ТАРТУСКИЙ
ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ

ТРУДЫ

ВЫЧИСЛИТЕЛЬНОГО ЦЕНТРА

52

ТАРТУ
1985

ТАРТУСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ

ПОСТРОЕНИЕ СИСТЕМ ОБРАБОТКИ ДАННЫХ

ТРУДЫ ВЫЧИСЛИТЕЛЬНОГО
ЦЕНТРА

Выпуск 52

ТАРТУ 1985

Утверждено на заседании совета математического факультета ТГУ 25 октября 1985 года

ПОСТРОЕНИЕ СИСТЕМ ОБРАБОТКИ ДАННЫХ.
Труды вычислительного центра. Выпуск 52.
На русском языке.
Тартуский государственный университет.
ЭССР, 202400, г.Тарту, ул.Юликооли, 18.
Ответственный редактор Ю. Тапфер.
Подписано к печати 10.12.1985.
МВ 10551.
Формат 60x84/16.
Бумага писчая.
Машинопись. Ротапринт.
Условно-печатных листов 6,05.
Учетно-издательских листов 4,5. Печатных листов 6,5.
Тираж 200.
Заказ № 1245.
Цена 70 коп.
Типография ТГУ, ЭССР, 202400, г.Тарту, ул.Пялсона, 14.

МОНИТОР СИСТЕМЫ УПРАВЛЕНИЯ БАЗОЙ ДАННЫХ

Ю. Каазик, К. Куллман, К. Ээремаа

Общие принципы работы монитора системы РАМА (см. [1]), управляющего одновременным выполнением программ всех пользователей, описаны в [4]. Настоящая статья, как продолжение этой статьи, рассматривает функции монитора более подробно.

Реализованный в настоящее время первый вариант монитора (МОНРАМА) работает по сеансам, где под сеансом понимаем выполнение одного пакета заказов пользователей. В такой пакет разрешается объединять заказы максимально 15-ти пользователей, причем каждый пользователь может иметь сколько угодно отдельных программ - шагов. Программы разных пользователей выполняются параллельно, шаги каждого из них - последовательно.

Управляемый системой РАМА банк данных организован в виде т.н. фондов, где каждый фонд содержит выбранные по некоторому принципу файлы данных (напр.: файлы определенной группы пользователей, связанные по содержанию данные и т.п.). В один и тот же фонд с файлом данных обязательно должны входить как описание этого файла (см. [8]), так и необходимые легенды (см. [2]). Предусмотрена возможность сохранить в пределах фонда и готовые программы обработки данных. Сведения о находящихся в фонде файлах и их состояниях записываются в специ-

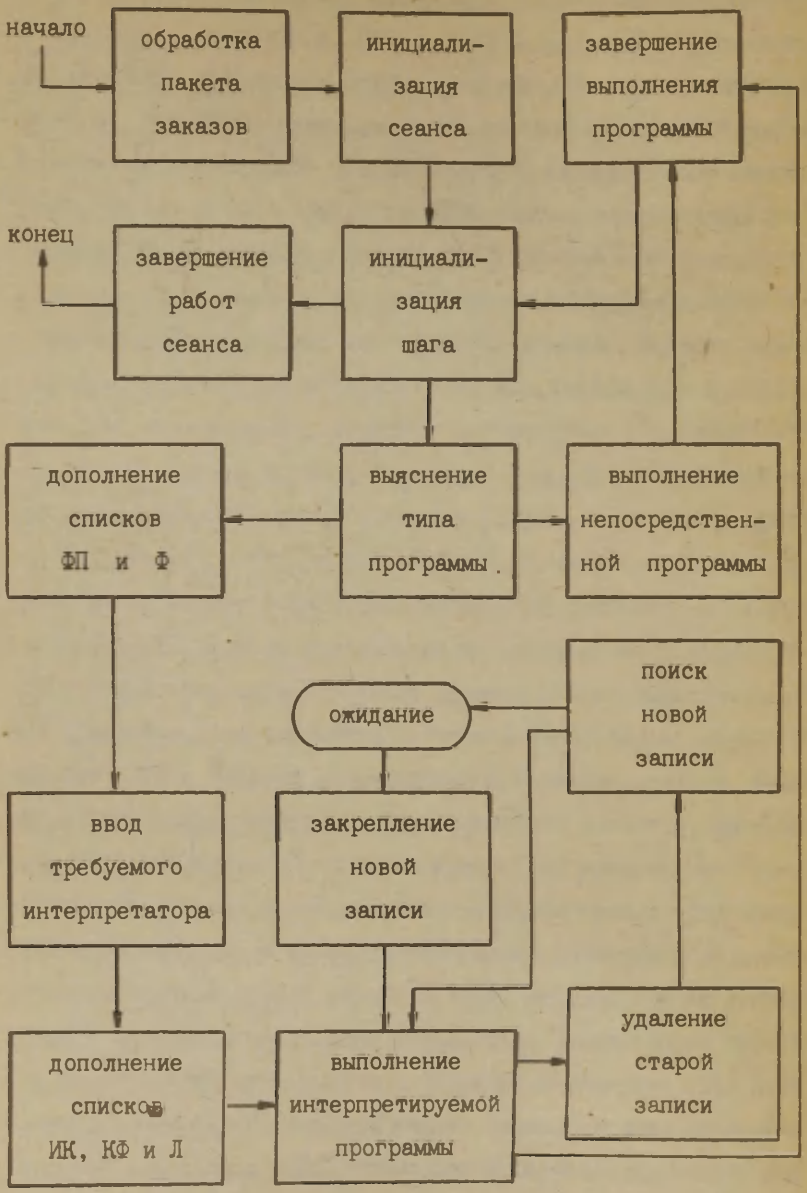
альный каталог фонда. Один пользователь, как правило, имеет на одном сеансе дело только с одним фондом, хотя нет ограничений на одновременное пользование данными нескольких фондов.

В начале сеанса МОНРАМА составляет для организации своей работы первоначальные варианты т.н. управляющих таблиц сеанса (УТС), которые при запуске очередного шага пользователя дополняются необходимой информацией. Их принципиальная схема приведена в [4] на стр. 5. По сравнению с этой схемой в ходе реализации введены лишь несущественные изменения. Основное из них состоит в том, что таблица монитора М задает теперь начала только трех списков (список П пользователей, список Ф открытых для пользователей файлов и список Л необходимых легенд), а список ОФ описаний файлов упразднен - каждому узлу списка Ф теперь непосредственно подчинено соответствующее описание файла (таким образом, описания файлов при необходимости вводятся в нескольких экземплярах). Каждому узлу списка Ф подчиняется еще таблица каталога (см.[5],стр.15) и список КФ используемых комплектов этого файла. Под каждым узлом списка КФ строится список ЗК введенных записей этого комплекта, ссылающийся на конкретные физические записи. Каждому узлу списка П подчинены список ФП файлов этого пользователя и список ИК имен комплектов, узлы которого совпадают с узлами списков ЗК. Списки ИК являются центральными при обработке записей файлов данных. Поэтому, для сокращения переходов по разным цепям ссылок, каждый узел ИК снабжается прямыми ссылками на все таблицы УТС, относящиеся к этому файлу и шагу пользователя. Кроме того, этот узел содержит информацию для работы над элементами соответствующей физической записи.

При составлении УТС опираются на данные заказа и информацию из используемых фондов. УТС необходимы прежде всего для совмещения одновременной работы нескольких программ над одними и теми же файлами, а также для сокращения операций обмена информацией. В каждый момент времени УТС показывают состояние работы системы РАМА на данном сеансе. Во избежание проведения недозволенных изменений в УТС или же их одновременной корректировки несколькими программами, ввод изменений в эти таблицы разрешается только специальными подпрограммами монитора, имеющими средства блокировки параллельной работы.

Упрощенная схема работы монитора приведена на следующей странице. Эта работа начинается с ввода и анализа пакета написанных на TCL (см. [7]) заказов. В ходе анализа заказ преобразовывается и записывается в рабочий файл сеанса. По результатам анализа стандартная процедура вызова следующей, управляющей части монитора дополняется нужными DD-предложениями и передается в очередь работ операционной системы ОС.

Управляющая часть монитора открывает файл системной печати, вводит данные о пользователях, создает в УТС таблицу монитора и список пользователей. С каждым пользователем при этом связываются два выходных файла - один из них предназначен для записи результатов решения, в другой помещается протокол выполнения заказа пользователя и описание состояния файлов данных в конце сеанса. После такой подготовки начинается параллельная работа: для каждого пользователя стартуется свой инициализатор шага (по сути дела одна и та же реентерабельная программа). Управляющая часть монитора переходит в состояние ожидания окончания работ всех пользователей.


начало
обработка пакета заказов
инициализация сеанса
завершение выполнения программы
конец
завершение работ сеанса
инициализация шага
дополнение списков ФП и Ф
выяснение типа программы
выполнение непосредственной программы
поиск новой записи
ожидание
ввод требуемого интерпретатора
закрепление новой записи
дополнение списков ИК, КФ и Л
выполнение интерпретируемой программы
удаление старой записи

Инициализатор шага анализирует возможность запуска очередного шага пользователя, дополняет УТС, открывает необходимые файлы данных и следит за ходом работы. Первым делом вводится заказ на выполнение шага и проверяется, выполнены ли заданные пользователем условия запуска (если они имеются). Если условия выполнены и шаг не связан с использованием файлов данных, то управление передается соответствующей рабочей программе (на схеме такая программа называется непосредственной программой). В таком случае УТС данного пользователя на рассматриваемом шаге содержат только таблицу П.

Если для выполнения шага требуются файлы данных, то инициализатор шага создает в УТС список ФП, заодно проверяя существование этих файлов в фонде. Далее блокируется ввод изменений в УТС другими инициализаторами и проверяется, совместимы ли указанные в заказе режимы использования файлов данных с режимами остальных пользователей, имеющими в данный момент доступ к этим файлам. В случае несовместимости режимов данный шаг вводится в состояние ожидания необходимых ему файлов. Если требуемые файлы данных еще не имеют пользователей, то список файлов Ф дополняется необходимыми узлами и связанными с ними описаниями ОФ. Дальнейшее дополнение УТС управляется уже рабочими программами пользователей.

Каждая системная рабочая программа (интерпретатор, утилит), обращающаяся к файлам данных, имеет стандартное начало для связывания имен комплектов с соответствующими файлами. Обычно в программах пользователя каждое имя комплекта (как переменная) связывается только с некоторой легендой (см., напр. [3]), а в заказе на выполнение этой программы опреде-

ляется, какие имена комплектов к каким файлам относятся. При этом описание соответствующего файла должно обязательно содержать связанную с именем комплекта легенду (иначе заказ считается ошибочным). По этим данным в УТС дополняются списки ИК, КФ и Л, а с каждым именем комплекта связывается рабочая область - информационное поле для обращения к конкретным записям. После составления названных таблиц УТС готовы для выполнения шага с использованием файлов данных.

Распределением записей из файлов данных между пользователями управляет специальная часть монитора - реентерабельная программа GETPUT. Эта программа ищет необходимые записи, вводит их в оперативную память, закрепляет за пользователями и выводит после завершения обработки.

Поиск записи происходит при этом всегда по ее ключу доступа, структура которого определяется описанием соответствующего файла (см.[8]). Ключи доступа всех записей данного файла собраны в каталог файла (см. [5]), так что просмотру подлежат лишь блоки файла, содержащие каталог. При поиске учитывается также возможность, что во время сеанса в оперативной памяти могут быть новые записи, не внесенные еще в каталог - их ключи доступа включаются в каталог лишь при выводе.

Поиск записи не обязательно проводят для ее последующего ввода. Напр., перед добавлением новой записи, такой поиск означает проверку, что данный файл еще не содержит записи с тем же ключом доступа. В этом случае, вместо ввода, в оперативной памяти генерируется новая запись, в которой заполнены лишь поля ключевых атомов. Если же запись ищется для ее удаления, то вместо ввода ее ключ доступа просто удаляется из каталога.

При вводе записи в оперативную память (или же при создании новой записи) она снабжается заголовком для размещения разной системной информации об этой записи. Из такой информации следует отметить указатели создания, изменения и удаления. Указатель создания необходим для выделения вновьсозданных записей, не внесенных еще в каталог файла. Указатель изменения регистрирует любое изменение, внесенное в запись: по этому указателю устанавливается необходимость вывода записи после обработки. Положение указателя удаления меняется тогда, когда кто-либо из пользователей удаляет эту запись: во избежание недоразумений запись физически удаляется лишь после окончания ее обработки всеми пользователями, за которыми она закреплена.

В программах пользователей, как правило, нет специальных операторов освобождения записи. Признаком освобождения является начало поиска новой записи по этому же имени комплекта. Если при освобождении окажется, что за этой записью не прикреплены другие пользователи, то запись выводится и в каталог вносится ее новый адрес. Вывод отпадает в том случае, когда по указателю изменения запись сохраняет свое первоначальное содержание.

Программа GETFUT организует также и циклы по записям. Признаком начала цикла является значение FIRST параметра F. После обработки очередной записи всегда снова обращаются к программе GETPUT, придавая параметру F значение NEXT. При таком обращении ищется следующая (по величине ключа доступа) запись, удовлетворяющая всем ограничениям, заданным в фильтре соответствующего цикла.

Напр., в случае языка DML интерпретация оператора цикла FOR (см. [3], стр. II8), а также оператора обновления REPL (см. [3], стр. I29) происходит по следующей упрощенной схеме:

Если в фильтре для управления циклом ограничение задано в виде условия, то это соответствующим параметром сообщается программе GETPUT, которая тогда выдает не очередную запись, а лишь значения ее ключевых атомов для проверки ограничения. Поэтому признак выхода из GETPUT может иметь три значения: "запись имеется" (на схеме обозначение +), "записи нет" (-) или "требуется проверка" (?). В последнем случае программа пользователя (напр., интерпретатор языка DML) проверяет ограничения фильтра и снова возвращается в программу GETPUT с положительным или отрицательным ответом - соответственно для выдачи этой записи или продолжения поиска.

Чтобы объединить указанные два способа обращения (т.е. F = FIRST и F = NEXT), программа GETPUT в любом случае ищет

запись, ключ доступа которой или совпадает с заданным ключом, или же является следующим по величине среди имеющихся.

Поиск требуемой записи всегда начинают по тем записям, которые уже имеются в оперативной памяти. Для этого просматривается список записей данного комплекта (список ЗК по обозначениям статьи [4]). Если подходящей записи там нет, или же ключ доступа обнаруженной записи не совпадает с данным ключом, то поиск продолжается по каталогу файла.

Следует отметить, что на время поиска по каталогу (или других операций с каталогом) просматриваемый файл монопольно прикрепляется к данному пользователю. Такое прикрепление, однако, не относится к записям этого файла, уже имеющимся в оперативной памяти.

Предположим теперь, что поиск записи предпринят с целью ее обработки, а не для проверки отсутствия или же удаления.

Если запись найдена по каталогу, то она вводится в оперативную память (см. [6]) и закрепляется за данным пользователем, т.е. адрес физической записи записывается в заданный узел списка ИК (см. [4], стр. 7).

Если запись найдена в оперативной памяти, то ее закрепление за пользователем может сопровождаться некоторым ожиданием. Дело в том, что хотя одновременное обращение разных пользователей к записям одного и того же файла и разрешается, каждая конкретная запись монопольно закрепляется за одним пользователем. Поэтому закрепление начинается с включения данного пользователя в специальную очередь, которая создается в заголовке физической записи. В виде исключения разрешается лишь удаление записи (т.е. изменение положения указа-

теля удаления) без ожидания освобождения этой записи. Включение пользователя в очередь сопровождается проверкой возможности "мертвой хватки".

Принципы организации файлов подробно описаны в [5]. Напомним только, что допустимы файлы с фиксированной и с плавающей границей. Разница между ними заключается в том, что в файлах первого типа обновляемая запись записывается на прежнее место (старый вариант стирается), а в файлах второго типа - на новое место, за последней границей. Во втором случае каждая граница идентифицируется временем ее фиксирования, и тем самым граница определяет некоторую версию файла. Физически границей файла является совокупность блоков каталога, выводимых в конце сеанса. Для каждого файла с плавающей границей создается специальный каталог версий. При этом введение изменений разрешается только в самую последнюю версию, все остальные доступны только для чтения в монопольном режиме пользования (см. [7]).

Таким образом, в файлах с плавающей границей имеется несколько версий одной и той же записи, соответствующих времени их создания и изменения. Во избежание чрезмерного роста разных версий одного файла, на каждом сеансе может порождаться только одна версия. Это достигается тем, что пользователю, обращающемуся к некоторой записи, всегда предоставляется ее самая последняя версия - в случае, когда рассматриваемая запись не изменилась на этом сеансе, ее последняя модификация совпадает с последней версией.

По окончанию работы сохраняются лишь самые последние модификации записей, все промежуточные становятся не только не-

доступными, но и физически устраняются. При этом, если разные пользователи изменили одни и те же элементы данных одной записи, то в новой версии файла сохраняются данные, записанные пользователем, имеющим доступ к записи в качестве самого последнего на этом сеансе.

Использованные на сеансе файлы закрываются в конце сеанса. В смысле системы РАМА это действие заключается в устранении промежуточных модификаций, в выводе каталога файла (т.е. в фиксировании новой границы) и печати сообщения о состоянии файла всем пользователям, которые работали с рассматриваемым файлом. Извещение пользователей о состоянии файла в конце работы сеанса связано с возможностью аварийной ситуации. Именно, если по каким-то причинам (например, переполнение файла, сбой ввода блока каталога и т.п.) не удается закончить работу с файлом нормально, то файл объявляется находящимся в аварийной ситуации. Признак аварийной ситуации вводится в каталог фонда и доступ к таким файлам имеют только системные программы для анализа причин аварии и ее устранения. Так как в файлах с плавающей границей сохраняются предыдущие версии, то самым простым способом устранения аварийной ситуации является повторение всех работ сеанса, относящихся к рассматриваемому файлу.

В конце сеанса выдается общий протокол работы монитора и протоколы работы пользователей. В протоколе монитора приводятся данные о ходе работ пользователей и об использовании ими файлов. Протоколы работы пользователей характеризуют течение хода работ по шагам и извещают пользователей о состояниях файлов и об их версиях.

Литература

I. Каазик Ю., Томбак М., Система РАМА для управления базой данных. Труды ВЦ ТГУ, 1978, 41, 3-6.

2. Изотамм А., Каазик Ю., Томбак М., Язык определения записи. Труды ВЦ ТГУ, 1978, 41, 7-64.

3. Каазик Ю., Рауп А., Язык манипулирования данными. Труды ВЦ ТГУ, 1978, 41, 97-140.

4. Каазик Ю., Томбак М., Совместное пользование данными в системе РАМА. Труды ВЦ ТГУ, 1982, 49, 3-11.

5. Рауп А., Организация файлов в системе РАМА. Труды ВЦ ТГУ, 1982, 49, 12-25.

6. Изотамм А., Представление записи в системе РАМА. Труды ВЦ ТГУ, 1982, 49, 26-41.

7. Ээремаа К., Язык управления заданиями системы РАМА. Труды ВЦ ТГУ, 1983, 50, 3-22.

8. Каазик Ю., Ээремаа К., Реализация языка описания файлов. Труды ВЦ ТГУ, 1984, 51, 3-11.

О РЕАЛИЗАЦИИ ЯЗЫКА МАНИПУЛИРОВАНИЯ ДАННЫМИ

Т. Тамме

Описание языка DML для манипулирования данными в системе РАМА приведено в статьях [4] и [5]. В статье [5] также рассматриваются основные идеи реализации этого языка. Настоящая статья посвящена описанию работы DML-транслятора.

1. Структура транслятора

DML-транслятор состоит из пяти частей, предназначенных для выполнения следующих действий:

1) синтаксический анализ DML-программы и построение соответствующего дерева анализа;

2) формирование заголовки DML-программы, создание таблиц имен комплектов и переключателей;

3) рекурсивная трансляция операторов программы;

4) преобразование констант программы;

5) окончательное формирование и вывод внутреннего представления DML-программы.

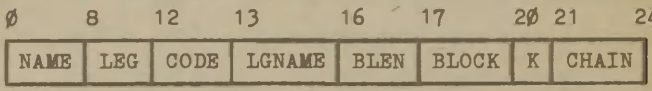
Все подпрограммы транслятора являются повторно выполнимыми. Транслятор предназначен для работы под управлением специального монитора системы РАМА, хотя его можно использовать и в качестве самостоятельной программы.

2. Внутреннее представление программы

Внутреннее представление DML-программы создается как связной участок памяти, содержащий кроме оттранслированной программы еще заголовок, таблицу имен комплектов, таблицу переключателей, а также список имен легенд и констант.

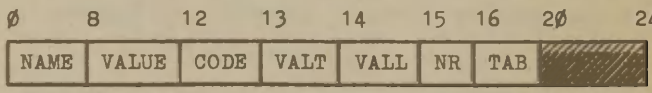
Заголовок программы содержит имя DML-программы, адрес и длину таблицы имен комплектов, адрес и длину таблицы переключателей, длину внутреннего представления и ссылку на корень дерева оттранслированной DML-программы.

Элемент таблицы имен комплектов имеет структуру:



где поле NAME содержит идентификатор имени комплекта, поле LEG адрес легенды, поле CODE определяет тип и режим использования имени комплекта, поле LGNAME содержит адрес имени легенды, поля BLEN и BLOCK определяют соответственно длину в словах и адрес блока для значений внутренних ключей записи, поле K содержит количество ключей записи и поле CHAIN адрес соответствующего звена в системной цепи имен комплектов.

Элемент таблицы переключателей имеет структуру:



где поле NAME содержит имя переключателя. Поля VALUE, VALT и VALL определяют соответственно адрес, тип и длину начального значения переключателя. Поле CODE задает информацию о типе и режиме использования переключателя. Поля NR и TAB содержат длину и адрес цепи переключателей того же имени.

Все ссылки во внутреннем представлении являются относительными. Память для внутреннего представления выделяется в начале работы транслятора по первичной оценке. При заполнении этого участка памяти спрашивается новый участок и все представление дублируется из старого участка в новый.

3. Оттранслированная программа

Оттранслированная программа является продуктом преобразований над первоначальным деревом анализа DML-программы, которое описано в статье [5]. Структура дерева анализа DML-программы соответствует логической структуре программы. Каждому оператору, составному имени, фильтру, константе и т.д. сопоставляется одна или несколько вершин дерева.

В ходе работы транслятора дерево несколько изменяется. Например, в интересах более эффективного выполнения программы в случаях CASE- и IF-операторов добавляют вспомогательные ссылки, а все WITH-операторы удаляются из дерева. Составные имена представляются вершинами, содержащими ссылки на соответствующие вершины дерева описания рассматриваемой легенды.

Несколько изменена и структура вершины дерева оттранслированной DML-программы. Такая вершина, длина которой расширена до 20 байтов, имеет теперь формат:

байт 1	байт 2	байт 3	байт 4	
B	RIGHT			слово 1
NR	DOWN			слово 2
TYPE	LINK			слово 3
SYMB	ROW		SEM	слово 4
CODE	AT	J	L	слово 5

Поля B, RIGHT и DOWN предназначены для ссылок внутри дерева (их значение см. [2], стр. 35). Поле SEM содержит семантический код вершины. Поля SYMB и ROW используются соответственно для сохранения номера символа в строке и номера строки в исходном тексте DML-программы. Значения остальных полей зависят от семантического кода рассматриваемой вершины. Поле NR содержит либо номер имени комплекта или переключателя в соответствующей таблице, либо длину константы в байтах, либо порядковый номер замка в системной цепи замков, либо число движений вверх по операторам LEAVE, BACK и STOP. Поле TYPE задает тип соответствующих данных, а поле LINK - либо адрес константы, либо ссылку на вершину дерева описания легенды.

Поле CODE содержит информацию о типе или же о режиме использования рассматриваемой повторяющейся группы в записи. Значением поля AT является либо количество ключей на пути до заданного места, либо количество ключей, принадлежащих замку. Поле J определяет либо количество замков на пути, либо вариант доступа к замку. Поле L содержит количество ключей или же характеристику организации рассматриваемой повторяющейся группы. Пятое слово иногда используется для размещения дополнительных ссылок внутри дерева.

4. Типы данных

Типы данных для языка DML описаны в статье [5]. Типы кодируются кодами языка определения записи (см. [1], стр. 20).

По длине представления в памяти машины типы разделены в подтипы, характеристики которых можно задавать с помощью таблицы 1. Соответствие между подтипами и данными следующее:

1-2 двоичные числа без знака,
3-4 двоичные числа со знаком,
5-7 числа с плавающей запятой,
8 десятичные числа,
9 шестнадцатеричные строки,
10-12 календарные даты (как десятичные числа),
13-14 символьные строки.

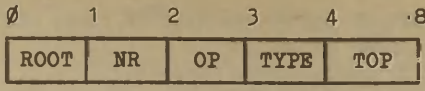
Подтип 13 предназначен для строк фиксированной длины.

Таблица 1.

пор. номер подтипа	внутренний код	макс. длина представления	тип	мин. длина представления
1.	10	4	INT	4
2.	11	2	INT	2
3.	12	4	INT	4
4.	13	2	INT	2
5.	20	4	REAL	4
6.	21	8	REAL	8
7.	22	16	REAL	16
8.	30	16	DEC	1
9.	40	255	HEX	1
10.	50	4	DATE	4
11.	51	5	DATE	5
12.	52	3	DATE	3
13.	60	255	TEXT	1
14.	61	255	TEXT	1

В ходе реализации пришлось расширить число допустимых преобразований между типами: разрешено теперь и преобразование каждого скалярного типа в строковый тип TEXT.

Для нахождения типов выражений и других вершин дерева используется т.н. WRF-магазин, который строится из областей сохранения подпрограмм транслятора. При выделении памяти для такой области заказываются 8 лишних байтов - эти поля и образуют WRF-магазин, элемент которого имеет структуру:



Поле ROOT используется в качестве признака головки магазина. Поле NR содержит номер имени комплекта в соответствующей таблице (в программах для трансляции WITH-оператора и фильтра). Значением поля OP является семантический код рассматриваемой вершины. Поле TYPE задает тип соответствующих данных. Поле TOP содержит либо адрес некоторой вершины дерева анализа, либо адрес вершины в дереве описания легенды.

Для DML-программы данными являются константы, переключатели и части записи, соответствующие составным именам программы. Транслятор приписывает всем данным конкретный подтип. Если какая-то подпрограмма транслятора обнаружит новые данные, то их тип записывается в WRF-магазин надпрограммы. Эта программа в свою очередь сделает необходимые преобразования над типами и засылает полученный тип в свою надпрограмму.

5. Составные имена

Трансляция составных имен DML-программы осуществляется по следующему упрощенному алгоритму:

1) если первой компонентой составного имени является имя комплекта, то имеем дело с вершиной дерева описания соответствующей легенды и переходим к пункту 4;

2) если первой компонентой является переключатель, то переходим в режим трансляции переключателя;

3) если в WRF-магазине найдется подходящее имя комплекта, то ищем вершину в дереве описания легенды и переходим к пункту 4, иначе заменим составное имя константой (конец работы);

4) ищем из дерева анализа DML-программы ближайший фильтр, которому подчиняется данное место;

5) если такого фильтра нет, то переходим к пункту 8;

6) проверим ключи на пути (в случае дополнительного доступа проверяются только типы ключей);

7) если для повторяющейся группы путь не заканчивается на уровне рассматриваемой группы, то имеем дело с экземпляром соответствующей группы;

8) если путь к месту определен не полностью, то проверим, является ли составное имя частью сравнения в каком-то критерии фильтра; если составное имя находится в левой части сравнения и идентично с именем соответствующего ключа, то заменим рассматриваемое составное имя на ключевое слово KEY.

6. Фильтры

В языке DML каждый фильтр однозначно связывается с каким-то составным именем. Поэтому трансляция фильтра начнется с трансляции соответствующего составного имени. Если выясняется, что имени не соответствует ни одной вершины дерева описания легенды, то фильтр определен некорректно. Точно таким же образом как при анализе составного имени ищется ближайший фильтр, которому данный фильтр подчиняется и проверяются соответствующие ключи места. Оставшиеся ключи ставят в соответствие критериям рассматриваемого фильтра.

Если ключей оказывается меньше чем критериев, то лишние критерии удаляются. Если составное имя определяет какую-то повторяющуюся группу записи, то лишние критерии указывают на использование конкретного экземпляра данной группы и устра-

нению не подлежат. Если же критериев не хватает, то в конец фильтра добавляют циклы по всем экземплярам рассматриваемой повторяющейся группы (или по всем записям).

В системе РАМА повторяющимся группам, не имеющим явных ключей, приписывается ключ типа INDEX. Поэтому DML-транслятор требует, чтобы пользователь указал в фильтре порядковый номер экземпляра в случае таких повторяющихся групп.

Если фильтр указывает на использование дополнительным доступом к месту, то данное составное имя (определяющее место) связывается со специальной константой. Соответствующая такой константе вершина дерева анализа содержит ссылку на специальную форму составного имени в области констант. Сама константа используется DML-интерпретатором при реализации дополнительных доступов.

Так как DML-транслятор сам прибавляет или устраняет критерии, то при написании фильтров требуется особенная внимательность. Для иллюстрации использования фильтров рассмотрим в качестве примера следующую легенду (см. [3]):

```
LEG ШКОЛЫ KEY=ШКОЛА TEXT
* 1 ШКОЛА PICT=50
* 1 КОЛИЧ CONST NAT
* 1 КЛАСС REP=КОЛИЧ KEY=НОМЕР
  * 2 НОМЕР PICT=3
  * 2 КОЛИЧ CONST MAX=40 NAT
  * 2 УЧЕНИК REP=КЛАСС.КОЛИЧ PICT=20
      SORT KEY=ФАМ,ИМЯ
    * 3 ИМЯ
    * 3 ФАМ
    * 3 ПОЛ SCOPE=[М,Ж]
  * 2 ПРЕДМЕТ REP PICT=8
END
```

Для описания цикла по всем школам, определенным этой легендой, можно в DML-программе задать оператор (предпологая, что соответствующее имя комплекта определена выше)

FOR ШКОЛЫ

который в том случае, если ключи записи в программе еще не встретились, принимается как оператор

FOR ШКОЛЫ (*)

Если же ключ записи уже встречался, то указанный выше оператор означает лишь разовое выполнение подчиненного оператора над записью, полученной по предыдущему фильтру.

Для обращения ко всем ученикам третьих классов всех школ, фамилия которых KAASIK, можно задать оператор

REPL УЧЕНИК (*;3;KAASIK)

Транслятор прибавляет в конец приведенного фильтра еще цикл по всем именам учеников, т.е. оператор принимается к выполнению уже в виде:

REPL УЧЕНИК (*;3;KAASIK;*)

Однако, для удаления всех рассмотренных только что учеников нельзя использовать оператор

DEL УЧЕНИК (*;3;KAASIK;*)

Дело в том, что режим удаления относится здесь только к двум последним критериям (ключам), а цикл берется по всем записям. Вообще, в операторе удаления не советуется использовать в одном фильтре циклы и наборы ключей разных уровней записи. Например, для реализации требуемого удаления можно представить отрывок программы

FOR УЧЕНИК (*;3;KAASIK;*)
    DEL УЧЕНИК

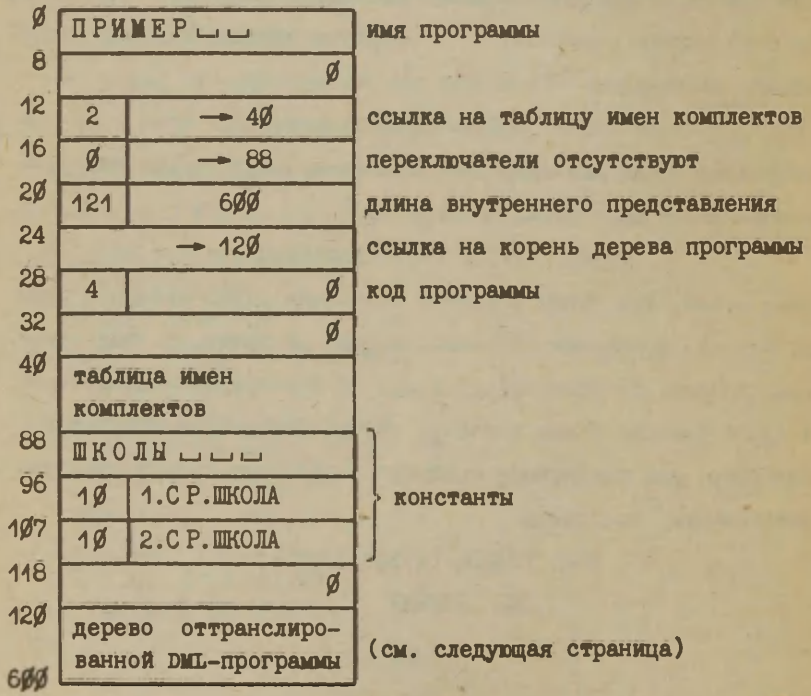
7. Пример внутреннего представления

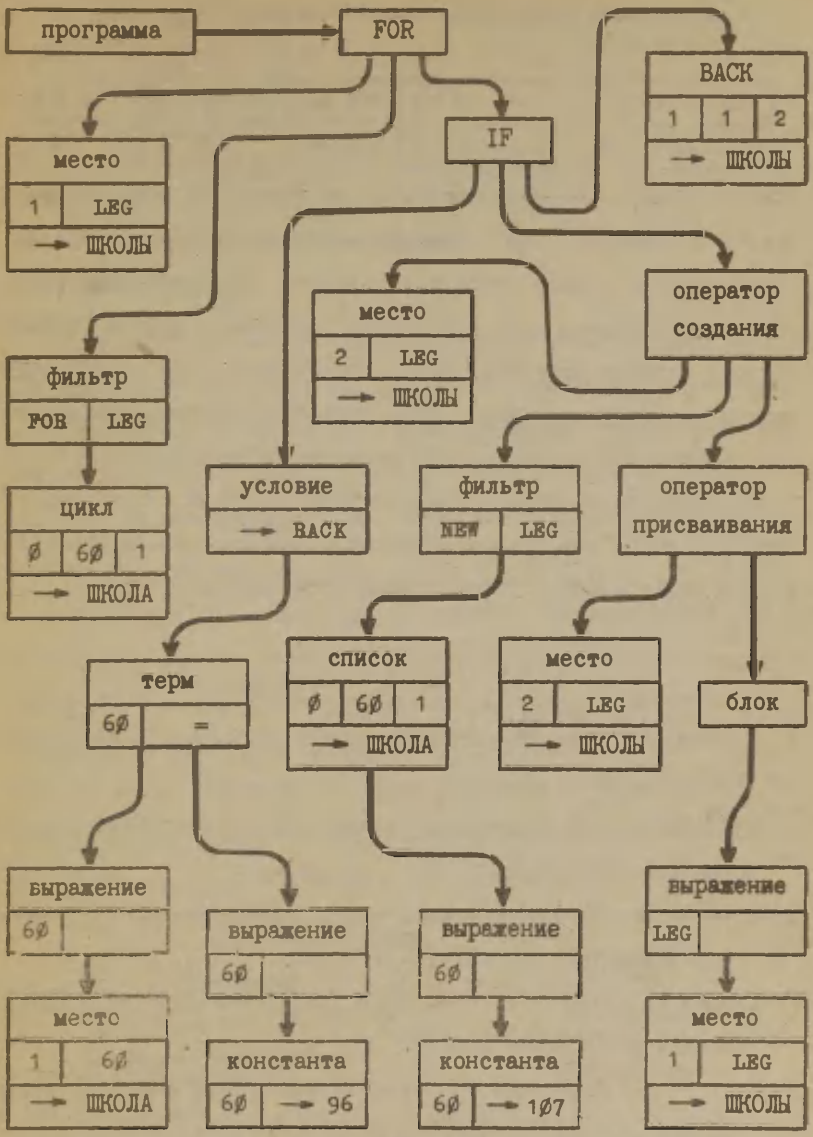
Рассмотрим DML-программу, опирающую на легенду ШКОЛЫ:

```
DML ПРИМЕР
LEGEND ШКОЛЫ SET X,Y
FOR X(*)
    IF X.ШКОЛА='1.СР.ШКОЛА'
       THEN NEW Y('2.СР.ШКОЛА')
            Y:=X
       ELSE BACK X(1)
    FI
```

Эта программа просматривает все школы и создает запись для 2-ой средней школы с данными 1-ой средней школы.

Транслятор строит для приведенной DML-программы следующее внутреннее представление:




программа
FOR
BACK
1 1 2
→ ШКОЛЫ
место
1 LEG
→ ШКОЛЫ
IF
место
2 LEG
→ ШКОЛЫ
оператор создания
фильтр
FOR LEG
цикл
Ø 6Ø 1
→ ШКОЛА
условие
→ BACK
фильтр
NEW LEG
оператор присваивания
терм
6Ø =
список
Ø 6Ø 1
→ ШКОЛА
место
2 LEG
→ ШКОЛЫ
блок
выражение
6Ø
выражение
6Ø
выражение
6Ø
выражение
LEG
место
1 6Ø
→ ШКОЛА
константа
6Ø → 96
константа
6Ø → 1Ø7
место
1 LEG
→ ШКОЛЫ

7

Таблица имен комплектов имеет следующую структуру:

Ø	8	12	13	16	2Ø	21	24
X␣␣␣␣␣␣␣␣	Ø	R	→ 88	Ø	1	Ø	
Y␣␣␣␣␣␣␣␣	Ø	W	→ 88	Ø	1	Ø	

где R покажет, что имя комплекта используется лишь для чтения, а W означает, что с помощью этого имени можно и писать.

Структура дерева оттранслированной DML-программы схематически изображена на предыдущей странице. Первая строка каждой вершины описывает там семантический код, а остальные строки содержат важнейшие характеристики вершины. Конструкция → обозначает ссылку либо на вершину дерева описания легенды, либо на вершину рассматриваемого дерева, либо на константу. Слово LEG свидетельствует о наличии ключей записи в данном фильтре или о рассмотрении всей записи.

Литература

I. Изотамм А., Дерево описания записи в системе РАМА. Труды ВЦ ТГУ, 1980, 43, 3-35.

2. Изотамм А., Представление записи в системе РАМА. Труды ВЦ ТГУ, 1982, 49, 26-41.

3. Изотамм А., Каазик Ю., Томбак М., Язык определения записи. Труды ВЦ ТГУ, 1978, 41, 7-64.

4. Каазик Ю., Рауп А., Язык манипулирования данными. Труды ВЦ ТГУ, 1978, 41, 97-140.

5. Рауп А., Реализация языка манипулирования данными. Труды ВЦ ТГУ, 1980, 45, 36-65.

РЕАЛИЗАЦИЯ ЯЗЫКА ВЫВОДА ДАННЫХ

Я.Р. Пейал, В.К. Соо

Язык DOL для вывода данных системы РАМА описан в [5]. Реализация языка привела, однако, к пересмотру ряда вопросов, связанных в основном с использованием составных имен. Настоящая статья представляет результаты этой работы: уточнения в описании языка и некоторые специальные средства.

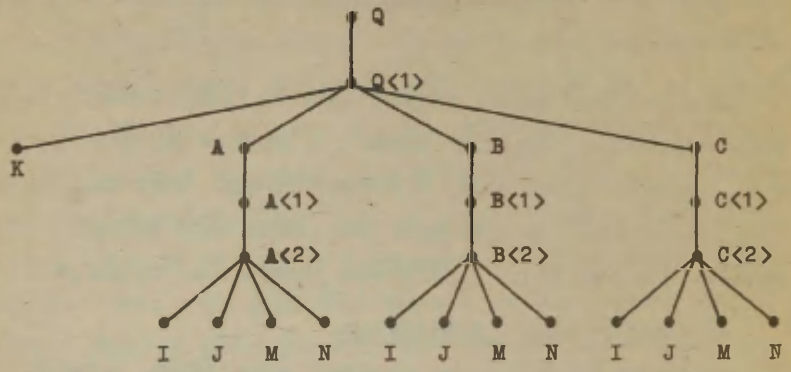
1. Принципы работы транслятора

Обработка DOL-программы проходит в два этапа. Сначала по тексту программы и описанию записи строится внутреннее представление программы, которое затем интерпретируется в соответствии с имеющимися данными. Предусмотрен и режим отладки, в котором без обращения к данным печатается "макет таблицы".

Пример 1. Проиллюстрируем работу транслятора на небольшом примере. Пусть описание записи (легенда) имеет вид:

```
LEG   Q   KEY=K   NAT
  *1 K
  *1 A   REP   KEY=I,J
    *2   I
    *2   J
    *2   M
    *2   N
  *1 B   LIKE=A
  *1 C   LIKE=A
END
```

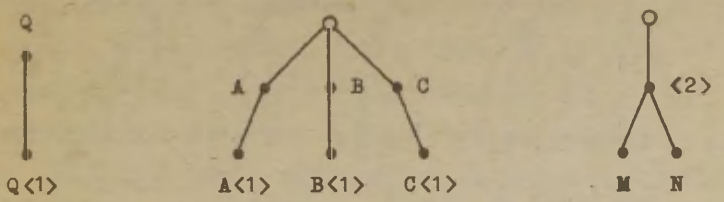
Дерево описания записи (см.[1]) преобразуется DOL-транслятором так, что каждой компоненте составного ключа (включая ключ записи) соответствует отдельный промежуточный уровень. В случае приведенной легенды это дерево принимает следующий вид:



Рассмотрим теперь следующую DOL-программу:

```
DOL  QTAB
  LEG  Q
  ** DATA
    A ( * ; 5:8 )
    B ( * ; 5:8 )
    G ( * ; 5:8 )
  ** TAB  DIV=[ Q<1> ]
    - Q<1> TITLE=(LEPT/'K=',(K)=2/)
  ** ROW  DIV=[ A<1> , B<1>, C<1> ]
    - A<1> NOTAB ¤ FORMAT/(I)=3/
    - B<1> NOTAB ¤ FORMAT/(I)=3/
    - C<1> NOTAB ¤ FORMAT/(I)=3/
  ** COL  DIV=[ M , N ]
    - ROW<2> NOTAB ¤ FORMAT/(J)=1/
    - M PICT=2
    - N PICT=3
END
```

Среди внутренних структур, которые образуются транслятором, имеются т.н. деревья макета, принимающие для данной программы и легенды следующий вид:



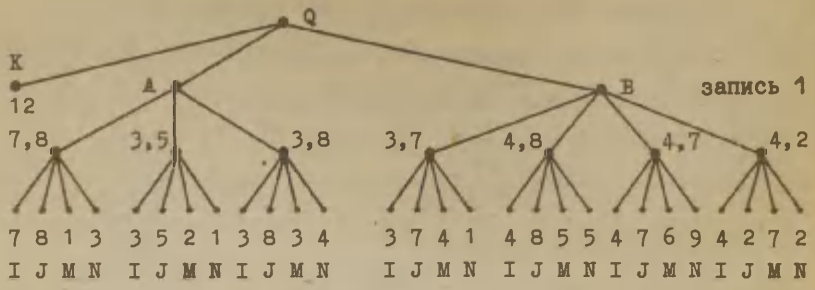
макет пня | макет шапки строк | макет шапки столбцов

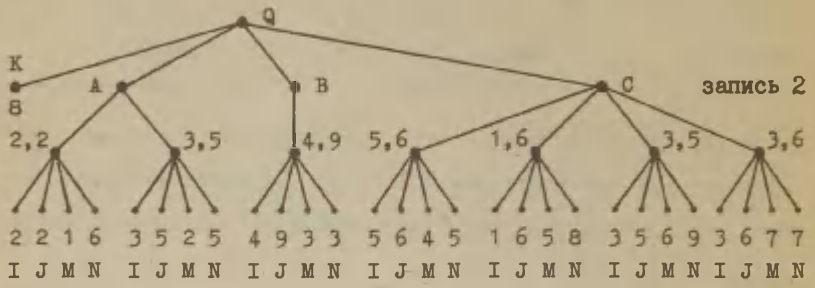
(пустыми кружками здесь обозначены вершины, явно не представленные в соответствующей шапке). По этим деревьям в режиме отладки печатается макет таблицы, который в данном случае получается в виде:

K = * *

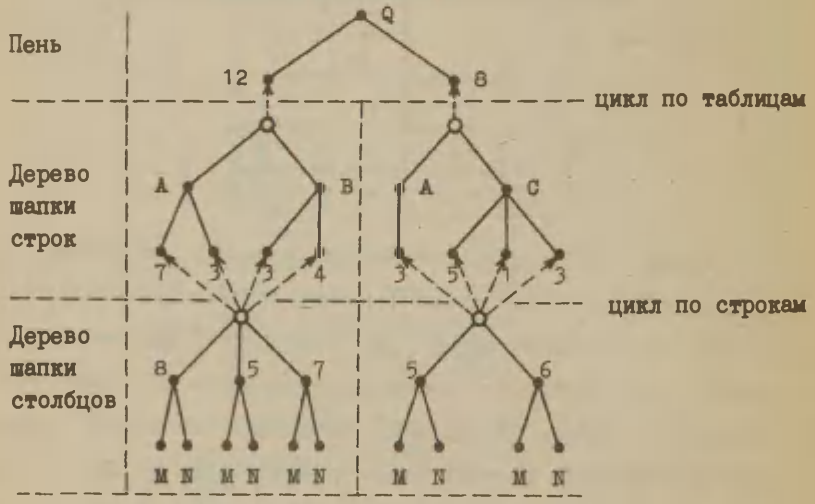
		*	
		M	N
A	* * *	* *	* * *
B	* * *	* *	* * *
C	* * *	* *	* * *

В макете отражаются все разделы исходной программы, за исключением описания данных (и самих данных). Для иллюстрации работы интерпретатора над физическими данными предположим, что конкретный заказ на выполнение программы QTAB обращается к файлу, содержащему следующие две записи комплекта Q (структура записей здесь соответствует дереву описания записи до его обработки DOL-транслятором):





Ход интерпретации показан на следующем рисунке:



Деревья таблицы 1 Деревья таблицы 2

Сначала макет пня превращается в пень: каждому листу пня будет соответствовать одна таблица. Затем для каждой таблицы создаются дерево шапки строк и дерево шапки столбцов. При этом учитываются заданные в программе критерии для выбора данных. В нашем примере дерево шапки столбцов содержит те значения ключей, соответствующие вершине <2>, которые встречаются в данной записи и принадлежат отрезку [5,8].

Результат работы программы приведен на следующем рисунке (но таблицы здесь расположены не последовательно, а рядом):

K=12

		8		5		7	
		M	N	M	N	M	N
A	7	1	3				
	3	3	4	2	1		
B	3					4	1
	4	5	5			6	9

K= 8

		5		6	
		M	N	M	N
A	3	2	5		
C	5			4	5
	1			5	8
	3	6	9	7	7

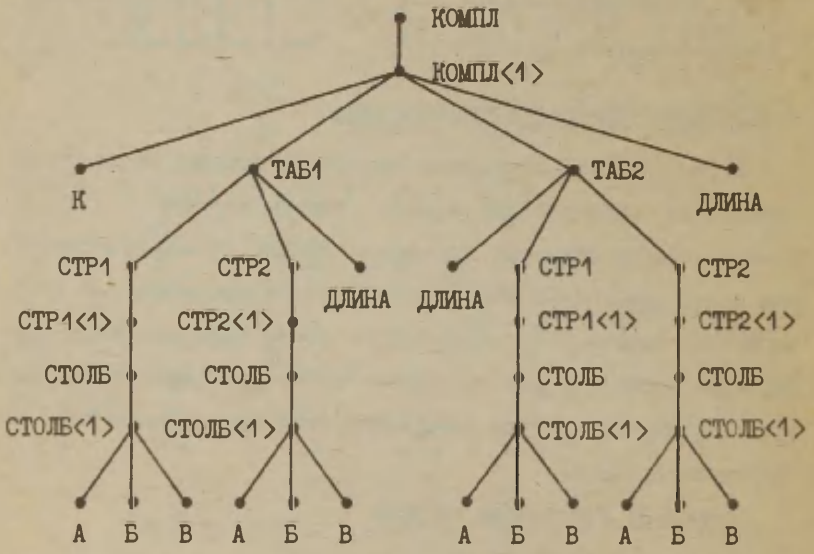
2. Использование составных имен

В системе РАМА составное имя обычно ссылает на некоторую однозначно определенную вершину дерева легенды (см. [3]). Несколько иная ситуация возникает, однако, в DOL-программах, где часто необходимо только одним именем указывать на целое множество вершин, полные составные имена которых имеют совпадающие окончания, но различные префиксы. Иллюстрируем такую ситуацию при помощи следующего сравнительно простого, но типичного примера.

Пример 2. Рассмотрим легенду

```
LEG КОМПЛ KEY=K NAT
 *1 K
 *1 ТАБ1
   *2 СТР1 AHRAY[3]
     *3 СТОЛБ REP=A HASH
       *4 А
       *4 Б
       *4 В
   *2 СТР2 REP=2
     *3 СТОЛБ LIKE=СТР1.СТОЛБ
   *2 ДЛИНА
 *1 ТАБ2
   *2 ДЛИНА
   *2 СТР1 LIKE=ТАБ1.СТР1
   *2 СТР2 LIKE=ТАБ1.СТР2
 *1 ДЛИНА
END
```

в случае которой преобразованное DOL-транслятором дерево описания записи принимает следующий вид:

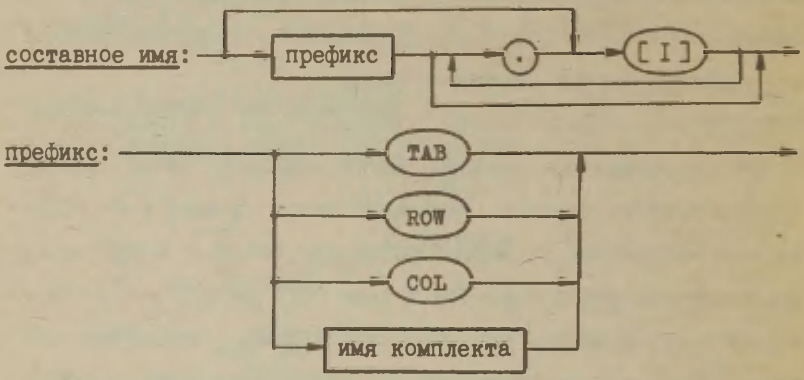


Предположим, что из каждой записи этого комплекта требуется печатать две таблицы, имеющие общий вид:

		СТОЛБ					
		10		3		2	
		Б	В	Б	В	Б	В
СТР1	1						
	2						
	3						
СТР2	1						
	2						

Для получения таких таблиц в соответствующей DOL-программе необходимо задать разбиение таблиц в виде: [ТАБ1,ТАБ2]. Разбиение строк в этих таблицах должно быть определено вершинами, имеющими полные составные имена
ТАБ1.СТР1<1> , ТАБ1.СТР2<1> , ТАБ2.СТР1<1> и ТАБ2.СТР2<1>. В DOL-программе обе таблицы имеют, однако, общее описание строк и поэтому вершины разбиения строк должны быть представлены независимо от вершин разбиения таблиц. В данном случае разбиение строк задается в виде [СТР1<1>,СТР2<1>]. Такая запись воспринимается как множество вершин, уточненные составные имена которых получаются добавлением в начало каждого имени в списке (составных) имен соответствующих вершин разбиения таблиц, но только без номеров уровня ключа. Таким же образом представляется и нижнее разбиение (в наших таблицах разбиение столбцов), исходя теперь из вершин среднего разбиения (разбиения строк). В данном случае разбиение столбцов должно быть, следовательно, задано в виде [Б,В].

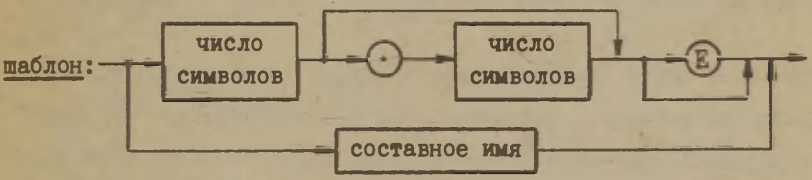
Из примера выясняется, что с каждым составным именем в DOL-программе в скрытом виде связано и некоторое множество префиксов, т.е. составных имен, одно из которых для точного определения вершины дерева описания записи добавляется к составному имени, заданному в DOL-программе (если ничего добавить не надо, то таким префиксом можно считать имя комплекта). Для большинства конструкций, содержащих составные имена, эти префиксы определены однозначно (как, например, в разбиениях). Но в язык DOL введены и конструкции, не имеющие таких естественных префиксов: их выбор желательно оставить программисту. Поэтому введено несколько измененное определение составного имени в языке DOL:



Префиксы TAB, ROW и COL указывают соответственно на составные имена вершин разбиения таблиц, разбиения строк и разбиения столбцов. Имя комплекта означает лишь то, что префиксы фактически не добавляются. Напомним еще, что по умолчанию именем комплекта считается имя легенды и в DOL-программе не допускается совпадение имени комплекта с именем какой-либо вершины в легенде.

Во многих конструкциях языка DOL префиксы составных имен определены однозначно. Например, в заголовке программы, а также в описаниях данных или таблиц единственным допустимым префиксом является имя комплекта. В описании строк или столбцов при задании среднего разбиения префиксом может быть только TAB, а при задании нижнего разбиения - соответственно COL или ROW. Последнее относится также к названиям ортогонов и значениям в форматах ключевых ортогонов. Во всех этих случаях префиксы можно опускать (добавляются по умолчанию).

Приступим теперь к рассмотрению тех конструкций языка DOL, в которых префикс играет существенную роль. В связи с тем, что в языке RDL изменена конструкция "длина" (см. [2] с. 37), соответствующее изменение введено и в язык DOL. Конструкция "шаблон" определяется теперь следующей схемой:



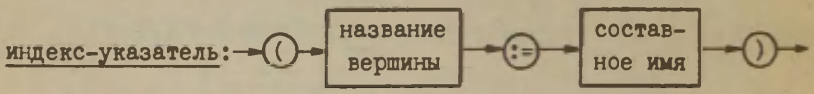
Составное имя здесь должно ссылать на атомы, однозначно определенные для каждой таблицы. Это значит, что префиксом может быть имя комплекта или TAB. В первом случае атом должен находиться в ветке корня записи (понятие "ветка" определено в статье [4]). При префиксе TAB составное имя может ссылать на несколько атомов (один для каждой таблицы), которые должны находиться в ветках вершин разбиения таблиц. По умолчанию префиксом составного имени в конструкции "шаблон" считается имя комплекта.

Пример 3. Для таблиц примера 2 описание ортогона в шапке столбцов можно задавать в виде:

- СТОЛБ<1> NOTAB И FORMAT/(A)=ТАБ.ДЛИНА/

Этим указывается, что количество символов для печати ключа А следует в первой таблице взять по значению атома ТАБ1.ДЛИНА, а во второй по значению атома ТАБ2.ДЛИНА . При желании печатать все таблицы по одному и тому же шаблону в формате можно указать (А)=ДЛИНА и шаблон определяется по значению атома с полным составным именем КОМПЛ.ДЛИНА или просто ДЛИНА .

Небольшое изменение введено и в конструкцию "индекс-указатель" (см. [5] с. 58), которая представляется теперь в виде следующей схемы:



Слева от знака ":=" указывается индекс (измерение) массива, значение которого заменяется (как и в конструкции "значение" в формате ключевого ортогона этим индексом может быть лишь ключ основного доступа на пути от вершины предыдущего разбиения до вершины рассматриваемого ортогона). Составное имя справа от знака ":=" должно (аналогично составному имени в конструкции "шаблон") ссылать на вершину, которая однозначно определена для каждой таблицы. Единственное различие в том, что ссылаемая вершина должна являться корнем одномерного массива (повторяющейся группы).

Пример 4. В целях иллюстрации использования префиксов перепишем DOL-программу ОЦЕНКИ, которая приведена в статье [5] на с. 60, добавляя ко всем составным именам префиксы:

```
DOL ОЦЕНКИ LEG ШКОЛА
 TITLE=(MIDD/(ШКОЛА.НОМЕР)=2, 'СРЕДНЯЯ ШКОЛА'|
 '2. СЕМЕСТР'|'УСПЕВАЕМОСТЬ ПО КЛАССАМ'/)
 FOOTN=(LEFT/DATE/  RIGHT/'ПОДПИСЬ:      '/)
** DATA  ШКОЛА (ШКОЛА.НОМЕР=5)
         ШКОЛА.ТАБЕЛЬ (2;*)
** TAB   DIV=[ШКОЛА.КЛАСС<1>]
 -ШКОЛА.КЛАСС<1> TITLE=(LEFT/(ШКОЛА.КЛАСС.НОМЕР),'КЛАСС'/)
** ROW   DIV=[TAB.УЧЕНИК<3>]  LINES=NO
 -TAB.УЧЕНИК NOTAB
 -TAB.УЧЕНИК<*> NOTAB ¤ FORMAT/INDEX,
     (TAB.ФАМИЛИЯ)=8, ' ', (TAB.ИМЯ)=8/
** COL   DIV=[ROW.ТАБЕЛЬ<2>,ROW.СРЕДНЯЯ]
 -ROW.ТАБЕЛЬ NOTAB
 -ROW.ТАБЕЛЬ<1> NOTAB ¤ NOTAB
 -ROW.ТАБЕЛЬ<2> NOTAB PICT=1
      ¤ FORMAT/(ROW.ТАБЕЛЬ<2>:=TAB.ПРЕДМЕТЫ)=8/
 -ROW.СРЕДНЯЯ PICT=1.2
END
```

3. Заполнение угла таблицы

Описанные до сих пор средства в языке DOL позволяют создать лишь таблицы с пустыми верхними левыми углами. Во многих случаях, однако, целесообразно переместить некоторые ортогоны из шапки строк в этот угол.

Пример 5. Рассмотрим таблицы, приведенные в статье [5] на с. 60. Если в соответствующей DOL-программе удалить атрибут NOTAB у ортогона УЧЕНИК, то получаем таблицы в виде:

		РУС.ЯЗЫК	...	СРЕДНЯЯ
УЧЕНИК	1. ААМОС , ТИЙУ	5		4.86
. . . . .	. . . . .	. . . . .	. . . . .	. . . . .
	38. ТИКАН , ТАРМО	5		4.67

Более экономный вид таких таблиц следующий:

УЧЕНИК	РУС.ЯЗЫК	...	СРЕДНЯЯ
1. ААМОС , ТИЙУ	5		4.86
. . . . .	. . . . .	. . . . .	. . .
38. ТИКАН , ТАРМО	5		4.67

В целях достижения такого вида введен новый атрибут CORNER, который можно включить в список атрибутов ортогона шапки строк. При ключевом ортогоне этот атрибут используется только для части имен ключей (индексов). Вершина в макете шапки строк, соответствующая ортогону с атрибутом CORNER, должна иметь точно один потомок (необязательно прямой), которому не приписан атрибут NOTAB и который покрывает все листья макета шапки строк.

Литература

I. Изотамм А., Дерево описания записи в системе РАМА. Труды ВЦ ТГУ, 1980, 43, 3-35.

2. Изотамм А., Представление записи в системе РАМА. Труды ВЦ ТГУ, 1982, 49, 26-41.

3. Изотамм А., Техника распознавания составных имен. Труды ВЦ ТГУ, 1980, 45, 3-20.

4. Изотамм А., Каазик Ю., Томбак М., Язык определения записи. Труды ВЦ ТГУ, 1978, 41, 7-64.

5. Пейал Я., Соо В., Томбак М., Язык вывода данных. Труды ВЦ ТГУ, 1982, 49, 42-61.

ИСПОЛЬЗОВАНИЕ РАСШИРЯЕМОГО ЯЗЫКА В СПТ

Я.Р. Пейал, В.К. Соо, М.О. Томбак

Развитие вычислительной техники требует оснащения новых ЭВМ программными средствами в краткие сроки. Эту цель можно достичь, если соблюдать условия мобильности и простой реализуемости программного обеспечения. Именно поэтому в последние годы широкое применение на микро-ЭВМ нашли расширяемые языки типа ФОРТ (см. [2]).

В настоящей статье рассматривается такая система построения трансляторов (СПТ), которая в качестве объектного языка использует язык ФОРТ (см. [3]). Сама система также написана на этом же языке.

Основная идея предлагаемого подхода заключается в том, что в описании исходного языка семантика разбивается на две части - перевод и собственно семантика. Семантикой при этом называется расширение языка ФОРТ всеми понятиями, имеющимися в исходном языке, а переводом - алгоритм перевода с исходного языка на (расширенный) ФОРТ. Такой подход позволяет использовать для задания алгоритма перевода довольно простой механизм - несколько модифицированный синтаксически управляемый (СУ-) перевод (см. [1]).

1. Синтаксически управляемый перевод

Схема СУ-перевода - это кортеж $(N, \Sigma, \Delta, P, S)$, где N - нетерминальный альфавит, $S \in N$ - фиксированный начальный символ (аксиома), Σ - входной альфавит, Δ - выходной альфавит, а P - множество правил перевода вида

$$A_o \longrightarrow x_o A_1 x_1 \ldots x_{p-1} A_p x_p, z_o B_1 z_1 \ldots z_{p-1} B_p z_p \qquad (1)$$

($x_i \in \Sigma^*$, $z_i \in \Delta^*$, $A_i \in N$, $B_i \in N$), где вектор $(B_1,\ldots,B_p)$ является некоторой перестановкой вектора $(A_1,\ldots,A_p)$.

Если $(B_1,\ldots,B_p) = (A_1,\ldots,A_p)$ во всех правилах перевода из P, то получаем схему т.н. простого СУ-перевода.

Схема СУ-перевода естественным образом определяет множество пар цепочек, выводимых из пары (S, S) (точное определение см. [1], стр. 249). Первые компоненты всех выводимых пар составляют входной язык L_1, а вторые компоненты - выходной язык L_2. Если (x,y) является выводимой парой, то цепочка y называется переводом цепочки x.

В нашем случае рассматривается перевод на язык ФОРТ, в котором большинство конструкций имеет постфиксный вид. Известно, что для перевода из инфиксной записи в постфиксную достаточно пользоваться такой простой СУ-схемой, в которой $z_o = \ldots = z_{p-1} = \Lambda$ (Λ - обозначение пустой строки). Нам придется рассматривать более общий случай, так как некоторые ФОРТ-конструкции отличаются от постфиксного вида, а также не все исходные языки строго инфиксны.

В дальнейшем будем использовать такую схему перевода, в которой все правила имеют вид

$$A_o \longrightarrow x_o A_1 x_1 A_2 \ldots x_{n-1} A_n x_n, u B_1 B_2 \ldots B_n v \qquad (2)$$

$(u,v \in \Delta^*)$. При помощи правил перевода такого вида можно задавать любой СУ-перевод, так как правило вида (1) можно заменить множеством правил вида (2):

$$A_o \to C_p A_p x_p,\ C_p B_p z_p$$
$$C_p \to C_{p-1} A_{p-1} x_{p-1},\ C_{p-1} B_{p-1} z_{p-1}$$
. . . . . .
$$C_2 \to x_o A_1 x_1,\ z_o B_1 z_1$$

Индукцией по длине вывода легко показать, что такая замена сохраняет переводящую мощность СУ-схемы.

В практике часто применяется понятие класса лексем, которое позволяет сократить описание исходного языка (одним метасимволом выражается целое множество конкретных лексем). Обычно выделяются классы "идентификатор", "константа" и т.п. Для нас существенно, чтобы представители таких классов сохранялись при переводе. Поэтому будем считать, что Σ и Δ содержат одинаковые классы лексем, а соответствующие метасимволы l могут ввестись в схему только при помощи правил вида

$$A \to l, l \qquad (3)$$

$(A \in N,\ l \in \Sigma \cap \Delta)$. Такое соглашение также не ограничивает общности СУ-схемы, но позволяет переставить представители классов лексем, если это нужно.

Пример 1. Рассмотрим следующую СУ-схему:

$N = \{$prog,id,const,dcls,ops,dcl,op,var,dim,exp,mor$\}$,

$\Sigma = \{$[I],[C],'VAR','ARRAY',':=','+','-','(',')',';'$\}$,

$\Delta = \{$[l],[C],'Ø','VARIABLE','ARRAY','!','+','-','@'$\}$,

S = prog и P состоит из правил:

```
1)  prog → dcls ';' ops          ,  dcls ops
2)  dcls → dcl                   ,  dcl
3)  dcls → dcls ';' dcl          ,  dcls dcl
4)  ops → op                     ,  op
5)  ops → ops ';' op             ,  ops op
6)  dcl → 'VAR' id               ,  'Ø' 'VARIABLE' id
7)  dcl → 'ARRAY' id '(' dim ')' ,  dim id
8)  id → [I]                     ,  [I]
9)  dim → const                  ,  const 'ARRAY'
10) const → [C]                  ,  [C]
11) op → var ':=' exp            ,  exp var '!'
12) var → id                     ,  id
13) var → id '(' exp ')'         ,  exp id
14) exp → mon                    ,  mon
15) exp → exp '+' mon            ,  exp mon '+'
16) exp → exp '-' mon            ,  exp mon '-'
17) mon → const                  ,  const
18) mon → var                    ,  var '@'
```

Здесь [I] обозначает класс идентификаторов, [C] - класс целых констант без знака, а апострофы трактуются как метасимволы, выделяющие ключевые слова и разделители.

При такой интерпретации программе

```
VAR A ; ARRAY B(6) ; A := 4 ; B(3) := A+2
```

соответствует перевод

```
Ø  VARIABLE  A
6  ARRAY  B
4  A  !
A  ●  2  +  3  B  !
```

Так как правила перевода имеют вид (2), то способ их задания может быть несколько упрощен. Для этого отбрасываем вторые компоненты правил и введем новые обозначения:

PRE = u,

POST = v,

PERM = $(k_1,\ldots,k_n)$ задает перестановку вектора $(1,\ldots,n)$, при помощи которой из вектора $(A_1,\ldots,A_n)$ получен вектор $(B_1,\ldots,B_n) = (A_{k_1},\ldots,A_{k_n})$.

По этой информации вторые компоненты правил вида (2) могут быть восстановлены. Для сокращения записи при этом считаем, что по умолчанию PRE = Λ, POST = Λ и PERM = $(1,\ldots,n)$. Вторые компоненты правил вида (3) также можно отбрасывать.

Пример 2. Для задания СУ-схемы примера 1 достаточно дополнить первые компоненты правил перевода (которые образуют КС-грамматику исходного языка) следующей информацией:

```
6)  PRE = 'Ø'  'VARIABLE'
7)  PERM = (2,1)
9)  POST = 'ARRAY'
11) POST = '!' , PERM = (2,1)
13) PERM = (2,1)
15) POST = '+'
16) POST = '-'
18) POST = '●'
```

Указанный вид схемы СУ-перевода позволяет использовать дерево вывода исходной программы (см. [1], стр. 163) для выполнения перевода. Чтобы обеспечить перевод представителей классов лексем, определяем еще PRE и POST для терминальных вершин дерева вывода следующим образом. В каждой такой вершине PRE = Λ; если вершине соответствует представитель класса лексем (идентификатор, константа и т.п.), то принимается POST = текст лексемы, в противном случае POST = Λ.

Для внутренних вершин PRE, POST и PERM уже определены, так как каждой внутренней вершине соответствует некоторое правило перевода.

Приведем теперь алгоритм СУ-перевода на дереве вывода программы (T является любой вершиной дерева вывода).

Алгоритм GENTEXT(T):

G1. Выдать текст PRE(T).

G2. Если T имеет прямые нетерминальные потомки $T_1,\ldots,T_n$, то переставить их согласно PERM(T). Вместе с вершинами переставляются и поддеревья, корнями которых они служат.

G3. Если T имеет прямые потомки $V_1,\ldots,V_m$, то для каждого $i = 1,\ldots,m$ выполнить GENTEXT(V_i).

G4. Выдать текст POST(T).

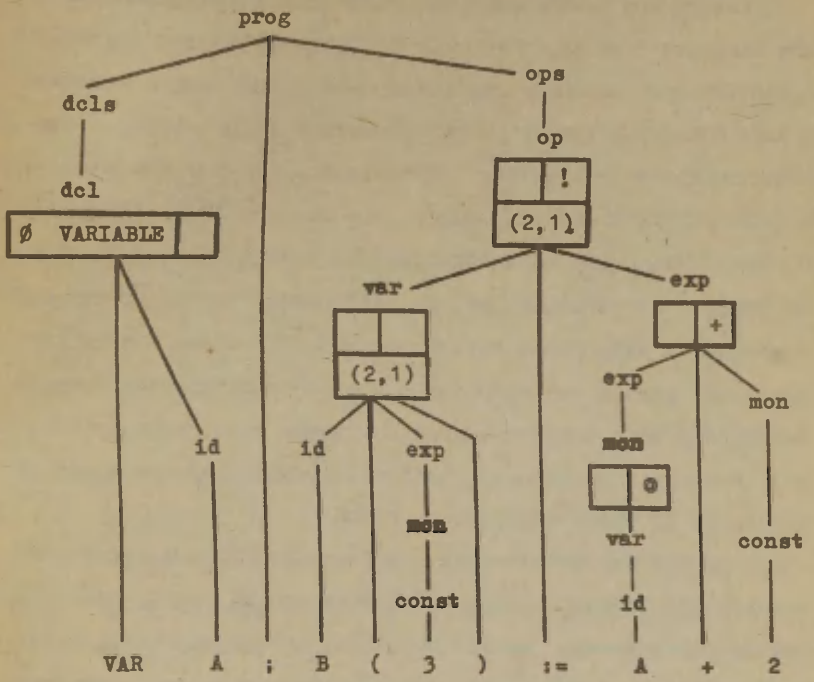
G5. Выход.

Выполнив этот алгоритм на корне дерева вывода получим в выходном потоке перевод входной цепочки.

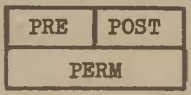
Пример 3. Согласно СУ-схеме из примеров 1 и 2 исходной программе

VAR A ; B(3) := A+2

соответствует дерево вывода следующей структуры:



Здесь PRE, POST и PERM, если они определены не по умолчанию, изображены в прямоугольниках следующим образом:



(если перестановка вершин не требуется, то PERM-часть прямоугольника опускается).

Выполнив GENTEXT(prog) получим текст

Ø VARIABLE A A @ 2 + 3 B !

который и является переводом исходной программы.

2. Модификация метода перевода

Главную трудность перевода составляют структурные различия входного и выходного языков. СУ-схема является удобным средством для перевода отдельных конструкций языков программирования. Но выбранный нами объектный язык - ФОРТ - налагает некоторые более общие требования, которым СУ-схема не всегда удовлетворяет. Например, все объекты языка ФОРТ должны быть описаны до их использования. Если в реализуемом языке такого ограничения нет, то необходимо сначала перевести описания, а лишь потом остальную часть. СУ-схема в этом случае может оказаться громоздкой и плохо поддающейся методам синтаксического анализа (или такой схемы не существует вообще). В целях преодоления подобных трудностей нам пришлось несколько обобщить алгоритм перевода.

В результате выполнения алгоритма GENTEXT над некоторой внутренней вершиной V дерева вывода мы получим в выходном потоке определенную часть перевода. Обозначаем такую часть через $\tau(V)$. Соответствующий отрезок входной цепочки образует крону поддерева с корнем V.

Вышеупомянутую проблему о перестановке описаний можно поставить в следующем виде: перевести цепочку

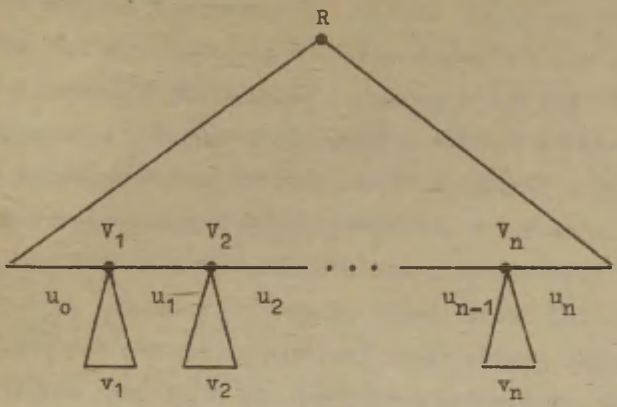
$$u_0 v_1 u_1 v_2 u_2 \cdots u_{n-1} v_n u_n$$

в цепочку

$$v_1' v_2' \cdots v_n' u',$$

где $u_i \in \Sigma^*$, $v_j \in \Sigma^*$ является кроной поддерева, корень которого обозначаем через V_j, $v_j' = \tau(V_j) \in \Delta^*$, а $u' \in \Delta^*$ является оставшейся частью перевода.

Дерево вывода рассматриваемой входной цепочки можно схематически представить в виде:



Одним путем для получения желаемого результата является следующий порядок действий:

1) для каждого j = 1,...,n выполнить GENTEXT(V_j);

2) удалить из дерева вывода поддеревья с корнями $V_1,...,V_n$;

3) выполнить GENTEXT(R), где R - корень дерева вывода.

Чтобы получить более общий алгоритм, будем предполагать, что корни поддеревьев, подлежащих переводу раньше других, помечены некоторым специальным образом. Такие метки при этом можно приписать к правилам грамматики исходного языка, так как любой внутренней вершине дерева вывода соответствует некоторое правило грамматики. Указанные метки будем в дальнейшем называть режимами и обозначать через REG.

В случае проблемы о перестановке описаний вершины, снабженные режимами друг другу не подчинялись и обрабатывались слева направо. Чтобы избавиться от возможных ограничений при присваивании режимов правилам грамматики, следует определить

порядок обработки для любого расположения вершин, имеющих режимы. Такой порядок уже зафиксирован косвенно: если считать, что определен REG(R), то необходимый порядок вытекает из того, что R обрабатывается после обработки потомков. Поэтому обходим любую вершину с режимом при обработке в концевом порядке - сначала потомки слева направо, а потом предок.

В целях получения более гибкого метода перевода, допускаем в вершинах с режимами наряду с применением алгоритма GENTEXT возможность некоторых дополнительных действий. Таким образом, REG может иметь разные значения для разных правил грамматики. Каждое такое значение определяет некоторый набор действий, подлежащих выполнению при обработке соответствующей вершины дерева вывода.

Приведем теперь алгоритм, управляющий выполнением действий, определяемых режимами (W является любой вершиной дерева вывода).

Алгоритм SEARCH(W):

S1. Если W имеет прямые потомки $W_1,\ldots,W_m$, то для каждого $i = 1,\ldots,m$ выполнить SEARCH(W_1).

S2. Если в вершине W определен режим, то:

S2.1. выполнить действия, указанные в REG(W);

S2.2. удалить поддеревья с корнями $W_1,\ldots,W_m$, но не считать вершину W в дальнейшем терминальной вершиной (при выполнении шага G2 в алгоритме GENTEXT).

S3. Выход.

Исходной вершиной для алгоритма SEARCH является корень дерева вывода.

Опираясь на модифицированный таким образом алгоритм перевода, переходим к рассмотрению возможных режимов. В данной статье опишем лишь два режима, которые уже применялись при разработке конкретных трансляторов, но в принципе количество режимов не ограничено.

Режим TRANS в вершине W означает, что в выходной поток поступает перевод поддерева с корнем W. Именно этот режим используется для перестановки описаний.

Режим TRANS(W):

T1. Выполнить GENTEXT(W).

T2. PRE(W) := Λ; POST(W) := Λ.

ФОРТ-программы обычно представляются как т.н. ":"-определения (colon definitions). Поэтому введен режим COLON, который обеспечит перевод операторов в такие определения. Если REG(W) = COLON(W), то ς(W) оформляется как отдельное ":"-определение, для которого генерируется специальное системное имя. Чтобы в дальнейшем пользоваться этим определением, его имя хранится как POST(W).

Режим COLON(W):

C1. Генерировать новое системное имя.

C2. Выдать строку, состоящую из ':' и системного имени.

C3. Выполнить GENTEXT(W).

C4. PRE(W) := Λ; POST(W) := системное имя.

Предполагается, что строка ';', закрывающая ":"-определение, выдается алгоритмом GENTEXT.

	REG	PRE	POST	PERM
prog ⟶ unit	TRANS			
unit ⟶ seq	COLON		;	
seq ⟶ stm				
seq ⟶ seq ';' stm				
stm ⟶ dcl	TRANS			
stm ⟶ var ':=' exp			↕	(2,1)
stm ⟶ header body		BEGIN	REPEAT	
dcl ⟶ 'VAR' id		VAR		
dcl ⟶ 'DIM' id '(' dim ')'				(2,1)
id ⟶ [I]				
dim ⟶ const			ARRAY	
const ⟶ [C]				
var ⟶ id				
var ⟶ id '(' exp ')'				(2,1)
exp ⟶ mon				
exp ⟶ exp '+' mon			+	
exp ⟶ exp '-' mon			-	
mon ⟶ const				
mon ⟶ var			@	
mon ⟶ '(' exp ')'				
header ⟶ 'WHILE' cond			WHILE	
body ⟶ 'DO' seq 'OD'				
cond ⟶ exp '<' exp			<	

3. Пример перевода

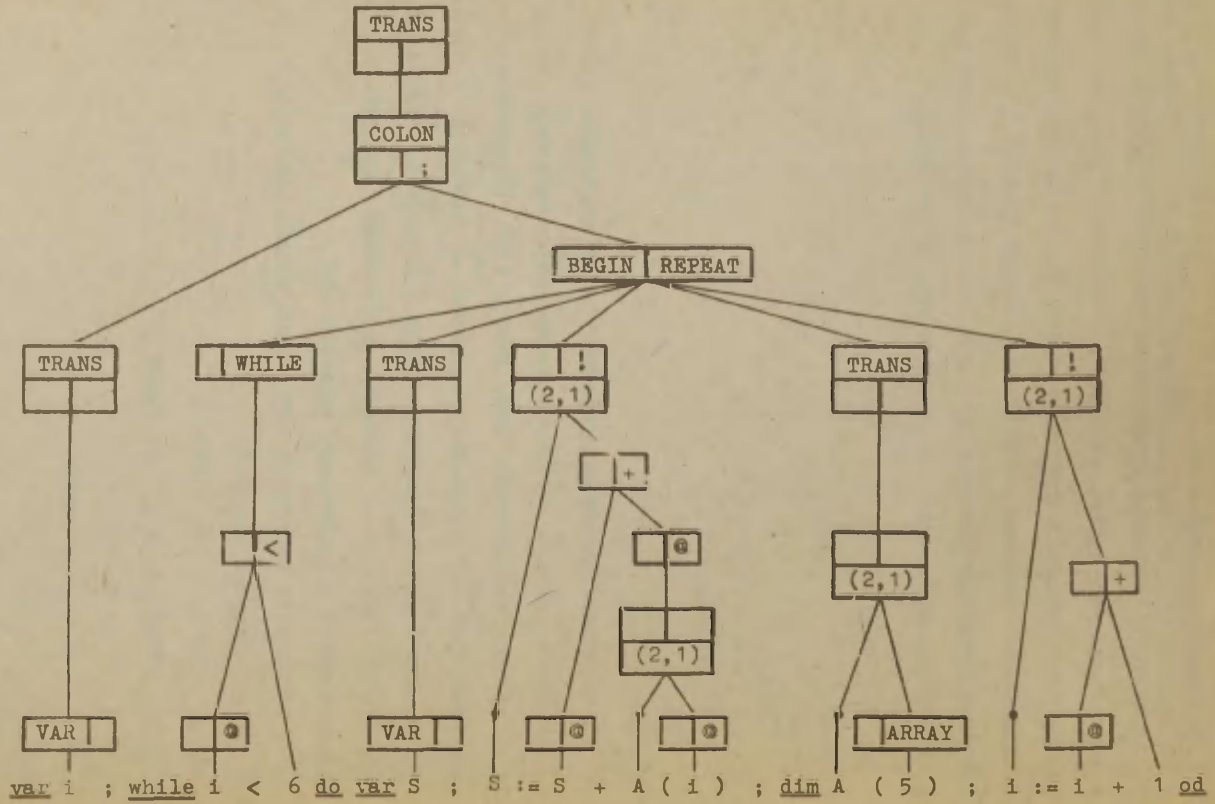
В качестве примера рассмотрим небольшой исходный язык, в котором имеются целые константы, переменные, массивы, операторы присваивания и операторы цикла. На предыдущей странице приведено описание этого языка (левый столбец), а также описание перевода (столбцы REG, PRE, POST и PERM). Иллюстрируем перевод при помощи следующей исходной программы:

```
var i ;
while i<6
  do
    var S ;
    S := S + A(i) ;
    dim A(5) ;
    i := i + 1
  od
```

На следующей странице изображено синтаксическое дерево, полученное из дерева вывода удалением всех ненужных вершин (т.е. вершин, для которых PRE, POST и PERM определяются по умолчанию, REG не определен и которые не участвуют в перестановках). В прямоугольниках, как и в примере 3, размещены PRE, POST и PERM, но в вершинах с режимами верхняя часть прямоугольника содержит еще значение REG.

В результате выполнения алгоритма SEARCH над корнем этого дерева, получим перевод:

```
 VAR I   VAR S     5  ARRAY  A
:  SYS1  BEGIN  I  @  6  <  WHILE
S  @  I  @  A  @  +  S  !
I  @  1   +   I  !  REPEAT  ;
SYS1
```


TRANS
COLON
;
BEGIN
REPEAT
TRANS
WHILE
TRANS
!
(2,1)
TRANS
!
(2,1)
+
<
@
(2,1)
(2,1)
+
VAR
@
VAR
@
@
ARRAY
@
var i ; while i < 6 do var S ; S := S + A (i) ; dim A (5) ; i := i + 1 od

В этой ФОРТ-программе все объекты определены до того, как они будут использованы. Также выполняется требование, по которому конструкция

BEGIN...WHILE...REPEAT

должна находиться внутри ":"-определения. В данной программе этому определению дается сгенерированное системное имя SYS1. Появление имени SYS1 в конце перевода (в языке ФОРТ это означает использование объекта SYS1) гарантирует запуск программы. Именно поэтому в корне дерева должен быть установлен режим TRANS.

* * *

Заключение

В данной статье главное внимание уделялось вопросам построения подходящего метода перевода. На основе построенного таким образом метода нами составлена система построения трансляторов (СПТ ТАРТУ), в которой расширяемый язык ФОРТ одновременно выступает как

1) объектный (промежуточный) язык,

2) язык определения семантики,

3) инструментальный язык.

Если учитывать, что язык ФОРТ относительно просто реализуется на микро- и мини-ЭВМ, то такой подход облегчает перенос системы на новые модели ЭВМ. При этом немаловажно, что СПТ ТАРТУ составлена с учетом работы в условиях ограниченной оперативной памяти.

В рамках данной статьи не были затронуты многие проблемы, которые возникают при реализации конкретных языков программирования - проверка контекстных условий, организация интерактивной работы, описание типов данных и т.п. Оказывается, что в пределах описанного здесь подхода подобные проблемы большей частью разрешимы.

Литература

I. Ахо А., Ульман Дж., Теория синтаксического анализа, перевода и компиляции, т.I., "Мир", Москва, 1978.

2. Loeliger R.G., Threaded interpretive languages, "BYTE Books", 1981.

3. Ragsdale W.F., Pig-FORTH installation manual, glossary, model, San Carlos, August 1980.

СХЕМАТИЧЕСКАЯ ФОРМА ЗАПИСИ СТРУКТУР

Ю.К.Кихо

1. Введение

Используемые при общении с ЭВМ знаковые системы записи программ и структур данных разделяются на два принципиально разных класса.

Наиболее распространенной является чисто текстовая форма записи, в которой объект представляется в виде последовательности символов. При этом, для указания структуры объекта используются как круглые, так и словесные скобки (IF..FI, DO..OD, BEGIN..END и т.п.), а также возможность записи текста лесенкой.

Этот способ с середины 50-х годов бесконкурентно господствовал до самого последнего времени. Лишь сравнительно недавно разработаны новые методы и соответствующие системы графического, стилизованно-чертежного представления структур [1,2]. В этих системах более элементарные компоненты объекта (например, формулы, простые операторы и их последовательности) представляются в традиционной текстовой форме. Для указания же их композиции в более сложные структуры применяются графические средства. Преимущество графических знаковых систем перед чисто текстовыми способами заключается в

более продуктивном использовании мощного зрительного аппарата и ассоциативного мышления человека.

Метод Р-схем [I], появившийся исторически первым из методов графической записи алгоритмов и программ, основан на представлении объектов в виде нагруженных по дугам графов (Р-схем). В таких графах у дуг записываются элементарные текстовые компоненты структуры.

Разработанный в 1983/84 годах метод схематического программирования [2] заключается в представлении структур в виде вертикальных схем. Схема изображается как охваченный слева особой скобкой набор элементарных компонент и схем, содержащий также специальные символы выхода в виде т.н. выходных стрелок.

Общей для указанных графических систем является возможность наглядного представления не только алгоритмов и программ, но и структур данных, синтаксических диаграмм и т.п. Эти методы могут служить основой также и для сквозной автоматизации работы программистов.

Главным преимуществом метода схематического программирования перед Р-технологией является его крайняя простота и близость к традиционным методам программирования. По существу, добавляются только правила размещения компонент структур в вертикальном направлении, при этом отпадает необходимость применения ключевых слов в роли скобок.

На рис. I и 2 приведены различные представления одной и той же простой программы (из [I]).

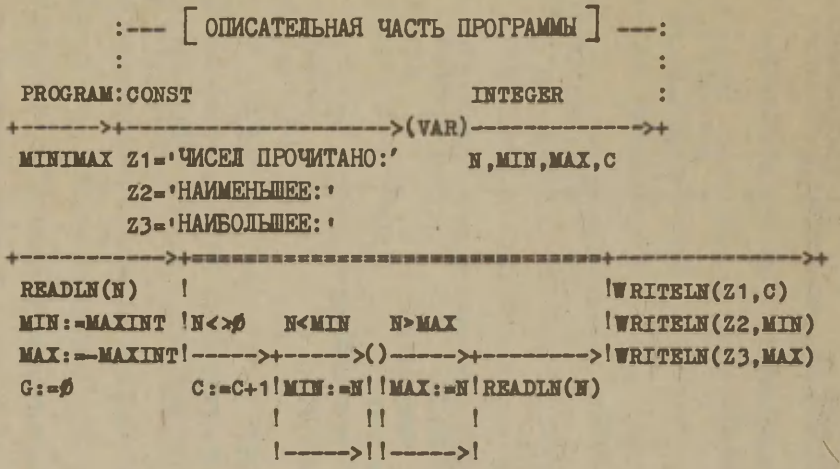


Рис. I. Р-схема программы MINIMAX

a)
```
PROGRAM MINIMAX;
(*ОПИСАТЕЛЬНАЯ ЧАСТЬ*)
CONST
   Z1='ЧИСЕЛ ПРОЧИТАНО:';
   Z2='НАИМЕНЬШЕЕ:';
   Z3='НАИБОЛЬШЕЕ:';
VAR
   N,MIN,MAX,C:INTEGER;
(*КОНЕЦ ОПИСАНИЙ*)
BEGIN
   READLN(N);
   MIN:=MAXINT;
   MAX:=-MAXINT;
   C:=Ø;
   WHILE N<>Ø DO
      BEGIN
         C:=C+1;
         IF N<MIN
            THEN MIN:=N;
         IF N>MAX
            THEN MAX:=N;
         READLN(N)
      END;
   WRITELN(Z1,C);
   WRITELN(Z2,MIN);
   WRITELN(Z3,MAX)
END.
```

b)
```
PROGRAM MINIMAX;
┌ ОПИСАТЕЛЬНАЯ ЧАСТЬ
CONST
 Z1='ЧИСЕЛ ПРОЧИТАНО:';
 Z2='НАИМЕНЬШЕЕ:';
 Z3='НАИБОЛЬШЕЕ:';
VAR
  N,MIN,MAX,C:INTEGER;
└
┌
  MIN:=MAXINT;
  MAX:=-MAXINT;
  C:=Ø;
  ╒
    READLN(N);
  N<>Ø?
    C:=C+1;
    ┌
    │N<MIN?
    │  MIN:=N;
    └
    ┌
    │N>MAX?
    │  MAX:=N;
    └
  ╘
  WRITELN(Z1,C);
  WRITELN(Z2,MIN);
  WRITELN(Z3,MAX)
└
```

Рис. 2. Чисто текстовая запись (на Паскале) и схематическое представление (на Е-Паскале) программы MINIMAX.

В настоящей работе идея схематического программирования развита несколько дальше. Исходя из определения более общего понятия - понятия Е-схемы - заполняются некоторые пробелы, существующие в исходном описании схематической программы [2]. Во-первых, указывается возможность рассматривать Е-схему как инструкцию множественного выбора (оператора типа CASE). Во-вторых, благодаря более общему определению понятия цикла, уточняется семантика выполнения неограниченного цикла, причем охватывается и понятие цикла типа WHILE .

Подробности созданных конкретных систем схематического программирования (на базе языков Бейсик, Фортран, ПЛ/I, Паскаль и Ассемблер) здесь не приводятся, но некоторые особенности схематизированных языков Е-Фортран и Е-Паскаль в примерах указаны.

2. Понятие Е-схемы

2.I. П о н я т и е Е-с х е м ы соответствует общему методу получения сложных объектов из более элементарных конструктивных элементов. Согласно этому методу, любой составной объект представляется как множество последовательностей, составленных из более простых объектов. Множество всех исходных, "атомарных" объектов называется базисом класса Е-схем.

Если А - некоторое множество, то через A^* обозначим множество всех последовательностей из элементов А (в том числе пустая последовательность и кортежи), через $A^{\sqsubset}$ - множество всех непустых конечных подмножеств множества A^*.

Если фиксировано некоторое базисное множество B, то классом E-схем в базисе B называется множество

$$\mathcal{E} = \bigcup_{i=1}^{\infty} E_i ,$$

где $E_i = \left(\bigcup_{j=0}^{i-1} E_j\right)^{\Gamma}$, причем $E_0 = B$.

Элементы множества $\mathcal{E}$ называют E-схемами в базисе B.

Заметим, что элементы базиса не являются E-схемами.

Сама E-схема не задает конкретный объект, например, программу. Она лишь описывает строение объекта или, точнее, строение множества объектов, получающихся из E-схем в результате задания различных интерпретаций. В частности, E-схему можно интерпретировать как схему программы [3], структуру данных, синтаксическую диаграмму и т.п. При этом, как правило, множество интерпретируется "дизъюнктивно" - как совокупность возможных вариантов, а последовательность "конъюнктивно"-как список обязательных составных частей, указанных в требуемом порядке.

Рассмотрим, например, как синтаксическое понятие "инструкция_присваивания" (на языке ФОРТРАН 77) описывается в виде E-схемы, т.е. как иерархия последовательностей и множеств.

Пусть в базисе B содержатся следующие элементы: имя_переменной, имя_элемента_массива, имя_подцепочки, выражение, метка, =, ASSIGN , TO. Тогда "инструкция_присваивания"-это множество из двух вариантов:

- последовательность из трех членов: левая_сторона, =, выражение,
- последовательность из четырех членов: ASSIGN , метка, TO, имя_переменное,

а "левая_сторона",в свою очередь, является множеством из трех

вариантов (каждый из которых одночленная последовательность):

- имя_переменной,
- имя_элемента_массива,
- имя_подцепочки.

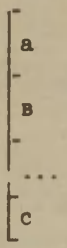
В целях практического применения Е-схем необходимо фиксировать некоторую конкретную форму их записи - конкретный синтаксис. Лишь тогда на основе Е-схем возможно разработать конкретный язык (например, язык программирования), интерпретировать базис и доопределить семантику Е-схем (например, осмыслить понятие "выполнение Е-схемы").

2.2. К о н к р е т н ы й с и н т а к с и с. В практике, особенно при представлении Е-схемами структур программ, удобнее всего придерживаться следующей вертикальной формой их записи.

Последовательность a_1, a_2, ... записывается в виде

a_1
a_2
... ,

а множество из элементов a, в, ... , с в виде



Охватывающая элементы множества вертикальная скобка называется линией уровня.

Пустая последовательность обозначения не имеет. Е-схема,

которая состоит из пустой последовательности (пустая E-схема) обозначается через [.

Заметим, что строго вертикальная форма не подходит одинаково хорошо для всех применений E-схем (см. рис. 3а). Иногда, особенно при изображении структуры текстовых "горизонтальных" конструкций, целесообразно предусматривать горизонтальное расположение некоторых последовательностей (см. рис. 3 b).

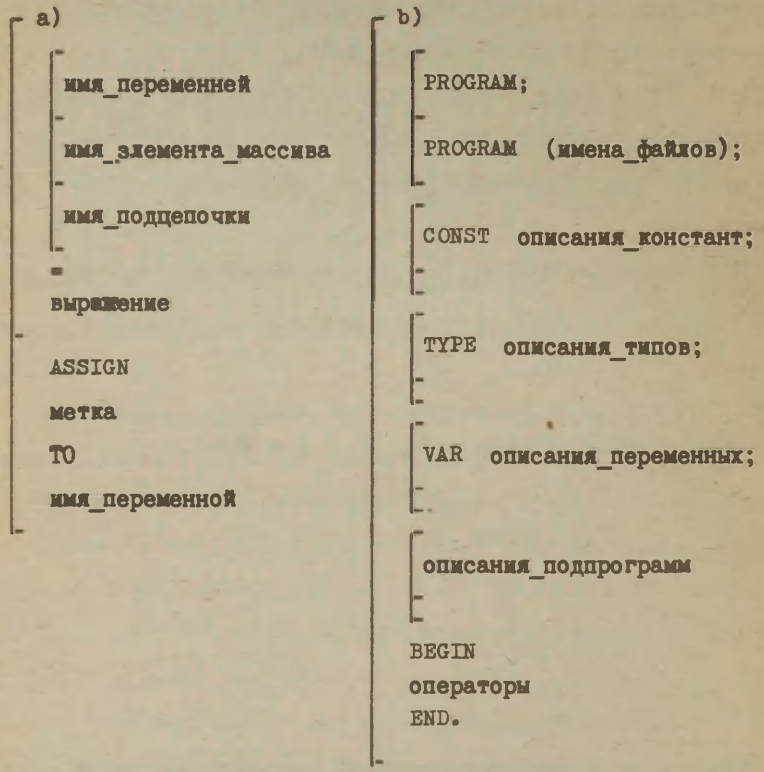


Рис. 3. Схематическое представление понятий "инструкция_присваивания" (a) и "программы" (b) в Фортране 77 и Паскале соответственно.

Представление объекта в виде Е-схемы вышеуказанной формы называется его схематическим представлением или Е-записью, или Е-наброском.

Необходимые в конкретных случаях атрибуты, такие как имя, спецпризнаки, комментарии и т.д.,принято записывать в начале схемы, в строке за началом (⌐) линии уровня.Кроме того,комментарии допускается писать ещё в строке за ограничителем (⊢) элементов и концом (⊣) линии уровня.

Например, в Е-Фортране спецпризнак - следующий за началом линии уровня символ, а в Е-Паскале - следующее за началом слово (до пробела).Соответственно, в Е-Фортране имя схемы может быть записано со второго символа (после ⌐), а в Е-Паскале - после первого пробела.

Отметим, что вместе с возможностью именовать (под)схемы привлекается круг общих вопросов, связанных со способом представления структур в виде именованных записей,вложенных друг в друге. Это,во-первых,проблема изображения структур "по частям",когда часть структуры изображается отдельно,а в "главной" структуре вместо этой части указывается лишь её имя,во-вторых, рекурсивная вложенность структур, когда в качестве имени части указывается имя "главной" структуры.Границы применения таких средств устанавливаются отдельно в каждой системе реализации.

2.3. П р и м е р ы Е-с х е м. Е-схему в пустом базисе можно интерпретировать как И-ИЛИ структуру (см. рис. 4).

На рис.5 приведены примеры Е-схем в одноэлементном базисе $B = \{+\}$ (заметим,что Е-схема на рис.4 является Е-схемой и в этом базисе). На рис.6 рекурсивной Е-схемой дано синтаксическое описание идентификатора (в базисе $B = \{$буква,цифра$\}$).

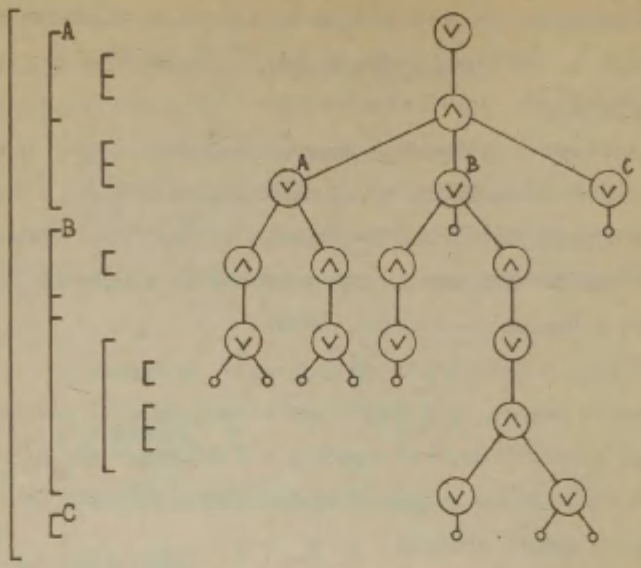


Рис. 4. E-схема в пустом базисе как изображение И-ИЛИ дерева (A,B,C - имена подструктур).

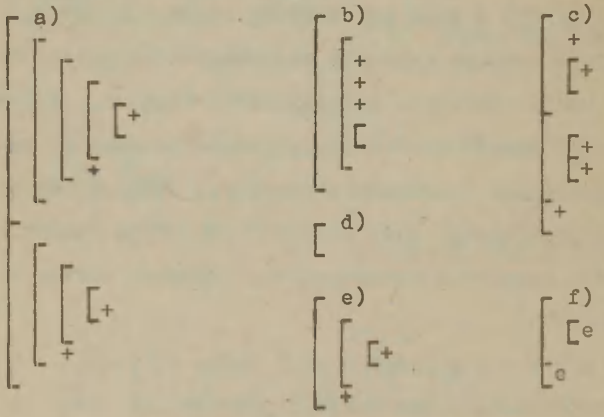


Рис. 5. Примеры E-схем (a,b,c,d,e) в базисе {+} и запись E-схемы a по частям (f).

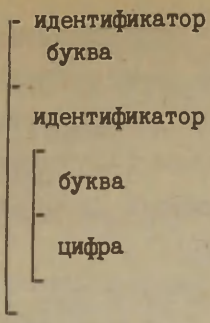


Рис. 6. Синтаксическое описание идентификатора.

Использование Е-схемы в целях структуризации описаний продемонстрировано на рис. 7.

```
TYPE PERSON =
┌RECORD
│  NAME:
│  PACKED ARRAY [ 1..15 ] OF CHAR;
│  BD:
│  ┌RECORD
│  │  DAY: 1..31;
│  │  MONTH: 1..12;
│  │  YEAR: 1950..1990
│  └
└
```

Рис. 7. Фрагмент описания типа данных на Е-Паскале.

2.4. Дополнительные понятия.

Моносхемой называется Е-схема мощности I, т.е. схема вида [C, где C- последовательность.

Полисхемой или множественной схемой называется Е-схема мощности свыше I.

E-схема называется непосредственной (или I-ой) охватывающей схемой для членов своих элементов-последовательностей.

Непосредственная охватывающая схема схемы, являющейся, в свою очередь, (i-1)-ой (i>1) охватывающей схемой для некоторого члена, называется i -ой охватывающей схемой для этого члена (см.рис.8).

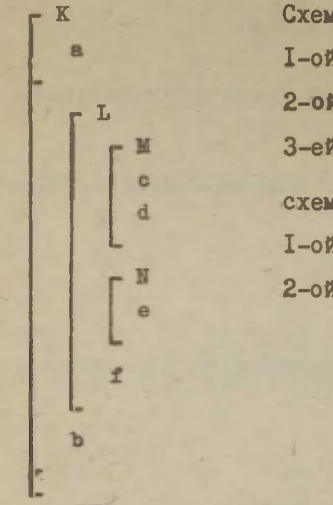


Рис. 8. Охватывающие схемы.

3. E-схемы структур управления

3.I. Рассмотрим применение E-схем для записи алгоритмов и программ. В этом случае естественно предполагать, что базису принадлежат преобразователи и распознаватели состояния [4]. В настоящем первые из них называются элементарными инструкциями и другие условиями. Преследуя идею схематического программирования, дополним базис ещё элементами третьего типа - терминаторами.

Итак, базис класса схематических программ состоит из трех непересекающихся не более счетных множеств символов:

(I) множество ELIN операторных символов, называемых элементарными инструкциями;

(2) множество COND предикатных символов, называемых условиями;

(3) множество EXIT функциональных символов, называемых терминаторами.

Типичной интерпретацией этого базиса является сопоставление элементарным инструкциям операторов (как описания, так и преобразования) из некоторого текстового языка программирования, т.н. базового языка (БЯ). Условиям тогда сопоставляются логические выражения на том же БЯ и терминаторам - арифметические выражения (в реализованных пока системах, также как и в приводимых ниже примерах, терминатору сопоставляется натуральная константа). Связанное с терминатором число называется его протяжением.

Пусть B_0 = ELIN $\cup$ COND $\cup$ EXIT.

Инструкциями в базисе B_0 называются элементы множества ELIN $\cup \mathcal{E}$, где $\mathcal{E}$ - класс Е-схем в базисе B_0. Условно-последовательной инструкцией (УП-инструкцией) называется моносхема в базисе B_0. Множественная схема в базисе B_0 называется множественной инструкцией. Схематической программой называется конечная непустая последовательность инструкций.

3.2. К о н к р е т н ы й с и н т а к с и с Е-схем в B_0. Форма записи элементарных инструкций и условий существенно опирается на БЯ. Незначительные изменения указываются конкретно для каждой системы реализации. Например, в Е-Фортране

можно метку указывать только для оператора описания формата, за словом FORMAT.

Как правило, во всех реализациях запись условия заканчивается вопросительным знаком.

Форма терминатора независима от БЯ.Терминатор с постоянным протяжением n записывается в виде ←n , при этом n называется также длиной стрелки. Такая стрелка может быть продлена до линии уровня n -ой охватывающей схемы; в этом случае длина стрелки может быть опущена.

3.3. В ы п о л н е н и е схематической программы происходит последовательно, инструкция за инструкцией, в порядке их расположения в последовательности. Это значит, что выполнение программы начинается с выполнения первой инструкции, а выполнение i -ой инструкции (i>1) начинается тогда и только тогда, когда завершено выполнение (i-1)-ой инструкции. Выполнение схематической программы завершено, когда завершено выполнение последней инструкции.

Выполнение элементарной инструкции происходит согласно конкретной интерпретации операторного символа.

Выполнение УП-инструкции имеет общую направленность сверху вниз. Но главной особенностью, по сравнению со схематической программой, является то, что (I) при выполнении некоторые члены могут пропускаться и (2) выполнение данной УП-инструкции может привести к завершению некоторой охватывающей её схемы. Такие "скачки" и "выходы" регулируются при помощи условий и терминаторов.

Более точно выполнение УП-инструкции $\left[\begin{smallmatrix}5\\C\end{smallmatrix}\right.$,где C - после-

довательность, описывается следующим образом.

Если C - пустая последовательность, то выполнение $\diamond$ завершено, иначе начинается выполнение её первого члена.

Пусть c_i - очередной выполняемый член в C ($i>1$).

Если c_i - инструкция, то c_i выполняется. После завершения выполнения c_i начинается выполнение члена c_{i+1}, но если c_i был последним членом C, то выполнение $\diamond$ завершено.

Если c_i - условие, то проверяется условие c_i, т.е. вычисляется логическое значение (0 или I). При ответе "I" выполнение c_i завершено, и начинается выполнение члена c_{i+1} (но если c_i был последним членом в C, то выполнение $\diamond$ завершено). При ответе "0" выполнение c_i завершено, и начинается выполнение члена c_k ($k>i$), причем c_{k-1} - терминатор и $c_{i+1}, \dots, c_{k-2}$ - не терминаторы (другими словами, пропускаются члены C до ближайшего терминатора включительно). Если такого c_k нет, то выполнение схемы завершено.

Если c_i - терминатор, то завершается выполнение n -ой охватывающей его схемы, где n - протяжение терминатора c_i. Если такой схемы нет, то не завершается ни одна инструкция.

Выполнение множественной инструкции происходит всегда независимо от порядка, в котором элементы этой схемы приведены в конкретной программе. В принципе, множественная инструкция выполняется недетерминированно, причем может выполняться нуль, один или несколько её элементов, возможно даже параллельно.

Пока в существующих системах схематического программирования множественная инструкция применяется только в смысле множественного выбора (оператора типа CASE), определим семантику только для множественной схемы соотвествующего част-

ного вида (см. рис. 9).

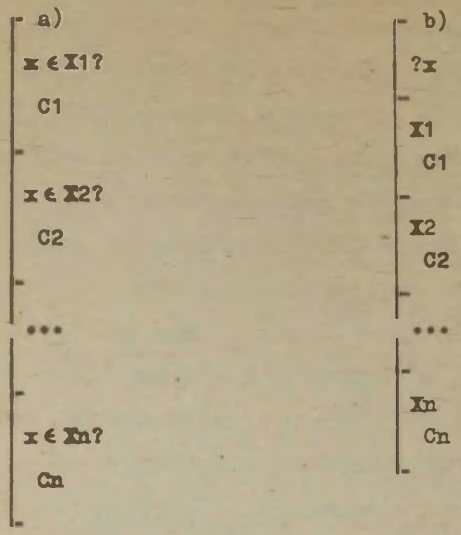


Рис. 9. Е-схема множественного выбора (а) и её сокращенная запись (b).

Такая схема выполняется как УП-инструкция вида $[C_i$, где i такое (одно из таких), что $x \in X_i$ в момент начала выполнения множественной инструкции.

На рис. 10 приведены конкретные примеры инструкций множественного выбора.

4. Понятие цикла

4.1. Понятие Е-схемы служит основой для более общего подхода к понятию цикла. Оказывается возможным, а также совсем естественным для основных интерпретаций, определить синтаксическое понятие цикла как одну разновидность Е-схемы.

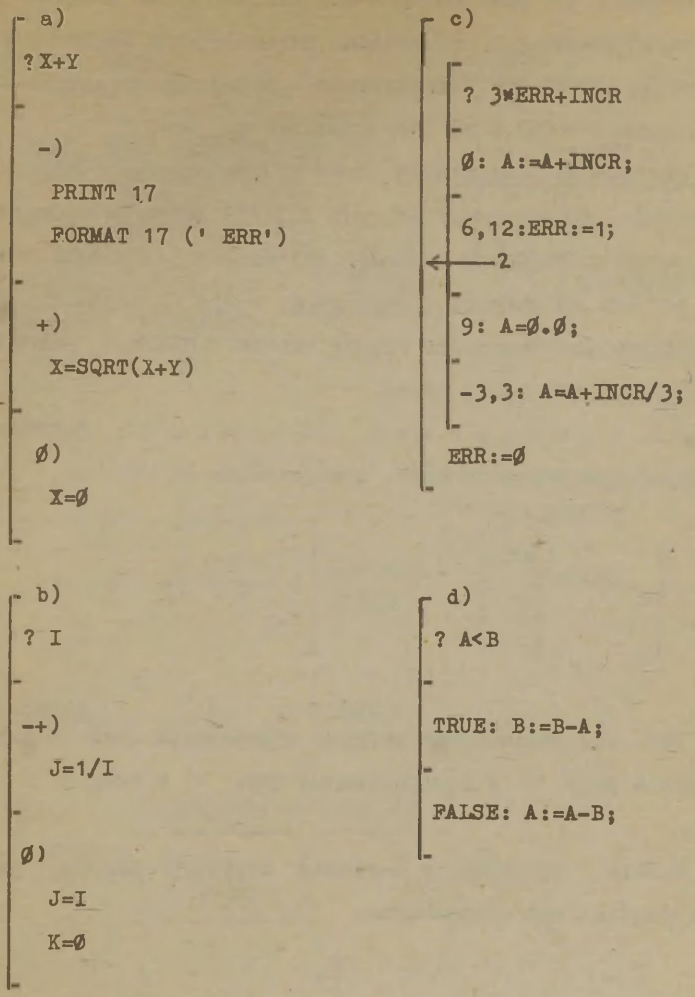


Рис. I0. Примеры инструкций множественного выбора на Е-Фортране (a,b) и Е-Паскале (c,d).

Это значит, что теоретически нет надобности во введении разных конструкций для обозначения циклических и ациклических структур, также как пользоваться принципиально разными методами исследования этих схем управления.

Циклической моносхемой (или просто циклом) называется моносхема, единственный элемент которой является периодической последовательностью. Если эта последовательность конечна, то цикл называется огражденным, в других случаях неогражденным. Повторяющаяся группа членов в цикле называется содержанием или телом цикла.

4.2. К о н к р е т н ы й с и н т а к с и с. Допускается пользоваться обозначениями, приведенными на рис. II.

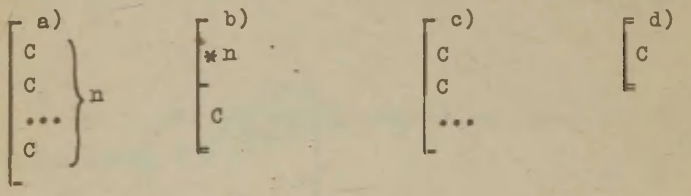
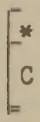


Рис. II. Обозначения циклов: огражденный цикл a записывается в виде b и неогражденный цикл c в виде d .

Иногда, например в Е-схемах структур данных, удобно пользоваться ещё обозначением



в смысле (произвольного представителя) класса всех огражденных циклов с содержанием C (см. рис. I2).

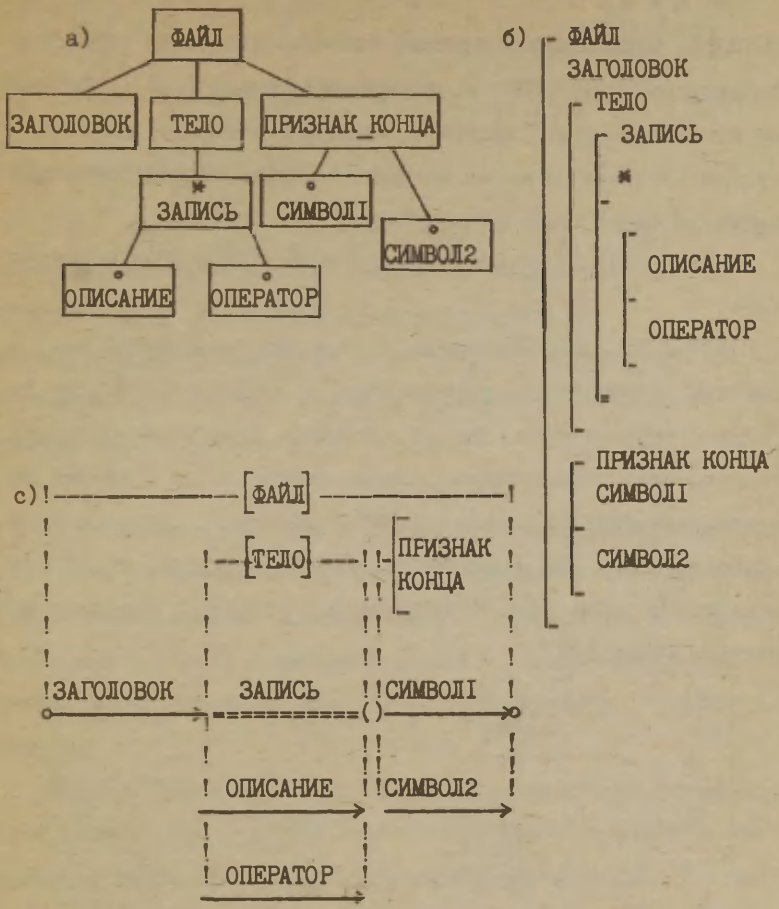


Рис. I2. Описание структур файла а) в диаграммах Джексона, б) Е-схемой, с) Р-схемой.

4.3. Ц и к л ы в с х е м а х у п р а в л е н и я. В работе [2] были введены понятия ациклической схемы (АЦС) неограниченного цикла (НОЦ) и ограниченного цикла (ОЦ), с указанием конкретной формы записи и семантики отдельно для каждого из них. Определенные на основе Е-схемы соответствующие понятия хорошо с ними согласуются.

Понятие АЦС [2] точно совпадает с понятием УП-инструкции (п.3).

Понятия НОЦ тоже одинаковы, с той лишь разницей, что в настоящем устранена неопределенность в выполнении НОЦ, не содержащего терминаторов. Именно, учитывая определение НОЦ(как периодической УП-инструкции), легко увидеть, что в случае обнаружения условия со значением "0" в НОЦ без терминаторов выполнение этого НОЦ завершается. Другими словами, в НОЦ без терминаторов любое условие оказывается условием продолжения цикла (см. рис. I3).

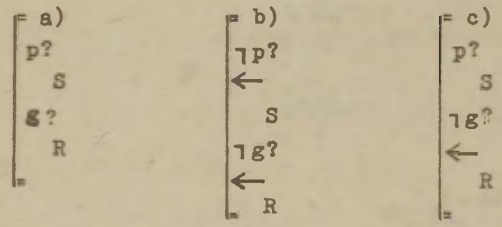


Рис. I3. Две функционально эквивалентные НОЦ (a и b) и неэквивалентная с ними схема (c).

Что касается понятия ОЦ с параметром [2], то его можно считать частным случаем определенного в настоящем ОЦ (см. рис. I4a).

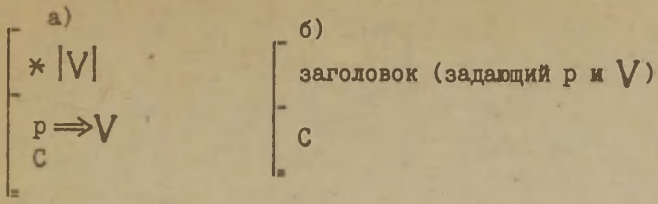


Рис. I4. а) Семантика огражденного цикла с параметром(p) по диапазону (V); p ⇒ V означает операцию изъятия (первого) элемента из V и присвоения его переменной p;
б) общая форма обозначения таких циклов в схематических алгоритмах и программах; конкретная форма заголовка зависит от реализации.

Опыт применения схематического программирования показывает, что установленное в [2] ограничение на тело ОЦ (запрещающее использование в нем условий и терминаторов) может оправдываться лишь в системах учебного назначения. В практическом же программировании оно является слишком ограничительным.

Вводим, наконец, понятие слабого терминатора. Поскольку, по существу, оно является чисто синтаксическим приемом, позволяющим более просто и наглядно записывать циклы, то ограничимся рассмотрением этого понятия на уровне конкретного синтаксиса.

Рассмотрим цикл, содержание которого - (одна) УП-инструкция ↄ (см. рис. I5).

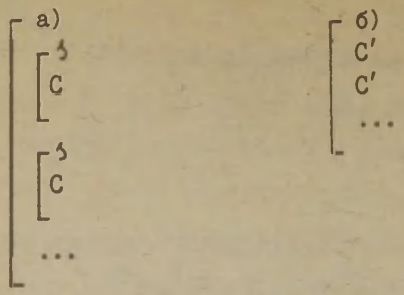


Рис. I5. Сокращенная форма (б) цикла (а);
C′ получено из C изменением некоторых стрелок.

Стрелку, которая доходит (может быть продлена) до линии уровня ℶ , можно назвать терминатором шага цикла – её выполнение приводит опять к началу (следующей) схемы ℶ .

В цикле рассматриваемого вида можно опускать линию уровня схемы ℶ , если длины стрелок, проходящих через неё, уменьшить на единицу, а терминаторы шага заменить на стрелки особого вида, т.н. слабые стрелки (той же длины). Слабая стрелка длины изображается либо в виде ⟵(n), либо в форме продленной пунктирной стрелки, доходящей до n-ой охватывающей схемы. Пример применения слабой стрелки см. на рис. I6$_b$.

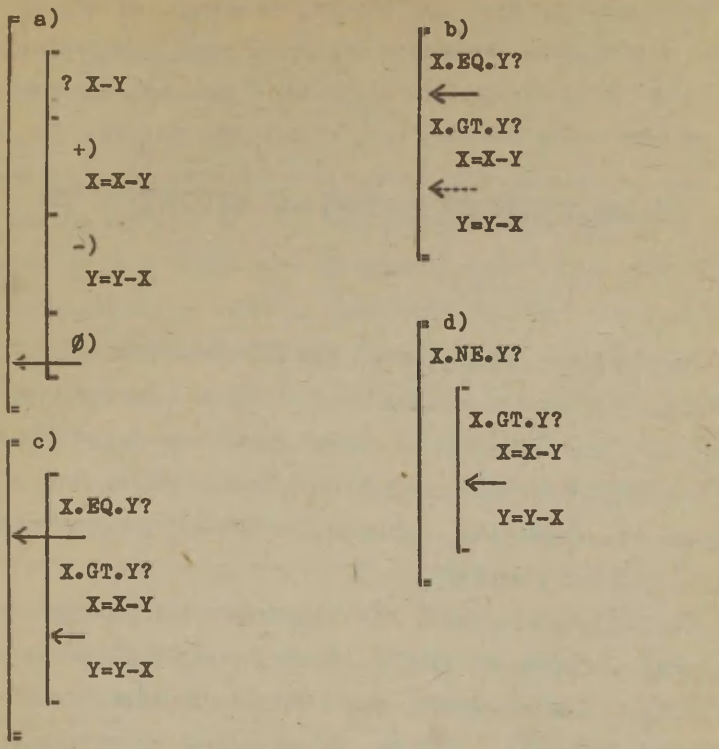


Рис. I6. Четыре реализации алгоритма Эвклида на Е-Форт-
ране.

Литература

I. Вельбицкий И.В., Райков Л.Д., Маклаков А.Я., Мкртумян А.А., Ушаков И.Б.. Новая графическая форма записи алгоритмов и программ. УСиМ, № 6, с.5I-56.

2. Кихо Ю., Схематическое программирование. Труды ВЦ ТГУ,№50, Тарту, I984, с.52-68.

3. Котов В.Е., Введение в теорию схем программ., "Наука",Новосибирск, I978.

4. Ершов А.П. Введение в теоретическое программирование., "Наука", М., I977.

УПРАВЛЕНИЕ ВНЕШНИМИ УСТРОЙСТВАМИ ПЕРСОНАЛЬНОЙ ЭВМ

О. Тоом

Разработчики персональной ЭВМ ТГУ поставили своей главной целью создание простой и в то же время достаточно мощной микро-ЭВМ, которая могла бы найти применение в учебном процессе высших и средних учебных заведений. Учитывалась также возможность применения персональной ЭВМ для управления экспериментальной аппаратурой.

При разработке новой ЭВМ необходимо предусмотреть возможность ее серийного выпуска. В этой связи важными факторами являются как стоимость, так и технологичность изготовления. Пригодная для массового производства персональная ЭВМ должна быть изготовлена на широкораспространенной элементной базе и способна работать с широким ассортиментом внешних устройств. При этом необходимо учитывать стоимость не только создания, но и тиражирования аппаратного оборудования, а также программного обеспечения.

Поскольку тиражирование программы (напр., драйвера внешнего устройства) обходится намного дешевле тиражирования аппаратуры (напр., контроллера внешнего устройства), то аппаратная часть ЭВМ должна быть спроектирована как можно проще - простота аппаратуры к тому же важна при ее ремонте.

При работе с внешними устройствами простая аппаратура (и, соответственно, более сложное программное обеспечение) часто обеспечивает и большую универсальность персональной ЭВМ. Ведь простой контроллер ориентирован не на какой-либо конкретный тип внешнего устройства, а на целый класс устройств. Под управлением одного и того же контроллера могут работать устройства ввода с перфоленты, вывода на перфоленту, различные типы печатающих устройств, кассетный магнитофон и т.п. Для каждого устройства необходимо лишь написать соответствующий драйвер. Конечно, разработки аппаратуры и программного обеспечения при таком подходе тесно связаны и должны проводиться одновременно. Именно такого подхода и придерживались разработчики персональной ЭВМ ТГУ.

Для построения контроллеров внешних устройств во многих микро-ЭВМ используются специальные интегральные схемы ввода/вывода (см. [1]). Такой подход в известной степени облегчает создание программ ввода/вывода и стандартизирует эти программы. В персональной ЭВМ ТГУ, в связи с недостаточным распространением специализированных интегральных схем (напр., серии К580), было выбрано другое решение - построение подсистемы ввода/вывода на базе стандартных интегральных схем серии К555 (малой и средней степени интеграции). С точки зрения программиста такое решение сравнимо с первым и также широко используется (напр., в "Apple II", см. [3]).

В качестве процессора персональной ЭВМ ТГУ был выбран тип КР580ИК80А (аналог процессора "Intel 8080"). Этот 8-разрядный микропроцессор широко распространен и имеет хорошее программное обеспечение.

Объем оперативной памяти может составлять 32К или 64К байт, а объем постоянной памяти от 4К до 16К байт. Так как адресное пространство процессора КР580ИК80А составляет всего 64К байт, то должна быть предусмотрена возможность программного выбора используемой области памяти. В описываемой ЭВМ схема выбора памяти связана с организацией дисплея.

Дисплей растровый (можно использовать любой бытовой телевизор), размер экрана 256×384 точки. Для хранения изображения используется поле оперативной памяти в 12К байт, организованное как битовая матрица: для высвечивания точки на экране телевизора надо записать 1 в соответствующий бит памяти. Основной операцией, связанной с памятью экрана, является запись. Учитывая это обстоятельство, для постоянной памяти выделена та же область адресов, что и для экрана дисплея. Запись по этим адресам выполняется в оперативную память, а чтение - из постоянной памяти. При необходимости читать содержимое памяти экрана можно программно установить режим, в котором и чтение происходит из оперативной памяти.

В первоначальной версии персональной ЭВМ ТГУ дело этим и ограничилось. Однако, практика программирования показала, что программы, находящиеся в постоянной памяти, должны иметь полный доступ к оперативной памяти экрана. По этой причине была введена следующая модификация: все стековые операции работают только с оперативной памятью. Таким образом оказалось возможным все системные программы, реализующие операции с экраном, целиком разместить в постоянной памяти, что существенно повышает надежность ЭВМ. В текущей версии в постоянную память записаны все программы вывода текста на экран и

редактирования входной строки на экране. Так как описываемая ЭВМ не имеет аппаратурного знакогенератора, то эти программы довольно сложны. Это, однако, окупается простотой аппаратуры и возможностью легкой замены шрифта на экране (в частности, можно использовать и такие шрифты, для которых нет промышленной памяти знакогенератора).

Клавиатура персональной ЭВМ ТГУ матричная. Код вводимого символа определяется драйвером клавиатуры, записанным в постоянную память ЭВМ. За счет этого контроллер клавиатуры получился сравнительно простым (состоит всего из одной интегральной схемы). Имеется также возможность полной перестройки клавиатуры путем замены соответствующей таблицы кодировки на какую-нибудь новую.

В число т.н. встроенных внешних устройств, кроме дисплея и клавиатуры, входят еще громкоговоритель и интерфейс магнитофона (в качестве внешней памяти можно использовать любой бытовой магнитофон).

Для расширения возможностей персональной ЭВМ в ней предусмотрена возможность подключения до 7 дополнительных внешних устройств, каждое из которых должно иметь собственный контроллер. По общепринятой схеме каждому дополнительному устройству выделена своя область адресов.

Используемый процессор имеет специальные команды ввода и вывода. Операндом в этих командах является 8-разрядное число - т.н. номер порта (т.е. номер входного или выходного регистра). В описываемой ЭВМ каждому дополнительному внешнему устройству присвоено 16 адресов ввода и вывода. По каждому адресу можно передавать один байт.

Широко распространено и использование части адресного пространства памяти для управления внешними устройствами. Этот подход обеспечивает некоторые преимущества при составлении программ ввода/вывода, но несколько усложняет аппаратурную часть и не допускает использование полнообъемной оперативной памяти. По этой причине в описываемой ЭВМ была выбрана схема использования номеров входных/выходных регистров.

При создании базового программного обеспечения для персональной ЭВМ ТГУ возникла необходимость написания многих драйверов. Например, в числе подключаемых к ЭВМ внешних устройств пока имеется три различных принтера, а в будущем их может оказаться и больше. Драйверы всех печатающих устройств мало отличаются между собой (или от драйвера перфоратора). Однако, программа, обслуживающая все типы внешних устройств, получилась бы неэффективной и громоздкой. Так как драйверы дополнительных устройств, как правило, не выгодно записывать в постоянную память (чтобы не перегружать ее), то можно для каждого типа устройств иметь специальный драйвер и допускать некоторую избыточность программ.

Как показывает практика, многими внешними устройствами можно управлять с помощью метода квитирования (handshaking). Этот метод (описанный, напр., в [1] и [2]) представляет собой асинхронный обмен данными. Более быстрый синхронный обмен в персональной ЭВМ ТГУ используется только при работе с накопителями на гибких магнитных дисках (разработаны контроллеры и драйверы для 5- и 8-дюймовых устройств). Скорость обмена данными с таким накопителем составляет около 30 000 байтов в секунду. Процессор типа КР580ИК80А может с такой

скоростью только читать данные, а не проверять их наличие. По этой причине контроллер дискового накопителя снабжен схемой, которая при чтении обеспечивает синхронную работу процессора с накопителем. Запись на диск происходит синхронно с тактовой частотой процессора.

Для управления более медленными внешними устройствами метод квитирования вполне оправдан. Известный недостаток этого метода - неэффективное использование времени процессора - в персональной ЭВМ окупается за счет простоты аппаратуры.

При методе квитирования каждое внешнее устройство представляется тремя множествами сигналов:

1) сигналы состояния, сообщающие процессору о состоянии внешнего устройства;

2) сигналы управления, с помощью которых процессор может задавать команды внешнему устройству;

3) сигналы данных, по которым происходит обмен содержательной информацией между процессором и внешним устройством.

В самых общих чертах алгоритм обмена данными состоит из четырех частей:

1) ждать готовность внешнего устройства;

2) активировать устройство;

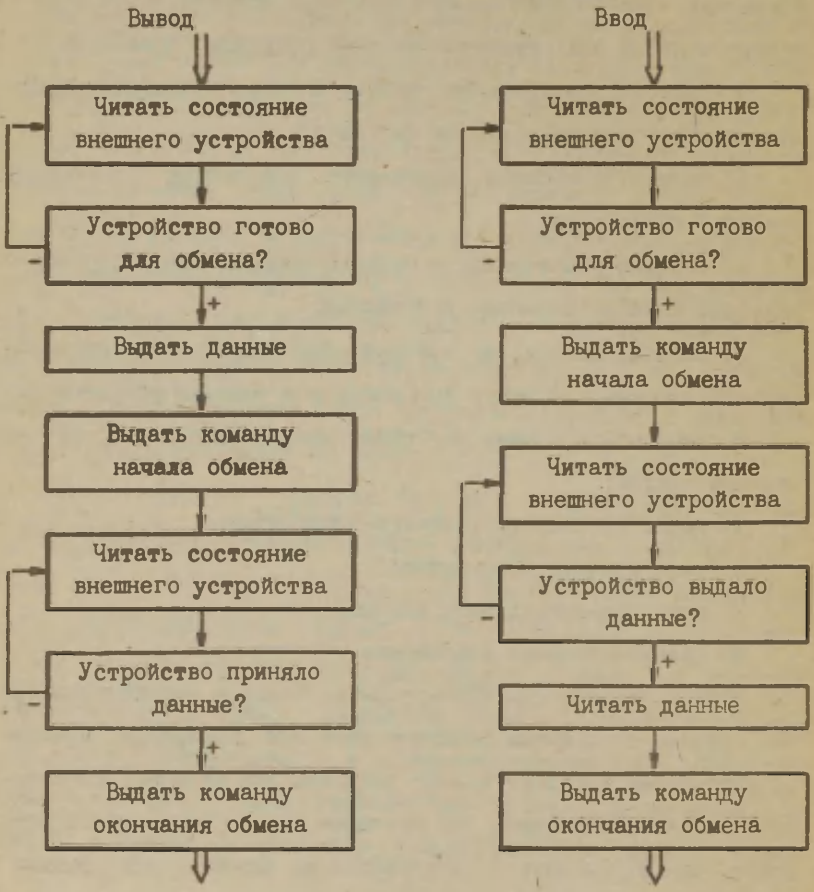
3) принимать/передавать данные;

4) дезактивировать устройство.

В персональной ЭВМ ТГУ алгоритмы ввода и вывода различны. Это связано со схемным решением этой ЭВМ. Контроллер внешнего устройства содержит только регистры для хранения управляющих сигналов и данных, но не имеет возможности временного упорядочения сигналов. Устройства же вывода, как правило,

требуют, чтобы данные были записаны в регистр контроллера раньше стробирующего их сигнала. Данные нельзя выдавать даже одновременно со стробирующим сигналом: в этом случае из-за неодинаковых скоростей переключения регистров внешнее устройство может получить неверные данные (т.н. проблема гонок).

Таким образом, для ввода и вывода в персональной ЭВМ ТГУ приняты следующие общие алгоритмы:



При составлении драйвера для конкретного внешнего устройства по этим блок-схемам необходимо знать следующее:

1) является ли устройство входным или выходным;

2) какие сигналы состояния определяют готовность устройства (номер регистра, номер разряда, полярность);

3) какими сигналами устройство сообщает процессору о приеме/передаче данных;

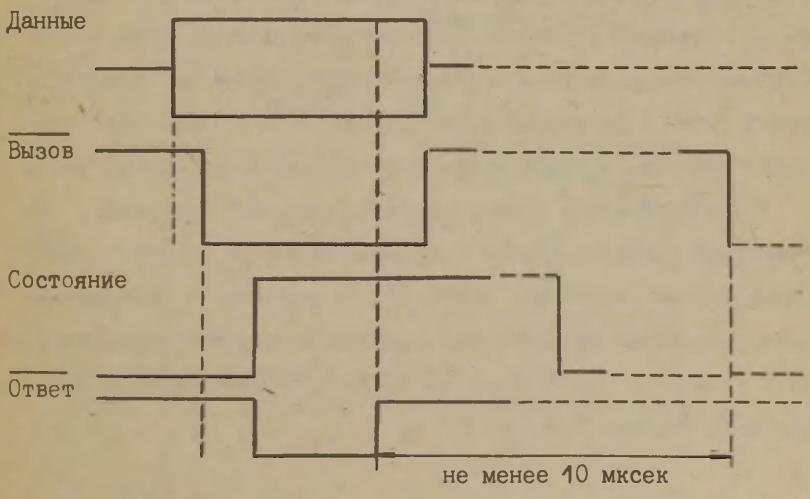
4) какими сигналами управления можно активировать устройство (сообщить о начале обмена данными);

5) какими сигналами можно дезактивировать устройство;

6) какие временные ограничения имеются при работе с этим устройством.

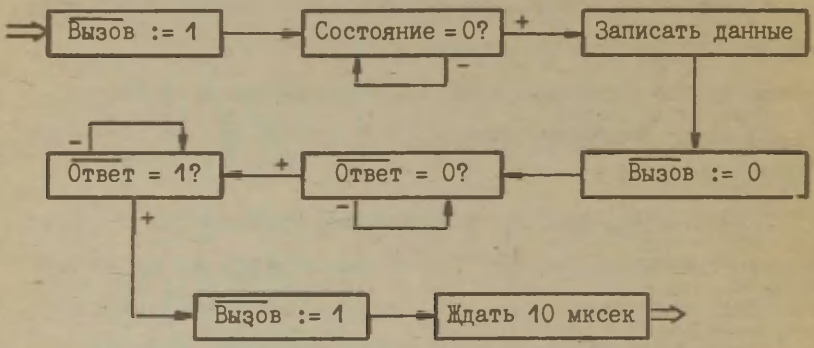
Множества сигналов и их назначения определяются в описаниях конкретных устройств. Там же задаются временные отношения между сигналами, чаще всего в виде временных диаграмм.

Пример. Работа печатающего устройства УПЗС описывается в виде следующей временной диаграммы:



Сигнал "$\overline{\text{Вызов}}$" является здесь управляющим, а сигналы "Состояние" и "$\overline{\text{Ответ}}$" - сигналами состояния. На временной диаграмме задана и полярность сигналов: сигнал "Состояние" активен в состоянии логического нуля, а сигналы "$\overline{\text{Вызов}}$" и "$\overline{\text{Ответ}}$" в состоянии логической единицы.

Подпрограмма вывода одного символа на это печатающее устройство представляется теперь следующей блок-схемой:



Текст подпрограммы вывода символа, написанной по этой блок-схеме на языке ассемблера процессора КР580ИК80А, приведен на следующей странице. Эта программа проста, но ею невозможно пользоваться для управления печатающими устройствами других типов (временные отношения между сигналами могут оказаться другими, сигналы могут быть другой полярностью и т.д.).

В такой ситуации может оказаться полезной программа, генерирующая драйверы. Такая программа получает в качестве исходных данных описание внешнего устройства, а результатом работы программы является текст драйвера на языке ассемблера. Такой результат удобно использовать при реконфигурации операционной системы ЭВМ.

```
; подпрограмма вывода одного символа
; на печатающее устройство УПЗС
; код символа в аккумуляторе
; программа сохраняет все регистры
;
; порты, присвоенные УПЗС
status      equ     Øf9h        ; ввод сигналов состояния
ctrl        equ     Øfch        ; вывод упр. сигнала
data        equ     Øf8h        ; вывод кода символа
;
; расположение сигналов в портах
voz1        equ     ØØØ1ØØØØb ; снимает Вызов
vozØ        equ     ØØØØØØØØb ;
otvet       equ     ØØ1ØØØØØb ; маски сигналов
sost        equ     Ø1ØØØØØØb ; состояния
;
print:      push    psw         ; сохранять регистры
            push    psw         ; A  и  F
            mvi     a,voz1      ; Вызов := 1
            out     ctrl        ;
sostØ:      in      status      ; ждать, пока Состояние
            ani     sost        ; не Ø
            jnz     sostØ       ;
            pop     psw         ; записать данные
            out     data        ;
            mvi     a,vozØ      ; вызывать УПЗС
            out     ctrl        ;
otvØ:       in      status      ; ждать положительного
            ani     otvet       ; фронта сигнала
            jnz     otvØ        ; Ответ
otv1:       in      status      ;
            ani     otvet       ;
            jz      otv1        ;
; следующие две команды обеспечивают и
; необходимую задержку в 1Ø мксек
            pop     psw         ; восстанавливать регистры
            ret                 ;
```

В настоящее время ведется разработка генератора драйверов, выполненного в виде макробиблиотеки.

Литература

I. Клингман Э., Проектирование микропроцессорных систем. М., "Мир", 1980.

2. Хилбурн Дж., Джулич П., Микро-ЭВМ и микропроцессоры. М., "Мир", 1979.

3. Apple II Reference Manual.

О ПРЕДБЛОКАХ В ЗАДАЧЕ ОПТИМАЛЬНОЙ ОРГАНИЗАЦИИ БАЗЫ ДАННЫХ

Т.Э. Фаликс

В работах [1], [2] описана математическая модель оптимальной организации базы данных, отдельные элементы которой состоят из наборов ключевых слов (или пар вида (имя признака, значение признака)). Обычно для оптимизации таких баз данных применяется метод кластеризации ([3], [4]), основанный на попарном сравнении элементов базы данных. В отличие от этих методов в указанной выше модели в основу положена идея групповой, коллективной оценки произвольного множества элементов. Как показано в [1], решение оптимизационной задачи приводит к необходимости распознавать специальные множества, названные минимальными блоками. В работе [6] полностью исследован случай, когда алфавит (совокупность всех ключевых слов) содержит 6 букв, а отдельные элементы содержат по 3 буквы. Для пятибуквенного алфавита решение задачи было приведено в работе [5]. Даже в этих случаях при малых размерах алфавита задача нахождения всех минимальных блоков оказалась достаточно сложной. В [1] были указаны некоторые свойства, присущие всем минимальным блокам, однако этих свойств недостаточно, чтобы выявлять минимальные блоки в алфавитах больших размерностей. При решении этой задачи выяснилась важная

роль множеств другого вида, названных предблоками. Понятие предблока является в некотором смысле дуальным понятию минимального блока: предблоки - это максимальные множества, не являющиеся блоками. В настоящей работе изучается строение простейших предблоков, а затем и предблоков более сложного вида. Приводится теорема, описывающая строение любого предблока в любом алфавите. Вероятно, эта теорема позволит создать алгоритм нахождения достаточно богатого списка предблоков на ЭВМ. В статье для небольших алфавитов (до 82 букв) улучшена оценка возможного числа букв в минимальном блоке.

1. Постановка задачи и основные определения

Напомним некоторые определения, введенные в [1].

Пусть $V = \{a,b,c,\ldots\}$ - конечный алфавит из n букв. Произвольное непустое подмножество алфавита V называется словом и записывается, например, в виде abc, вместо обычной записи $\{a,b,c\}$. Разумеется, abc, bca и т.д. - одно и то же слово.

Для любого непустого множества M слов через kw(M) обозначим множество различных букв, входящих в слова из M. Число элементов множества kw(M) называется уровнем M и обозначается l(M). Число

$$p(M) = 2^{l(M)} - 1 \qquad (1)$$

называется весом множества M.

Слово s накрывает слово s', если $kw(s) \supseteq kw(s')$. Множество M накрывает множество M', если для любого слова $s' \in M'$ существует слово $s \in M$ такое, что s накрывает s'.

Следующая задача названа в [1] задачей о разбиении: найти такое разбиение множества M на непересекающиеся подмно-

жества $M_1,\ldots,M_m$, для которого сумма

$$P = \sum_{i=1}^{m} p(M_i) = \sum_{i=1}^{m} (2^{l(M_i)} - 1) \qquad (2)$$

является наименьшей.

Для произвольного множества M и его произвольного разбиения сумма (2) называется весом разбиения. Вес разбиения, дающего решение задачи о разбиении, обозначим $P_{min}(M)$. Если множество M состоит из слов $u_1,\ldots,u_k$, то, очевидно, что

$$P_{min}(M) \leq \min(p(M), \sum_{i=1}^{k} p(u_i)). \qquad (3)$$

Множество B называется блоком, если

$$P_{min}(B) = p(B). \qquad (4)$$

Блок B является минимальным, если он не содержит других блоков того же уровня. Это означает, что после удаления из B хотя бы одного слова оставшееся множество перестает быть блоком.

Множество слов $\mathcal{P}$ называется предблоком, если оно не является блоком, а все накрывающие его и не совпадающие с ним множества слов являются блоками.

Теорема 1. Минимальное разбиение любого предблока состоит из отдельных слов.

Предположим противное. Пусть минимальное разбиение предблока $\mathcal{P}$ содержит множество слов $M \subseteq \mathcal{P}$ и $s_1, s_2 \in M$. Рассмотрим слово s, состоящее из $kw(s_1) \cup kw(s_2)$. Пусть $\mathcal{P}'$ получено из $\mathcal{P}$ заменой слов s_1 и s_2 на слово s. $\mathcal{P}'$ накрывает $\mathcal{P}$, $\mathcal{P}' \neq \mathcal{P}$ и $\mathcal{P}'$, очевидно, не является блоком. Следовательно, $\mathcal{P}$ не является предблоком, что противоречит условию.

Отметим важное свойство: если множество слов является (не является) блоком (предблоком), то и множество слов, по-

лученное из данного любой перестановкой букв, обладает тем же свойством. Поэтому в дальнейшем все подмножества слов в данном алфавите рассматриваются с точностью до перестановки слов и букв данного алфавита (или с точностью до действия группы подстановок букв данного алфавита).

2. Регулярные множества симплексов и простейшие предблоки

Фиксируем алфавит из n букв (n-алфавит). Каждому слову s в этом алфавите поставим в соответствие слово $A_n s$ в этом же алфавите, состоящее из тех и только тех букв данного алфавита, которые не входят в s. Назовем его противоположным к слову s. Очевидно, $A_n(A_n s) = s$. Каждому подмножеству слов M в n-алфавите поставим в соответствие подмножество $A_n M$ слов в том же алфавите, состоящее из слов, противоположных словам подмножества M.

Лемма 1. Слово s тогда и только тогда не накрывается множеством слов M, когда оно имеет по крайней мере одну общую букву с каждым словом из $A_n M$.

Доказательство очевидно.

Составим все возможные слова, выбирая по одной букве из каждого слова множества $A_n M$. Обозначим множество этих слов $\mathfrak{D}M$. Множество $\mathfrak{D}M$ описывает совокупность слов n-алфавита, не входящих в M: M не содержит тех и только тех слов, которые накрывают хотя бы одно слово из $\mathfrak{D}M$.

Напомним, что симплексом называется n-мерный многогранник, являющийся выпуклой оболочкой n + 1 точек (вершин симплекса), которые не лежат в (n - 1)-мерной плоскости. Грани симплекса есть симплексы меньшей размерности. Каждое слово

из k букв представим в виде (k - 1)-мерного симплекса, у которого каждая вершина соответствует одной из этих k букв. Множество слов есть в таком случае множество симплексов различной размерности, быть может, имеющих общие вершины, ребра и т.д. Особую роль среди таких множеств симплексов играют регулярные множества симплексов, определяемые индуктивно.

Множество R_0, состоящее из двух нульмерных симплексов (двух точек), есть регулярное множество. Пусть R - регулярное множество симплексов. Произвольный принадлежащий ему симплекс заменим на пару непересекающихся симплексов на единицу большей размерности. Полученное в результате множество симплексов будет регулярным. Других регулярных множеств симплексов нет. Подчеркнем, что регулярные множества симплексов состоят из непересекающихся симплексов.

Теперь между вершинами симплексов регулярного множества и буквами некоторого алфавита установим взаимно-однозначное соответствие. Тогда это множество симплексов задает некоторое подмножество слов в алфавите, мощность которого равна сумме вершин всех симплексов.

Лемма 2. Пусть M - множество слов, симплексы которых не пересекаются (т.е. каждая буква алфавита входит только в одно слово). Тогда минимальное разбиение множества $A_n M$ состоит либо из отдельных слов $A_n M$, либо из одного множества - всего $A_n M$.

Это справедливо потому, что любая буква алфавита содержится в любой паре слов множества $A_n M$. Если бы минимальное разбиение множества $A_n M$ содержало множество, состоящее из двух или более слов, то в это множество входили бы все буквы

алфавита и оно накрывало бы все остальные слова из A_nM. В этом случае разбиение, состоящее из множества A_nM, являлось бы минимальным разбиением. Отметим, что утверждение леммы 2 остается справедливым, если суммарное число вершин регулярного множества симплексов меньше или равно мощности алфавита.

Дефектом разбиения множества M слов в n-алфавите на непересекающиеся подмножества $M_1,\ldots,M_m$ называется разность

$$2^n - \sum_{i=1}^{m} p(M_i).$$

Лемма 3. Пусть R - регулярное множество симплексов, имеющее n вершин. Пусть $n \le r$ и каждой вершине поставлена в соответствие буква r-алфавита (разным вершинам - разные буквы). Тогда дефект минимального разбиения множества A_rR равен числу симплексов множества R.

По лемме 2 минимальное разбиение множества A_rR либо состоит из отдельных слов A_rR либо совпадает со всем множеством A_rR. Подсчитаем вес разбиения, состоящего из отдельных слов. Множество A_rR_0 при любом $r \ge 2$ состоит из двух $(r-1)$-буквенных слов и имеет вес $2(2^{r-1}-1) = 2^r - 2$. Дефект его минимального разбиения равен $2^r - (2^r - 2) = 2$. Пусть утверждение леммы 3 верно для некоторого регулярного множества R в любом алфавите, для которого имеет смысл множество A_rR. Докажем справедливость его для регулярного множества $\tilde{R}$, полученного из R применением одного шага индуктивного определения регулярного множества. Пусть регулярное множество симплексов $\tilde{R}$ получено из R заменой некоторого k-мерного симплекса на два $(k+1)$-мерных симплекса. Понятие множества слов $A_r\tilde{R}$ имеет смысл не для всех r, при которых имеет смысл A_rR,

так как число вершин множества $\tilde{R}$ больше числа вершин множества R. Однако, для всех r, при которых определено $A_r\tilde{R}$, определено и A_rR. Сравним веса A_rR и $A_r\tilde{R}$. При подсчете весов A_rR и $A_r\tilde{R}$ совпадут все слагаемые, кроме того, которое соответствовало замененному k-мерному симплексу. Замененному k-мерному симплексу соответствовало слагаемое $2^{r-(k+1)}-1$, которое при подсчете веса $A_r\tilde{R}$ заменилось на два равных слагаемых $2^{r-(k+2)}-1$. Но $2(2^{r-(k+2)}-1) = 2^{r-(k+1)}-2$. Таким образом вес нового множества A_rR на единицу меньше веса старого множества, а дефект минимального разбиения - на единицу больше. Значит при каждом индуктивном переходе к новому регулярному множеству число симплексов увеличивается на единицу, дефект минимального разбиения тоже увеличивается на единицу, и их равенство сохраняется.

Теорема 2. Пусть R - регулярное множество симплексов, имеющее n вершин, каждой из которых приписана буква n-алфавита. Тогда A_nR - предблок в n-алфавите.

Применяя к R лемму 2, получаем, что минимальное разбиение множества A_nR состоит из отдельных слов. По лемме 3 вес этого разбиения равен 2^n минус число симплексов множества R. Следовательно, A_nR не является блоком. Осталось показать, что добавление любого слова, которое не накрывается множеством A_nR, превращает A_nR в блок. По лемме 1 всякое такое слово имеет по крайней мере одну общую букву с каждым из симплексов R и, так как симплексы не пересекаются, состоит по крайней мере из стольких букв, сколько симплексов в R. Тем самым вес этого слова превосходит дефект множества A_nR, и оно не может входить в минимальное разбиение отдельно. Возь-

мем ту составляющую минимального разбиения этого множества, которая содержит добавленное слово. Каждая составляющая минимального разбиения является блоком и потому не может состоять из двух слов. Если кроме добавленного эта составляющая содержит еще хотя бы два слова из A_nR, то она имеет вес $2^n - 1$, так как любые два слова из A_nR содержат все буквы алфавита. Но тогда эта составляющая является блоком уровня n.

Рассмотрим теперь r-алфавит, включающий все буквы n-алфавита ($n \leq r$). Для произвольного слова s в n-алфавите слово A_rs получается из слова A_ns добавлением всех новых букв алфавита r.

Теорема 3. Пусть R - регулярное множество симплексов, имеющее n вершин, каждой из которых приписана буква n-алфавита. Пусть r-алфавит включает все буквы n-алфавита. Тогда A_rR - предблок в r-алфавите.

Доказательство теоремы 3 почти совпадает с доказательством теоремы 2. Теоремы 2 и 3 описывают специальный вид предблоков. Такие предблоки назовем регулярными.

3. Об оценке числа букв в минимальном блоке

В работе [2] приведено доказательство следующей теоремы: пусть уровни всех слов блока B не превосходят k, тогда

$$l(B) \leq ck,$$

где c - константа, заключенная между 4,5 и 5. Там же указано, что при малых k константа c существенно меньше. Ниже приводится теорема, позволяющая улучшить эту константу для $k < 18$, причем улучшение тем больше, чем меньше k. При $k \geq 18$ теорема работы [2] дает лучший результат, чем теорема 4.

Теорема 4. Пусть в r-алфавите существует регулярное множество R, состоящее из d слов. Тогда не существует блока уровня r, все слова которого имели бы уровень не больше d-1.

По лемме 1 множество $A_r R$ накрывает все слова уровня d-1 и меньше. По лемме 2 разбиение $A_r R$ на отдельные слова имеет дефект d и, следовательно, $A_r R$ не является блоком.

Применение теоремы 4 к отдельным регулярным множествам при малых d дает следующие интересные результаты.

Пример 1. Рассмотрим при r = 8 регулярное множество симплексов, состоящее из четырех одномерных симплексов (отрезков). Существование такого регулярного множества симплексов в 8-алфавите по теореме 3 означает отсутствие блоков уровня 8, состоящих из слов уровня, не превосходящего 3.

Пример 2. При r = 12 регулярное множество состоит из трех отрезков и двух треугольников. Следовательно, не существует блока уровня 12, состоящего из слов уровней, не превосходящих 4.

Продолжая этот процесс, т.е. заменяя один из отрезков предыдущего регулярного множества симплексов на два треугольника, получим следующие результаты.

Не существует блока уровня 16 и более, состоящего из слов уровня, не превосходящего 5.

Не существует блока уровня 20 и более, состоящего из слов уровня не превосходящего 6.

Не существует блока уровня 24 и более, состоящего из слов уровня не превосходящего 7. В этом последнем случае соответствующее регулярное множество симплексов состоит из восьми треугольников. Чтобы продолжить этот процесс, нужно

поочередно каждый из треугольников заменять на два тетраэдра. При этом число симплексов будет увеличиваться с каждым шагом на единицу, а число вершин на 5 (а не на 4, как в предыдущих случаях). Таким образом получим, что для уровня 8 и более не существует блоков, состоящих из слов, уровень которых не превосходит k:

r	8	12	16	20	24	29	34	39	44
k	3	4	5	6	7	8	9	10	11
r	49	55	59	64	70	76	82	88	...
k	12	13	14	15	16	17	18	19	...

Вообще, при любом $r = m2^m$ существует предблок, состоящий из 2^m $(m-1)$-мерных симплексов. Поэтому не существует блока уровня $m2^m$ и более, состоящего из слов, уровень которых не превосходит $2^m - 1$.

4. Описание произвольных предблоков

Используя регулярные предблоки, удается построить предблоки другого вида с помощью операции склейки.

Склейкой на множестве слов назовем отождествление любых двух букв, не принадлежащих одному и тому же слову.

Пусть исходное множество слов M в r-алфавите таково, что минимальное разбиение множества $A_r M$ состоит из отдельных слов. Тогда для таких множеств слов M в r-алфавите определим понятие допустимой склейки.

Склейка множества слов M в r-алфавите называется допустимой, если множество слов $\tilde{M}$, получившееся в результате склейки таково, что $A_{r-1}\tilde{M}$ имеет минимальное разбиение, состоящее из отдельных слов.

Всюду в дальнейшем рассматриваются только разбиения множеств слов на отдельные слова. Поэтому для краткости будем далее говорить "дефект", имея в виду "дефект разбиения на отдельные слова".

Применим допустимую склейку k раз к исходному регулярному множеству R. Хотя дефекты исходного противоположного к R множества $A_r R$ и множества, противоположного результату склейки $A_{r-k}\tilde{R}$ совпадают, полученное множество $A_{r-k}\tilde{R}$ не обязательно является предблоком. Это может случиться по двум различным причинам.

Во-первых, в результате допустимой склейки может случиться, что, хотя множество $A_{r-k}\tilde{R}$ не содержит блоков, оно содержит такие симплексы, что их объединение накрывается блоком, вес которого превосходит сумму весов этих симплексов незначительно, не более, чем на дефект $A_r R$. Тогда замена объединения этих симплексов на накрывающий их блок показывает, что $A_{r-k}\tilde{R}$ не является предблоком (однако включать такого типа склейки в число недопустимых не стоит, так как их результаты могут послужить для получения предблоков способом, описанным далее). Во-вторых, если склеек произведено достаточно много, может случиться, что найдутся такие несколько вершин (в малом количестве), что каждому симплексу принадлежит по крайней мере одна из них. Тогда слово, составленное из букв этих вершин, не накрывается $A_{r-k}\tilde{R}$. Если его вес не превосходит дефекта $A_{r-k}\tilde{R}$, это слово может быть добавлено к $A_{r-k}\tilde{R}$. Это означает, что $A_{r-k}\tilde{R}$ не предблок, но после добавления одного или нескольких таких слов станет предблоком. Покажем, что таким образом может быть получен любой предблок.

Лемма 4. Пусть R - множество непересекающихся симплексов размерностей $k_1, k_2, \ldots, k_d$ соответственно. R регулярно тогда и только тогда, когда имеет место равенство

$$\frac{1}{2^{k_1}} + \frac{1}{2^{k_2}} + \ldots + \frac{1}{2^{k_d}} = 2,$$

или эквивалентное равенство

$$2^{n-k_1-1} + 2^{n-k_2-1} + \ldots + 2^{n-k_d-1} = 2^n. \qquad (*)$$

Если R - регулярно, то это равенство вытекает из того, что оно верно при d = 2 и остается верным при каждом шаге индуктивного перехода определения регулярного множества. Докажем обратное. Пусть R - множество симплексов, для которых выполнено (*). При этом можно считать, что $k_1, k_2, \ldots, k_d$ упорядочены в порядке возрастания. Опишем получение R по индуктивному определению регулярного множества. Применим индуктивный переход к множеству R_0 столько раз и таким образом, чтобы получилось регулярное множество симплексов размерности k_1 в количестве 2^{k_1+1}. Затем к этому множеству применим индуктивный переход столько раз и таким образом, чтобы получилось регулярное множество симплексов, состоящее из симплекса размерности k_1 и симплексов размерностей k_2 и т.д. На последнем шаге этого процесса получаются симплексы размерностей k_1, $k_2, \ldots, k_{d-1}$ в том же количестве, что и в множестве R и еще несколько симплексов размерности k_d. Осталось доказать, что их число совпадает с числом симплексов размерности k_d у исходного множества R. Так как равенство (*) выполняется для множества R и для полученного регулярного множества, причем все слагаемые вида 2^{n-k_i-1} при $i = 1, \ldots, d-1$ в (*) совпадают, то совпадает и число слагаемых вида 2^{n-k_d-1}.

Теорема 5. Пусть $\mathcal{P}$ - предблок. Тогда найдутся такие множества слов R и $\mathfrak{D}$, что $\mathcal{P} = A_n\tilde{R} \cup \mathfrak{D}$, где $\tilde{R}$ получено из некоторого регулярного множества R с помощью допустимых склеек. Каждое слово из $\mathfrak{D}$ содержит по крайней мере одну букву из каждого слова $\tilde{R}$, суммарный вес слов $\mathfrak{D}$ не превосходит дефекта $A_n\tilde{R}$. $\mathfrak{D}$ может быть пусто.

Пусть $\mathcal{P}$ - предблок, состоящий из m слов уровней $s_1, \ldots, s_m$ соответственно, где $s_1 \geq \ldots \geq s_m$. Докажем сначала, что

$$P_m = \sum_{i=1}^{m} 2^{s_i} - 2^n \geq 0$$

и существует d такое, что

$$P_d = \sum_{i=1}^{d} 2^{s_i} - 2^n = 0.$$

Предположим, что $P_m < 0$. Тогда $-P_m$ делится на 2^{s_m} (s_m - наименьшее среди $s_1, \ldots, s_m$) и следовательно $-P_m \geq 2^{s_m}$. Теперь самое короткое слово предблока $\mathcal{P}$ (уровня s_m) заменим на накрывающее его слово, добавив произвольную не входившую ранее в него букву. Новое слово имеет уровень $s_m + 1$. Ни одно слово предблока $\mathcal{P}$ не накрывает его. Все остальные слова предблока $\mathcal{P}$ оставим без изменения. Полученное множество слов накрывает предблок $\mathcal{P}$ и не является блоком, так как его вес

$$\sum_{i=1}^{m-1} (2^{s_i} - 1) + (2^{s_m+1} - 1) =$$

$$= \sum_{i=1}^{m} 2^{s_i} + 2^{s_m} - m \leq 2^n - m < 2^n - 1,$$

что противоречит определению предблока. Следовательно, $P_m \geq 0$. Теперь найдем d такое, что $P_d \leq 0$, но $P_{d+1} \geq 0$. Как и выше, $-P_d$ делится на 2^{s_d} и либо равен нулю, либо не меньше,

чем 2^{s_d}, но, так как следующее слагаемое $2^{s_{d+1}} \leq 2^{s_d}$, это означает, что $P_d = 0$. Заметим теперь, что для любого $i = 1, 2, \ldots, d$ уровень слова, противоположного к i-ому слову, равен $n - s_i$, а размерность соответствующего симплекса $n - s_i - 1$. Множество слов, противоположных первым d словам предблока $\mathcal{P}$, обозначим $\tilde{R}$. Если теперь его "расклеить", т.е. построить множество R, состоящее из непересекающихся слов уровней $n - s_1, \ldots, n - s_d$, то по лемме 4 оно будет регулярным. Дефект R равен d. Рассмотрим теперь $\mathcal{P}/A_n\tilde{R}$. Обозначим его $\mathfrak{D}$. Так как $\mathcal{P}$ предблок, суммарный вес слов $\mathfrak{D}$ не превосходит d-1. Слова из $\mathfrak{D}$ не накрываются ни одним словом из $A_n\tilde{R}$, а потому, по лемме 1, имеют по крайней мере одну общую букву с каждым словом из $\tilde{R}$.

Следствие. Пусть $\mathcal{P}_n$ – предблок в n-алфавите и $\mathcal{P}_n = A_n\tilde{R}\cup\mathfrak{D}$, где $\tilde{R}$ и $\mathfrak{D}$ описаны в формулировке теоремы 5. Пусть r-алфавит содержит n-алфавит. Тогда $\mathcal{P}_r = A_r\tilde{R}\cup\mathfrak{D}$ не является блоком уровня r.

Действительно, дефект $A_r\tilde{R}$ в r-алфавите совпадает с дефектом $A_n\tilde{R}$ в n-алфавите, т.к. $\tilde{R}$ – регулярно. Вес $\mathfrak{D}$ одинаков в r-алфавите и в n-алфавите. Следовательно, дефект $\mathcal{P}_r$ в r-алфавите совпадает с дефектом $\mathcal{P}_n$ в n-алфавите и $\mathcal{P}_r$ не является блоком уровня r.

В заключение отметим еще следующий факт. Множество M слов в n-алфавите тогда и только тогда является блоком, когда для любого предблока в n-алфавите найдется слово из M, не принадлежащее этому предблоку. Справедливость этого утверждения непосредственно вытекает из определений блока и предблока.

Литература

I. Сокгобензон В. Я., Цаленко М. Ш., Об оптимальной организации информации в базах данных. - М.: Труды ВНИИДАД, 1981, 12.

2. Цаленко М. Ш., Об информационной константе, связанной с оптимальной организацией базы данных. - Научно-техническая информация. Сер. 2. Информационные процессы и системы, 1981, № 7, с. 20-23.

3. Солтон Дж., Динамические библиотечно-информационные системы. - М.: Мир, 1979.

4. Bergmark D., Salton I., Clustered file generation and its application on computer science taxonomies. - В кн.: Information processing. 1977, North. Holland, p. 77-82.

5. Цаленко М. Ш., Комбинаторные задачи информационного поиска. - Квант, 1979, № 12, с. 6-12.

6. Фаликс Т. Э., Цаленко М. Ш., К задаче об оптимальной организации базы данных. - В сб.: Вопросы информационной технологии. М.: ВНИИСИ, 1983, № 6, с. 73-83.

Содержание

70 коп.