дальнейшего движения по ссылке A вместо

ссылки БРАТ. Для обмена информацией и хранения промежуточных результатов все семантические подпрограммы при этом используют один общий стек.

Покажем в качестве примера, как проходит интерпретация условного оператора, представленного на рисунке 2. Выполняя этот оператор, интерпретатор сперва проходит поддерево условия С. Результат вычисления соответствующего логического значения помещается в стек в качестве верхнего элемента и при возвращении в корень поддерева переходят к выполнению семантической подпрограммы для IF, которая имеет следующую структуру:

1. проверить, является ли верхний элемент стека логической константой "true";

2. если да, то выбросить из стека верхний элемент и идти дальше по ссылке БРАТ;

3. если нет, то выбросить из стека верхний элемент и идти дальше по ссылке А.

Таким образом, в зависимости от значения условия С далее проходят либо поддерево оператора S1, либо поддерево оператора S2. В обоих случаях проход поддерева кончается работой семантической подпрограммы для OP, которая выполняет следующие действия:

1. если поле А равно нулю, то идти по ссылке БРАТ;

2. в противном случае идти по ссылке А.

Выполнение DML-программы кончается, когда интерпретатор попадает обратно в корень дерева программы.

2. Понятие типа в языке DML

В описании языка DML часто встречается конструкция "выражение", образуемое из констант без знака, имен переменных и функций. Синтаксис языка не предъявляет никаких ограничений на то, какие константы и переменные могут быть включены в одно выражение или каким требованиям должны отвечать выражения, встречающиеся в операторах. Однако, такая полная свобода вызвала бы большие трудности при написании транслятора (из-за ввода разных автоматических преобразований типов) и была бы кроме того непрактичной, включая множество ненужных комбинаций. Во избежание этих трудностей вводятся понятия типов данных и выражений, а также устанавливаются соответствующие ограничения, которые должны быть соблюдены при написании DML-программ.

2.1. Типы скалярных величин. Скалярными величинами в выражениях языка DML являются атомы соответствующих легенд, константы и переключатели. По языку RDL одним из свойств атома является его тип (см. [2], стр. 18). Несколько иное понятие типа атома введем и в язык DML.

В языке DML в один тип объединяются те атомы, которые имеют одинаковое внутреннее представление и для которых существуют одинаковые правила обращения, например, проведения арифметических операций. Заметим еще, что сечение атома имеет одинаковый с ним тип. Всего различаются атомы пяти типов.

Двоичными целыми числами (в дальнейшем обозначим их ключевым словом INT) считаются все те атомы, в описаниях которых встречается одно из ключевых слов INT, NAT, CONST или

INDEX. Внутренним представлением таких чисел в памяти ЭВМ является число с фиксированной точкой длиной в слово или полуслово (в зависимости от возможного запаса значений).

Десятичными целыми числами (DEC) считаются атомы типа DEC, DATE и FDATE. Значения этих атомов хранятся в виде упакованных десятичных чисел.

Вещественными числами (REAL) считаются только атомы, которые описаны как атомы типа REAL. Их значения имеют формат чисел с плавающей точкой, а длина (полуслово, слово или двойное слово) зависит от возможного запаса значений.

Строками (TEXT) считаются атомы, в описании которых встречается ключевое слово TEXT. Значениями этих атомов являются строки символов, которые хранятся в коде ДКОИ на полях длиной до 255 байтов.

Шестнадцатеричными числами (HEX) считаются атомы, тип которых в легенде определен словом HEX. Шестнадцатеричные числа по существу являются битовыми строками, длина которых кратно полубайту.

В легенде можно использовать еще атомы типа EXTERNAL, не входящие ни в один из названных выше типов. При помощи ОМЛ-программы таким атомам нельзя присвоить значения и даже использовать имена этих вершин в выражениях. Поэтому, атомы типа EXTERNAL в дальнейшем не будут упоминаться.

Хотя атомы делятся на пять различных типов, константы всех этих типов явно написать нельзя. Исходя из определения константы (см. [1], стр. 110 и 116), при трансляции различаются константы только трех типов: INT, REAL и TEXT. Во всех тех местах, где понадобятся десятичные или шестнадцате-

ричные числа, следует использовать соответственно константы типа INT или TEXT. Значения этих констант автоматически преобразуются в нужный тип.

Для написания текстовых констант обычно приходится использовать конструкцию "стринг". Но апострофы можно опускать в конструкции "слово", т.е. когда текст имеет вид идентификатора и не совпадает с именами атомов из используемых в программе легенд, именами переключателей или комплектов, а также ключевыми словами языка DML (такими, как IF, DO, STOP, REPL, LEGEND, FOR и т.д.).

Так, например, следующие операторы присвоения написаны правильно (в предположении, что X имя комплекта, а T является атомом типа TEXT):

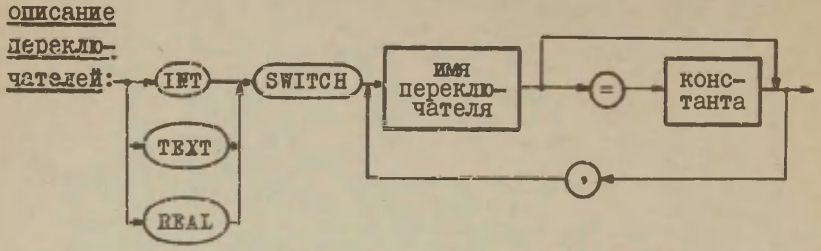
```
X.T := '3Ø'
X.T := 'M3Ø'
X.T :=  M3Ø
X.T := 'АМСТЕРДАМ'
```

но следующие неправильно:

```
X.T := 3Ø
X.T := АМСТЕРДАМ
```

Первый из двух последних примеров порождает ошибку уже при транслировании, так как 3Ø является константой типа INT, а не TEXT. Обрабатывая последний пример, транслятор ошибки не выдает, но при выполнении программы атом T получает значение 'АМСТЕРДА', т.е. пропадает последний символ. Итак, на место текстовой константы можно писать и идентификатор длиннее восьми символов, но из них учитываются только первые восемь.

Тип переключателя выясняется по описанию переключателя, которое имеет теперь новую форму (ср. [1], стр. 106):



Тип определяемых констант указывается первым ключевым словом, а константы, устанавливающие стандартные значения переключателей, должны иметь одинаковый с переключателями тип.

2.2. Преобразование типов. Во многих случаях, когда одновременно пользуются атомами различных типов, возникает вопрос преобразования типа. При реализации языка DML полагалось, что в таких случаях, как правило, каждый раз в программе следует явно указать нужное преобразование. Например, для преобразования вещественного числа в двоичное целое можно использовать функции ENTIER и ROUND. Функция ENTIER дает целое число, получаемое при отбрасывании дробной части. ROUND является функцией округления вещественного числа и определяется следующим образом:

$$\mathrm{ROUND}(X) = \begin{cases} \mathrm{ENTIER}(X+0.5), & \text{если } X \geq 0, \\ -\mathrm{ENTIER}(0.5-X), & \text{если } X < 0. \end{cases}$$

Например, если X является именем комплекта, A атомом типа REAL, а I1 и I2 атомами типа INT, то после выполнения составного оператора

```
DO  X.A := 3.7;
    X.I1 := ENTIER(X.A);
    X.I2 := ROUND (X.A)  OD
```

атомы I1 и I2 принимают соответственно значения 3 и 4.

Иногда тип все-таки приходится преобразовать автоматически, хотя бы из-за отсутствия явной возможности написать константы типа DEC и HEX. Поэтому решено, что при необходимости двоичные и десятичные целые числа автоматически преобразуются друг в друга (для этого используются команды Ассемблера CVD и CVB), а в шестнадцатеричное число преобразуется строковая величина при условии, что она имеет нужную форму, т.е. состоит из символов 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E и F, а длина не превышает 31 символа. Так как преобразование целых чисел в вещественные традиционно выполняется трансляторами, то это реализовано и в языке DML.

A \ B	INT	DEC	REAL	TEXT	HEX
INT	+	+			
DEC	+	+			
REAL	+	+	+		
TEXT				+	
HEX				+	+

Таблица 1.

В таблице 1 плюсами указаны все преобразования типа, которые при необходимости выполняются DML-интерпретатором ав-

томатически. В этой таблице B обозначает первоначальный тип и A тип результата преобразования (можно считать, что таблица покажет возможные типы операндов при операции A := B). В дальнейшем мы будем называть тип B допустимым по отношению к A, если тип B может автоматически преобразоваться в тип A.

2.3. Типы выражений. Арифметические операции реализованы пока только между скалярными операндами, принадлежащими к арифметическим типам, т.е. к INT, DEC и REAL. В таблице 2 приведены типы результатов операций A + B, A - B, A * B и A / B. Унарный минус сохраняет тип операнда.

A \ B	INT	DEC	REAL
INT	INT	INT	REAL
DEC	INT	DEC	REAL
REAL	REAL	REAL	REAL

Таблица 2.

Типом выражения считается тип его значения. Если выражение содержит арифметические операции, то тип выражения можно определить по таблице 2. В противном случае типом выражения будет тип его единственного члена.

В определениях языка DML понятие "выражение" встречается в конструкциях "сравнение", "фильтр" и "путь", а также еще в операторах выбора и присваивания. При этом использование нескалярных выражений разрешается пока только в операторах присваивания.

В случае операндов сравнения требуется, чтобы один из них имел допустимый тип по отношению к другому. Аналогичное требование ставится и для выражения, которое является членом фильтра или пути: оно должно иметь допустимый тип по отношению к типу соответствующего ключа. В операторе выбора все константы, стоящие перед внутренними операторами, должны быть одинакового типа, а выражение в начале оператора выбора должно иметь допустимый по отношению к этим константам тип.

2.4. Типы структур. Основными структурными понятиями в системе РАМА являются запись, группа и атом, причем запись можно также рассматривать как группу. Группы делятся на неповторяющиеся, альтернативные и повторяющиеся. Один экземпляр повторяющейся группы или один вариант альтернативной группы считается неповторяющейся группой.

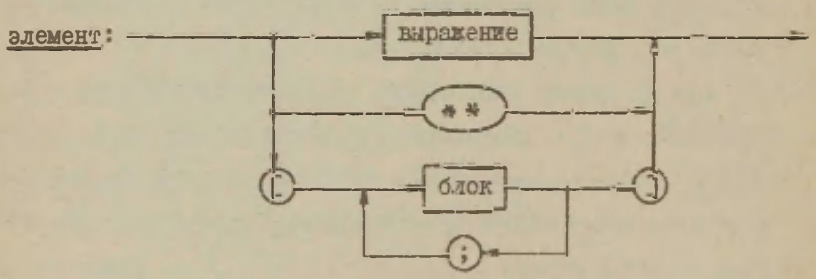
Две группы считаются группами одного типа, если они оба неповторяющиеся, оба альтернативные или оба повторяющиеся с одинаковой организацией (REP, LIST или ARRAY) и имеют одинаковое количество потомков, которые попарно, начиная слева, являются группами или атомами одного типа. Для массивов требуется еще одинаковое число индексов и их значений (если таковые указаны). Таким образом: две группы имеют одинаковый тип, если у них одинаковая структура и типы соответствующих атомов в поддеревьях совпадают.

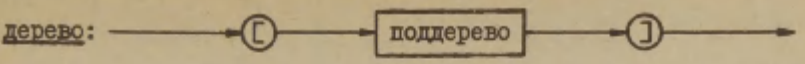
Группа имеет допустимый по отношению к другой группе тип, если у них одинаковая структура и атомы в поддереве первой группы имеют допустимый тип по отношению к типам соответствующих атомов другой группы. Например, по следующим трем отрывкам легенд

```
*2 A REP RAT        *2 B              *3 C
  *3 A1 REP           *3 B1 REAL        *4 C1 REP REAL
    *4 A2               *4 B2             *5 C2
    *4 A3               *4 B3             *5 C3
  *3 A4               *3 B4 NAT         *4 C4 NAT
```

можно установить, что одинаковый тип имеют группа B1 и экземпляр группы C1, а тип экземпляра группы A1 допустим по отношению к ним. Кроме того, группа A1 и экземпляр группы A имеют допустимый тип соответственно к группам C1 и C.

Обращаемся теперь к оператору присваивания (см. также [1], стр. 108-115). Первое из перечисленных в начале оператора имен переменных определяет для данного оператора требуемый тип данных. Другие имена переменных должны быть именами структур, имеющих типы, допустимые по отношению к этому типу. Выражения и константы в правой части оператора должны также иметь типы, допустимые по отношению к требуемому типу данных. Прежде, чем предложить соответствующие примеры, представим новые определения конструкций "элемент" и "дерево", так как в их синтаксисе круглые скобки заменены при реализации квадратными:





В приводимых примерах мы опираемся на описания групп A, B и C из предыдущего примера, предполагая, что имена комплектов для них соответственно X, Y и Z.

Правила использования типов не нарушены, например, в следующих операторах присваивания:

```
Y.B1 := [4, 5.3]
Z.C1, X.A1 := [Ø, * ; ROUND(Y.B3), 1]
```

В первом операторе первый элемент блока имеет тип, допустимый по отношению к типу атома B2, а тип второго элемента совпадает с типом атома B3. Во втором операторе тип группы A1 допустим по отношению к группе C1 и они могут одновременно находиться в левой части оператора (но не могут быть там в обратном порядке). Обеим этим группам рассматриваемым оператором присваивается два экземпляра.

При использовании имен повторяющихся групп в операторах присваивания следует учесть, что такое имя может обозначать либо один экземпляр группы, либо собрание всех имеющихся экземпляров этой группы. Например, имя X.A1 в левой части оператора присваивания само собой еще не указывает, хотим мы присвоить значение лишь одному текущему экземпляру, или же всей группе. В DML-программах эти случаи различаются следующим образом: если путь проходит до рассматриваемой группы, то ее имя обозначает текущий экземпляр; если же путь заканчивается уже в отцовской группе, то имени группы соответствует полное собрание экземпляров этой группы.

Например, если пути в комплектах X, Y и Z заканчиваются соответственно в отцовских группах групп A, B и C, то в результате выполнения операторов

```
FOR X.A(3) Z.C1 := X.A1;
REPL Z.C1(1) Z.C1 := Y.B1
```

первый экземпляр группы C1 имеет значение группы B1, а в качестве последующих экземпляров C1 взяты экземпляры группы A1 (из третьего экземпляра группы A), начиная со второго.

Когда желают исключить все экземпляры повторяющейся группы, то простейшим методом является присвоение всей группе пустого значения (т.е. звездочки). Например, оператор

```
FOR X.A(5) X.A1 := *
```

удаляет всю группу A1 из пятого экземпляра группы A. Заметим, что по первоначальному описанию языка DML (см. [1], стр. 113) такое действие запрещалось.

При трансляции оператора присваивания DML-транслятор обращает внимание только на типы атомов, а согласованность таких свойств, как длина, запас значений и максимальное значение не проверяется. Например, в случае отрывка легенды:

```
*5 A NAT MAX = 10
*5 B NAT MAX = 500
```

соответствующая DML-программа может содержать оператор X.A := X.B (где X имя комплекта), и даже оператор X.A := 90, однако сообщение об ошибке не выдается. Дело в том, что во время рассчета атомы в рабочей области могут иметь значения, превышающие ограничения, указанные в легенде. Лишь при вы-

полнении операции обновления перед пересылкой данных из рабочей области в буфер проверяют соответствие значений атомов соответствующим описаниям в легенде и отрицательный ответ отменяет обновление.

Что касается атомов, являющихся основными ключами групп, то запрещается указывать их имена в левой части оператора присваивания. Основным ключам значения присваиваются только операторами обновления, где эти значения указываются в конструкции "путь". Когда внутри оператора обновления корню обновляемой группы присваивается значение, то новые значения получают все атомы группы кроме основных ключей. Например, если легенда содержит следующее определение группы

```
*2 A REP NAT KEY=B
  *3 B
  *3 C
  *3 D
```

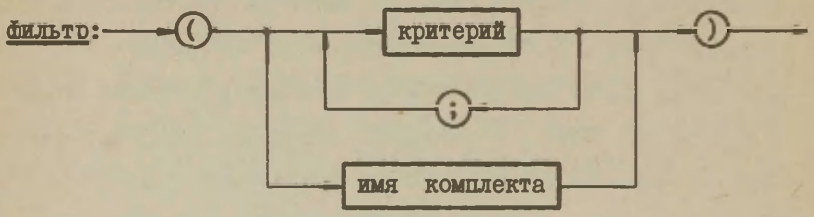
то оператор X.A := Z.A (где X и Z имена комплектов) равносилен следующему составному оператору

```
DO  X.B := ** ;
    X.C := Z.C;
    X.D := Z.D  OD
```

Если правую часть оператора присваивания необходимо задавать в виде блока, то на местах основных ключей должны стоять две звездочки, гарантирующие сохранение старого значения. Это значит, что в предыдущем примере вместо оператора X.A := Z.A можно писать X.A := [**, Z.C, Z.D], но нельзя писать просто X.A := [Z.C, Z.D] .

3. Операторы извлечения и обновления данных

При реализации языка DML некоторым изменениям подверглись как синтаксис, так и семантика операторов извлечения и обновления данных. Так, например, общий вид оператора цикла и его семантика остались прежними, но несколько изменились конструкции "фильтр" и "критерий":



Фильтр может теперь состоять только из имени комплекта: значения ключей для определения места берутся тогда из текущих значений ключей этого комплекта. Аналогичная возможность имеется и в операторах обновления (см. [1], стр. 135). Имя комплекта в фильтре или пути по существу является только упрощением записи, позволяющим опускать перечень имен всех ключей. Например, если отрывок легенды имеет вид:

```
*4 A REP KEY=B,C NAT
  *5 B
  *5 C
```

то начало оператора цикла

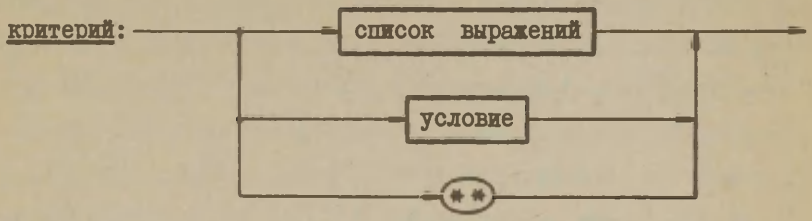
```
FOR X.A(1:5; *), Y.A(X)
```

является сокращением для

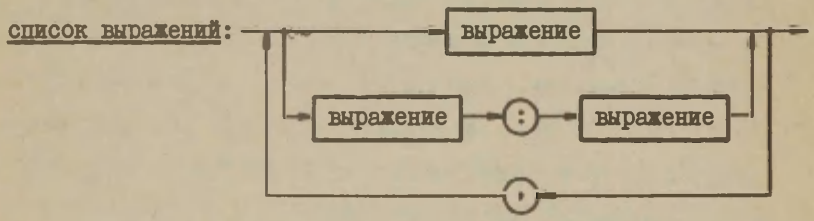
```
FOR X.A(1:5; *), Y.A(X.B; X.C)
```

Имя комплекта в конструкции "фильтр" и имя комплекта в конструкции "место" должны, конечно, соответствовать одной и той же легенде.

Разделителем критериев в фильтре является теперь не запятая, а точка с запятой. Это обусловлено изменением в синтаксисе конструкции "критерий":



Вместо "выражения" в этом определении появилось новое понятие "список выражений":



где все выражения должны иметь одинаковый тип.

Список выражений позволяет упростить некоторые условия. Например, если требуются те экземпляры группы A, в которых ключ B имеет значения 3, 5, 6, 7 и 10, то можно написать

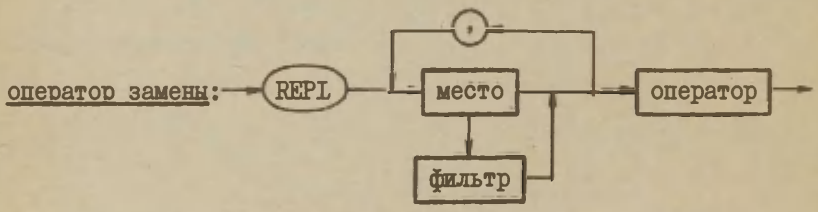
```
FOR X.A(3,5:7,10;*)
```

вместо более длинного варианта

```
FOR X.A(A=3|A>=5&A<=7|A=10;*)
```

В связи с этим примером следует указать еще на одну возможность использования переключателя. Целесообразным представляется разрешить задать в заказе в качестве значения переключателя не только одну константу, а список констант. Список констант имеет аналогичный синтаксис со списком выражений, где только вместо выражений встречаются константы. Если переключателю задается значение в виде одной константы, то этот переключатель можно использовать везде в выражениях DML-программы. Если же переключателю задается значение в виде целого списка констант, то он может встречаться только в конструкциях "фильтр" и "путь". Использование этого переключателя в других местах программы приводит к снятию задания. DML-транслятор, анализируя программу, следит за использованием переключателей и выдает сообщения, для каких переключателей в качестве значения можно задать список констант.

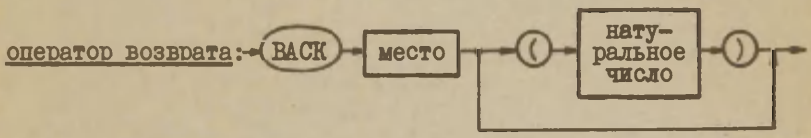
В первоначальном описании языка DML был определен один оператор обновления для замены, добавления и включения (см. [1], стр. 129). После введенных изменений будем говорить о трех разных операторах. Оператор замены имеет теперь одинаковый синтаксис с оператором цикла:



Главным нововведением при замене является то, что разрешается заменить экземпляры не только с указанием конкретного

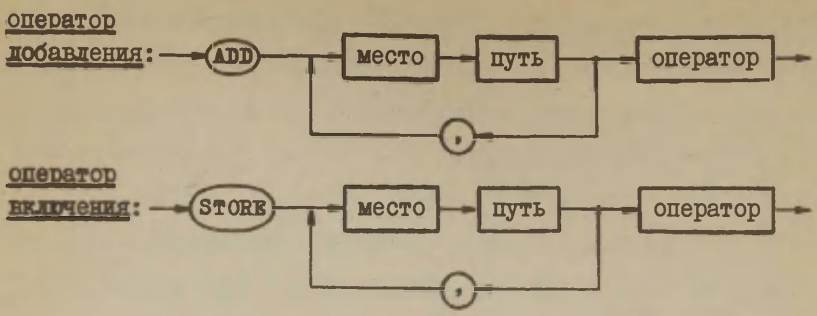
пути, а также с указанием фильтра, т.е. организовать цикл замены. Оператор замены работает теперь как оператор цикла, только после выполнения внутреннего оператора перед обращением к следующему экземпляру текущий экземпляр помещается из рабочей области обратно в буфер.

Для окончания циклической замены до исчерпывания экземпляров можно использовать оператор выхода. Если же во время обработки выясняется, что очередной экземпляр заменить не надо, но цикл продолжать все-таки следует, то для этого нужно иметь средство, обеспечивающее возвращение из внутреннего оператора в начало оператора замены для перехода к следующему подходящему экземпляру. В этих целях и введен еще один новый оператор:

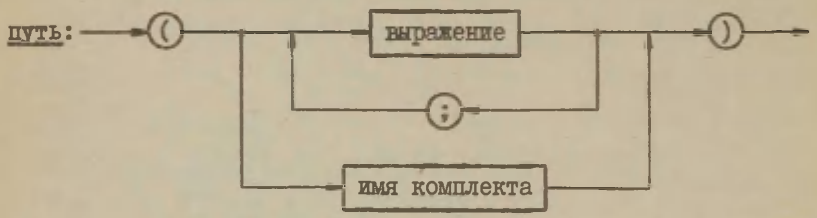


Оператор возврата можно использовать внутри операторов цикла и замены. Он обеспечивает возврат к подбору нового экземпляра. Натуральное число в скобках указывается тогда, когда оператор возврата находится внутри нескольких операторов цикла или замены, связанных с одним и тем же местом. Тогда это число указывает, на сколько таких операторов назад нужно возвратиться. Аналогичным образом указывается и место перехода для оператора выхода (см. [1], стр. 128).

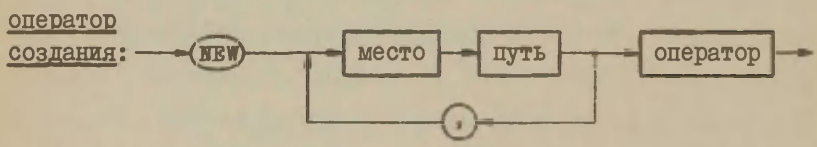
Операции добавления и включения изменениям не подвергались (см. [1], стр. 130), новые лишь их названия:



Так, как в фильтре разделителем является теперь точка с запятой, то соответствующее изменение следует ввести и в синтаксис конструкции "путь":



В ходе изменений выявилась необходимость добавить еще новый оператор обновления данных:



Соответствующая операция состоит в следующем. В рабочей области создается новый экземпляр со значениями ключей, указанными в пути. Все другие атомы группы получают пустые значения, а собрания подчиненных повторяющихся групп не существуют. Лишь после этого выполняется внутренний оператор. При завершении этой работы экземпляр группы помещается в буфер.

Создание и включение являются аналогичными операциями. Основное различие между ними состоит в том, что при включении всегда проверяется, имеется ли уже экземпляр с указанными в пути значениями ключей и при его наличии он становится текущим; при создании же такого контроля не проводят. Когда экземпляр с заданными в пути значениями ключей не существует в базе данных, то результаты создания и включения одинаковы. Если же такой экземпляр имеется, то при создании весь экземпляр заменяется на новый, а при включении все вершины старого экземпляра, которым во внутреннем операторе не присваивали новое значение, сохранят свои старые значения.

Например, если группа A описана следующим образом:

```
*2 A REP NAT
  *3 B
  *3 C
```

то после выполнения оператора

```
DO  STORE X.A(1)  X.B := 5;
    STORE X.A(1)  X.C := 4  OD
```

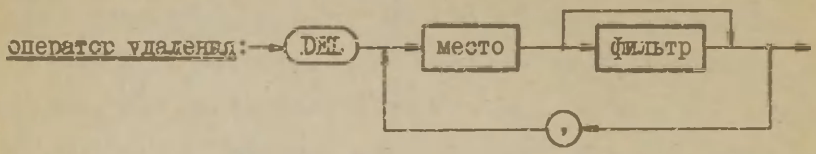
первый экземпляр группы имеет значение [5,4]. Однако оператор

```
DO  STORE X.A(1)  X.B := 5;
    NEW   X.A(1)  X.C := 4  OD
```

присвоит первому экземпляру значение [*,4].

В конструкции "путь" разрешается использование переключателей, которые своим значением могут получить список констант. При каждом выполнении оператора добавления, включения или создания значением соответствующего ключа выбирается тогда следующая константа из списка.

Несколько изменился в ходе реализации и оператор удаления. Если раньше удалить можно было только по одному экземпляру, указывая для этого конкретный путь (см. [1], стр. 138), то теперь разрешается и циклическое удаление данных:



Например, вместо того, чтобы написать оператор

```
FOR X(Y.A : Y.B) DEL X
```

можно теперь писать

```
DEL X(Y.A : Y.B)
```

Уточнить следует еще семантику операции обновления массивов. При обновлении повторяющихся групп важное значение имеет вопрос, существует ли уже экземпляр группы с заданными значениями ключей. В качестве ключей в массиве имеются индексы. Как только один из элементов (т.е. экземпляров) массива получает значение, то с этого момента будут существовать и все другие элементы массива со всевозможными значениями индексов, хотя они могут иметь пустое значение. Поэтому, вопрос о существовании элемента массива при обновлении сводится к тому, имеет ли он значение или нет. Когда элементом массива является структура, то этот элемент существует тогда, когда хотя-бы один атом структуры имеет непустое значение. Заметим, однако, что оператор цикла всегда проходит подряд все элементы массива, несмотря на существование или отсутствие их значений.

4. Определение языка

4.1. Терминальный альфавит. Элементы терминального алфавита языка DML слово (в дальнейшем i), натуральное число без знака (n), число (c) и стринг (s) совпадают с соответствующими элементами терминального альфавита языка BDL (см. [2], стр. 58–59). Кроме них в терминальный альфавит языка DML входят еще следующие элементы:

число без знака (r): натуральное или вещественное число без знака в обычном или полулогарифмическом виде;

разделители: , = ( ) [ ] : * ** := . + - # /
¬ ¬= < <= > >= | & ;

служебные слова: ADD BACK CASE DEL DML DO ELSE FI FO
FOR IF INT KEY LEAVE LEGEND NEW OD OF PROC REAL
REPL SET STOP STORE SWITCH TEXT THEN VAR WITH

4.2. Нетерминальный альфавит. Список элементов нетерминального альфавита несколько различается от понятий языка DML, приведенных в статье [1] и в настоящей статье, где в интересах лучшего понимания были использованы более общие понятия. Кроме вышеупомянутых изменений языка DML укажем еще на одно: в описание легенды нельзя уже непосредственно включить саму легенду – можно лишь указать имя готовой легенды (ср. [1], стр. 105). Это, как и все остальные изменения отражены как в нетерминальном альфавите, так и в продукциях.

В нетерминальный альфавит языка DML теперь входят следующие понятия (в порядке их определения):

программа, описания, описание, описание легенды, имена комплектов, описание переключателей, переключатели, константа, константа без знака, оператор, оператор присваивания, левая часть, составное имя, блок, элемент, блоки, выражение, одночлен, первичное выражение, функция, список параметров, оператор процедуры, оператор останова, оператор выхода, оператор возврата, оператор удаления, ограничения, ограничение, фильтр, список критериев, критерий, список выражений, промежуток, условие, конъюнкция, терм, сравнение, отношение, составной оператор, операторы, условный оператор, действие, оператор выбора, варианты, вариант, константы, оператор приставки, оператор цикла, оператор замены, оператор добавления, локализации, локализация, путь, выражения, оператор включения, оператор создания.

4.3. Продукции.

<программа> → DML i <описания> <оператор>

<описания> → <описание>

→ <описания> ; <описание>

<описание> → <описание легенды>

→ <описание переключателей>

<описание легенды> → LEGEND i

→ VAR LEGEND i

→ LEGEND i <имена комплектов>

→ VAR LEGEND i <имена комплектов>

<имена комплектов> → SET i

→ <имена комплектов> = i

→ <имена комплектов> , i

<описание переключателей> → INT SWITCH <переключатели>
→ TEXT SWITCH <переключатели>
→ REAL SWITCH <переключатели>

<переключатели> → i
→ i = <константа>
→ <переключатели> , i
→ <переключатели> , i = <константа>

<константа> → <константа без знака>
→ c

<константа без знака> → i
→ s
→ r
→ *

<оператор> → <оператор присваивания>
→ <оператор процедуры>
→ <оператор останова>
→ <оператор выхода>
→ <оператор возврата>
→ <оператор удаления>
→ <составной оператор>
→ <условный оператор>
→ <оператор выбора>
→ <оператор приставки>
→ <оператор цикла>
→ <оператор замены>
→ <оператор добавления>
→ <оператор включения>
→ <оператор создания>

<оператор присваивания> → <левая часть> := <блок>

→ <левая часть> := <дерево>

<левая часть> → <составное имя>

→ <левая часть> . <составное имя>

<составное имя> → i

→ <составное имя> . i

<блок> → <элемент>

→ <блок> , <элемент>

<элемент> → <выражение>

→ **

→ [<блоки>]

<блоки> → <блок>

→ <блоки> ; <блок>

<выражение> → <одночлен>

→ <выражение> + <одночлен>

→ <выражение> - <одночлен>

→ + <выражение>

→ - <выражение>

<одночлен> → <первичное выражение>

→ <одночлен> # <первичное выражение>

→ <одночлен> / <первичное выражение>

<первичное выражение> → (<выражение>)

→ <константа без знака>

→ <составное имя>

→ <функция>

<функция> → i ( )

→ i (<список параметров>)

<список параметров> → <выражение>

→ <список параметров> , <выражение>

<оператор процедуры> → PROC 1

→ PROC 1 (<список параметров>)

<оператор останова> → STOP

<оператор выхода> → LEAVE <составное имя>

→ LEAVE <составное имя> (n)

<оператор возврата> → BACK <составное имя>

→ BACK <составное имя> (n)

<оператор удаления> → DEL <ограничения>

<ограничения> → <ограничение>

→ <ограничения> , <ограничение>

<ограничение> → <составное имя>

→ <составное имя> <фильтр>

<фильтр> → (<список критериев>)

→ (1)

<список критериев> → <критерий>

→ <список критериев> ; <критерий>

<критерий> → <список выражений>

→ <условие>

→ **

<список выражений> → <выражение>

→ <промежуток>

→ <список выражений> , <выражение>

→ <список выражений> , <промежуток>

<промежуток> → <выражение> : <выражение>

<условие> → <конъюнкция>

→ <условие> | <конъюнкция>

<конъюнкция> → <терм>

→ <конъюнкция>& <терм>

<терм> → <сравнение>

→ (<условие>)

→ ¬(<условие>)

<сравнение> → <выражение> <отношение> <выражение>

→ KBY <отношение> <выражение>

<отношение> → =

→ ¬=

→ <

→ <=

→ >

→ >=

<составной оператор> → DO <операторы> OD

<операторы> → <оператор>

→ <операторы> ; <оператор>

<условный оператор> → IF <условие> THEN <действие>

<действие> → <оператор> FI

→ <оператор> ELSE <оператор> FI

<оператор выбора> → CASE <выражение> OF <варианты> FO

<варианты> → <вариант>

→ <варианты> ; <вариант>

<вариант> → <константы> : <оператор>

<константы> → <константа>

→ <константы> , <константа>

<оператор приставки> → WITH i <оператор>

<оператор цикла> → FOR <ограничения> <оператор>

<оператор замены> → REPL <ограничения> <оператор>

<оператор добавления> → ADD <локализации> <оператор>

⟨локализации⟩ → ⟨локализация⟩

→ ⟨локализации⟩ , ⟨локализация⟩

⟨локализация⟩ → ⟨составное имя⟩

→ ⟨составное имя⟩ ⟨путь⟩

⟨путь⟩ → (⟨выражения⟩)

→ (i)

⟨выражения⟩ → ⟨выражение⟩

→ ⟨выражения⟩ ; ⟨выражение⟩

⟨оператор включения⟩ → STORE ⟨локализации⟩ ⟨оператор⟩

⟨оператор создания⟩ → NEW ⟨локализации⟩ ⟨оператор⟩

Литература

1. Каазик Ю., Рауп А., Язык манипулирования данными. Труды ВЦ ТГУ, 1978, № 41, 97-140.

2. Изотамм А., Каазик Ю., Томбак М., Язык определения записи. Труды ВЦ ТГУ, 1978, № 41, 7-64.

ИСПОЛЬЗОВАНИЕ СИСТЕМЫ РАМА ДЛЯ РЕАЛИЗАЦИИ СИСТЕМЫ СТАТИСТИЧЕСКОЙ ОБРАБОТКИ ДАННЫХ

А. Рауп

В вычислительном центре ТГУ создана система статистической обработки данных, общее описание которой приведено в статье [4]. Всю эту систему можно условно разбить на две части. Одна часть системы занимается вводом и корректировкой данных, а также преобразованием введенных данных путем их слияния, выделения подмножеств и т.д. Вторая часть системы охватывает программы, реализующие разные конкретные статистические методы.

Представляется естественным использовать для реализации хотя бы первой части этой системы средства, созданной в ВЦ ТГУ СУБД РАМА. Для ввода и корректировки данных в СУБД РАМА предназначен язык DIL (см. [7]). Более сложные же преобразования данных можно при этом произвести с помощью программ, написанных на языке DML (см. [2]).

В данной статье рассматриваются примеры использования системы РАМА в этом направлении. Приводимые RDL-легенды и DML-программы можно считать иллюстрациями к описаниям этих языков, опубликованных в статьях [1], [2] и [3].

1. Описание данных системы статистической обработки

При использовании системы РАМА в первую очередь необходимо описать структуру данных, применяя для этого язык RDL. В системе статистической обработки данных считается, что всякую совокупность данных можно представить в виде матрицы или же совокупности матриц. Основной единицей данных в этой системе является т.н. стандартная матрица.

Описание стандартной матрицы приведено в [5] (на стр. 76-79), где ее структура представлена в виде деклараций языка PL/I. Представленному там описанию соответствует следующая RDL-легенда:

```
LEG STMATRIX
  *1 NAME TEXT PICT=8
  *1 BASENAME TEXT PICT=8 PSEUDO
  *1 DMY DATE
  *1 TEXTLN NAT
  *1 M CONST
  *1 N CONST
  *1 REGISTER REP=M NAT SORT
  *1 VARIABLE ARRAY[M] REAL
    *2 VMEAN PSEUDO
    *2 VS PSEUDO
    *2 VMAX PSEUDO
    *2 VMIN PSEUDO
    *2 VSIZE PSEUDO
    *2 VTEXT TEXT
      *3 TYP SCOPE=[R,A,B,C]
      *3 PRECIS NAT
      *3 VNAME PICT=25
      *3 FORMAT
    *2 VALUE ARRAY[N]
END
```

Имена вершин имеют в этой легенде следующие значения.

NAME обозначает имя стандартной матрицы, т.е. имя соответствующего однозаписного файла.

BASENAME содержит имя первичного файла, исходя из которого образован настоящий. Если файл первичный, то BASENAME и NAME совпадают (первичными считаются файлы, содержащие непосредственно данные измерений). BASENAME описан как PSEUDO потому, что этот атом заполняется автоматически программами, производящими преобразования данных.

Время составления файла пишется в атом DMY.

Атом TEXTLN предназначен для хранения максимальной допустимой длины текста, содержащегося в атоме FORMAT.

M и N указывают размеры матрицы, обозначая соответственно число признаков и число индивидов.

Повторяющийся атом REGISTER хранит числовые идентификаторы (или просто номера) признаков.

В массиве VARIABLE содержатся как значения, так и общие характеристики имеющихся M признаков. Непосредственно после ввода данных для каждого признака вычисляются следующие общие характеристики:

VMEAN - среднее,
VS - стандартное отклонение,
VMAX - максимальное значение признака,
VMIN - минимальное значение признака,
VSIZE - объем выборки

(первые два из этих характеристик вычисляются только для численных и упорядоченных признаков).

В группе VTEXT хранится текст, характеризующий данный признак. Этот текст разбивается на четыре компонента.

В атоме TYP указывается тип признака. Система статистической обработки данных различает признаки четырех типов: признаки типов R и A являются количественными, качественные упорядочиваемые признаки принадлежат к типу B, а неупорядочиваемые к типу C.

Содержимое атома PRECIS означает для R-признаков точность печати результатов, а для всех остальных признаков предполагаемый максимум значений.

VNAME обозначает имя признака, которое будет использовано только при печати результатов.

Атом FORMAT содержит определение множеств классификации признака. Это определение не используется при преобразовании данных, а только в некоторых программах непосредственной статистической обработки.

Массив VALUE содержит значения признака.

Приведенное описание дает нам только общее представление об имеющихся данных и их структуре. Так как все данные собраны здесь в одну запись, то это описание является весьма неудобным для практического применения.

Рассматривая данные стандартной матрицы видим, что некоторые из них являются общими для всей матрицы. Выделив эти данные в отдельную запись, можно остальные данные, связанные с конкретными признаками, описать также как записи. Таким образом, удобнее дать новое описание стандартной матрицы с помощью двух легенд, из которых STANDMX опишет общие данные:

```
LEG STANDMX
  *1 NAME TEXT PICT=8
  *1 BASENAME TEXT PICT=8 PSEUDO
  *1 DMY DATE
  *1 TEXTLN NAT
  *1 M NAT
  *1 N CONST
  *1 REGISTER REP NAT SORT
END
```

Данные одного признака (строки матрицы) описаны с помощью легенды VARIABLE:

```
LEG VARIABLE KEY=NR
  *1 NR NAT
  *1 N CONST
  *1 VREAL REAL
    *2 VMEAN PSEUDO
    *2 VS PSEUDO
    *2 VMAX PSEUDO
    *2 VMIN PSEUDO
    *2 VSIZE PSEUDO
  *1 VTEXT TEXT
    *2 TYP SCOPE=[R,A,B,C]
    *2 PRECIS NAT
    *2 VNAME PICT=25
    *2 FORMAT
  *1 VALUE ARRAY[N] REAL
END
```

Разбиение описания данных на две части обусловливало также изменение свойств некоторых атомов и групп. Так, например, атом M теперь уже не имеет свойства CONST, а число повторений атома REGISTER не указано. Это позволяет нам при написании соответствующих DML-программ проще добавить и удалить признаки.

Число индивидов N повторяется теперь в каждой записи признаков, чтобы можно было определить размер массива VALUE. В легенде STANDMX атом N имеет свойство CONST только из-за предосторожности, хотя не определяет количества повторений какой-либо группы.

Когда по легенде STMATRIX номера признаков хранились только в повторяющемся атоме REGISTER, то теперь номер каждого признака должен находиться среди других данных признака, а по новому описанию этот номер является также еще и ключом признака. Характеристики признака теперь соеденены в одну группу с именем VREAL только ради некоторой экономии при написании легенды.

Для полного описания данных необходимо представить еще описание файла стандартной матрицы на языке FDL (см. [8]). Соответствующая FDL-легенда, описывающая файл, состоящий из одной записи общих данных и любого числа записей для строк матрицы (признаков), имеет следующий вид:

```
INDEX FILE STATDATA
   *1 STANDMX
   *1 VARIABLE
EOF
```

2. Операции преобразования стандартной матрицы

Для преобразования стандартной матрицы в системе статистической обработки предназначены операции SORT, ADD, SELECT, NEWVAR и PROGNOSIS (см. [6], стр. 106). Последняя из этих операций заменяет отсутствующие значения признака их прогнозами (опираясь на корреляционную матрицу) и в настоящей статье рассматриваться не будет. Подробное описание первых четырех операций можно найти в [6]. Здесь мы лишь коротко напомним их суть и приведем соответствующие DML-программы.

Самой простой из названных операций является SELECT, которая из стандартной матрицы X образует матрицу Y, содержащую только те признаки, номера которых заданы в списке L1. Новые номера этих признаков в матрице Y должны быть заданы в другом списке L2. Если список L2 опущен, то считают, что L1 = L2 и перенумерация признаков не производится.

Соответствующая операции SELECT DML-программа должна, в принципе, произвести следующие действия:

```
FOR X(L1) NEW Y(L2)  Y:=X
```

Полная DML-программа будет, конечно, длиннее, так как в ней должны быть описаны используемые комплекты записей и произведены некоторые вспомогательные действия.

В файлах матриц X и Y имеются записи двух типов. Для записей общих данных (описуемых легендой STANDMX) будем использовать соответственно имена XS и YS, а для записей признаков (легенда VARIABLE) имена XV и YV. В качестве входной информации передаются списки L1 и L2, а также имя новой матрицы, которое будет записано в атом YS.NAME .

Образование матрицы Y удобно начинать с заполнения записи YS, используя для этого информацию из XS. При этом регистр номеров признаков REGISTER остается пустым, а число признаков M приравняется нулю. Далее по списку L1 будем поочередно читать записи XV и по списку L2 образовать соответствующие записи в YV. При каждом таком образовании M увеличивается на единицу и номер признака заносится в REGISTER.

Таким образом, программа SELECT, реализующая вышеназванные действия, принимает вид:

```
DML SELECT
   LEGEND STANDMX SET XS,YS;
   LEGEND VARIABLE SET XV,YV;
   INT  SWITCH  L1,L2;
   TEXT SWITCH  YNAME
   FOR XS
      NEW YS
         DO YS.NAME:=YNAME;
            YS.BASENAME:=XS.BASENAME;
            YS.DMY:=TODAY( );
            YS.N:=XS.N;
            YS.M:=Ø;
            FOR XV(L1)
               NEW YV(L2)
                  DO YV:=XV;
                     YS.M:=YS.M+1;
                     ADD YS.REGISTER (YV.NR)
                  OD
      OD
```

Если при совпадении списков мы хотим не указать второго списка в заказе программы, то концовку приведенной программы можно переписать в виде:

```
FOR XV(L1)
   DO IF L2= * THEN NEW YV(L1) YV:=XV
               ELSE NEW YV(L2) YV:=IV FI;
      YS.M:=YS.M+1;
      ADD YS.REGISTER (YV.NR)
   OD
```

Операция ADD используется в системе статистической обработки для объединения двух стандартных матриц: из матриц X1 и X2, имеющих одинаковое число индивидов, образуется новая матрица Y. Для выбора признаков, подлежащих включению в Y, должны быть заданы два списка номеров признаков L1 и L2, соответственно для X1 и X2. При этом признаки из X1 имеют приоритет в том смысле, что если какой-то номер встречается в обоих списках, то соответствующий признак выбирается из матрицы X1. Следовательно, для выполнения операции ADD надо сперва по списку L1 включить в Y все признаки из X1, а потом по списку L2 добавить к ним еще признаки из X2. На языке DML это означает проведение следующих действий:

```
DO FOR X1(L1) NEW Y(X1)  Y:=X1;
   FOR X2(L2) ADD Y(X2)  Y:=X2 OD
```

Напомним, что добавление (оператор ADD на языке DML) производится лишь при отсутствии записи или группы с указанным значением ключа. Это и гарантирует приоритет матрицы X1.

Полная DML-программа для соединения двух стандартных матриц X1 и X2 должна, конечно, содержать еще описания необходимых комплектов и переключателей, а также операторы для формирования записи общих данных. Используя аналогичные с программой SELECT имена комплектов и переключателей, получим следующую программу:

```
DML ADDX1X2
   LEGEND STANDMX  X1S,YS;
   LEGEND VARIABLE X1V,X2V,YV;
   INT  SWITCH  L1,L2;
   TEXT SWITCH  YNAME
   FOR X1S
      NEW YS
         DO YS.NAME:=YNAME;
            YS.BASENAME:=X1S.BASENAME;
            YS.DMY:=TODAY( );
            YS.N:=X1S.N;
            YS.M:=Ø;
            FOR X1V(L1) NEW YV(X1V)
               DO YV:=X1V;
                  YS.M:=YS.M+1;
                  ADD YS.REGISTER (YV.NR) OD;
            FOR X2V(L2) ADD YV(X2V)
               DO YV:=X2V;
                  YS.M:=YS.M+1;
                  ADD YS.REGISTER (YV.NR) OD
         OD
```

Операция **NEWVAR** предназначена в системе статистической обработки для образования новых признаков Y_l на базе исходных признаков $X_{l_1},\ldots,X_{l_k}$ по заданным преобразованиям

$$Y_L = F_l(X_{l_1},\ldots,X_{l_k})$$

(которые могут быть, например, арифметическими выражениями или правилами перекодирования).

При реализации этой операции возникают затруднения, так как нельзя уже написать одну общую DML-программу, пригодную для всех случаев. Для каждого нового признака нужно написать новую программу, в которой учтены число исходных признаков и проводимое преобразование. В качестве примера приводим общий вид таких программ (на некотором обобщении языка DML), из которого легко получаются конкретные DML-программы:

```
DML NEWVAR1
   LEGEND STANDMX  SET YS;
   LEGEND VARIABLE SET YV,X1V,...,XkV
   FOR X1V(L1),...,XkV(Lk)
      REPL YS
         ADD YV(L)
            DO YV.N:=X1V.N;
               YV.TYP:=R;
               FOR X1V.VALUE,...,XkV.VALUE
                 NEW YV.VALUE (X1.VALUE)
                    YV.VALUE:=F(X1V.VALUE,...,XkV.VALUE);
               YS.M:=YS.M+1;
               ADD YS.REGISTER (L)
            OD
```

Программа NEWVAR1 вычисляет признак Y_1 по преобразованию

$$Y_1 = F(X_{l_1}, \ldots, X_{l_k})$$

и добавляет его к матрице Y. Хотя все исходные признаки X_{l_1}, X_{l_2}, ..., X_{l_k} взяты из одной матрицы X, мы были вынуждены ввести для них k имен комплектов, так как при вычислении нового признака их значения могут понадобиться одновременно.

Для получения конкретной DML-программы следует установить k, заменить L, L1,..., Lk настоящими номерами признаков (или описать их как переключателей) и написать вместо функции F нужное преобразование. Например, для преобразования

$$Y_5 = \sqrt{X_2 + X_4},$$

где индексы означают номера признаков, получим DML-программу

```
DML NEWVAR2
   LEGEND STANDMX  SET YS;
   LEGEND VARIABLE SET YV,X1V,X2V
   FOR X1V(2),X2V(4)
      REPL YS
         ADD YV(5)
            DO YV.N:=X1.N;
               YV.TYP:=R;
               FOR X1V.VALUE,X2V.VALUE
                 NEW YV.VALUE (X1V.VALUE)
                   YV.VALUE:=SQRT(X1V.VALUE+X2V.VALUE);
               YS.M:=YS.M+1;
               ADD YS.REGISTER (5)
            OD
```

Причинами тому, почему нельзя обойтись одной общей программой, являются два обстоятельства. Во первых, число имен комплектов зафиксировано в каждой DML-программе, но число исходных признаков не определено. Во вторых, преобразование для получения нового признака должно быть написано в теле программы и у нас нет средств, чтобы задать его в заказе к выполнению программы. Эти трудности можно, однако, в значительной степени преодолеть путем использования макросредств. Уже простейшие макрогенераторы могут заметно упростить написание программ одноразового пользования. Покажем это на примере, используя макросистему, описанную в [7] (стр. 93).

Для вычисления нового признака с применением максимально трех исходных признаков предлагаем программу NEWVAR3, содержащую пять макровызовов:

```
DML NEWVAR3
   LEGEND STANDMX  SET YS;
   LEGEND VARIABLE SET YV,X1V,X2V,X3V
   FOR X1V(@1)),X2V(@2)),X3V(@3))
      REPL YS
         ADD YV(@4))
            DO YV.N:=X1.N;
               YV.TYP:=R;
               FOR X1V.VALUE,X2V.VALUE,X3V.VALUE
                  NEW YV.VALUE (X1V.VALUE)
                     YV.VALUE:=@5);
               YS.M:=YS.M+1;
               ADD YS.REGISTER (@4))
            OD
```

Для использования этой программы ее каждый раз надо заново протранслировать: пустить через макрогенератор макроопределения макроимен @1, @2, @3, @4, @5 и за ними текст программы NEWVAR3. Например, для вычисления признака

$$Y_7 = X_1 + X_3 - X_4$$

макроопределения следует задать следующими:

```
@ @1) 1@ @
@ @2) 3@ @
@ @3) 4@ @
@ @4) 7@ @
@ @5) X1V.VALUE+X2V.VALUE-X3V.VALUE@@
```

Эту-же программу можно, однако, применить и для меньше чем трех исходных признаков. Вместо номера отсутствующего признака тогда в макроопределении можно повторить номер первого исходного признака. Например, эквивалентную с программой NEWVAR2 программу получим макроопределениями

```
@ @1) 2@ @
@ @2) 4@ @
@ @3) 2@ @
@ @4) 5@ @
@ @5) SQRT(X1V.VALUE+X2V.VALUE)@@
```

Хотя эта программа и несколько избыточна (в ней встречается имя ненужного комплекта X3), для ее получения пришлось меньше писать. Применение более мощных макрогенераторов позволяет сократить такую избыточность (ведь использованный в примерах макрогенератор не имеет даже понятия параметра).

Заметим, что образованные во всех предыдущих примерах новые признаки имеют только значения и тип. Вершины PRECIS, VNAME и FORMAT должны получить свои значения отдельно. Для вычисления группы VREAL можно использовать следующую DML-программу CALCUL, которая вычисляет статистики для всех тех признаков матрицы X, у которых этого раньше не сделано (для этого проверяется, присвоено ли значение вершине VSIZE):

```
DML CALCUL
   LEGEND VARIABLE SET X
   WITH X
   REPL X
      DO IF VSIZE != * THEN BACK X FI;
         VMEAN,VS,VSIZE:=0;
         VMAX:=MAXIMUM( );
         VMIN:=MINIMUM( );
         FOR VALUE
            IF VALUE != * THEN
               DO VSIZE:=VSIZE+1;
                  VMEAN:=VMEAN + VALUE;
                  IF VALUE < VMIN THEN VMIN:=VALUE FI;
                  IF VALUE >VMAX THEN VMAX:=VALUE FI;
               OD FI;
         VMEAN:=VMEAN / VSIZE;
         FOR VALUE IF VALUE != * THEN
            VS:=(VALUE-VSIZE)#(VALUE-VSIZE) + VS FI;
         VS:=VS / (VSIZE-1);
         VS:=SQRT(VS)
      OD
```

В системе статистической обработки данных операция SORT применяется также, как и SELECT для уменьшения размера матрицы. Но кроме списка L номеров признаков матрицы X, принадлежащих переводу в матрицу Y дается еще условие $C(X_{i_1},\ldots,X_{i_n})$ для выбора индивидов. При реализации здесь возникают аналогичные проблемы как и при операции NEWVAR. Так как решение этих проблем тоже аналогичное, мы не будем приводить примеров соответствующих программ.

В итоге можно констатировать, что если задачи преобразования данных имеют несложные параметры типа списка констант, то соответствующие DML-программы легко пишутся. Если же исходная информация имеет более сложную структуру, содержа, например, выражения или условия, то для новой задачи придется каждый раз писать новую программу. Такую работу можно, однако, существенно упростить применением макрогенераторов.

Литература

1. Изотамм А., Каазик Ю., Томбак М., Язык определения записи. Труды ВЦ ТГУ, 1978, № 41, 7-64.
2. Каазик Ю., Рауп А., Язык манипулирования данными. Труды ВЦ ТГУ, 1978, № 41, 97-140.
3. Рауп А., Реализация языка манипулирования данными. Наст. сборник.
4. Тоодинг Л.М., Система статистической обработки данных в вычислительном центре ТГУ. Труды ВЦ ТГУ, 1977, № 40, 3-7.
5. Тоодинг Л.М., Структура данных. Труды ВЦ ТГУ, 1977, №40, 75-85.
6. Тоодинг Л.М., Преобразование стандартной матрицы. Труды ВЦ ТГУ, 1977, № 40, 106-114.
7. Каазик Ю., Ээльма П., Ввод и корректировка данных. Труды ВЦ ТГУ, 1978, № 41, 75-96.
8. Каазик Ю., Образование файлов. Труды ВЦ ТГУ, 1978, № 41, 65-74.

ПРИНЦИПЫ ОБРАБОТКИ ГРАФОВ НА ЕС ЭВМ

Ю. Кихо

Граф с отмеченными вершинами и дугами, т.н. ограф является распространенной формой представления информационных структур. В работе [1] даны некоторые способы отображения графа в памяти ЭВМ "Минск-32". В связи с переходом на ЕС ЭВМ возникает необходимость заново фиксировать машинные форматы графа, чтобы обеспечить совместимость соответствующих программ обработки, а также предотвратить дублирование при программировании общих операций над графами.

В настоящем сообщении в качестве базисного представления графа предлагается формат, аналогичный связному коду ографа в [1]. Как показывает опыт, именно этот способ является наиболее гибким и универсальным в случае широкого класса графовых задач.

Рассматриваемое представление графа основывается на понятии связного списка с заголовком. В таком списке каждый элемент обязательно включает поле связи (одно машинное слово), которое содержит ссылку на следующий элемент списка;

поле связи последнего элемента содержит пустую ссылку. Ссылками служат действительные адреса, а пустая ссылка представляется в виде специальной конфигурации $FF000000_{16}$ (значение NULL указателя в ПЛ/I). Все элементы списка, в том числе и заголовок, имеют одинаковую структуру, т.е. принадлежат к одному и тому же типу структур. В данном случае, например, будут рассматриваться списки, состоящие из т.н. атомов и списки, состоящие из списков. Связной список α, который состоит из элементов β_0 (заголовок), β_1, ..., β_m будем обозначать через α: $\underline{\beta_0}\ \beta_1 \ \ldots\ \beta_m$. Особенность заголовка заключается в том, что его поля, кроме поля связи, имеют назначение, отличное от назначения соответствующих полей у остальных элементов списка.

Переходим теперь к рассмотрению форматов графов, представленных в виде связных списков. Единицей памяти или атомом при отображении графа служит 16-байтовое поле, выровненное на границе слова. Такие атомы будем обозначать через букву "a" с нижними индексами, причем добавлением верхнего индекса (1, 2, 3 или 4) будут указываться соответствующие слова, входящие в атом (например, a_i^3 – третье слово в атоме a_i).

Базисный формат n-вершинного графа*) есть связной список

$$G : \underline{T_0}\ T_1 \ \ldots\ T_n ,$$

где $T_0 : \underline{a_{00}}$, а для i=1,2,...,n

$$T_i : \underline{a_{i0}}\ a_{i1} \ \ldots\ a_{ik_i},$$

---

*) Здесь, как и в [1], предполагается, что вершины графа пронумерованы, а также пронумерованы дуги, исходящие из одной и той же вершины.

где k_i – количество дуг, исходящих из i-ой вершины графа. Полем связи в G служит a^3_{10} (i=0,1,...,n). Поля остальных атомов подсписков имеют при каждом i=1,2,...,n следующие назначения:

a^4_{ij} (j=0,1,...,k_i) – поле связи в T_i,

a^3_{ij} (j=1,2,...,k_i) – ссылка на (заголовок) T_l,

где l – номер вершины, в которую входит j-ая дуга, исходящая из i-ой вершины.

Таким образом, в базисном формате графа свободными (незанятыми) полями являются первые два слова всех атомов, а также a^4_{00}. Эти поля предназначены для записи дополнительной информации, связанной с графом, его вершинами и дугами. К такой информации относятся, в частности, метки вершин и дуг графа. Но, поскольку в виде графа можно изображать самые различные информационные структуры, придав соответствующий смысл его меткам, то нецелесообразным представляется полностью унифицировать формат графа. Тем не менее для стандартизации программ обработки графов необходимо соблюдать общие принципы построения конкретных форматов и составления соответствующих программ обработки.

В качестве таких принципов предлагаются следующие стандартные соглашения.

1. Основой каждого конкретного формата графа служит базисный формат графа. Конкретный формат получается лишь уточнением назначения свободных полей базисного формата.

2. Информация, связанная с i-ой вершиной, указывается в атоме a_{10}, а информация о j-ой дуге, исходящей из i-ой вершины, указывается в атоме a_{1j}. Свободные поля заголовка (a_{00})

главного списка используются для изображения дополнительной информации об графе. Таким образом, атомы a_{i0} ($i \neq 0$) соответствуют вершинам изображаемого графа, а атомы a_{ij} ($i \neq 0$, $j \neq 0$) – его дугам.

3. Из первых двух полей атома первое слово предусматривается для указания текущей информации, второе – для (более) постоянной информации. Это значит, что первое слово является рабочем полем в программах обработки графа, содержимое которого не определено ни при входе в программу, ни при выходе из нее. Второе слово атома содержит либо метку либо ссылку на метку соответствующей вершины или дуги. В противоположность первому слову, содержимое второго слова определено как при входе так и при выходе из программ обработки графа.

4. Способ записи (формат) сопровождающей информации, указанной двумя первыми словами атомов, должен быть одинаковым для всех вершин, а также для всех дуг. (На самом деле, это требование вытекает уже из требования единства назначений полей элементов связного списка.) Например, можно применить формат, в котором метки вершин указываются при помощи ссылок во втором слове соответствующих атомов, а метки дуг записаны непосредственно во втором слове соответствующих атомов. Но не допускается, например, формат, в котором метки некоторых вершин записаны непосредственно во втором слове атомов, а метки остальных вершин даны посредством ссылок.

5. Для большего однообразия структур п р о г р а м м обработки графовой информации рекомендуется во всех циклах прохода (под)списков продвижение текущего указателя предусмотреть в конце тела цикла. Следовательно, перед первым

```
DCL 1 ATOM BASED(P),
      2 I BIN FIXED(31),     /* РАБОЧЕЕ ПОЛЕ */
      2 M PTR,               /* ССЫЛКА НА МЕТКУ */
      2 (VER,DUGA) PTR;      /* ПОЛЯ СВЯЗИ */
DCL METKA CHAR(8) BASED(R); /* ПОЛЕ МЕТКИ */
DCL (P,Q,R,S) PTR;          /* ТЕКУЩИЕ УКАЗАТЕЛИ */

    ...
                  /* P:= АДРЕС ЗАГОЛОВКА ОГРАФА */
    ...
         P=VER;   /* P НА ПЕРВУЮ ВЕРШИНУ */
/* ЦИКЛ1: ПРОСМОТР ВСЕХ ВЕРШИН */
         DO WHILE (P¬=NULL);
                  /* P - АДРЕС РАССМАТРИВАЕМОЙ ВЕРШИНЫ */
                  /* I - РАБОЧЕЕ ПОЛЕ У ЭТОЙ ВЕРШИНЫ */

            ...
           R=M;  /* МЕТКА ВЕРШИНЫ(P) ДОСТУПНА */
            ...

           Q=DUGA;  /* Q НА ПЕРВУЮ ДУГУ У ВЕРШИНЫ(P) */
  /* ЦИКЛ2: ПРОСМОТР ВСЕХ ДУГ У ВЕРШИНЫ(P) */
           DO WHILE (Q¬=NULL);
                    /* Q - АДРЕС РАССМАТРИВАЕМОЙ ДУГИ */
                    /* Q->I - РАБОЧЕЕ ПОЛЕ У ЭТОЙ ДУГИ*/

              ...
             S=Q->M;  /* S->МЕТКА - МЕТКА ДУГИ(Q) */
              ...

             Q=DUGA;  /* ПЕРЕХОД К СЛЕДУЮЩЕЙ ДУГЕ */
  /* ЦИКЛ2*/END;

            ...
           P=VER;/* ПЕРЕХОД К СЛЕДУЮЩЕЙ ВЕРШИНЕ */
/* ЦИКЛ1*/END;
```

входом в цикл, текущий указатель должен быть установлен на первом просматриваемом элементе списка. На стр. 87 приведены фрагменты программы с такой организацией циклов.

6. Стандартными программами ввода графа с перфокарт, вывода его на печать в виде таблицы связей или же плоского рисунка (и некоторыми другими программами) применяется формат графа, в котором все метки предполагаются заданными в 8-байтовых полях в символьной форме; в атомах даются указатели на эти поля. (Такой формат используется и в приведенном на стр. 87 примере.) Поэтому, если это позволит характер рассматриваемой информации, желательно применить такой же формат меток, в противном случае необходимо составить соответствующие программы преобразования форматов с целью установления связи с стандартными программами.

В заключении отметим, что вопрос о соблюдении стандартных соглашений об изображении и обработке графов на ЭВМ, это, в первую очередь, вопрос о повышении производительности труда программиста. Из-за разнообразия решаемых на ЭВМ задач и соответствующих объектов, почти для каждой конкретной задачи можно было бы построить специальную схему изображения обрабатываемой информации, обеспечивающую получение в некотором смысле оптимальных программ. Но, как правило, затраты на разработку и отладку новых специальных приемов отображения и обработки графовой информации не окупаются небольшим повышением эффективности получаемых программ. Единым подходом к программированию графовых задач исключается создание такого слишком специального и дорогостоящего программного обеспечения.

Изложенные выше принципы отображения и обработки графовых структур заложены в основу создаваемого в настоящее время соответствующего пакета программ для ЕС ЭВМ. К таким программам относятся, например, программы анализа циклической структуры графа и порождения канонической нумерации вершин графа, программы ввода-вывода графовой информации и др.

Литература

1. Кихо Ю., Машинные форматы графов. Труды ВЦ ТГУ, 1975, № 33, 40-52.

ПОИСК ПО ДРЕВОВИДНЫМ КЛЮЧАМ

Я. Каазик

Задача поиска записей является одной из центральных в программировании. Самым распространенным методом организации такого поиска является метод ключей, при котором с каждой записью связывают уникальный ключ, так что файл представляет собой множество пар $\{(D_i, \gamma_i)\}$, где D_i – ключ, а γ_i – соответствующая запись.

Обычно ключи выбирают из некоторого специального множества векторов. Однако, при таком способе невсегда удобно выбирать для всех записей файла уникальные ключи. В настоящей статье проблема поиска записей рассматривается для случая древовидных ключей.

Пусть некоторый банк данных содержит множество записей, ключами D_i которых являются отмеченные корневые деревья. При этом уникальность ключа означает, что в множестве ключей $\{D_i\}$ нет изоморфных деревьев. Задача поиска записи по заданному ключу (дереву) D сводится к выяснению, существует ли такой i, при котором $D = D_i$, где равенство понимается в смысле изоморфизма деревьев.

Поиск путем непосредственного сравнения всех деревьев D_i с ключом D нас не устраивает, так как установление эквивалентности слишком трудоемко и не позволяет эффективно использовать быстрые внешние запоминающие устройства (диски). Как известно, для быстрого чтения записи с диска нужен порядковый номер записи. В рассматриваемой задаче это означает, что нам нужна функция Φ, которая преобразует заданное дерево D в целое число следующим образом:

$$\Phi(D) = \begin{cases} i, & \text{если } D = D_i, \\ 0, & \text{если } D \notin \{D_i\}. \end{cases}$$

При наличии такой функции с диска следует по ключу D читать запись $\mathfrak{z}_{\Phi(D)}$. В данной статье предлагается один возможный метод построения функции Φ, используя для этого подходящим образом составленное И/ИЛИ-дерево.

Будем предполагать, что И/ИЛИ-дерево T составлено таким образом, что множество (T) всех входящих в T максимальных И-деревьев совпадает с $\{D_i\}$. Такое использование И/ИЛИ-дерева в качестве описателя множества деревьев, как правило, уменьшает объем памяти, необходимой для хранения $\{D_i\}$, а также число сравнений вершин при установлении изоморфизмов. Экономия тем значительнее, чем больше совпадающих частей имеют деревья D_i и чем ближе эти совпадающие части находятся к корням деревьев.

Существование указанного И/ИЛИ-дерева T для любого заданного множества $\{D_i\}$ не требует доказательства. Вопросы же, связанные с построением наилучшего такого дерева, здесь не рассматриваются.

Определения и обозначения

Отмеченное корневое дерево $T = (V,E)$ является И/ИЛИ-деревом, если задано разбиение множества его вершин $V = A \cup O$ так, что при любых $s,t \in O$

1) $(s,t) \notin E$,

2) $\varrho(t) > 1$,

где значением функции ϱ является число непосредственных подчиненных аргумента.

Вершины из множества A называются И-вершинами, а вершины из множества O - ИЛИ-вершинами.

Во всех последующих примерах для обозначения конкретных И-вершин будут использованы строчные латинские буквы, а для ИЛИ-вершин - цифры. Все примеры опираются на дерево T, изображенное на рисунке 1.

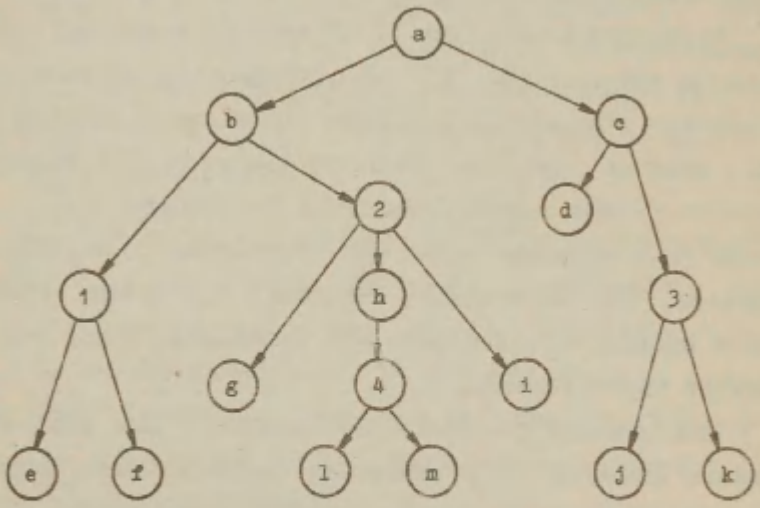


Рис. 1.

Для обозначения любой вершины И/ИЛИ-дерева в дальнейшем будем использовать буквы v, w, но в случае ИЛИ-вершины буквы s, t. Множество всех непосредственных подчиненных вершины $v \in V$ обозначаем через $[v]$, т.е.

$$[v] = \{w : (v,w) \in E\}.$$

Таким образом, $|[v]| = \varrho(v)$. Поддерево (в обычном смысле) с корнем v обозначаем как $T_v = (V_v, E_v)$.

Каждую ИЛИ-вершину t интерпретируем как некий (безусловный) селектор, выбирающий точно одну вершину $v \in [t]$. Операцию такого выбора назовем операцией выбора вершины и отмечаем через $v * T$. Результатом этой операции является дерево (V^v, E^v), в котором

$$V^v = (V \setminus V_t) \cup V_v,$$
$$E^v = ((V^v \times V^v) \cap E) \cup \{(w,v)\}, \quad \text{где} \quad t \in [w].$$

Например, два результата выбора конкретной вершины в дереве рисунка 1 показаны на рисунке 2.

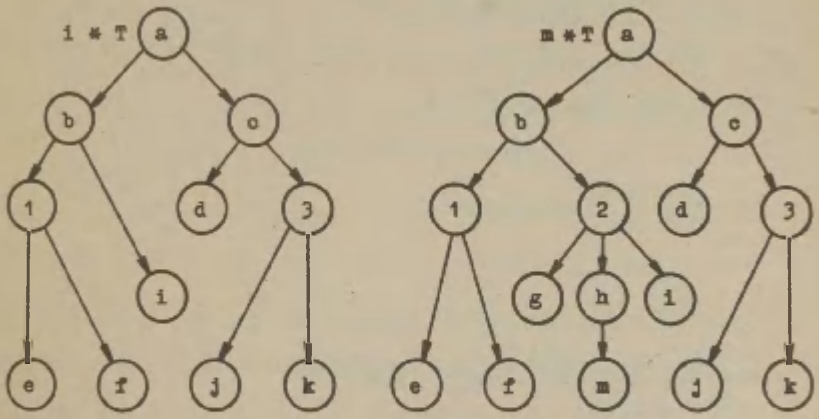


Рис. 2.

Дерево $v * T$ является, вообще говоря, опять И/ИЛИ-деревом. Следовательно, операцию выбора вершины можно в нем применить для любой другой ИЛИ-вершины s. Например, если $w \in \in [s]$, то получаем дерево

$$w * v * T .$$

По определению операции $*$ такая запись корректна лишь в случае $w \in V^v$, т.е. когда выбор вершины v не исключает возможности выбора вершины w. С другой стороны, при $v \in V^w$ также существует дерево

$$v * w * T .$$

Если условия $v \in V^w$ и $w \in V^v$ выполняются одновременно (выборы вершин v и w независимы), то, очевидно, имеет место коммутативность операции выбора вершины:

$$v * w * T = w * v * T .$$

Для того, чтобы этим равенством можно было пользоваться независимо от расположения вершин s и t в дереве, мы должны расширить определение операции выбора вершины на тот случай, когда выбираемая вершина отсутствует в дереве. Поэтому будем считать, что если $w \notin V^v$, то

$$w * v * T = v * T,$$

а при $v \notin V^w$ аналогично

$$v * w * T = w * T.$$

Так как из-за $s \neq t$ одновременно не может быть $v \notin V^w$ и $w \notin V^v$, то коммутативность рассматриваемой операции очевидна и в общем случае.

Проведение операции выбора вершины при фиксированной $t \in \sigma$ для всех $v \in [t]$ дает нам некоторое множество И/ИЛИ-деревьев. Операцию образования такого множества назовем операцией удаления вершины t из И/ИЛИ-дерева T и обозначим

$$[t] * T = \{ v * T : v \in [t] \}.$$

Так как элементами множества $[t] * T$ являются И/ИЛИ-деревья, то на каждое из них можно опять применить операцию выбора вершины, т.е. если $s \in \sigma$, $s \neq t$ и $w \in [s]$, то определено множество деревьев

$$w * [t] * T = \{ w * v * T : v \in [t] \}.$$

Теперь непосредственно получается обобщение рассматриваемой операции - операция удаления нескольких ИЛИ-вершин из дерева. Например, в случае двух таких вершин имеем

$$[s] * [t] * T = \{ w * v * T : v \in [t], w \in [s] \}.$$

При этом, конечно, имеет место равенство

$$[s] * [t] * T = [t] * [s] * T.$$

Если применить операцию удаления ИЛИ-вершины из дерева для всех $t \in \sigma$, то получим множество всех входящих в T максимальных И-деревьев. Таким образом, если

$$\sigma = \{ t_1, t_2, \ldots, t_m \},$$

то имеет место равенство

$$(T) = [t_1] * [t_2] * \ldots * [t_m] * T.$$

При этом результат не зависит от порядка удаления ИЛИ-вершины t_i. Например, для дерева T с рисунка 1 множество (T) состоит из 16-и деревьев, изображенных на рисунках 3 и 4.

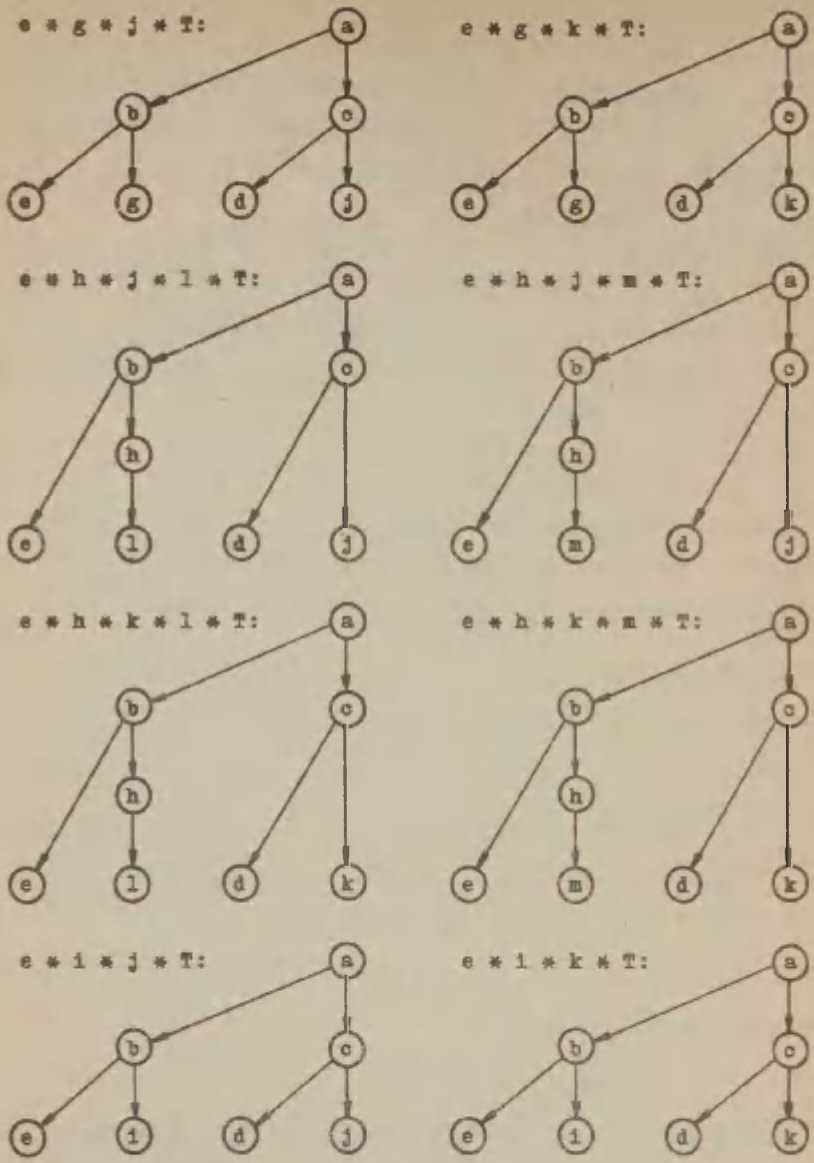


Рис. 3.

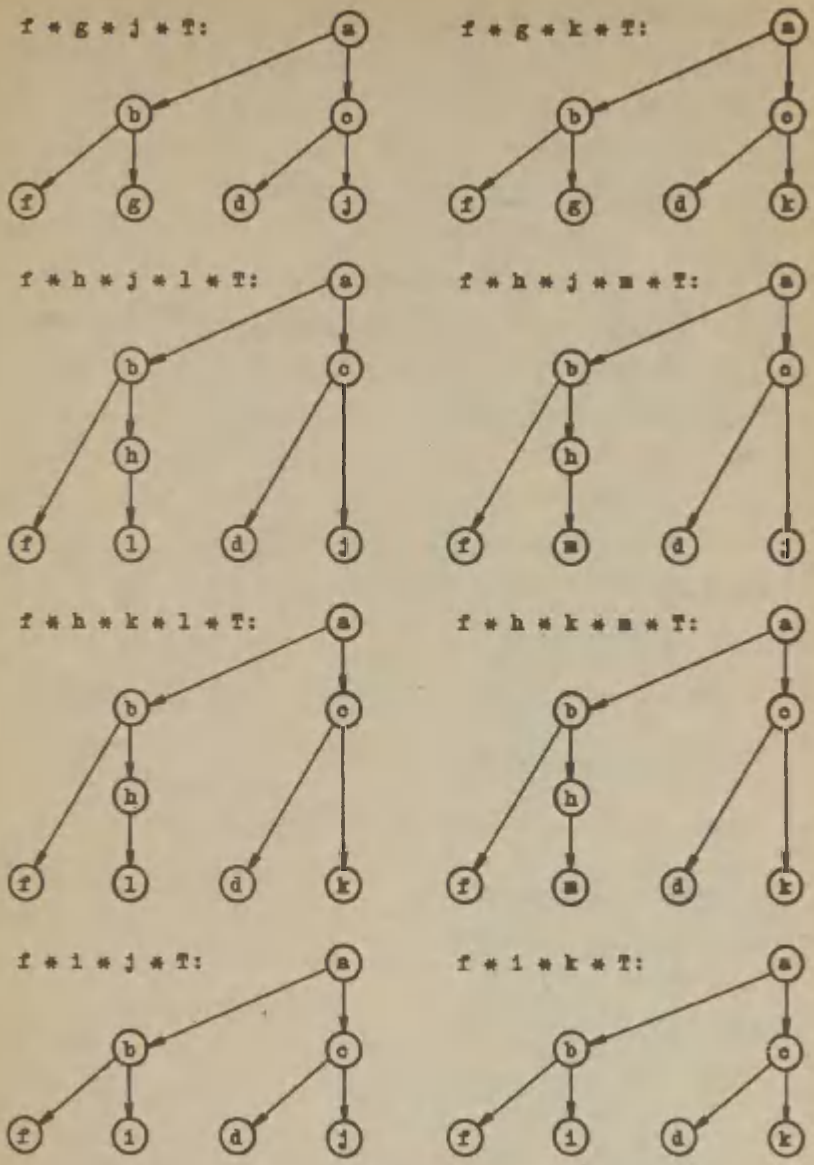


Рис. 4.

Возникает еще естественный вопрос об определении мощности множества (T). В общем случае, когда (T) может содержать изоморфные деревья, это весьма трудная задача. Нетрудно, однако, построить функцию $N(T)$, значение которой в случае отсутствия изоморфных деревьев в (T) равняется мощности множества (T), а в общем случае дает верхнюю оценку для $|(T)|$.

Функцию $N(T)$ можно построить путем последовательного определения значений $N(T_v)$ для всех поддеревьев T_v так, чтобы в случае отсутствия изоморфных деревьев в (T_v) было $N(T_v) = |(T_v)|$. Тогда $N(T) = N(T_r)$, где r - корень дерева T.

В случае висящей вершины $v \in A$ естественно принимать $N(T_v) = 1$. Двигаясь вверх по дереву для любой вершины v определим теперь значение $N(T_v)$ искомой функции через ее значения в непосредственно подчиненных вершинах $w \in [v]$:

1) если $v \in \mathcal{O}$, то

$$N(T_v) = \sum_{w \in [v]} N(T_w);$$

2) если же $v \in A$, то

$$N(T_v) = \prod_{w \in [v]} N(T_w).$$

Например, в случае дерева T рисунка 1 (в этом дереве все метки различны и изоморфных деревьев в (T), следовательно, быть не может) получим:

$N(T_e)=N(T_f)=N(T_g)=N(T_i)=N(T_m)=N(T_i)=N(T_j)=N(T_k)=N(T_d)=1$,

$N(T_1)=N(T_e)+N(T_f)=2$, $N(T_4)=N(T_i)+N(T_m)=2$,

$N(T_3)=N(T_j)+N(T_k)=2$, $N(T_c)=N(T_d)\cdot N(T_3)=2$,

$N(T_h)=N(T_4)=2$, $N(T_2)=N(T_g)+N(T_h)+N(T_i)=4$,

$N(T_b)=N(T_1)\cdot N(T_2)=8$, $N(T)=N(T_a)=N(T_b)\cdot N(T_c)=16$.

Построение функции Ф

Далее предполагаем, что И/ИЛИ-дерево T построено по заданному множеству $\{D_1\}$ таким образом, что (T) = $\{D_1\}$, причем (T) не содержит изоморфных И-деревьев, т.е. требуем чтобы было |(T)| = N(T).

Перед тем, как перейти к непосредственному построению функции Ф для получения порядкового номера записи Ф(D) по заданному ключу-дереву D, исследуем, разбивает ли заданная ИЛИ-вершина t множество (T) на классы в том смысле, чтобы при любых $v_1, v_2 \in [t]$, $v_1 \neq v_2$ было

$$(v_1 * T) \cap (v_2 * T) = \emptyset.$$

Допустим, что найдется И-дерево D^* такое, что

$$D^* \in (v_1 * T) \cap (v_2 * T).$$

По крайней мере некоторые деревья в множестве $(v_1 * T)$ содержат вершину v_1, которая отсутствует во всех деревьях множества $(v_2 * T)$. С другой стороны, вершина v_2 отсутствует во всех деревьях множества $(v_1 * T)$. Таким образом, в дереве D^* не может быть вершин v_1 и v_2, а также других вершин из [t]. Такая ситуация, что в дереве D^* нет ни одной вершины из множества [t] возможна лишь тогда, когда среди предков вершины t имеется ИЛИ-вершина, т.е. между вершиной t и корнем r существует некая ИЛИ-вершина t^* или же $r \in \mathcal{O}$.

В целях иллюстрации этой ситуации на рисунке 5 приведено множество (i * T) для дерева рисунка 1. Во всех деревьях этого множества отсутствуют все вершины из [4]. С другой стороны, рисунок 3 представляет собой множество (e * T), в котором для любого $t \in \mathcal{O}$ имеются элементы из [t].

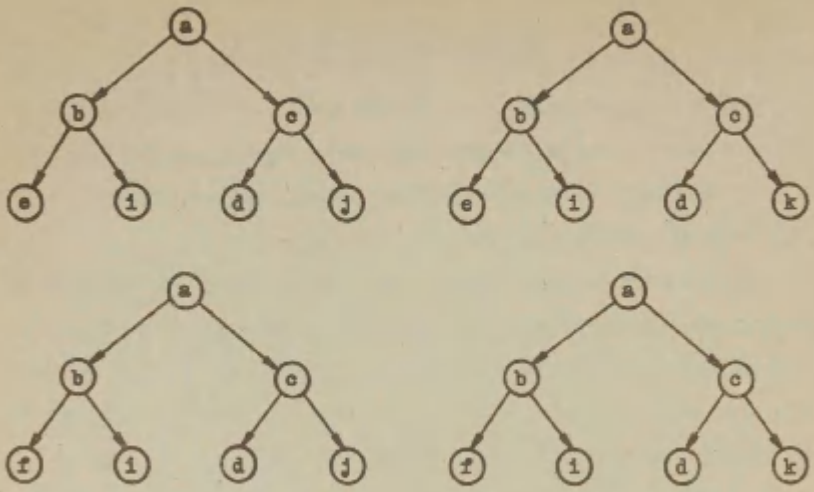


Рис. 5.

Если между корнем r и вершиной t нет ИЛИ-вершин, то все деревья из $(v_1 * T)$ содержат вершину v_1 (а деревья из $(v_2 * T)$ соответственно вершину v_2) и поэтому

$$(v_1 * T) \cap (v_2 * T) = \emptyset.$$

Пусть теперь задано дерево D, о котором известно, что $D \in (T)$. Если мы выбираем вершину $t_1 \in \mathcal{O}$ так, что среди ее предков нет ни одной ИЛИ-вершины, то из сказанного следует, что существует точно одна вершина $v^1_{i_1} \in [t_1]$ такая, что $D \in (v^1_{i_1} * T)$.

Предположим далее, что элементы множества $[t_1] = \{v^1_1, v^1_2, \dots, v^1_{\varrho(t_1)}\}$ каким-то образом упорядочены. Тогда легко построить для $\Phi(D)$ первую оценку

$$\sum_{j=1}^{i_1-1} N(v^1_j * T) < \Phi(D) \leq \sum_{j=1}^{i_1} N(v^1_j * T),$$

где в случае $i_1 = 1$ нижняя оценка равна нулю.

Если $\mathcal{O} = \{t_1\}$, т.е. больше ИЛИ-вершин в дереве T нет, то естественно принимать $\Phi(D) = i_1$. Если же t_1 не единственная ИЛИ-вершина, то далее в И/ИЛИ-дереве $v^1_{i_1} * T$ выбираем вершину $t_2 \in \mathcal{O}$ вышеуказанным образом, т.е. так, чтобы среди ее предков не было ни одной ИЛИ-вершины. Предполагая упорядоченность множества $[t_2]$, найдем единственную вершину $v^2_{i_2} \in [t_2]$ так, что

$$D \in (v^2_{i_2} * v^1_{i_1} * T).$$

Нижняя граница для $\Phi(D)$ принимает теперь вид

$$\sum_{j=1}^{i_1-1} N(v^1_j * T) + \sum_{j=1}^{i_2-1} N(v^2_j * v^1_{i_1} * T) < \Phi(D).$$

Продолжая этот процесс до тех пор, пока

$$v^k_{i_k} * v^{k-1}_{i_{k-1}} * \ldots * v^1_{i_1} * T = D$$

можем интересующее нас число принять равным выражению

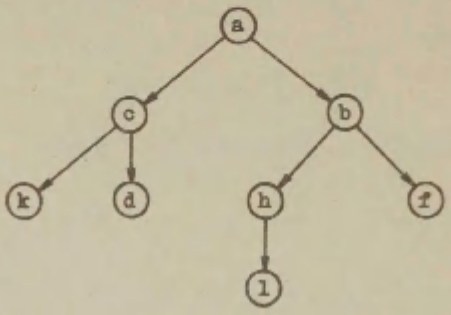
$$\Phi(D) = \sum_{j=1}^{i_1-1} N(v^1_j * T) + \sum_{j=1}^{i_2-1} N(v^2_j * v^1_{i_1} * T) + \ldots$$
$$\ldots + \sum_{j=1}^{i_{k-1}-1} N(v^{k-1}_j * v^{k-2}_{i_{k-2}} * \ldots * v^1_{i_1} * T) + i_k.$$

Таким образом, чтобы И/ИЛИ-дерево T (не содержащее изоморфных И-деревьев) было пригодным для вычисления функции $\Phi(D)$, мы должны:

1) упорядочить множество ИЛИ-вершин $\mathcal{O}$ по уровням сверху вниз,
2) при каждом $t \in \mathcal{O}$ произвольным образом упорядочить множество $[t]$.

Если в дереве рисунка 1 вершины из $\mathfrak{G}$ упорядочить по меткам, а их подчиненные по расположению на рисунке слева направо, то мы получаем упорядочение множества всех И-деревьев, как показано на рисунках 3 и 4.

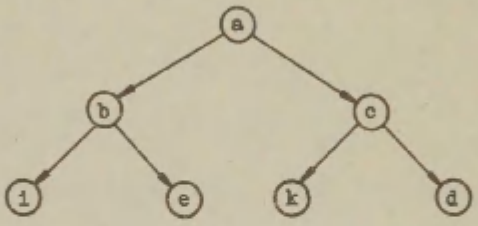
Для иллюстрации вычисления значений функции Φ рассмотрим еще два примера. Пусть ключ D задан, например, в виде дерева:



Так как D изоморфно с деревом $f * h * k * l * T$, то

$$\Phi(D) = N(e * T) + N(g * f * T) + N(j * h * f * T) + 1 = 8 + 2 + 2 + 1 = 13.$$

Если же ключ D задан в виде дерева:



то $D = e * i * k * T$ и следовательно

$$\Phi(D) = 0 + N(g * e * T) + N(h * e * T) + 2 = 2 + 4 + 2 = 8 .$$

Содержание

ВОПРОСЫ ОБРАБОТКИ ДАННЫХ СО СЛОЖНОЙ СТРУКТУРОЙ.
Труды вычислительного центра.
Выпуск 45.
На русском языке.
Тартуский государственный университет.
ЭССР, 202 400, г.Тарту, ул.Юликооли, I8.
Ответственный редактор Ю.Таммер.
Подписано к печати I4.XI.I980.
МВ 09709.
Формат 30x42/4.
Бумага писчая.
Машинопись. Ротапринт.
Условно-печатных листов 6,04.
Учетно-издат. листов 4,I9. Печатных 6,5.
Тираж 300.
Заказ № I20I.
Цена 65 коп.
Типография ТГУ, ЭССР, 202400, Тарту, ул. Пялсона, I4.

65 коп.