Госстрой СССР
ЦНИПИАСС

ФОНД АЛГОРИТМОВ И ПРОГРАММ ДЛЯ ЭВМ
(В ОТРАСЛИ "СТРОИТЕЛЬСТВО")

РЕФАЛ

Базисный РЕФАЛ и его реализация на вычислительных машинах

(методические рекомендации)

СПЕЦИАЛЬНЫЙ РАЗДЕЛ

V

МОСКВА 1977

УДК 001(47):681.3.04:800.92

Дается полное описание подмножества универсального алгоритмического языка рекурсивных функций (базисного РЕФАЛа), предназначенного для символьных преобразований, а также приводятся методы его трансляции и особенности реализации на отечественных вычислительных машинах типа: ЕС ЭВМ, БЭСМ-6, "Минск-32", М-220, М-222, БЭСМ-4.

Одно из главных применений РЕФАЛа – использование его в качестве средства описания специализированных языков программирования и общения с ЭВМ, потребность в которых возникает во многих областях исследования, в частности при создании автоматизированных систем управления и проектирования в строительстве.

В подготовке рекомендаций участвовали: ЦНИПИАСС, МИФИ, ИПМ АН СССР, МГУ и другие организации.

Авторы:
гл. I, II и п.I гл. III – В.Ф. Турчин
п.2 гл. III – И.Б. Щенков
п.3 гл. III и гл. V – С.А. Романенко
гл. IV – С.А. Романенко, Анд. В. Климов
гл. VI, VIII – Анд. В. Климов
гл. VII – Е.Б. Травкина
гл. IX – Арк. В. Климов
гл. X – А.Г. Красовский, В.Ф. Хорошевский

© *Центральный научно-исследовательский и проектно-экспериментальный институт автоматизированных систем в строительстве (ЦНИПИАСС) Госстроя СССР, 1977 г.*

ВВЕДЕНИЕ

Расширение сферы применения вычислительной техники при решении различных задач планирования и управления выдвигает серьезные проблемы сопряжения между системами понятий экономики и программного обеспечения современных ЭВМ. Процесс перевода экономических понятий в машинные часто выполняется цепочкой специалистов: постановщик задач – алгоритмист – программист, с существенными конфликтами и трениями на "стыках".

Разработка систем программирования, позволяющих легко вводить и описывать специализированные языки, содержащие заданную систему понятий, помогает в большой степени ускорить и облегчить постановку, алгоритмизацию, программирование и отладку насущных задач экономики.

Рефал* – машинно-независимый алгоритмический язык, ориентированный на так называемые "символьные преобразования": перевод с одного языка (искусственного или естественного) на другой, алгебраические выкладки, решение информационно-логических задач и т.п. Говоря более точно, рефал – универсальный метаязык для преобразования объектов языковой природы. Важнейшим приложением рефала является его использование в качестве метаязыка для построения системы макрокоманд и специализированных языков.

Первоначально рефал был создан как довольно абстрактный мета-алгоритмический язык [1, 2], обладавший некоторыми чертами, которые делали трудной его реализацию на вычислительных машинах. Затем он был упрощен [3, 4], что облегчило создание эффективных трансляторов для этого языка. В процессе практического использования языка выработался определенный стиль программирования на нем, и стало ясно, что решающую роль в этом стиле играет понятие рекурсивной функции. Так возникло новое название языка, являющиееся сокращением от "РЕкурсивных Функций АЛгоритмический язык".

***Названия алгоритмических языков, образованные как сокращения, пишутся обычно заглавными буквами. Однако это становится неудобным, когда название встречается в различных падежах и используется в словесных конструкциях (рефал-машина и т.п.). Поэтому в дальнейшем слово "рефал" будет набираться строчными буквами и вообще считаться полноправным словом, хотя и возникшим как новообразование.*

По мере разработки методов эффективной реализации рефала стало возможным постепенно отказываться от прежних ограничений, а также вводить в язык новые средства, не противоречащие его исходным идеям и не нарушающие его цельности. Возникла необходимость как-то обозначить различные (по объему возможности) версии языка. Базисный рефал - это тот объем языка, который (за некоторыми исключениями) реализован в настоящее время на вычислительных машинах.

В рефале объединены две черты, которые во время его создания были воплощены соответственно в языках ЛИСП [5,6] и КОМИТ [7]: использование рекурсивной функции в качестве основного понятия языка и введение шаблонов для распознавания строк знаков. Из известных языков програмирования рефал ближе всего к языку СНОБОЛ [8] (хотя при создании рефала в 1965-1966 гг. авторы не были знакомы со СНОБОЛом). Наиболее существенное отличие рефала от СНОБОЛа - отсутствие оператора перехода. Эта черта определила тот стиль программирования на рефале, который получил в последнее время название "структуризованного" (structured) программирования и рассматривается многими как важное достижение в развитии программистской техники. Программирование на рефале структуризовано и не может быть иным по самой природе языка рефал.

Настоящие методические рекомендации содержат:

1. Описание рефала (в базисной версии) и методов программирования на нем.

2. Общие принципы построения рефал-системы на основе компиляции.

3. Руководства пользователю по запуску программ на различных ЭВМ.

I. ОПИСАНИЕ ЯЗЫКА

1. Универсальный метаязык программирования

Практическое использование узкоспециализированных языков встречается с трудностями трансляции. Если много пользователей решают много однотипных задач, для них создается проблемно-ориентированный язык и транслятор к нему. Но в случае нестандартных задач, обладающих своей иерархией понятий и требующих узкоспециализированных трансляторов, затраты труда программистов значительно возрастают. Чтобы сделать приемлемым на практике метод программирования, включающий создание иерархии специализированных языков и трансляторов для каждой крупной задачи, необходим универсальный метаязык, удовлетворяющий двум требованиям. Во-первых, он должен быть удобным для человека. Текст на метаязыке должен представляться не в виде сложной программы, а в виде семантического описания языка, с которого производится трансляция, в виде набора предложений, выражающих понятия этого языка через более простые понятия. Во-вторых, метаязык должен допускать эффективную реализацию на вычислительной машине, иначе его систематическое использование повлечет большие потери машинного времени.

Существующие в настоящее время языки макрокоманд и аппарат процедур (например, в АЛГОЛе-60) способны выразить иерархию понятий в рамках данного языка, но не могут вывести за его пределы. Иерархия макрокоманд или процедур - это нечто гораздо меньшее, чем иерархия языков. Универсальный метаязык позволит программисту без затруднения строить языковые иерархии, предоставляя возможность:

1) описывать сколь угодно сложные преобразования одного текста в другой без каких бы то ни было ограничений на характер преобразования;

2) вводить в процесс трансляции любую внешнюю информацию, являющуюся, быть может, отражением проблемной ориентации языка или подъязыка;

3) обращать язык сам на себя, описывая преобразование только что введенных или потенциально возможных правил преобразования.

Система программирования, основанная на универсальном языке, либо сильно ограничивает возможности программиста, либо приводит

к созданию сложнейшего языка, трудного не только для реализации, но и для изучения, и для использования. При помощи же метаязыка можно получить гораздо более гибкую систему, способную к бесконечному развитию, так как система программирования, основанная на универсальном метаязыке, свободна от ограничений на вводимые понятия. Действительно, так как наш метаязык направлен на алгоритмические языки и в то же время он сам также является алгоритмическим языком, в качестве объектов работы он может иметь свои собственные тексты.

Итак, метаязык должен быть удобен для описания манипуляций над произвольными (следствие универсальности) текстовыми объектами, то есть над последовательностями символов из некоторого конечного алфавита. Эта ориентация языка получила название "преобразование символьной (или текстовой) информации", или просто "символьные преобразования" – "Symbol manipulation" в литературе на английском языке.

Язык рефал (алгоритмический язык рекурсивных функций) задуман и функционирует как универсальный метаязык для описания преобразований языковых объектов. Чтобы яснее показать связь понятия рекурсивной функции с задачей создания универсального метаязыка, будем на первых порах называть его языком *M*. Описав неформально важнейшие понятия рефала в процессе их становления, мы дадим (в разд.10) его формальное описание.

2. Понятие конкретизации

Машинно-независимые алгоритмические языки удобны для записи задач из определенной области вследствие того, что они строятся на основе формализации ряда понятий, важных и характерных для данной специальной области. Язык, предназначенный для описания любых языков и понятий (метаязык) будет удобен для человека лишь в том случае, если он схватит какие-то чрезвычайно общие и в то же время важные черты человеческой языковой деятельности.

Важнейшей чертой естественных языков и формализованных языков математики является наличие в них иерархии понятий. Принцип образования сложных и абстрактных понятий из более простых и конкретных путем конструирования и абстрагирования, несомненно, лежит в основе построения любого языка. Таким образом, метаязык должен позволять описывать иерархии понятий. Понятия изображаются языковыми объектами, в естественных языках – это слова, группы слов, предложения. Понимать языковый объект – значит уметь пройти в обратном направлении путь его построения

до самых нижних этажей иерархии понятий. Понимать абстрактное понятие – значит уметь его конкретизировать в каждой заданной ситуации, понимать сложное понятие – значит уметь свести его к ряду более простых. И то и другое означает замену языкового объекта, занимающего более высокое положение в иерархии, на ряд объектов, занимающих в ней более низкое положение. Эту операцию будем называть конкретизацией языкового объекта.

Итак, семантика языкового объекта определяется правилом его конкретизации, а семантика языка в целом – совокупностью правил конкретизации, которая позволяет путем ряда шагов свести каждый языковый объект к некоторым несводимым элементарным объектам. Операцию конкретизации можно также определить как переход от имени к значению.

Эта схема определения семантики объектов языка является упрощенной, однако без упрощения невозможна никакая формализация, а ведь наша задача и заключается в семантических описаниях. Поэтому при построении языка M эту схему примем за основу. Введем два знака: k и $\perp$, которые будем называть конкретизационными или функциональными скобками*, и в которые будем заключать языковой объект, подлежащий конкретизации. Так что, например, если x есть некоторая переменная величина, то $kx\perp$ (конкретизация x) будет изображать значение этой величины. Другой пример: объект $k\,28+7\perp$ будет рано или поздно заменен (если только правильно определена операция сложения) на объект 35. Выполнение конкретизации – переход от имени к значению – объявим основной (и, по существу, единственной) операцией в языке M. Эту операцию будет выполнять машина C_M, "понимающая" язык M. (Когда вместо "язык M" будем говорить "рефал", машину C_M будем называть "рефал-машиной"). Информацию для выполнения всевозможных конкретизаций будем записывать в виде предложений языка M (правил конкретизации).

Очевидно, что явное введение конкретизационных скобок непосредственно вытекает из задач, поставленных перед языком M. В специализированных формализованных языках, опирающихся на фиксированную иерархию понятий, обычно договариваются о такой системе обозначений, которая отражала бы положение языкового объекта в этой иерархии. Разделение производится по типу знаков и по другим синтаксическим признакам. Так, в АЛГОЛе идентификатор может иметь смысл переменной и принять, например, значение 2.7183. Синтаксический разбор текста указывает, где имя, где значение и когда надо от имени перейти к значению. В нашем случае, когда проекти-

---

**Знак k будем также называть знаком конкретизации, а знак $\perp$ – конкретизационной точкой.*

руется метаязык, рассчитанный на произвольные системы понятий и обозначений, переход от имени к значению должен быть указан с помощью специальных знаков.

3. Знаки, символы, выражения

Теперь необходимо уточнить структуру объектов языка *M*. Подобно естественным языкам язык *M* является одномерным знаковым языком, т.е. его объекты суть последовательности некоторых знаков. Под знаком мы понимаем минимальную синтаксическую единицу языка, не расчленимую на составные части аналогично буквам или фонемам естественного языка (в письменной и устной формах, соответственно). Так как метаязык *M* должен быть применим к любым языкам, не следует сильно ограничивать набор знаков языка *M*. В то же время нам необходимо, очевидно, какое-то число специфических знаков с фиксированным значением. Поэтому мы объявим, что все знаки языка делятся на собственные и объектные знаки. Набор объектных знаков не фиксируется. Будем только считать, что он ограничен. Здесь будем использовать в качестве объектных знаков заглавные буквы русского и латинского алфавитов, цифры и такие общепринятые математические знаки, как +, –, = и т.п. Что касается собственных знаков, то в данной (неформальной) части описания языка будем вводить их по мере надобности. К настоящему моменту известно два собственных знака: *k* и ⊥. Ограничение объектных знаков заглавными буквами дает возможность использовать строчные латинские буквы в качестве собственных знаков рефала.

Буквы или фонемы в естественном языке не имеют как таковые значения, они служат лишь материалом для построения минимальных семантических единиц – морфов: корней слов, суффиксов, предлогов и т.п. При этом морф может состоять из одной буквы, например "в" (предлог) или из нескольких букв, например "дуб" (корень), "из" (предлог), "ая" (окончание). В языке *M* минимальную семантическую единицу мы назовем символом и положим, что символ может изображаться как одним объектным знаком, так и группой знаков, ограниченной тем или иным способом слева и справа. В качестве ограничителя предлагаем пользоваться знаком ′ (кавычка), который будем рассматривать как собственный знак языка *M*. Итак, определяем символ как объектный знак или как последовательность объектных знаков, взятую в кавычки (составной символ).

Примеры символов:

Z

=

′ЕСЛИ′

′КОНЕЦ. ПЕРЕХОД К СЛЕД-ПРОЦЕДУРЕ′

Примеры последовательностей знаков, которые не являются символами:

XYZ

'*kx*⊥'

ЕСЛИ'

''*B*

В естественных языках морфы соединяются в слова, словосочетания и предложения, имеющие определенную синтаксическую структуру. Путем синтаксического разбора можно каждый языковой объект разложить на составляющие его объекты и восстановить путь построения этого объекта из элементарных частей. То же относится и к формализованным языкам. Синтаксические средства, используемые для конструирования сложных объектов из простых (а следовательно, и для разложения сложных объектов на простые), могут быть самыми разнообразными; создавая язык *М*, мы не можем предполагать в его объектах какой-то определенный, специализированный синтаксис. Тем не менее, есть одна черта, общая для всех синтаксических схем, которую мы не можем не учесть при построении метаязыка: это тот факт, что синтаксический анализ (и синтез) всегда порождает определенное дерево языковых объектов (синтаксический анализ фразы русского языка см.на рис.1.1 и синтаксический анализ алгебраического выражения - на рис.1.2). Простейший и наиболее привычный способ изображения такой структуры в виде линейной последовательности знаков - это использование скобок. Этот способ широко применяется во многих формализованных языках, начиная со школьной алгебры. Мы привыкли сразу выделять скобки в ряду других знаков; мы относимся к ним не как к полноправным символам, а как к специальным знакам, служащим для придания объекту иерархической структуры. Это отражено на рис.1.2, где скобки не указаны как элементы строк, наряду с подвыражениями и знаками операций, а проявляются только в построении дерева.

Итак, для изображения синтаксических конструкций введем в языке *М* круглые скобки, которые будем рассматривать как собственные знаки языка. Круглые скобки, придающие структуру объектам работы, будем называть **структурными** в отличие от конкретизационных скобок, указывающих на необходимость выполнения конкретизации. Заметим, что кавычка, которой мы пользуемся для ограничения составных символов, тоже может быть названа скобкой, а именно, **символьной** скобкой. Причина, по которой символьная скобка одна, а скобки двух других типов - парные, состоит в том, что при построении сложных символов мы ограничиваемся одноуровневыми структурами, поэтому можем обойтись одним ограничителем.

Теперь введем понятие **выражения**, которое можно определить как последовательность знаков, правильно построенную относительно всех трех типов скобок: символьных, структурных и функциональных.

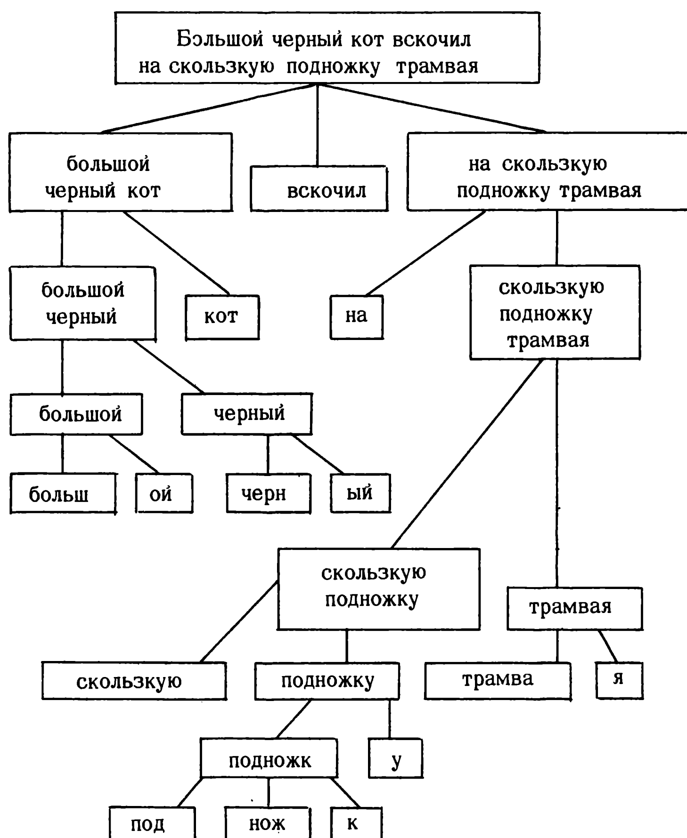


Рис.1.1

Расшифровка этого определения такова:

1. Пустой объект есть выражение.

2. Символ есть выражение. Свободная переменная есть выражение (О свободных переменных см.ниже).

3. Последовательность выражений есть выражение.

4. Выражение, взятое в структурные или функциональные скобки, есть выражение.

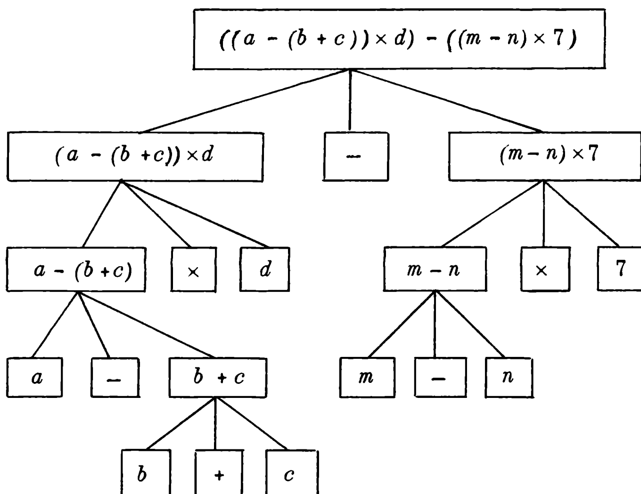


Рис. 1.2

5. Объект, который на основании предыдущего не может быть квалифицирован как выражение, не есть выражение.

Примеры выражений:

$ABC(\ )$

$A = (B + C)/2$

k 'ЕСЛИ' $A < B$ 'ТО' A 'ИНАЧЕ' $B\perp$

$k\ k\ X\perp (kY\perp)\perp$

Примеры последовательностей символов, которые не являются выражениями:

k 'СЛОЖ' A, B

$)\ kX\perp($

k 'СЛОЖ' $A, (B\perp)$

Полезно также определить **терм** как некоторый частный случай выражения. Терм – это либо символ, либо выражение, взятое с структурные или конкретизационные скобки. Следовательно, выражение всегда есть последовательность из некоторого числа (быть может, нуля) термов.

Поскольку скобки мы рассматриваем не как объектные знаки, а как собственные знаки, договоримся, что язык M будет иметь дело только с выражениями. Последовательности символов, не являющиеся выражениями, т.е. неправильно построенные относительно

структурных или конкретизационных скобок, объявляем неправомочными в качестве языковых объектов и допускаем их лишь в качестве составных частей выражений.

4. Предложения

Алгоритмические языки можно разделить на две группы. Первую группу образуют языки, которые назовем языками операторного типа. Элементарными единицами программы являются здесь операторы, т.е. приказы, выполнение которых сводится к четко определенному изменению четко определенной части памяти машины. Типичным представителем этой группы является язык машины Тьюринга. Сюда же относятся языки конкретных вычислительных машин, а также такие широко распространенные языки программирования, как ФОРТРАН и АЛГОЛ. Языки второй группы назовем языками сентенциального типа. Программа, написанная на таком языке, представляется в виде набора предложений (соотношений, правил, формул), которые машина, понимающая данный язык, умеет каким-то образом применять к обрабатываемой информации. Язык нормальных алгоритмов А.А.Маркова является примером сентенциального языка, созданного с теоретическими целями. Язык ЛИСП может служить примером сентенциального языка для практических целей.

Оба типа алгоритмических языков имеют прообразы в естественных языках: операторные языки – в виде повелительного наклонения (приказания), сентенциальные – в виде изъявительного наклонения (описания). Сравнивая операторные и сентенциальные языки, нельзя не обратить внимание на соотношение между использованием изъявительного и повелительного наклонения в естественных языках. Мы видим, что изъявительное наклонение является несравненно более распространенным и образует, в сущности, основу языка, в то время как повелительное наклонение предстает в виде некоторой специальной модификации. Нетрудно понять и причины этого. Если язык достаточно развит, т.е. содержит много сложных понятий, то основная масса текстов на этом языке выполняет задачу выражения связей между понятиями, т.е. является описаниями. При наличии этой основной массы приказания, даже очень сложные ("подправьте-ка, пожалуйста, в этой статье литературный стиль"), занимают совсем мало места, если даже допустить, что язык в целом и создавался-то для выражения подобных приказаний.

Итак, относительный вес изъявительного наклонения является мерой развитости языка. Современные вычислительные машины понимают только операторные языки и находятся, таким образом, на уровне животных, язык которых содержит исключительно повелительное наклонение. Создавая проблемно-ориентированные алгоритмические языки, люди в той или иной степени вводят в них изъявитель-

ное наклонение, приближая машину к человеку. Рассмотрим, например, АЛГОЛ. Хотя этот язык и является операторным по своей структуре, наличие в нем арифметических и логических выражений является элементом изъявительного наклонения, а наличие описаний – это уже явное использование изъявительного наклонения.

Язык *М* будет у нас сентенциальным в своей основе и по своей природе, ибо он и задуман как язык для описания связей, соотношений между понятиями. Вся информация, которую нужно передать на языке *М* , должна быть выражена в виде п р а в и л к о н к р е т и з а ц и и. Если продолжать аналогию с естественным языком, то правила конкретизации соответствуют предложениям естественного языка, поэтому мы будем также называть их п р е д л о ж е н и я м и (см. табл.1). Отделять предложения друг от друга будем собственным знаком § (параграф), помещая его перед каждым новым предложением.

Т а б л и ц а 1

Соответствие между объектами языка *М* и естественного языка

Уровень	Язык *М* (рефал)	Естественный язык
1	Знак	Фонема, буква
2	Символ	Морф
3	Выражение	Слово, словосочетание
4	Предложение	Предложение

Поскольку правило конкретизации есть указание для замены одного языкового объекта на другой, предложение должно состоять из л е в о й ч а с т и (заменяемый объект) и п р а в о й ч а с т и (объект, заменяющий левую часть). Для разделения левой и правой частей нам понадобится еще один собственный знак. Им может послужить знак ~ ("тильда"). Итак, предложение, выражающее тот факт, что значение переменной величины X есть 137, мы запишем в виде

$$\S kX \sim 137$$

т.е. выражение $kX \perp$ (конкретизация X) должно быть заменено на 137. Конкретизационная точка $\perp$ в конце левой части подразумевается. Мы могли бы договориться писать:

$$\S kX\perp \sim 137$$

однако для краткости точку опускаем, вернее, сливаем ее со знаком ~, так что с точки зрения скобочной структуры последний играет роль правой функциональной скобки. Между знаком § и первым знаком k можно вставлять последовательность объектных знаков, которая будет служить номером предложения, или комментарием к нему, например:

$$\S 1.1\ kX \sim 137$$

При представлении рефал-объектов на бумаге (в отличие от объектов, вводимых в вычислительную машину) будем, как правило, пользоваться "скорописью" – сокращенным представлением, которое однозначно переводится в стандартное, формально описанное представление. Введем первое правило скорописи: в предложениях без комментария знак § можно опускать, при этом каждое предложение должно начинаться с определенной позиции новой строки, а если предложение не умещается в строке, то переносимая часть должна начинаться с другой позиции.

Опишем теперь структуру машины C_M, которая, используя предложения, будет выполнять конкретизации. Очевидно, что должно существовать какое-то выражение, которое является объектом работы машины C_M. Будем считать, что это выражение находится в поле зрения машины C_M. Кроме того, машина C_M должна иметь поле памяти для хранения предложений. Работа машины C_M будет складываться из последовательных шагов, каждый из которых представляет выполнение одного акта конкретизации.

Пусть в поле памяти машины C_M находится §1.1, а в поле зрения – выражение

$kX\perp$

Тогда за один шаг машина C_M заменит содержимое поле зрения на выражение

137

после чего она остановится, ибо знаков конкретизации в поле зрения больше нет, и, следовательно, делать ей нечего.

Так как поле памяти содержит, вообще говоря, набор (последовательность) предложений, может оказаться, что для выполнения данной конкретизации пригодно не одно, а несколько предложений. Например, в поле памяти, кроме § 1.1, может стоять еще предложение

§1.2 $kX \sim 274$

Неоднозначность, которая отсюда может возникнуть, устраняется следующим образом. Договоримся, что машина C_M просматривает предложения в том порядке, в котором они стоят в поле памяти, и применяет первое из них, которое окажется подходящим, после чего шаг считается выполненным.

Поле зрения машины C_M может также содержать сколько угодно конкретизационных скобок, причем они могут быть как угодно вложены друг в друга. Следовательно, необходимо договориться, каким образом будет машина C_M выбирать выражение, с которого надо начинать процесс конкретизации.

Назовем *ведущим* знаком конкретизации тот знак k, который вместе с парным ему знаком $\perp$ ограничивает выражение, подлежащее конкретизации в первую очередь. Так как прежде чем вычислять значение функции, необходимо вычислить значения всех аргументов,

введем следующее соглашение. Ведущим знаком k является первый из знаков k, в области действия которого (т.е. в последовательности знаков до парной скобки $\perp$) нет ни одного знака k. Теперь можно так описать работу машины C_M. В начале каждого шага машина C_M просматривает поле зрения слева направо в поисках ведущего знака конкретизации. Найдя его, она сравнивает терм, начинающийся с этого знака, с левыми частями предложений, стоящих в поле памяти. Найдя подходящее выражение, она делает в поле зрения необходимую замену, после чего выполнение данного шага заканчивается и машина приступает к выполнению следующего шага.

Пусть, например, поле памяти содержит набор предложений:

$kX \sim 137$

$kX \sim 274$

$kY \sim 2$

$k137+2 \sim 139$

а поле зрения содержит выражение:

$kkX\perp + kY\perp\perp$

На первом шаге ведущим знаком будет знак k, стоящий перед символом X. В результате замены поле зрения принимает вид:

$k137 + kY\perp\perp$

Теперь первоочередной конкретизации подлежит выражение $kY\perp$. Первые два предложения в поле памяти оказываются неприменимыми, и применяется третье предложение. В результате выполнения второго шага поле зрения принимает вид:

$k137 + 2\perp$

На третьем шаге применяется четвертое предложение, и в поле зрения оказывается выражение

139

которое уже не содержит знаков k.

5. Свободные переменные

Во всех рассмотренных до сих пор примерах предложения, которые оказывались применимыми для выполнения конкретизации, имели левые части, в точности совпадающие с конкретизируемым выражением. Чтобы заставить машину C_M выполнить сложение $137+2=139$, мы ввели в поле памяти специальное предложение. Очевидно, нельзя в поле памяти запастись предложением на каждую пару складываемых чисел. Чтобы записать предложение, применимое более чем к одному конкретизируемому выражению, необходимо ввести в него свободные переменные, которые при различных применениях предложения могут принимать различные значения.

Языковые объекты, с которыми имеет дело машина C_M, это всегда выражения, включая сюда в качестве частных случаев термы и символы. Поэтому мы введем три типа свободных переменных,

соответственно синтаксическому типу тех объектов, которые могут быть их значениями. Для изображения свободных переменных введем три собственных знака – признака типа свободных переменных: e – признак выражения, t – признак терма и s – признак символа. Сначала рассмотрим так называемые простые свободные переменные. Они изображаются парой знаков, состоящей из признака и следующего за ним произвольного объектного знака – индекса свободной переменной. Так, $e1$, $e2$, eA – свободные переменные выражения. Свободная переменная выражения может принять в качестве значения любое выражение. Значением свободной переменной терма (скажем $t1$) может быть любой терм, например A или (AB), но никак не AB. Значением свободной переменной вида sA может быть только символ.

В скорописи будем помещать индекс переменной в нижнюю позицию, заменяя одновременно заглавную букву на строчную, например:

e_a, t_1, s_x

Это эквивалентно

$eA, t1, sX$

Разрешая использовать в левых и правых частях предложений свободные переменные, получаем мощное изобразительное средство и завершаем конструирование основного ядра языка M. Теперь машина C_M, чтобы решить, применимо ли данное предложение к данному конкретизируемому выражению, должна определить, можно ли придать свободным переменным в левой части предложения такие значения, чтобы левая часть совпала с конкретизируемым выражением; разные вхождения одной и той же переменной должны заменяться на одно и то же значение. Это действие машины C_M будем называть синтаксическим отождествлением. В случае успешного синтаксического отождествления свободные переменные, входящие в левую часть, принимают определенные значения (соответствующие, конечно, их синтаксическому типу), и при подстановке правой части предложения в поле зрения на место свободных переменных вписываются их значения. В правой части предложения разрешается использовать только такие свободные переменные, которые входят в левую часть. Если синтаксическое отождествление невозможно, предложение неприменимо.

Рассмотрим несколько примеров. Допустим, мы хотим описать понятие "первый символ выражения". Это значит, что, например, выполнение конкретизации:

k ПЕРВЫЙ СИМВОЛ ВЫРАЖЕНИЯ $Z(AB)+F\perp$

дало бы символ Z, выполнение конкретизации

k ПЕРВЫЙ СИМВОЛ ВЫРАЖЕНИЯ $+\perp$

дало бы символ $+$ и т.д.

Задачу решает следующее предложение:

§ **2.1** k ПЕРВЫЙ СИМВОЛ ВЫРАЖЕНИЯ $s_1 e_2 \sim s_1$

При отождествлении конкретизируемого выражения в первом из двух приведенных примеров свободная переменная s_1 принимает зна-

чение Z, а свободная переменная e_2 – значение $(AB)+F$. В результате замены в поле зрения оказывается символ Z. Во втором примере s_1 принимает значение +, а e_2 – значение пустого выражения.

Для создания языкового объекта, имеющего значение первого символа некоторого выражения, мы использовали русские слова "первый символ выражения" в их натуральном виде. С точки зрения языка M эти слова образуют выражение, состоящее из 23 символов (считая пробелы). Ясно, что это не слишком экономно, а главное, не нужно, ибо эти слова выступают в § 2.1 как единое целое. Поэтому разумно заменить эти 23 символа на один символ. Можно, например, образовать составной символ из букв, которые входят в соответствующие русские слова, причем сократить их, чтобы не писать уж слишком много букв:

§ 3.1 k 'ПЕРВСИМ' $s_1 e_2 \sim s_1$

Допустим, что при наличии в поле памяти одного лишь § 3.1 мы поместили в поле зрения выражение

k 'ПЕРВСИМ'⊥

или выражение

k 'ПЕРВСИМ' $(ABC)+B$⊥

т.е. мы хотим, чтобы машина C_M ответила на вопрос, каков первый символ пустого выражения или первый символ выражения, начинающегося со скобки. И в том и в другом случае § 3.1 окажется неприменимым, и машина C_M придет в состояние аварийной остановки. Она не понимает задачи; понятие 'ПЕРВСИМ' оказалось не вполне определенным. Мы можем дополнить его тем или иным способом. Простейший способ – это объявить, что в обоих сомнительных случаях первый символ есть пустое выражение. Для этого мы должны дополнить описание понятия 'ПЕРВСИМ' еще одним предложением:

§ 3.2 k 'ПЕРВСИМ' $e_1 \sim$

и поместить его в поле памяти после § 3.1. Здесь мы сталкиваемся с очень важным следствием определенного нами алгоритма работы машины C_M: результат конкретизации существенно зависит от порядка расположения предложений в поле памяти. Если поле памяти имеет вид:

§ 3.2 k 'ПЕРВСИМ' $e_1 \sim$

§ 3.1 k 'ПЕРВСИМ' $s_1 e_2 \sim s_1$

то второе предложение (§ 3.1) не будет работать никогда, ибо всегда сработает первое предложение, и "первый символ выражения", определенный таким образом, всегда будет принимать пустое значение. Если же предложения стоят в таком порядке:

§ 3.1 k 'ПЕРВСИМ' $s_1 e_2 \sim s_1$

§ 3.2 k 'ПЕРВСИМ' $e_1 \sim$

то во всех случаях, когда выражение-аргумент начинается с символа, сработает § 3.1. И только когда § 3.1 неприменим, сработает § 3.2.

В таком положении § 3.2 можно прочитать так: "Во всех остальных случаях заменить конкретизируемое выражение на "пусто".

Мы приходим, таким образом, к следующему правилу, касающемуся расположения предложений: сначала – частные случаи, затем – общие предложения.

Допустим, что нас почему-либо интересуют выражения, которые начинаются с десятичной цифры, и мы хотим описать понятие 'ПЕРЦ' таким образом, чтобы конкретизация

k'ПЕРЦ'$\mathscr{E}\perp$

давала первый символ выражения $\mathscr{E}$, если он является цифрой, и пустое выражение, если $\mathscr{E}$ не начинается с цифры* Можно описать это понятие с помощью набора из одиннадцати предложений:

k'ПЕРЦ' $0e_1 \sim 0$

. . . .

k'ПЕРЦ' $9e_1 \sim 9$

k'ПЕРЦ' $e_1 \sim$

Однако ясно, что в этой записи много повторений и ее можно было бы значительно сократить, если ввести переменную, которая может принять в качестве значения любую цифру и только цифру. Такая возможность есть в базисном рефале. Кроме описанных выше простых свободных переменных, разрешается еще использование специфицированных переменных символа, образованных из:

1) признака типа s;

2) спецификатора, который представляет собой либо последовательность символов, взятую в скобки, либо один составной символ;

3) индекса, который, как и в случае простых переменных, является объектным знаком.

Если спецификатор имеет вид ($\mathscr{P}$), где $\mathscr{P}$ – последовательность символов, то свободная переменная может принять в качестве значения только символ, входящий в $\mathscr{P}$. Таким образом, вместо приведенных выше одиннадцати предложений мы можем записать всего два:

k'ПЕРЦ's $(0123456789)_1 e_2 \sim s_1$

k'ПЕРЦ' $e_1 \sim$

В скорописи по-прежнему записываем индекс в нижней позиции – теперь у спецификатора. У свободных переменных, входящих в правую часть предложения, не должно быть спецификатора (ибо в нем, очевидно, нет необходимости).

Если свободная переменная цифры используется несколько раз, то можно избежать многократного выписывания набора цифр с помощью второй модификации спецификатора. Введем составной символ 'ЦИФРА' и запишем предложение

k 'ЦИФРА'$\sim$0123456789

**Заглавные рукописные буквы латинского алфавита будем употреблять в качестве метасимволов для обозначения объектов рефала.*

Теперь мы можем употреблять конструкцию

s 'ЦИФРА'$_1$

в качестве свободной переменной цифры. В общем виде: спецификатор $\mathscr{D}$ (где $\mathscr{D}$ – составной символ) эквивалентен спецификатору (k $\mathscr{D}$1). Однако из этого правила есть несколько исключений, когда составные символы служат для описания потенциально бесконечных классов символов или классов, связанных с конкретной реализацией языка на вычислительной машине. Мы введем два класса символов, независимых от реализации: класс объектных знаков и класс составных символов. Первый описывается спецификатором 'ОБЗН' , второй – спецификатором 'СОСТ'.

В предложении рефала индекс свободной переменной связан с единственной переменной. Поэтому, например, нельзя употреблять в одном и том же предложении переменные e_a и s_a – это будет синтаксической ошибкой. Однако переменная символа может входить в левую часть предложения и как простая, и как специфицированная. Более того, она может иметь и различные спецификаторы. Например, допустимы конструкции:

$s\,(ABC)_1\,e_2\,s_1$

$s_1\,e_2\,s\,(ABC)_1$

$s\,(ABC)_1\,e_2\,s\,(AB)_1$

Поскольку при отождествлении всем вхождениям одной и той же свободной переменной в левую часть сопоставляется одно и то же значение, оно должно быть согласовано со всеми спецификаторами. Поэтому, например, последняя из приведенных конструкций эквивалентна конструкции.

$s\,(AB)_1\,e_2\,s_1$

Подчеркнем, что областью действия свободных переменных в рефале является предложение. Переменная в правой части принимает значение только благодаря процессу отождествления; в котором участвует левая часть. Это значение сохраняется, только пока выполняется данный шаг, и теряется при переходе к следующему шагу. Свободные переменные в разных предложениях могут иметь один и тот же идентификатор, и тем не менее никакой семантической связи между ними не будет.

6. Рекурсивные функции

Человек, который хочет описать на языке M какой-либо алгоритм (например, алгоритм выделения первого символа выражения), должен написать некоторое число предложений для поля памяти машины C_M и некоторое выражение (по существу, начальные данные) для поля

зрения. Затем надо запустить машину C_M, и она выполнит требуемое преобразование. Очевидно, для того, чтобы правильно описать алгоритм, необходимо совершенно ясно представлять, как будет работать машина C_M с данным набором предложений в поле памяти. Это ставит человека в трудное положение, ибо, имитируя работу машины C_M, он должен на каждом шаге просматривать сверху донизу все предложения, стоящие в поле памяти, чтобы найти первое из них, которое окажется применимым. И если число предложений в поле памяти велико, а для сложных алгоритмов оно, несомненно, будет велико, – задача написания алгоритма может оказаться практически неразрешимой. Следовательно, необходим какой-то способ, с помощью которого можно было бы на каждом шаге исключать из рассмотрения все лишние предложения и оставить лишь небольшую, хорошо обозримую группу предложений, имеющих отношение к данному акту конкретизации. Иными словами, надо как-то разбить предложения на небольшие группы, и иметь полную уверенность в их независимости друг от друга.

Способ разрешения этой проблемы напрашивается уже из рассмотрения предложений § 3.1 и § 3.2, описывающих понятие "первый символ выражения", иными словами, алгоритм выделения первого символа выражения. В левой части обоих предложений непосредственно за знаком k следует символ 'ПЕРВСИМ'. Поэтому эти предложения могут оказаться подходящими для конкретизации только в том случае, если в конкретизируемом выражении непосредственно вслед за k стоит тот же символ 'ПЕРВСИМ'. Допустим, что в дополнение к понятию 'ПЕРВСИМ' мы пожелали описать понятие 'ПОСЛСИМ' – выделение последнего символа выражения – с помощью следующих предложений:

§ 4.1 k'ПОСЛСИМ'$e_1 s_2 \sim s_2$

§ 4.2 k'ПОСЛСИМ'$e_1 \sim$

Эту группу предложений можно поместить в поле памяти вместе с группой 'ПЕРВСИМ' в любом положении относительно нее. Если конкретизируемое выражение имеет вид k 'ПЕРВСИМ'$\mathscr{E}\perp$, где $\mathscr{E}$ – произвольное выражение, то можно принимать в расчет лишь группу предложений § 3.1 и § 3.2; если оно имеет вид k 'ПОСЛСИМ' $\mathscr{E}\perp$, то идут в расчет только § 4.1 и § 4.2.

Итак, договоримся, что во всех предложениях непосредственно за знаком k в левой части будет следовать какой-либо символ. Этот символ будем называть *детерминативом* предложения. Тем самым все предложения разбиваются на группы, образованные предложениями с одним и тем же детерминативом. Порядок расположения в памяти групп предложений с различными детерминативами несуществен. Порядок расположения предложений внутри группы, вообще говоря, весьма существен, и мы будем отражать его в нумерации предложений, помещая после точки порядковый номер предложения в группе. Последовательность знаков, предшествующая точке, нумерует группу предложений с данным

детерминативом. Опуская часть номера, начинающуюся с точки, мы ссылаемся на группу предложений в целом. Если у предложения нет комментария, тем не менее можно сослаться на него, указав детерминатив и порядковый номер предложения в группе с данным детерминативом. Так, § *'АБВ'*.3 означает третье предложение с детерминативом *'АБВ'*.

Группа предложений определяет функцию, определенную на некотором множестве выражений и принимающую значение из того же множества. Действительно, пусть у нас есть некоторая группа предложений с детерминативом $\mathscr{F}$ и некоторое выражение $\mathscr{E}$. Поместим в поле памяти машины C_M эту группу предложений, а в поле зрения – выражение

$$k\mathscr{F}\mathscr{E}\perp$$

после чего включим машину C_M. Если машина C_M, совершив определенное число шагов, придет в состояние нормальной остановки (вызванное отсутствием в поле зрения знаков конкретизации), то выражение, оказавшееся в поле зрения, будет значением функции $\mathscr{F}$ при аргументе $\mathscr{E}$. Если машина C_M придет в состояние аварийной остановки, или же вообще не остановится никогда, функция $\mathscr{F}$ при аргументе $\mathscr{E}$ не определена. Обычно в математике функциональной записи $\mathscr{F}(\mathscr{E})$ соответствует в языке *М* запись $k\mathscr{F}\mathscr{E}\perp$. Это вызвано необходимостью отличать функциональные скобки от обычных ("структурных").

Теперь осталось сделать один только шаг, чтобы прийти к понятию рекурсивной функции. Вспомним, как мы решили проблему доопределения понятия 'ПЕРВСИМ', когда оно было определено лишь предложением § 3.1. Мы приняли решение считать первый символ пустым и в том случае, когда выражение пустое, и в том случае, когда оно начинается со скобки. Первое вполне логично, но второе заставляет подумать о более естественном решении. Естественно было бы определить первый символ выражения как первый символ первого подвыражения, не содержащего скобок, т.е. подвыражения, образующего самую левую ветвь на дереве синтаксического анализа скобочной структуры. Например, у выражения $(AB\ (C))D$ первый символ будет A, а у выражения $(((X)))YZ$ первый символ – X. Что же касается выражения $(\ )AB$, то его первый символ, как первый символ пустого подвыражения, будем считать пустым.

Как написать предложение, порождающее нужный нам алгоритм?

Если выражение-аргумент начинается с левой скобки, то мы, очевидно, должны взять подвыражение, которое эта левая скобка – вместе с парной ей правой скобкой – ограничивает, и взять у этого подвыражения первый символ. Соответствующее предложение имеет вид:

$$k\text{'ПЕРВСИМ'}\ (e_1)e_2 \sim k\text{'ПЕРВСИМ'}e_1\perp$$

Тот факт, что e_1 – не произвольная последовательность символов, а обязательно в ы р а ж е н и е, т.е. последовательность, сбалансирован-

ная по скобкам, обеспечивает правильность сопоставления скобок в конкретизируемом выражений скобкам в левой части предложения. Если какая-то скобка в поле зрения отождествлена (поставлена в соответствие) с какой-то скобкой в левой части предложения, то парная ей скобка в поле зрения может отождествляться только с парной скобкой в левой части.

Полный набор предложений, описывающих понятие (функцию) 'ПЕРВСИМ' таков:

§ 5.1 k 'ПЕРВСИМ' $s_1\ e_2 \sim s_1$

§ 5.2 k 'ПЕРВСИМ' $(e_1)\,e_2 \sim k$ 'ПЕРВСИМ' $e_1 \perp$

§ 5.3 k 'ПЕРВСИМ' $\sim$

В определении функции 'ПЕРВСИМ' имеет место – благодаря § 5.2 – рекурсия: при замене левой части предложения на правую функция вызывает саму себя (при другом значении аргумента). Пусть в поле зрения стоит выражение

k 'ПЕРВСИМ' $((ZY)\,{+}{+})\,AB\perp$

На первом шаге будет применен § 5.2, свободная переменная e_1 примет значение $(ZY)\,{+}{+}$, а e_2 – значение AB, и в поле зрения окажется выражение:

k 'ПЕРВСИМ' $(ZY)\,{+}{+}\perp$

После второго шага будем иметь

k 'ПЕРВСИМ' $ZY\perp$

Теперь будет применен § 5.1 с результатом

Z

Функции, определенные с помощью подстановок, включающих, в частности, возможность рекурсии, называются рекурсивными функциями. Подчеркнем, что этот термин относится к способу определения, задания функции. Рекурсия может быть не прямая, а косвенная, через посредство других функций. В следующем примере

$kAe_1 \sim kB(e_1)\perp$

$kB(s_1 e_2) \sim kCe_2\perp$

$kCe_1 s_2 \sim kAe_2\perp$

функция A вызывает функцию B, которая вызывает функцию C, а эта последняя функция снова обращается к A. Таким образом, определение функции A содержит рекурсию.

Требование наличия рекурсии не является обязательным, это общее понятие; функция 'ПЕРВСИМ', определенная предложениями § 3, также относится к множеству рекурсивных функций. Однако для множества рекурсивных функций в целом возможность использования рекурсии имеет решающее значение. Не используя рекурсии, мы не смогли бы, например, описать функцию, которая, подобно функции § 5, обеспечивает вхождение в скобки на любую глубину. Без рекурсии можно написать предложение, обеспечивающее вход в скобки первого уровня:

k'ПЕРВСИМ' $(s_1 e_2) e_3 \sim s_1$
или второго, третьего и т.д. уровней:

k'ПЕРВСИМ' $((s_1 e_2) e_3) e_4 \sim s_1$

k'ПЕРВСИМ' $(((s_1 e_2) e_3) e_4) e_5 \sim s_1$

или, наконец, обеспечить – путем введения в поле памяти всех этих предложений – возможность входа в скобки на глубину, не превышающую заданной; но описать функцию, входящую на любую глубину, без рекурсии невозможно.

Итак, язык M оказался языком рекурсивных функций, определенных на множестве произвольных символьных выражений. С такой областью определения связывается понятие алгоритма – в отличие от классических рекурсивных функций, определенных на множестве натуральных чисел. Поэтому дадим этому языку название "алгоритмический язык рекурсивных функций", сокращенно – РЕФАЛ.

Продемонстрируем близость записи алгоритмов на рефале к обычным рекурсивным определениям, используемым в математике. Возьмем классический пример: определение числа и действий над числами в рекурсивной арифметике. Определение числа основывается на объектной постоянной "нуль" и одноместной функции следования. Нуль будем обозначать, как обычно, знаком 0. Объект, следующий за произвольным объектом a , обозначают обычно через a' . Мы же, чтобы не путать знак ' (штрих) с рефал-кавычкой, будем пользоваться вместо штриха буквой S . Итак, $0S$ будет изображать 1, $0SS$ – 2 и т.д.

Определение числа состоит из следующих трех пунктов:

1. Нуль есть число.

2. Объект, следующий за числом, есть число, иначе говоря, aS есть число, если a – число.

3. Объект, который с помощью первых двух пунктов не может быть квалифицирован как число, не есть число.

Чтобы записать это определение на рефале, мы должны ввести предикат 'ЧИСЛО', т.е. такую функцию, которая принимает значение 'ИСТИНА', если ее аргумент есть число, и значение 'ЛОЖЬ' – в противном случае.

Описание этого предиката на рефале почти буквально повторяет приведенное выше словесное определение:

§ 6.1 k'ЧИСЛО' $0 \sim$ 'ИСТИНА'

§ 6.2 k'ЧИСЛО' $e_a S \sim k$'ЧИСЛО' $e_a \perp$

§ 6.3 k'ЧИСЛО' $e_a \sim$ 'ЛОЖЬ'

Рекурсивное определение сложения состоит из двух пунктов:

1. $+(a, 0) = a$

2. $+(a, bS) = (+(a, b))S$

(записываем сложение как двуместную функцию, сохранив знак + в качестве символа функции).

Описание сложения на рефале имеет вид:

$\S 7.1\ k + e_a, 0 \sim e_a$

$\S 7.2\ k + e_a, e_b S \sim k + e_a, e_b \perp S$

Здесь знак + является детерминативом функции сложения, а аргументы, как и в приведенной выше записи, разделяются запятой.

Предложения § 6 и § 7 дают рефал-машине полную информацию, необходимую для выполнения алгоритмов распознавания числа и сложения чисел. Пусть, например, в поле зрения введено выражение:

$k + 0SS, 0SSS \perp$

Работа рефал-машины может быть описана следующей таблицей.

Таблица 2

Выполнение рефал-машиной операции сложения 2+3=5

Номер шага i	Используемое предложение	Состояние поля зрения после i-го шага
		$k + 0SS, 0SSS\perp$
1	§ 7.2	$k + 0SS, 0SS \perp S$
2	§ 7.2	$k + 0SS, 0S \perp SS$
3	§ 7.2	$k + 0SS, 0 \perp SSS$
4	§ 7.1	$0SSSSS$

Итак, хотя работа рефал-машины определена при любом наборе предложений в ее поле памяти, мы тем не менее накладываем на набор предложений следующие ограничения:

1. В левой части предложения вслед за знаком конкретизации должен стоять символ (детерминатив).

2. Все предложения с одним детерминативом должны следовать непосредственно друг за другом.

Эти ограничения не входят в формальное описание рефала, но они облегчают реализацию языка практически, не убавляя его изобразительной силы, поэтому вводятся во входной язык трансляторов с рефала. Из соображений удобства реализации полезно ввести еще одно ограничение, которого мы будем придерживаться в дальнейшем:

3. Детерминативы предложений могут быть только составными символами, причем составляющая их цепочка объектных знаков должна включать только буквы или цифры и начинаться с буквы (идентификатор в смысле АЛГОЛа).

Для компактной записи предложений в скорописи мы введем еще два правила. Первое разрешает использование греческой буквы вместо конструкции, состоящей из знака конкретизации и следующего

за ним детерминатива. Определения греческих букв могут записываться в отдельной строке в виде равенств, разделенных запятыми, например:

$\alpha = k$'ПЕРВСИМ', $\beta = k$'$Z12$', $\nu = k$'ЧИСЛО'

Имея такие определения и встречая греческую букву, мы заменяем ее на соответствующую конструкцию. Предложения § 6 мы можем записать скорописью следующим образом:

$\nu 0 \sim T$

$\nu e_a S \sim \nu e_a \perp$

$\nu e_a \sim F$

(Здесь, опять в целях сокращения, мы представляем истинностные значения буквами T и F).

У греческих букв разрешается использовать верхние индексы. При расшифровке индекс (который может представлять собой не только один знак, но и последовательность знаков) приписывается к телу составного символа. При наличии приведенного выше определения букв α, β и ν определены также буквы с индексами $\alpha^1 \beta^{abc}$ и т.п. Они расшифровываются как k'ПЕРВСИМ1' и k'$Z12ABC$'. Итак, греческие буквы превращаются, по существу, в обыкновенные функциональные символы. Мы можем использовать их и не приводя соответствующего составного символа. Подразумевается здесь некая стандартная расшифровка, например $\alpha = k$'АЛЬФА' и т.д.

Второе правило разрешает в скорописи опускать знак $\perp$, если он стоит в конце максимального подвыражения, включающего парный ему знак k. Это правило основывается на том обстоятельстве, что знаки $\perp$ в такой позиции могут быть однозначно восстановлены по правилам синтаксиса. Для этого достаточно в конце каждого подвыражения (т.е. в конце всего выражения и перед каждой правой скобкой) дописать столько знаков $\perp$, сколько надо, чтобы получилась синтаксически правильная конструкция. Например, в записи

k'ЧИСЛО' $0SSS$

мы должны приписать в конце один знак $\perp$:

k'ЧИСЛО' $0SSS\perp$

С учетом определения буквы ν мы можем в скорописи писать просто

$\nu 0SSS$

а остальное – дело расшифровки.

Еще несколько примеров. Конструкция

$\alpha(\beta s_a e_1)(\beta F)$

однозначно восстанавливается до

$\alpha(\beta s_a e_1 \perp)(\beta F \perp)\perp$

Запись

$\alpha \beta XYZ$

означает

$\alpha \beta XYZ \perp\perp$

а в выражении

$\alpha XY \perp \beta XYZ \perp$

можно опустить второй знак $\perp$, но никак не первый, иначе это будет воспринято так:

$*XY\beta XYZ\perp\perp$

7. Алгоритм проектирования

Если левая часть предложения содержит более одной свободной переменной выражения, то может случиться, что существует несколько вариантов приписывания свободным переменным значений, которые приводят к отождествлению конкретизируемого выражения с левой частью предложения. Пусть, например, в поле памяти рефал-машины находится предложение

$\S 8.1 * e_1 ; \; e_2 \sim \beta e_1 \perp * e_2 \perp$

а в поле зрения – выражение

$* A1: = A2; \; B1: = B2; \; \text{'GOTO'} L \perp$

Оно может быть отождествлено с левой частью таким образом, что свободные переменные примут следующие значения:

e_1 $A1:=A2$

e_2 $B1:=B2;$ 'GOTO' L

Однако возможен и второй вариант отождествления, при котором значения переменных суть

e_1 $A1:=A2; \; B1:=B2$

e_2 'GOTO' L

Следовательно, необходимо договориться, как поступает рефал-машина при наличии такой неоднозначности. Мы примем следующее соглашение: рефал-машина выбирает тот вариант отождествления, при котором первая слева свободная переменная выражения принимает наиболее короткое значение, а если это не устраняет неоднозначности, то такой же отбор проводится по значению второй переменной выражения, затем третьей и т.д. В нашем примере будет выбран первый вариант.

Приведенная формулировка соглашения кратко выражает результат синтаксического отождествления, и она используется в формальном описании базисного рефала (с небольшим уточнением, о котором будет речь ниже). Однако, чтобы уяснить себе действия рефал-машины, лучше опираться на сам процесс, на алгоритм, который используется при синтаксическом отождествлении объектного выражения $\mathscr{E}$ как типового выражения $\mathscr{L}$ (о б ъ е к т н ы м мы называем выражение, не содержащее ни знаков k , ни свободных переменных; т и п о в ы м – выражение не содержащее знаков k).Этот алгоритм удобнее описывать с обратной точки зрения – как алгоритм п р о е к т и р о в а н и я типового выражения $\mathscr{L}$ на объектное выражение $\mathscr{E}$.

Символы, скобки и свободные переменные назовем э л е м е н т а м и выражений $\mathscr{L}$ и $\mathscr{E}$. Промежутки между элементами и концы вы-

ражений будем называть **узлами**. Если узлу из $\mathcal{L}$ сопоставлен узел из $\mathcal{E}$, будем говорить, что первый узел **спроектирован** на второй. Если в $\mathcal{L}$ узел $\mathcal{K}_1$ расположен левее $\mathcal{K}_2$, то в $\mathcal{E}$ проекция $\mathcal{P}_1$ узла $\mathcal{K}_1$ не может находиться правее проекции $\mathcal{P}_2$ узла $\mathcal{K}_2$. Если $\mathcal{P}_1$ и $\mathcal{P}_2$ – проекции концов некоторого элемента, то выражение, ограниченное узлами $\mathcal{P}_1$ и $\mathcal{P}_2$, назовем **проекцией элемента**. Проекции свободных переменных (**значения**) должны удовлетворять требованиям, предъявляемым к значениям. Проекции остальных элементов должны быть тождественными самим элементам. Предполагается, что перед началом процесса концы выражения $\mathcal{L}$ спроектированы на концы выражения $\mathcal{E}$. Пример проектирования показан на рисунке 1.3.

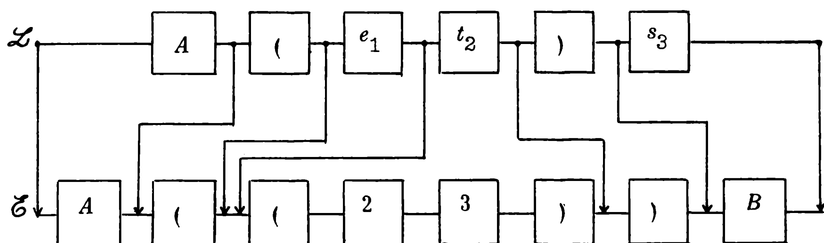


Рис.1.3

Символы, скобки, свободные переменные символа и терма назовем **жесткими** элементами. Если у такого элемента спроектирован один конец, то второй конец проектируется однозначно, либо же делается вывод о невозможности дальнейшего проектирования (**ситуация тупика**). По отношению к первым трем типам элементов это очевидно. Что касается свободной переменной терма, то она может проектироваться либо на символ, либо на выражение в скобках. В последнем случае парная скобка находится в выражении $\mathcal{E}$ однозначно по правилам скобочного синтаксиса. Можно считать, что парные скобки (и в $\mathcal{L}$, и в $\mathcal{E}$) как бы соединены между собой, так что найдя одну скобку, мы тут же находим и парную ей. Поэтому спроектировав одну скобку, мы тут же можем спроектировать и другую.

На всех стадиях проектирования действует общий принцип: если спроектирован один из концов жесткого элемента, то делается попытка спроектировать и другой конец, т.е. элемент в целом. Порядок проектирования, очевидно, не существен, однако для определенности примем следующее соглашение: если какая-либо скобка получила проекцию, немедленно проектируется парная ей скобка; в остальном же раньше проектируются те элементы, которые располо-

жены левее. Если типовое выражение $\mathscr{L}$ состоит из одних лишь жестких элементов, то приведенное правило полностью определяет проекционные номера элементов и весь алгоритм проектирования. Приведем в качестве примера следующее выражение, где над элементами надписаны их проекционные номера:

$$\overset{1}{B}\ \overset{2}{(}\ \overset{4}{s_1}\ \overset{5}{A}\ \overset{6}{(}\ \overset{8}{t_2}\ \overset{9}{C}\ \overset{10}{D}\ \overset{7}{)}\ \overset{3}{)}$$

Здесь, если проектирование какого-либо из элементов оказывается невозможным, невозможно и проектирование (отождествление) в целом.

Займемся теперь свободными переменными выражения. Скажем заранее, что каждому вхождению свободной переменной в $\mathscr{L}$ будет соответствовать, как каждому из уже рассмотренных элементов, совершенно определенный проекционный номер. Если некоторая свободная переменная входит в $\mathscr{L}$ несколько раз, то вхождение с наименьшим проекционным номером назовем г л а в н ы м, а остальные вхождения – п о в т о р н ы м и. Это определение относится, конечно, и к свободным переменным символа и терма, но для этих элементов оно не было связано с порядком проектирования. Проектирование повторного вхождения переменной символа или терма отличается от проектирования главного вхождения только дополнительной проверкой на совпадение проекции с проекцией соответствующего главного вхождения. Для вхождения свободной переменной выражения тот факт, что оно является повторным, весьма существен, ибо дает возможность по проекции одного конца определить проекцию другого конца, т.е. превышает повторное вхождение в ж е с т к и й элемент. Поэтому повторные вхождения всех свободных переменных мы причислим к жестким элементам и распространим на них сформулированные выше правила.

Для главных вхождений свободных переменных выражения рассмотрим два случая:

1. В результате предшествовавших актов проектирования оба конца переменной получили проекции. Такие переменные (точнее, главные вхождения) называются з а к р ы т ы м и. Никаких проверок в этом случае не делается, и переменная автоматически получает проекцию и очередной проекционный номер.

2. Левый конец переменной спроектирован, а жестких элементов со спроектированным концом больше нет. Правый конец переменной выражения как бы повисает в воздухе. Такие вхождения называются о т к р ы т ы м и переменными выражения. Проектирование их производится так. Сначала правый конец проектируется на тот же узел, что и левый конец, т.е. переменная получает пустое значение и процесс проектирования продолжается. Как только возникает тупиковая ситуация, мы возвращаемся к последней открытой переменной, которая получила значение и у д л и н я е м ее значение, т.е. переносим про-

екцию правого конца на один терм вправо и с этого места продолжаем проектирование, присваивая элементам $\mathscr{L}$ новые проекции. Если удлинение последней открытой переменной невозможно, то удлиняется предпоследняя и т.д. Если в тупиковой ситуации ни одной переменной удлинить нельзя, то отождествление невозможно.

Легко понять, что если на некотором уровне скобочной структуры есть всего одна свободная переменная выражения, то она будет закрытой, а если из n (учитывая лишь главные вхождения), то первые $n-1$, будут открытыми, а последняя – закрытой. Возьмем такой пример:

$$\begin{matrix} 1 & 4 & 3 & 5 & 6 & 8 & 7 & 9 & 10 & 11 & 2 \\ A & e_1 & (& s_a & (& e_2 &) & t_b & = & e_3 &) \end{matrix}$$

По проекционным номерам легко восстановить последовательность действий при проектировании. Все три переменные выражения здесь закрытые. Они получают значения путем ограничения частей выражения $\mathscr{E}$ слева и справа в процессе движения по жестким элементам, как по мостикам. Символы и термы дают возможность сделать один шаг влево или вправо, а скобки – осуществить "большой скачок".

Теперь рассмотрим типовое выражение, образующее левую часть § 8.1:

$$\begin{matrix} 1 & 2 & 3 & 4 \\ \text{'АЛЬФА'} & e_1 & ; & e_2 \end{matrix}$$

и его проектирование на конкретизируемое выражение

$$\text{'АЛЬФА'}\, A1 := A2;\ B1 := B2;\ \text{'GOTO'}\, L$$

После проектирования символа 'АЛЬФА' мы оказываемся в положении, когда нет жестких элементов, готовых к проектированию. Действительно, у элемента 3 не спроектирован ни один из концов. У свободной переменной e_2 не спроектирован левый конец, поэтому она не удовлетворяет ни одному из приведенных выше двух условий и также не готова к проектированию. Переменная e_1 – открытая. Придаем ей пустое значение и продолжаем проектирование. Теперь элемент 3 готов к проектированию, но проектирование невозможно, так как в конкретизируемом выражении на нужном месте стоит не точка с запятой, а знак A . Констатируем тупик и удлиняем переменную e_1, придавая ей значение A . Снова констатируем тупик и снова удлиняем e_1 и т.д., пока e_1 не примет значения $A1 := A2$. Теперь проектирование элемента 3 заканчивается успешно, и мы видим, что элемент 4 – закрытая переменная. К ней отходит оставшаяся часть конкретизируемого выражения, и проектирование (отождествление) заканчивается.

Чтобы завершить описание проектирования, нам осталось только рассмотреть те случаи, когда на каком-то этапе возникает несколь-

ко конкурирующих между собой открытых или закрытых переменных, например:

$$\overset{1}{(} e_1 + e_2 \overset{2}{)} e_3 + e_4 \overset{4}{(} e_5 + e_6 \overset{3}{)}$$

или

$$e_1 \overset{2}{(} e_2 \overset{1}{)}$$

Мы будем руководствоваться общим принципом: меньший проекционный номер получает тот элемент, который расположен левее. Поэтому распределение номеров будет следующим:

$$\overset{1}{(}\ \overset{5}{e_1}\ \overset{6}{+}\ \overset{7}{e_2}\ \overset{2}{)}\ \overset{8}{e_3}\ \overset{9}{+}\ \overset{10}{e_4}\ \overset{4}{(}\ \overset{11}{e_5}\ \overset{12}{+}\ \overset{13}{e_6}\ \overset{3}{)}$$

$$\overset{3}{e_1}\ \overset{2}{(}\ \overset{4}{e_2}\ \overset{1}{)}$$

Рассмотрим еще один пример:

$$\overset{10}{e_1}\ \overset{11}{+}\ \overset{12}{e_2}\ \overset{2}{(}\ \overset{3}{t_3}\ \overset{4}{(}\ \overset{6}{*}\ \overset{7}{*}\ \overset{5}{)}\ \overset{9}{e_1}\ \overset{8}{B}\ \overset{1}{)}$$

На первый взгляд может показаться, что первое вхождение переменной e_1 – открытое. В действительности же оно повторное. Проектируя жесткие элементы, мы добираемся до вхождения e_1 (элемент 9), которое является закрытым. Теперь элемент 10 становится жестким и все проектирование заканчивается без удлинений. Но просмотр здесь, конечно, есть. Проектируя повторное вхождение, просматриваем его проекцию одновременно с проекцией главного вхождения.

В описании процесса отождествления (проектирования), как оно велось до настоящего момента, имела место существенная неэквивалентность направлений просмотра выражений $\mathcal{L}$ и $\mathcal{E}$ – основным было направление слева направо. Для отождествления в обратном направлении в базисном рефале вводится следующая возможность: если непосредственно перед знаком k в левой части вписать терм (R), то основным становится направление справа налево, иначе говоря, в описании процесса проектирования надо всюду слова "левый" и "правый" (вместе с их производными) поменять местами. Это скажется на значениях открытых переменных: теперь наиболее короткие значения будут принимать переменные, расположенные правее. Предложение с левой частью

$$(R) \cdot e_1 ; e_2 \sim$$

будет находить первую точку с запятой при движении справа налево (не считая, конечно, тех, которые входят в скобки, ибо e_2 должно проектироваться на выражение).

8. Общие принципы реализации рефала

Чтобы выполнять с помощью электронной вычислительной машины алгоритмы, описанные на рефале, надо написать программу, имитирующую работу рефал-машины. Входной информацией для этой программы будет набор предложений, предназначенный для поля памяти рефал-машины, (алгоритм) и выражение, помещаемое перед началом работы в поле зрения (начальные данные). Если набор предложений сохраняет в процессе работы вычислительной машины свой натуральный вид или претерпевает лишь небольшую модификацию, то такая программа называется, по сложившейся терминологии, интерпретатором для данного языка (рефала). Если набор предложений подвергается перед началом работы существенной переработке, т.е. рассматривается, по существу, как информация для создания нового текста на машинном или машинно-ориентированном языке, то такая программа носит название компилятора. Интерпретаторы и компиляторы объединяют под общим наименованием трансляторов с данного языка. В настоящем разделе изложим основные принципы, на которых построены ныне действующие трансляторы с рефала (см. [16]). Эти принципы общие для интерпретаторов и компиляторов и касаются, главным образом, представления в реальной вычислительной машине информации, образующей содержимое поля зрения абстрактной рефал-машины. Это представление всегда одинаково и не зависит от программы на рефале. Заметим, что в настоящее время разрабатывается такая реализация, при которой представление поля зрения будет существенно зависеть от алгоритма, являясь результатом его глубокого анализа. Это позволит в дальнейшем существенно повысить эффективность работы рефал-транслятора.

Действия, совершаемые рефал-машиной над содержимым поля зрения, заключаются в удалении одних подвыражений и замене их другими. Поэтому информацию в поле зрения надо хранить таким образом, чтобы эти действия не приводили к ненужной переписи тех частей поля зрения, которые не затрагиваются непосредственно данной подстановкой. Это достигается с помощью так называемой списковой структуры памяти, отведенной под поле зрения. Такая структура позволяет избежать и других необязательных для выполнения алгоритма действий (просмотров), которые входят в формальное описание абстрактной рефал-машины. В рефал-трансляторе списковая структура выглядит следующим образом.

Основной единицей структуры является звено; память, отведенная под поле зрения, представляет собой совокупность некоторого числа звеньев. Звено - это группа двоичных разрядов, допускающая прямую адресацию. В машинах с достаточно большой ячейкой основной памяти звеном может являться одна ячейка. Так обстоит дело,

например, в случае машин М-20, БЭСМ-6 и др. В машинах серии ИБМ-360 в качестве звена может служить последовательность из шести или восьми байтов. Звено состоит из четырех полей: поля признака π и трех адресных полей A_1, A_2, A_3 (см.рис.1.4). Размер

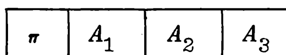


Рис.1.4

адресных полей должен быть таким, чтобы они могли содержать адрес любого звена, входящего в поле зрения. Поля A_2 и A_3 служат для связывания звеньев в линейную последовательность; поле A_2 всегда содержит адрес предыдущего звена в этой последовательности, поле A_3 содержит адрес следующего звена. Содержимое поля A_1 зависит от значения признака π , которое в свою очередь зависит от того, какому рефал-объекту соответствует данное звено. Звено может изображать либо символ, либо скобку. Кроме того, при различных реализациях может быть введено несколько типов символов (см.ниже) и для их распознавания отведено несколько разрядов в поле признака.

Если звено изображает символ, то поле A_1 содержит код этого символа. Если звено изображает скобку, то поле A_1 содержит адрес парной скобки.

Благодаря такому представлению информации становится возможным осуществлять весьма значительные преобразования текста (вставки, перестановки и вычеркивания), не переписывая частей текста, а лишь изменяя адреса связи. Пусть, например, в поле зрения находится выражение $m(ab+cd)$, представленное восемью подряд идущими звеньями в оперативной памяти вычислительной машины (см.рис.1.5). Допустим, что нам надо преобразовать это выражение в $m(cd)ab$. Для этого достаточно "перешить" цепочку ab и удалить символ +. В результате получается структура, показанная на рис.1.6. Ясно, что, если бы вместо цепочек ab и cd были сколь угодно длинные цепочки, число операций, необходимых, чтобы выполнить преобразование, осталось бы тем же самым.

Звено, содержащее символ +, оказалось лишним, на рис.1.6 оно лишено связей. Это звено подшивается к "свободной памяти" - цепочке звеньев, не несущих в данный момент информации о поле зрения. Если в выражение, содержащееся в поле зрения (в дальнейшем мы это выражение будем для краткости называть просто полем зрения), требуется сделать вставку, то необходимые звенья, уже связанные в цепочку, отщепляются от той же "свободной памяти". Итак, звенья, отведенные под поле зрения, всегда образуют две непрерывные цепочки: одна изображает собственно поле зрения, другая пред-

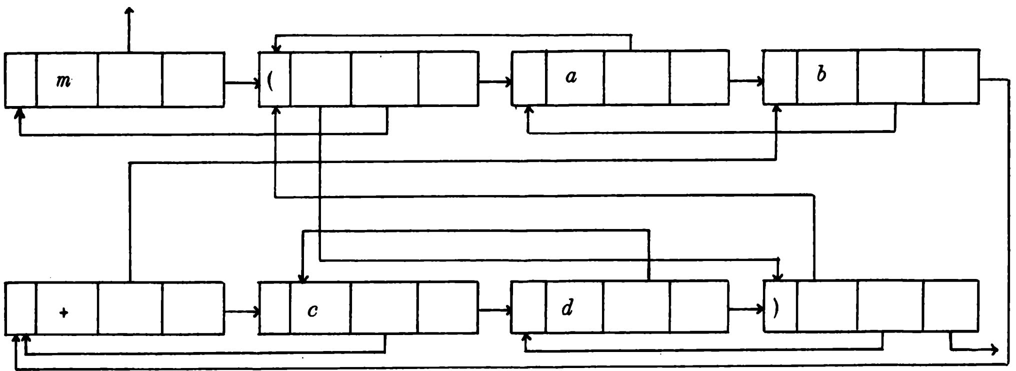


Рис.1.5

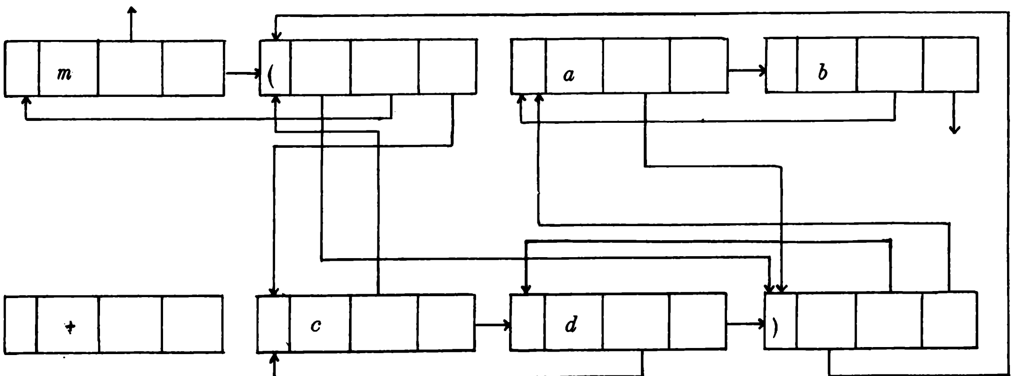


Рис.1.6

ставляет свободную память. Действия транслятора над полем зрения сводятся к перешивке частей внутри первой цепочки и обмену частями между первой и второй цепочками. В результате этих действий физическое расположение звеньев в памяти машины теряет всякую связь (если она существовала вначале) с логическим порядком звеньев, отражающим порядок расположения объектов в поле зрения абстрактной рефал-машины.

Благодаря тому, что в звене, изображающем скобку, хранится адрес парной скобки, становится возможным непосредственное проектирование парной скобки в процессе отождествления без просмотра содержимого этой пары скобок, а путем простого перехода по адресу. Становится также возможным отождествление терма, начинающегося со скобки, без просмотра содержимого скобок.

Теперь коснемся представления в поле зрения конкретизационных скобок. Согласно формальному описанию языка, рефал-машина должна в начале каждого шага просматривать поле зрения в поисках ведущего знака конкретизации k . Ясно, что буквальное выполнение этого требования повлечет серьезную потерю эффективности транслятора. К счастью, определение ведущего знака k позволяет избежать ненужных просмотров поля зрения. Достигается это следующим образом.

Если в выражение входит более одного знака k , то для них можно ввести отношение порядка, которое мы назовем старшинством. Самым младшим знаком k объявим ведущий знак k . Следующим по старшинству будет тот знак k , который станет ведущим, если удалить из выражения ведущий знак k и парную ему конкретизационную точку. Продолжая эту процедуру, распределим по старшинству все знаки конкретизации. Для каждого знака k определяем непосредственно следующий за ним по старшинству знак k как тот знак, который станет ведущим, если удалить данный знак k и все младшие знаки k вместе с парными точками.

Итак, все знаки конкретизации в поле зрения можно связать в непрерывную цепь стрелками, которые от каждого знака k ведут к следующему за ним по старшинству. Приведем такой пример.

$$k\ 'A'BBk'B'(XY)\perp k'C'(YX)k\ 'D'\perp\perp B\perp$$

Чтобы зафиксировать эту цепь в трансляторе, надо с каждым знаком конкретизации связать четыре адреса: адреса предыдущего и последующего звеньев, адрес парной точки и адрес следующего по старшинству знака конкретизации. Возможны различные способы осуществления этого в машине; мы не будем на них останавливаться. Так или иначе, но в трансляторе должны храниться адреса знаков k в поле зрения, связанные друг с другом в цепь по старшинству. Пусть в поле зрения знаки k связаны в цепь по старшинству. Если терм, начинающийся с ведущего знака k, заменить на любое рабочее выражение, в котором знаки k также связаны в цепь по стар-

шинству, и связать старший знак k вставляемого выражения с тем знаком k, который по старшинству непосредственно следовал за ведущим знаком k в поле зрения, то и результирующее выражение будет обладать правильной цепью по старшинству знаков k. Отсюда следует такой алгоритм выбора транслятором ведущего знака k:

1. В начале шага по адресу ведущего знака конкретизации определяется адрес следующего за ним по старшинству знака конкретизации.

2. Если в правой части используемого предложения нет знаков k, то знак k, найденный в п.1, объявляется ведущим.

3. Если в правой части используемого предложения есть знаки k, то в выражении, порождаемом в поле зрения как копия правой части, знаки k связываются в цепь по старшинству. (Легко понять, что это требует числа операций, зависящего лишь от вида правой части предложения, но не от вида и размеров значений свободных переменных).

4. Старший знак k в полученном выражении связывается со знаком k, найденным в п.1. Младший знак k становится ведущим.

Займемся теперь кодами символов. Объектные знаки при реализации рефала интерпретируются как группы двоичных разрядов, имеющих, как правило, определенное внешнее представление на устройствах ввода-вывода. Так как одна и та же машина может быть оборудована различными устройствами ввода-вывода, желательно, чтобы кодом объектного знака могла быть любая последовательность двоичных разрядов фиксированной длины. Для машин с байтовой структурой памяти код объектного знака - это произвольный байт (восемь двоичных разрядов).

Одна из основных задач составных символов - служить детерминативом группы предложений. При реализации удобно сопоставить детерминативу адрес того участка поля памяти, который содержит данную группу предложений, а еще лучше - просто закодировать детерминатив этим адресом. Тогда, выбрав код детерминатива вслед за ведущим знаком k, мы сразу находим нужные нам предложения. Говоря программистским языком, код детерминатива функции будет адресом перехода на подпрограмму этой функции. Так как на промежуточных этапах трансляции используется обычно ассемблер, удобно рассматривать тело детерминатива как метку - отсюда то ограничение на детерминативы, которое мы ввели в разд.6: его тело должно иметь вид алгольного идентификатора. Мы получаем, таким образом, определенный класс составных символов; будем называть их *символы - метки*. Их можно использовать не только в качестве детерминативов функций, а также в качестве разделителей и т.п. В отличие от кода объектного знака, код символа-метки принимает значение конкретного адреса лишь перед началом работы машины и зависит от множества случайных обстоятельств.

Простейший способ представления чисел при программировании на рефале - это в виде последовательности знаков, изображающих число в десятичной системе счисления, как установлено, например, в АЛГОЛе. При таком представлении число, с точки зрения рефал-транслятора, ничем не отличается от любой другой последовательности объектных знаков. Однако этот способ весьма расточителен и в отношении памяти машины, и в отношении времени, затрачиваемого на арифметические действия. Так, число 1971 будет занимать четыре полных звена. При действиях над так представленными числами надо по очереди выбирать знаки из памяти, после чего либо моделировать десятичную арифметику, либо тратить время на перевод в двоичную систему и обратно.

Для экономного представления чисел вводим символы-числа (или числовые символы). С точки зрения абстрактного рефала, фиксированного в формальном описании, символы-числа - это составные символы специального вида, а именно, с телом, образованным последовательностью десятичных цифр, например: '0', '1', '514', '1378993'. При реализации числовой символ представляется звеном со специальным признаком π, код которого (т.е. содержимое поля A_1) есть двоичная запись числа, образующего тело составного символа. Содержательно - это неотрицательное целое число. Если в звене на запись числа отводится n разрядов, то максимальным числом, представимым в виде одного символа, будет $2^n - 1$. Целые числа, больше чем $2^n - 1$, можно представить в виде последовательности числовых символов, где каждый предыдущий символ имеет вес в 2^n раз больший, чем следующий, иначе говоря, в 2^n-ичной системе счисления. Числовые символы - это, в сущности, цифры ("макроцифры") в 2^n-ичной системе счисления. В этой же системе счисления можно представить и дробные числа, например:

$$'356'.'1587' = '356' + '1587' \times 2^{-n}$$

Но как же выполнять действия над числовыми символами? В абстрактной рефал-машине специальных функций для этого не предусмотрено (такое решение диктуется хотя бы тем, что разрядность n числовых символов может быть разной при разных реализациях). Эти функции определяются на уровне реализации как "машинные процедуры".

При описании абстрактной рефал-машины предполагается, что мы можем непрерывно следить за состоянием поля памяти и поля зрения. При воплощении рефал-машины на реальной вычислительной машине это, разумеется, не так. Уже поэтому необходимы какие-то процедуры, позволяющие выводить из машины информацию по мере необходимости, не прекращая работу рефал-транслятора. Может оказаться также полезным вводить какую-то информацию в рефал-машину в процессе ее работы, Далее, при обработке больших массивов информации необходимо осуществлять обмен между оперативной памятью и внешними запоминающими устройствами: барабанами, дисками, лента-

ми. Наконец, при выполнении некоторых преобразований (например, арифметических действий) имеет смысл забыть на время о языке рефал и передать управление программе, написанной в кодах вычислительной машины. Все эти действия выполняются при реализации рефала с помощью машинных процедур. Так как машинных процедур может быть сколько угодно, и даже те из них, которые необходимы практически всегда (например, печать), могут иметь различные модификации, ни одна из этих процедур не включена в описание абстрактной рефал-машины. Однако возможность использования машинных процедур является непременной и существенной чертой системы программирования на рефале.

Каждой машинной процедуре, подобно процедуре, описанной на рефале, соответствует определенный составной символ – детерминатив. Предполагается, что транслятор содержит список детерминативов всех задействованных машинных процедур. Машинная процедура выполняется тогда, когда ведущим термом в поле зрения становится терм вида

$$k \mathscr{M} \mathscr{E} \perp$$

где $\mathscr{M}$ – детерминатив данной машинной процедуры, а $\mathscr{E}$ произвольное выражение (содержащее необходимую информацию). При этом управление передается на некоторую подпрограмму, заданную набором кодов вычислительной машины. С точки зрения абстрактной рефал-машины выполнение шага этим заканчивается, а дальнейшее преобразование информации и возвращение к нормальному началу следующего шага рефал-машины возлагается на программу машинной процедуры.

Аппарат машинных процедур служит для передачи управления от абстрактной рефал-машины к конкретной вычислительной машине и обратно. Он позволяет сочетать использование рефала с использованием всех других средств программирования, которые имеются в распоряжении на данной машине.

Набор машинных процедур – дело конкретной реализации рефала. Однако несколько машинных процедур можно считать стандартными и необходимыми при любой реализации.

Это, прежде всего, машинная процедура печати с детерминативом 'ПЕЧ'. Когда ведущим становится терм

$k\,'\text{ПЕЧ}'\,\mathscr{E} \perp$

рефал-транслятор печатает выражение $\mathscr{E}$ и уничтожает в поле зрения функциональные скобки с детерминативом, оставляя вместо конкретизируемого выражения одно выражение $\mathscr{E}$. Мы определяем процедуру печати таким образом для того, чтобы в процессе работы рефал-машины можно было печатать выражения, не портя этих выражений. Часто бывает нужна и другая процедура – выводящая выражение на печать и уничтожающая его в поле зрения. Этой процедуре присвоим стандартный детерминатив 'ВЫВ' (т.е. "вывести из поля

зрения на печать") и опишем ее на рефале, используя процедуру 'ПЕЧ':

k 'ВЫВ' e_1 ~ k 'СТЕРЕТЬ' k 'ПЕЧ' e_1 ⊥ ⊥
k 'СТЕРЕТЬ' e_1 ~

Далее, необходима хотя бы одна машинная процедура, позволяющая вводить информацию в процессе выполнения алгоритма. Она может иметь детерминатив 'ВВОД', а в качестве аргумента - указание об источнике. Выражение

k 'ВВОД' § ⊥

в результате выполнения конкретизации заменяется в поле зрения на вводимую информацию. Возможно объединение детерминатива 'ВВОД' с аргументом § в несколько различных процедур ввода с пустым аргументом.

Наконец, если реализация допускает числовые символы, то необходимы машинные процедуры, выполняющие арифметические действия над ними. Описания этих процедур мы откладываем до описания реализаций. Но кроме действий над числовыми символами, полезно еще уметь распознать их. Спецификатор 'СОСТ' свободной переменной символа относится ко всему классу составных символов, так что переменная s 'СОСТ'$_1$ может иметь значением как символ-метку, так и символ-число. Чтобы различать эти два подкласса, в реализациях вводят "машинные спецификаторы" 'СИМЕТ' и 'СИЧИС'. Как и машинные процедуры, машинные классы символов не входят в абстрактный рефал, образуя его расширение.

Число и характер машинных процедур и спецификаторов ничем не ограничивается. Возможна реализация "в машинном виде" и таких процедур (спецификаторов), которые допускают описание на рефале, например - с целью повышения эффективности. Используя соответствующие составные символы, программист может не знать, как будет выполняться конкретизация. При использовании разных трансляторов, и даже при различных запусках одного и того же транслятора, конкретизация может быть выполнена либо в результате перехода на программу, написанную в коде машины, либо путем использования предложений, написанных на рефале.

При вводе в вычислительную машину вся информация предстает в виде последовательности байтов, т.е., с точки зрения рефала, объектных знаков. Поэтому необходимо договориться, как изображать выражения и предложения рефала последовательностями объектных знаков. Этот способ изображения (м е т а к о д) вместе со способом размещения информации на бланках будет дан при описании конкретных реализаций рефала.

9. Закапывание и выкапывание

Базисный рефал содержит еще один аппарат, описание которого мы отложили до настоящего момента, так как он тесно связан с реализацией языка на вычислительных машинах. Это аппарат *закапывания и выкапывания.* Поле памяти и поле зрения рефал-машины не разделены непреодолимой стеной; для перемещения информации из одного поля в другое служат две стандартные функции с детерминативами 'ЗК' ("закопать") и 'ВК' ("выкопать"). Как выполняются эти функции на уровне абстрактного рефала, описано в разд.10. Здесь мы опишем их выполнение в плане реализации на вычислительной машине.

Обращение к процедуре закапывания имеет вид

k'ЗК'$\mathcal{C} = \mathcal{E}\perp$

где $\mathcal{C}$ – символ, а $\mathcal{E}$ – произвольное выражение. Это обращение может быть прочитано, как "закопать выражение $\mathcal{E}$ под именем $\mathcal{C}$". Выполняется эта процедура следующим образом. Звено в поле зрения, предшествующее ведущему знаку k, соединяется со звеном, следующим за парной функциональной точкой, так что все конкретизируемое выражение исключается из поля зрения. Однако при этом звенья, фиксирующие выражение $\mathcal{E}$, остаются нетронутыми; выражение $\mathcal{E}$ исчезает из поля зрения абстрактной рефал-машины, но остается в памяти реальной вычислительной машины в том самом месте и в том самом виде, в котором оно было прежде. Символ $\mathcal{C}$ вместе с адресами начального и конечного звеньев выражения $\mathcal{E}$ запоминается в специальной области памяти, отведенной для процедур 'ЗК' и 'ВК'. Транслятор как бы закапывает выражение $\mathcal{E}$ и забивает над ним колышек, помеченный именем $\mathcal{C}$.

Обращение к процедуре выкапывания имеет вид

k'ВК'$\mathcal{C}\perp$

и читается: "Выкопать последнее выражение, закопанное под именем $\mathcal{C}$". Выполняется процедура 'ВК' следующим образом. Из области памяти, отведенной для процедур 'ЗК' и 'ВК', выбираются адреса, связанные с последним закапыванием под именем $\mathcal{C}$ и с их помощью конкретизируемое выражение заменяется на закопанное выражение. Тройка, состоящая из имени $\mathcal{C}$ и адресов двух концов закопанного выражения, стирается, так что следующее выкапывание с тем же именем $\mathcal{C}$ даст выражение, закопанное предпоследним. Если от транслятора требуют выкопать выражение, указывая имя, под которым в данный момент ничего не закопано (в частности, было закопано, но уже выкопано), то происходит аварийная остановка.

Закапывание есть в сущности, присваивание имени $\mathcal{C}$ значения $\mathcal{E}$. Выкапывание же отличается от обычной замены имени $\mathcal{C}$ на его значение тем, что здесь отсутствует дублирование, снятие копии со значения $\mathcal{E}$, так что в результате выкапывания $\mathcal{E}$ оно перестает

быть значением имени $\mathcal{C}$. Введение такого аппарата в качестве основного связано со стремлением избежать дублирования выражений $\mathcal{E}$ (которые могут быть весьма громоздкими), когда это не нужно по смыслу алгоритма. Дублирование, если в нем есть необходимость, надо описать в явном виде.

При практическом использовании рефала важно, что машинная реализация процедур 'ЗК' и 'ВК' так, как это описано выше, приводит к весьма быстрому их выполнению, не требующему переписи и даже просмотра закапываемых и выкапываемых выражений. Это следует учитывать при составлении программ на рефале. С другой стороны, закопанные выражения продолжают занимать место в памяти, отведенной под поле зрения, и это может вызвать нехватку памяти. Поэтому процедуры 'ЗК' и 'ВК' рекомендуется использовать лишь для оперативного запоминания, а для более долговременного запоминания информации использовать машинные процедуры, которые хранят ее в сжатом виде или выносят на внешние запоминающие устройства.

В скорописи комбинация k 'ЗК' может быть заменена на язык ↓ , а комбинация k 'ВК' – на знак ↑ .

Пример. Допустим, что нам надо хранить список "выделенных" букв, который время от времени будет использоваться, а также каким-то образом изменяться (например, пополняться). Введем имя 'СПИС', под которым будем закапывать этот список. Начальное формирование списка выделенных букв может быть осуществлено конкретизацией:

k'ЗК' 'СПИС' = АБВГ ⊥

Чтобы приписать справа к списку выделенных букв три буквы КЛМ, надо выполнить конкретизацию

k 'ЗК' 'СПИС' = k 'ВК' 'СПИС'⊥ КЛМ ⊥

Допустим, надо удалить из списка букву В. Для этого сначала определим процедуру (функцию) удаления произвольной буквы 'УДАЛ'. В аргументе этой функции будем помещать в скобки выражение, из которого букву надо удалять.

k 'УДАЛ' $s_a (e_1 s_a e_2) \sim e_1 e_2$

k 'УДАЛ' $s_a (e_1) \sim e_1$

Второй параграф относится к случаю, когда нужной буквы в выражении нет.

Удаление буквы В из списка букв достигается конкретизацией

k'ЗК' 'СПИС' = k 'УДАЛ' В (k 'ВК' 'СПИС' ⊥) ⊥⊥

Поставим задачу описать предикат 'ВЫДЕЛ', определяющий, является ли данная буква "выделенной", т.е. входит ли она в 'СПИС'. Здесь необходима вспомогательная функция 'ПРОВ', проверяющая, входит ли буква в данное выражение.

$\alpha = k$ 'ВЫДЕЛ', $\beta = k$ 'ПРОВ'
$\alpha s_a \sim \beta s_a \sim (\uparrow$ 'СПИС')
$\beta s_a (e_1 s_a e_2) \sim T\downarrow$ 'СПИС' $= e_1 s_a e_2$
$\beta s_a (e_1) \sim F\downarrow$ 'СПИС' $= e_1$

Процедура 'ПРОВ', после того как она использует список выделенных букв, снова закапывает его под именем 'СПИС', иначе первое же использование предложений § 'ПРОВ' привело бы к безвозвратной утере списка.

10. Формальное описание базисного рефала

1. Синтаксис

Значительную часть синтаксиса мы опишем в бэкусовской нормальной форме

1.1. Знаки

<знак> : : = <собственный знак> | <объектный знак>
<собственный знак> : : = § | <скобка> | <признак типа переменной>
<скобка> : : = <символьная скобка> | <структурная скобка> | <конкретизационная скобка>
<символьная скобка> : : = '
<структурная скобка> : : = (|)
<конкретизационная скобка> : : = k | $\perp$ | $\sim$
<признак типа переменной> : : = s | t | e

Объектным знаком может быть любой знак, отличный от собственного знака. Предполагается, что алфавит объектных знаков конечен, хотя он и не фиксируется в описании языка.

1.2. Символы и выражения

<символ> : : = <объектный знак> | <составной символ>
<составной символ> : : = '<тело составного символа>'
<тело составного символа> : : = <объектная цепочка>
<объектная цепочка> : : = <объектный знак> | <объектная цепочка> | <объектный знак>
<выражение> : : = <пусто> | <выражение> <терм>
<пусто> : : =
<терм> : : = <символ> | <свободная переменная> | (<выражение>) | k <выражение> $\perp$
<свободная переменная> : : = <простая переменная> | <специфицированная переменная>
<простая переменная> : : = <признак типа переменной> <индекс>
<индекс> : : = <объектный знак>
<специфицированная переменная> : : = s <спецификатор> <индекс>

<спецификатор> : : = (<объектная цепочка>) | <составной символ>

Выражение, которое не содержит знаков конкретизации (но, вообще говоря, содержит свободные переменные), будем называть **типовым** выражением. Выражение, не содержащее свободных переменных (но, вообще говоря, содержащее знаки конкретизации), будем называть **рабочим** выражением. Выражение, которое не содержит ни свободных переменных, ни знаков конкретизации, называется **объектным** выражением. В следующей таблице перечислены частные случаи понятия выражения и указано, какие знаки они могут содержать (+) и какие не могут (–).

Таблица 3

Соответствие между видом и составом выражений

Вид выражения	*s t e*	*k* ⊥	' ( )	Объектные знаки
Объектная цепочка	–	–	–	+
Объектное выражение	–	–	+	+
Рабочее выражение	–	+	+	+
Типовое выражение	+	–	+	+
Выражение	+	+	+	+

1.3. Предложения

<предложение> : : = § <комментарий> <указатель реверса> <левая часть> <правая часть>

<комментарий> : : = <объектная цепочка> | <ПУСТО>

<указатель реверса> : : = <ПУСТО> | (*R*)

<левая часть> : : = *k* <типовое выражение> ~

<правая часть> : : = < выражение>

В одном предложении не должно быть свободных переменных с одинаковыми индексами, но разными признаками типа. В правую часть предложения могут входить только те переменные, которые входят в левую часть, при этом спецификаторы опускаются.

Областью действия знака *k* называется выражение, начинающееся непосредственно за этим знаком *k* и кончающееся перед парной ему конкретизационной точкой. **Ведущим** знаком *k* в некотором выражении называется первый знак *k* , в области действия которого не содержится ни одного знака *k* .

2. Синтаксическое отождествление

2.1. Говорят, что объектное выражение $\mathcal{E}_0$ может быть **отождествлено как** типовое выражение $\mathcal{E}_t$, если свободные пере-

менные в $\mathscr{E}_t$ могут быть заменены с соблюдением указанных ниже правил на такие выражения, называемые их значениями, что $\mathscr{E}_0$ совпадает с $\mathscr{E}_t$. Правила таковы:

2.1.1. Значением переменной вида $s\mathscr{X}$, где $\mathscr{X}$ – объектный знак, может быть любой символ, значением переменной $t\mathscr{X}$ – любой терм, значением переменной $e\mathscr{X}$ – любое выражение.

2.1.2. Значением переменной вида $s(\mathscr{P})\mathscr{X}$, где $\mathscr{P}$ – объектная цепочка, может быть любой символ, входящий в $\mathscr{P}$. Значениями переменных s'ОБЗН'$\mathscr{X}$ и s'СОСТ'$\mathscr{X}$ могут быть, соответственно, объектные знаки и составные символы. Переменная вида $s\mathscr{D}\mathscr{X}$, где $\mathscr{D}$ – составной символ отличный от 'ОБЗН' и 'СОСТ', эквивалентна переменной $s(\mathscr{P})\mathscr{X}$, где $\mathscr{P}$ – результат конкретизации $k\mathscr{D}\perp$ (см.п.3).

2.1.3. Все вхождения одной и той же переменной (т.е. переменной с одним и тем же индексом) заменяются на одно и то же значение.

2.2. Если существует несколько вариантов придания значений свободным переменным, приводящих к совпадению $\mathscr{E}_t$ с $\mathscr{E}_0$, то эта неоднозначность снимается одним из двух способов, которые будем называть отождествлением слева направо и отождествлением справа налево. При отождествлении слева направо (справа налево) из всех вариантов выбирается тот, при котором самая левая (правая) свободная переменная выражения в $\mathscr{E}_t$ принимает наикратчайшее значение. Если это не устраняет неоднозначности, то такой же отбор производится по второй слева (справа) переменной выражения и т.д.

2.3. Под отождествлением терма $k\mathscr{E}_0\perp$ как левой части предложения $k\mathscr{E}_t\sim$ мы понимаем отождествление $\mathscr{E}_0$ как $\mathscr{E}_t$.

3. Рефал-машина

Рефал-машина – это кибернетическое устройство, состоящее из двух потенциально бесконечных запоминающих устройств (поля памяти и поля зрения) и процессора, преобразующего их содержимое. В каждый момент времени поле памяти содержит некоторый набор предложений, а поле зрения – некоторое рабочее выражение. Работа рефал-машины представляет собой последовательность однотипных действий, называемых шагами.

Выполнение шага начинается с нахождения в поле зрения ведущего знака k. Терм, начинающийся с ведущего знака k, называется конкретизируемым термом. Если в поле зрения нет знаков k, рефал-машина приходит в состояние нормальной остановки. Найдя ведущий знак k, рефал-машина исследует конкретизируемый терм.

3.1. Если он имеет вид

k'ЗК'$\mathscr{C} = \mathscr{E}\perp$

где $\mathscr{C}$ символ, а $\mathscr{E}$ – выражение, то в поле памяти на первое место

заносится предложение

§k 'ВК' $\mathcal{C}$ ~ $\mathcal{E}$

и конкретизируемый терм уничтожается.

3.2. Если конкретизируемый терм имеет вид

k 'ВК' $\mathcal{C}$ ⊥

то рефал-машина отыскивает в поле памяти первое предложение вида

§k 'ВК' $\mathcal{C}$ ~ $\mathcal{E}$

уничтожает его, а конкретизируемый терм заменяет на $\mathcal{E}$. Если такого предложения не нашлось, происходит аварийная остановка.

3.3. В остальных случаях рефал-машина сравнивает конкретизируемый терм с первым, вторым и т.д. предложением в поле памяти, ища подходящее предложение, т.е. такое, что конкретизируемый терм может быть отождествлен как его левая часть. При этом, если указатель реверса пуст, то отождествление проводится слева направо, в противном случае – справа налево. Найдя первое подходящее предложение, машина заменяет в поле зрения конкретизируемый терм на его правую часть, в которую вместо свободных переменных подставлены значения, присвоенные им в процессе отождествления. Если подходящего предложения не нашлось, происходит аварийная остановка.

II. ПРИЕМЫ ПРОГРАММИРОВАНИЯ

1. Открытые и закрытые переменные. Форматы функций

Пусть надо просмотреть данное выражение и каждую группу из нескольких подряд идущих пробелов заменить на один пробел. Пробел рассматриваем как объектный знак, изображаемый на письме "корытом" ␣ . Говоря о просмотре, подразумеваем и всегда будем подразумевать в дальнейшем просмотр без захода в скобки, т.е. только на верхнем уровне скобочной структуры. Такое понятие просмотра будем считать первичным, а имея в виду просмотр с заходом в скобки, будем это специально оговаривать и называть такой просмотр сквозным.

Как подойти к решению этой задачи? После выполнения требуемого преобразования в нашем выражении не должно остаться ни одной пары стоящих рядов пробелов ␣␣ . Следовательно, всюду, где мы встречаем пару пробелов, надо один из них выкинуть. Этого можно достигнуть следующим предложением:

§ 1.1 k 'ЛИКППР' e_1␣␣e_2 ~ k 'ЛИКППР' e_1␣e_2 ⊥

Здесь ЛИКППР (ликвидация повторных пробелов) – детерминатив нужной функции. В правой части предложения мы, выкинув один из пробелов, снова подвергаем результат действию функции 'ЛИКППР',

чтобы выкинуть другие повторные пробелы (рекурсия). Когда повторных пробелов нет, надо кончить работу, оставив аргумент неизменным. Это достигается предложением

§ 1.2 k 'ЛИКППР' $e_1 \sim e_1$

которое помещается после предложения § 1.1.

Приведенная пара предложений дает описание, которое является правильным, но неэффективным, ибо приводит к многократным просмотрам, которых можно было бы избежать. Действительно, переменная e_1 в предложении § 1.1 – открытая, т.е. требующая просмотра. Просмотр этот будет всегда начинаться с начала предложения и кончаться ближайшей парой пробелов (отождествление слева направо). Между тем начинать просмотр с самого начала нет никакой необходимости, так как по определению процесса отождествления переменная e_1 будет иметь значение, которое заведомо не содержит пары пробелов на верхнем уровне скобочной структуры. Поэтому в правой части § 1.1 можно вынести e_1 из области действия знака конкретизации:

e_1 k 'ЛИКППР'␣ e_2 ⊥

Таким образом, описание функции принимает вид

∝ = k 'ЛИКППР'

∝ e_1␣␣e_2 ~ e_1 ∝ ␣e_2

∝ e_1 ~ e_1

Эта пара предложений описывает абсолютно эффективный алгоритм: просматриваем аргумент до ближайшей пары пробелов, выбрасываем один из них и продолжаем просмотр (на следующем шаге рефал-машины) с этого места; если пары пробелов не нашлось, кончаем работу без дополнительных просмотров, ибо переменная e_1 во втором предложении – закрытая.

Задача 1.*

Описать функцию ∝ , которая в заданной последовательности символов оставляет лишь первое вхождение каждого символа, а все повторные вхождения уничтожает.

Часто бывает необходимо как-то разметить объект работы, разделить его на части, играющие различные роли в процессе выполнения алгоритма. Пусть, например, задан список "запрещенных" символов и некая строка символов, из которой надо выбросить все запрещенные, т.е. входящие в первый список символы. Очевидно, надо как-то отделить строку – объект работы от списка запрещенных символов. Попробуем использовать в качестве разделителя символ ⚐ (флажок) в предположении, что такой символ не встречается ни в

**Решения задач см. в конце главы (раздел 8).*

том, ни в другом объекте. Тогда обращение к функции β, решающей поставленную задачу, можно записать в виде

$$\beta \mathscr{E}_1 \vDash \mathscr{E}_2 \perp$$

где $\mathscr{E}_1$ – обрабатываемая строка, а $\mathscr{E}_2$ – список символов.

Функция опишется предложениями:

§ 2.1 $\beta \vDash e_2 \sim$

§ 2.2 $\beta\, s_1\, e_2 \vDash e_a\, s_1\, e_b \sim \beta\, e_2 \vDash e_a\, s_1\, e_b$

§ 2.3 $\beta\, s_1 e_x \sim s_1\, \beta\, e_x$

(Обратите внимание на порядок предложений!)

Кроме того неудобства, что на объект работы накладывается определенное ограничение – отсутствие символа $\vDash$, – это решение имеет гораздо более серьезный недостаток. Так как переменная e_2 в § 2.2 – открытая, машина будет на каждом шаге работы совершать просмотр обрабатываемого выражения (точнее, его оставшейся части), ища флажок. Этот просмотр совершенно не нужен; отождествив первый символ s_1, мы могли бы сразу начинать просмотр списка $\mathscr{E}_2$ в поисках такого же символа – свободная переменная должна быть, конечно, открытой по смыслу алгоритма. Чтобы устранить ненужный просмотр, мы должны сделать переменную e_2 в § 2.2 закрытой. Для этого в качестве разделителя надо использовать не символ, а скобки. С равным успехом можно использовать любой из двух форматов:

$$\beta (\mathscr{E}_1)\, \mathscr{E}_2 \perp$$

$$\beta\, \mathscr{E}_1 (\mathscr{E}_2) \perp$$

Второй формат представляется более естественным с точки зрения смысла функции β. В этом формате описание функции преобразуется к следующему виду:

§ 3.1 $\beta (e_2) \sim$

§ 3.2 $\beta\, s_1\, e_2\, (e_a\, s_1\, e_b) \sim \beta\, e_2\, (e_a\, s_1\, e_b)$

§ 3.3 $\beta\, s_1\, e_x \sim s_1\, \beta\, e_x$

Таким образом, в рефале в качестве разделителей служат, в основном, скобки. Разделитель-символ можно использовать в тех случаях, когда это не приводит к дополнительным просмотрам. Например, если некоторое выражение так или иначе должно просматриваться терм за термом, то признаком окончания просмотра вполне может служить символ. Используются символы-разделители и в тех случаях, когда отделяемые объекты невелики, и потерей времени на их дополнительные просмотры пренебрегают. Может возникнуть вопрос: зачем вообще использовать в качестве разделителей символы, когда есть скобки? Ответ таков: во-первых, символ занимает одно

звено, в то время как пара скобок – два звена; во-вторых, иногда удобно ввести разделитель-символ, чтобы избежать нагромождения скобок.

Пару скобок, служащих для отделения части аргумента, мы будем называть **сумкой**. Так, в приведенном выше примере мы помещаем в сумку список запрещенных символов. Часто в сумку помещают просмотренную часть аргумента – это бывает необходимо в тех случаях, когда просмотренную часть нельзя выносить за конкретизационные скобки, так как она еще может понадобиться в процессе выполнения алгоритма. Рассмотрим такой пример. При наборе последовательности символов на устройствах ввода часто используется система коррекции, позволяющая уничтожать один или несколько последних набранных символов путем набора нужного числа "символов зачеркивания". Пусть в качестве такового используется символ логического отрицания $\neg$. Тогда, например, строка

$$ABC\neg 1+CD=0\neg\neg\neg 2=0$$

должна быть преобразована в строку

$$AB1+CD2=0$$

Опишем это преобразование на рефале. Будем просматривать строку слева направо и просмотренную часть помещать в сумку. Это значит, что "рабочая" функция должна иметь структуру

$$k\,\mathscr{F}(\mathscr{E}_1)\,\mathscr{E}_2\perp$$

где $\mathscr{F}$ – ее детерминатив. Однако основная функция, описывающая процесс коррекции, – присвоим ей детерминатив 'КОРР' – имеет один аргумент и удобно было бы обращаться к ней в виде

$$k\,\text{'КОРР'}\mathscr{E}\perp$$

без всяких скобок. Поэтому сначала мы должны описать переход к "рабочей" функции с заведением сумки (первоначально пустой), а затем уже и саму эту функцию:

$$\gamma = k\,\text{'КОРР'}$$

$$\gamma e_1 \sim \overset{1}{\gamma}(\ )\,e_1$$

$$\overset{1}{\gamma}(e_1 s_a)\neg e_2 \sim \overset{1}{\gamma}(e_1)\,e_2$$

$$\overset{1}{\gamma}(e_1)\,e_2\neg e_3 \sim \overset{1}{\gamma}(e_1 e_2)\neg e_3$$

$$\overset{1}{\gamma}(e_1)\,e_2 \sim e_1 e_2$$

Задача 2.

а) Описать функцию γ двумя предложениями без введения вспомогательной функции и сумки. Проанализировать эффективность алгоритма.

б) Модифицировать описание функции $\overset{1}{\gamma}$ так, чтобы ни в одном предложении не было открытых переменных.

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

Аргумент функции, состоящий из двух частей, можно записать – ради симметрии – и в виде структуры с двумя сумками

$$(\mathscr{E}_1)(\mathscr{E}_2)$$

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

$$(\mathscr{E}_1)(\mathscr{E}_2)\mathscr{E}_3$$

$$((\mathscr{E}_1)\mathscr{E}_2)\mathscr{E}_3$$

$$(\mathscr{E}_1)(\mathscr{E}_2)(\mathscr{E}_3)\mathscr{E}_4$$

и т.п. Можно также вставлять между сумками мнемонические символы для удобства программиста, например:

$$\S k\ \text{'ПРЕОБР'}\ (e_1)\ \text{'ОПЕРАТ'}\ (e_2)\ \text{'РЕЗУЛЬТ'}\ (e_3) \sim \ldots$$

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

$$((\mathscr{E}_1)\mathscr{E}_2)((\mathscr{E}_3)\mathscr{E}_4)\mathscr{E}_5$$

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

$$\phi((e_1)t_a e_2)((e_3)t_a e_4)e_5 \sim$$
$$\qquad \phi((e_1 t_a)e_2)((e_3 t_a)e_4)\ e_5 t_a$$
$$\phi((e_1)t_a e_2)((e_3)t_b e_4)e_5 \sim$$
$$\qquad \phi((e_1 t_a)\ e_2)((e_3 t_b)e_4)e_5$$
$$\phi((e_1)e_2)((e_3)e_4)e_5 \sim (e_1 e_2)(e_3 e_4)(e_5)$$

Задача 3.

Описать процедуру ψ, просматривающую выражение терм за термом и применяющую к очередному терму процедуру α, если этот терм встречается в обработанной части выражения (на верхнем уровне скобочной структуры), и процедуру β – в противном случае. Испробовать два формата: $(\mathcal{E}_1)\mathcal{E}_2$ и $\mathcal{E}_1(\mathcal{E}_2)$, где $\mathcal{E}_1$ – обработанная часть, и сравнить работу рефал-машины в обоих случаях.

2. Размерность просмотра

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

Приведем простой пример квадратичного просмотра (ср.задачу 1). Пусть надо обнаружить в объектном выражении наличие двух одинаковых термов. Это может быть достигнуто предложением с левой частью

$$\phi e_1 t_a e_2 t_a e_3 \sim$$

Здесь две открытых переменных e_1 и e_2, поэтому размерность просмотра $\nu = 2$. Посмотрим, какие действия будет совершать рефал-машина. Сначала она придаст переменной e_1 пустое значение, т.е. в качестве терма t_a выберет первый терм выражения. Затем пустое значение будет придано переменной e_2, и если второй терм не совпадает с первым, переменная e_2 будет удлиняться в поисках такого терма. Просмотрев все выражение и не найдя нужного терма, машина приступит к удлинению переменной e_1, т.е. выберет в качестве t_a второй терм и снова станет искать тождественный ему терм и т.д. Таким образом, рефал-машина совершает именно те, и только те действия, которые необходимы по содержанию алгоритма, так что работа рефал-транслятора будет столь же эффективна, как если бы была написана специальная программа.

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

$$\phi e_1 A e_x Z e_2 A e_x Z e_3 \sim$$

Она порождает просмотр третьей размерности. Первая открытая переменная – e_1, вторая – e_x (первое вхождение), третья – e_2. Переменная e_3 – закрытая. Проанализировав работу рефал-машины, мы найдем, что в тех случаях, когда отождествление возможно, рефал-машина, как и в предыдущем примере, совершит лишь абсолютно необходимые действия, но если отождествление невозможно, может случиться, что машина будет совершать лишние действия. Мы покажем это на более простом примере. Пусть левая часть предложения имеет вид:

$$\phi e_1 A e_x Z e_2 \sim$$

Допустим, что в просматриваемом выражении встречается много символов A , но нет ни одного символа Z . Найдя первый символ A , рефал-машина станет удлинять e_x, ища Z. Из того, что символ Z не найден, уже можно сделать вывод о невозможности отождествления. Однако рефал-машина, как она определена в формальном описании, будет удлинять e_1 и каждый раз искать несуществующий символ Z. Только перебрав все символы A, она объявит о невозможности отождествления. Этот недостаток абстрактной рефал-машины может быть, конечно, исправлен на уровне реализации. Хороший рефал-транслятор должен распознавать такие ситуации и не делать лишних просмотров.

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

§ 4.1 Выделение цепочки, начинающейся символом A и кончающейся символом Z :

k 'ВЫДЦAZ' $e_1 A e_x Z e_2 \sim A e_x Z$

§ 4.2 Если нужной цепочки нет

k 'ВЫДЦAZ' $e_1 \sim$ 'УВЫ'

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

§ 5.1 Сначала находим A :

k 'ВЫДЦAZ' $e_1 A e_2 \sim k$ 'ИСКZ'$(e_1 A) e_2 \perp$

§ 5.2 k 'ВЫДЦAZ' $e_1 \sim$ 'УВЫ'

§ 6.1 Теперь ищем Z:

k 'ИСКZ' $(e_1 A) e_x Z e_2 \sim A e_x Z$

§ 6.2 k 'ИСКZ' $e_1 \sim$ 'УВЫ'

В левой части предложения § 4.1 две открытые переменные. Предложения §§5,6 содержат в левых частях не более чем по одной открытой переменной. Размерность просмотра, необходимого для отождествления, снижается с двух до единицы. Естественно поставить вопрос: нельзя ли вообще обойтись без открытых переменных? Ответ на этот вопрос утвердительный. Конечно, устранение открытых переменных не устраняет просмотра, если он необходим по смыслу алгоритма, но просмотр этот осуществляется не в процессе отождествления, а путем многократного обращения к одной и той же рекурсивной функции или к ряду рекурсивных функций, вызывающих одна другую. Общий принцип устранения открытой переменной таков: с помощью введения вспомогательной функции описать процедуру набора (удлинения) данной открытой переменной, выполняемую при отождествлении. При этом можно ликвидировать ненужные просмотры, что и было продемонстрировано в приведенном выше примере.

Устраним открытую переменную e_1 в § 5.1. Нам придется ввести вспомогательную функцию $\propto$, содержащую сумку, куда будет набираться терм за термом значение переменной e_1.

§ 7 Начальное значение пусто:

k 'ВЫДЦАZ' $e_1 \sim \propto (\ ) e_1 \perp$

§ 8.1 Когда дошли до символа A :

$\propto (e_1) A e_2 \sim k$ 'ИСКZ' $(e_1 A) e_2 \perp$

§ 8.2 Пока до него не дошли:

$\propto (e_1) t_x e_2 \sim \propto (e_1 t_x) e_2 \perp$

§ 8.3 Когда его нет:

$\propto (e_1) \sim$ 'УВЫ'

Предложения § 8.1 и § 8.3 почти совпадают с предложениями § 5.1 и § 5.2, соответственно. Отличие состоит только в том, что открытая переменная e_1 заключается в скобки и становится тем самым закрытой. Удлинение e_1 описывается рекурсивно предложением § 8.2. Так как оно является более общим, чем § 8.1, оно должно быть помещено после него.

В простейших случаях устранение открытой переменной возможно без введения вспомогательных функций.

Пусть в заданной последовательности символов надо каждый знак "+" заменить на знак "–". С использованием открытой переменной задача решается двумя предложениями:

$\phi e_1 + e_2 \sim e_1 - \phi e_2$

$\phi e_1 \sim e_1$

Без открытой переменной для описания функции ϕ требуется три предложения:

$\phi + e_1 \sim \phi e_1$

$\phi s_x e_1 \sim s_x \phi e_1$

$\phi \sim$

Это описание хуже, чем предыдущее: оно не только содержит больше предложений, но и выполняется медленнее, так как требует рекурсивного обращения к функции ϕ для каждого терма. В тех случаях, когда ~~наличие от~~крытых переменных не требует лишних просмотров, оно приводит к ускорению работы машины.

Задача 4.

Решение задачи 1, приведенное в разд.8 не вполне эффективно: проекция переменной e_2 при использовании первого предложения будет напрасно просматриваться на следующем шаге. Ликвидировать этот недостаток с помощью понижения размерности просмотра.

3. Размножение переменных

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

$$\phi(e_1)e_2 \sim (e_2)e_2$$

может в сотни раз превышать время, затрачиваемое при применении предложения

$$\phi(e_1)e_2 \sim (e_2)e_1$$

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

Аналогичные соображения применимы к размерам правой части предложения. Необходимо учитывать, что если правая часть содержит много символов, которых нет в левой части, то транслятору понадобится заметное время, чтобы вписать эти символы в поле

зрения. И если эти символы на следующем этапе становятся ненужными и удаляются из поля зрения, а затем снова оказываются нужны и снова вписываются и удаляются, и так далее, то такой режим работы нельзя признать эффективным. Вместо этого следует в начале работы закопать нужный список символов в поле зрения и выкапывать его, когда в нем появляется потребность, не забывая, конечно, снова закапывать его после использования. Процедуры закапывания и выкапывания выполняются, как известно, весьма быстро, независимо от размеров списка.

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

§ 9.1 k 'ПОРЯДОК' $s_a s_a \sim T\ s_a s_a$

§ 9.2 Необходима дополнительная информация – алфавитный порядок букв

k 'ПОРЯДОК' $e_1 \sim \propto e_1$ АБВ...ЭЮЯ ⊥

§ 9. А.1 $\propto s_a s_b e_1 s_a e_2 s_b e_3 \sim T\ s_a s_b$

§ 9. А.2 $\propto s_a s_b e_1 \sim F\ s_a s_b$

Это решение обладает тем недостатком, о котором мы говорили выше: каждый раз при использовании предложения § 9.2 рефал-транслятор будет заново формировать в поле зрения список из 32 символов, а на следующем шаге – при использовании предложений § 9А.1 или § 9 А.2 – будет его уничтожать.

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

k 'ЗК' 'АЛФАВИТ' = АБВ...ЭЮЯ ⊥

Функция 'ПОРЯДОК' описывается предложениями

§ 10.1 k 'ПОРЯДОК' $s_a s_a \sim T\ s_a s_a$

§ 10.2 k 'ПОРЯДОК' $e_1 \sim \propto e_1 k$ 'ВК' 'АЛФАВИТ' ⊥ ⊥

§ 10. А.1 $\propto s_a s_b e_1 s_a e_2 s_b e_3 \sim T\ s_a s_b\ k$ 'ЗК' 'АЛФАВИТ' = $e_1 s_a e_2 s_b e_3$ ⊥

§ 10. А.2 $\propto s_a s_b e_1 \sim F\ s_a s_b k$ 'ЗК' 'АЛФАВИТ' = e_1 ⊥

Заметим, что, хотя правые части в предложениях § 10.А записываются довольно длинно, время, требуемое для замены, невелико. Ведь комбинация пяти переменных $e_1 s_a e_2 s_b e_3$ входит в правую часть точно так же, как и в левую, поэтому никакой перешивки здесь не требуется: алфавит закапывается целиком и без просмотров.

4. Разветвления и циклы

В алгоритмических языках разветвления чаще всего описываютс с помощью условных выражений (выражений в широком смысле, включая, например, операторы АЛГОЛа-60) в сочетании с предикатами – функциями, принимающими одно из двух истинностных значений: T или F. На рефале нетрудно описать семантику условного выражения таким образом, чтобы его можно было использовать в привычном виде:

§ 11.1 Условное выражение. Первые два предложения работают тогда, когда уже выполнена конкретизация предиката до истинностного значения.
k'ЕСЛИ'(T) 'ТО'(e_1)'ИНАЧЕ' $e_2 \sim e_1$

§ 11.2 k'ЕСЛИ'(F) 'ТО'(e_1)'ИНАЧЕ' $e_2 \sim e_2$

§ 11.3 Это предложение приводит к конкретизации предиката, если он записывается без функциональных скобок. Его можно истолковать так: чтобы конкретизировать условное выражение, сначала конкретизируют предикат.
k'ЕСЛИ' $(e_p)e_1 \sim k$'ЕСЛИ' $(ke_p\perp)e_1\perp$

Нетрудно описать на рефале и логические связки, например:

k'НЕ'$T \sim F$

k'НЕ'$F \sim T$

k'НЕ'$e_1 \sim k$'НЕ'$ke_1\perp\perp$

Задача 5.

Описать логические связки 'И' и 'ИЛИ' в формате $(\mathscr{E}_1)$ $(\mathscr{E}_2)$, соблюдая тот же принцип, что и выше: предложения должны работать как в том случае, когда аргументы записываются с явным введением знаков k, так и в том случае, когда знаки k отсутствуют.

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

Однако эта система имеет другой, гораздо более серьезный недостаток.(Мы имеем в виду не столько систему записи логических выражений, сколько систему использования предикатов в условных выражениях,интерпретируемых согласно § 11.)

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

k'СЛЕДУЕТ' (e_1) 'ЗА' $e_2\perp$

Это отношение широко используется при проведении аналитических выкладок для упорядочения выражений относительно коммутативных операций. Аргументы здесь могут быть весьма громоздки. И пусть надо определить процедуру упорядочения двух выражений, используя это отношение таким образом, чтобы второе выражение "следовало" за первым. Мы записываем:

§ 12 k 'УПОРЯД' $(e_1)(e_2)$ ~
k 'ЕСЛИ' ('СЛЕДУЕТ' (e_2) 'ЗА' e_1)
'ТО' $((e_1)(e_2)$ 'ИНАЧЕ' $(e_2)(e_1)$

Это описание привычно для глаза, и оно будет правильно работать. Его недостаток в том, что оно приводит к неэффективному использованию машины, ибо требует размножения переменных, которое совершенно бессмысленно с точки зрения алгоритма. Каждая из переменных e_1 и e_2 левой части входит в правую часть трижды и следовательно воспроизводится дважды, причем без всякой необходимости, ибо обе копии тут же уничтожаются: первая копия уничтожается при конкретизации предиката (когда аргумент исчезает и остается только истинностное значение), вторая копия – при конкретизации условного выражения в соответствии с § 11.

Не следует думать, что эта неэффективность отражает какой-то недостаток языка рефал. Скорее наоборот: описав условное выражение на рефале, мы увидели воочию тот источник неэффективности, который коренится в самом принципе использования классических предикатов, однако приводит к неприятностям только в случае операций с громоздкими аргументами. Действительно, когда мы описываем алгоритмы на обычном операторном языке, например, на языке вычислительной машины, мы не думаем об аргументах как о физических объектах, которые переносятся, воспроизводятся, уничтожаются и т.п. Все эти работы мы оставляем на долю тех частей системы, которые отвечают за реализацию языка. Как же будет происходить реализация условного оператора, подобного рассмотренному выше, если бы он был написан на машинном языке? Пусть упорядочиваемые объекты – числа, которые хранятся в ячейках оперативной памяти, а отношение следования – это отношение "больше-меньше".

Чтобы сравнить два числа, машина переписывает в специальные регистры сначала одно из них, а затем другое. Это и есть первое размножение. Затем машина производит вычитание, в результате чего вырабатывается истинностное значение предиката, например сигнал ω, а обе копии аргументов уничтожаются. В зависимости от значения предиката управление передается в ту или иную ячейку, и выполняется программа переписи чисел в нужном порядке. Это – второе размножение.

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

Приложим этот принцип к предикатам. Вместо обычных предикатов, которые аргумент заменяют на истинностное значение, будем использовать функции, которые аргумент сохраняют и только приписывают к нему истинностное значение. Такие функции будем называть предикатами, сохраняющими аргумент, или кратко C-предикатами. В частности, определим функцию 'СЛЕДУЕТ' таким образом, что конкретизация

$$k \text{ 'СЛЕДУЕТ'} (e_1) \text{ 'ЗА'} \ e_2 \perp$$

даст либо

$$T(e_1) e_2$$

либо

$$F(e_1) e_2$$

Подчеркнем, что использование той или иной формы C-предикатов – этот вопрос, решение которого целиком зависит от программиста, ибо он пишет как предложения, определяющие предикат, так и предложения, где он используется. Программист может, например, если это ему покажется удобным, определить отношение 'СЛЕДУЕТ' таким образом, что результатом конкретизации будет либо

$$\text{'СЛЕДУЕТ'} (e_1) \text{ 'ЗА'} \ e_2$$

либо

$$\text{'ПРЕДШЕСТВУЕТ'} (e_1) e_2$$

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

Используя С-предикаты, можно описать разветвление без всякого размножения переменных. Для этого надо разнести два варианта подставляемого выражения по разным предложениям: одно предложение будет описывать случай, когда предикат дал значение T, другое – значение F. Процедура упорядочивания, рассчитанная на C-предикат 'СЛЕДУЕТ', будет описываться двумя предложениями:

§ 13.1 k 'УПОР' $T(e_1)e_2 \sim (e_2)(e_1)$

§ 13.2 k 'УПОР' $F(e_1)e_2 \sim (e_1)(e_2)$

Такое описание предполагает, что при обращении к функции 'УПОР' аргумент уже обработан функцией 'СЛЕДУЕТ', то есть для того, чтобы действительно упорядочить пару выражений $\mathscr{E}_1$ и $\mathscr{E}_2$, надо выполнить конкретизацию

k 'УПОР' k 'СЛЕДУЕТ' $(\mathscr{E}_1)$ 'ЗА' $\mathscr{E}_2 \perp\perp$

Это экономно, но не всегда удобно, поэтому функцию 'УПОР' лучше все-таки определить так, чтобы ее аргументом была пара упорядочиваемых выражений. Следовательно, надо обеспечить сначала применение к аргументу C-предиката 'СЛЕДУЕТ', а затем – с помощью вспомогательной функции – разветвление по результату.

$\alpha = k$ 'УПОР'

$\alpha\,(e_1)e_2 \sim \alpha^1 k$ 'СЛЕДУЕТ' (e_1) 'ЗА' $e_2 \perp\perp$

$\alpha^1 T(e_1)e_2 \sim (e_2)(e_1)$

$\alpha^1 F(e_1)e_2 \sim (e_1)(e_2)$

Это описание по своей структуре и по приемам программирования, отразившимся в нем, ничем не отличается от примеров, рассмотренных выше. Разветвление алгоритмического процесса осуществляется путем исследования аргумента на предмет вхождения в него в определенных местах определенных символов (T, F). Но наличие в группе предложений более чем одного предложения всегда порождает разветвление процесса, регулируемое синтаксическим анализом аргумента. Поэтому в рефале предикаты теряют то исключительное положение, которое они имеют в операторных языках как единственное средство управления ветвлением процесса. Специфика C-предикатов по сравнению с другими функциями состоит в том, что они не преобразуют аргумент, а только анализируют его, или, если говорить строго формально, преобразуют его таким образом, что приписывают к нему те или иные синтаксические указатели в зависимости от результата анализа. Разделение функций на те, которые только анализируют, и те, которые только преобразовывают, принятое в операторных языках, при программировании на рефале оказывается чаще всего ненужным, ибо в описание каждой

функции можно включить синтаксический анализ аргумента. Обычно рефал-функция, преобразуя аргумент, оставляет тем самым информацию в виде простых синтаксических признаков для следующей функции, которая будет использовать эти признаки для управления ветвлением. Это делает функции более емкими, а описание – более компактным. В тех случаях, когда результат преобразования не обладает сам по себе такими признаками, можно создать их искусственно, приписывая тот или иной синтаксический указатель и объединив опять-таки предикат и преобразователь в одной функции. Например, для процедуры упорядочения многих термов можно в качестве основы принять не предикат следования для двух термов, а процедуру упорядочения двух термов. Если мы ожидаем, что нам понадобится информация о том, были ли первоначально термы в нормальном порядке или их пришлось переставить, эта процедура должна быть определена так, чтобы давать результат

'НОРМ' $t_1 t_2$

в первом случае и результат

'ПЕРЕСТ' $t_2 t_1$

– во втором.

Итак, при программировании на рефале предикаты обычно растворяются в общей массе функций.

Задача 6.

Используя описанную выше процедуру упорядочения двух термов (присвоим ей детерминатив 'УПОР 2'), описать процедуру 'УПОР' с форматом

k 'УПОР' $\mathcal{T}_1\ \mathcal{E}_2$

ставящую терм $\mathcal{T}_1$ на нужное место в упорядоченной последовательности термов $\mathcal{E}_2$.

Задача 7.

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

То, что на операторном языке выглядит как возвращение в процессе выполнения алгоритма к уже пройденной точке, на сентенциальном языке называется рекурсией. Описание на рефале функции, которая прямо или косвенно вызывает саму себя, с точки зрения операторного языка есть описание цикла (в широком смысле слова). И обратно, если некий алгоритм описан в виде цикла на операторном языке, то при описании его на рефале цикл перейдет в рекурсивную функцию от тех переменных, которые участвуют в цикле.

Так как обычно в цикле используются дополнительные переменные, кроме тех, от которых зависит результат, обращение к циклу осу-

ществляется с помощью вспомогательной функции, вызывающей ту рекурсивную функцию, которая выполняет необходимую работу.

Рассмотрим в качестве примера вычисление факториала. Его можно описать на рефале, воспользовавшись функцией 'ВЧЦ' – "вычисления над целыми числами", если понадобились точные значения факториалов больших чисел. Напоминаем формат функции 'ВЧЦ': k'ВЧЦ'$\mathcal{Z}\mathcal{N}_1^o,\mathcal{N}_2^o \perp$, где $\mathcal{Z}$ – знак операции, $\mathcal{N}_1^o$ и $\mathcal{N}_2^o$ – операнды, записанные с помощью "числовых символов".

Простейшее описание функции факториала (детерминатив 'FACT') повторяет обычное рекуррентное определение:

$\phi = k$'FACT', $\omega = k$'ВЧЦ'

§ 14.1 ϕ'0' $\sim$ '1'

§ 14.2 $\phi e_n \sim \omega \times \phi\omega - e_n$,'1'$\perp\perp$, $e_n \perp$

В этом описании рекурсивное обращение к функции ϕ входит в аргумент функции ω (внешней). Поэтому, прежде чем начнет выполняться умножение, распишется длинное произведение, содержащее столько множителей, сколько было обращений к ϕ , то есть численно равное аргументу этой функции. На языке АЛГОЛ-60 такому описанию факториала соответствует следующее описание рекурсивной процедуры – функции FACT:

```
integer procedure FACT (n);
value n; integer n;
FACT: = if n = 0 then 1
else FACT (n - 1) × n
```

При исполнении этой программы будет иметь место то же накопление обращений к FACT перед началом основного счета. Это не всегда желательно как при использовании АЛГОЛа, так и при использовании рефала. На АЛГОЛе можно ликвидировать многократное обращение к функции FACT, если описать ее без рекурсии (в смысле АЛГОЛа), но с использованием оператора цикла:

```
integer procedure FACT (n);
value n; integer n ;
begin integer f,m;
f : = 1;
for m : = 1 step 1 until n do
f : = f×m;
FACT : = f
end
```

Аналогичная модификация может быть выполнена и на рефале:

$$\S\ 15\ \phi e_n \sim \phi^1('1')(\omega + e_n, '1')'1'$$

$$\S\ 16.1\ \phi^1(e_n)(e_n)e_f \sim e_f$$

$$\S\ 16.2\ \phi^1(e_m)(e_n)e_f \sim \phi^1(\omega + e_m, '1')(e_n)$$

$$\omega \times e_f, e_m$$

Здесь первое предложение соответствует введению вспомогательных (локальных) переменных и присваиванию им начальных значений; второе – проверке цикла на окончание; третье – выполнению тела цикла. В предложении § 16.2, определяющем функцию ϕ^1, обращение к самой ϕ^1 объемлет всю правую часть предложения, поэтому размножения каких-либо конфигураций правой части не происходит, а происходит лишь их многократная конкретизация. Этот частный вид рекурсии и есть цикл в узком смысле слова, как он определен, например, в АЛГОЛе.

5. Разделение алгоритма на функции

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

Обычные основания для введения новой функции таковы:

1. Членение объекта согласно определенной схеме, задаваемой типовым выражением в левой части предложения.

2. Создание разветвления путем описания различных частных случаев отдельными предложениями с одним и тем же детерминативом. Классическим примером служат функции, использующие C-предикаты.

3. Изменение формата (заведение сумок), диктуемое, как правило, необходимостью включить в рекурсивный процесс новый объект (аргумент) или несколько объектов.

4. Предварительная обработка аргумента для эффективной работы основной функции или нескольких функций.

Если в правую часть предложения входит подвыражение вида

$$k \mathcal{F} \mathcal{E} \perp$$

где $\mathcal{E}$ – любое рабочее выражение (содержащее, возможно, знаки конкретизации), то будем говорить, что данное предложение вызы-

вает функцию $\mathcal{F}$. Если хотя бы одно предложение с детерминативом $\mathcal{F}_1$ вызывает функцию $\mathcal{F}_2$, будем говорить, что функция $\mathcal{F}_1$ вызывает функцию $\mathcal{F}_2$. Не всякое вхождение знака k в правую часть предложения является вызовом определенной функции. За знаком k может следовать свободная переменная символа или выражения, тогда вызываемая функция определится только динамически, в процессе выполнения программы. Такие ситуации мы рассмотрим в разделе о метафункциях, а пока предположим, что они отсутствуют, так что с каждой функцией можно связать список (возможно, пустой) функций, которые она вызывает, и никаких других функций, кроме входящих в этот список, она вызвать не может.

Отношения вызова между функциями могут быть изображены в виде ориентированного графа, где вершины соответствуют функциям и помечаются их детерминативами, а вызов функцией $\mathcal{F}_1$ функции $\mathcal{F}_2$ изображается дугой (стрелкой), направленной от $\mathcal{F}_1$ к $\mathcal{F}_2$. Циклы на таком графе наглядно изображают рекурсию.

Пусть некоторый объект надо подвергнуть последовательной обработке двумя или большим числом функций $\alpha, \beta, \gamma \ldots$ и т.д. Это можно осуществить двумя способами. Во-первых, можно определить функцию ϕ, которая к аргументу последовательно применяет независимо друг от друга описанные функции $\alpha, \beta, \gamma \ldots$, например:

$$\phi e_1 \sim \ldots \gamma \beta \alpha e_1 \perp \perp \perp \ldots$$

В более сложных случаях в правую часть могут входить и другие объекты: пустые сумки и т.п. Такой способ порождает граф вызовов, изображенный на рис.**2.1**, поэтому мы будем называть его **горизонтальным** соединением функций в последовательность.

Во-вторых, можно определить функцию α таким образом, чтобы, окончив работу, она вызывала функцию β, передав ей свое значение в качестве аргумента. Аналогично, β вызывает γ и т.д. Соответствующий граф вызовов имеет вертикальное строение (см.рис.**2.2**), поэтому назовем такое соединение **вертикальным**. Достоинством горизонтального соединения является независимое определение функций. С другой стороны, вертикальное соединение часто оказывается удобнее, особенно, когда при передаче аргумента его надо разбивать на части.

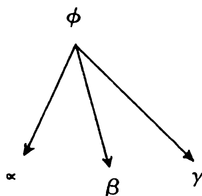


Рис.2.1

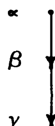


Рис.2.2.

Пусть две функции – 'F1' и 'F2' – заданы наборами из n и m предложений:

k 'F1' $\mathscr{E}_{11}$ ~ ...
k 'F1' $\mathscr{E}_{12}$ ~ ...
...
k 'F1' $\mathscr{E}_{1n}$ ~ ...
k 'F2' $\mathscr{E}_{21}$ ~ ...
...
k 'F2' $\mathscr{E}_{2m}$ ~ ...

Эти две функции можно слить в одну функцию 'F', определив ее $n+m$ предложениями

k 'F' 'F1' $\mathscr{E}_{11}$ ~ ...
...
k 'F' 'F1' $\mathscr{E}_{1n}$ ~ ...
k 'F' 'F2' $\mathscr{E}_{21}$ ~ ...
...
k 'F' 'F2' $\mathscr{E}_{2m}$ ~ ...

где в правых частях вызовы функций 'F1' и 'F2': k 'F1' $\mathscr{E}\perp$ и k 'F2' $\mathscr{E}\perp$ заменены на k 'F' 'F1' $\mathscr{E}\perp$ и k 'F' 'F2' $\mathscr{E}\perp$ соответственно.

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

Назовем *однопробной* такую функцию Φ, которая описывается не более чем двумя предложениями, причем второе предложение , если оно присутствует, имеет в качестве аргумента свободную переменную выражения:

ϕ e_1 ~ ...

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

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

§ $X.1\ \phi \mathcal{E}_1 \sim \mathcal{E}'_1$

§ $X.2\ \phi \mathcal{E}_2 \sim \mathcal{E}'_2$

. . .

§ $X.n\ \phi \mathcal{E}_n \sim \mathcal{E}'_n$

Введем $n-1$ новых функций: $\alpha_2, \alpha_3, \dots \alpha_n$, и определим n однопробных функций следующими наборами предложений:

§ $X1.1\ \phi \mathcal{E}_1 \sim \mathcal{E}'_1$

§ $X1.2\ \phi e_1 \sim \alpha_2 e_1 \perp$

§ $X2.1\ \alpha_2 \mathcal{E}_2 \sim \mathcal{E}'_2$

§ $X2.2\ \alpha_2 e_1 \sim \alpha_3 e_1 \perp$

. . .

§ $Xn.1\ \alpha n \mathcal{E}_n \sim \mathcal{E}'_n$

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

6. Действия над многочленами от одной переменной

Чтобы проиллюстрировать методы работы на рефале, рассмотрим одну несложную задачу из области алгоритмизации алгебраических выкладок.

Пусть нам надо выполнять на машине действия над полиномами от одной переменной с рациональными коэффициентами; сложение, вычитание, умножение, дифференцирование, интегрирование и т.п. В качестве более конкретной задачи возьмем такую: выдать на печать некоторое число полиномов, определяемых рекуррентными соотношениями, например тридцать начальных полиномов Лежандра. С чего начать решение этой задачи?

Прежде всего, надо выбрать форму представления полинома в рефал-машине. Можно было бы перенести один к одному обычную запись:

$$a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

(где a_k – рациональные числа), заменив x^k на $x \uparrow k$, однако это, во-первых, порождает лишние просмотры, во-вторых, включает ненужную информацию. Так как переменная у нас одна, буква x вообще может быть опущена. Далее, если выписывать все коэффициенты a_k – в частности и те, которые равны нулю, – то можно опустить и показатель степени. Каждый коэффициент заключим в пару скобок. Таким образом, полином примет вид

$$(a_0)(a_1)(a_2)\ldots(a_n)$$

Такая форма записи будет невыгодной для полиномов, подобных $1 + x^{10}$, однако мы все-таки остановимся на ней, предполагая, что полиномы с большим числом "дырок" встречаются нечасто. Для полиномов определенной четности эта форма примерно эквивалентна записи, включающей показатели, а для полиномов без "дырок" она приводит к экономии. Кроме того, можно предчувствовать, что простота и определенная регулярность данной формы записи объектов даст возможность сделать алгоритмы преобразования объектов краткими и эффективными. Заметим, впрочем, что это предчувствие может и обмануть. При описании алгоритмов на рефале нередко обнаруживается, что выбранная форма представления объектов является не наилучшей. Тогда приходится переделывать ее. Достижение единства формы и содержания требует иногда нескольких шагов по методу проб и ошибок.

Теперь надо установить форму представления рациональных чисел и действий над ними. Остановимся на обычной записи, иллюстрируемой примерами: 0, 1, 9, 15, 2/3, 224/15, –16, –5/7. Действия над рациональными числами будет выполнять функция 'РАЦ', имеющая тот же формат, что функция 'ВЧЦ'. Функцию 'РАЦ' описывать не будем, предполагая, что она либо уже описана (и хранится в библиотеке рефал-программ), либо реализована как машинная процедура, либо, наконец, будет описана позже.

Приступим к описанию действий над полиномами. Начнем со сложения. Введем функцию 'СЛ' с двумя сумками: k 'СЛ' $(\mathscr{P}_1)$ $(\mathscr{P}_2) \perp$, где $\mathscr{P}_1$ и $\mathscr{P}_2$ – складываемые полиномы. Алгоритм почленного сложения чрезвычайно прост:

k 'СЛ' $((e_a)e_1)((e_b)e_2) \sim (k$ 'РАЦ' $+ e_a, e_b \perp)$ $\quad k$ 'СЛ' $(e_1)(e_2)\perp$

k 'СЛ' $(e_1)(\ ) \sim e_1$

k 'СЛ' $(\ )(e_2) \sim e_2$

Написав эти три предложения, уже готовимся перейти к следующей функции, как вдруг замечаем, что упустили одно важное обстоятельство: если несколько последних членов в сумме полиномов обратились в нуль, их надо отбросить. Простейший способ исправить эту ошибку – выполнить над результатом сложения еще одно преобразование, которому придадим детерминатив 'СОКР' – сокращение концевых нулей. Применяя горизонтальное соединение функций, запишем:

$$k\ \text{'СЛ'}\ e_x \sim k\ \text{'СОКР'}\ k\ \text{'СЛ1'}\ e_x \perp\perp$$

а в описании функции 'СЛ' заменим всюду 'СЛ' на 'СЛ1'. Для функции 'СОКР' можно предложить следующее описание:

$$k\ \text{'СОКР'}\ e_1 (0) \sim k\ \text{'СОКР'}\ e_1 \perp$$

$$k\ \text{'СОКР'}\ e_1 \sim e_1$$

Если полином целиком состоит из нулей, то функция 'СОКР' превращает его не в (0), а в < пусто> Это заставляет нас задуматься над вопросом, который следовало бы разрешить раньше – при формулировке понятия "многочлен": включить ли в число многочленов пустой объект? Ответ на этот вопрос вовсе не так очевиден, как это может показаться с первого взгляда, ибо с точки зрения регулярности синтаксиса часто бывает удобно допустить, что если объект состоит из произвольного числа элементов, то число это, в частности, может быть равно и нулю. С точки зрения чистого синтаксиса это более логично, чем ограничение одним элементом. Тем не менее, в данном случае мы потребуем, чтобы свободный член полинома присутствовал всегда. Тогда описание функции 'СОКР' придется несколько изменить.

$$k\ \text{'СОКР'}\ e_1 t_2 (0) \sim k\ \text{'СОКР'}\ e_1 t_2 \perp$$

$$k\ \text{'СОКР'}\ e_1 \sim e_1$$

Теперь процедура сложения будет давать правильный результат. Но с точки зрения эффективности она имеет небольшой изъян: если складываемые полиномы имеют разную длину, то функцию 'СОКР' можно и не применять. Как исправить этот изъян? Ясно, что считать члены было бы нелепо. Однако, мы замечаем, что при работе функции 'СЛ' первое предложение становится неприменимым как раз тогда, когда кончаются равные отрезки в обоих полиномах. Следовательно, в этот-то момент и можно (если нужно) вставить обращение к 'СОКР'. Для этого придется от горизонтального соединения функций перейти к вертикальному, и, значит, результат сложения членов мы не можем теперь выбрасывать за пределы функциональных скобок, как делали раньше. Будем помещать этот результат между второй сумкой и конкретизационной точкой. Вспомогательная функция 'СЛ1' становится ненужной, и мы получаем описание:

$$k\ \text{'СЛ'}\ ((e_a)\, e_1)\, ((e_b)\, e_2)\, e_r \sim k\ \text{'СЛ'}\ (e_1)(e_2)\ e_r (k\ \text{'РАЦ'}\ + e_a, e_b \perp) \perp$$

$$k\ \text{'СЛ'}\ ()\ (\ )\, e_r \sim k\ \text{'СОКР'}\ e_r \perp$$

$$k\ \text{'СЛ'}\ (e_1)(\ )\ e_r \sim e_r e_1$$

$$k\ \text{'СЛ'}\ (\ )(e_2)\, e_r \sim e_r e_2$$

Вычитание полиномов описывается совершенно так же, надо только в обращении к 'РАЦ' заменить знак + на –. Нерационально из-за такого незначительного изменения добавлять к программе еще четыре параграфа. Нельзя ли этого избежать? Можно. Вместо двух функций – сложения и вычитания введем одну функцию СВ ("сложение-вычитание") и сразу же за детерминативом будем ставить знак + или –. В остальном формат 'СВ' – такой же как у 'СЛ'. Первый параграф описания 'СВ' имеет вид

$$k\ \text{'СВ'}\ s_f((e_a)\, e_1)((e_b)\, e_2)\, e_r \sim k\ \text{'СВ'}\ s_f (e_1)(e_2)\, e_r (k\ \text{'РАЦ'}\ s_f e_a, e_b \perp) \perp$$

Остальные предложения модифицируются аналогично.

Перейдем к умножению (детерминатив 'УМН'). Прежде всего опишем простую функцию 'УМН 1' умножения полинома на константу. Формат $-k$ 'УМН 1'$\mathscr{P}$ $(\mathscr{C}) \perp$, где $\mathscr{P}$ – полином, а $\mathscr{C}$ – константа.

$$k\ \text{'УМН1'}\ (e_a)\, e_1\, (e_c) \sim (k\ \text{'РАЦ'} \times e_a, e_c \perp)\ k\ \text{'УМН1'}\ e_1\, (e_c) \perp$$

$$k\ \text{'УМН1'}\ (e_c) \sim$$

Для выполнения умножения, очевидно, понадобится сумматор. Разметим его в аргументе функции 'УМН' перед конкретизационной точкой. Будем помножать первый аргумент на последовательные члены второго аргумента и сдвигать каждый раз сумматор на один член влево. Схема умножения показана на рис.2.3.

Арг_1	A	B	C	D	...		A	B	C	D	...
Арг_2	E	F	G		...	$\Longrightarrow$	F	G			...
Сумм	U	V	W	X	...		V'	W'	X'		...
	AE	BE	CE	DE	...						
Сумм	U' ←	V'	W'	X'	...						

Рис.2.3.

Ей соответствует следующее описание:

k 'УМН' $(e_1)((e_a)\, e_2)e_c \sim k$ 'УМН2' $(e_1)(e_2)$

k 'СВ' + $(e_c)(k$ 'УМН' $e_1(e_a)\perp)\perp\perp$

k 'УМН' $(e_1)(\ )e_c \sim e_c$

k 'УМН2' $t_1 t_2 t_a e_3 \sim t_a\, k$ 'УМН' $t_1 t_2 e_3 \perp$

Этот алгоритм не свободен от некоторых излишеств, однако ими можно пренебречь.

Задача 8.

Описать функции 'ДИФ' и 'ИН', дающие производную и неопределенный интеграл от многочлена.

Присвоим детерминатив 'ПСТ' функции подстановки. Конкретизация k 'ПСТ' $(\mathscr{E}_x)\mathscr{P}\perp$ дает значение полинома $\mathscr{P}$, когда переменная принимает значение $\mathscr{E}_x$. Воспользуемся схемой Горнера, основанной на том, что выражение

$$a_0 + \ldots + a_{n-1}x^{n-1} + a_n x^n$$

можно представить в виде

$$a_0 + \ldots + (a_{n-1} + a_n x)x^{n-1}$$

k 'ПСТ' $(e_x)e_1(e_b)(e_a) \sim$

k 'ПСТ' $(e_x)e_1(k$ 'РАП' $+ k$ 'РАП' $\times e_a, e_x \perp, e_b \perp)\perp$

k 'ПСТ' $(e_x)(e_a) \sim e_a$

Предоставляем читателю описать определенный интеграл k 'ИНТ' $(\mathscr{E}_a, \mathscr{E}_b)\mathscr{P}\perp$ от полинома $\mathscr{P}$ в пределах от $\mathscr{E}_a$ до $\mathscr{E}_b$.

Обратимся к задаче вычисления полиномов Лежандра. Они определяются формулами:

$$P_0(x) = 1$$

$$P_1(x) = x$$

$$P_n(x) = \frac{1}{n}[(2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x)]$$

Пусть конкретизация k 'ЛЕЖ' $\mathscr{N}\perp$ должна давать $\mathscr{N}$-й полином Лежандра. Как организовать вычисление? Было бы неразумно вычислять каждый раз заново понадобившийся нам полином. Поэтому составим программу таким образом, чтобы вычисленные однажды полиномы Лежандра хранились в списке, закопанном, скажем, под тем же именем 'ЛЕЖ'. При конкретизации k 'ЛЕЖ' $\mathscr{N}\perp$ полином, уже вычисленный раньше, будет извлекаться из списка, а еще не вычисленный будет вычисляться по рекуррентным формулам и заноситься в список. Список организуем таким образом:

$$(2, \mathscr{P}_2)(3, \mathscr{P}_3)\ldots(N, \mathscr{P}_N)$$

Начальные полиномы Лежандра включим прямо в рефал-предложения. Первый шаг составления программы выделяет случаи $n = 0$ и $n = 1$ и констатирует тот факт, что для $n \geqslant 2$ функция 'ЛЕЖ' требует дополнительной информации в виде списка 'ЛЕЖ':

k'ЛЕЖ' 0 ~ (1)

k'ЛЕЖ' 1 ~ (0) (1)

k'ЛЕЖ' e_n ~ k'ЛЕЖ 1' (e_n) k'ВК' 'ЛЕЖ' ⊥ ⊥

Функция 'ЛЕЖ 1' проверяет, вычислен ли нужный полином, и если нет, то обращается к функции 'ЛЕЖ 2', которая вычисляет полином по рекуррентной формуле. Заметим, что прежде чем начать конкретизацию функции 'ЛЕЖ 2', надо снова закопать список 'ЛЕЖ', ибо он понадобится для рекуррентной формулы.

k'ЛЕЖ' (e_n) e_1 (e_n, e_p) e_2 ~ e_p
  k'ЗК' 'ЛЕЖ' = e_1 (e_n, e_p) e_2 ⊥

k'ЛЕЖ1' (e_n) e_c ~ k'ЗК' 'ЛЕЖ' = e_c ⊥
  k'ЛЕЖ3' (e_n) k'ЛЕЖ2' e_n ⊥ ⊥

Обращение к функции 'ЛЕЖ 2' мы заключили в функциональные скобки с детерминативом 'ЛЕЖ 3'. Эта последняя функция нужна, чтобы должным образом распорядиться результатом работы функции 'ЛЕЖ 2'. Его надо, во-первых, выдать в качестве результата всей конкретизации, а во-вторых, снять с него копию и пополнить список 'ЛЕЖ' (поэтому в аргумент функции 'ЛЕЖ 3' пришлось еще включить e_n):

k'ЛЕЖ 3' (e_n) e_p ~ e_p k'ЗК' 'ЛЕЖ' = k'ВК' 'ЛЕЖ' ⊥ (e_n, e_p) ⊥

Функцию 'ЛЕЖ 2' можно описать одним предложением, в точности следующим рекуррентной формуле, однако, чтобы не запутаться в скобках, в качестве предварительного шага вычислим встречающиеся в этой формуле числовые коэффициенты

k'ЛЕЖ2' e_n ~ k'ЛЕЖ4' (e_n) (k'РАЦ' − e_n, 1 ⊥)
  (k'РАЦ' − k'РАЦ' × 2, e_n ⊥, 1 ⊥) ⊥

k'ЛЕЖ4' (e_n) (e_m) (e_k) ~
  k'УМН1' k'СВ' − (k'УМН' (k'ЛЕЖ' e_m ⊥)
  ((0) (e_k)) ⊥) (k'УМН1' k'ЛЕЖ' k'РАЦ' − e_n, 2 ⊥ ⊥
  (e_m) ⊥) ⊥ (1/e_n) ⊥

Печать начальных полиномов Лежандра будем получать путем конкретизации k'ПЧЛЖ' №1. В начале выдачи отпечатаем заголовок. Перед каждым полиномом будем отдельной строкой давать распечатку, указывающую его порядок. При редактировании печати надо пом-

нить, что каждое новое обращение к функции 'ПЕЧ' вызывает переход на новую строку.

k'ПЧЛЖ' e_n ~ k'ПЕЧ' ␣␣␣ КОЭФФИЦИЕНТЫ␣ ПОЛИНОМОВ␣ ЛЕЖАНДРА␣ ПОРЯДКА␣N: ⊥
     k'ПЧЛЖ1' ( ) e_n ⊥

k'ПЧЛЖ1' (e_n) e_n ~ k'ПЕЧ'␣␣␣ N = e_n ⊥
     k'ПЕЧ' k'ЛЕЖ' e_n ⊥ ⊥

k'ПЧЛЖ1' (e_n) e_m ~ k'ПЕЧ'␣␣␣N = e_n ⊥
     k'ПЕЧ' k'ЛЕЖ' e_n ⊥ ⊥
     k'ПЧЛЖ1' (k'РАЦ' + e_n ⊥) e_m ⊥

Выдача на печать, порождаемая этой программой, имеет такой вид:

```
КОЭФФИЦИЕНТЫ ПОЛИНОМОВ ЛЕЖАНДРА ПОРЯДКА   :
  N = 0
(1)
  N = 1
(0) (1)
  N = 2
(-1/2) (0) (3/2)
и т.д.
```

7. Сквозные просмотры

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

Вспомним функцию ликвидации повторных пробелов ∝ = k'ЛИКППР', которую мы определили в п.1:

∝e_1␣␣ e_2 ~ e_1∝␣e_2

∝e_1 ~ e_1

Эта функция оставляет нетронутыми те части аргумента, которые заключены в скобки. Мы хотим теперь модифицировать ее таким образом, чтобы обработке подвергались подвыражения на любой глубине скобочной структуры. Простейшее решение задачи – вставить между первым и вторым предложениями еще одно предложение, описывающее вхождение в скобки:

∝e_1␣␣ e_2 ~ ∝e_1 ⊥ ∝␣e_2
∝e_1 (e_2) e_3 ~ e_1 (∝e_2) ∝e_3
∝e_1 ~ e_1

Здесь мы вынуждены были изменить также и первое предложение, заключив в функциональные скобки переменную e_1 в правой части, ибо она должна быть еще просмотрена на предмет вхождения в

скобки. Переменную e_1 в правой части второго предложения выносим из конкретизационных скобок, так как она уже не содержит ни повторных пробелов, ни скобок.

При таком описании просмотр выражения выполняется не подряд, а сначала отщепляется отрезок до ближайшей пары пробелов, а затем уже осуществляется вхождение в скобки. Этот порядок можно изменить, если переставить местами первое и второе предложения. Легко заметить, что в обоих случаях производится двукратный просмотр обрабатываемого выражения. Можно построить и такую процедуру, чтобы просмотр был однократным. Для этого надо устранить открытые переменные:

$$\propto \sqcup\sqcup e_1 \sim \propto \sqcup e_1$$
$$\propto (e_1) e_2 \sim (\propto e_1) \propto e_2$$
$$\propto s_1 e_2 \sim s_1 \propto e_2$$
$$\propto \sim$$

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

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

Приступим к описанию нашей процедуры, которую обозначим через ϕ. Очевидно, на каждом этапе будет необходим список уже встреченных символов*. Поместим его в сумку, которую расположим в конце аргумента. Следовательно, необходимо ввести вспомогательную функцию ϕ^1:

$$\phi e_1 \sim \phi^1 e_1 (\ )$$

Если бы не было скобок, функция ϕ^1 описывалась предложениями:

§ 17.1 $\phi^1 s_a e_b (e_1 s_a e_2) \sim \phi^1 e_b (e_1 s_a e_2)$

§ 17.2 $\phi^1 s_a e_b (e_s) \sim s_a \phi^1 e_b (e_s s_a)$

§ 17.3 $\phi^1 t_s \sim$

(Просмотренную часть ϕ^1 оставляет вне функциональных скобок).

Что же должна делать функция ϕ^1, встречая скобки? Попробуем по аналогии с приведенным выше примером написать

$$\phi^1 (e_1) e_2 e_s \sim (\phi^1 e_1 t_s) \phi^1 e_2 t_s$$

Легко видеть, что это предложение не обеспечивает правильной работы. В самом деле, когда отработает конкретизация, введенная в скобки, список t_s будет уничтожен в соответствии с предложением § 17.3, и конкретизация функции ϕ^1 будет продолжаться со старым списком t_s, не учитывающим символов, появившихся в скобках. Здесь мы и встречаемся с проблемой обмена информацией между различными ветвями дерева. Ее можно решить несколькими способами.

Первый способ – "поднятие информации" из той ветви, где она получена, на следующий уровень скобочной структуры. В нашем случае это означает, во-первых, что мы не должны уничтожать список t_s после конца просмотра. Для этого § 17.3 надо переписать следующим образом:

$$\phi^1 t_s \sim t_s$$

Во-вторых, войдя в скобки, мы должны оставить на верхнем уровне функциональные скобки, которые обеспечат извлечение информации из объектных скобок и продолжение работы с этой информацией. Обозначим через ϕ^2 вспомогательную функцию, выполняющую эту задачу. Итак,

$$\phi^1 (e_1) e_2 t_s \sim \phi^2 (\phi^1 e_1 t_s) e_2$$

$$\phi^2 (e_1 t_s) e_2 \sim (e_1) \phi^1 e_2 t_s$$

Учитывая, что функция ϕ^1 теперь оставляет в конце список t_s, и для его уничтожения надо ввести еще одну вспомогательную функцию, получаем следующее описание:

**Просмотренная часть не может сама служить этим списком, так как не является выражением.* (footnote in italics)

§ 18 $\phi e_1 \sim \phi^3 \phi^1 e_1(\ )$

§ 19.1 $\phi^1 s_a e_b (e_1 s_a e_2) \sim \phi^1 e_b (e_1 s_a e_b)$

§ 19.2 $\phi^1 s_a e_b (e_s) \sim s_a \phi^1 e_b (e_s s_a)$

§ 19.3 $\phi^1 (e_1) e_2 t_s \sim \phi^2 (\phi^1 e_1 t_s) e_2$

§ 19.4 $\phi^1 t_s \sim t_s$

§ 20 $\phi^2 (e_1 t_s) e_2 \sim (e_1) \phi^1 e_2\ t_s$

§ 21 $\phi^3 e_1 t_s \sim e_1$

Как видим, оно получилось довольно громоздким.

Второй способ передавать информацию, минуя скобочные структуры – это закапывать и выкапывать информацию при переходе из одной ветви дерева в другую. Чтобы применить этот способ к данному случаю, введем имя 'СПИС' и определим функцию ϕ^1 так, чтобы она в конце каждого просмотра закапывала бы под этим именем терм t_s, содержащий список встречающихся символов. Тогда при каждом обращении к ϕ^1 (за исключением первого) надо выкапывать этот список и ставить его на должное место в аргументе. В конце работы надо выкопать и уничтожить список, чтобы он не занимал понапрасну память. Получаем следующее описание:

§ 22 $\phi e_1 \sim \phi^1 e_1(\ )$ k'УНИЧТ' k'ВК' 'СПИС'⊥⊥

§ 23.1 совпадает с 19.1

§ 23.2 совпадает с 19.2

§ 23.3 $\phi^1 (e_1) e_2 t_s \sim (\phi^1 e_1 t_s) \phi^1 e_2$ k'ВК' 'СПИС' ⊥⊥

§ 23.4 $\phi^1 t_s \sim$ k'ЗК' 'СПИС' $= t_s$ ⊥

§ 24 k'УНИЧТ' $e_1 \sim$

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

Третий, самый радикальный способ выполнения сквозных операций состоит в том, чтобы на время выполнения операции вообще ликвидировать скобочную структуру, заменив скобки специальными символами, которые до этой замены в выражении не встречались.

Пусть это будут 'L' и 'R'. Присвоим функции, которая осуществляет эту замену, детерминатив 'РАЗСК'. Она описывается весьма просто:

k 'РАЗСК' $e_1 (e_2) e_3 \sim e_1$ 'L' k 'РАЗСК' e_2 'R' $e_3 \perp$

k 'РАЗСК' $e_1 \sim e_1$

Легко видеть, что эта процедура требует однократного просмотра выражения. Противоположная процедура – спаривание скобок с детерминативом 'СПАРСК' – превращает символы 'L' и 'R' в должным образом связанные скобки. Она несколько сложнее, чем разъединение скобок, но, как мы сейчас увидим, также требует лишь однократного просмотра выражения.

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

Как подойти к этой задаче?

Начнем с исследования простейших примеров. Если между "скобкой" 'L' и "скобкой" 'R' нет никаких "скобок", то они, очевидно, образуют пару:

a 'L' $b\,c\,d$ 'R' e

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

a 'L' $b\,c$ 'L' $d\,e$ 'R' $f\,g\,h$ 'L' i 'R' j 'R' m

Тогда мы откладываем спаривание первой скобки и, встретив 'R', спариваем ее со второй. Идя дальше и снова (после h) встречая 'L', ставим ее на очередь и спариваем с первой же 'R'. И лишь встречая еще одну 'R', спариваем ее с первой 'L', которая стала теперь очередной.

Таким образом, мы пропускаем все 'L', а встретив 'R', спариваем ее с последней неспаренной 'L'. Просмотренная часть аргумента представляет собой объект, который может быть назван л е в о й мультискобкой. Он содержит некоторое число неспарен-

ных символов 'L', но не содержит 'R'. Спаренные 'L' и 'R' уже заменены на настоящие скобки. Например, когда стрелка просмотра находится в приведенном выше выражении между символами f и g, левая мультискобка есть

a 'L' $bc(de)f$

Число символов 'L' определяет глубину, на которой находится стрелка в скобочной структуре.

Теперь обсудим вопрос о синтаксисе объектов для написания рефал-программы. Мы могли бы хранить мультискобку в натуральном виде, помещая ее в сумку. Однако нетрудно видеть, что такой способ потребует многократного просмотра аргумента. Первый просмотр происходит, когда мы удлиняем мультискобку, и он, разумеется, неизбежен. Остальные просмотры производятся (справа налево) при поиске последнего в мультискобке символа 'L'. Как их избежать? Использовать для расчленения мультискобки на отрезки от 'L' до 'L' структурные скобки! В процессе просмотра мы, конечно, должны будем следить за подновлением мультискобки с сохранением необходимой структуры.

Мультискобку вида

$\mathscr{E}_1$ 'L' $\mathscr{E}_2$ 'L' $\mathscr{E}_3$

можно представить в виде структуры

$(\mathscr{E}_1)(\mathscr{E}_2)(\mathscr{E}_3)$

или в виде структуры

$((\mathscr{E}_1)\mathscr{E}_2)\mathscr{E}_3$

Вторая структура оказывается удобнее. Приняв ее, получаем следующее описание алгоритма спаривания скобок:

k 'СПАРСК' $e_x \sim \propto(\ )\, e_x$

$\propto(e_1)$ 'L' $e_2 \sim \propto((e_1))\, e_2$

$\propto((e_1)e_2)$ 'R' $e_3 \sim \propto(e_1(e_2))\, e_3$

$\propto(e_1)s_a e_2 \sim \propto(e_1 s_a)\, e_2$

$\propto(e_1) \sim e_1$

Задача 9.

Функция 'СПАРСК' описана в расчете на то, что "скобки" 'L' и 'R' образуют правильную скобочную структуру. Модифицировать это описание таким образом, чтобы в случае, когда 'L' и 'R' не образуют правильной структуры, печаталось сообщение об ошибке.

Аналогично левой мультискобке можно ввести понятие правой мультискобки и записывать структуру

$\mathscr{E}_1$ 'R' $\ldots\mathscr{E}_{n-1}$ 'R' $\mathscr{E}_n$

в виде

$\mathscr{E}_1(\ldots\mathscr{E}_{n-1}(\mathscr{E}_n)\ldots)$

Основываясь на этих понятиях, опишем еще один (четвертый) способ организации сквозных просмотров. Если мы движемся по выражению с заходом в каждую очередную пару скобок (т.е., по существу, рассматривая скобки как символы 'L' и 'R'), то в каждый момент времени просмотренная часть является левой мультискобкой, а непросмотренная – правой мультискобкой. Пусть, например, точка просмотра находится в скобочной структуре на глубине 2. Эту ситуацию можно изобразить так:

$$\mathcal{E}_1(\mathcal{E}_2(\mathcal{E}_3{\downarrow}\mathcal{E}_4)\mathcal{E}_5)\mathcal{E}_6.$$

(Здесь точка просмотра показана стрелкой). При нашем представлении мультискобок просмотренная часть будет иметь вид:

$$((\mathcal{E}_1)\mathcal{E}_2)\mathcal{E}_3$$

а непросмотренная:

$$\mathcal{E}_4(\mathcal{E}_5(\mathcal{E}_6))$$

Таким образом, просматриваемое подвыражение поднимается до основного уровня структуры.

Организуя сквозной просмотр, будем, как и прежде, заключать просмотренную часть в форматные скобки. Здесь, однако, возникает следующая трудность: надо уметь отличать скобки, введенные для изображения структуры правой мультискобки, от "натуральных", еще не подвергшихся обработке скобок, которые могут входить в непросмотренную часть. "Мультискобочные" скобки обладают той особенностью, что правая скобка заканчивает все выражение. Но ведь и исходное выражение может кончаться на правую скобку. Пусть, например, оно таково:

$$(a)\,bc\,(d)$$

В тот момент, когда точка просмотра находится непосредственно за символом c, аргумент имеет вид:

$$[(a)\,bc]\,(d)$$

(форматные скобки, отделяющие просмотренную часть, мы для ясности сделали квадратными).

Допустим теперь, что начальным выражением было

$$a\,(bc)\,d$$

Когда точка просмотра перепрыгивает символ c, возникает ситуация

$$[(a)\,bc]\,(d)$$

неотличимая от первой.

Чтобы преодолеть эту трудность, будем одновременно с заведением форматных скобок приписывать к просматриваемому выражению символ *; таким образом, окончание непросмотренной части на правую скобку будет однозначным признаком того, что она образует структуру мультискобки. В конце просмотра звездочка, естественно, должна быть удалена.

Вернемся к нашему примеру – функции ϕ, которая из всех вхождений каждого символа оставляет только первое

§ 25 Первая сумка заводится для просмотренной части, вторая – для списка встреченных символов

$\phi e_1 \sim \phi^1(\ )\ e_1 * (\ )$

§ 26.1 Окончание просмотра

$\phi^1 (e_1) * t_s \sim e_1$

§ 26.2 Символ, который уже встречался,

$\phi^1(e_1)s_a e_b (e_2 s_a e_3) \sim \phi^1 (e_1) e_b (e_2 s_a e_3)$

§ 26.3 Символ, который еще не встречался,

$\phi^1(e_1)s_a e_b(e_s) \sim \phi^1(e_1 s_a) e_b (e_s s_a)$

§ 26.4 Пара скобок, которая соответствует правой скобке в исходном выражении,

$\phi^1((e_1) e_2 (e_3) t_s \sim \phi^1(e_1 (e_2)) e_3 t_s$

§ 26.5 Левая скобка в исходном выражении

$\phi^1(e_1)(e_2) e_3 t_s \sim \phi^1((e_1)) e_2 (e_3) t_s$

8. Решения задач

1. $\ast e_1 s_a e_2 s_a e_3 \sim e_1 \ast s_a e_2 e_3$

$\ast e_1 \sim e_1$

По поводу эффективности этого алгоритма см.разд.2 и задачу 4.

2. а) $\gamma e_1 s_a \neg e_2 \sim \gamma e_1 e_2$

$\gamma e_1 \sim e_1$

Алгоритм требует многократного просмотра всего текста;

б) $\gamma^1(e_1 s_a) \neg e_2 \sim \gamma^1(e_1) e_2$

$\gamma^1(e_1) s_a e_2 \sim \gamma^1(e_1 s_a) e_2$

$\gamma^1(e_1) \sim e_1$

3. В первом формате:

$\psi (e_1 t_a e_2) t_a e_b \sim \psi (e_1 t_a e_2 \ast t_a) e_b$

$\psi (e_1) t_a e_b \sim \psi (e_1 \beta t_a) e_b$

$\psi (e_1) \sim e_1$

При использовании вторoго формата алгоритм остается таким же, как и при первом формате.

4. $\ast s_a e_1 \sim \ast^1(s_a) e_1$

$\ast \sim$

$\ast^1(s_a e_1) e_2 s_a e_3 \sim \ast^1(s_a e_1 e_2) e_3$

$\ast^1(s_a e_1) e_2 \sim s_a \ast e_1 e_2$

5. k 'И' $(F)\,e_1 \sim F$

k 'И' $(T)\,(s\,(TF)_1) \sim s_1$

k 'И' $(T)\,(e_1) \sim k\,e_1 \perp$

k 'И' $(e_1)\,(F) \sim F$

k 'И' $(e_1)\,(e_2) \sim k$ 'И' $(k\,e_1 \perp)\,(e_2) \perp$

6. k 'УПОР' $t_1 e_2 \sim k$ 'УП1' $(\ )\,t_1\ e_2 \perp$

k 'УП1' $(e_1)\,t_a t_b e_2 \sim k$ 'УП2' $(e_1)\,k$ 'УП2' $t_a t_b \perp e_2 \perp$

k 'УП1' $(e_1)\,t_a \sim e_1 t_a$

k 'УП2' (e_1) 'НОРМ' $t_a t_b e_2 \sim e_1 t_a t_b e_2$

k 'УП2' (e_1) 'ПЕРЕСТ' $t_b t_a e_2 \sim k$ 'УП1' $(t_1 t_b)\,t_a e_2 \perp$

7. $\sigma = k$ 'СИММ'

Обычный предикат:

$\sigma \sim T$

$\sigma\, s_a \sim T$

$\sigma\, s_a e_1 s_a \sim \sigma\, e_1$

$\sigma\, e_1 \sim F$

C-предикат:

$\sigma\, e_1 \sim \sigma^1(\ )\,e_1(\ )$

$\sigma^1(e_1)\,(e_2) \sim T\,e_1 e_2$

$\sigma^1\,(e_1)\,s_a(e_2) \sim T\,e_1 s_a e_2$

$\sigma^1(e_1)\,s_a e_2 s_a(e_3) \sim \sigma^1(e_1 s_a)\,e_2(s_a e_3)$

$\sigma^1(e_1)\,e_2(e_3) \sim F\,e_1 e_2 e_3$

8. $\delta = k$ 'ДИФ'

$\delta\, t_1 \sim (0)$

$\delta\, t_1 e_2 \sim \delta^1(1)\,e_2$

$\delta^1(e_n)\,(e_a)\,e_1 \sim (k$ 'РАЦ' $\times\, e_a,\ e_n \perp\ )$

$\delta^1(k$ 'РАЦ' $+\ e_n,\ 1 \perp\ )\ e_1 \perp\ \ \delta^1(e_n) \sim$

$\eta = k$ 'ИН'

$\eta\,(0) \sim 0$

$\eta\, t_1 e_2 \sim (0)\ t_1 \eta^1(2)\,e_2$

$\eta^1(e_n)\,(e_a)\,e_1 \sim (k$ 'РАЦ' $/\,e_a,\ e_n \perp\ )$

$\eta^1(e_n) \sim$

9. k'СПАРСК' $e_x \sim *(\ )\ e_x$

$*(e_1)$ 'L' $e_2 \sim *($'L' $(e_1))\, e_2$

$*($'L' $(e_1)\, e_2)$ 'R' $e_3 \sim *(e_1\, (e_2))\, e_3$

$*(e_1)$ 'R' $e_3 \sim k$ 'ПЕЧ' ОШИБКА: НЕПАРНАЯ ПРАВАЯ СКОБКА e_1 'R' $\perp$

$*(e_1)\, s_a e_2 \sim *(e_1 s_a)\, e_2$

$*($'L' $(e_1)\, e_2) \sim k$ 'ПЕЧ' ОШИБКА: НЕПАРНЫЕ ЛЕВЫЕ СКОБКИ 'L' $(e_1) \perp$

$*(e_1) \sim e_1$

III. ТЕХНИКА ИСПОЛЬЗОВАНИЯ РЕФАЛА

1. Трансляционные задачи

В этом разделе разберем несколько задач, возникающих при создании трансляторов с формализованных языков программирования. Начнем с такой хорошо известной задачи как преобразование обычной записи арифметических выражений в польскую форму записи. Напомним, что если в обычной записи арифметическая функция (операция) F от аргументов A_1 и A_2 записывается в виде

$$(A_1)F(A_2) \qquad (*)$$

и дополнительно вводятся правила старшинства операций, позволяющие опускать часть скобок, то в польской форме то же выражение имеет вид:

$$F\, A_1, A_2 \qquad (**)$$

(для простоты ограничиваемся бинарными операциями, так что $-x$ надо представлять в виде $0-x$). В качестве элементарных операндов будем употреблять идентификаторы и числа; последние – в любой форме записи, но с тем ограничением, чтобы в записи не использовались знаки арифметических действий (+, –, ×, /, ↑), скобки и запятая.

Если бы правил старшинства не было, и все скобки, требуемые формой (*) должны были присутствовать, то перевод выражения в польскую форму (**) мог быть осуществлен такой функцией:

$\phi(e_1)\, s_f(e_2) \sim s_f \phi e_1 \perp, \phi e_2 \perp$

$\phi e_a \sim e_a$

Согласно правилам старшинства знаки "+" и "–" связывают "рыхлее" всего, поэтому при синтаксическом анализе выражения мы должны прежде всего разбить его на отрезки от одного знака аддитивной операции до другого (игнорируя, разумеется, знаки операций, входящие в скобки). Действия одного старшинства выполняются слева направо, т.е. запись

$a + b - c - d$

означает

$((a+b)-c)-d$

Поэтому на верхнем уровне скобочной структуры оказывается последний знак операции. Так как запись первичных операндов не содержит знаков операций, мы можем обнаружить нужный нам знак операции просмотром справа налево. Первое предложение описания функции

$\phi = k$ 'ПОЛ'

принимает вид

$(R) \phi e_1 s(+-)_f e_2 \sim s_f \phi e_1 \perp, \phi e_2 \perp$

Если в выражении не оказалось знаков аддитивной операции, то надо переходить к анализу на следующих уровнях старшинства. Так возникают еще два предложения

$(R) \phi e_1 s(\times /)_f e_2 \sim s_f \phi e_1 \perp, \phi e_2 \perp$

$(R) \phi e_1 \uparrow e_2 \sim \uparrow \phi e_1 \perp, \phi e_2 \perp$

Теперь пришла очередь скобок, которые связывают сильнее, чем все знаки операций:

$\phi(e_1) \sim \phi e_1 \perp$

Если ни одно из предыдущих предложений не сработало, то аргумент функции есть первичный операнд, и его надо оставить без изменений:

$\phi e_a \sim e_a$

Итого – пять предложений. Это описание весьма компактно, хотя не вполне эффективно, о чем будет сказано ниже.

Рассмотрим теперь задачу перевода на машинный язык операторов в языке АЛГОЛ-60, имеющих вид:

<простая переменная> := <арифметическое выражение>

Нашей целью является описать программирование арифметических выражений, поэтому упростим детали, имеющие с этой точки зрения второстепенное значение. Индексные скобки [ ] заменим на обычные круглые скобки; в качестве ограничителей параметров разрешим использовать только запятые; запретим использовать строки в качестве фактических параметров. Ограничители if и then снабдим дополнительными скобками: левой после if и правой перед then . Это, конечно, связано с тем, что синтаксически эти символы выступают в АЛГОЛе в роли скобок, поэтому от указанного добавления анализ условных выражений заметно упрощается. Еще одно небольшое ограничение будет введено ниже. В остальном мы оставим синтаксис арифметических выражений неизменным, но до конца доведем только программирование, связанное с простыми переменными и числами.

Вычислительная машина, для которой будем составлять программу, пусть будет одноадресной; идентификаторы переменных будут служить в качестве символических адресов; разрешается также использовать литеральные константы в виде:

= <число>

где <число> записывается так, как это предусмотрено в АЛГОЛе. Все числа представляются в машине единственным способом, и различия между типами real и integer игнорируются. Машина имеет следующие команды:

СЧ – считать число по заданному адресу в сумматор;

ЗП – записать число из сумматора по заданному адресу;

СЛ, ВЫЧ, УМН, ДЕЛ, СТЕП – выполнить соответствующее действие над содержимым сумматора (первый операнд) и числом по указанному адресу (второй операнд);

ИДТИ – безусловный переход по указанному адресу;

ИДОТР – переход, если число в сумматоре отрицательно;

ИДНОТ – переход, если число в сумматоре неотрицательно;

ИДНУЛ – переход, если в сумматоре нуль.

Код операции будем отделять от операнда запятой, в конце команды будем ставить точку с запятой, а метку отделять двоеточием. Возьмем в качестве примера оператор

$x := \text{if } (x \geq y) \text{ then } (x - y) \times 2.5 \text{ else } 0$

Его перевод на язык машины таков:

```
СЧ, x;
ВЫЧ, y;            M1: СЧ, = 0;
ИДОТР, M1;         M2: ЗП, = x;
СЧ, x;
ВЫЧ, y;
УМН, = 2.5;
ИДТИ, M2;
```

Эта программа получена независимым переводом условия и вариантов арифметического выражения. Если связать эти части процесса, можно было бы несколько улучшить программу, выкинув две команды, повторно вычисляющие разность $x - y$. Однако это потребовало бы значительного усложнения алгоритма трансляции. Наш алгоритм таких оптимизаций выполнять не будет.

Каждому арифметическому выражению можно сопоставить программу (перевод выражения), которая вырабатывает на сумматоре значение выражения.

Сначала надо сформулировать общие принципы перевода простого (безусловного) арифметического выражения. Будем считать, что бокового эффекта (изменения значения одних подвыражений при вы-

числении других подвыражений) не существует, поэтому порядок вычисления подвыражений можно выбирать произвольно, руководствуясь лишь соображениями эффективности программы. Для синтаксического анализа выражения остаются в силе те соображения, которые были высказаны в связи с переводом в польскую форму. Пусть выделен знак операции F, образующий высший уровень структуры выражения (т.е. изображающий действие, выполняемое последним); следовательно, выражение можно представить в виде

$\mathscr{E}_1 F \mathscr{E}_2$

Допустим, что и $\mathscr{E}_1$, и $\mathscr{E}_2$ – сложные выражения, требующие использования сумматора для своего вычисления. Тогда надо сначала вычислить значение $\mathscr{E}_2$, запомнить его в рабочей ячейке, затем вычислить $\mathscr{E}_1$ и производить действие F. Предпочтительность такого порядка связана с тем, что арифметические команды нашей машины всегда имеют сумматор в качестве первого аргумента, а ячейку с указанным адресом – второго. Если бы мы сначала вычисляли $\mathscr{E}_1$, пришлось бы, как нетрудно сообразить, выполнить две лишних пересылки: в рабочую ячейку и обратно в сумматор. Итак, перевод приведенного выражения имеет вид:

перевод $\mathscr{E}_2$

ЗП, R_i

перевод $\mathscr{E}_1$

код операции F, R_i

Здесь R_i – рабочая ячейка. Очевидно, число рабочих ячеек, которые могут понадобиться в процессе вычисления, не ограничено. Отведено для них поле и пометим первую ячейку идентификатором РАБ. Тогда адрес рабочей ячейки R_i будет РАБ$+i$ при $i = 0, 1, 2, \ldots$ Переводя подвыражение $\mathscr{E}_1$, мы должны учесть, что рабочая ячейка R_i уже занята. Поэтому функция "перевод" должна иметь еще один аргумент – смещение i в поле рабочих ячеек. Итак, перевод $(i)\mathscr{E}$ есть перевод выражения $\mathscr{E}$ с использованием рабочих ячеек, начиная с ячейки РАБ $+ i$. Теперь основное правило перевода можно сформулировать следующим образом (почти на рефале):

перевод $(i)\ \mathscr{E}_1 F \mathscr{E}_2$

есть

перевод $(i)\mathscr{E}_2$

ЗП, РАБ $+ i$;

перевод $(i+1)\ \mathscr{E}_1$

код операции F, РАБ$+ i$;

Если $\mathscr{E}_2$ является элементарным операндом, т.е. идентификатором или числом, то можно сделать более простой перевод, а именно:

перевод $(i)\ \mathscr{E}_1$

код операции $F, \mathscr{E}_2$;

Наконец, перевод элементарного операнда $\mathscr{E}$ есть просто СЧ, $\mathscr{E}$;

В большинстве случаев, руководствуясь этими правилами, мы получим наилучший возможный перевод. Например, выражение

$$a \times b + (c - d) \times e$$

переводится так:

СЧ, c ;
ВЫЧ, d ;
УМН, e ;
ЗП, РАБ;
СЧ, a ;
УМН, b ;
СЛ, РАБ;

Однако рассмотрим такое выражение

$a + b \times c$

Пользуясь нашими правилами, получаем перевод:

СЧ, b ;
УМН, c;
ЗП, РАБ;
СЧ, a ;
СЛ, РАБ;

В то же время существует лучший перевод, использующий коммутативность сложения:

СЧ, b ;
УМН, c ;
СЛ, a ;

Поэтому добавим к нашим правилам еще одно: при коммутативной операции F и элементарном операнде $\mathscr{E}_1$ выражение $\mathscr{E}_1 F \mathscr{E}_2$ надо переводить как $\mathscr{E}_2 F \mathscr{E}_1$, т.е.

перевод (i) $\mathscr{E}_2$

код операции $F, \mathscr{E}_1$;

В отличие от простых выражений, которые мы переводили выше в польскую форму, арифметические выражения в АЛГОЛе имеют довольно сложный синтаксис, поэтому, прежде чем приступать к собственно программированию, подвергнем выражение предварительной синтаксической обработке. В первую очередь мы руководствуемся следующими двумя соображениями: 1) запись чисел в АЛГОЛе может содержать знаки + и –; необходимо распознать их с тем, чтобы при программировании не принять за знаки операций; 2) так как при составлении программы мы должны отличать элементарные операнды от неэлементарных, желательно выделить их и снабдить синтаксическими указателями, для легкого распознавания. Чтобы удовлетворить обоим требованиям, будем в процессе предварительного синтаксического анализа помечать идентификаторы и числа звездоч-

кой и заключать в скобки; кроме того, в случае чисел будем непосредственно за звездочкой ставить знак равенства, чтобы в процессе перевода уже не надо было различать переменные и числа, а программировать все элементарные операнды единообразно. Например, выражение

$x/(\mathrm{sum} + 0.544_{10} - 6)$

после предварительной обработки должно будет иметь вид:

$(*x)/((*\ \mathrm{sum}) + (* = 0.544_{10} - 6))$

Итак, для функции 'АВЫР' – перевод арифметического выражения – запишем:

k 'АВЫР' $e_1 \sim k$ 'АВЫР1' (0) k 'ПРОБ' $e_1 \perp\perp$

где 'ПРОБ' – предварительная обработка выражения.

Вообще говоря, описывать алгоритм обработки объектов, имеющих заданную синтаксическую структуру, надо так: положить перед собой формальное описание синтаксиса и в соответствии с ним составлять предложения на рефале. Однако при описании функции 'ПРОБ' воспользуемся тем обстоятельством, что эта функция имеет дело лишь с идентификаторами и числами, поэтому глобальный синтаксический разбор не требуется. Что же касается выделения идентификаторов и чисел, то тут мы будем действовать в строгом соответствии с формальным синтаксисом.

Будем осуществлять синтаксический разбор, двигаясь слева направо. При этом каждой синтаксической конструкции (за исключением простейших) удобно сопоставить рефал-функцию, "отщепляющую" экземпляр этой конструкции от левого конца обрабатываемого объекта. Такие функции всегда будут иметь формат

$k\ F(\mathscr{E}_1)\mathscr{E}_2 \perp$

где $\mathscr{E}_1$ – обработанная часть объекта, а $\mathscr{E}_2$ – еще не обработанная. Действие функции $\mathscr{F}$ должно сводиться к отщеплению от $\mathscr{E}_2$ конструкции типа $\mathscr{F}$ и перемещению ее в скобки. Например, функция 'ИДЕНТ' (идентификатор) должна быть определена так, что выполнение конкретизации

k 'ИДЕНТ' ( ) sum1 + sum2 $\perp$

дает

(sum1) + sum2

Идентификатор описан в АЛГОЛе следующей бэкусовской формой:

<идентификатор> : : = <буква>|<идентификатор> <буква>
|<идентификатор> < цифра>

Для целей анализа слева направо удобнее переписать это определение так (мы вводим здесь те сокращения, которые будем использовать в рефал-программе):

<идент> : : = <буква> <бцх>

<бцх> ::= <буква> <бцх> | <цифра> <бцх> | <пусто>
(<бцх> – "буквенно-цифровой хвост").

Соответствующие предложения на рефале:

k'ИДЕНТ' $(e_1)\, s$ 'БУКВА'$_a e_2 \sim k$'БЦХ' $(e_1 s_a)\, e_2 \perp$

k'БЦХ' $(e_1)\, s$ 'БУКВА'$_a e_2 \sim k$'БЦХ' $(e_1 s_a)\, e_2 \perp$

k'БЦХ' $(e_1)\, s$ 'ЦИФРА'$_a e_2 \sim k$'БЦХ' $(e_1 s_a)\, e_2 \perp$

k'БЦХ' $e_x \sim e_x$

Превращая правила бэкусовской формы в предложения на рефале, отдельные символы, с которых начинаются правые части правил, вводим в левые части предложений, а для синтаксических понятий, определяемых с использованием рекурсии, вводим соответствующие рекурсивные функции и обращаемся к ним в правой части предложения. Если правая часть правила содержит несколько рекурсивных понятий подряд, например:

<$\mathcal{F}$> ::= <$\mathcal{F}_1$> <$\mathcal{F}_2$> <$\mathcal{F}_3$>

то в правой части рефал-предложения мы должны записать последовательное обращение к соответствующим отщепляющим функциям:

$k\mathcal{F}\, e_x \sim k\mathcal{F}_3\, k\mathcal{F}_2\, k\mathcal{F}_1\, e_x \perp\perp\perp$

Синтаксис числа в АЛГОЛе (несколько модифицированный для наших целей по сравнению с "Формальным описанием"):

<ЧБЗ> ::= $_{10}$<целое> | <цифра> <цх> <порп> | <цифра> <цх> <прдрп> <порп>

<целое> ::= <знак + –> <цифра> <цх> | <цифра> <цх>

<знак + –> ::= + | –

<цх> ::= <цифра> <цх> | <пусто>

<порп> ::= $_{10}$<целое> | <пусто>

<прдрп> ::= . <цифра> <цх> | <пусто>

(Сокращения: чбз – число без знака, цх – цифровой хвост, порп – порядок или пусто, прдрп – правильная дробь или пусто).

Соответствующие предложения на рефале:

k'ЧБЗ' $(e_1)_{10}\, e_2 \sim k$'ЦЕЛОЕ'$(e_{1\,10})\, e_2 \perp$

k'ЧБЗ' $(e_1)\, s$'ЦИФРА'$_a e_2 \sim k$'ПОРП' k'ЦХ' $(e_1 . s_a)\, e_2 \perp\perp$

k'ЧБЗ' $(e_1)\, s$'ЦИФРА'$_a e_2 \sim k$'ПОРП' k'ПРДРП' k'ЦХ' $(e_1 s_a)\, e_2 \perp\perp\perp$

k'ЦЕЛОЕ' $(e_1)\, s\, (+-)_a\, s$ 'ЦИФРА'$_b e_2 \sim k$'ЦХ' $(e_1 s_a s_b)\, e_2 \perp$

k'ЦЕЛОЕ' $(e_1)\, s$ 'ЦИФРА'$_a e_2 \sim k$'ЦХ' $(e_1 s_a)\, e_2 \perp$

k'ЦХ' $(e_1)\, s$ 'ЦИФРА'$_a e_2 \sim k$'ЦХ' $(e_1 s_a)\, e_2 \perp$

k'ЦХ' $e_x \sim e_x$

k'ПОРП' $(e_1)_{10}\, e_2 \sim k$'ЦЕЛОЕ' $(e_{1\,10})\, e_2 \perp$

k 'ПОРП' $e_x \sim e_x$

k 'ПРДРП' $(e_1).s$ 'ЦИФРА'$_a$ $e_2 \sim k$ 'ЦХ' $(e_1.s_a)e_2 \perp$

k 'ПРДРП' $e_x \sim e_x$

Согласно синтаксису АЛГОЛа первичным арифметическим выражением может быть лишь <число без знака>, но не <число>, поэтому мы даже не завели функции для отщепления объекта <число>. Эта черта синтаксиса АЛГОЛа отражает обычное понимание выражений, согласно которому, например, в выражении

$x - 24$

знак минус есть знак операции, а не знак числа 24, а конструкции вида

$x + - 24$

запрещены. Знаки "+" и "–" разрешаются перед первым термом арифметического выражения и относятся к терму в целом. Поэтому, если действовать в строгом соответствии с формальным синтаксисом, оператор

$x := -24 \times y$;

надо переводить так:

СЧ, y ;
УМН, = 24;
ЗП, РАБ;
СЧ, = 0;
ВЫЧ, РАБ;
ЗП, x ;

Введя отрицательную константу – 24, можно этот перевод сократить на целых три команды, и так как этот случай весьма распространен, такая оптимизация необходима. Заметим, что она становится возможной вследствие свойств умножения и деления относительно перемены знака одного из операндов; в выражении

$- 2 \uparrow n$

знак минус нельзя включить в числовую константу.

Итак, процедура 'ПРОБ' будет включать знак минус в числовую константу, если она стоит на первом месте в выражении, и за ней не следует знак возведения в степень. В противном случае будем приписывать перед выражением нулевой терм. Знак плюс в начале выражения можно вообще отбросить. Следовательно, после предварительной обработки останутся только бинарные операции.

Опишем функцию 'ПРОБ':

$\alpha = k$ 'ПРОБ'

$\alpha - e_1 \sim \alpha^1 \beta e_1$

$\alpha + e_1 \sim \alpha^2 \beta e_1$

$\alpha e_1 \sim \alpha^2 \beta e_1$

Здесь функция β будет производить отщепление идентификатора или числа; в случае, когда ни то, ни другое невозможно, она будет приписывать спереди знак $\neg$. Кроме того, при отщеплении числа она будет вписывать признак литеральной константы =.

$$\beta s\,'\text{БУКВА}'_a\, e_1 \sim k\,'\text{БЦХ}'\,(s_a)\, e_1 \perp$$
$$\beta s\,'\text{ЧЗ}'_a\, e_1 \sim k\,'\text{ЧБЗ}'\,(=)\, s_a e_1 \perp$$
$$\beta e_1 \sim \neg e_1$$
$$k\,'\text{ЧЗ}' \sim 0123456789._{10}$$

('ЧЗ' – числовой знак).

Функция α^1 описывает обработку начальной части арифметического выражения и заводится исключительно ради манипуляций со знаком минус. Функция α^2 описывает продолжение обработки. Обе функции предполагают, что вначале к аргументу применена функция β, поэтому в правых частях предложений они встречаются только в комбинации $\alpha^i\beta$

$$\alpha^1 (=e_1) \uparrow e_2 \sim (* = 0) - (* = e_1) \uparrow \alpha^2 \beta e_2$$
$$\alpha^1 (= e_1)\, e_2 \sim (* = -e_1)\, \alpha^2 \beta e_2$$
$$\alpha^1 (e_1) e_2 \sim (* = 0) - (* e_1)\, \alpha^2 \beta e_2$$
$$\alpha^2 (e_1) e_2 \sim (* e_1)\, \alpha^2 \beta e_2$$
$$\alpha^2 \neg \sim$$
$$\alpha^2 \neg (e_1) e_2 \sim (\alpha e_1)\, \alpha^2 \beta e_2$$
$$\alpha^2 \neg s(,\text{else})_a e_2 \sim s_a \alpha e_2$$
$$\alpha^2 \neg s_a e_2 \sim s_a \alpha^2 \beta e_2$$

Таким образом, обработка является сквозной; ей подвергаются также списки индексных выражений и фактических параметров; сделанные нами синтаксические упрощения позволяют не отличать эти последние друг от друга и от вхождения в скобки, указывающие порядок действий в выражениях.

Приступим к описанию функции 'АВЫР 1'.

Открыв формальное описание АЛГОЛа, выпишем синтаксическое определение арифметического выражения:

<авыр> : : = <прав>

| if (<логвыр>) then <прав> else <авыр>

(прав – простое арифметическое выражение, логвыр – логическое выражение).

Соответствующие предложения на рефале:

k'АВЫР1' t_i if (e_c) then e_1 else e_2~ ...

k'АВЫР1' e_x~ k'ПРАВ' e_x ⊥

Логическое выражение может быть, согласно синтаксису АЛГОЛа, условным, поэтому мы снабдили ограничители if и then скобками, чтобы легко распознавать парный символ. Отрезок между then и else должен быть простым выражением и, следовательно, не должен содержать символов else ; поэтому его выделение производится с помощью открытой переменной e_1 . Вместо правой части первого предложения мы пока поставили многоточие и займемся ею ниже. Опишем сначала функцию 'ПРАВ' – программирование простого арифметического выражения.

Согласно синтаксису, простое арифметическое выражение имеет трехуровневую структуру, отражающую три уровня старшинства операций: терм, множитель и первичное выражение. Мы будем производить синтаксический анализ так же, как при переводе в польскую форму записи, с той лишь разницей, что каждому уровню сопоставим отдельную функцию: 'ТЕРМ', 'МНОЖ' и 'ПЕРВ' , соответственно. Это делается из соображений эффективности программы. При использовании единственной функции мы вынуждены перед применением каждого предложения, кроме первого, убеждаться в неприменимости предыдущих предложений, что не всегда необходимо. В данном случае это было бы особенно неприятно, так как на каждом уровне анализа надо, кроме основного случая, рассматривать еще один или два оптимизационных.

Синтаксис первичного выражения в АЛГОЛе определяется следующими правилами:

<первичное выражение> : : = <число без знака> | <переменная> |< указатель функции>| (<арифметическое выражение>)

<переменная> : : = <простая переменная> |< переменная с индексами>

<простая переменная> : : = < идентификатор >

<переменная с индексами> : : = < идентификатор массива> (<список индексов>)

<указатель функции> : : = < идентификатор процедуры > <совокупность фактических параметров >

<совокупность фактических параметров >: : = < пусто> |(<список фактических параметров >)

Из этих правил видно, что указатель функции с пустой совокупностью фактических параметров не отличается синтаксически (точнее, в бесконтекстном скелете синтаксиса) от простой переменной. Различие между ними обнаруживается при анализе описаний. В настоящем трансляторе функция 'ПРОБ' должна не только выделять

идентификаторы, но и заменять их на адресные выражения с учетом блочной структуры. При этом используется информация, полученная при обработке описаний. Функция 'ПРОБ' должна также оставлять при адресном выражении информацию о типе величин и другие признаки, которые могут понадобиться при программировании, в частности должно появиться различие между простой переменной и указателем функции. Так как мы игнорируем обработку описаний, запретим использование функций без параметров. Тогда конструкция (* 𝓔), входящая в качестве операнда, всегда означает элементарный операнд: простую переменную или число (которое вследствие введенной нами оптимизации может быть как без знака, так и со знаком).

Итак, вспоминая схему перевода, получаем следующее описание функции 'ПРАВ':

k 'ПРАВ' $(e_i)\, e_1 s(+-)_f (*\, e_2) \sim k$ 'ПРАВ' $(e_i)\, e_1 \perp k$ 'КОД' $s_f \perp e_2$;

k 'ПРАВ' $(e_i)(*\, e_1) + e_2 \sim k$ 'ПРАВ' $(e_i) e_2 \perp$ СЛ, e_1;

(R) k 'ПРАВ' $(e_i) e_1 s(+-)_f e_2 \sim k$ 'ТЕРМ' $(e_i) e_2 \perp$ ЗП, РАБ+ e_i ;

k 'ПРАВ' (k 'ПЛЮС1' $e_i \perp$) $e_1 \perp$
    k 'КОД' $s_f \perp$ РАБ+ e_i ;

k 'ПРАВ' $e_x \sim k$ 'ТЕРМ' $e_x \perp$

Здесь первое предложение соответствует второму случаю схемы, второе – третьему, а третье – первому (основному). В третьем предложении проекция переменной e_2 не содержит знаков аддитивной операции на основном уровне скобочной структуры, поэтому ее уже можно переводить как <терм>. Это и есть та экономия, которая получается из-за заведения нескольких функций. Функция 'КОД' описывается предложениями:

k 'КОД' $+ \sim$ СЛ,

k 'КОД' $- \sim$ ВЫЧ,

k 'КОД' $\times \sim$ УМН,

k 'КОД' $/ \sim$ ДЕЛ,

k 'КОД' $\uparrow \sim$ СТЕП,

Функция 'ПЛЮС 1' увеличивает целое десятичное число на единицу. Ее нетрудно описать на рефале, но лучше реализовать как машинную операцию.

Функции 'ТЕРМ' и 'МНОЖ' описываются аналогично функции 'ПРАВ':

k 'ТЕРМ' $(e_i) e_1 s(\times/)_f (*e_2) \sim k$ 'ТЕРМ' $(e_i) e_1 \perp k$ 'КОД' $s_f \perp e_2$;

k 'ТЕРМ' $(e_i)(*\, e_1) \times e_2 \sim k$ 'ТЕРМ' $(e_i)\, e_2 \perp$ УМН, e_1;

(R) k 'ТЕРМ' $(e_i) e_1 s(\times/)_f e_2 \sim k$ 'МНОЖ' $(e_i) e_2 \perp$ ЗП, РАБ + e_i ;

k 'ТЕРМ' (k 'ПЛЮС1' e_i ⊥) e_1 ⊥ k 'КОД' s_f ⊥РАБ + e_i ;

k 'ТЕРМ' e_x ~ k 'МНОЖ' e_x ⊥

k 'МНОЖ' (e_i) e_1 ↑ (* e_2) ~ k 'МНОЖ' (e_i) e_1 ⊥ СТЕП, e_2 ;

k 'МНОЖ' (e_i) e_1 ↑ e_2 ~ k 'ПЕРВ' (e_i) e_2 ⊥ ЗП, РАБ+ e_i ; k 'МНОЖ' (k 'ПЛЮС1' e_i ⊥) e_1 ⊥ СТЕП, РАБ + e_i ;

k 'МНОЖ' e_x ~ k 'ПЕРВ' e_x ⊥

Для функции 'ПЕРВ' нет необходимости различать простую переменную и число. Указатель функции и переменная с индексами неразличимы без обращения к описаниям, и мы поручим программирование этого случая функции 'ПИУФ' (переменная с индексами или указатель функции). Эта функция должна порождать программу занесения на сумматор значения соответствующего первичного выражения. Описание функции 'ПЕРВ' принимает вид:

k 'ПЕРВ' (e_i) (* e_1) ~ СЧ, e_1 ;

k 'ПЕРВ' (e_i) (* e_1)(e_2) ~ k 'ПИУФ' (e_i) (e_1) (e_2) ⊥

k 'ПЕРВ' (e_i) (e_1) ~ k 'АВЫР1' (e_i) e_1 ⊥

Применяя в последнем предложении функцию 'АВЫР1', а не 'АВЫР', мы учитываем, что функция 'ПРОБ' была определена как сквозная, так что предварительная обработка проекции e_1 уже выполнена. Эти предложения заканчивают описание программирования простых (безусловных) арифметических выражений.

Для перевода условных выражений введем функцию 'УП' – условный переход, такую, что конкретизация

k 'УП' ($\mathcal{I}$) ($\mathcal{G}$) 'НА' $\mathcal{M}$ ⊥

порождает программу, которая вычисляет логическое выражение $\mathcal{G}$ и если оно имеет значение true (т.е. условие выполнено), передает управление на метку $\mathcal{M}$. Для вычисления логического выражения могут понадобиться рабочие ячейки, поэтому в аргумент введено число $\mathcal{I}$ – номер первой свободной рабочей ячейки. Перевод условного выражения

if ($\mathcal{G}$) then $\mathcal{E}_1$ else $\mathcal{E}_2$

построим по схеме:

k 'УП' ($\mathcal{I}$) ($\mathcal{G}$) 'НА' $\mathcal{M}_1$ ⊥

k 'АВЫР1' ($\mathcal{I}$) $\mathcal{E}_2$ ⊥

ИДТИ, $\mathcal{M}_2$;

$\mathcal{M}_1$: k 'ПРАВ' ($\mathcal{I}$) $\mathcal{E}_1$ ⊥

$\mathcal{M}_2$:

Здесь $\mathcal{M}_1$ и $\mathcal{M}_2$ – метки, которые должны порождаться в процессе перевода, причем таким образом, чтобы при каждом применении этого правила они были бы новыми и чтобы они не могли совпасть ни с одной из меток, которые будут порождаться при использовании других правил, требующих введения меток. Будем строить метки, приписывая к букве M десятичное число $\mathcal{N}$. Максимальное использованное к данному моменту число $\mathcal{N}_{max}$ будем хранить закопанным под именем N. В самом начале процесса трансляции надо будет закопать под именем N число 0; в свое время мы напишем соответствующее предложение. Введем функцию 'НОВЧ' – новое число, которая под произвольным указанным символом будет закапывать наименьшее неиспользованное число, одновременно увеличивая $\mathcal{N}_{max}$ на единицу:

k 'НОВЧ' $s_a \sim k$ 'НОВЧ1' $s_a k$ 'ПЛЮС1' k 'ВК' N ⊥⊥⊥

k 'НОВЧ1' $s_a e_n \sim k$ '3К' $s_a = e_n$ k '3К' $N = e_n$ ⊥

Функция 'МЕТКА' конструирует метку, используя число, закопанное под указанным символом:

k 'МЕТКА' $s_a \sim k$ 'МЕТ1' $s_a k$ 'ВК' s_a ⊥⊥

k 'МЕТ1' $s_a e_n \sim M e_n k$ '3К' $s_a = e_n$⊥

Теперь, выполнив несколько конкретизаций

k 'НОВЧ' 1 ⊥ k 'НОВЧ' 2 ⊥ ...

можем использовать в качестве меток $\mathcal{M}_1, \mathcal{M}_2$ и т.д. выражения

k 'МЕТКА' 1⊥ , k 'МЕТКА' 2⊥

и т.д.

Однако не все проблемы этим исчерпаны. Действительно, допустим, что в соответствии с приведенной выше схемой мы записали правую часть предложения таким образом:

k 'НОВЧ' 1 ⊥ k 'НОВЧ' 2 ⊥

k 'УП' $t_i(e_c)$ 'НА' k 'МЕТКА' 1 ⊥⊥

k 'АВЫР1' $t_i e_2$ ⊥

ИДТИ, k 'МЕТКА' 2 ⊥ ;

k 'МЕТКА' 1 ⊥ : k 'ПРАВ' $t_i e_1$ ⊥

k 'МЕТКА' 2 ⊥:

Такое предложение не обеспечит построения правильной программы, ибо между двумя конкретизациями, порождающими первую метку, будет выполняться конкретизация функций 'УП' и 'АВЫР1', которая также может потребовать конструирования новых меток, вследствие чего число, закопанное под символом 1, изменится. То же относится и ко второй метке. Чтобы обеспечить правильное построе-

ние программы, надо было бы сначала выполнить все конкретизации, создающие метки, а потом уже продолжать программирование, обращаясь рекурсивно к функциям 'УП', 'АВЫР1', 'ПРАВ'. Но порядок выполнения конкретизаций в рефале определен однозначно. Как же быть?

Выход из затруднения осуществляется путем "пассивирования" и "активирования" знаков конкретизации. Заменим те знаки конкретизации, которые подлежат учету во вторую очередь, символом '*K*', сопровождаемым левой скобкой, а соответствующие конкретизационные точки – на правую скобку. И заключим всю правую часть в конкретизационные скобки с детерминативом 'АКТ' – активизация символов '*K*'. Эта функция превращает каждую конструкцию вида

$'K'\ (\mathcal{E})$

в конструкцию

$k\,\mathcal{E}\perp$

и выполняет конкретизацию.

Функция 'АКТ' универсальная и описывается следующими предложениями:

$k\,'\text{АКТ}'\ e_1\,'K'\,(e_2)\,e_3 \sim k\,'\text{АКТ1}'\ e_1{\perp}k k'\text{АКТ}'\ e_2{\perp}{\perp}\ k\,'\text{АКТ}'\ e_3\perp$

$k\,'\text{АКТ}'\ \ e_1 \sim k'\text{АКТ1}'e_1\perp$

$k\,'\text{АКТ1}'\ e_1(e_2)\,e_3 \sim e_1(k\,'\text{АКТ}'e_2\perp)\,k\,'\text{АКТ1}'\ e_3\perp$

$k\,'\text{АКТ1}'\ e_1 \sim e_1$

Теперь мы можем окончательно сформировать предложение § 'АВЫР1'.1:

$k\,'\text{АВЫР1}'\ t_i\ \text{if}\ (e_c)\ \text{theh}\ e_1\ \text{else}\ e_2 \sim$
$\quad k\,'\text{АКТ}'\ k\,'\text{НОВЧ}'\,1\perp k\,'\text{НОВЧ}'\,2\perp$
$\quad 'K'\,('\text{УП}'\ t_i(e_c)\ '\text{НА}'\,k'\text{МЕТКА}'\ 1\perp)$
$\quad 'K'\ ('\text{АВЫР1}'\ t_i\,e_2)$
$\quad \text{ИДТИ},k\,'\text{МЕТКА}'\,2\perp;$
$k'\text{МЕТКА}'\ 1\perp\!:\ 'K'\ ('\text{ПРАВ}'\ t_i e_1)$
$k'\text{МЕТКА}'2\perp\!:\perp$

Обратимся к описанию функции 'УП'. Согласно синтаксису логических выражений они могут быть условными или простыми.

$k\,'\text{УП}'\ t_i\ (\text{if}\ (e_i)\ \text{then}\ e_1\ \text{else}\ e_2)\ '\text{НА}'\ e_m \sim$
$\quad k\,'\text{АКТ}'\ k'\text{НОВЧ}'\,1\perp k\,'\text{НОВЧ}'\,2\perp$
$\quad 'K'\ ('\text{УП}'\ t_i(e_c)\ '\text{НА}'\ k\,'\text{МЕТКА}'\ 1\perp)$
$\quad 'K'\ ('\text{УП}'\ t_i(e_2)\,'\text{НА}'\ e_m)$

ИДТИ, k 'МЕТКА' 2 ⊥;

k 'МЕТКА' 1 ⊥: 'К' ('УППР' $t_i(e_1)$ 'НА' e_m)

k 'МЕТКА' 2⊥:⊥

k 'УП' e_x ~ k 'УППР' e_x ⊥

Здесь 'УППР' – условный переход по простому логическому выражению. Простое логическое выражение имеет структуру, аналогичную простому арифметическому выражению. Ради простоты исключим операции ≡ и ⊃ . Остальные логические операции располагаются в порядке усиления связи следующим образом: дизъюнкция ∨ , конъюнкция ∧ , отрицание ¬ . Для них легко построить необходимые конструкции:

(R) k 'УППР' $t_i(e_1 \vee e_2)$ 'НА' e_m ~

k 'УППР' $t_i(e_1)$ 'НА' e_m ⊥

k 'УППР' $t_i(e_2)$ 'НА' e_m ⊥

(R) k 'УППР' $t_i(e_1 \wedge e_2)$ 'НА' e_m ~

k 'АКТ' k 'НОВЧ' 1 ⊥ k 'НОВЧ' 2⊥

'К' ('УППР' $t_i(e_1)$ 'НА' k 'МЕТКА' 1 ⊥)

ИДТИ, k 'МЕТКА' 2 ⊥;

k 'МЕТКА' 1⊥: 'К' ('УППР' $t_i(e_2)$ 'НА' e_m)

k 'МЕТКА' 2⊥:⊥

k 'УППР' $t_i(\neg e_1)$ 'НА' e_m ~

k 'АКТ' k 'НОВЧ' 1⊥

'К' ('УП1' $t_i(e_1)$ 'НА' k 'МЕТКА' 1 ⊥)

ИДТИ, e_m;

k 'МЕТКА' 1⊥:⊥

k 'УППР' e_x ~ k 'УП1' e_x ⊥

k 'УП1' t_i (true) 'НА' e_m ~ ИДТИ, e_m ;

k 'УП1' t_i (false) 'НА' e_m ~

k 'УП1' $t_i((* e_1))$ 'НА' e_m ~ СЧ, e_1; ИДНОТ, e_m;

k 'УП1' $t_i((* e_1)(e_2))$ 'НА' e_m ~ k 'ПИУФ' $t_i(e_1)e_2$ ⊥

ИДНОТ, e_m;

k 'УП1' $t_i((e_c))$ 'НА' e_m ~ k 'УП' $t_i(e_c)$ 'НА' e_m ⊥

k 'УП1' $t_i(e_1 s (= \geqslant <)_f e_2)$ 'НА' e_m ~

k 'ПРАВ' $t_i e_1 k$ 'МИНУС' $e_2 \perp\perp$

k 'КОП' $s_f \perp e_m$;

k 'УП1' $t_i(e_1 s(> \leqslant)_f e_2)$ 'НА' $e_m \sim$

k 'ПРАВ' $t_i e_2 k$ 'МИНУС' $e_1 \perp\perp$

k 'КОП' $s_f \perp e_m$;

k 'УП1' $t_i (e_1 \neq e_2)$ 'НА' $e_m \sim k$ 'УППР' $t_i(\neg e_1 = e_2)$ 'НА' $e_m \perp$

k 'КОП' $= \sim$ ИДНУЛ,

k 'КОП' $s(\geqslant\leqslant)_f \sim$ ИДНОТ,

k 'КОП' $s(< >)_f \sim$ ИДОТР,

$\varkappa = k$ 'МИНУС'

$\varkappa(* = 0) \sim$

$\varkappa s(+ -)_f e_1 \sim \varkappa^1 s_f e_1$

$\varkappa e_1 \sim - \varkappa^1 e_1$

$\varkappa^1 e_1 + e_2 \sim \varkappa^1 e_1 \perp - \varkappa^1 e_2$

$\varkappa^1 e_1 - e_2 \sim e_1 + \varkappa^1 e_2$

$\varkappa^1 e_1 \sim e_1$

('УП 1' – условный переход по первичному логическому выражению).

Заметим, что использованный нами способ программирования отрицания и конъюнкции – простейший, но отнюдь не наилучший. Оптимизировать программу можно было бы внесением знака отрицания внутрь логического выражения (используя известные свойства операций $\vee$ и $\wedge$) до уровня первичных логических выражений.

Синтаксис АЛГОЛа гласит:

<первичное логическое выражение> : : = <логическое значение> | <переменная> | <указатель функции> | < отношение> | (<логическое выражение >)

<логическое значение> : : = **true** | **false**

<отношение> : : = <прав> <знотн > <прав >

<знотн> : : = < | $\leqslant$ | = | $\geqslant$ | > | $\neq$

(знотн – знак операции отношения)

Как и в случае арифметических выражений, разделим случай логической переменной на случаи простой переменной и переменной с индексами. Значение логической переменной **true** будем хранить в

виде числа +1, а значение false – в виде числа –1. С учетом этого обстоятельства должна быть определена и функция 'ПИУФ'.

Вернемся еще раз к вопросу об эффективности синтаксического анализа. Использованный нами способ расчленения арифметических и логических выражений на подвыражения в соответствии со старшинством операций является простым и достаточно эффективным для практического применения, хотя, вероятно, не оптимальным. Легко видеть, что даже при заведении специальной функции для каждого уровня старшинства он требует нескольких (по числу уровней) просмотров выражения. Существуют другие алгоритмы синтаксического анализа, которые устраняют повторные просмотры, правда, ценой увеличения числа операций, совершаемых в процессе просмотра, в то время как наш алгоритм требует весьма простых просмотров. Ясно, что эти алгоритмы также можно описать на рефале. Какой из возможных алгоритмов окажется в конечном счете эффективнее и насколько, об этом можно с уверенностью судить, только проведя специальное исследование.

Этим закончено описание функции 'АВЫР'. Напомним, что конкретизация

$$k\,'\text{АВЫР}'\mathscr{E}\perp$$

где $\mathscr{E}$ – арифметическое выражение, дает программу, заносящую на сумматор значение выражения $\mathscr{E}$. Теперь осталось только проверить, что в левой части оператора присваивания стоит идентификатор, и приписать команду засылки числа в соответствующую ячейку. Присвоим функции, которая решает задачу в целом, детерминатив 'ПОП' – программирование оператора присваивания:

$$k\,'\text{ПОП}'\,e_1 := e_2 \sim k\,'\text{ПОП}1'\,(k\,'\text{ИДЕНТ}'\,(\;)\,e_1\perp)\,e_2\perp$$

$$k\,'\text{ПОП}1'\,((e_1))e_2 \sim k\,'\text{АВЫР}'\,e_2\perp\;\;\text{ЗП},e_1;$$

Мы видели здесь использование рефала в качестве метаязыка для описания алгоритмического языка. В результате конкретизации функции 'АВЫР' получаем программу для машины-исполнителя. Рассмотрим теперь более общую задачу построения трансляторов с помощью рефала.

Пусть $\mathscr{P}$ – программа на некотором алгоритмическом языке, а $\mathscr{E}$ – начальные данные, т.е. объект, к которому применяется алгоритм $\mathscr{P}$. Опишем на рефале рекурсивную функцию с детерминативом 'L' (идентифицирующим данный язык) таким образом, что выполнение конкретизации

$$k\,'\text{L}'\,\mathscr{P}\,(\mathscr{E})\perp \tag{1}$$

будет рассматриваться как применение алгоритма $\mathscr{P}$ к объекту $\mathscr{E}$, и, в частности, результат этой конкретизации (когда он существует)

будет результатом применения алгоритма. Описание рекурсивной функции 'L' есть наиболее прямое и непосредственное выражение семантики языка. Используя программистский термин, можно сказать, что здесь происходит и н т е р п р е т а ц и я программы (текста $\mathscr{P}$). Функцию 'L' мы называем и н т е р п р е т и р у ю щ е й ф у н к ц и е й языка.

Интерпретация происходит на рефал-машине, которая, в свою очередь, моделируется на определенной м а ш и н е - и с п о л н и т е л е типа описанной выше (в дальнейшем машину-исполнитель или компьютер мы будем обозначать M_r). В целях эффективности необходимо скомпилировать программу, осуществляющую алгоритм $\mathscr{P}$ для машины M_r, т.е. перевести $\mathscr{P}$ с рефала на язык машины M_r. Это значит, что мы должны проследить, как происходит конкретизация (1) с заданным $\mathscr{P}$, но п р о и з в о л ь н ы м $\mathscr{E}$, и отобразить объекты и действия рефал-машины на объекты и действия машины M_r.

Решение этой задачи в соответствии с теорией компиляции осуществляется в два этапа: а н а л и з к о н ф и г у р а ц и й и о т о б р а ж е н и е.

Конфигурацией мы называем обобщенное состояние рефал-машины, описываемое находящимся в ее поле зрения рефал-выражением со свободными переменными. Иначе говоря, конфигурация есть множество рабочих выражений в поле зрения, получающееся заменой свободных переменных на их всевозможные значения. Например, множество (1) с заданным $\mathscr{P}$, но произвольным $\mathscr{E}$ есть конфигурация

$k\,'\mathrm{L}'\,\mathscr{P}\ (e_1) \perp$

которая представляет собой начальную конфигурацию для компиляции программы $\mathscr{P}$.

Эквивалентные преобразования рефал-программ, в частности п р о г о н к а (см. [24-25]), позволяют выполнить последовательные шаги рефал-машины над обобщенным полем зрения и проследить, как конфигурации расщепляются и превращаются одна в другую. Результатом этого прослеживания является г р а ф к о н ф и г у р а ц и й.

На этапе отображения рефал-машины на машину M_r различные конфигурации отображаются на различные состояния точки управления в машине M_r, а свободные переменные отображаются на информационные поля M_r. В результате граф конфигураций отображается на программу для M_r. Те операции, которые могут осуществляться машиной M_r, но не описываются на рефале, рассматриваются как "машинные процедуры" и в процессе компиляции переводятся в соответствующие коды машины M_r. Например, описанная машина имела команду сложения, записываемую в виде:

СЛ,А (*)

в результате которой содержимое ячейки А складывается с сумматором и сумма помещается в сумматор. На рефале вводится, соответ-

ственно, функция с детерминативом СЛ и форматом обращения:

$$k\,'\text{СЛ}'\,(e_1)(e_2)\perp \qquad (**)$$

причем характер выражений e_1 и e_2 и результата конкретизации не уточняется, пока речь идет о рефал-машине. Необходимо только отметить, что переход конфигурации (**), где e_1 отображено на сумматор, а e_2 – на ячейку А, в конфигурацию e_3, отображенную снова на сумматор, описывается на языке машины M_r командой (*).

В частном случае, когда машина M_r представляет собой рефал-машину, применение изложенных методов приводит к оптимизации программы на рефале. Если же алгоритм, осуществляющий процесс компиляции, также описан на рефале, то он становится одним из объектов своей работы, в результате чего мы автоматически получаем важнейшие системные программы.

Пусть рекурсивная функция 'СР', описанная на рефале, выполняет компиляцию заданной рефал-программы в программу для некоей машины M_r. Так как объектом работы функции 'СР' являются рефал-предложения, можно воспользоваться каким-либо метакодом, превращающим предложения в выражения. Это будет метакод, близкий к метакоду-А (см. главу X, разд.2).
Вот ключ к нему:

e_1	→* E1	k	→* K( )
A	→ A	( )	→( )
*	→* V	'OMEGA'	→'OMEGA'

Метакод предложения имеет вид:

$$((\mathcal{L})=\mathcal{R})$$

где $\mathcal{L}$ и $\mathcal{R}$ – метакоды левой и правой частей соответственно.

Для того чтобы задание на компиляцию было полностью определенным, необходимо, кроме описания всех участвующих функций, указать еще начальную и конечную конфигурации и их отображение на машину M_r. Это будет достигаться введением двух условных функций: 'ОМА' (т.е. ω^a – начальная конфигурация рефал-машины) и 'OMZ'(ω^z – конечная конфигурация). Для рассмотренного выше случая (компиляция программы $\mathcal{P}$ на языке 'L') начальная конфигурация будет иметь вид:

$$k\,'\text{OMA}'\,e_1 \sim k\,'\text{L}'\mathcal{P}\,(e_1)\perp$$

Для спецификации отображения при переводе этого предложения в метакод сделаем в левой части такую замену:

E1 →(E1 'IN' M1,'M2)

Это означает, что значение переменной e_1 располагается в машине M_r в поле с границами M1 и M2.

Метакод предложения, описывающего выходную конфигурацию, будет иметь вид:

(('OMZ' (*E1 'IN' M1,M2)) = *E1)

Это означает только то, что результат конкретизации (скомпилированная программа) рассматривается как значение одной свободной переменной выражения, которое размещается в машине M_T в том же поле (M1,M2).

Итак, для того чтобы осуществить перевод программы $\mathscr{P}$, надо выполнить конкретизацию:

k'CP'$\mathscr{D}$ ['L'] (('OMA' (*E1 'IN' M1,M2)) =
*K('L'$\mathscr{M}$ [$\mathscr{P}$] (*E1))) (('OMZ'
(*E1 'IN' M1,M2)) = *E1)⊥ (*IC*)

Через $\mathscr{D}$['L'] мы обозначили описание функции 'L' в метакоде, через $\mathscr{M}$[$\mathscr{P}$] – метакод текста $\mathscr{P}$.

Эта конкретизация выполняется с помощью рефал-интерпретатора, т.е. мы получаем компилятор, работающий в режиме интерпретации. Конкретизация требует $\mathscr{P}$ и дает программу для компьютера, которая помещает $\mathscr{E}$ в поле (M1,M2) и в нем же вырабатывает результат.

Следующее выражение описывает процесс компиляции интерпретатора:

k'CP'$\mathscr{D}$ ['L'] (('OMA' (*E1 'IN' M1,M2)
(*E2 'IN' N1,N2)) = *K('L' *E2(*E1)))
(('OMZ' (*E1 'IN' M1,M2)) = *E1)⊥ (CI)

Конкретизация требует только описания 'L' и дает программу для компьютера, которая помещает $\mathscr{P}$ в поле (N1,N2), $\mathscr{E}$ – в поле (M1, M2) и вырабатывает путем интерпретации результат в поле (M1,M2).

Совершая еще один метасистемный переход, т.е. компилируя процесс конкретизации (IC) для произвольного $\mathscr{P}$, получаем:

k'CP'$\mathscr{D}$ ['CP'] (('OMA' (*E1 'IN' N1,N2)) =
*K('CP'$\mathscr{M}$ [$\mathscr{D}$['L']] (('OMA'
(*VE1 'IN' M1,M2)) = *VK('L' *E1(*VE1))) (CC)
(('OMZ' (*VE1 'IN' M1,M2)) = *VE1)))
(('OMZ' (*E1 'IN' N3',N4)) = *E1)⊥

что представляет собой скомпилированный компилятор. Конкретизация требует только описания 'L' и дает программу, которая помещает в поле (N1,N2) и порождает в поле (N3,N4) скомпилированную программу, такую же как в случае (*IC*).

Сделав еще один метасистемный переход, можно написать выражение (CCC в нашей системе обозначений). конкретизация которого дает скомпилированный для машины компилятор компиляторов.

2. Действия над полиномами

Каждому полиному, зависящему от переменных из некоторого конечного множества, будет поставлено в соответствие одно и только одно выражение рефала. Будем называть такое выражение **представлением** соответствующего полинома. (Термин "представление" будет использоваться также для выражений рефала, отображающих объекты, отличные от полиномов.) Выбор представления обусловливается рядом факторов. При ориентации на большой объем действий над полиномами определяющим фактором следует считать эффективность выполнения алгоритмов, основанных на выбранном представлении. По нашему мнению, рассматриваемое ниже представление удовлетворяет этой ориентации.

Представление полиномов

Исходным пунктом построения представлений полиномов является составление списка переменных, от которых могут зависеть эти полиномы. Переменные будем обозначать через x_i, где i – номер переменной в списке. Порядок расположения переменных в списке определяет ход построения представлений и в результате – их окончательный вид. Список остается неизменным в процессе построения представлений. Максимальное значение i обозначим через n. Список может быть, в частности, пустым $(n = 0)$.

Введем объекты, которые будем называть "полиномами с индексом" и обозначать через P_m, где $0 \leq m \leq n$. (Для различения индивидуальных объектов будем вводить еще верхний индекс P_m^1, P_m^2 и т.п.). Каждый такой объект есть совокупность некоторого полинома, зависящего от переменных из списка, и числа m, являющегося верхней гранью индексов этих переменных, но необязательно **точной** верхней гранью. Отсюда следует, в частности, что нулевое значение индекса может быть только у константы. Обратное, вообще говоря, неверно: константе можно приписать любой индекс $i \leq n$.

Обозначив представление произвольного полинома P_m через $\mathscr{R} P_m$, дадим следующее его рекуррентное определение:

1. Представление являющегося константой полинома есть представление этой константы. Ограничения на представление констант накладываются следующие:

а) представление константы не должно оканчиваться на выражение, заключенное в скобки;

б) представлением нулевой константы должно быть пустое выражение.

2. Если полином P_m не является константой, его можно разбить на два слагаемых таким образом, что одно слагаемое не зависит от переменной x_m, а другое можно записать в виде произведения x_m и некоторого полинома. Таким образом,

$$P_m = P^1_{m-1} + x_m P^2_m \qquad (*)$$

Представление полинома P_m определяется тогда следующим рекуррентным соотношением:

$$\mathcal{R}P_m = \mathcal{R}P^1_{m-1}(\mathcal{R}P^2_m) \qquad (**)$$

Пункты определения можно рассматривать как правила построения представлений. Первое правило описывает терминальный шаг построения, второе - нетерминальный. Поскольку согласно (**) представления некоторых полиномов распадаются на два других, процесс построения представления можно изобразить в виде дерева, вершиной которого является исходный полином. Этот процесс закончится, когда все концевые ветви дерева станут терминальными. В результате представление не являющегося константой полинома состоит из представлений полиномиальных коэффициентов, разделенных определенным образом скобками. Будем называть такие скобки полиномиальными. Полиномиальным термом будем называть пару полиномиальных скобок вместе с содержащимся в них представлением промежуточного полинома. Будем говорить, что пара полиномиальных скобок изображает переменную x_m, если эта пара скобок заключает представление такого промежуточного полинома, который согласно (*) умножается на x_m. Будем говорить, что две пары полиномиальных скобок из двух представлений соответствуют друг другу, если они изображают одну и ту же переменную. Термин "соответствие" будет использоваться также и для полиномиальных термов.

Рассмотрим детальнее общую структуру представлений полиномов.

В представлении любого исходного, не являющегося константой полинома крайние правые полиномиальные скобки всегда изображают переменную x_n и отмечают, что слева от них расположено представление полинома, который потенциально может зависеть только от переменных $x_1, x_2, \dots, x_{n-1}$, так как его индекс равен $n-1$, а внутри них расположено представление полинома, потенциально зависящего от переменных $x_1, x_2, \dots, x_n$, так как его индекс равен n. Аналогичное утверждение справедливо для представления любого промежуточного полинома с индексом $m < n$.

Таким образом, каждая следующая соседняя слева пара полиномиальных скобок изображает следующую по уменьшению индекса переменную и содержит представление полинома, потенциально зависящего от числа переменных, меньшего на единицу. Вхождение внутрь полиномиальных скобок означает переход к полиному, степень которого уменьшена по крайней мере на единицу. Очевидно, фиксирование максимального значения индекса n делает представление полиномов однозначным.

От представления полинома легко перейти к его аналитическому виду. Для этого перед каждой парой полиномиальных скобок нужно поставить изображаемую ею переменную в качестве множителя и знак +, после чего раскрыть полиномиальные скобки.

Рассмотрим несколько примеров представлений полиномов.

Зафиксируем множество четырех переменных x_1, x_2, x_3, x_4. Каждый пример демонстрирует переход от полинома к его представлению. Представления констант имеют их аналитический вид. (Напомним, что нулевые константы должны быть представлены пустыми выражениями).

1. $5 \to 5$
2. $x_4 \to (1)$
3. $x_1 \to (1)\,(\,)\,(\,)\,(\,)$
4. $-4x_3^3 \to (((-4)))\,(\,)$
5. $x_3^2 + 2x_3x_4 + x_4^2 \to ((1))\,((2)\ (1))$
6. $x_1^2 + 2x_1x_4 + x_4^2 \to ((1))\,(\,)\,(\ )\ ((2)\ (\,)(\,)\,(1))$
7. $1 + 10x_3 - 2x_4 + 9x_4x_2 + 6x_4x_3 - 8x_4x_3x_2 + 7x_4x_3^2 - 3x_4^2 + 5x_4^2x_3 + 4x_4^3 \to$ $1\,(10)\ (-2\,(9)\,(6\,(-8)\,(7))\,(-3\,(5)\,(4)))$

Приведем пример обратного преобразования, т.е. преобразования представления полинома в его аналитический вид:

$$1\,(((3))\,(\,))\ (-2\,((2))\,((-1)\,(\,))\,(-5)) \to 1 + x_3\,(+x_2\,(+x_2\,(3)) +$$

$$+ x_3\,(0)) + x_4\,(-2 + x_2\,(+x_2\,(2)) + x_3\,(+x_2\,(-1) + x_3(0)) + x_4\,(-5)) \equiv$$

$$\equiv 1 + 3x_3x_2^2 - 2x_4 + 2x_4x_2^2 - x_4x_3x_2 - 5x_4^2$$

Сложение полиномов

Опишем функцию $\ll^{p}$, осуществляющую сложение полиномов и имеющую формат

$$\ll^{p}(\mathscr{R}\,P_i^1)\,(\mathscr{R}\,P_i^2)\mathscr{P}\bot$$

где P_i^1 и P_i^2 – складываемые полиномы, в том числе и промежуточные, $0 \leq i \leq n$, а $\mathcal{P}$ – накапливаемый результат последовательного сложения соответствующих полиномиальных термов. Если рассматривать $\mathcal{P}$ как полином, можно полагать, что его индекс всегда равен n. При внешнем обращении к α^p $i = n$, а $\mathcal{P}$ является пустым выражением. Индекс i последовательно уменьшается на единицу после каждого рекурсивного обращения функции α^p самой к себе согласно следующему предложению:

$$\S\ 1.1 \quad \alpha^p(e_0\ (e_p))(e_q(e_r))\,e_s \sim \alpha^p(e_0)(e_q)(\alpha^p(e_p)(e_r)\perp)\ e_s\perp$$

являющемуся первым предложением описания функции α^p. Таким образом последовательно складываются соответствующие полиномиальные термы.

Если предложением § 1.1 все соответствующие полиномиальные термы просуммированы, то в одном из слагаемых могут остаться полиномиальные термы, которым не нашлось соответствующих. Эти термы должны быть перенесены в результат без изменения. Для этого осуществляется переход к функции α^1 следующими предложениями:

$$\S\ 1.2 \quad \alpha^p(e_0\ (e_p))(e_d)\ e_s \sim \alpha^1(e_0)\ e_d\perp(e_p)\ e_s$$

$$\S\ 1.3 \quad \alpha^p(e_c)(e_q(e_r))\,e_s \sim \alpha^1(e_q)\ e_c\perp(e_r)\ e_s$$

В случае же отсутствия или исчерпания полиномиальных термов в аргументах-слагаемых эти аргументы являются либо представлениями ненулевых констант, либо пустыми выражениями (которые можно рассматривать как представления нулевых констант). В любом случае можно обратиться к функции α^c, осуществляющей сложение констант в нужном представлении и имеющей формат

$$\alpha^c(\mathcal{C}_1)\mathcal{C}_2\perp$$

где $\mathcal{C}_1$ и $\mathcal{C}_2$ – представления складываемых констант.

Описание функции α^c не приводится по той причине, что, во-первых, оно зависит от вида констант и способа их представления и, во-вторых, реализация этой функции, как правило, тесно связана с арифметическими машинными операциями.

Окончание описания функции α^p выглядит следующим образом:

$$\S\ 1.4 \quad \alpha^p(e_c)\ (e_d)\ e_s \sim \alpha^c(e_c)\ e_d\perp\beta\, e_s\perp$$

Чтобы понять, для чего введена вспомогательная функция β, рассмотрим сложение двух полиномов: (-1) и (1). Складывая их, получим после сложения коэффициентов, т.е. приведения подобных, $\beta(\,)\perp$. Очевидно, функция β должна отбросить пустые скобки.

Оставшиеся функции α^1 и β, участвующие в сложении полиномов, описываются следующим образом:

§ 2.1 $\alpha^1(e_p(e_q))\, e_c \sim \alpha^1(e_p)\, e_c \perp (e_q)$

§ 2.2 $\alpha^1\, e_1 \sim \alpha^c\, e_1 \perp$

§ 3.1 $\beta(\,)\, e_v \sim \beta\, e_v \perp$

§ 3.2 $\beta\, e_v \sim e_v$

Перемножение полиномов

Опишем функцию μ^p, осуществляющую перемножение двух полиномов и имеющую формат

$$\mu^p(\mathcal{R}\, P_n^1)\mathcal{R}\, P_n^2 \perp$$

где P_n^1 и P_n^2 - исходные перемножаемые полиномы.

Перемножение полиномов начинается с проверки, не является ли нулем хотя бы один из сомножителей. Ненулевые аргументы передаются для перемножения функции μ^1:

§ 4.1 $\mu^p(\,)\, e_q \sim$

§ 4.2 $\mu^p(e_p) \sim$

§ 4.3 $\mu^p\, e_1 \sim \mu^1 e_1 \perp$

Рассмотрим описание функции μ^1, имеющей формат

$$\mu^1\, (\mathcal{R}\, P_m^1)\mathcal{R}\, P_m^2 \perp$$

где P_m^1 и P_m^2 - ненулевые перемножаемые полиномы. Структура ее первого аргумента может быть одной из четырех следующих:

1-й случай: $(\mathcal{R}\, P_m^{12})$, где P_m^{12} - ненулевой промежуточный полином. Согласно (**) такое представление имеет место тогда, когда первый аргумент имеет вид $x_m P_m^{12}$. Перемножение полиномов в этом случае описывается следующим предложением:

§ 5.1 $\mu^1((e_q))\, e_r \sim (\mu^1(e_q)\, e_r \perp)$

Внешние скобки в правой части изображают переменную x_m, которая переходит в произведение из первого сомножителя.

2-й случай: $\mathcal{R}\, P^{11}_{m-1}()$, где P^{11}_{m-1} – ненулевой промежуточный полином. Пустые крайние правые полиномиальные скобки первого аргумента указывают согласно (**) на его независимость от переменной x_m. Для выполнения перемножения в этом случае вводится функция μ^2, задачей которой является перемножение двух полиномов с различными индексами: индекс первого сомножителя на единицу меньше индекса второго. Обращение к μ^2 осуществляется следующим предложением:

§ 5.2 $\mu^1 (e_p\ (\ ))\ e_r \sim \mu^2\ (e_p)\ e_r \perp$

3-й случай: $\mathcal{R}\, P^{11}_{m-1}\ (\mathcal{R}\, P^{12}_m)$, где P^{11}_{m-1} и P^{12}_m – ненулевые промежуточные полиномы. Согласно (**) такое представление соответствует полиному, имеющему вид:

$$P^{11}_{m-1} + x_m\ P^{12}_m$$

Перемножение в этом случае выполняется по формуле

$$(P^{11}_{m-1} + x_m P^{12}_m)\ P^2_m = P^{11}_{m-1}\ P^2_m + x_m\ P^{12}_m\ P^2_m$$

следующим предложением:

§ 5.3 $\mu^1\ (e_p\ (e_q))\ e_r \sim \alpha^p\ (\mu^2 (e_p)\ e_r \perp)\ ((\mu^1\ (e_q\ )\ e_r \perp))\perp$

Вторая пара скобок второго аргумента функции α^p изображает переменную x_m, которая переходит из второго слагаемого первого аргумента.

4-й случай: аргумент является ненулевой константой. В этом случае происходит обращение к функции μ^3, осуществляющей умножение константы на полином и имеющей формат

$$\mu^3\ (\mathcal{C})\mathcal{R}\ P_k$$

где $\mathcal{C}$ – ненулевая константа, $0 \leqslant k \leqslant n$. Этот переход описывается следующим предложением:

§ 5.4 $\mu^1\ e_1 \sim \mu^3\ e_1 \perp$

Функция μ^2 имеет следующий формат

$$\mu^2\ (\mathcal{R}\, P^1_{m-1})\mathcal{R}\ P^2_m \perp$$

где P^1_{m-1} и P^2_m – ненулевые полиномы. Описывая μ^3, мы должны

помнить, что результатом ее работы должен быть полином с индексом m. При разборе структуры ее аргументов выделяются те же четыре случая, что и для функции μ^1, с той лишь разницей, что классификация проводится по виду второго аргумента.

1-й случай: $(\mathscr{R} P_m^{22})$, что соответствует второму аргументу, имеющему вид $x_m P_m^{22}$. Перемножение полиномов в этом случае описывается следующим предложением:

§ 6.1 $\mu^2(e_p)(e_r) \sim (\mu^2(e_p)\, e_r \perp)$

Внешние скобки в правой части изображают переменную x_m, переходящую в произведение из второго сомножителя.

2-й случай: $\mathscr{R} P_{m-1}^{21}(\ )$, т.е. второй аргумент не зависит от x_m. В этом случае осуществляется переход к умножению полиномов с одинаковыми индексами предложением

§ 6.2 $\mu^2(e_p)\, e_q\, (\ ) \sim \mu^1(e_p)\, e_q \perp (\ )$

Пустые скобки в правой части изображают переменную x_m, определяя тем самым значение индекса результата, равное m.

3-й случай: $\mathscr{R} P_{m-1}^{21}(\mathscr{R} P_m^{22})$, т.е. второй аргумент имеет вид

$$P_{m-1}^{21} + x_m P_m^{22}$$

Перемножение полиномов в этом случае выполняется по формуле

$$P_{m-1}^{1}\,(P_{m-1}^{21} + x_m P_m^{22}) = P_{m-1}^{1}\, P_{m-1}^{21} + x_m P_{m-1}^{1}\, P_m^{22}$$

следующим предложением:

§ 6.3 $\mu^2(e_p)\, e_q (e_r) \sim \mu^1(e_p)\, e_q \perp (\mu^2(e_p)\, e_r \perp)$

Взаимно подобных среди складываемых произведений не будет, так как второе произведение умножается на переменную x_m, которая не входит в первое произведение. Поэтому сложение таких произведений осуществляется непосредственным приписыванием их друг к другу с учетом значений индексов.

4-й случай: аргумент является ненулёвой константой. Этот случай распадается на два подслучая в соответствии с видом первого аргумента:

а) первый аргумент не является константой. Поскольку его индекс равен $m-1$, переход к умножению константы на полином должен

сопровождаться появлением в результате пустых скобок, изображающих переменную x_m для определения значения m его индекса. Этот случай описывается следующим предложением:

§ 6.4 $\mu^2 (e_0 (e_p))\, e_c \sim \mu^3 (e_c)\, e_0 (e_p) \perp (\ )$

б) первый аргумент является ненулевой константой. Поскольку константе можно приписать любой индекс без изменения вида ее представления, то при переходе к перемножению констант пустые скобки вводить не нужно. Окончание описания функции μ^2 имеет следующий вид:

§ 6.5 $\mu^2 e_1 \sim \mu^c\, e_1 \perp$

Относительно функции μ^c можно сказать то же, что было сказано выше относительно функции α^c.

Функция μ^3 осуществляет последовательное умножение константы на полиномиальные термы второго аргумента:

§ 7.1 $\mu^3 (e_c)\,(e_q) \sim (\mu^3 (e_c)\, e_q \perp)$

§ 7.2 $\mu^3 (e_c)\, e_p (\ ) \sim \mu^3 (e_c)\, e_p \perp (\ )$

§ 7.3 $\mu^3 (e_c)\, e_p (e_q) \sim \mu^3 (e_c)\, e_p \perp (\mu^3 (e_c)\, e_q \perp)$

§ 7.4 $\mu^3\, e_1 \sim \mu^c\, e_1 \perp$

Эта функция является последней в описании алгоритма перемножения полиномов.

Дифференцирование полиномов

Рассмотрим дифференцирование полиномов одной переменной. Пусть исходный полином записывается в виде:

$$a_0 + x(a_1 + x(a_2 + \ldots + x(a_l)\ldots))$$

и пусть результат дифференцирования имеет аналогичный вид:

$$a_1 + x(2a_2 + x(3a_3 + \ldots + x(l a_l)\ldots))$$

Тогда процесс дифференцирования можно представить состоящим из двух частей:

а) вынесение на внешний скобочный уровень полинома содержимого внешней пары скобок;

б) умножение коэффициента a_i, $2 \leqslant i \leqslant l$, на i.

Для дифференцирования полиномов многих переменных вводятся функции θ, θ^1 и θ^2. Функция θ выполняет действие (а) с учетом

зависимости полиномов от многих переменных, а функции θ^1 и θ^2 выполняют действие (б).

Формат функции θ следующий:

$$\theta(\mathscr{J})\,(\mathscr{R}\ P_m)\mathscr{P}\perp$$

где $\mathscr{J}$ – представление числа $j=m-d$, где d – индекс переменной дифференцирования; P_m – дифференцируемый полином; $\mathscr{P}$ – результат последовательного дифференцирования его полиномиальных термов. Если рассматривать $\mathscr{P}$ как полином, можно полагать, что его индекс всегда равен n. При внешнем обращении к θ $m=n$, а $\mathscr{P}$ является пустым выражением.

Описание функции θ начинается с предложений, работающих в случае $j=0$, т.е. когда крайние правые полиномиальные скобки полинома P_m изображают переменную дифференцирования. Если эти скобки пустые, дифференцирование заканчивается в соответствии с предложением

§ 8.1 $\theta(z)\,(e_p(\,)\,)\,e_s \sim \beta e_s \perp$

где z – представление нуля. В противном случае эти скобки снимаются и происходит обращение к функции θ^1

§ 8.2 $\theta(z)(e_p(e_q))\,e_s \sim \theta^1\,e_q(e_s)\perp$

В случае $j>0$ функция θ последовательно переносит продифференцированные полиномиальные термы в результат. Если очередной терм нулевой, в результат переносятся пустые полиномиальные скобки и число j уменьшается на единицу функцией γ:

§ 8.3 $\theta(e_x)(e_p(\,))\,e_s \sim \theta(\gamma\,e_x\perp)\,(e_p)\,(\,)\,e_s\perp$

Если очередной терм ненулевой, дифференцирование осуществляется следующим предложением:

§ 8.4 $\theta(e_x)\,(e_p\,(e_q))\,e_s \sim \theta(\gamma e_x\perp)\,(e_p)\ \theta\,(e_x)\,(e_q)\perp)\,e_s\perp$

Если все полиномиальные термы дифференцируемого полинома исчерпаны (или их не было), процесс дифференцирования заканчивается работой предложения

§ 8.5 $\theta(e_x)(e_p)\,e_s \sim \beta e_s\perp$

Функция θ^1 осуществляет вынесение множителя, стоящего при переменной дифференцирования, входящей в первой степени в промежуточный полином, следующим образом:

§ 9.1 $\theta^1\ e_p(\,)(e_s) \sim e_p(\,)\,e_s$

§ 9.2 $\theta^1\ e_p(e_q)(e_s) \sim e_p(\theta^2\,(\nu^2\perp)\,e_q\perp)e_s$

§ 9.3 $\theta^1\ e_c(e_s) \sim e_c\,\beta e_s\perp$

Результатом работы функции ν^2 должна быть константа, равная двум, так как предложением § 9.2 осуществляется переход к множителю при переменной дифференцирования, входящей во второй степени в промежуточный полином. Предложения § 9.1 и § 9.3 описывают случаи, когда переменная дифференцирования входит только в первой степени в промежуточный полином и процесс дифференцирования нужно закончить.

В отличие от θ^1 функция θ^2 осуществляет умножение множителей при переменной дифференцирования на ее степень, отличную от единицы. θ^2 описывается следующим образом:

§ 10.1 $\theta^2(e_k)(e_q) \sim (\theta^2(\nu^{p1} e_k \perp)\, e_q \perp)$

§ 10.2 $\theta^2(e_k)\, e_p(\ ) \sim \mu^3(e_k) e_p \perp (\ )$

§ 10.3 $\theta^2(e_k)\, e_p(e_q) \sim \mu^3(e_k) e_p \perp (\theta^2(\nu^{p1} e_k \perp)\, e_q \perp)$

§ 10.4 $\theta^2 e_1 \sim \mu^c e_1 \perp$

Функция ν^{p1} прибавляет единицу к своему аргументу, что необходимо при переходе функции θ^2 к следующей степени переменной дифференцирования.

На этом описание дифференцирования закончено.

3. Обработка графов

Постановка задачи.

Пусть дан конечный ориентированный граф $\mathcal{G}$ и одна из вершин этого графа n. Требуется построить все несамопересекающиеся пути, которые выходят из вершины n.

Прежде чем писать программу на рефале, необходимо тщательно продумать внутреннее представление (в виде выражений) обрабатываемых объектов. Изящество, простота и эффективность программы во многом зависят от удачного выбора представления.

В этой задаче в первую очередь следует выбрать представление графа $\mathcal{G}$.

Рассмотрим, например, граф, изображенный на рис.3.1. Он имеет четыре вершины: A, B, C и D. От A направлены две дуги к вершинам B и C, от B – к C, от C – к D. Из D не выходит ни одной дуги.

Следовательно, чтобы описать граф, достаточно перечислить все его вершины и при каждой вершине указать те вершины, в которые направлены дуги, выходящие из этой вершины.

Пусть из вершины n направлены дуги в вершины $n_1, n_2, \ldots n_l$.

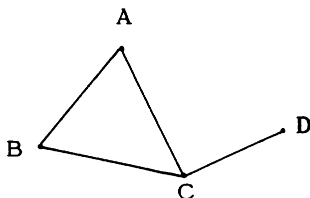


Рис.3.1.

Тогда опишем вершину n следующим образом: $n(n_1 n_2 \dots n_l)$. В частности, возможно $l = 0$.

Описанием графа будем считать выражение, которое получается приписыванием друг к другу описаний всех его вершин. Так, граф, приведенный в качестве примера, опишется следующим образом:

$A(BC)$ $B(C)$ $C(D)$ $D(\,)$

или, например, так:

$C(D)$ $D(\,)$ $B(C)$ $A(BC)$

Теперь выберем представление для результата работы программы. После работы программы должно получиться множество всех путей, выходящих из начальной вершины.

Путь , последовательно проходящий через вершины будем изображать как: $n_1, n_2, \dots, n_p$, $(n_1 n_2 \dots n_p)$

Представление для множества путей определим как конкатенацию представлений всех составляющих его путей.

Так, из точки A есть следующие пути:

(A)
(AB)
(ABC)
$(ABCD)$
(AC)
(ACD)

Результатом работы программы будет выражение:

(A) (AB) (ABC) $(ABCD)$ (AC) (ACD)

а может быть и такое выражение:

(ABC) (AB) (ACD) $(ABCD)$ (A) (AC)

Теперь следует выбрать формат обращения к функции рефала. Выберем следующий:

$\phi\,(s_x)(e_g) \perp$

Здесь s_x – начальная вершина, а e_g – представление графа.

Легко убедиться, что поставленную задачу решает функция ϕ описание которой занимает два предложения:

$$\phi e_p (e_1 s_x e_2)(e_3 s_x t_a e_4) \sim$$
$$\phi e_p s_x t_a (e_3 e_4) \perp$$
$$\phi e_p (e_2)(e_3 s_x t_a e_4) \perp$$
$$\phi e_p t_a t_g \sim (e_p)$$

Настоятельно рекомендуется прокрутить программу вручную для графа, приведенного в качестве примера, и "прочувствовать" работу этой функции.

Для этого надо прокрутить выражение:

$$\phi (A)(A(BC)B(C)C(D)D(\ ))\perp$$

Должно получиться:

$$(ABCD)(ABC)(AB)(ACD)(AC)(A)(\ )$$

Здесь считается, что пустой путь ( ) удовлетворяет условию задачи.

Тщательно рассмотрев работу функции ϕ, легко заметить, что основным недостатком ее работы является многократное размножение графа в процессе работы. Поэтому предлагается другая функция, которая делает то же самое, но более экономно расходует память.

Обращение к этой функции следующее:

$$\psi ((s_x))(e_g)()\perp$$

где s_x - начальная вершина, а e_g - граф. Описание функции занимает три предложения

$$\psi e_z (e_1 (e_2 s_x e_3))(e_4 s_x t_a e_5)(e_p) \sim$$
$$\psi e_z (e_1 e_2 s_x (e_3))(t_a)(e_4 e_5)(e_p s_x) \perp$$
$$\psi e_z (e_1\ (e_2))(e_g)(e_p s_x) \sim$$
$$(e_p s_x)\psi e_z (s_x (e_1 e_2) e_g)(e_p) \perp$$
$$\psi (e_g)(\ ) \sim$$

Как эта функция работает, легко разобраться на основании предыдущего примера.

IУ. РЕАЛИЗАЦИЯ РЕФАЛА. ЯЗЫК СБОРКИ

1. Промежуточный язык

Компилятор с АЛГОЛа-60 переводит исходную программу в команды машины. Это не составляет труда, поскольку команды большинства ЭВМ предназначены для выполнения сложений, умножений, пересылок и тому подобных операций.

Другое дело - реализация рефала. Выполнение рефал-программы разлагается на такие операции, как отождествление, замена, подстановка. В большинстве ЭВМ команды, реализующие эти операции, отсутствуют.

В такой ситуации естественным выходом является интерпретация отсутствующих команд: эти команды выполняются специальной программой - интерпретатором, которая расшифровывает код операции каждой команды, выбирает аргументы и выполняет ее.

Мы опишем язык, который является системой команд рефал-машины. Назовем его я з ы к о м с б о р к и.

Язык сборки, так же как и система команд ЭВМ, состоит из конечного числа операторов. Каждый оператор будем записывать в виде: <имя оператора>, <аргументы, разделенные запятыми>;

При реализации языка сборки используется метод интерпретации. Интерпретатор языка сборки содержит для каждого оператора подпрограмму, которая выполняет его. Мы опишем эти подпрограммы на алголоподобном языке, названном нами языком звеньев. Тем самым мы определим точную семантику операторов языка сборки.

Таким образом, получается схема реализации рефала, изображенная на рисунке 4.1.

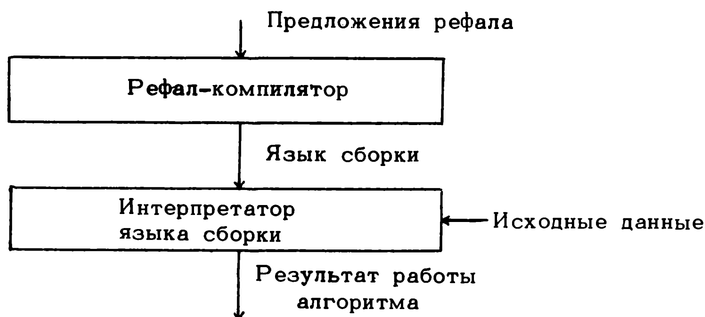


Рис.4.1

Проблема реализации рефала сразу же четко расслаивается на две подзадачи: перевод на язык сборки и реализация языка сборки. Что же это дает?

Во-первых, поскольку и рефал, и язык сборки машинно-независимы, мы можем все алгоритмы перевода на язык сборки и оптимизации описать в машинно-независимом виде.

Во-вторых, поскольку перевод с рефала на язык сборки сводится к некоторому преобразованию символьной информации, его можно описать на рефале. Это делает его хорошо обозримым и доступным для дальнейшего совершенствования и модификации.

В-третьих, максимально упрощая язык сборки, мы облегчаем его реализацию для любой конкретной вычислительной машины, а тем самым, реализацию рефала.

Кроме того, предлагаемая система облегчает дальнейшее развитие рефала, так как реализацию расширений рефала можно свести в основном к добавлению новых операторов языка сборки.

2. Организация памяти

Информация в поле зрения организована в двусвязанный список (см.главу 1, разд.8). В настоящей главе для полей звена будем использовать следующие обозначения: КОД, ПР и СЛ (рис.4.2).

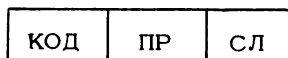


Рис.4.2

Содержимое поля КОД полностью не детализируется в общем описании языка сборки, однако предполагается, что по нему всегда можно различать в поле зрения символ, левую скобку и правую скобку. Причем для левой скобки поле КОД должно содеражать в себе адрес парной к ней правой скобки, а для правой скобки - адрес парной левой скобки.

Отсюда следует, что размер поля КОД должен быть больше, чем размер полей ПР и СЛ. Поэтому поле КОД всегда можно подразделить на два подполя: П и ПАР (рис.4.3).

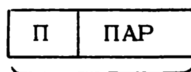


Рис.4.3

Размер подполя ПАР, естественно, совпадает с размером полей ПР и СЛ. Подполе П всегда должно содержать достаточную информацию, чтобы можно было отличить символ от скобки и левую

скобку от правой. Больше ничего о содержимом поля КОД не предполагается. Это находится в полном соответствии с тем, что рефал-машина оперирует с символами как с некоторыми неделимыми объектами, природа которых ей безразлична. Так, например, символ может быть объектным знаком, меткой или числом, при дальнейшем развитии рефала могут появиться символы другой природы, однако это не должно отражаться на интерпретаторе языка сборки, так как он не различает символы равных видов, а тождественность двух символов устанавливает по полному совпадению соответствующих им полей КОД.

Поясним сказанное на примере машины БЭСМ-6.

Звено в машине БЭСМ-6 занимает одну ячейку (48 разрядов). Размер полей ПР и СЛ и подполя ПАР равен размеру адреса - 15 разрядов. Под признак остается 3 разряда (рис. 4.4).

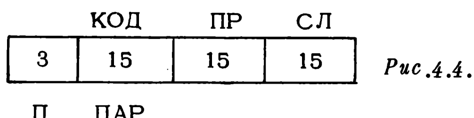


Рис.4.4.

Принята такая кодировка признаков:

001 (
011)
000 объектный знак
010 метка
100 число

В зависимости от значения признака П в подполе ПАР записывается или адрес парной скобки, или дополненный левыми нулями код объектного знака, или адрес, равный значению метки, или значение числа.

3. Таблица элементов

Основная задача синтаксического отождествления - проанализировать выражение, стоящее в поле зрения в области действия ведущего знака конкретизации, и, в случае если отождествление возможно, запомнить те адреса звеньев из поля зрения, которые могут потребоваться в дальнейшем для замены левой части на правую.

Все адреса заносятся в одномерный массив, называемый таблицей элементов (ТЭ).

Элементом предложения рефала называется любой входящий в него символ, объектная скобка или свободная переменная. Элементом поля зрения называется любой символ или структурная скобка, входящие в него.

Чтобы не возникла терминологическая путаница, будем называть элементы массива ТЭ строками таблицы элементов.

Строки ТЭ нумеруются, начиная с 0: ТЭ [0], ТЭ [1], ТЭ[2] и т.д. В каждую строку помещается один адрес звена поля зрения.

Процесс отождествления сводится к установлению соответствия между элементами поля зрения и элементами левой части предложения. При этом одному элементу левой части могут ставиться в соответствие и несколько элементов из поля зрения. Поскольку установление соответствия на практике сводится к занесению некоторых адресов в ТЭ, одному элементу левой части всегда будет соответствовать один или два адреса в ТЭ.

Если элемент левой части символ или структурная скобка - в ТЭ заносится адрес сопоставляемого ему элемента из поля зрения.

Если элемент - переменная терма или выражения, ему будут соответствовать две строки в ТЭ. В первую из них будет занесен адрес начала, а во вторую - адрес конца выражения из поля зрения, сопоставляемого данному элементу левой части.

Соответствие между строками в ТЭ и элементами левой части устанавливает рефал-компилятор в процессе порождения операторов языка сборки.

4. Переменные НЭЛ, Г1 и Г2

Номером элемента левой части предложения называется номер соответствующей ему строки в ТЭ. Если элементу соответствуют две строки, номером элемента называется больший из этих номеров.

В процессе отождествления таблица элементов последовательно заполняется.

Переменная НЭЛ при работе интерпретатора языка сборки всегда установлена на первую незаполненную строку в ТЭ. Каждый оператор отождествления заполняет определенное число строк в ТЭ и увеличивает значение НЭЛ на соответствующую величину так, чтобы она снова указывала на первую свободную строку в ТЭ.

Как правило, один оператор языка сборки осуществляет проектирование одного элемента левой части предложения, но некоторые операторы проектируют сразу по нескольку элементов.

В отличие от переменной НЭЛ, которая в процессе отождествления движется по таблице элементов, переменные Г1 и Г2 в процессе отождествления движутся по полю зрения. В каждый момент отождествления Г1 и Г2 содержат адреса каких-то двух элементов поля зрения.

Их роль понять нетрудно, если учесть, что приступать к проектированию какого-либо элемента левой части можно лишь после того, как у него уже спроектирован хотя бы один соседний элемент. При этом во время отождествления всегда оказывается, что между спро-

ектированными элементами остаются еще непроанализированные "дыры" (рис.4.5), которые постепенно "заполняются" при дальнейшем отождествлении. В самом начале отождествления нам известно только два адреса: знака *k* и точки, поэтому все пространство между ними представляет одну сплошную дыру (рис.4.6).

В конце отождествления, когда анализ завершен и все элементы левой части спроектированы, таких дыр не остается вовсе (рис.4.7).

В середине отождествления может быть несколько таких дыр. Обычно в течение некоторого времени обрабатывается одна дыра, затем какая-то другая и т.д.

Переменные Г1 и Г2 служат для запоминания адресов звеньев. При этом Г1 и Г2 всегда установлены соответственно на левый и правый конец той дыры, из которой в данный момент проектируются элементы (рис.4.8).

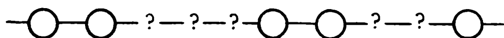


Рис.4.5

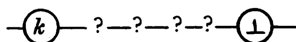


Рис.4.6

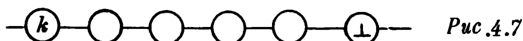


Рис.4.7

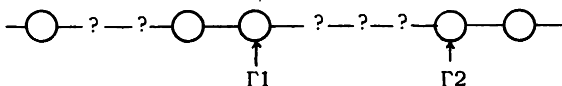


Рис.4.8

Проектирование очередного элемента приводит к тому, что либо Г1 сдвигается вправо, либо Г2 сдвигается влево и тем самым разрыв сужается. При этом надо следить, чтобы Г1 оставалась всегда левее чем Г2.

Перед обращением к функции ведущие *k* и точка превращаются в структурные скобки. В таблице элементов заполняются 3 строки:

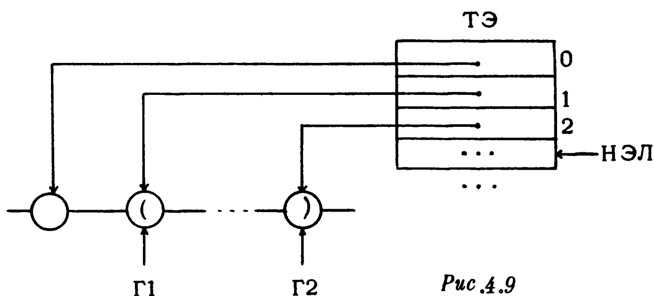


Рис.4.9

ТЭ [0] - адрес звена, предшествующего k; ТЭ [1] - адрес k; ТЭ [2] - адрес точки. Переменной НЭЛ присваивается значение 3. Г1 и Г2 устанавливаются на k и точку соответственно (рис.4.9). Это делает оператор EST, который завершая очередной шаг, подготавливает следующий (см.разд.7).

5. Язык звеньев

Опишем действия, выполняемые каждым оператором языка сборки, на так называемом языке звеньев. Мы предполагаем, что вы знаете язык АЛГОЛ-60, поэтому при описании языка звеньев будем пользоваться терминологией АЛГОЛа-60.

Для большинства операторов даны и словесные описания, поэтому при первом чтении вы можете обращаться к описанию на языке звеньев только тогда, когда у вас возникают вопросы.

Язык звеньев - операторный язык. Каждый оператор языка звеньев записывается на одной строке следующим образом:

<метка> <оператор> ;

Метка отделяется от оператора одним или несколькими пробелами. Она может быть опущена.

Существует 4 типа операторов языка звеньев.

1) Оператор присваивания:

<левая часть> = <правая часть>

2) Оператор перехода:

GO<метка>

и

RETURN

3) Условный оператор:

IF <булевское выражение> <оператор перехода>

4) Оператор процедуры - имеет такой же вид, как в АЛГОЛе-60.

Более точный смысл конструкций языка звеньев объяснен в словаре, который помещен в конце этой главы. В словарь вынесены собственные слова языка звеньев (IF, GO, RETURN), идентификаторы процедур и переменных, выделители полей (ПР, СЛ, КОД, П, ПАР), т.е. все идентификаторы, кроме меток, которые вы встретите в тексте на языке звеньев.

В словаре даны расшифровки идентификаторов, объяснена их семантика и для некоторых указаны операторы языка сборки, в которых они используются.

6. Отождествление объектных выражений

Оператор ЗНАЧ, ϕ осуществляет проектирование символа из левой части на соответствующий символ в поле зрения.

При обращении к оператору ЗНАЧ, ϕ границы Г1 и Г2 должны уже быть установлены на края какой-то дыры (рис.4.10).

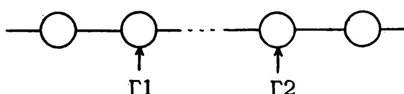


Рис.4.10

Оператор ЗНАЧ, ϕ передвигает Г1 на следующее звено и проверяет, чтобы Г1 при этом не совпала с Г2. Если Г1 совпадет с Г2, это будет означать, что между Г1 и Г2 в момент обращения к оператору ничего не было и, стало быть, символа ϕ тоже нет (рис.4.11).

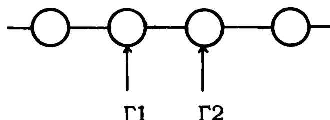


Рис.4.11

Предположим, однако, что Г1 не совпала с Г2. Тогда оператор ЗНАЧ, ϕ проверяет, содержит ли звено, на которое передвинулась Г1, символ ϕ. Если содержит - проектирование возможно. Оператор заносит в ТЭ [НЭЛ] адрес этого звена, а значение НЭЛ увеличивает на единицу (рис.4.12). Если же это звено не содержит символ ϕ, проектирование элемента считается невозможным. Во всех та-

ких случаях любой из операторов передает управление на подпрограмму НЕОТ (неотождествление), которая предпринимает необходимые действия. Подробнее о ней будет сказано в дальнейшем.

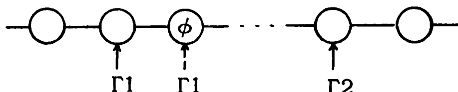


Рис.4.12

Приводим описание оператора ЗНАЧ, ф на языке звеньев. В первую очередь, посмотрите в словаре, который находится в конце этой главы, слова АРГ и СДГ1. В дальнейшем за ответами на все вопросы по языку звеньев обращайтесь к этому словарю.

```
ЗНАЧ  СДГ1;
        IF КОД(Г1) ≠ АРГ GO  НЕОТ;
         ТЭ [НЭЛ] = Г1;
         НЭЛ = НЭЛ + 1;
         RETURN;
```

Оператор ПРОВ; не заполняет ТЭ и не сдвигает НЭЛ, Г1 и Г2. Единственное его назначение - проверить, что Г1 и Г2 установлены в поле зрения на соседние звенья, т.е. что между ними ничего нет. Он необходим, когда нужно закончить обработку какой-либо дыры, чтобы убедиться, что она полностью проанализирована. Если между Г1 и Г2 что-то еще есть, ПРОВ; передает управление на подпрограмму НЕОТ.

```
ПРОВ IF СЛ(Г1) ≠  Г2  GO НЕОТ;
         RETURN;
```

Имея только два оператора: ЗНАЧ, ф и ПРОВ мы уже можем перевести на язык сборки левые части следующего вида:

k« А ~ ...

k « ЖАБА ~...

и т.д.

В самом деле, расставим над второй левой частью номера элементов (в данном случае они совпадут с проекционными номерами). Получим:

```
1  3 4 5 6 2
k« ЖАБА ~
```

(Детерминатив не занимает ни одного звена, поэтому при отождествлении не учитывается).

Отождествление произведет такая последовательность операторов:

ЗНАЧ,Ж; ЗНАЧ,А; ЗНАЧ,Б; ЗНАЧ,А;ПРОВ;

Интересно отметить, что при работе этих операторов НЭЛ, Г1 и Г2 будут все время двигаться нужным образом "сами собой" и будут "автоматически" заполняться нужные строки в ТЭ.

Подобным образом работают почти все остальные операторы отождествления.

Оператор ЗНАЧЯ, ϕ полностью симметричен оператору ЗНАЧ, ϕ (рис.4.13).

```
ЗНАЧЯ          СДГ2;
               IF КОД(Г2) ≠ АРГ GO НЕОТ;
               ТЭ [НЭЛ] = Г2;
               НЭЛ = НЭЛ + 1;
               RETURN ;
```

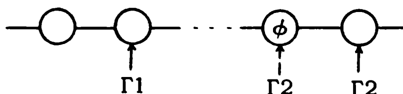


Рис.4.13

Используя ЗНАЧЯ, ϕ можно было бы осуществить другую допустимую интерпретацию последнего примера

1 4653 2
k« ЖАБА ~...

с помощью последовательности операторов:

ЗНАЧЯ,А; ЗНАЧ,Ж; ЗНАЧЯ,Б; ЗНАЧ,А; ПРОВ;

При этом порядок отождествления уже иной и соответствие между элементами левой части и строками таблицы элементов тоже другое.

Оператор СКОБ осуществляет проектирование пары скобок с левого конца дыры (рис.4.14). Он сдвигает Г1 на одно звено вправо,

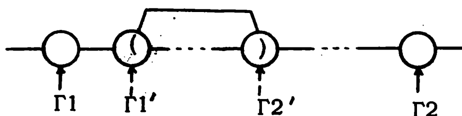


Рис.4.14

проверяет "не занято" ли уже оно Г2, а затем проверяет наличие в этом звене левой скобки. Если левая скобка есть, ее адрес заносится в ТЭ [НЭЛ]. Затем по адресу, содержащемуся в поле ПАР,

находит парную правую скобку и ее адрес заносит в ТЭ [НЭЛ + 1]. После этого 12 устанавливается на правую скобку, а НЭЛ сдвигается на две единицы. Таким образом, СКОБ; проектирует сразу два элемента и заполняет две строки в ТЭ.

```
СКОБ   СДГ1;
         IF ¬ СК(Г1) GO НЕОТ;
         Г2 = ПАР (Г1);
         ТЭ [НЭЛ]= Г1;
         ТЭ [НЭЛ + 1] = Г2;
         НЭЛ = НЭЛ + 2;
         RETURN;
```

Оператор СКОБЯ осуществляет проектирование пары скобок с правого конца дыры (рис.4.15).

```
СКОБЯ   СДГ2;
          IF СК(Г2) GO НЕОТ;
          ТЭ[НЭЛ + 1] = Г2;
          Г2 = ПАР(Г2);
          ТЭ [НЭЛ] = Г2;
          НЭЛ = НЭЛ + 2;
          RETURN;
```

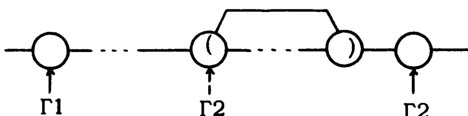


Рис.4.15

Видно, что оператор СКОБЯ существенно асимметричен к оператору СКОБ; это связано с асимметричностью стандартного порядка отождествления, которое производится слева направо и от которого рефал-компилятор отступает только в тех случаях, когда это приводит к повышению эффективности.

Кроме того, адрес правой скобки заносится в ТЭ [НЭЛ + 1], а адрес левой – в ТЭ [НЭЛ], хотя фактически правая скобка проектируется раньше левой. Тут мы видим некоторое отступление от принципа, согласно которому адреса элементов заносятся в ТЭ в том порядке, в котором они проектируются. Однако при замене удобно, чтобы адрес правой скобки всегда шел в ТЭ сразу после адреса парной к ней левой скобки. По той же причине адрес начала любого элемента левой части будет предшествовать в ТЭ адресу конца этого элемента (если, конечно, ему соответствуют две строки в ТЭ).

Оператор УГР, *n,m* не сдвигает НЭЛ и не заполняет таблицу элементов. Он производит установку границ Г1 и Г2, занося в Г1 адрес из ТЭ [*n*], а в Г2 из ТЭ [*m*].

В приведенном ниже описании УГР, *n, m* на языке звеньев вам будут непонятны третий и четвертый операторы. Можете пока считать, что их нет. Их назначение описано в разд.7 (оператор ЗАКР).

```
УГР     Г1 = ТЭ[N];
        Г2 = ТЭ [M] ;
        IF  ТЭ [M]≠ 0 RETURN ;
        Г2 = СЛ(ТЭ [M + 1 ]) ;
        RETURN ;
```

Имея только эти операторы, уже можно отождествить любую левую часть, представляющую собой объектное выражение, причем как стандартным, так и любым из допустимых способов.

```
1      35674  8  2
kф     (X())  Y  ~ . . .
СКОБ; ЗНАЧ,X; СКОБ; ПРОВ; УГР,7,4; ПРОВ; УГР,4,2;
ЗНАЧ,У; ПРОВ;
```

Это стандартный порядок отождествления. Однако в числе допустимых может быть и такой порядок отождествления:

```
1  46785  3  2
kф (X())  Y  ~...
ЗНАЧЯ,У; СКОБ; ЗНАЧ,X; СКОБЯ, ПРОВ; УГР,7,8; ПРОВ;
УГР,5,3; ПРОВ;
```

7. Отождествление свободных переменных

При отождествлении свободных переменных символа, терма и закрытого выражения не возникает каких-либо особых затруднений по сравнению с объектными выражениями.

Так, оператор СИМ осуществляет проектирование главного вхождения переменной символа. Устроен он совершенно аналогично оператору ЗНАЧ, ф только еще проще. Он сдвигает Г1 вправо на одно звено, проверяет Г1 и Г2 на совпадение, и если Г1 и Г2 не совпадают, убеждается, что в том звене, на которое указывает Г1, находится не скобка. Оператор СИМ; не интересуется, какой конкретный символ там стоит. Поэтому он и не имеет аргумента (рис.4.16).

```
СИМ  СДГ1;
     IF  СК(Г1) GO НЕОТ;
     ТЭ [НЭЛ] = Г1;
     НЭЛ = НЭЛ + 1;
     RETURN;
```

Оператор СИМЯ; работает аналогично (рис.4.17).

```
СИМЯ  СДГ2;
      IF  СК(Г2) GO НЕОТ;
      ТЭ [НЭЛ] = Г2;
      НЭЛ = НЭЛ + 1;
      RETURN;
```

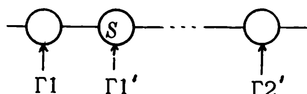


Рис.4.16

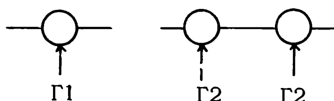


Рис.4.17

Таким образом, левую часть

1 3 4 2

$k \phi s_x \ s_y \sim \quad \ldots$

спроектирует такая последовательность операторов:

СИМ; СИМ; ПРОВ;

Но можно сделать и так:

1 4 3 2

$k \phi s_x \ s_y \sim \quad \ldots$

СИМЯ; СИМ; ПРОВ;

Хотя порядок проектирования при этом не будет соответствовать стандартному.

Для проектирования повторных вхождений переменных символа вместо оператора СИМ используется оператор СТС, n ("старый символ"), а вместо оператора СИМЯ используется оператор СТСЯ, n Аргумент n - это номер главного вхождения переменной символа. Эти операторы делают дополнительную проверку на совпадение полей КОД у главного и повторного вхождений.

```
СТС  СДГ1;
     IF  КОД(Г1) ≠ КОД(ТЭ[N])    GO НЕОТ;
     ТЭ [НЭЛ] = Г1;
     НЭЛ = НЭЛ + 1 ;
     RETURN;

СТСЯ  СДГ2;
        IF КОД(Г2) ≠ КОД(ТЭ [N]) GO НЕОТ;
          ТЭ [НЭЛ] = Г2;
          НЭЛ = НЭЛ + 1;
          RETURN;
```

Левая часть

1 3 4 2

$k\ \phi\ s_x\ s_y \sim$ ...

переведется на язык сборки так:

СИМ; СТС,3; ПРОВ;

Однако ее можно перевести и так (сохраняя те же номера элементов, но не сохраняя вид движения Г1 и Г2):

СИМ; СТСЯ,3 ПРОВ;

Оператор ЗАКР; производит проектирование закрытой переменной выражения.

Для каждой переменной выражения отводится две строки таблицы элементов, где запоминаются адреса начала и конца непустого выражения. Если выражение пустое, вместо адреса начала записывается нуль, а адрес конца - это адрес звена, предшествующего выражению.

Оператор ЗАКР учитывает это различие между пустым и непустым выражением (рис.4.18 и 4.19).

```
 ЗАКР   IF СЛ(Г1) = Г2 GO ЗАКР1;
        ТЭ [НЭЛ] = СЛ(Г1);
        ТЭ [НЭЛ + 1] = ПР(Г2);
        НЭЛ = НЭЛ + 2;
        RETURN;
 ЗАКР1  ТЭ [НЭЛ] = 0;
        ТЭ [НЭЛ + 1] = Г1;
        НЭЛ = НЭЛ + 2;
        RETURN;
```

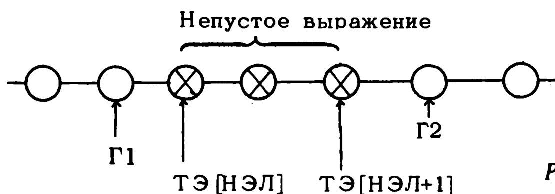


Рис.4.18

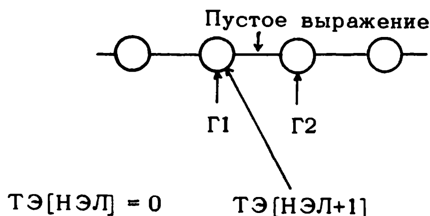


Рис.4.19

Другие операторы тоже должны учитывать различие между пустыми и непустыми выражениями. Теперь вы можете вернуться к описанию оператора УГР, n, m (п.6) и разобраться, зачем нужна проверка на ТЭ [m] = 0.

Теперь можем осуществить проектирование, например, такой левой части:

1 3 5,6 4 2

$k\phi$ s_x e_1 s_x ~

СИМ; СТСЯ,3; ЗАКР;

Здесь над e_1 стоят два номера. Это потому, что e_1 занимает две строки в ТЭ: в ТЭ [5] будет адрес начала e_1, а в ТЭ [6] - адрес конца.

Оператор СТВ, n ("старое выражение") осуществляет проектирование повторного вхождения переменной выражения.

Его аргумент является номером элемента левой части, соответствующего главному вхождению проектируемой переменной, т.е. ситуация аналогична оператору СТС, n. Разница в том, что главному вхождению переменной выражения соответствуют две строки в ТЭ, а в качестве аргумента указывается номер второй из них. И вообще, во всех операторах языка сборки в явном виде всегда указывается номер последней строки, соответствующей данному элементу.

При обращении к оператору СТВ, n происходит следующее. Оператор в ТЭ[n - 1] и в ТЭ [n] находит адреса начала и конца главного вхождения. Затем в ТЭ [НЭЛ] заносится адрес звена, следующего за Г1. Это делается для того, чтобы потом не возвращаться назад, если проектирование удастся. Затем Г1 начинает двигаться по полю зрения слева направо. При этом каждое очередное звено сравнивается с соответствующим звеном главного вхождения. Если произойдет несовпадение содержимого хотя бы двух звеньев, проектирование считается невозможным. Проектирование считается законченным, если все звенья главного вхождения исчерпаны. При этом Г1 оказывается установленным как раз на конец выражения, и его содержимое заносится в ТЭ [НЭЛ + 1], после чего НЭЛ сдвигается на две единицы (рис.4.20).

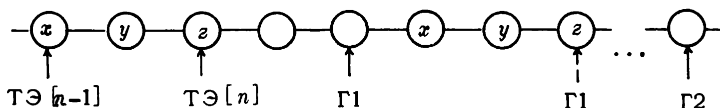


Рис.4.20

```
СТВ   IF  ТЭ[N- 1] = 0  GO СТВ2;
          ТЭ [НЭЛ] = СЛ(Г1);
          АСТ = ПР(ТЭ [N- 1] );
СТВ1  IF  АСТ = ТЭ [N] GO  СТВ3;
          АСТ = СЛ(АСТ);
          СДГ1;
      IF  КОД(Г1) = КОД(АСТ) GO СТВ1;
      IF ¬СК(Г1) GO НЕОТ;
      IF  П(Г1) = П(АСТ) GO СТВ1;
      GO НЕОТ;
СТВ2 ТЭ [НЭЛ] = 0;
СТВ3 ТЭ [НЭЛ + 1] = Г1;
          НЭЛ = НЭЛ + 2;
          RETURN;
```

Оператор СТВЯ, *n* работает аналогично, но только отщепляет выражение не справа от Г1; а слева от Г2.

```
СТВЯ ТЭ [НЭЛ + 1] = ПР(Г2);
     IF   ТЭ[N- 1] = 0 GO СТВЯ2;
          АСТ = СЛ(ТЭ [N] );
СТВЯ1 IF  АСТ = ТЭ[N- 1]GO СТВЯ3;
     АСТ = ПР(АСТ);
     СДГ2;
     IF КОД(Г2) = КОД(АСТ)GO СТВЯ1;
     IF СК(Г2) GO НЕОТ;
     IF П(Г2) = П(АСТ) GO СТВЯ1;
     GOНЕОТ;
СТВЯ2 ТЭ [НЭЛ] = 0;
     НЭЛ = НЭЛ + 2;
     RETURN;
СТВЯ3 ТЭ [НЭЛ] = Г2;
     НЭЛ = НЭЛ + 2;
     RETURN;
```

Теперь можно перевести на язык сборки такие левые части:

```
1  3   5,6  4   7,8  2
k ϕ(  e_a   )  e_a   ~
```

СКОБ; ЗАКР; УГР,4,2; СТВ,6; ПРОВ;

```
1  3  5,6   4  9,10  7,8  2
kϕ  (  (e_a   )  e_1    e_a   ~
```

СКОБ; ЗАКР; УГР,4,2; СТВЯ,6; ЗАКР;

Оператор ТЕРМ проектирует главное вхождение терма. Терм иногда может рассматриваться как частный случай выражения, поэтому в ТЭ ему отводятся две строки.

Работает он следующим образом. Г1 сдвигается на соседнее звено вправо. В ТЭ [НЭЛ] заносится содержание Г1. Затем, если Г1 указывает на левую скобку, Г1 переставляется на парную к ней правую скобку, если же он указывает на символ, то остается неподвижным. Затем в ТЭ [НЭЛ + 1] заносится содержимое Г1, а НЭЛ сдвигается на две единицы.

```
ТЕРМ    СДГ1;
        ТЭ [НЭЛ] = Г1;
        IF ¬ СК(Г1) GO ТЕРМ1;
        Г1 = ПАР(Г1)
ТЕРМ1   ТЭ [НЭЛ + 1] = Г1;
        НЭЛ = НЭЛ + 2;
        RETURN;
```

Оператор ТЕРМЯ работает аналогично, только вместо Г1 фигурирует Г2. В ТЭ [НЭЛ]заносится адрес начала, а в ТЭ [НЭЛ + 1]- адрес конца терма.

```
ТЕРМЯ   СДГ2;
        ТЭ [НЭЛ + 1] = Г2;
        IF ¬ СК(Г2) GO ТЕРМЯ1;
        Г2 = ПАР(Г2)
ТЕРМЯ1  ТЭ [НЭЛ] = Г2;
        НЭЛ = НЭЛ + 2;
        RETURN;
```

Теперь можно спроектировать такую левую часть:

1	3,4	7,8	5,6	2
$k\phi$	t_x	e_1	t_y	$\sim$

ТЕРМ; ТЕРМЯ; ЗАКР;

А как же спроектировать повторное вхождение терма? Поскольку информация, заносимая для него в ТЭ, ничем не отличается от информации, заносимой для выражения , новых операторов не потребуется: можно использовать операторы СТВ, n и СТВЯ,n

1	3,4	7,8	5,6	2
$k\phi$	t_x	e_1	t_x	$\sim$

ТЕРМ; СТВЯ,4; ЗАКР;

Пользуясь введенными операторами, мы уже можем проектировать левые части любой сложности, лишь бы они не содержали открытых переменных выражения.

Например:

1	3	5	7,8	6	11,12	9	13,14	10	4	15,16	17,18	19,20	24,25	22,23	21	2
$k\phi$	(	(	e_1	)	e_2	(	e_3	)	)	e_1	t_x	t_x	l_4	t_x	s_y	$\sim$

СКОБ; СКОБ; ЗАКР; УГР,6,4; СКОБЯ; ЗАКР; УГР,9,10; ЗАКР; УГР,4,2; СТВ,8; ТЕРМ; СТВ,18; СИМЯ; СТВЯ,18; ЗАКР;

Отождествление открытых переменных выражения представляет собой несколько более сложную задачу, так как тут уже не удастся обойтись простым последовательным выполнением операторов языка сборки.

8. Отождествление открытых переменных выражения

Предположим, что нужно произвести проектирование, скажем, такой левой части:

1 3,4 5 6,7 2
$k \phi\ e_1\ s_x\ e_2$ ~

Здесь переменная e_1 - открытая. Если допустить, что проектирование открытой переменной производит оператор УД, получим следующую последовательность операторов:

УД; СИМ; ЗАКР;

Как ее должен выполнить интерпретатор языка сборки? Очевидно, что когда при последовательном выполнении операторов интерпретатор в первый раз войдет в оператор УД, он должен придать e_1 пустое значение. Это сводится к тому, чтобы в ТЭ [НЭЛ] занести ноль, а в ТЭ [НЭЛ + 1] занести адрес из Г1. После этого выполняются следующие операторы как обычно. Если оператор СИМ; сможет произвести проектирование (ЗАКР это всегда может), отождествление благополучно закончится, e_1 будет пустое, а s_x и e_2 примут надлежащие значения. Но что будет, если СИМ не сможет произвести проектирование? Тогда он передаст управление на подпрограмму НЕОТ, которая должна вернуть интерпретатор к выполнению оператора УД, причем предварительно восстановим те значения НЭЛ, Г1 и Г2, которые были в момент первого входа в оператор УД.

При вторичном (и всех последующих) входе в УД он должен работать уже не так, как в первый раз. Теперь УД должен произвести удлинение переменной e_1, добавить к ней справа один терм и установить Г1 на правый конец e_1. Фактически удлинение сводится к изменению адреса конца e_1, хранящегося в ТЭ [НЭЛ + 1]. После удлинения e_1 НЭЛ сдвигается на две единицы и последующие операторы выполняются как обычно.

Как интерпретатор практически может осуществить возврат на удлинение? Очевидно, что для этого ему нужно при первом выполнении оператора УД запомнить в некоторой ячейке текущие значе-

ния Г1, Г2, НЭЛ и адрес самого оператора УД (назовем его адресом возврата АВ) (рис.4.21).

Затем, в случае невозможности проектирования, подпрограмма НЕОТ извлекает информацию, заносит соответствующие значения на Г1, Г2, НЭЛ и передает управление на оператор с адресом АВ. Но в левой части может быть много открытых переменных и удлиняется всегда самая последняя из них, поэтому ясно, что на самом деле потребуется не одна ячейка, а стек, который мы назовем стеком переходов (сокращенно СП) (рис.4.22).

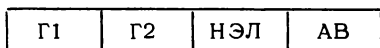


Рис.4.21

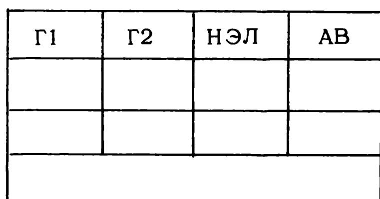


Рис.4.22

Каждая строка СП содержит информацию для восстановления Г1, Г2, НЭЛ и адрес перехода. Переменная ГП всегда указывает на первую свободную ячейку в СП.

Кроме того, чтобы интерпретатору не пришлось различать первое и последующие вхождения в оператор УД, целесообразно этот оператор разбить на два оператора: ПУД (подготовка удлинения) и УД (собственно удлинение).

Оператор ПУД заносит в первую свободную строку стека переходов текущие значения Г1, Г2, НЭЛ и адрес следующего оператора. Затем он увеличивает значение ГП на единицу, тем самым опять устанавливая ее на первую свободную строку СП. После чего ПУД придает пустое значение выражению, заполняя ТЭ[НЭЛ] и[ТЭ НЭЛ+1] и передает управление на оператор, следующий за УД (рис.4.23).

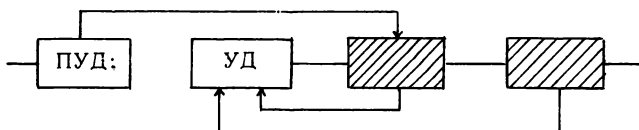


Рис.4.23

Функции подпрограммы НЕОТ сводятся к тому, что она уменьшает ГП на единицу, извлекает информацию из СП [ГП], заносит ее на Г1, Г2, НЭЛ и передает управление по адресу возврата. При этом мы попадем, конечно уже на оператор УД, так как именно его адрес находится в стеке переходов.

Оператор УД удлиняет переменную выражения, если это возможно, и передает управление на следующий оператор. Если же удлинение невозможно, надо удлинять предыдущее открытое выражение, и УД передает управление на НЕОТ.

Получается схема переходов, изображенная на рис. 4.24.

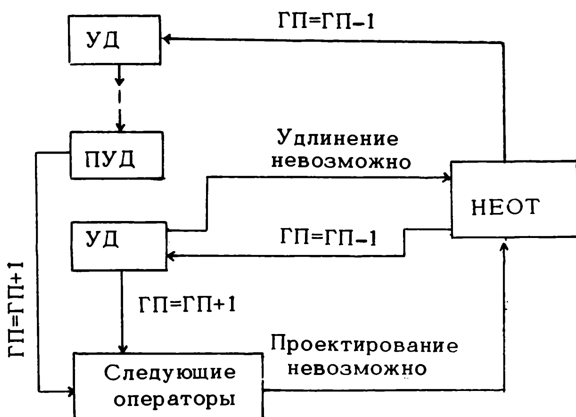


Рис.4.24

Поскольку операторы ПУД; УД всегда встречаются только вместе и в таком порядке, мы в дальнейшем будем писать просто УД. Рефал-компилятору незачем знать, что это на самом деле обозначение двух операторов.

Перед чтением описания операторов ПУД и УД на языке звеньев, посмотрите в словаре, что такое СЧА, ЗПСП, СЧСП.

```
ПУД  ЗПСП(Г1, Г2, НЭЛ, СЧА);
     ГП = ГП + 1;
     ТЭ [НЭЛ] = 0;
     ТЭ [НЭЛ + 1] = Г1;
     НЭЛ = НЭЛ + 2;
     СЧА = СЧА + 1;
     RETURN;
```

```
УД    IF      ТЭ [НЭЛ] ≠ 0 GO УД1;
      ТЭ [НЭЛ] = СЛ(Г1);
УД1   Г1 = ТЭ [НЭЛ + 1] ;
      СДГ1;
      IF ¬СК(Г1) GO УД2;
      Г1 = ПАР(Г1);
УД2   ГП = ГП + 1;
      ТЭ [НЭЛ + 1] = Г1;
      НЭЛ = НЭЛ + 2;
      RETURN;
```

Подпрограмма НЕОТ оформлена как оператор языка сборки. И хотя оператор НЕОТ не нужен при переводе предложений рефала на язык сборки, вы можете использовать его при написании машинных процедур.

```
НЕОТ  ГП = ГП - 1;
      СЧСП(Г1, Г2, НЭЛ, СЧА);
      RETURN;
```

9. Передача управления с одного предложения на другое

Если какой-либо из операторов отождествления не может произвести проектирование элемента, мы должны либо вернуться на удлинение, либо перейти на следующее предложение.

Для управления переходами с одного предложения на другое служит оператор УПЕР, ϕ ("установка перехода"). В простейшем случае он применяется как показано на рис.4.25. То есть в начале каждого предложения, переведенного на язык сборки, ставится оператор УПЕР, ϕ его аргументом является метка, которой помечено следующее предложение процедуры. Перед последним предложением процедуры УПЕР, ϕ не ставится. Это означает, что в случае неотождествления в последнем предложении отождествление вообще невозможно и управление будет передано на авостную подпрограмму IMP.

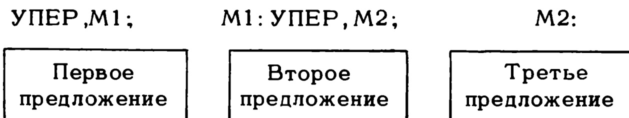


Рис.4.25

Поскольку НЕОТ всегда только лишь распаковывает последнюю ячейку стека переходов, ясно, что УПЕР, ϕ должен заносить какую-то информацию в стек переходов. Причем такую же, какую заносит в СП оператор ПУД! А именно: текущие значения Г1, Г2, НЭЛ и адрес перехода. После чего ГП необходимо увеличить на единицу и передвинуть на свободную строку СП. В качестве адреса перехода заносится, естественно, адрес начала следующего предложения, а не адрес оператора, следующего за УПЕР, ϕ , и в этом – единственное отличие от ПУД, приводящее, впрочем, к качественно другому результату (рис.4.26).

Г1	Г2	3	IMP
Г1	Г2	3	АПЕР
Г1	Г2	НЭЛ	АВ
Г1	Г2	НЭЛ	АВ

←ГП

Рис.4.26

Выше мы всегда предполагали, что перед началом отождествления Г1 содержит адрес *k* , Г2 содержит адрес точки, а НЭЛ = 3.

Теперь же видно, что УПЕР, ϕ и НЕОТ работают таким образом, что достаточно придать Г1, Г2 и НЭЛ надлежащие значения перед входом в самое первое предложение, а дальше эти значения будут восстанавливаться "автоматически" перед входом в очередное предложение и тут же снова запоминаться очередным УПЕР, ϕ .

Самая верхняя строка СП имеет вид, изображенный на рис.4.27.

Это обеспечивает переход на подпрограмму IMP в случае, когда для всех предложений отождествление невозможно.

Кроме того, из описания работы УПЕР, ϕ ясно, что переходы между предложениями можно организовать и следующим образом (см.рис.4.28).

Г1	Г2	3	IMP

Рис.4.27

```
УПЕР,М2;
УПЕР,М1;            М1:                  М2:
```

Первое предложение	Второе предложение	Третье предложение

Рис.4.28

Хотя этот способ ничем не лучше предыдущего, он показывает, что возможности УПЕР, ϕ гораздо шире, чем предполагалось сначала. И эти возможности можно существенно использовать для целей оптимизации.

```
УПЕР   ЗПСП (Г1, Г2, НЭЛ, АРГ);,
       ГП = ГП + 1;
       RETURN;
```

Поясним сказанное конкретным примером. Рассмотрим функцию:

$$\begin{matrix} 1 & & 3,4 & 7,8 & 6 & 5 & 2 \\ k & \phi & t_x & e_1 & A & s_z & \sim \\ 1 & & 3,4 & 7,8 & 6 & 5 & 2 \\ k & \phi & t_a & e_2 & B & s_x & \sim \end{matrix}$$

Переведем левые части на язык сборки. Получим:

ТЕРМ; СИМЯ; ЗНАЧЯ,А; ЗАКР; ...

ТЕРМ; СИМЯ; ЗНАЧЯ,В; ЗАКР; ...

Отсюда видно, что первые операторы у обоих предложений совпадают. Если отождествление в первом предложении окажется невозможным, во втором предложении мы будем делать лишнюю работу, в точности повторяя отождествление первых двух элементов и занося в ТЭ второй раз те же самые адреса.

Между тем, можно первые два оператора применить только один раз, а если в дальнейших операторах отождествление окажется невозможным, переходить не на самое начало второго предложения, а прямо в то место, с которого начинаются расхождения.

```
ϕ  ТЕРМ;
   СИМЯ;
   УПЕР, М1;
   ЗНАЧЯ,А;            М1: ЗНАЧЯ;В;
   ЗАКР;                   ЗАКР;
```

В нашем примере описание функции содержит только два предложения, поэтому, если первые два оператора не смогут произвести проектирование, отождествление будет признано невозможным. Допустим, что ТЕРМ и СИМЯ произвели проектирование. Дальше идет УПЕР,М1. Он заполняет текущие значения Г1, Г2, НЭЛ и адрес перехода в СП. Если в дальнейшем из оператора ЗНАЧЯ,А произойдет переход на НЕОТ, значения Г1, Г2, НЭЛ, бывшие перед входом в ЗНАЧЯ,А, восстановятся произойдет переход на М1 и второе предложение будет работать сразу с оператора ЗНАЧЯ,В.

Одним словом, УПЕР, ϕ позволяет осуществить обход по дереву. В случае невозможности проектирования происходит возврат назад к ближайшей вершине и переход на следующую ветвь (рис.4.29).

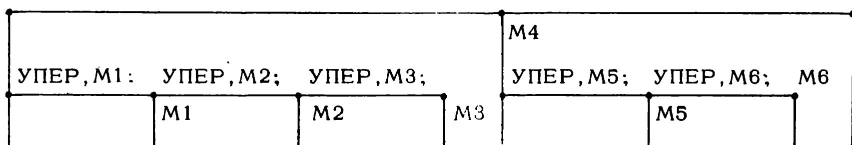


Рис.4.29

Если в тексте на языке сборки стоит оператор УД, это значит, что возможен возврат на этот оператор. Поэтому объединить части параграфов, стоящие после УД, уже нельзя. Это значит, что оператор УПЕР, ϕ не может стоять после оператора УД.

10. Оператор КУД, n;

Рассмотрим такой пример:

```
1      3,4  5  6,7  8  9,10  2
k  ф   e1  +  e2   *   e3   ~   ...
```

Переведем левую часть на язык сборки:
УД; ЗНАЧ,+; УД; ЗНАЧ,*; ЗАКР;
Пусть ведущая область конкретизации имеет вид:

$$k\ \phi\ \underbrace{+++++\ldots++++}_{100\ \text{раз}}\ \perp$$

Как будет происходить отождествление? Сначала e_1 примет значение "пусто", отождествится первый +. Затем e_2 будет удлиняться 99 раз в поисках *. Когда дальнейшее удлинение e_2 станет невозможным, удлинится e_1, после чего e_2 будет удлиняться 98 раз и т.д. В конце концов удлинение станет невозможным.

Давайте оценим объем работы, которую проделала рефал-машина, чтобы убедиться, что отождествление невозможно. Очевидно, что e_2 удлинялась $99 + 98 + \ldots + 2 + 1 = \frac{99 \cdot 100}{2} = 4950$ раз.

Кроме того, e_1 удлинялась 100 раз. Итого 5050 удлинений!

Между тем, совершенно очевидно, что если выражение не содержит *, то и любая его часть тем более не содержит ее. Поэтому после того,

как переменная e_2 "просмотрела" область конкретизации и "не нашла" *, сразу можно сделать вывод о невозможности отождествления. Таким образом, можно было бы обойтись 99 удлинениями, т.е. сократить объем работы в $\frac{5050}{99} = 50,101010 \ldots$ раз

Усложним пример. Пусть функция ϕ содержит два предложения.

$$k \phi e_1 + e_2 * e_3 \sim \ldots$$
$$k \phi e_1 + e_2 \mathrm{A}\, e_3 \sim \ldots$$

К каждому предложению применимы предыдущие рассуждения. Однако возможности оптимизации этим еще не исчерпаны. Легко видеть, что если первое предложение неприменимо, рефал-машина начнет проектировать левую часть второго предложения. При этом, если первое предложение "не нашло" знак +, то второе предложение заведомо неприменимо. Если же первое предложение "нашло" +, то во втором предложении отождествление можно начинать сразу с проектирования переменной e_2 .

Давайте рассмотрим, как выразить на языке сборки подобную оптимизацию.

Если среди операторов отождествления нет оператора УД, то в случае непроектирования в любом из операторов управление будет передано на следующее предложение. Однако, как только встретился оператор УД, из всех последующих операторов управление будет передаваться не на следующий параграф, а на него (рис.4.30), и только в случае невозможности

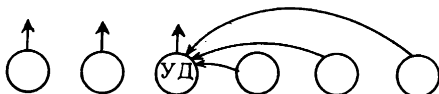


Рис.4.30

удлинения УД передаст управление на следующий параграф. Совокупность тех операторов, которые передают управление на некоторый оператор УД естественно назвать областью удлинения для этого оператора. Здесь, правда, возникает еще одно затруднение: ведь после УД может встретиться еще один УД, и тогда схема переходов будет иной (см.рис.4.31).

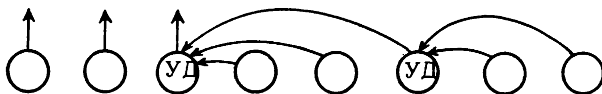


Рис.4.31

Последние два оператора не передают управление на первый УД непосредственно, а делают это только через "свой", второй УД. Поэтому мы будем рассматривать оператор УД вместе с его областью удлинения как один составной оператор цикла. Во всех отношениях он ведет себя как

обычный оператор языка сборки: мы входим в него, затем внутри него происходит какая-то работа и либо проектирование удается, и тогда должен выполняться следующий оператор, либо происходит обращение к НЕОТ. Вот почему нельзя было слить операторы, идущие после УД (да и сами УД): при этом нами разрушался оператор цикла, функционирующий как нечто целостное. Но если совпадут два оператора цикла целиком – их уже можно будет объединить!

Трудность состоит в том, что по теперешнему описанию оказывается, что область удлинения для любого оператора УД охватывает все операторы отождествления, идущие после него. При этом, объединяя два оператора цикла, мы тем самым вообще объединяем две какие-то левые части (что может случиться только, если на языке сборки они тождественно совпадают).

Из сказанного очевидно, что мы сможем сливать операторы цикла в том случае, если будем как-то ограничивать области удлинения, с тем чтобы они не простирались до самого конца отождествления. Кроме того, само ограничение области удлинения почти всегда само по себе является значительной оптимизацией, особенно если область удлинения содержала операторы цикла.

Итак, пусть даны операторы (см.рис. 4.32).

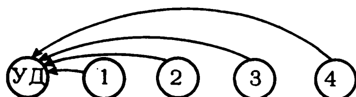


Рис.4.32

Допустим, мы хотим, чтобы в области удлинения входили операторы 1 и 2, а операторы 3 и 4 – уже не входили.

Сделать это можно, применив оператор КУД,1 (рис.4.33).

Рис.4.33

Работает он очень просто: уменьшает значение ГП на единицу. При этом получается, что если теперь 3 или 4 обратится к НЕОТ, то НЕОТ сделает переход не по последней строке СП, а по предпоследней. Значит, переход произойдет куда угодно, но только не на изображенный оператор УД.

Получаем стройную схему для оператора цикла:

УД; <область удлинения> КУД,1;

Видно, что если трактовать УД как левую "скобку", а КУД,1 как правую "скобку", то левую часть предложения, переведенную на язык сборки, можно трактовать как "выражение" в смысле рефала, а оператор – как "терм". Причем оператор цикла – это "выражение" в скобках, а элементарный оператор – "символ".

Однако стройность картины будет портить то обстоятельство, что все КУД,1, стоящие в самом конце "левой части", можно опустить" (что до сих пор предполагалось). Кроме того, несколько операторов КУД,1, идущих подряд будут объединяться в один оператор, так что КУД, n эквивалентно

$$\underbrace{\text{КУД,1; . . . . . КУД,1;}}_{n \text{ раз}}$$

```
КУД ГП = ГП- N ;
    RETURN ;
```

Теперь мы можем перевести предложения функции ϕ с учетом оптимизации:

$$\overset{1}{k}\,\phi\;\overset{3,4}{e_1}\;\overset{5}{+}\;\overset{6,7}{e_2}\;\overset{8}{*}\;\overset{9,10}{e_3}\;\overset{2}{\sim}\ldots$$

$$\overset{1}{k}\,\phi\;\overset{3,4}{e_1}\;\overset{5}{+}\;\overset{6,7}{e_2}\;\overset{8}{\mathrm{A}}\;\overset{9,10}{e_3}\;\overset{2}{\sim}\ldots$$

```
ϕ:
        УД;
        ЗНАЧ,+;
        КУД,1;
        УПЕР,М1;
        УД;                      М1:  УД;
        ЗНАЧ, *;                      ЗНАЧ,А;
        ЗАКР;                         ЗАКР;
```

11. Операторы замены

Когда синтаксическое отождествление закончено, начинается замена левой части предложения на правую.

Программа на языке сборки не создает правую часть заново, как это делается в рефал-интерпретаторе, а подвергает выражение из поля зрения, отождествленное с левой частью, ряду последовательных преобразо-

ваний. Каждому элементарному преобразованию соответствует определенный оператор преобразования. В их число входят, например, такие, как трансплантация (т.е. перестановка куска выражения из одного места в другое), вшивание нового символа или скобки в поле зрения, размножение значения свободной переменной и т.п.

Все эти преобразования производятся над полем зрения, однако мы можем наглядно представлять себе дело так, будто в процессе замены преобразуется не поле зрения, а левая часть предложения рефала, постепенно переходя в правую часть.

Это оправдано в том смысле, что такое преобразование левой части действительно изображает соответствующие преобразования над полем зрения, причем в общем виде.

При этом мы будем говорить о перестановке, размножении, выбрасывании свободных переменных, хотя интерпретатор языка сборки все это будет проделывать над их значениями.

12. Необходимость ограничения «свободы выбора»

Приступая к замене, мы имеем сравнительно небольшое число элементарных преобразований. Все они сводятся к перестановке, уничтожению или изготовлению элементов. Причем применяются они достаточно свободно. Можно, например, действовать следующим образом, вставить в левую часть символ А, а затем удалить его, снова вставить в левую часть символ А, а затем удалить его и так далее. Ясно, что подобный процесс никогда не закончится. И хотя пример, может быть, сильно утрирован, самое основное он передает правильно: если представить слишком большую свободу выбора, может случиться так, что мы даже перестанем различать, какое преобразование приближает нас к цели, а какое удаляет, или вообще "зациклимся". Чтобы этого избежать, в дальнейшем наложим на применяемые преобразования некоторые ограничения, которые, оставляя большую свободу выбора, тем не менее будут гарантировать, что каждое примененное преобразование все же заведомо приблизило нас к конечной цели: правой части.

13. Роль переменной Г в процессе замены

В дальнейшем будем называть правую часть предложения целью (Ц), а преобразуемое выражение – объектом работы (О) или просто объектом.

Перед началом замены объект полностью совпадает с левой частью, а в конце замены должен полностью совпасть с правой частью. В процессе замены объект проходит ряд промежуточных состояний, постепенно преобразуясь.

Наложим на процесс преобразования следующее ограничение: потребуем чтобы в любой момент замены в объекте можно было указать положение некоторой границы, обладающей следующими свойствами:

Та часть объекта, которая стоит слева от границы, должна полностью совпадать с началом цели. Та часть объекта, которая находится справа от границы, еще не совпадает с целью и требует дополнительной работы.

Каждое элементарное преобразование должно сдвигать границу хотя бы на один элемент вправо. Это гарантирует нам, что если цель содержит N элементов, преобразование левой части в правую потребует не больше чем N элементарных преобразований.

Переменная Г (граница) как раз и играет роль такой разграничительной черты. Она всегда установлена на правый конец того элемента объекта, которым заканчивается часть объекта, совпадающая с целью (рис.4.34).

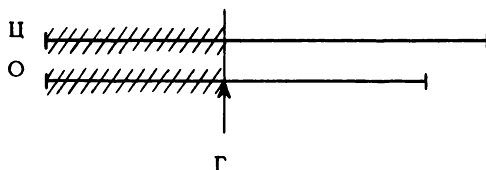


Рис.4.34

На рисунке 4.34 заштрихованы совпадающие части О и Ц.

Проиллюстрируем высказанные положения простым примером. Пусть нам дан объект

$$a\ b\ c\ d\ e\ f$$

и цель

$$f\ b\ c\ a$$

В этом примере, чтобы избежать ненужной детализации, мы обозначили латинскими буквами произвольные элементы объекта и соответствующие элементы цели. Процесс преобразования объекта в цель изображен на рис.4.35.

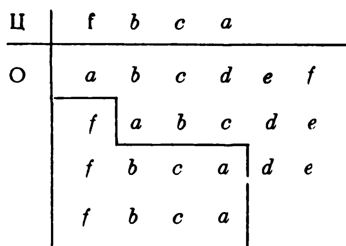


Рис.4.35

Из этой схемы видно, что в самом начале преобразования граница находится перед самым левым элементом О, поскольку уже первые элементы О и Ц (a и f) не совпадают. Первое преобразование заключается в том, что мы трансплантируем элемент f в самое начало объекта. После этого первые элементы О и Ц начинают совпадать и граница сдвигается вправо, устанавливаясь между f и a. Затем трансплантируем участок bc, вставляя его перед границей. После этого совпадут с целью уже элементы $fbca$ и границу можно сдвигать вправо на три элемента. Теперь начало объекта совпадает со всей целью, поэтому следующим преобразованием мы отбрасываем у О остаток de и получаем полное совпадение объекта с целью.

В этом примере мы обошлись только перестановкой и уничтожением элементов, поскольку цель меньше объекта, но в других случаях может потребоваться порождение новых элементов.

14. Порождение объектных выражений

Рассмотрим некоторые конкретные операторы замены. Оператор NS, ϕ производит порождение нового символа ϕ. Он выбирает одно звено из списка свободной памяти, формирует из него символ ϕ и вшивает его в поле зрения сразу же за звеном, на которое установлена переменная Г. Затем оператор передвигает Г на новый символ (рис.4.36).

```
NS    КОД(АНС) = АРГ;
NS1   ПРК = СЛ(Г);
      СШИВ(Г,АНС);
      Г = АНС;
      ВСВ;
      СШИВ(Г,ПРК);
      RETURN;
```

(Метка NS1 поставлена потому, что на нее передают управление некоторые другие операторы замены).

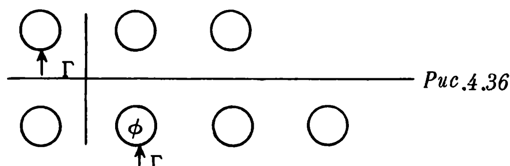


Рис.4.36

Оператор BL вставляет в поле зрения левую скобку сразу же после Г, после чего передвигает Г на эту скобку. Следует отметить, что у этой левой скобки еще нет парной правой скобки, поэтому поле КОД в соответствующем ей звене еще нельзя заполнить окончательно (рис.4.37.).

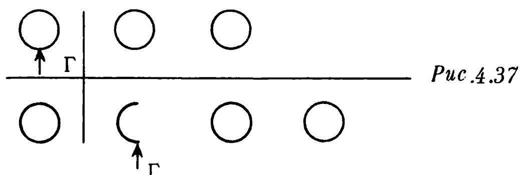


Рис.4.37

Оператор BR вшивает в поле зрения правую скобку сразу же после Г, после чего передвигает Г на эту скобку (рис.4.38).

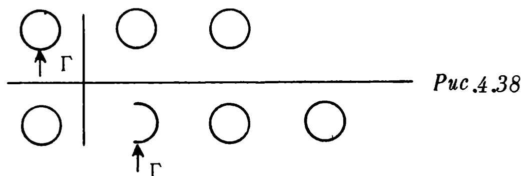


Рис.4.38

Следует отметить, что в момент вшивания правой скобки парная к ней левая скобка уже стоит в поле зрения. Поэтому оператор BR должен связать между собой левую и правую скобки.

Для связывания парных скобок используется следующий алгоритм. В ячейке АСК всегда хранится адрес самой правой неуравновешенной левой скобки. Все неуравновешенные левые скобки связаны в односторонний список (рис.4,39).

Рис.4.39

Когда в поле зрения вшивается новая левая скобка, в поле КОД соответствующего ей звена заносится адрес из АСК, а в АСК заносится адрес новой левой скобки (рис.4.40).

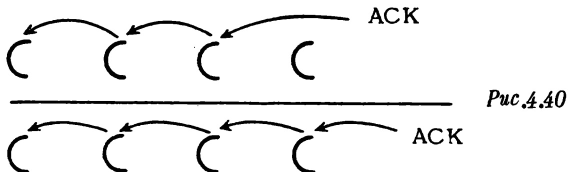


Рис.4.40

Тем самым новая левая скобка присоединяется к списку неуравновешенных левых скобок.

Когда в поле зрения вшивается новая правая скобка, получается, что в ячейке АСК находится адрес парной к ней левой скобки. Этот адрес заносится в поле ПАР правой скобки, после чего в АСК заносим адрес предпоследней неуравновешенной скобки, а в поле ПАР последней неуравновешенной скобки адрес парной к ней правой скобки (рис.4.41).

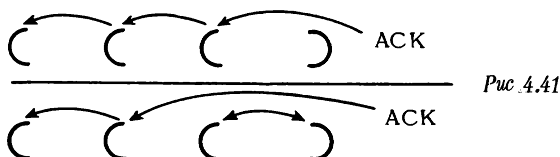


Рис.4.41

Тем самым последняя левая скобка исключается из списка левых неуравновешенных скобок.

```
BL  ПАР(АНС) = АСК;
    АСК = АНС;
    GO NS1;
 BR СЛСК = ПАР(АСК);
    СВЯЗ(АСК,АНС);
    АСК = СЛСК;
    GO NS1;
```

Напоминаем, что метка NS1 описана в операторе NS, Y;

С помощью операторов NS, ϕ; BL и BR можно породить любое объектное выражение. Например:

A (B( )) C

порождает последовательность операторов

NS, A; BL; NS, B; BL; BR:, BR; NS, C;

Простейший рефал-компилятор для порождения объектных выражений будет пользоваться только этими операторами. Однако для повышения эффективности полезно ввести еще следующие операторы:

Оператор SUB, ϕ действует следующим образом. Из звена, следующего непосредственно за Г в поле зрения, формируется символ ϕ. После чего Г передвигается на этот символ (рис.4.42).

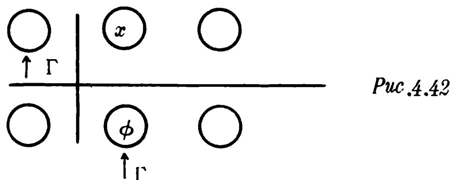


Рис.4.42

Этот оператор работает значительно быстрее, чем NS, ϕ , так как он только меняет содержимое поля КОД у звена, уже стоящего в поле зрения, в то время как NS, ϕ вынужден формировать все три поля у нового символа: и КОД, и ПР, и СЛ.

```
SUB Г = СЛ(Г);
    КОД(Г) = АРГ;
    RETURN;
```

15. Размножение свободных переменных

Оператор MULS, n изготовляет копию значения переменной символа и вставляет ее сразу же после Г. Затем Г устанавливается на новый символ. Аргумент n – это нормер той строки в ТЭ, которой соответствует размножаемое вхождение свободной переменной символа (рис.4.43).

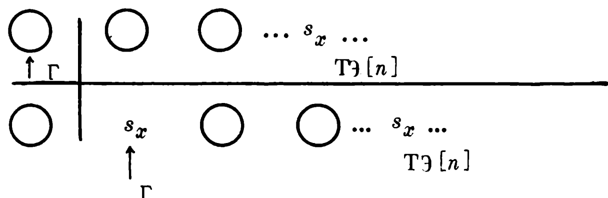


Рис.4.43

Надо сказать, что безразлично, какое именно вхождение переменной размножать, поскольку все они имеют одинаковое значение (однако рефал-компилятор, для определенности, размножает главные вхождения).

```
MULS   КОД(АНС)= КОД(ТЭ [N]);
       GO NS1;
```

Оператор MULE, n работает совершенно аналогично оператору MULS, n. Он размножает значение переменной выражения и вставляет его после Г. Затем устанавливает Г на правый конец вшитого значения.

Для размножения выражения нужно знать адрес его конца и адрес его начала. Но адрес конца хранится в ТЭ [n] , а адрес начала – в ТЭ [$n-1$], поэтому второго аргумента не требуется.

```
MULE   IF ТЭ [N-1] = 0       RETURN;
       АСТ = ТЭ [N-1] ;
       ПРК = СЛ(Г);
       СШИВ(Г,АНС);
       GO MULE2;
```

```
MULE1  АСТ = СЛ(АСТ);
MULE2  Г = АНС;
       ВСВ;
       IF  СК(АСТ) GO MULE4;
       КОД(Г) = КОД(АСТ);
MULE3  IF  АСТ ≠ ТЭ [N]  GO MULE 1:
       СШИВ(Г,ПРК);
       RETURN;

MULE4  IF  СКП(АСТ)  GO MULE5;
       ПАР(Г) = АСК;
       АСК = Г;
       GO MULE1;
MULE5  СЛСК = ПАР(АСК);
       СВЯЗ(АСК,Г);
       АСК = СЛСК;
       GO MULE3 ;
```

Для размножения терма можно было бы ввести оператор MULW, *n*, но этого не требуется, поскольку в ТЭ информация заносится точно та же, что и для выражения. Поэтому для размножения термов употребляется оператор MULE, *n*; .

16. Перестановка участков объекта

Оператор TPL, *n*, *m* производит перестановку ("трансплантацию") участка объекта. Аргументы *n* и *m* задают начало и конец переставляемого участка ("трансплантанта"): *n* –номер элемента, предшествующего первому элементу трансплантанта, *m* –номер последнего элемента трансплантанта.

Трансплантант изымается из объекта, оставшаяся после него "дыра" зашивается, после чего трансплантант вставляется в объект после Г. Г устанавливается на правый конец трансплантанта, т.е. в Г заносится адрес из ТЭ [*m*] (рис. 4.44).

```
TPL  IF ТЭ [N] = ТЭ [M] RETURN ;
     ПРК = СЛ(ТЭ [N] );
     СШИВ(ТЭ [N], СЛ(ТЭ [M] ));
     СШИВ(ТЭ [M], СЛ(Г));
     СШИВ(Г,ПРК);
     Г = ТЭ [M] ;
     RETURN ;
```

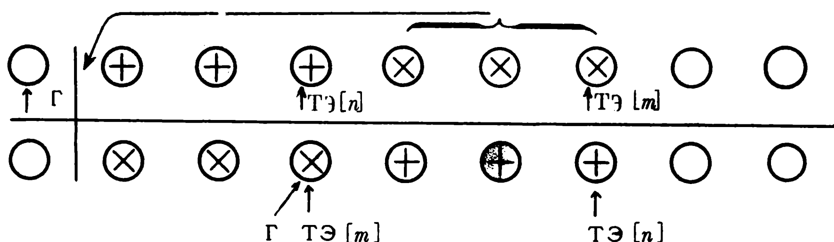


Рис.4.44

Если трансплантант состоит из одного элемента, его тоже можно было бы переставить с помощью оператора TPL, *n*, *m*, однако для этой цели полезно ввести еще два оператора TPLS, *n*; и TPLE, *n* . Эти операторы имеют только один аргумент. Простейший рефал-компилятор будет пользоваться только этими операторами (для трансплантации свободных переменных).

Оператор TPLS, *n* служит для транплантации только таких элементов, которые состоят из одного звена и которым соответствует одна строка в ТЭ. То есть трансплантант может быть символом, скобкой или свободной переменной символа (рис. 4.45).

```
TPLS СШИВ(ПР(ТЭ [N]),СЛ(ТЭ [N]));
     СШИВ(ТЭ [N],СЛ(Г));
     СШИВ(Г,ТЭ [N] );
      Г = ТЭ [N] ;
      RETURN;
```

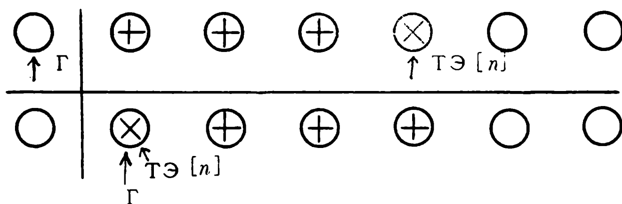


Рис.4.45

Оператор TPLE, *n* служит для трансплантации таких элементов, которые состоят из одного или нескольких звеньев и которым соответствуют две строки в ТЭ. То есть трансплантант может быть свободной переменной терма или выражения (рис.4.46).

```
TPLE  IF ТЭ[N-1] = 0 RETURN ;
      СШИВ(ПР(ТЭ [N-1] ),СЛ(ТЭ [N]));
      СШИВ(ТЭ [N] , СЛ(Г));
      СШИВ(Г,ТЭ[N- 1] );
      Г = ТЭ [N] ;
      RETURN ;
```

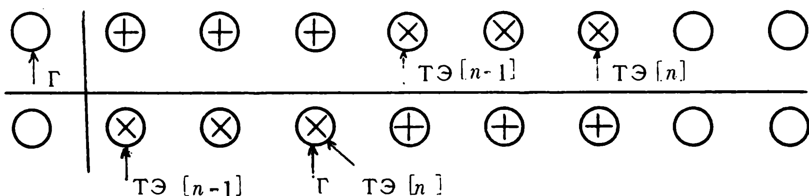


Рис.4.46

17. Оператор OUT, n;

Оператор OUT, *n* уничтожает часть объекта, которая начинается после Г и кончается элементом с номером *n* . При этом Г остается на месте (рис.4.47).

```
OUT   IF  Г = ТЭ [N]   RETURN ;
      ПРК = СЛ(ТЭ [N]);
      СШИВ(ТЭ [N] ,АНС);
      АНС = СЛ(Г);
      СШИВ(Г,ПРК);
      RETURN ;
```

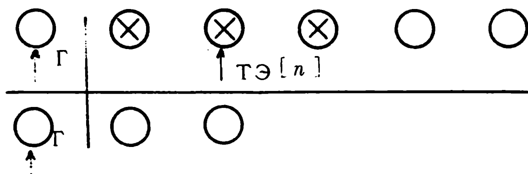


Рис.4.47

Употреблять этот оператор во время замены можно многократно. Однако рефал-компилятор пользуется им очень осторожно. Во-первых, уничтожаемый участок не должен содержать те свободные переменные, которые еще могут понадобиться в дальнейшем для формирования правой части. Во-вторых, выбрасываемый участок может содержать подходящие

куски для формирования правой части. А если их выбросить, дальше придется изготовлять их заново посредством NS, ϕ BLи BR. Поэтому рефал-компилятор предпочитает OUT, n в середине замены вообще не использовать, а ненужные куски "выталкивать" трансплантациями в правый конец объекта, откуда они удаляются в самом конце замены одним оператором OUT, n . В примере из 2.4.2, кстати, так и делалось.

18. Оператор УГ, n;

Оператор УГ, n производит установку границы Г на элементе с номером n . То есть адрес, хранящийся в ТЭ, заносится в Г.

```
УГ   Г = ТЭ [N];
     RETURN ;
```

19. Пример преобразования объекта в цель

Объект: s_z AB ( ) t_x
Цель: t_x AB s_z

Легко убедиться, что требуемое преобразование совершит последовательность операторов:

TPLE, 9 ; TPL, 3,5 ; УГ3 ; OUT, 7 ;

20. Проблема пустых выражений

Предположим, что нам дан объект (e_1)X и цель () e_1. Очевидно, что последовательность преобразований должна быть такой:

$$\overset{3}{(}\ \overset{5,6}{e_1}\ \overset{4}{)}\ \overset{7}{X} \rightarrow \overset{3}{(}\ \overset{4}{)}\ \overset{5,6}{e_1}\ \overset{7}{X} \rightarrow \overset{3}{(}\ \overset{4}{)}\ \overset{5,6}{e_1}$$

Этой последовательности преобразований соответствует последовательность операторов языка сборки;

УГ, 3; TPLS, 4; УГ, 6; OUT, 7;

Сначала устанавливаем Г на левую скобку и трансплантируем правую скобку. Затем Г устанавливается на правый конец e_1 и выбрасывается все, что начинается после ТЭ [6] и кончается ТЭ [7] , т.е. символ X. Это рассуждение верно, если ТЭ [6] содержит адрес последнего элемента выражения e_1.

А что получится, если e_1 пустое?

Тогда, как было отмечено еще в разд.7, в качестве адреса конца выражения будет занесен адрес конца предшествующего ему элемента, а в качестве адреса начала - ноль (рис.4.48).

В нашем конкретном случае пустота e_1 приведет к тому, что в ТЭ [5] будет содержаться ноль, а в ТЭ [6] – адрес левой скобки. То есть ТЭ [5] =0, ТЭ [6] = ТЭ [3] (рис.4.49).

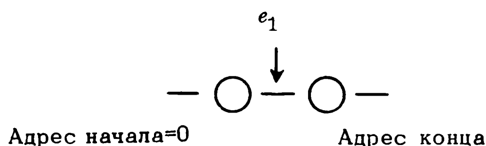


Рис.4.48

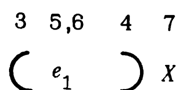


Рис.4.49

После применения оператора TP LS, 4; правая скобка будет идти сразу после левой. Оператор TPLS,4 все сделает правильно; изымет правую скобку и зашьет "дыру" между ТЭ [6] и ТЭ [7], а затем вставит скобку после Г, установленного на ТЭ [3]. Поскольку ТЭ [6] совпадает с ТЭ [3] получится, что оператор TPLS,4 уберет правую скобку из поля зрения, а затем вставит ее обратно на то же самое место. Таким образом, действительно, адресом конца пустого выражения считается адрес конца предшествующего ему элемента. Оператор TPLS,4 переставил правую скобку в нужное место, но адреса в ТЭ [5] и в ТЭ [6] не изменились. Если e_1 непустое, этого и следовало ожидать, но в случае пустого e_1 получается, что оно осталось на прежнем месте (рис.4.50).

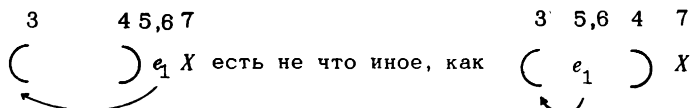


Рис.4.50

Дальше работает УГ,6. Он устанавливает Г на ТЭ [6], т.е. фактически на левую скобку. Затем работает оператор OUT,7, который выбрасывает все, что начинается после Г и кончается на ТЭ [7]. В данном случае получится, что он должен выбросить)X, и он действительно выбросит (рис.4.51).

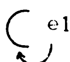


Рис.4.51

Мы пришли к неправильному результату потому, что игнорировали важное качественное отличие, существующее между пустыми и непустыми выражениями. Непустой переменной выражения при отождествлении оказывается сопоставленным какой-то реальный участок поля зрения, у которого есть первое и есть последнее звено. Адрес первого и адрес последнего звена заносим в таблицу элементов во время отождествления, и в дальнейшем при замене эти адреса не меняются, так как при всяких перестановках, вставках и уничтожениях участков поля зрения физическое расположение звеньев в памяти машины не меняется, а вставки, перестановки и вычеркивания сводятся к изменению адресов связи у звеньев, т.е. содержимого полей ПР и СЛ. Поэтому адреса, занесенные в ТЭ при отождествлении, в процессе замены уже не меняются.

Совсем иное дело – пустые выражения! Физически в поле зрения им ничто не соответствует. Поэтому приходится подразумевать, что пустое выражение как бы незримо присутствует между двумя реальными звеньями в поле зрения. Очевидно, что если мы теперь захотим переставить такое выражение в другое место поля зрения, мы это можем сделать, только изменив соответствующие ему адреса в таблице элементов.

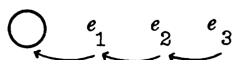
Таким образом, адреса, соответствующие в ТЭ пустым выражениям, в процессе замены придется непрерывно "подправлять". Для этого потребуются некоторые дополнительные средства, о которых будет сказано далее.

21. Решение проблемы пустых выражений

Как уже говорилось, все затруднения при манипуляциях пустыми выражениями проистекают от того, что пустое выражение не имеет в поле зрения каких-то "своих", соответствующих ему звеньев. Адресом его конца считается адрес конца предшествующего ему элемента. Назовем этот предшествующий элемент базой данного выражения (рис.4.52).

Рис.4.52

Если несколько пустых выражений идут подряд, это определение все равно сохраняет силу (рис. 4.53).



фактически представляет собой

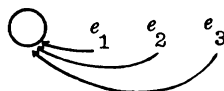


Рис.4.53

В процессе преобразования поля зрения возможны следующие случаи изменения базы: либо само выражение переставляется в другое место и его базой становится другой элемент, либо база данного выражения переставляется в другое место или совсем выбрасывается из поля зрения, либо между базой данного выражения и самим выражением вставляется какой-то новый элемент, либо база представляет собой пустое выражение и адрес ее конца изменился. Все перечисленные случаи сводятся к тому, что выражение отрывается от "своей" базы и у него появляется новая база. Нужно произвести коррекцию его адреса конца в таблице элементов так, чтобы "привязать" его к новой базе. Например, вставляем после базы символ A (рис. 4.54), но выражение осталось "привязанным" к старой базе. Но после коррекции адреса базой должен стать символ A (рис. 4.55).

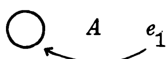


Рис.4.54

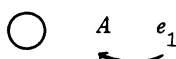


Рис.4.55

Оператор CORA, n, m как раз и служит вышеперечисленным целям. (m – это номер исправляемого выражения; n – это номер новой базы, к которой мы хотим "привязать" выражение m).

Работает оператор следующим образом. Сначала он проверяет содержимое ТЭ [m – 1]. Эта строка в ТЭ должна содержать адрес начала исправляемого выражения. Если ТЭ [m-1]≠0, это означает, что выражение не пустое и для него коррекцию адреса проводить не нужно. Если же ТЭ [m – 1] = 0, это значит, что выражение m пустое и его нужно привязывать к новой базе. Оператор CORA, n, m делает это следующим образом: содержание ТЭ [n] заносится в ТЭ [m]

```
CORA IF ТЭ[M - 1] ≠ 0 RETURN ;
         ТЭ [M] = ТЭ[N] ;
         RETURN ;
```

Кроме того, нужно предусмотреть возможность "привязывать" выражение к тому элементу, на который установлена Г. При этом может случиться, что этот элемент вообще не представлен в ТЭ. Для этого введен вариант оператора с одним аргументом: CORAO, n, где n – номер исправляемого выражения

```
CORAO IF ТЭ [N - 1] ≠ 0  RETURN ;
          ТЭ[N] = Г;
          RETURN ;
```

Теперь мы можем вернуться к рассмотрению примера из разд.20.

$$\begin{array}{ccccccccccccc} 3 & 5,6 & 4 & 7 & & 3 & 4 & 5,6 & 7 & & 3 & 4 & 5,6 \\ (& e_1 &) & X & \to & (&) & e_1 & X & \to & (&) & e_1 \end{array}$$

Последовательность действий такая:

Сначала оператором УГ,3 устанавливаем Г на левую скобку. Затем оператором TPLS,4 переставляем правую скобку.

При этом e_1 отрывается от своей старой базы, левой скобки, и получает новую базу, правую скобку. Значит, если e_1 пустое, необходимо сделать коррекцию адреса. Поэтому после TPLS,4 должен идти оператор CORA,4,6 . Дальше будет работать оператор УГ,6, который как раз использует скорректированное значение адреса и оператор OUT,7; .

Итак, правильная последовательность операторов должна быть такой:

УГ,3; TPLS ,4; CORA,4,6; УГ,6; OUT,7;

Эта последовательность операторов будет работать правильно как в случае пустого e_1 , так и в случае непустого e_1 , поскольку оператор CORA,4,6 просто ничего не делает в случае непустого выражения.

Рассмотрим другой пример.

$$\begin{array}{cccccccccccc} 3 & 4,5 & 6 & & 3 & & 4,5 & 6 & & 3 & & 4,5 \\ X & e_1 & Y & \to & X & A & e_1 & Y & \to & X & A & e_1 \end{array}$$

В этом примере правильная последовательность операторов будет такой:

УГ,3; NS,A ; CORAO,5 УГ,5; OUT,6;

Обратите внимание, что здесь символ A, порожденный оператором NS,A, вообще никак не представлен в ТЭ. Однако Г "автоматически" устанавливается на него, и это использовано в операторе CORAO,5 ;

Такая ситуация носит общий характер. Все вновь порожденные элементы не имеют номеров, и их адреса не заносятся в таблицу элементов. Поэтому алгоритм замены построен так, что все порожденные элементы сразу же оказываются слева от Г и в дальнейшем не подвергаются воздействию операторов.

Вернемся, однако, к коррекции адресов. Что делать, например, при таком преобразовании:

$$\begin{array}{cccccccccc} 3 & 4,5 & 6,7 & 8,9 & & 3 & & 4,5 & 6,7 & 8,9 \\ X & e_1 & e_2 & e_3 & \to & X & A & e_1 & e_2 & e_3 \end{array}$$

Здесь при вшивании символа A меняется база у e_1 , что может вызвать необходимость коррекции e_2 и e_3. Очевидно, что правильный результат дает последовательность операторов:

УГ,3; NS,A; CORAO,5; CORA,5,7; CORA,7,9;

Сначала исправляется адрес у e_1 , затем к нему привязывается e_2 к e_2 привязывается e_3 . Очевидно, что результат будет правильным, если e_1 не пустое, а e_2 и e_3 пустые, хотя при этом e_2 и e_3 будут привязываться к той же самой базе. Однако работа CORA, n,m сводится к простой пересылке адреса из одной строки таблицы элементов в другую, что выполняется чрезвычайно быстро (одна–две команды машины). Поэтому гораздо проще (и быстрее!) пересылать адреса, чем устраивать дополнительное распознавание множества частных случаев.

22. Устранение лишних коррекций

Итак, можно сделать вывод, будто операторы CORA, n, m нужно ставить чуть ли не после каждого "обычного" оператора. Однако это не совсем так.

Например, при преобразовании

$$\overset{3}{X}\ \overset{4,5}{e_1} \rightarrow \overset{3}{X}\ A\ B\ C\ Д\ \overset{4,5}{e_1}$$

формально пришлось бы написать такую последовательность операторов:

УГ, 3 ; NS, A ; CORAO, 5; NS,B; CORAO, 5; NS, C; CORAO, 5 ; NS, Д; CORAO, 5 ;

Но возникает вопрос, нужно ли вообще производить коррекцию адреса для e_1 в этом примере? Ведь адрес, хранящийся в ТЭ [5] , нигде не используется, значит, он нам вообще безразличен. Поэтому все коррекции в данном случае можно убрать.

Видоизменим теперь этот пример так, чтобы адреса e_1 использовались

$$\overset{3}{X}\ \overset{5,6}{e_1}\ \overset{4}{Y} \rightarrow \overset{3}{X}\ A\ B\ C\ Д\ \overset{5,6}{e_1}$$

Теперь последовательность операторов такая:

УГ,3 ; NS, A ; CORAO, 6; NS, B ; CORAO, 6; NSC ; CORAO, 6; NS, Д ; CORAO, 6 ; УГ,6 ; OUT, 4;

Теперь уже нельзя просто так убрать все коррекции. Адрес правого конца e_1 используется в операторе УГ,6. Значит, оператор CORAO,6, стоящий перед УГ,6 нужен. Но спрашивается, какой смысл в действиях всех предыдущих CORAO,6? Зачем исправлять ТЭ [6] первые три раза, когда все равно будет использовано только последнее исправление! Поэтому первые три коррекции адреса можно не делать. Остаются операторы:

УГ,3 ; NS, A ; NS,B; NS ,C; NS, Д; CORAO, 6 ; УГ , 6 ; OUT, 4;

Таким образом, можно выдвинуть следующие два принципа оптимизации:

1) Если после оператора CORA,n,m нигде не используется ТЭ [m], этот оператор можно опустить.

2) Если в тексте на языке сборки стоят два оператора CORA,n,m; .... CORA,n',m, а в операторах, заключенных между ними, ТЭ [m] нигде не используется, первый оператор CORA,n,m можно исключить.

Из этих принципов получается простой алгоритм оптимизации: просматриваем текст на языке сборки от конца до оператора CORA,n,m Пытаемся исключить этот оператор. Потом продолжаем просмотр до следующего оператора CORA,n',m' и т.д.

Практика показывает, что в результате такой чистки в конечной программе остается незначительное количество операторов CORA,n,m которые составляют только 2–3% от общего числа операторов замены.

Необходимость делать коррекцию адресов все равно, конечно, остается досадной помехой для человека, пишущего на языке сборки, так как всегда остается опасность пропустить где-либо необходимый оператор CORA,n,m. Однако язык сборки с самого начала мыслился как машинно-ориентированный, а для рефал-компилятора коррекция адресов не представляет никаких затруднений.

23. Детерминативы

В описываемом варианте языка сборки детерминативы функций не занимают звеньев поля зрения. При отождествлении они поэтому не рассматриваются как обычные составные символы.

Оказалось удобным рассматривать детерминативы как метки и помечать этими метками входы в соответствующие им функции.

Так, например, для функции

```
    k'ПСИ' ~ ....
    k'ПСИ' X ~ ....
```

на языке сборки получаем:

```
ПСИ: УПЕР,Ь1;
     ПРОВ;                   Ь1: ЗНАЧ,X;
                                 ПРОВ;
      :                           :
```

Видно, что тело составного символа ПСИ непосредственно перешло в метку ПСИ, которой помечено первое предложение функции, переведенное на язык сборки.

При этом нам потребовалось как-то пометить и остальные предложения функций. Проще всего использовать для этого какие-либо необычные метки, например начинающиеся с мягкого знака, и одновременно запретить употребление детерминативов такого вида. При этом исключается возможность совпадения "служебных" меток с детерминативами функций.

Однако в программе на рефале могут встречаться и такие составные символы-метки, которые одновременно не являются детерминативами каких-либо функций. В этом случае с каждой из таких меток мы будем связывать некую фиктивную функцию, которую можно формально считать состоящей из нулевого числа предложений.

На языке сборки такая функция изображается с помощью оператора НЕОТ, о котором говорилось в разд.8 этой главы.

Например, пусть ФИ – составной символ-метка, не являющийся детерминативом. Тогда на языке сборки ему будет соответствовать фиктивная функция:

ФИ: НЕОТ;

Таким образом, каждому детерминативу после перевода на язык сборки будет соответствовать метка, а после загрузки языка сборки в память машины – даже некоторый конкретный адрес, прямо указывающий, где находится первый оператор данной функции.

24. Области конкретизаций

Удобно, чтобы перед началом отождествления ведущие k и $\perp$ оказались структурными скобками (рис. 4.56).

Но это не означает, что другие области конкретизации должны быть организованы так же. Некоторые связи являются избыточными. Порвав эти связи, мы можем использовать некоторые поля звеньев для связи областей конкретизации в список, о котором говорилось в разд.8 главы 1.

Имея адрес точки, можно восстановить перед началом шага поле ПАР звена k и поле ПР звена, следующего за точкой. Поэтому эти поля используются для хранения детерминатива и ссылки на следующую по старшинству область конкретизации (рис. 4.57).

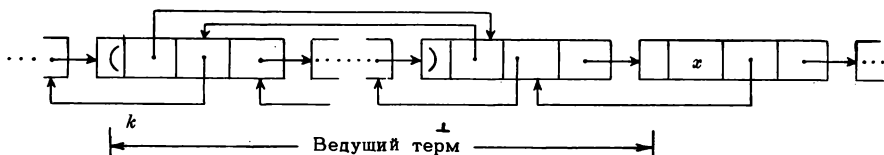


Рис.4.56

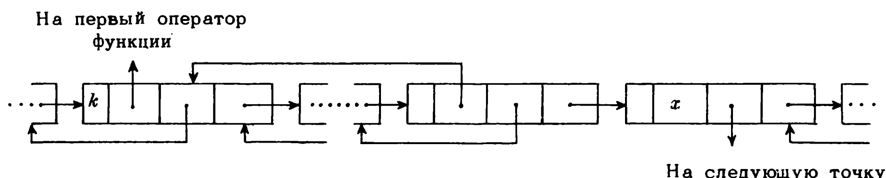


Рис.4.57

Звено со ссылками будем изображать так, как показано на рис. 4.58. Тогда список областей конкретизации можно проиллюстрировать рисунок 4.59. (СТОП – машинная процедура, которая прекращает работу рефал-машины. k' СТОП' и $\perp$ ставятся автоматически при формировании начального поля зрения

k' СТОП' <поля зрения> $\perp$).

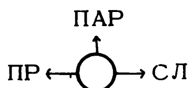


Рис.4.58

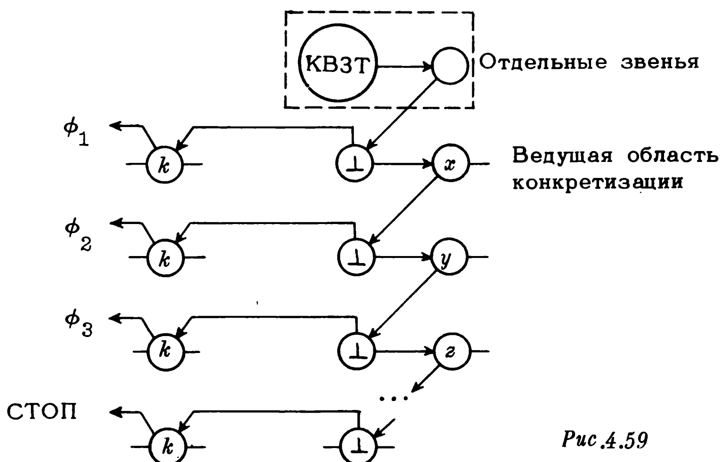


Рис.4.59

Интерпретатор языка сборки занимает два звена: КВЗТ и СЛ(КВЗТ). В поле ПР второго звена всегда хранится адрес ведущей точки.

Может возникнуть вопрос: правомочно ли разрушение поля ПР у "чужого" звена и почему выбрано именно это звено в то время, как у k есть еще два соседних звена и у точки еще одно?

При ответе на этот вопрос нужно учесть порядок выполнения конкретизаций (рис. 4.60).

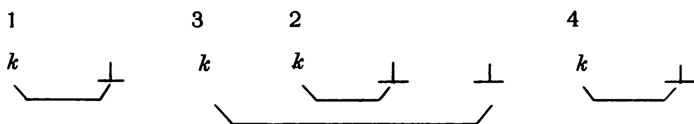


Рис.4.60

Действительно, в период от формирования k и $\perp$ до их выполнения может измениться поле зрения левее k и между k и $\perp$, а правее $\perp$ звенья останутся нетронутыми.

Приведем еще один пример. Пусть поле зрения имеет вид:

$$k\ \phi_3 \ldots\ k\ \phi_1 \ldots \perp k\ \phi_2\ \ldots \perp \perp$$

Тогда связи между звеньями будут организованы так, как изображено на рис. 4.61. (Неотмеченные поля звеньев на рис. 4.61 имеют обычные значения).

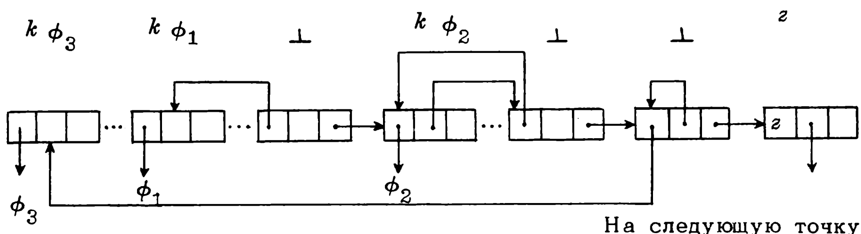


Рис.4.61

25. Активизация скобок

Процесс порождения конкретизационных скобок происходит так: сначала в поле зрения ставятся структурные скобки; а затем они активизируются, т.е. превращаются в конкретизационные скобки.

Для активизации структурных скобок служит оператор АКТО, ϕ, где ϕ – детерминатив.

Предполагается, что в момент применения оператора переменная Г установлена на ту правую скобку, которую мы хотим активизировать.

Действие оператора изображено на рис. 4.62.

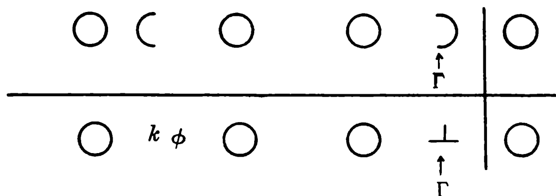


Рис.4.62

Например, выражение

$$k\,'ФИ'\perp$$

можно породить такой последовательностью операторов:

BL; BR; АКТО, ФИ;

При связывании конкретизационных скобок в список используется свойство точек: конкретизации выполняются в той последовательности, в которой расположены точки в поле зрения.

В этом можно убедиться, рассмотрев пример:

$$\begin{matrix} 5 & 1 & 1 & 4 & 2 & 2 & 3 & 3 & 4 & 5 \\ k & k & \perp & k & k & \perp & k & \perp & \perp & \perp \end{matrix}$$

Выражение порождается слева направо; скобки активизируются, как правило, сразу после формирования правой скобки. Поэтому активизация скобок происходит в той последовательности, в какой они будут становиться ведущими.

Это иллюстрирует пример:

$$k\ \phi_2\ \ k\ \phi_1 \perp \perp\ k\ \phi_3 \perp$$

BL; BL; BR; АКТО, ϕ_1 ; **BR;** АКТО, ϕ_2 ;
BL; BR; АКТО, ϕ_3 ;

Опишем теперь, как конкретизационные скобки связываются в список.

Для организации списка областей конкретизации используются две переменные: ПРЕДТ (предыдущая точка) и СЛЕДТ (следующая точка). Как было отмечено, перед началом шага ведущие k и $\perp$ превращаются в структурные скобки (рис. 4.63).

С помощью переменной СЛЕДТ запоминается адрес следующей точки. Переменная ПРЕДТ устанавливается на звено КВЗТ.

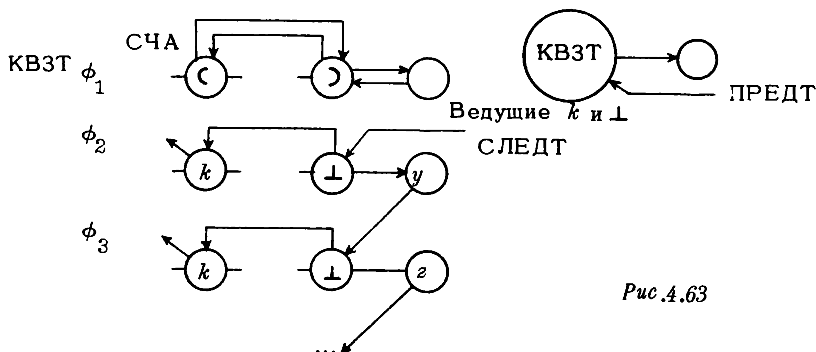


Рис.4.63

Рассмотрим момент, в процессе замены, когда уже порождены несколько конкретизационных скобок (рис. 4.64).

В середине рисунка изображены две скобки, которые предстоит активизировать.

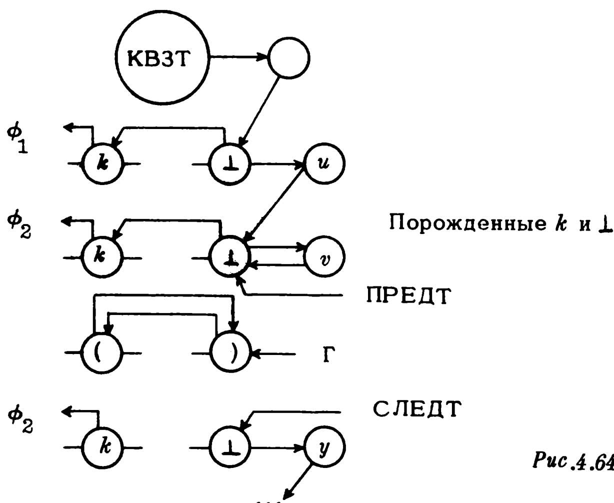


Рис.4.64

Активизация происходит так. В поле ПАР левой скобки записывается детерминатив. В поле ПР звена *v* (его можно найти как СЛ (ПРЕДТ)) записывается адрес правой скобки (т.е. значение Г). Затем переменной ПРЕДТ присваивается значение Г.

```
АКТО   ПАР(ПАР(Г)) = АРГ;
       ПР(СЛ(ПРЕДТ)) = Г;
       ПРЕДТ = Г;
       RETURN;
```

Теперь вы можете оценить удобство использования двух специальных звеньев КВЗТ и СЛ(КВЗТ): первые скобки активизируются по общему алгоритму.

По окончании замены нужно "зашить дырку" в списке, т.е. в звено, следующее за точкой с адресом ПРЕДТ заслать значение переменной СЛЕДТ. Это делает оператор EST, который будет описан в разд. 26.

Иногда для активизации можно использовать скобки, оставшиеся от конкретизируемого выражения. Адреса этих скобок хранятся в таблице элементов, куда они были занесены при отождествлении.

В этом случае используется оператор АКТ, n, ϕ, который активизирует правую скобку с номером n (рис. 4.65).

```
АКТ    ПАР(ПАР(ТЭ [N])) = АРГ;
       ПР(СЛ(ПРЕДТ)) = ТЭ [N];
       ПРЕДТ = ТЭ [N];
       RETURN;
```

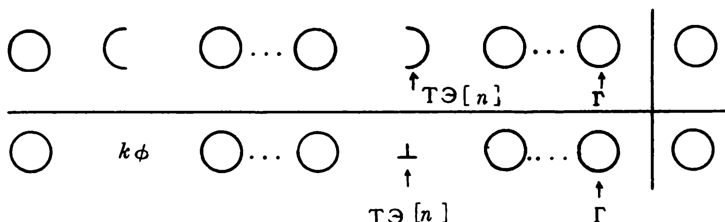


Рис.4.65

Переведем на язык сборки предложение:

```
 1        2
 k 'ФИ' ~  k ' АЛЬФА '  k'  БЕТА' ⊥⊥
```

ПРОВ; УГ,1; BL ; BR ; АКТО,БЕТА; АКТ,2,АЛЬФА; EST;
Последним стоит оператор EST, о котором и пойдет речь.

26. Завершение очередного и начало нового шага

Оператор EST завершает шаг рефал-машины и вместе с тем начинает выполнение нового шага.

Работа EST начинается с того, что "зашивается дырка" в списке областей конкретизации.

Затем подготавливается начальное состояние для нового шага: Г1 и Г2 устанавливают на ведущие k и ⊥ , адрес первого оператора функции пересылается из поля ПАР знака k в СЧА, переменная СЛЕДТ устанавливается на точку, которая станет ведущей, если новый шаг не породит знаков конкретизации, а ПРЕДТ – на звено КВЗТ.

Затем k и ⊥ превращаются в структурные скобки, после чего заканчивается подготовка: ГП устанавливается на начало стека переходов, адрес k заносится в ТЭ [1] , адрес точки в ТЭ [2] , в ТЭ [0] заносится адрес звена, предшествующего в поле зрения знаку k . (Это делается потому, что в процессе замены иногда бывает нужно поставить Г перед самым левым элементом объекта)

Установка НЭЛ = 3 завершает выполнение оператора EST . После этого интерпретатор языка сборки переходит к выполнению первого оператора функции, адрес которой находится в СЧА.

```
EST ПР(СЛ(ПРЕДТ)) = СЛЕДТ;
    Г2 = ПР(СЛ(КВЗТ));
    Г1 = ПАР(Г2);
    СЧА = ПАР(Г1);
    СЛЕДТ = ПР(СЛ(Г1));
    ПРЕДТ = АКВЗТ;
    ПР(СЛ(Г2)) = Г2;
    ПАР(Г1) = Г2;
    ГП = 1;
    ТЭ [0] = ПР(Г1);
    ТЭ [1] = Г1;
    ТЭ [2] = Г2;
    НЭЛ = 3;
    RETURN;
```

27. Дополнительные операторы

Ниже приведены операторы, которые введены для удобства программирования, экономии памяти и времени. Выполнение каждого из этих операторов эквивалентно выполнению нескольких ранее описанных операторов. Тем не менее, для каждого из них существует подпрограмма в интерпретаторе языка сборки. Эти подпрограммы приведены в конце на языке звеньев.

1) ЗНТ, n, $\underbrace{XY \ldots Z;}_{n \text{ объектных знаков } (n < 256)}$

Эквивалентная последовательность:

$$\underbrace{\text{ЗНАЧ},X;\ \ \text{ЗНАЧ},У;\ \ldots\ \text{ЗНАЧ},\ Z;}_{n \text{ операторов}}$$

2) ЗНТЯ, n , $\underbrace{XY \ldots Z;}_{n \text{ объектных знаков } (n < 256)}$

Эквивалентная последовательность:

$$\underbrace{\text{ЗНАЧЯ},X;\ \text{ЗНАЧЯ},Y;\ \ldots\text{ЗНАЧЯ},Z}_{n \text{ операторов}};$$

3) УДЗН, ϕ – эквивалент УД; ЗНАЧ, ϕ ;

4) УДСК – эквивалент УД; СКОБ;

5) УДСТС, n – эквивалент УД; СТС, n ;

Во время работы интерпретатора языка сборки операторы УДЗН, ϕ; УДСК и УДСТС, n представлены в виде:

ПУДЗН; УДЗН, ϕ ;
ПУДСК; УДСК;
ПУДЗН; УДСТС, n;

Схема переходов изображена на рис. 4.66.

Сравните с оператором УД; разд.8.

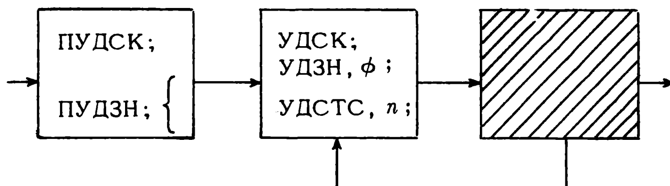


Рис.4.66

6) ИСК, ϕ эквивалентен УД; ЗНАЧ, ϕ ; КУД,1;

7) TEXT n, $\underbrace{\text{XY ... Z}}_{n \text{ объектных знаков } (n < 256)}$;

Эквивалентная последовательность

$\underbrace{\text{NS, X; NS, Y; ... NS, Z;}}_{n \text{ операторов}}$

8) BLR эквивалентен BL; BR.

Следующие два оператора не являются сокращениями, но дают возможность программировать на языке сборки эффективнее.

9) SBL отличается от BL тем же, чем SUB, ϕ отличается от NS, ϕ. Он формирует левую скобку из звена, следующего за Г.

10) SBR превращает звено, следующее за Г, в правую скобку.

11) SBLR заменяет последовательность операторов SBL; SBR.

```
ЗНТ    СДГ1;
       IF КОД(Г1) ≠ S (СЧА)  GO НЕОТ;
       ТЭ [НЭЛ] = Г1;
       НЭЛ = НЭЛ + 1;
       СЧА = СЧА + 1;
       N = N - 1;
       IF  N ≠ 0  GO ЗНТ;
       RETURN;
ЗНТЯ   СДГ2;
       IF КОД(Г2) ≠S (СЧА) GO НЕОТ;
       ТЭ [НЭЛ] = Г2;
       НЭЛ = НЭЛ + 1;
```

```
        СЧА = СЧА + 1;
        N = N - 1;
        IF N ≠ 0  GO ЗНТЯ;
        RETURN;
ПУДЗН   ЗПСП(Г1,Г2,НЭЛ,СЧА);
        ТЭ [НЭЛ] = 0;
        ТЭ [НЭЛ + 2] = Г1;
        RETURN;
УДЗН    IF ТЭ [НЭЛ] ≠ 0  GO УДЗН1;
        ТЭ [НЭЛ] = СЛ(Г1);
УДЗН1   Г1 = ТЭ [НЭЛ + 2];
УДЗН2   СДГ1;
        IF ¬СК(Г1)  GO УДЗН3;
        Г1 = ПАР(Г1);
        GO УДЗН2;
УДЗН3   IF КОД(Г1) ≠ АРГ  GO УДЗН2;
        IF  ТЭ [НЭЛ] ≠ Г1  GO УДЗН4;
        ТЭ [НЭЛ] = 0;
УДЗП4   ГП = ГП + 1;
        ТЭ [НЭЛ + 1]  = ПР(Г1);
        ТЭ [НЭЛ + 2]  = Г1;
        НЭЛ = НЭЛ + 3;
        RETURN;
ПУДСК   ЗПСП(Г1,Г2,НЭЛ,СЧА);
        ТЭ [НЭЛ] = 0;
        ТЭ [НЭЛ + 3]  = Г1;
        RETURN;
УДСК    IF ТЭ[ НЭЛ] ≠ 0 GO УДСК1;
        ТЭ [НЭЛ] = СЛ(Г1);
УДСК1   Г1 = ТЭ [НЭЛ + 3] ;
УДСК2   СДГ1;
        IF ¬СК(Г1)GO УДСК1;
        Г2 = ПАР(Г1);
        IF  ТЭ [НЭЛ] ≠ Г1 GO УДСК3;
        ТЭ [НЭЛ] = 0;
УДСК3   ГП = ГП + 1;
        ТЭ [НЭЛ + 1]  = ПР(Г1);
        ТЭ [НЭЛ + 2] = Г1;
        ТЭ [НЭЛ + 3]  = Г2;
        НЭЛ = НЭЛ + 4;
        RETURN;
УДСТС   IF  ТЭ [НЭЛ] ≠ 0 GO УДСТС1;
        ТЭ [НЭЛ] = СЛ(Г1);
```

```
УДСТС1 Г1 = ТЭ[НЭЛ + 2];
УДСТС2  СДГ1;
        IF ¬СК(Г1) GO УДСТС3;
        Г1 = ПАР(Г1);
        GO УДСТС2;
УДСТС3  IF КОД(Г1) ≠ КОД(ТЭ [N]) GOУДСТС2;
        IF ТЭ[НЭЛ] ≠ Г1 GO УДСТС4;
        ТЭ [НЭЛ] = 0;
УДСТС4  ГП = ГП + 1;
        ТЭ [НЭЛ + 1] = ПР(Г1);
        ТЭ [НЭЛ + 2] = Г1;
        НЭЛ = НЭЛ + 3;
        RETURN;
ИСК     ТЭ [НЭЛ] = СЛ(Г1);
ИСК1    СДГ1;
        IF ¬СК(Г1) GO ИСК2;
        Г1 = ПАР(Г1);
       GO ИСК1;
ИСК2    I FКОД(Г1) ≠ АРГ GO ИСК1;
        IF  ТЭ [НЭЛ] ≠ Г1 GO ИСК3;
        ТЭ [НЭЛ] = 0;
ИСК3    ТЭ [НЭЛ + 1] = ПР(Г1);
        ТЭ [НЭЛ + 2] = Г1;
        НЭЛ = НЭЛ + 3;
        RETURN;

TEXT    ПРК = СЛ(Г);
        СШИВ(Г,АНС);
TEXT1   КОД(АНС) =S(СЧА);
        Г = АНС;
        ВСВ;
        СЧА = СЧА + 1;
        N = N - 1;
        IF N ≠ 0 GO TEXT1;
        СШИВ(Г,ПРК);
        RETURN;
   BLR  ПРК = СЛ(Г);
        СШИВ(Г,АНС);
        Г = АНС;
        ВСВ;
        СВЯЗ(Г,АНС);
        Г = АНС;
        ВСВ;
        СШИВ(Г,ПРК);
        RETURN;
```

```
SBL   Г = СЛ(Г);
      ПАР(Г) = АСК;
      АСК = Г;
      RETURN;
SBR   Г = СЛ(Г);
      СЛСК = ПАР(АСК);
      СВЯЗ(АСК,Г);
      АСК = СЛСК;
      RETURN;
SBLR  Г = СЛ(Г);
      ПРК = СЛ(Г);
      СВЯЗ(Г, ПРК);
      RETURN:
```

28. Примеры

Теперь мы можем для любой функции, описанной на рефале, дать ее полный перевод на язык сборки.

```
1   3  5,6  4   2   1   3  5,6  4   2
kφ(    e1   )   ~   (  kφ  e1 ⊥   )
1   3  5,6  4   7  8,9  2   1   3  5,6  7   4  8,9  2
kφ(    e1   )  sx   e2    ~ kφ (    e1   sx )    e2  ⊥
1  3,4  2  3,4
kφ e1  ~   e1
```

На языке сборки после оптимизации получим:

```
φ :   УПЕР,Ь2;      Ы1:  СИМ;       Ь2:  ЗАКР;
      СКОБ;              ЗАКР;           УГ,0;
      ЗАКР;              УГ,6;           TPLE,4;
      УГР,4,2;           TPLS,7;         OUT,2;
      УПЕР,Ы1;           АКТ,2, φ        EST;
      ПРОВ;              EST;
      АКТ,4,φ
      EST;
```

Рассмотрим еще такой пример:

```
1          3   4,5 2  1          4,5 2 3
k 'REV'   sx  e1  ~  k   'REV'   e1 ⊥sx
1          3 5,6 4 7,8 2 1            7,8 2 3             5,6   4
k 'REV'   ( e1 ) e2 ~ k 'REV'   e2  ⊥ ( k 'REV'   e1 ⊥ )
1          2
k 'REV'   ~
```

```
REV:  УПЕР,Ь3;     Ь3:  УПЕР,Ь4;      Ь4:  ПРОВ;
      СИМ;              СКОБ;              УГ,0;
      ЗАКР;             ЗАКР;              OUT, 2;
      УГ,1;             УГР,4,2;           EST;
      ТРL,3,2;          ЗАКР;
      АКТО,REV;         УГ,1;
      EST;              ТРL,4,2;
                        АКТО, REV;
                        УГ,3;
                        BL;
                        CORAO, 6;
                        УГ,6;
                        BR;
                        АКТО, REV;
                        EST;
```

Сложней всего преобразование левой части в правую происходит во втором предложении второго примера:

$((e_1)\,e_2) \to (e_2)(e_1) \to$

$k\,'\mathrm{REV}'\,e_2 \perp (e_1) \to$

$k\,'\mathrm{REV}'\,e_2 \perp ((e_1) \to$

$k\,'\mathrm{REV}'\,e_2 \perp ((e_1)) \to$

$k\,'\mathrm{REV}'\,e_2 \perp (k\,'\mathrm{REV}'\,e_1 \perp)$

Этот способ в два раза короче стандартного:

$\langle$пусто$\rangle \to (\;\to (e_2 \;\to$

$(e_2) \to k\,'\mathrm{REV}'\,e_2 \perp \to$

$k\,'\mathrm{REV}'\,e_2 \perp (\;\to$

$k\,'\mathrm{REV}'\,e_2 \perp ((\;\to$

$k\,'\mathrm{REV}'\,e_2 \perp ((e_1 \;\to$

$k\,'\mathrm{REV}'\,e_2 \perp ((e_1) \;\to$

$k\,'\mathrm{REV}'\,e_2 \perp (k\,'\mathrm{REV}'\,e_1 \perp \;\to$

$k\,'\mathrm{REV}'\,e_2 \perp (k\,'\mathrm{REV}'\,e_1 \perp \;)$

Следует отметить, что оптимизация дает больший эффект не для таких простых примеров, а для сложных и громоздких предложений, где, как правило, есть такие большие куски, которые без (или почти без) изменения переходят из левой части в правую, и где левые части различных предложений функции похожи друг на друга. В качестве примера можно привести функцию ПОСТ из рефал-компилятора.

1 3 5,6 4 7 8 9 11 12 15,16 14 13 10 19,20 17 21,22 23 24,25 18 2

k'ПОСТ' ($e_{\text{с}}$) $s_{\text{н}}$ $s_{\text{к}}$ ('E' s_x e_1 'E' s_y) e_b (e_2 s_x e_3) ~
($e_{\text{с}}$) $s_{\text{н}}$ $s_{\text{к}}$ ('E' s_x e_1 'E' s_y) e_b (e_2 s_x e_3)

k'ПОСТ' ($e_{\text{с}}$) $s_{\text{н}}$ $s_{\text{к}}$ ('E' s_x e_1 'E' s_y) e_b (e_2 s_y e_3) ~
($e_{\text{с}}$) $s_{\text{н}}$ $s_{\text{к}}$ ('E' s_x e_1 'E' s_y) e_b (e_2 s_y e_3)

k' ПОСТ ($e_{\text{с}}$) $s_{\text{н}}$ $s_{\text{к}}$ ('E' s_x e_1 'E' s_y) e_2 ~
k'ПОСТ' ($e_{\text{с}}$) $s_{\text{н}}$ $s_{\text{к}}$ ('E' s_x e_1 'E' s_y)) e_2 ⊥

k 'ПОСТ' e_1 ~ e_1

После перевода этой функции на язык сборки и оптимизации получается:

```
ПОСТ:  УПЕР,Ь3;
       СКОБ;
       ЗАКР;
       УГР,4,2;
       СИМ;
       СИМ;
       СКОБ;
       ЗНАЧ, 'E';
       СИМ;
       СИМЯ;
       ЗНАЧЯ, 'E' ;
       ЗАКР;
       УГР,10,2;
       УПЕР,Ь2;
       СКОБЯ;
       ЗАКР;
       УГР,17,18;
       УПЕР,Ь1;
       УДСТС,12;
       ЗАКР;
       УГ,0;
       TPL,1,18;
       OUT,2;
       EST;
Ь1:    УДСТС,13;
       ЗАКР;
       УГ,0;
       TPL,1,18;
       OUT,2;
       EST,
Ь2:    ЗАКР;
       УГ,6,
       TPL,4,10;
       АКТ,2,ПОСТ;
       EST ;
Ь3.    ЗАКР;
       УГ,0;
       TPL,1,3;
       OUT,2;
       EST;
```

Хорошо видно, что хотя у всех параграфов громоздкие правые части, замена делается очень быстро и просто. Левые части у всех предложений тоже громоздкие, однако они похожи друг на друга,

поэтому длинное отождествление получилось только у первого предложения, а для остальных от отождествления почти ничего не осталось.

29. Словарь понятий языка звеньев

АКВЗТ - адрес КВЗТ (квазиточка - см.). Константа, т.е. переменная, имеющая постоянное значение, а именно - значение адреса фиктивного звена КВЗТ. Используется в операторе EST.

АНС - адрес начала ССП (список свободной памяти - см.). Переменная АНС имеет значение адреса первого звена на ССП. Адрес второго звена - это СЛ(АНС), третьего - СЛ(СЛ(АНС)) и т.д. Переменная АНС используется операторами, которые вставляют новые звенья в поле зрения (NS , MULE и т.п.), и оператором ' OUT , который пополняет ССП.

АРГ - аргумент. Если каждый оператор языка сборки рассматривать как процедуру, то АРГ - это формальный параметр в операторах, аргументом которых является КОД-значение (см.): ЗНАЧ, ϕ; УДЗН, ϕ; NS, ϕ и т.д. - или адрес: УПЕР, ϕ; АКТ, n, ϕ. То есть при обращении к подпрограмме, соответствующей оператору языка сборки, переменная АРГ получает значение аргумента ϕ.

АСК - адрес скобки. Переменная используется в операторах BL; BR; MULE, n; SBL; SBR для спаривания структурных скобок (см разд.14).

АСТ - адрес старого выражения. Переменная, используется в операторах СТВ, n; СТВЯ, n; MULE, n для просмотра старого выражения.

ВСВ - выделение свободного звена. Процедура, тело которой можно изобразить так:

АНС = СЛ(АНС); IF АНС = 0 GO СПИ;

где СПИ - точка входа в аварийную программу "свободная память исчерпана". О списке свободной памяти см. ССП.

Г - граница при замене (см.разд.13).

Г1 - первая граница при отождествлении (см.разд.4).

Г2 - вторая граница при отождествлении (см.разд.4).

ГП - глубина перехода. Индекс стэка переходов (см.разд.8).

ЗПСП - запись в стэк переходов. Процедура, имеющая четыре параметра: Г1 (адрес звена), Г2 (адрес звена), НЭЛ (целое число), АВ (адрес возврата). Заполняет строку стэка переходов СП [ГП] (см.разд.8).

КВЗТ - квазиточка. Отдельные звенья КВЗТ и СЛ (КВЗТ) используются операторами АКТ, n, ϕ; АКТО, ϕ; EST; при формировании списка областей конкретизации (см.разд.25, 26).

КОД - выделитель поля звена. О структуре звена см.разд.2. Для того, чтобы прочесть или изменить значение какого-нибудь поля звена, надо к нему обратиться так:

<выделитель поля> (< адрес звена >)

Каждому полю соответствует одноименный выделитель поля: КОД, П, ПАР, СЛ, ПР.

КОД-значение – это значение, которое можно записать в поле КОД, то есть состоящее из двух подполей П и ПАР.

М – формальный параметр процедуры, описывающей оператор языка сборки (см.АРГ). Имеет тип "целое".

НЭЛ – номер элемента (см.разд.3).

П – признак. Выделитель поля (см.КОД).

ПР – адрес предыдущего звена. Выделитель поля (см.КОД).

ПРЕДТ – предыдущая точка. Переменная, используемая операторами АКТ, n, ϕ; АКТО, ϕ; EST. (см.разд.25, 26).

ПРК – правый конец. Вспомогательная переменная, принимающая значение адреса звена. Используется в операторах замены при изменении поля зрения. Свое название переменная ПРК получила от того, что при разрывах списковой структуры она указывает на правый конец разрыва (рис.4.67).

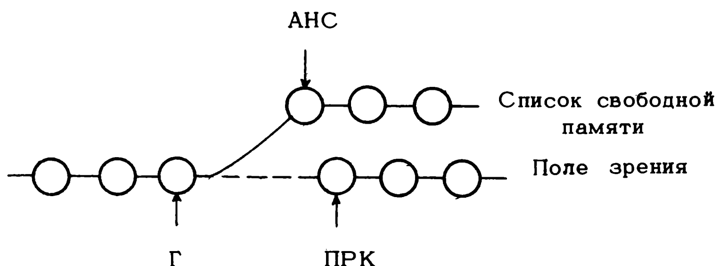


Рис.4.67

СВЯЗ – связка скобок. Процедура, имеющая два параметра – адреса двух звеньев. СВЯЗ формирует из указанных звеньев структурные скобки: первое звено становится левой скобкой, второе – парной правой скобкой. Для этого в поля П заносятся признаки левой и правой скобки соответственно, а в поле ПАР каждого из двух звеньев заносится адрес другого звена, т.е. парной скобки.

СДГ1 – сдвиг Г1. Процедура, тело которой имеет вид:

```
Г1 = СЛ(Г1);
 IF Г1 = Г2 GO НЕОТ;
```

СДГ2 – сдвиг Г2. Процедура, тело которой имеет вид:

```
Г2 = ПР(Г2);
 IF Г1 = Г2 GO НЕОТ;
```

СК – скобка. Булевская процедура-функция, имеющая один параметр – адрес звена. Принимает значение "истина" тогда и толь-

ко тогда, когда это звено является структурной скобкой (правой или левой).

СКЛ - скобка левая.

СКП - скобка правая.

СЛ - адрес следующего звена. Выделитель поля (см.КОД).

СЛЕДТ - следующая точка. Переменная используется операторами АКТ,*n*,*ф*; АКТО, *ф*; EST; (см.разд.25, 26).

СЛСК - следующая скобка. Вспомогательная переменная, принимающая значение адреса звена. Используется в операторах **BR**; **MULE,*n*; SBR** при спаривании структурных скобок.

ССП - список свободной памяти. Звенья, не занятые в поле зрения, связаны в список, называемый списком свободной памяти. ССП организован так же, как поле зрения. В поле СЛ последнего звена ССП записан ноль, что служит признаком конца ССП (см.ВСВ).

СЧА - счетчик адреса операторов языка сборки. Считается, что операторы языка сборки записываются в памяти машины и интерпретируются аналогично командам машины. При этом значением СЧА является адрес очередного оператора языка сборки, который нужно выполнить. Интерпретатор языка сборки выбирает этот оператор, дешифрует его код, присваивает формальным параметрам (N,M, АРГ) значения аргументов оператора и передает управление на соответствующую подпрограмму, сдвинув СЧА на следующий оператор. Оператор языка звеньев **RETURN** (см.) возвращает управление интерпретатору языка сборки, который выбирает следующий оператор. Сказанное справедливо для всех операторов, кроме ЗНТ,*n*, <текст>; ЗНТЯ, *n*,<текст>;TEXT,*n* ,<текст>, для которых выбирается только первый аргумент *n*, СЧА устанавливается на начало текста и передается управление подпрограмме оператора, которая сама выбирает символы, составляющие текст (см. *S*). При этом считается, что каждый символ имеет свой адрес, адрес следующего символа на единицу больше адреса предыдущего и вслед за последним символом располагается следующий оператор языка сборки, на который в конце установится СЧА. Помимо операторов ЗНТ, *n*, <текст>; ЗНТЯ, *n*, <текст>; TEXT,*n*,<текст>; переменная СЧА используется в операторе **EST**, где СЧА устанавливается на начало функции, и в операторах ПУД; ПУДСК; ПУДЗН при занесении адреса следующего оператора в стэк переходов.

СЧСП - считывание строки из стэка переходов. Процедура имеет четыре параметра, которые должны быть идентификаторами переменных. Четыре значения из строки стэка переходов СП [ГП] присваиваются этим переменным.

СШИВ - сшивка звеньев. Процедура, имеющая 2 параметра: А1 и А2 - адреса двух звеньев. Ее тело имеет вид:

```
СЛ(А1) = А2;
ПР(А2) = А1;
```

ТЭ – таблица элементов (см.разд. 3). Массив, каждый элемент которого может принимать значение адреса звена. Элемент массива ТЭ называется строкой таблицы элементов.

IF – если. Собственное слово языка звеньев. Стоит в начале условного оператора.

GO – перейти к . . . . Собственное слово языка звеньев. Стоит в начале оператора перехода.

N – формальный параметр процедуры, описывающей операторы языка сборки (см. АРГ). Имеет тип "целое".

RETURN – вернуться. Собственное слово языка звеньев. Оператор возврата – возвращает управление интерпретатору языка сборки, который выбирает следующий оператор.

S – *symbol* (символ). Процедура-функция, имеющая один параметр – текущее значение счетчика адреса СЧА (см.). Используется в операторах ЗНТ, *n*, <текст> ; ЗНТЯ, *n*, <текст>; TEXT, *n*, <текст>; для выборки очередного символа текста (см. СЧА). Формирует из символа, адрес которого – значение СЧА, КОД-значение (см.).

30. Список операторов языка сборки

АКТ,
АКТ, *n*, ф;
АКТО, ф;
BL;
BLR;
BR;
EST;
ЗАКР;
ЗНАЧ, ф;
ЗНАЧЯ, ф;
ЗНТ, *n*, <текст>;
ЗНТЯ, *n*, <текст>;
ИСК, ф;
КУД, *n*;
MULE, *n*;
MULS, *n*;
НЕОТ;
OUT, *n*;
ПРОВ;
ПУД;
ПУДЗН;
ПУДСК;
СИМ;
СИМЯ;
СКОБ;
СКОБЯ;
CORA, *n*, *m*;
CORAO, *n*;
СТВ, *n*;
СТВЯ, *n*;
СТС, *n*;
СТСЯ, *n*;
ТЕРМ;
ТЕРМЯ;
TEXT, *n*, <текст>;
TPL, *n*, *m*;
TPLE, *n*;
TPLS, *n*;
УГ, *n*;
УГР, *n*, *m*;
УД;
УДЗН, ф;
УДСК;
УДСТС, *n*;
УПЕР, ф;
NS, ф;
SBL;
SBLR;
SBR;
SUB, ф;

У. КОМПИЛЯТОР С РЕФАЛА НА ЯЗЫК СБОРКИ

1. Предварительные замечания

Здесь описан простейший компилятор с рефала на язык сборки. Этот компилятор не пытается ограничивать области удлинения с помощью оператора КУД, *m* и не пытается искать кратчайший способ замены левой части на правую. Он служит исходной точкой для построения более сложных компиляторов с рефала. Прежде чем разбираться в них, необходимо ознакомиться с простейшим компилятором. Кроме того, при реализации рефала на вычислительной машине из существования процессора для языка сборки и компилятора на него, написанного на рефале, еще не вытекает, что рефал реализован. Чтобы получить работающий компилятор, нужно вручную перевести компилятор на язык сборки.

С этого момента, имея в руках хотя бы один работающий вариант компилятора, можно строить его более сложные версии. Естественно, что при этом переводить вручную с рефала на язык сборки больше не потребуется.

Предлагаемый компилятор прост, а вследствие этого - невелик по объему. Поэтому странслировать его на язык сборки вручную не составляет особого труда.

2. Внутренняя структура данных

Представление рефала на перфокартах может быть самым разнообразным. То же самое можно сказать про язык сборки. Предлагаемый компилятор не зависит от внешнего представления данных. Он использует свое собственное представление для исходных предложений рефала и выходных операторов языка сборки. При этом предполагается, что исходные предложения рефала на входе преобразуются во внутренний код компилятора. Точно так же результирующие операторы языка сборки на выходе должны преобразовываться во внешний вид.

Соответствующие программы преобразования пишутся на рефале и зависят от внешнего представления данных. Здесь мы их не приводим.

Опишем внутреннее представление предложений рефала. Каждое предложение представляется в виде двух термов:*

(ЕО) (ЕЦ)

Выражение ЕО соответствует левой части предложения, а выражение ЕЦ - правой. Таким образом, ЕО и ЕЦ изображают некоторые выражения рефала.

---

* *В дальнейшем изложение ведется с использованием метакода-Б (см. главу У1, разд. 2).*

Назовем элементом выражения каждый из следующих объектов:

1. Символ.
2. Переменную.
3. Левую и правую скобку
4. Знак конкретизации или конкретизационную точку.

Во внутреннем коде каждый элемент выражения представляется в виде:

< тип элемента > < информация >

Здесь < тип элемента > - это любой из составных символов:

/Z/, /S/, /W/ , /E/,/B/, /K/,

а<информация>- некоторый терм.

Можно выписать следующую таблицу соответствия между представлением различных объектов рефала на перфокартах и во внутреннем коде компилятора:

На перфокарте	Во внутреннем коде компилятора
'A'.	/Z/'A'
'ABC'	/Z/ 'A'/Z/'B'/Z/'C'
/ФИ/	/Z/ ('ФИ')
SX	/S/'X'
WУ	/W/ 'У'
E1	/E/'1'
K	/B/ 'K'
.	/B/ 'T'

Скобки представлены в ЕО и ЕЦ по-разному.

В ЕО имеет место соответствие:

(	/B/ (
)	)

В ЕЦ имеет место соответствие:

(	/B/ 'L'
)	/B/ 'R'

Таким образом, предложение рефала

E1 /B/ 'T' (E2) = K /PA3M/ E1 (/B/ 'R'(E2)).

во внутреннем коде компилятора представится в виде:

(/E/ '1' /Z/ ('B') /Z/ 'T'/B/ (/E/ '2'))

(/B/ 'K' /Z/ ('PA3M') /E/ '1' /B/ 'L' /Z/ ('B')

/Z/ 'R'/B/ 'L'/E/ '2' /B/ 'R' /B/ 'R'/B/ 'T')

Теперь опишем внутреннее представление операторов языка сборки.

Каждый оператор языка сборки изображается термом. Проще всего представлены операторы, которые не имеют аргументов. Каждый такой оператор изображается символом-меткой, тело которого совпадает с названием оператора. Так, например, оператор ТЕРМ; представляется в виде /ТЕРМ/, оператор BLR – в виде /BLR/ и т.д.

Если оператор имеет хотя бы один аргумент, он изображается термом вида (SO EX), где SO является символом-меткой, а EX изображает аргументы оператора.

Аргументом оператора, как известно, может быть натуральное число, метка, объектный знак, составной символ рефала или текст, состоящий из объектных знаков.

Натуральное число n изображается символом-числом /n/. Метка изображается последовательностью объектных знаков, из которых состоит эта метка. Составной символ /x/ изображается (x). Здесь под x подразумевается тело этого составного символа.

Объектный знак или текст, состоящий из объектных знаков, изображаются сами собой:

Оператор	Его представление
TPL, 5,6	(/TPL / /5/ /6/)
ЗНАЧ,+;	(/ЗНАЧ/ '+')
NS, /ФИ/;	(/NS / ('ФИ'))
УПЕР,ПСИ;	(/УПЕР/ 'ПСИ')
АКТ, ХИ,8;	(/АКТ/ 'ХИ' /8/)
TEXT,4,ГОГА;	(/TEXT/ /4/ 'ГОГА')
SUB, (;	(/ SUB/ '(')
СКОБ;	/СКОБ/
BL ;	/BL /

Теперь обратимся к тексту компилятора на рефале.

Ядром компилятора является функция КОМП. Формат обращения к ней:

К /КОМП/ (ЕО) (ЕЦ).

где ЕО – левая часть, а ЕЦ – правая часть некоторого предложения рефала, представленные во внутреннем коде компилятора.

Результат работы этой функции имеет вид:

(EX) (EY)

Здесь EX – последовательность операторов отождествления, соответствующая EO, а EY – последовательность операторов замены, соответствующая EЦ.

Функция КОМП обращается к функциям ОТОЖН, ОПРАЗ и ЗАМ. ОТОЖН вырабатывает из левой части последовательность операторов отождествления. ОПРАЗ производит обработку правой части перед заменой. ЗАМ вырабатывает операторы замены.

Опишем ОТОЖН. Эта функция сразу же обращается к ОТОЖ и ОБР. Всю работу, в сущности, выполняют ОТОЖ и ОБР.

Формат обращения к ОТОЖ и ОБР следующий:

K/ОТОЖ/ ST(EC) EX (EA).

Здесь выражение (EC)EX является представлением для последовательности дыр, которые образуются в процессе проектирования элементов левой части. Скобки поставлены для того, чтобы отметить положение той дыры, из которой проектируется очередной элемент.

Каждая дыра представлена в виде

SH SK (EX)

Здесь EX сама дыра, а SH и SK–номер конца элемента, который предшествует в левой части дыре, и номер начала элемента, который в левой части непосредственно следует за дырой.

Спроектированные элементы из системы дыр вынуты.

ST – это номер первой свободной строки в таблице элементов.

EA – это таблица переменных, которые уже приняли значение. Для каждой переменной, принявшей значение, в ней есть два символа:

SX SN

Здесь SX – индекс этой переменной, а SN – номер конца главного вхождения этой переменной.

При работе ОТОЖ и ОБР используются функции P1 и P2. Эти функции – машинные процедуры. Формат обращения к ним

K/P1/ SX.

K/P2/ SX.

Здесь SX – символ-число. После вычисления P1 в поле зрения остается символ-число, на единицу больший, чем SX. Функция P2 увеличивает SX на две единицы.

Например: K /P1/ /26/. заменится на /27/, а K /P2/ /256/. заменится на/258/.

ОТОЖ и ОБР пытаются проектировать элементы из текущей дыры. Если это невозможно, они обращаются к функции ВЫДЫ, которая пытается выбрать другую дыру. Если это не удается, происходит обращение к функции ГУД, которая генерирует оператор УД.

Когда ни одной дыры не остается, компиляция левой части завершается.

Теперь опишем функцию ОПРАЗ. Она обращется к функции РАЗМ. Назначение РАЗМ следующее. Каждую комбинацию вида:

/B/ 'K' /Z/ WF ... /B/ 'T'

она преобразует в

/B/ 'L' ... /B/ 'R' /K/ WF

Это облегчает работу для функции ЗАМ, так как тогда порождение операторов АКТ, ϕ, n становится тривиальным.

Опишем функцию ЗАМ. Формат обращения к ней

K /ЗАМ/ ЕЦ (ЕА) (ЕВ).

ЕА – это таблица тех переменных, которые еще не оказались слева от Г в процессе замены.

ЕВ – таблица номеров концов главных вхождений всех переменных.

Описание КОМП завершено.

Чтобы скомпилировать целую функцию, мало перевести каждое из ее предложений в отдельности. Нужно еще расставить операторы УПЕР, с учетом совпадения левых частей.

Чтобы скомпилировать функцию, состоящую из предложений

E1, E2, ... EN

в поле зрения нужно последовательно занести области конкретизации

K/REFAL / E1.
K/REFAL / E2.
. . . .
K/REFAL / EN.
K/END /.

Функции REFAL и END обращаются к функциям REFA и КОМП.

Объединение совпадающих левых частей и расстановку операторов УПЕР, ϕ производит REFA. Разобрать ее работу представляем читателю.

ВК и ЗК, СЧ и ЗП – это машинные процедуры выкапывания и закапывания, считывания и записи.

Функция ПВЫВ подготавливает операторы языка сборки к выводу.

Ее основное назначение – заменять некоторые комбинации операторов на их сокращения. Так BL BR заменяется на BLR и т.д.

Функция ВЫВ выводит готовые операторы языка сборки куда следует.

```
КОМП     (EO)(ЕЦ) = K/КОМП1/(K/ОТОЖН/EO.)                        1
                (K/ОПРАЗ/ЕЦ.).
КОМП1    (EX WA) (ЕЦ) = (EX)(/УГ//0/)                             1
                K/ЗАМ/ЕЦ WA WA.(/'EST/)
ОТОЖН    EO = K/ОТОЖ/ /3/ ( ) /1/ /2/ (EO) ( ).
ОТОЖ     ST WC SH SK (/Z/ WX E1) E2 = (/ЗНАЧ/WX)                  1
```

```
    K/ОТОЖ/ K/P1/ST. WC ST SK(E1) E2.
ST WC SH SK(/B/WX E1) E2 = /СКОБ/                     1
    K/ОТОЖ/ K/P2/ST. WC ST K/P1/ST.                   2
    WX K/P1/ST. SK (E1) E2.
ST WC SH SK (/S/SX E1) E2 (EA SX SN EB) =             1
    (/СТС/ SN) K/ОТОЖ/ K/P1/ST. (WC                   2
    ST SK(E1) E2 (EA SX SN EB SX ST).
ST WC SH SK (/S/SX E1) E2 (EA) = /СИМ/                1
    K/ОТОЖ/ K/P1/ST. WC ST SK (E1) E2                 2
    (EA SX ST).
ST WC SH SK (/W/SX E1) E2 (EA SX SN EB) =             1
    (/СТВ/ SN) K/ОТОЖ/ K/P2/ST. WC                    2
    K/P1/ST. SK (E1) E2 (EA SX SN EB SX               3
    K/P1/ST.).
ST WC SH SK (/W/SX E1) E2 (EA) = /ТЕРМ/               1
    K/ОТОЖ/ K/P2/ST. WC K/P1/ST. (SK                  2
    (E1) E2 (EA SX K/P1/ST.).
ST WC SH SK (/E/SX E1) E2 (EA SX SN EB) =             1
    (/СТВ/SN) K/ОТОЖ/ K/P2/ST. WC                     2
    K/P1/ST. SK (E1) E2 (EA SX SN EB SX               3
    K/P1/ST.).
ST WC SH SK (/E/SX E1) E2 (EA) =                      1
    K/ОБР/ST WC SH SK (/E/SX E1) E2 (EA).
ST WC SH SK ( ) E2 = /ПРОВ/ K/ВЫДЫ/ST SH               1
    SK WC E2.
ОБР ST WC SH SK (E1 /Z/ WX) E2 = (/ЗНАЧЯ/ WX)         1
    K/ОБР/ K/P1/ST. WC SH ST (E1) E2.
ST WC SH SK (E1 /S/SX) E2 (EA SX SN EB) =             1
    (/СТСЯ/SN) K/ОБР/ K/P1/ST. WC SH                  2
    ST (E1) E2 (EA SX SN EB SX ST).
ST WC SH SK (E1 /S /SX) E2 (EA) = /СИМЯ/              1
    K /ОБР/ K/P1/ST. WC SH ST (E1) E2                 2
    (EA SX ST).
ST WC SH SK (E1 /W /SX) E2 (EA SX SN EB) =            1
    (/СТВЯ/SN) K/ОБР/ K/P2/ST. WC SH ST               2
    (E1) E2 (EA SX SN EB SX K/P1/ST.).
ST WC SH SK (E1 /W /SX) E2 (EA) = /ТЕРМЯ/             1
    K/ОБР/ K/P2/ST. (WC SH ST (E1) E2                 2
    (EA SX K/P1/ST.).
ST WC SH SK (E1 /E/SX) E2 (EA SX SN EB) =             1
    (/СТВЯ/SN) K/ОБР/ K/P2/ST. WC SH ST               2
    (E1) E2 (EA SX SN EB SX K/P1/ST.).
ST WC SH SK (/E/ SX) E2 (EA) = /ЗАКР/                 1
    K/ВЫДЫ/ K/P2/ST. SH ST WC E2                      2
    (EA SX K/P1/ST.).
```

```
         ST WC SH SK (E1 /E/SX)E2 = K/ВЫДЫ/ST                              1
             SH SK WC SH SK (E1 /E/SX)E2.
         ST WC SH SK (E1/B/ WX) E2 = /СКОБЯ/                               1
             K/ОБР/ K/P2/ST. WC SH ST (E1) ST                              2
             K/P1/ST. WX E2.
ВЫДЫ     ST SP SQ ( ) WA = WA
         ST SP SQ (EC) E1 = K/ВЫДЫ1/ST SP SQ                               1
             K/ПОИДЫ/ ( ) EC E1..
ВЫДЫ1    ST SP SQ (SH SK (E1) E2) WA = K/ГУГР/                             1
             SH SK SP SQ. K/ГУД/ST ( ).SH SK                               2
             (E1) E2 WA.
         ST SP SQ WC SH SK (E1)E2 = (/УГР/SH SK)                           1
             K/ОТОЖ/ST WC SH SK (E1) E2.
ГУГР     SH SK SH SK =
         SH SK E1 = (/УГР/SH SK)
ГУД      ST WC SH SK (/E/SX E1) E2 (EA) = /УД/                             1
             K/ОТОЖ/ K/P2/ST. WC K/P1/ST. SK                               2
             (E1)E2 (EA SX K/P1/ST.).
ПОИДЫ    (EC) SH SK(/E/SX E1 /E/SY)EB(E2 SX E3) =                          1
             (EC) SH SK (/E/SX E1 /E/SY) EB(E2 SX E3)
         (EC) SH SK (/E/SX E1 /E/SY)EB (E2 SY E3) =                        1
             (EC) SH SK (/E/SX E1 /E/SY) EB (E2 SY E3)
         (EC) SH SK (/E/SX E1 /E/SY) EB =                                  1
             K/ПОИДЫ/ (EC SH SK (/E/SX E1 /E/SY))EB.
         E1 = E1
ЗАМ      /Z/ WX EЦ WA WB = (/NS/WX) K/ЗАМ/EЦ WA WB.
         /B/'L' EЦ WA WB = /BL/ K/ЗАМ/EЦ WA WB.
         /B/ 'R' EЦ WA WB = /BR/ K/ЗАМ/EЦ WA WB.
         /S/ SX EЦ (E1 SX SN E2)WB = (/TPLS/SN)                            1
             K/ЗАМ/ EЦ (E1 E2) WB.
         /S/ SX EЦ WA (E1 SX SN E2) = (/MULS/SN)                           1
             K/ЗАМ/ EЦ WA (E1 SX SN E2).
         /W/ SX EЦ (E1 SX SN E2) WB = (/TPLE/SN)                           1
             K/ЗАМ/ EЦ (E1 E2 ) WB.
         /W/ SX EЦ WA (E1 SX SN E2) = (/MULE/SN)                           1
             K/ЗАМ/ EЦ WA (E1 SX SN E2).
         /E/ SX EЦ (E1 SX SN E2) WB = (/TPLE/SN)                           1
             K/ЗАМ/ EЦ (E1 E2) WB.
         /E/ SX EЦ WA (E1 SX SN E2) = (/MULE/SN)                           1
             K/ЗАМ/ EЦ WA (E1 SX SN E2).
         /K/ (EP) EЦ WA WB = (/AKTO/EP)                                    1
             K/ЗАМ/ EЦ WA WB.
         WA WB =

ОПРАЗ    EЦ = K/РАЗМ/ EЦ ( ).
РАЗМ     E1 /B/ 'T' (E2) = K/РАЗМ/ E1 (/B/'R' (E2)).
         E1 /B/ 'K' (/Z/ WF E2 (E3)) =                                     1
```

```
          K/РАЗМ/E1 (/B/ 'L' E2 /K/ WF E3).
        (E2) = E2
END     = K/END1/ K/BK/ 'T'..
END1    =
        EA = K/ВЫВ/ K/ПВЫВ/ K/ВДЕР/ ( ( ) EA)...
REFAL   EP = K/REFA/ K/КОМП/ EP..
REFA    (EL) (ER) = K/REFB/ (K/ВУД/EL. (/B/ ER))                1
          K/BK/ 'T' ..
REFB    (WO E1) = K/ЗК/ 'T =' WO E1 ( ).
        (WO E1) WO E2 = K/ЗК/'T=' WO K/ЗАД/ (E1)                1
          E2..
        (WO E1) E2 = K/ВЫВ/ K/ПВЫВ/ K/ВДЕР/                     1
          ((( /УПЕР/ K/СЧ/ 'M'.)) E2)..                         2
        (/МЕТ/ K/СЧ/ 'M'.). K/СДМ/.                             3
        K/ЗК/'T =' WO E1 ( ).
ВДЕР    ((EM) E1 (E2)) E3 = EM E1 K/ВДЕР/E2 E3.
        =
ЗАД     (WO E1)(E2 ((EУ) WO E3)) =                              1
          (E2 ((EУ) WO  K/ЗАД/ (E1) E3.))
        (WO E1) (E2 ((EУ) E3)) =                                1
          (E2 ((EУ (/УПЕР/ K/СЧ/' M' .))E3)                     2
          (((/МЕТ/ K/СЧ/' M' .)) WO E1( ))) K/СДМ/.
        (WO E1) WO E2 = WO K/ЗАД/ (E1) E2.
        (WO E1) E2 = (((( /УПЕР/ K/СЧ/'M'.)) E2)                1
         (((/МЕТ/ K/СЧ/' M'.)) WO E1 ( ))) K/СДМ/.
        ( ) ( ) = ( )
ВУД     E1 /УД/ E2 = E1(/B/ /УД/ E2)
        E1 = E2
ПВЫВ    (/B/E1) E2 = K/ПВЫВ/ E1 E2.
        /УД/ (/ЗНАЧ/ EA) E1 = (/УДЗН/EA)                        1
          K/ПВЫВ/E1.
        /УД//СКОБ/ E1 = /УДСК/ K/ПВЫВ/ E1.
        /BL//BR/ E1 = /BLR/ K/ПВЫВ/ E1.
        (/ЗНАЧ/SA) E1 = K/СЛОП/ /ЗНАЧ/                          1
          (/ЗНТ/ /1/ SA ) E1.
        (/ЗНАЧЯ/SA) E1 = K/СЛОП/ /ЗНАЧЯ/                        1
         (/ЗНТЯ/ /1/ SA) E1.
        (/NS /SA) E1 = K/СЛОП/ /NS /(/TEXT/ /1/ SA) E1.
        WO E1 = WO K/ПВЫВ/ E1.
        =
СЛОП    SO (SX/255/EA) E1 = (SX/255/EA)                         1
          K/ПВЫВ/ E1.
        SO (SX SN EA) (SO SB) E1 =                              1
          K/СЛОП/ SO (SX K/P1/SN. EA SB) E1.
        SO (SX SN EA)E1 = (SX SN EA) K/ПВЫВ/E1.
СДМ     = K/ЗК/ 'M=' K/P1/ K/BK/'M'...
```

УI. ВХОДНОЙ ЯЗЫК ДЛЯ ЭВМ ТИПА ЕС ЭВМ, БЭСМ-6, «МИНСК-32»

Инструкция для машин ЕС ЭВМ, БЭСМ-6,"Минск-32" разделена на две части. Эта глава содержит первую часть. Здесь описано то, что реализовано на всех трех машинах. Вторая часть (главы УП-1Х) содержит специфические для каждой машины элементы.

Реализация рефала на малых машинах БЭСМ-4, М-220 и М-222 имеет много особенностей. Поэтому инструкция пользователям, работающим на этих машинах, вынесена в отдельную главу. В настоящей главе от читателя не требуется никаких специальных знаний об ЭВМ. Если вы не ознакомились с реализацией рефала (главы 1У-У), то рекомендуем прочесть разд.1 главы 1У.

1. Перфокарта. Бланк

Рефал-программа набивается на перфокартах. Каждая перфокарта содержит 80 колонок. Одна колонка содержит один символ; таким образом, на каждой перфокарте помещается ровно 80 символов. Для того чтобы отперфорировать программу, вы должны записать ее на бланках, разграфленных на строки по 80 позиций в каждой. Одна строка бланка кодируется на одной перфокарте.

Некоторые позиции можно оставлять пустыми, поэтому для ЕС ЭВМ и БЭСМ-6 можете использовать бланки, имеющие 72 позиции, а для "Минск-32" можно использовать бланки, предназначенные для любых других языков.

Каждое предложение рефала записывается на одной или нескольких строках, то есть набивается на одной или нескольких картах. Если вы продолжаете предложение на следующей строке, то должны сообщить об этом транслятору. Это делается на разных машинах по-разному.

ЕС ЭВМ и БЭСМ-6 (рис. 6.1., 6.2). Для указания продолжения предложения на следующую строку запишите в 72-й позиции любой символ, отличный от пробела. Если в 72-й позиции пробел (на бланке ничего не написано), транслятор считает, что это последняя карта предложения.

72-ю позицию удобно использовать для нумерации карт внутри предложения; на первой карте напишите единицу в 72-й позиции, на второй - двойку и так далее, а на последней - 72-ю позицию оставьте пустой.

73 - 80-я позиции используются для нумерации карт, которая нужна при записи рефал-программы в архив на магнитную ленту или пакет дисков. Рефал-компилятор игнорирует эти позиции, поэтому вы можете оставить их пустыми.

В 1 - 71-й позициях записывается само предложение в метакоде-Б (см. разд. 2).

"Минск-32" (рис. 6.3). В 1 - 5-й позициях кодируется наименование массива карт (имя программы). 6 - 11-я позиции служат для нумерации карт.

ЕС ЭВМ	ПРОГРАММА	ЯЗЫК	ЦНИПИАСС	Отдел	Шифр темы	Составил	Лист
	ИСХОДНЫЕ ДАННЫЕ	ПК/ПЛ				Телефон Дата	Всего

```
         START
         EXTRN CARD,PROUT
         ENTRY DELO
DELO   =  K/PROUT/ K/SIM/ K/SPAR/() K/CARD/ ....
* 'PARENTHESES  PAIRING' FUNCTION  (IT IS A COMMENT)
SPAR     (E1) ' ' E2 = E1
         (E1) '(' E2 = K/SPAR/((E1)) E2.
         ((E1) E2) ')' E3 = K/SPAR/(E1(E2)) E3 .
         (E1) SX E2 = K/SPAR/ (E1 SX) E2 .
* 'TEST ON SYMMETRY' FUNCTION
SIM    = 'SYMMETRICAL'
        SA = 'SYMMETRICAL'
        SA E1 SA = K/SIM/ E1.
        (E1) = K/SIM/ E1 .
        () E1 () = K/SIM/ E1 .
        (WA E1) E2 (E3 WB) =                                        1
                    K/SIM/ WA (E1) E2 (E3) WB .
        E1 = 'NONSYMMETRICAL'
         END DELO
```

Рис.6.1

Программа	Программист	Дата	Стр–ца	

Метка 1–6	7	8–72	Номер 73–80
ьДЕЛО		СТАРТ	
		ВХОДН ДЕЛО	
		ВНЕШН КАРТА,П	
ДЕЛО		= К/П/ К/СИММ/ К/СПАР/() К/КАРТА/	
* ФУНК		ЦИЯ 'СПАРИВАНИЕ СКОБОК' (ЭТО КОММЕНТАРИЙ)	
СПАР		(Е1) ' ' Е2 = Е1	
		(Е1) '(' Е2 = К/СПАР/((Е1)) Е2 .	
		((Е1) Е2) ')' Е3 = К/СПАР/(Е1 (Е2)) Е3 .	
		(Е1) SX Е2 = К/СПАР/(Е1 SX) Е2 .	
* ФУНК		ЦИЯ 'ПРОВЕРКА ВЫРАЖЕНИЯ НА СИММЕТРИЧНОСТЬ'	
СИММ		= 'СИММЕТРИЧНОЕ'	
		SA = 'СИММЕТРИЧНОЕ'	
		SA Е1 SA = К/СИММ/ Е1 .	
		(Е1) = К/СИММ/ Е1 .	
		() Е1 () = К/СИММ/ Е1 .	
		(WA Е1) Е2 (Е3 WB) = 1	
		K/СИММ/ WA (Е1) Е2 (Е3) WB	
		Е1 = 'НЕСИММЕТРИЧНОЕ'	
		ФИНИШ	

Рис.6.2

ССК МИНСК	ПРОГРАММА		СОСТАВИЛ ДАТА ИДЕНТИФ СПАСИ ЛИСТ 001 ЛИСТОВ 01
Строка 0–11	Этикетка 12–16	17–21	Адреса и замечания 22–60
010		СТАРТ	
020		ВНЕШН	КАРТА, П
030		ВХОДН	ДЕЛО
040	ДЕЛО	= К/П/	К/СИММ/ К/СПАР/() К/КАРТА/ .
050	* ФУ	НКЦИЯ	'СПАРИВАНИЕ СКОБОК' (ЭТО КОММЕНТАРИЙ)
060	СПАР	(E1)	' ' E2 = E1
070		(E1)	'(' E2 = K/СПАР/((E1)) E2 .
080		((E1	)E2) ')' E3 = K/СПАР/(E1(E2)) E3 .
090		(E1)	SX E2 = K/СПАР/(E1 SX) E2 .
100	* ФУ	НКЦИЯ	'ПРОВЕРКА ВЫРАЖЕНИЯ НА СИММЕТРИЧНОСТЬ'
110	СИММ	= 'С	ИММЕТРИЧНОЕ'
120		SA =	'СИММЕТРИЧНОЕ'
130		SA E1	SA = K/СИММ/ E1 .
140		(E1)	= K/СИММ/ E1 .
150		() E1	() = K/СИММ/ E1 .
160		(WA E	1) E2 (E3 WB) = +
170			K/СИММ/ WA (E1) E2 (E3) WB .
180		E1 =	'НЕСИММЕТРИЧНОЕ'
190		ФИНИШ	
200			

Рис.6.3

Предложение записывается в 12 - 80-й позициях в метакоде-Б (см. разд.2).

Продолжение предложения на следующую карту обозначается знаком + (плюс) так же, как дефисом обозначается перенос слова в русском языке. Знак + ставится справа в любой из оставшихся позиций перфокарты.

2. Метакод-Б

В первой главе было упомянуто, что при вводе рефал-программ в машину используются метакоды. Здесь описан способ записи рефал-программ, получивший название м е т а к о д - Б.

На рис. 6.1, 6.2 и 6.3 записана в метакоде-Б следующая программа (объектные знаки, имеющие на печати вид правой или левой скобки, изображены жирным шрифтом в отличие от структурных скобок):

k 'ДЕЛО'$\sim k$ 'П' k 'СИММ' k 'СПАР' () k'КАРТА'$\perp\perp\perp\perp$

$\alpha = k$ 'СПАР'

$\alpha(e_1) \sqcup e_2 \sim e_1$

$\alpha(e_1)\mathbf{(}e_2 \sim \alpha((e_1))\, e_2 \perp$

$\alpha((e_1)e_2)\mathbf{)}e_3 \sim \alpha(e_1(e_2))\, e_3 \perp$

$\alpha(e_1)\, s_x\, e_2 \sim \alpha(e_1\, s_x)\, e_2 \perp$

$\beta = k$ 'СИММ'

$\beta \sim$ СИММЕТРИЧНОЕ

$\beta s_a \sim$ СИММЕТРИЧНОЕ

$\beta s_a\, e_1\, s_a \sim \beta\, e_1 \perp$

$\beta(e_1) \sim \beta\, e_1 \perp$

$\beta(t_a e_1)\, e_2 (e_3 t_b) \sim \beta\, t_a (e_1)\, e_2 (e_3)\, t_b \perp$

$\beta\, e_1 \sim$ НЕСИММЕТРИЧНОЕ

Эта программа читает одну перфокарту, на которой набито выражение, ограниченное справа пробелом. Затем функция СПАР превращает объектные знаки "левая скобка" и "правая скобка" в структурные скобки. К полученному выражению применяется предикат СИММ, который проверяет, симметричное выражение или нет. Результат, выданный функцией СИММ, печатается.

Машинные процедуры КАРТА и П будут описаны в разд.5 и 6 этой главы. На машинах ЕС ЭВМ русские буквы отсутствуют, поэтому на рис.6.1 используются английские слова и детерминативы.

Приступаем к описанию метакода-Б.

Свободные переменные изображаются двумя символами: первый – указатель типа переменной (S, W, E), второй – индекс переменной, который может быть цифрой или прописной буквой русского или латинского алфавита. На машинах ЕС ЭВМ имеется только латинский алфавит. В одном предложении у переменных, имеющих одинаковые индексы, указатели типа должны совпадать.

Примеры:

e_5	изображается в виде	E5
s_a	–	SA
t_1	–	W1
t_w	–	WW

Специфицированные переменные в существующих реализациях запрещены!

Скобка записывается одним символом: структурная левая – "(", структурная правая – ")", конкретизационные – "K" и "." . В отличие от скорописи конкретизационную точку опускать нельзя!

Для разделения левой и правой частей предложения вместо тильды используется знак равенства "=".

Цепочки объектных знаков заключаются в апострофы. На БЭСМ-6 и "Минск-32" апостроф – это правая кавычка. При этом все апострофы внутри цепочки представляются парами апострофов. Цепочка, состоящая из одних апострофов, удваивается, но апострофами дополнительно не обрамляется. Левая кавычка (БЭСМ-6 и "Минск-32") является обычным объектным знаком.

Примеры:

ABC	записывается в виде	'ABC'
A'C	–	'A"C'
'	–	''
"	–	""
'A'B	–	"A"B'
A‘B'	–	'A‘B"'

Составные символы обрамляются слешами (косыми чертами).

Например: /АЛЬФА/, /ДУБ1/, /142/.

В теоретическом описании рефала предполагается, что тело составного символа – это любая последовательность объектных знаков. Но при реализации рефала за составными символами закрепляются специальные функции: они могут быть либо метками (детерминативами), либо числами (макроцифрами).

Метка – это последовательность букв и цифр, начинающаяся с буквы, отличной от Ь (мягкий знак). На ЕС ЭВМ запрешены метки вида

Zddddddd , где *d* – цифра.

Длина метки не должна превышать 8 символов для ЕС ЭВМ, 6 символов для БЭСМ-6 и 5 символов для "Минск-32". Всякая метка должна быть описана (см. разд. 3).

Символ-число или макроцифра – это натуральное десятичное число, заключенное в слеши: /142/, /1024/, /0/. Она макроцифра не должна превышать $2^n - 1$, где $n = 31$ для ЕС ЭВМ, $n = 15$ для БЭСМ-6, $n = 36$ для "Минск-32".

Из-за того, что на БЭСМ-6 ограничение на максимальную величину макроцифры гораздо сильнее, чем на машинах ЕС ЭВМ и "Минск-32", в компилятор на БЭСМ-6 включено дополнительное средство, Если вы запишите между слешами число, большее чем $2^{15}-1$ (32767), то компилятор сам разобъет его на несколько макроцифр. В результате получится не один символ, а выражение! Разбиение на макроцифры происходит так. Компилятор делит число на 2^{15} с остатком. Если частное меньше 2^{15}, то формируются две макроцифры: слева частное, справа остаток. В противном случае компилятор делит частное на 2^{15}, полученное частное снова на 2^{15} и т.д. до тех пор, пока очередное частное не окажется нулем. Полученная последовательность остатков записывается в виде макроцифр: справа налево в процессе порождения. Эта последовательность макроцифр является записью исходного числа в 2^{15}-ичной системе счисления. В разд.8 этой главы вы прочтете о машинных процедурах, предназначенных для работы с такими числами.

Запись функций. На рисунках 6.2 и 6.3 изображена программа, состоящая из трех функций: ДЕЛО, СПАР, СИММ. В метакоде-Б знак §, комментарий и знак *k* в левой части предложения не пишутся. Детерминатив (составной символ-метка) указывается только у первого предложения функции. Он записывается с первой позиции (для "Минск-32" с 12-й позиции) без слешей. Метка отделяется от остальной части предложения одним или несколькими пробелами. Второе и следующие предложения функции записывайте без детерминатива, оставив слева по крайней мере один пробел.

Для читабельности программы вы можете пропускать любое количество пробелов между переменными, скобками, составными символами и цепочками объектных знаков. Цепочку объектных знаков можно разбить на несколько цепочек. Но при этом между ними должен стоять по крайней мере один пробел, так как:

'A' ␣ 'B' означает AB

'A' 'B' означает A'B

Если предложение не помещается в одной строке, его можно перенести на одну или несколько строк (см.разд.1). Если вы начали писать переменную, составной символ или цепочку объектных знаков на некоторой строке, то закончить ее нужно на той же строке. Разрывать переменные, символы и цепочки объектных знаков нельзя!

Введено такое ограничение на длину левой части предложения:

пусть *n* – число скобок, объектных знаков, составных символов и вхождений переменной символа в левой части;

m – число вхождений переменных терма и выражения.

Тогда должно выполняться условие

$n + 2m \leqslant 253$

На длину правой части ограничений не накладывается, но предложение в целом должно занимать не более 10 перфокарт.

В программу можно вставлять комментарии. Предложение-комментарий обозначается звездочкой в 1-й позиции (в 12-й позиции для "Минск-32). Предложение-комментарий может занимать только одну строку на бланке, но вы можете написать несколько предложений-комментариев подряд.

При записи программы на бланках ЕС ЭВМ и БЭСМ-6 перечеркивайте цифру ноль: Ø, , а букву О пишите как обычно (если только в вашей организации не принято противоположное соглашение). На машинах "Минск-32" нужно перечеркивать букву О, а цифра ноль изображается обычным кружочком.

В дальнейшем, приводя примеры программ, мы будем перечеркивать ноль.

3. Модули

Часто бывает удобно разбить рефал-программу на части и транслировать их по отдельности. Даже в простейших случаях вы будете использовать некоторые стандартные функции (машинные процедуры), которые кто-то когда-то написал, отладил, оттранслировал, и теперь они хранятся в библиотеке: готовые части вашей программы.

Поэтому результатом компиляции является не готовая программа, а "полуфабрикат", представленный на так называемом языке загрузки. Эти "полуфабрикаты" называются объектными модулями, а рефал-программа, точнее ее часть, подаваемая на вход компилятору, – символьным модулем (рис.6.4).

Рис.6.4

Функции, описанные в разных модулях, могут обращаться друг к другу (рис.6.5).

Метка ПСИ в первом модуле (см.рис.6.5) называется **внешней**, а метка ПСИ во втором модуле – **входной**. Некоторые метки имеют смысл только внутри одного модуля, и к ним нельзя обратиться из другого модуля. В нашем примере такой меткой может быть АЛЬФА. Она называется **внутренней** меткой первого модуля.

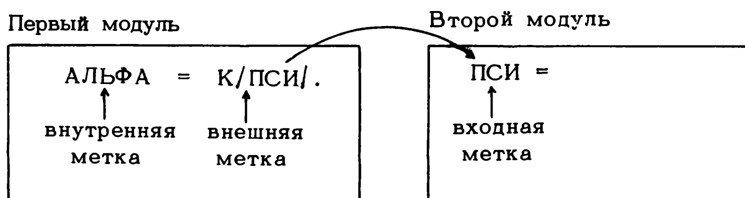


Рис.6.5

Для получения готовой программы (БЭСМ-6) или **загрузочного модуля** (ЕС ЭВМ) необходимо собрать объектные модули вместе и сопоставить входные и внешние метки из разных модулей или, как говорят, **отредактировать связи** (см.рис.6.4).

На "Минск-32" принята такая терминология:

символьный модуль – символьная программа;

объектный модуль – сегмент;

загрузочный модуль – сегмент или рабочая программа;

редакция связей – сборка.

Карты START и END. Символьный рефал-модуль должен начинаться с карты START и кончаться картой END . На этих картах записываются слова START или END соответственно в любом месте во 2–71-й позициях на ЕС ЭВМ и БЭСМ-6 и в 13–79-й позициях на "Минск-32". Кроме того, на ЕС ЭВМ через один или несколько пробелов после слова END указывается метка, являющаяся точкой входа (см.разд.4).

На БЭСМ-6 на карте START с 1-й позиции указывается имя модуля. Имя модуля отделяется от слова START одним или несколькими пробелами. Имя модуля не должно совпасть ни с одной входной точкой рефал-программы.

На БЭСМ-6 и "Минск-32" вместо слов START и END можно использовать слова СТАРТ и ФИНИШ соответственно.

Карты ENTRY и EXTRN . Вы должны указать компилятору, какие метки в вашем модуле являются входными (entry), а какие внешними (external). Для этого нужно записать предложения ENTRY и EXTRN . Эти предложения выглядят так:

ENTRY метка-1, метка-2, ... , метка-n
EXTRN метка-1, метка-2, ... , метка-n

Пропустив несколько позиций слева, вы записываете слово EXTRN или ENTRY, а затем через один или несколько пробелов перечисляете метки через запятую.

В примере на рис. 6.2 и 6.3. метка ДЕЛО является входной, КАРТА и П – внешними, СПАР и СИММ – внутренними.

В программе вы можете записать сколько угодно предложений EXTRN и ENTRY.

На ЕС ЭВМ и БЭСМ-6 предложения EXTRN и ENTRY можно поставить в любом месте между функциями.

На "Минск-32" все предложения EXTRN вы должны разместить сразу же после карты START. Предложения ENTRY записываются в любом месте между функциями.

На БЭСМ-6 и "Минск-32" вместо слов EXTRN и ENTRY можете использовать слова ВНЕШН и ВХОДН соответственно.

На БЭСМ-6 карта EXTRN имеет еще один формат:

метка-1 EXTRN .метка-2
метка-1 ВНЕШН .метка-2

Метка-1 пишется с первой позиции. Справа и слева от слова EXTRN или ВНЕШН ставится хотя бы один пробел.

Метка-1 – это внешняя метка, которую нужно сопоставить с входной меткой-2 в другом модуле. Этот формат используется, если нужно связать различные метки. Например, пусть в вашем модуле используется функция ФИ, которая описана в другом модуле под именем ПСИ. В этом случае в вашем модуле должно стоять предложение:

ФИ EXTRN .ПСИ

или

ФИ ВНЕШН .ПСИ

Карта EQU имеет вид:

метка-1 EQU метка-2

Карта EQU сообщает транслятору, что метка-1 эквивалентна метке-2, т.е. все составные символы вида /метка-1/ транслятор должен воспринимать как составные символы вида /метка-2/.

На ЕС ЭВМ и БЭСМ-6 метка-2 должна быть описана до карты EQU, в которой она используется.

На БЭСМ-6 и "Минск-32" вместо слова EQU можно использовать слово ЭКВ.

Составные символы-метки. Если вам нужен составной символ-метка, не являющийся названием функции, опишите его в виде "пустой" функции: с 1-й (ЕС ЭВМ и БЭСМ-6) или с 12-й ("Минск-32") позиции записывается метка, а в следующих позициях карты остаются пробелы. На рис.6.6 показано, как описать составные символы-метки ALFA и PSI.

Каждый составной символ-метка должен быть описан: либо как название функции, либо в виде пустой функции, либо как внешняя метка, либо картой EQU.

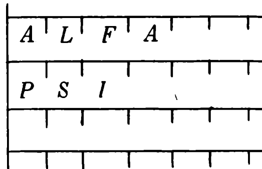


Рис.6.6

Приведем интересный пример использования составных символов-меток, являющихся детерминативами функций:

```
АЛЬФА E1 = K/МЕТАФ/(/A1//A2//A3/ /A4/)E1.
МЕТАФ ( ) E1 = E1 = K/МЕТАФ/(EA)  K SX E1 ..
```

Функция АЛЬФА эквивалентна функции:

```
АЛЬФА E1 = K/A4/K/A3/K/A2/K/A1/ E1 ....
```

В первом предложении функции МЕТАФ в правой части стоит терм

K SX E1.

При конкретизации этого терма должно произойти обращение к функции, детерминативом которой является значение SX . Такая форма записи справедлива для БЭСМ-6 и "Минск-32." На ЕС ЭВМ в такой ситуации нужно обратиться к машинной процедуре MF:

K/MF/SX EI.

которая передаст управление на функцию с детерминативом SX.

4. Начальное поле зрения

Перед началом работы рефал-программы формируется поле зрения вида:

K$\mathscr{D}$.

где $\mathscr{D}$ – детерминатив некоторой функции. Метка $\mathscr{D}$ должна быть описана в одном из модулей как входная.

Как правило, в вашей программе есть несколько входных меток. Рефал-машина должна выбрать одну из них. Эта "проблема выбора" на разных ЭВМ решается по-разному.

На ЕС ЭВМ имя начальной функции указывается на карте END того модуля, который при редактировании связей будет поставлен первым (рис.6.7). Эта входная метка пишется правее слова END через один или несколько пробелов.

На БЭСМ-6 и "Минск-32" входная метка $\mathscr{D}$ указывается при запуске программы на счет. О том, как это делается, вы можете прочесть в главах УШ (БЭСМ-6) и 1Х ("Минск-32").

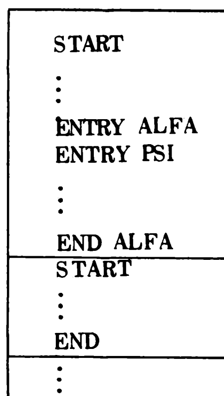


Рис.6.7

5. Машинная процедура КАРТА (CARD)

Данные для программы можно читать с перфокарт с помощью машинной процедуры, которая на БЭСМ-6 и "Минск-32" имеет детерминатив КАРТА, а на ЕС ЭВМ - CARD.

Результатом конкретизации

```
К /КАРТА/.
```

являются 80 объектных знаков очередной карты. При первом обращении читается первая карта с данными, в следующий раз - вторая и т.д.

Когда функция КАРТА (CARD) обнаружит, что карт больше нет, она выдаст "пусто" в качестве результата замены. Но если вы обратитесь к функции КАРТА (CARD) еще раз, то произойдет аварийный останов с соответствующей распечаткой.

Пример. Опишем функцию КАРТЫ, которая читает все карты сразу.

```
КАРТЫ = К/КАРТЫ1/ К/КАРТА/ . .
КАРТЫ1 =
        Е1 = Е1 К/КАРТЫ1/ К/КАРТА/ ..
```

Скобки и другие символы, которые совпадают с собственными знаками рефала, поступают в поле зрения как объектные знаки. Именно поэтому в программе на рис. 6.1-6.3 к прочитанной карте применяется функция, СПАР, которая объектные скобки превращает в структурные.

6. Печать

Для того чтобы отпечатать результаты счета, используйте функцию, которая на БЭСМ-6 и "Минск-32" называется ПЕЧ, а на ЕС ЭВМ - PRINT.

Обращение к ней имеет вид:

К/ПЕЧ/$\mathcal{E}$.

К/PRINT/$\mathcal{E}$.

В результате выполнения этой конкретизации на печатающее устройство с новой строки выводится выражение $\mathcal{E}$. Если выражение не помещается на одной строке, то оно продолжается на следующей.

Объектные и структурные скобки на печати не различаются.

Составные символы печатаются в кавычках.

Результатом замены функции ПЕЧ и PRINT является выражение $\mathcal{E}$.

На БЭСМ-6 и "Минск-32" пользователю предоставляется еще одна машинная процедура – функция П:

К/П/$\mathcal{E}$.

Функция П, так же как и функция ПЕЧ, печатает выражение $\mathcal{E}$, но в отличие от ПЕЧ в поле зрения остается "пусто".

7. Машинные процедуры ЗК, ВК, СЧ, ЗП

В главе 1 дана теоретическая интерпретация функции ЗК, ВК и сказано несколько слов о том, как они реализуются. Теперь мы разберем работу этих функций подробнее.

Названия функций означают: ЗК – закопать, ВК – выкопать, СЧ – считать, ЗП – записать. На ЕС ЭВМ эти функции имеют детерминативы: BR- bury (закопать), DG - dig (выкопать), CP - copy (скопировать), RP - replace (заменить).

Кроме поля зрения, в машине имеется еще одно выражение, которое будем называть **копилкой**. Копилка имеет вид

$$(\mathcal{E}_a\text{ ' = '}\mathcal{E}_1)(\mathcal{E}_b\text{ ' = '}\mathcal{E}_2)\ldots(\mathcal{E}_z\text{ ' = '}\mathcal{E}_n)$$

Смысл каждого терма копилки таков: $\mathcal{E}_a, \mathcal{E}_b, \ldots, \mathcal{E}_z$ – это имена, которые получили значения $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$.

Машинные процедуры ЗК, ВК, СЧ, ЗП (BR, DG, CP, RP) предназначены для пересылки информации из поля зрения в копилку и обратно.

Обращения к этим процедурам имеют вид:

К/ЗК/$\mathcal{E}_n$' = '$\mathcal{E}_0$.	К/BR/$\mathcal{E}_n$' = '$\mathcal{E}_0$.
К/ВК/$\mathcal{E}_n$.	К/DG/$\mathcal{E}_n$.
К/СЧ/$\mathcal{E}_n$.	К/CP/$\mathcal{E}_n$.
К/ЗП/$\mathcal{E}_n$' = '$\mathcal{E}_0$.	К/RP/$\mathcal{E}_n$' = '$\mathcal{E}_0$.

где $\mathcal{E}_0$ – произвольное выражение,

$\mathcal{E}_n$ – произвольное выражение, не содержащее знака = на нулевом уровне скобочной структуры.

При обращении к функции ЗК (BR):

K/ЗK/$\mathcal{E}_n$' = '$\mathcal{E}_0$.

терм ($\mathcal{E}_n$' = '$\mathcal{E}_0$) добавляется к копилке слева, т.е. копилка преобразуется так:

$\mathcal{E}$ → ($\mathcal{E}_n$' = '$\mathcal{E}_0$)$\mathcal{E}$

где $\mathcal{E}$ – содержимое копилки до обращения к функции ЗК.

В результате конкретизации в поле зрения ничего не остается.

Вы можете закопать несколько выражений под одним и тем же именем, Например, при конкретизации

K/ЗK/'X = A'. K/ЗK/'X = B'.

копилка изменится так:

$\mathcal{E}$ → ('X = B') ('X = A')$\mathcal{E}$

Функция ВК (DG):

K/BK/$\mathcal{E}_n$.

просматривает копилку слева направо в поисках терма вида ($\mathcal{E}_n$'='$\mathcal{E}_0$) и, если находит, удаляет его из копилки и выдает $\mathcal{E}_0$ в качестве результата замены, т.е. происходит такое преобразование:

поле зрения: K/BK/ $\mathcal{E}_n$. → $\mathcal{E}_0$

копилка: $\mathcal{E}'$ ($\mathcal{E}_n$' = '$\mathcal{E}_0$) $\mathcal{E}''$ → $\mathcal{E}'$ $\mathcal{E}''$

Если в копилке несколько выражений закопаны под одним именем, то выкапывается самое левое, т.е. закопанное в последний раз. Если вы повторно обратитесь к функции ВК (DG) с тем же $\mathcal{E}_n$, то получите выражение, закопанное в предпоследний раз, и т.д.

Пример. После конкретизации рабочего выражения

K/ЗK/'X = A'. K/ЗK/'X = B'. K/BK/'X'. K/BK/'X'.

в поле зрения останется 'BA'.

Если функция ВК (DG) не найдет в копилке нужного терма, то она выдает "пусто". (Обратите внимание, что в 1 главе функция ВК была определена иначе: в этом случае происходил авост "отождествление невозможно". На практике оказалось, что объявлять авост не всегда удобно). Таким образом, если копилка имеет вид

('X = A') ('X = B')

то поле зрения

(K/BK/'X'.) (K/BK/'X'.) (K/BK/'X'.)

через 3 шага превратится в

('A') ('B') ( )

а в копилке ничего не останется.

Функция СЧ (СР) так же, как и функция ВК (DG), находит в копилке выражение по имени и выдает его в качестве результата замены, но копилка при этом не изменяется, так как в поле зрения формируется копия выражения, т.е. функция СЧ (СР) работает так:

поле зрения: К/СЧ/ $\mathcal{E}_n$. → $\mathcal{E}_0$

копилка: $\mathcal{E}'(\mathcal{E}_n$ '=' $\mathcal{E}_1)\mathcal{E}''$ → $\mathcal{E}'(\mathcal{E}_n$ '=' $\mathcal{E}_0)\mathcal{E}''$

Функция ЗП (RP) добавляет в копилку новое выражение и выбрасывает выражение, закопанное в последний раз под тем же именем.

поле зрения: К/ЗП/ $\mathcal{E}_n'$ =' $\mathcal{E}_0$. → "пусто"

копилка: $\mathcal{E}'$ ($\mathcal{E}_n$ '=' $\mathcal{E}_1$) $\mathcal{E}''$ → ($\mathcal{E}_n$ '=' $\mathcal{E}_0)\mathcal{E}'\mathcal{E}''$

Эквивалентное описание на рефале имеет вид:

ЗП ЕИ'='Е1 = К/ЗП1/К/ВК/ЕИ.. К/ЗК/ЕИ'=' Е1.

ЗП1 Е1 =

Если у вас возникнет потребность проанализировать содержимое всей копилки (например, для того, чтобы "почистить" ее), пользуйтесь машинной процедурой ВКВСЕ (БЭСМ-6 и "Минск-32") или DGALL (ЕС ЭВМ). Обращение к функции ВКВСЕ (DGALL) имеет вид:

К/ВКВСЕ/. или К/DGALL/.

В результате конкретизации содержимое копилки передается в поле зрения. Например, если вначале копилка пуста, то рабочее выражение

К/ЗК/'Х=А'.К/ЗК/'У=В'.К/ВКВСЕ/.

через 3 шага превратится в

('У=В')('Х=А')

После работы функции ВКВСЕ (DGALL) в копилке ничего не остается.

Вы можете восстановить копилку, воспользовавшись функцией ЗКВСЕ, которая не является машинной процедурой, и поэтому вы должны описать ее на рефале, например, так:

ЗКВСЕ $\underline{\underline{\text{E}}}$1(Е2) = К/ЗК/Е2. К/ЗКВСЕ/Е1.

8. Машинные процедуры арифметики

Машинные процедуры ADD, SUB, MUL, DR (сложение, вычитание, умножение, деление с остатком) предназначены для работы с целыми числами, записанными с помощью макроцифр (см.разд.2 этой главы).

Целое число – это непустая последовательность макроцифр, перед которой может стоять объектный знак + или –. Если знак отсутствует, то подразумевается +.

Примеры целых чисел:

'+' /17/ /242/ /15Ø3/ /5/

/17/ /242/ /15Ø3/ /5/

'–' /2/

/Ø/

'–' /3/ /2Ø35/

Такое представление числа является записью в 2^n -ичной системе счисления (ЕС ЭВМ: n =31, БЭСМ-6: n =15, "Минск-32": n =36). Таким образом, на БЭСМ-6 три символа

'-' /3/ /2Ø35/

изображают число

$$-(3 \cdot 2^{15} + 2035 \cdot 2^0) = -100339.$$

Обращение к функциям ADD, SUB, MUL, DR имееи вид

$$K\mathscr{D}(\mathscr{N}_1)\mathscr{N}_2.$$

где $\mathscr{D}$ -детерминатив (/ADD/, /SUB/, /MUL/, /DR/)

$\mathscr{N}_1, \mathscr{N}_2$ – целые числа.

Результатом конкретизации функций ADD, SUB, MUL является целое число. Если результат положителен, знак + не ставится. Левые незначащие нули опускаются. Нулевой результат выдается в виде одной макроцифры /Ø/.

Пример. Функция, вычисляющая факториал неотрицательного числа:

```
FACT /Ø/ = /1/
        EX = K/MUL/(EX) K/FACT/K/SUB/(EX) /1/...
```

Функция DR выдает результат в виде

$$\mathscr{Q}(\mathscr{R})$$

где $\mathscr{Q}$ – частное, $\mathscr{R}$ – остаток.

И частное, и остаток выдаются без левых незначащих нулей и без знака +.

Деление на ноль приводит к авосту "отождествление невозможно".

Деление с остатком производится так: делятся два числа, не обращая внимание на знаки, а затем частному и остатку приписываются такие знаки, чтобы соблюдалась формула:

$$\mathscr{N}_1 = \mathscr{Q} \times \mathscr{N}_2 + \mathscr{R}$$

т.е. частное положительно, если знаки делимого и делителя совпадают, и отрицательно – в противном случае, а не равный нулю остаток всегда имеет знак делимого.

Примеры на деление.

```
K/DR /(/5/) /3/.  → /1/(/2/)
K/DR /(/5/) '-' /3/.  → '-'/1/(/2/)
K/DR /('-' /5/) /3/.  → '-' /1/('-' /2/)
K/DR /('-' /5/) '-' /3/.  → /1/('-' /2/)
```

Пример. Функция, вычисляющая наибольший общий делитель двух чисел по алгоритму Евклида.

Вид обращения: К/НОД/ ($\mathcal{N}_1$) ($\mathcal{N}_2$).

Программа на рефале.

```
НОД (E1) EA(/Ø/) = E1
        (E1)EA(E2) = K/НОД/(E2) K/DR/(E1) E2..
```

Все исходные данные поступают в поле зрения в виде цепочек объектных знаков (см.разд.5). Макроцифры нельзя ввести в поле зрения. Поэтому вы должны переводить исходные данные из десятичной в 2^n -ичную систему. 2^n -ичную систему будем условно называть двоичной.

Нам понадобится функция, которая превращает десятичную цифру (объектный знак) в макроцифру:

```
NUMB  'Ø' = /Ø/
      '1' = /1/
      '2' = /2/
       ⋮
      '9' = /9/
```

Для эффективности эта функция написана в виде машинной процедуры.

Используя машинную процедуру NUMB, вы должны описать в своей программе функцию CVB (convert to bynary – перевести в двоичную) и пользоваться ею при вводе исходных данных:

```
CVB SA = K/NUMB/SA.
    E1 SA = K/ADD/(K/NUMB/SA.) +
                 K/MUL/(/10/) K/CVB/E1 ...
```

Пример. (БЭСМ-6, 2^{15}=32768-ичная система). В результате конкретизации

```
K/CVB/'100000'.
```

получится

```
/3/ /1696/
```

Результаты счета нужно печатать в десятичной системе. Поэтому вам понадобится функция CVD (convert to decimal – перевести в десятичную). Функция CVD использует машинную процедуру SYMB, которая превращает макроцифру, меньшую /1Ø/, в объектный знак. Функцию SYMB можно изобразить на рефале так:

```
SYMB /Ø/= 'Ø'
     /1/= '1'
      ⋮
     /8/= '8'
     /9/= '9'
```

Функция CVD описывается так:

```
CVD   E1 = K/CVD1/K/DR/(E1)/10/..
CVD1  /∅/(SX) = K/SYMB/SX.
      E1 (SX) = K/CVD1/K/DR/(E1)/1∅/.. +
                K/SYMB/SX.
```

Пример. Если на БЭСМ-6 вы обратитесь к функции CVD так:

K/CVD//1//∅/.

то в результате получите '32768'.

9. Машинные процедуры для лексического анализа

Функция TYPE ("тип"). Обращение к функции TYPE имеет вид

K/TYPE/ℰ.

Функция TYPE анализирует первое звено (символ или скобку) выражения ℰ1 и выдает

𝒞ℰ

где 𝒞 – объектный знак, который определяется по следующей таблице:

𝒞	Первое звено ℰ	
'F'	составной символ-метка	(function)
'N'	составной символ-число	(number)
'L'	объектный знак-буква	(letter)
'D'	объектный знак-цифра	(digit)
'O'	прочий объектный знак	(other)
'B'	левая структурная скобка	(bracket)
'*'	ℰ1 пустое	

Пример. Опишем функцию ИДЕН, которая проверяет, является ли поданное на нее выражение идентификатором или нет (идентификатор – последовательность букв и цифр, начинающаяся с буквы). В результате конкретизации

K/ИДЕН/ℰ.

должно получиться (ℰ), если ℰ – идентификатор и *ℰ в противном случае.

Описание функций имеет вид:

```
* ПРОВЕРЯЕМ ПЕРВЫЙ СИМВОЛ
ИДЕН  E1 = K/ИДЕН1/ K/TYPE/ E1..
* БУКВА ЛИ ПЕРВЫЙ СИМВОЛ
ИДЕН1  'L' SA E1 = K/ИДЕН2/(SA)    K/TYPE/E1..
       SX E1 = '*' E1
* БУКВЫ И ЦИФРЫ ЛИ ОСТАЛЬНЫЕ СИМВОЛЫ
ИДЕН2 (EИ)  'L' SA E1 =  K/ИДЕН2/(EИ SA) K/TYPE/ E1 ..
      (EИ) 'D' SA E1 = K/ИДЕН2/(EИ SA) K/TYPE/E1..
      (EИ) '*' = (EИ)
      (EИ) SX E1 = '*' EИ E1
```

Функции FIRST и LAST предназначены для "отщепления" от выражения части фиксированной длины. К этим функциям нужно обращаться так:

K/FIRST/$\mathcal{N}\mathcal{E}$.

K/LAST/$\mathcal{N}\mathcal{E}$.

где $\mathcal{N}$ – макроцифра,

$\mathcal{E}$ – произвольное выражение.

Эти функции делят выражение на 2 части, т.е. представляют $\mathcal{E}$ в виде $\mathcal{E}_1\mathcal{E}_2$ с соблюдением условия:

для FIRST – $\mathcal{E}_1$ состоит из $\mathcal{N}$ термов;

для LAST – $\mathcal{E}_2$ состоит из $\mathcal{N}$ термов.

Если разделить $\mathcal{E}$ можно (т.е. $\mathcal{E}$ состоит не менее чем из $\mathcal{N}$ термов), то результат конкретизации имеет вид:

$(\mathcal{E}_1)\,\mathcal{E}_2$ для FIRST

$\mathcal{E}_1\,(\mathcal{E}_2)$ для LAST

Если количество термов в $\mathcal{E}$ меньше $\mathcal{N}$, то выдается

функцией FIRST – '*' $\mathcal{E}$

функцией LAST – $\mathcal{E}$ '*'

Пример. Опишем функцию ЧКАРТЫ (часть карты), которая читает очередную карту с помощью машинной процедуры КАРТА и выдает первые 72 символа карты. При этом последние 8 символов карты выбрасываются.

```
ВНЕШН КАРТА
ЧКАРТЫ = K/ЧКАР1/ K/LAST/ /8/ K/KАРТА/...
ЧКАР1  E1(E2) = E1
       '*' =
```

Напоминаем, что, когда во входном потоке не останется карт, функция КАРТА выдает "пусто". Именно эта ситуация распознается вторым предложением функции ЧКАР1.

Функции LENGW и LENGR. В результате конкретизаций

K/LENGW/ $\mathcal{E}$.

K/LENGR/ $\mathcal{E}$.

выдается

$\mathcal{N}\mathcal{E}$

где $\mathcal{N}$ – макроцифра, означающая:

для LENGW – длину выражения $\mathcal{E}$ в термах (т.е. количество термов в $\mathcal{E}$);

для LENGR – реальную длину выражения $\mathcal{E}$ (т.е. количество символов и скобок, составляющих выражение $\mathcal{E}$).

После описания функции MULTE мы приведем пример, в котором используется функция LENGR.

Функция MULTE размножает выражение в несколько раз. Обращение к функции MULTE имеет вид:

K/MULTE/ $\mathcal{N}\mathcal{E}$.

где $\mathcal{N}$ – макроцифра;

$\mathcal{E}$ – произвольное выражение.

Результатом замены является

$$\underbrace{\mathcal{E}\,\mathcal{E}\,\mathcal{E}\ldots\mathcal{E}}_{\mathcal{N}}$$

В частности, $\mathcal{N}$ может быть равно /Ø/, тогда выдается "пусто".

Пример. Опишем функцию ДП ("двойная печать"), которая печатает результаты счета в двух экземплярах: первый – с 1-й позиции, второй – с 66-й позиции. (Предполагается, что печатающее устройство имеет 128 позиций).

Например, в результате конкретизации

K/ДП/ 'СЕРЕНЬКИЙ'. K/ДП/ 'КОЗЛИК'.

отпечатаются 2 строки:

```
СЕРЕНЬКИЙ                          СЕРЕНЬКИЙ
КОЗЛИК                             КОЗЛИК
```

На функцию ДП наложим такое ограничение: печатаемое выражение не должно содержать составных символов и должно состоять не более, чем из 63 объектных знаков и скобок.

```
        ВНЕШН   П, LENGR, SUB
   ДП   E1=K/ДП1/K/LENGR/E1..
   ДП1  SN E1 = K/П/ E1 K/MULTE/ +
                    K/SUB/(/63/) SN . '␣'.'␣␣' E1.
```

10. Прокрутка

Для отладки рефал-программы вы можете организовать пошаговую печать или, как говорят, прокрутку. Прокрутка дает возможность на каждом шаге получить такую информацию:

1) номер шага;

2) ведущий терм (конкретизируемое выражение);

3) результат замены.

Вы можете заказать распечатку всех шагов работы рефал-программы, а также задать условия, которые запретят печать некоторых шагов. Здесь мы расскажем об этих условиях. О том, как заказать прокрутку и описать условия на каждой конкретной машине, вы прочтете в УII – 1Х главах.

Реализовано 4 вида условий.

1. Условие "по шагам". Вы можете указать номера начального и конечного шагов прокрутки. Условие "по шагам" запрещает печать шагов вне указанного диапазона. Будет или нет печать шагов в диапазоне от начального шага до конечного, зависит от остальных условий прокрутки. Если других условий нет, то печатаются все шаги в этом диапазоне.

2. Условие ">". Задавая это условие, вы должны указать один или несколько детерминативов. Условие ">" блокирует прокрутку до тех пор, пока не произойдет обращение к одной из указанных функций, т.е. пока не станет ведущим функциональный терм с одним из указанных детерминативов. В этот момент запоминается количество знаков К в поле зрения. Прокрутка ведется до тех пор, пока выполняется следующее условие: текущее количество знаков *k* больше запомненного или равно ему.

После этого прокрутка блокируется и возобновится лишь при следующем обращении к одной из указанных функций, и т.д.

Условие ">" применяется, когда вы интересуетесь работой только некоторых функций. Например, если в отлаженной программе вы изменили какую-то функцию и хотите посмотреть, не сделали ли при этом ошибку, то, задав условие ">" с детерминативом этой функции, вы получите на печати только ход вычисления этой функции: от каждого обращения до окончания вычисления значения функции.

3. Условие "<". Задавая это условие, вы должны указать один или несколько детерминативов. Условие "<" блокирует прокрутку при обращении к одной из указанных функций, т.е. когда станет ведущим функциональный терм с одним из этих детерминативов. В этот момент запоминается количество знаков *k* в поле зрения. Прокрутка возобновится при выполнении следующего условия: текущее количество знаков *k* меньше запомненного.

После этого печать шагов производится либо до нового обращения к одной из указанных функций, либо до тех пор, пока другие условия не запретят ее.

Условие "<" применяется, когда некоторые функции уже отлажены. Указав детерминативы в условии "<", вы запретите прокрутку хода вычисления этих функций.

При каждом обращении к одной из этих функций печатается:

1) номер шага, на котором произошло обращение к функции;

2) ведущий терм на этом шаге;

3) номер шага, когда закончилось вычисление значения функции;

4) полученное значение функции.

4. Условие "по детерминативам". Задавая это условие, вы должны указать один или несколько детерминативов. При этом на печать выводятся только те шаги, когда ведущий терм имеет детерминатив, который указан в условии "по детерминативам" или в условиях "> " и "< ". (Но и в этом случае при прокрутке шагов с детерминативами из условия "<" печатается значение функции, а не результат замены).

На ЕС ЭВМ в условиях " >" и " < " можно указать только один детерминатив. Разрешается задавать только одно из условий " > ", " < ", "по детерминативам". Вместе их задавать нельзя. К одному из этих условий можно добавить только условие "по шагам".

На БЭСМ-6 и "Минск-32" вы можете задавать условия прокрутки в любых комбинациях. При этом нужно руководствоваться следующим правилом.

Каждое из условий запрещает печать некоторых шагов:

1) "по шагам" – запрещает прокрутку вне указанного диапазона шагов;

2) " > " – запрещает прокрутку до обращения к некоторой функции и после окончания вычисления ее значения;

3) "< " – запрещает прокрутку хода вычисления указанных функций;

4) "по детерминативам" – запрещает печать шагов, кроме шагов с указанными детерминативами.

В результате действия нескольких условий печатаются только те шаги, которые не запрещены ни одним из условий.

УП. ЗАПУСК ПРОГРАММ НА МАШИНАХ СЕРИИ ЕС ЭВМ

Рефал реализован для машин ЕС ЭВМ в рамках операционной системы ОС. Для того чтобы можно было работать с рефалом, необходимо включить в системную библиотеку SYS1. LINKLIB загрузочные модули REFAL и RFC, включить в SYS1. PROCLIB каталогизированные процедуры REFAL и REFALG, а также организовать и каталогизировать библиотеку объектных модулей SYS1. REFAL и переписать в нее стандартные машинные процедуры.

Существующая реализация рефала рассчитана на модели машин с оперативной памятью до 512 килобайтов. Если вы работаете с машиной, у которой больше памяти, вы должны позаботиться о том, чтобы ваша программа работала в области памяти, где адреса не превышают 2^{19}, т.е. в первых 512 килобайтах. (В операционной системе есть средства, которые позволяют следить за этим). В будущем предусмотрено создание новой реализации рефала, где это ограничение будет снято.

Подразумевается, что пользователь сначала ознакомился с главой У1 и знает, как записать программу в метакоде-Б и как ее оформить. Кроме того, чтобы продуктивно работать на машине, необходимо хотя бы в общих чертах представлять структуру наборов данных в ОС ЕС ЭВМ, а так-

же ознакомиться с общими понятиями языка управления заданием (JOB CONTROL LANGUGE).

Мы приведем примеры запуска рефал-программ на счет и примеры типичных пакетов для компиляции и последующего счета рефал-программ.

1. Общая схема работы с рефал-программой

Интерпретатор языка сборки (ИЯС) выделен в отдельный загрузочный модуль, имя которого REFAL. Для счета любой рефал-программы надо вызвать в оперативную память модуль REFAL и передать ему информацию о том, какую программу он должен выполнять. REFAL отыщет указанную программу, загрузит ее в память и начнет выполнять.

Последовательность изготовления работоспособной программы такова:

- записанная в метакоде-Б программа подается на вход рефал-компилятора. Он распечатывает программу, производит синтаксический контроль и, если синтаксических ошибок нет, начинает компиляцию. Результатом компиляции будет символьный модуль на языке ассемблера;
- запускается ассемблер. Результат трансляции есть объектный модуль;
- получившийся объектный модуль и машинные процедуры объединяются редактором связей в единый загрузочный модуль.

Запуская на счет любую рефал-программу, вы кладете управляющую карту

```
// EXEC PGM=REFAL
```

а имя той рефал-программы, которую надо выполнить, указываете во входной информации для программы REFAL.

Приведем пример запуска на счет рефал-программы PROG, которая находится в библиотеке REFAL.LOAD на диске типа 2311, серийный номер диска AUTOL.

```
//JOBREF JOB 12,PETROV
//        EXEC PGM=REFAL
//STEPLIB DD DSN=REFAL.LOAD, DISP=SHR,
//              UNIT=2311, VOL=SER=AUTOL
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
 PROG
 /*
```

Напомним на этом примере общие положения языка управления заданием (JOB CONTROL LANGUAGE).

Все управляющие карты имеют в 1-й и 2-й позициях два слеша (слеш - наклонная черта). Начиная с третьей позиции, идет имя управляющей карты (если оно есть), затем через один или несколько пробелов код операции, затем через один или несколько пробелов - операнды. Если все необходимые операнды не помещаются на одной перфокарте, организуются карты продолжения. Для этого после операнда ставите запятую, а на следующей карте описываете оставшиеся операнды, начиная не ранее чем с 4-й позиции перфокарты (см.в примере описание управляющей карты с именем STEPLIB).

Любое задание начинается картой с кодом операции JOB (работа). В поле имени этой карты указывается имя задания при прохождении его в машине. В операндах может указываться учетный номер, фамилия и другая информация.

Задание может состоять из нескольких шагов. Шаг - это предписание операционной системе выполнить определенную программу. Шаг описывается несколькими картами.

Первой картой должна быть карта с кодом операции EXEC (execute - выполнить). Операндом служит имя выполняемой программы (в нашем примере REFAL), могут быть и другие операнды; например: параметры, которые надо передать программе, код условия. Операнды - ключевые, т.е. записываются в виде:

<ключевое слово> = <значение операнда>

В поле имени карты EXEC можно поставить произвольное имя. Оно может понадобиться, если вы захотите сослаться на этот шаг из других шагов задания. Если ссылок не предусмотрено, имя можно опустить.

Еще один тип управляющих карт - карты с кодом операции DD, так называемые DD-предложения. Эти карты служат для окончательного описания наборов данных (т.е. входной и выходной информации), с которыми имеет дело выполняемая программа. Частично каждый набор данных, с которым работает программа, описан в самой программе. Там же есть ссылка на имя DD-предложения, в котором будет дана дополнительная исчерпывающая информация об этом наборе данных. Поэтому при описании DD-предложения надо указывать именно то имя, на которое ссылается программа. В данном примере программа REFAL ссылается на два DD-предложения с именами SYSPRINT и SYSIN. DD-предложение

```
//SYSPRINT DD SYSOUT = A
```

означает, что набор данных SYSPRINT надо распечатать. DD-предложение

```
//SYSIN DD *
```

означает, что набор данных SYSIN идет вслед за этой картой во входном потоке.

Наборы данных SYSIN и SYSPRINT будут подробнее описаны в разд.2.

Не только рабочие программы (как REFAL), но и системные программы иногда требуют описания DD-предложений. В частности, в нашем примере DD-предложение с именем STEPLIB указывает операционной системе, в какой библиотеке отыскивать модуль PROG. Если DD-предложение STEPLIB не описано, операционная система будет отыскивать модуль PROG в системной библиотеке SYS 1.LINKLIB .

На примере описания набора данных STEPLIB скажем коротко об операндах DD -предложения.

Все операнды DD -предложения ключевые.

Если информация находится на магнитных носителях, как правило, требуется указывать имя набора данных DSN (data set name).

В нашем примере это имя библиотеки REFAL.LOAD.

DISP (диспозиция) служит для того, чтобы сообщить системе: существует ли уже набор данных или его надо организовать, что делать с набором по окончании работы программы (сохранить, или уничтожить, или передать следующим шагам задания). В нашем примере DISP = SHR означает, что набор данных STEPLIB уже существует, его надо сохранить и доступ к нему одновременно разрешен из других программ.

Если набор данных не зарегистрирован в системном каталоге, необходимо сообщить системе, где он находится. Для этого служат параметры UNIT (тип устройства) и VOL = SER = . . . – серийный номер тома.

Приведем теперь другой пример.

Рефал-компилятор сам написан на рефале: поэтому, чтобы запустить его, необходимо действовать так же, как при запуске любой рефал-программы. То есть подложить карту

```
//EXEC PGM = REFAL
```

и указать во входной информации имя рефал-компилятора RFC.

```
//COMPILE JOB 21 , IVANOV, MSGLEVEL = 1
//          EXEC PGM = REFAL , PARM = NOPRINT
//SYSPRINT DD SYSOUT = A
//LIB DD DSN =&LIB,SPACE = (4ØØ,(5ØØ,5Ø)) ,
//          DISP = (NEW,PASS),UNIT = SYSDA
//SYSIN DD *
RFC
          START
           ....
           END
/*
```

Получившийся в результате компиляции символьный модуль пишем во временный набор данных &LIB, чтобы затем передать его на ассемблирование.

О значении параметра NOPRINT будет сказано в разд.6. В описанном случае нет необходимости подкладывать карты с описанием STEPLIB, так как модуль RFC (рефал-компилятор) находится в системной библиотеке SYS1.LINKLIB.

2. Наборы данных и DD-предложения

Программа REFAL имеет дело со следующими наборами данных.

1. Имя DD-предложения SYSPRINT. Это последовательный набор данных, REFAL использует его для распечатки своих сообщений для прокрутки. Этот набор используется также машинной процедурой PRINT. Обычно подкладывается DD-предложение вида:

//SYSPRINT DD SYSOUT = A

Параметры DCB для этого набора: LRECL = 120, BLKSIZE = 120, RECFM = F. То есть печатается по 120 символов в строке.

2. Имя DD-предложения SYSIN. Это последовательный набор данных. Он является входным потоком для программы REFAL. Подкладывается DD-предложение вида:

//SYSIN DD *

Параметры DCB для этого набора: LRECL = 8Ø, BLKSIZE = 8Ø, RECFM = F.

Если не используется прокрутка (см.разд.5), то первой картой входного потока должна быть карта с именем того модуля (рефал-программы), который надо загрузить в память для выполнения. Имя модуля набивается с первой позиции перфокарты. Если запускается компиляция программы, то первой картой входого потока должна быть карта с именем компилятора (RFC).

После карты с именем выполняемой программы должна идти входная информация для машинной процедуры CARD. При компиляции - это рефал-программа в метакоде-Б, которую надо скомпилировать.

Наборы данных SYSIN и SYSPRINT описаны в модуле REFAL, поэтому соответствующие DD-предложения необходимо описывать при счете любых рефал-программ.

3. Рефал-программа может работать с собственным набором данных, не описанным в REFAL. Так, рефал-компилятор RFC создает выходной набор данных - результат компиляции. Имя DD-предложения LIB. Набор данных LIB - это последовательный набор данных. Он представляет собой ассемблерный символьный модуль. Обычно этот набор данных передается непосредственно на ассемблирование.

Параметры DCB для этого набора: LRECL= 80, BLKSIZE =400, RECFM = FB.

Чтобы рефал-программа могла совершать обмен с внешними устройствами, написаны специальные машинные процедуры LIBGET и LIBPUT. Эти процедуры и соответствующие им DD-предложения будут описаны в разд.4.

3. Каталогизированные процедуры REFAL и REFALG

Для удобства пользователя написаны каталогизированные процедуры REFAL и REFALG.

Процедура REFAL состоит из трех шагов. Шаг REF - работа рефал-компилятора, шаг ASM - ассемблирование получившегося символьного модуля, шаг LKED - редактирование связей. Эта процедура аналогична процедуре ASMFCL, только добавлен первый шаг - компиляция.

Приведем типичный пример запуска процедуры REFAL:

```
//STEP 1 EXEC REFAL
//REF.SYSIN DD *
RFC
          START
            .
            .
            .
          END
//LKED.SYSIN DD *
          INCLUDE REFLIB (RINOUT, ARITHM)
//LKED. SYSLMOD DD DSN = REFAL.LOAD (REFTEST),
//        DISP = OLD, UNIT = 2311, VOL = SER = AUTOL
```

Обращаем ваше внимание на то, что имя процедуры REFAL совпадает с именем модуля REFAL. Однако, когда запускаем программу, надо писать:

```
//        EXEC PGM = REFAL
```

а при запуске процедуры REFAL:

```
//        EXEC REFAL        или

//        EXEC PROC = REFAL
```

При использовании каталогизированных процедур в описании DD-предложения сначала указывается имя шага, затем через точку - имя DD-предложения, например:

```
//REF.SYSIN DD *
//LKED.SYSLMOD DD ....
```

Входной набор SYSIN на шаге редакции связей необходим для подключения машинных процедур (см.разд.4).

Выходной набор редактора SYSLMOD служит для записи готового загрузочного модуля в библиотеку. В этом примере он записывается под именем REFTEST в библиотеку REFAL.LOAD, которая находится на диске типа 2311, серийный номер AUTOL.

Процедура REFALG. В этой процедуре по сравнению с процедурой REFAL добавлен шаг GO – счет вашей программы. Для шага GO также необходимо описать входной набор данных SYSIN. Первой картой этого набора (если нет прокрутки) должна быть карта со стандартным именем GO (имя модуля, передаваемого редактором на счет).

Приведем пример запуска процедуры REFALG. Пользователь применяет стандартные машинные процедуры CARD и PRINT, которые находятся в объектном модуле RINOUT

```
//        EXEC REFALG
//REF.SYSIN DD *
RFC
        START
        .
        :            программа в метакоде – Б
        .
        END
//LKED.SYSIN DD *
INCLUDE REFLIB (RINOUT)
//GO.SYSIN DD *
GO
...             входная информация для CARD
/*
```

DD-предложение с именем REFLIB включено в процедуры REFAL и REFALG. В этом DD-предложении описана библиотека стандартных машинных процедур SYS1.REFAL. (Подробнее об этом см. в следующем разделе.)

4. Машинные процедуры

Кроме описанных в главе 6 машинных процедур, есть еще две стандартные машинные процедуры для чтения и записи информации.

Процедура LIBGET. Эта процедура работает аналогично процедуре CARD. Однако LIBGET берет записи не из входного потока SYSIN, а как правило, читает раздел из библиотеки. Имя DD-предложения для этого набора данных SYSU1. При описании этого набора данных надо указывать параметры DCB: RECFM и BLKSIZE. Длина логической записи определена: LRECL=80. LIBGET работает с последовательно организованным

набором данных, поэтому при чтении из библиотечного набора надо указывать в скобках имя раздела. Например:

```
//SYSUT 1 DD  DSN = LIB.SYM (TEST 1), DISP = SHR,
//          DCB = (RECFM = FB, BLKSIZE = 800),
//          UNIT = 2311, VOL = SER = AUTOL
```

Набор данных SYSUT 1 можно организовать на перфокартах:

```
//SYSUT 1 DD*
```

Формат обращения K /LIBGET/.

Результатом конкретизации K /LIBGET/. (так же, как и K/CARD/.) будет 80 объектных знаков. Обнаружив конец файла, процедура LIBGET выдаст пустой результат. При повторном обращении к кончившемуся файлу результат непредсказуем.

Процедура LIBPUT. Образует выходной набор данных, который можно записать на магнитный носитель или распечатать. Имя DD -предложения для этого набора данных SYSUT 2. Предполагается, что этот набор, вообще говоря, выводится в символьную библиотеку, поэтому длина логической записи равна 80 (LRECL = 80). Параметры DCD: RECFM и BLKSIZE надо указывать. Набор данных SYSUT2 - последовательный набор. Приведем примеры описания SYSUT 2:

а) пишем модуль в символьную библиотеку LIB.SYM под именем RF:

```
//SYSUT 2  DD  DSN = LIB.SYM (RF), DISP = OLD,
//             DCB = (RECFM = FB, BLKSIZE = 800),
//             UNIT = 2311, VOL = SER = AUTOL
```

б) распечатываем набор данных SYSUT 2

```
//SYSUT 2 DD  SYSOUT = A, DCB = (BLKSIZE = 80, RECFM = F)
```

Формат обращения K /LIBPUT /E1., где E1 - выражение, состоящее из объектных знаков и структурных скобок. Структурные левые и правые скобки LIBPUT превратит в объектные знаки, которые имеют на печати вид соответственно левой и правой круглой скобки: в таком виде они и запишутся в выходной файл. Вначале процедура LIBPUT отделяет первые 80 объектных знаков и структурных скобок и заносит их как очередную запись в выходной набор данных, затем следующие 80 (если они есть) и так далее. При необходимости она дополнит последнюю запись справа пробелами до 80. После этого LIBPUT уберет из поля зрения свой аргумент и прекратит работу. То есть результат конретизации K /LIBPUT /E1. - "пусто".

Чтобы подключить машинные процедуры к программе, надо на шаге редакции связей указать имена объектных модулей, в которых хранятся нужные процедуры.

Библиотека стандартных машинных процедур описана в каталогизированных процедурах REFAL и REFALG в DD-предложении с

именем REFLIB. Поэтому при запуске каталогизированных процедур это DD-предложение можно не описывать. Если вы храните какие-то машинные процедуры (или части вашей программы) в другой библиотеке, необходимо ее специально описывать. Например:

```
//      EXEC REFALG
//REF.SYSIN DD *
RFC
        START
        :          программа на рефале в метакоде - Б
        END
//LKED.SYSIN DD *
        INCLUDE REFLIB (RINOUT, LINOUT)
        INCLUDE MYLIB (PART 1, PART 2)
//LKED.MYLIB DD DSN = LIB.OBJ, DISP = SHR,
//        UNIT = 2311, VOL = SER = 231103
//GO. SYSIN DD *
GO
        ...  входная информация для процедуры  CARD

//GO. SYSUT 1 DD DSN = LIB. SYM (MEMBER 1), DISP = SHR,
//        DCB = (RECFM = FB, BLKSIZE = 800),
//        UNIT = 2311, VOL = SER = AUTOL
```

В приведенном примере используются машинные процедуры CARD, PRINT и LIBGET. На шаге редакции подключаются модули RINOUT и LINOUT (в которых находятся эти процедуры) и скомпилированные ранее части вашей программы PART 1 и PART 2, которые хранились в библиотеке LIB.OBJ.

В этом примере используется машинная процедура LIBGET, поэтому понадобилось на шаге GO описать DD-предложение SYSUT 1.

Приведем список стандартных машинных процедур с указанием объектных модулей, в которых они находятся:

CARD, PRINT, PROUT	хранятся в	модуле	RINOUT
BR, DG, CP, RP, DGALL	>>	>>	BURING
ADD, SUB, MUL, DR, NUMB, SYMB	>>	>>	ARITHM
TYPE, FIRST, LAST, LENGW, LENGR, MULTE	>>	>>	SYNTAX
LIBGET, LIBPUT	>>	>>	LINOUT
MF	>>	>>	MF

5. Прокрутка

Прокрутка выводится в тот же набор данных, что и сообщения программы REFAL и результат работы машинной процедуры PRINT, т.е. в набор данных с именем DD -предложения SYSPRINT.

Прокручивая очередной шаг, REFAL печатает номер шага, затем конкретизируемое выражение, затем - результат замены. Распечатка производится в метакоде-Б, ограничивается по краям тройками звездочек.

Управление прокруткой:

а) задание диапазона шагов. С первой позиции перфокарты набивается номер шага,с которого надо начать прокрутку, затем через запятую - номер шага, на котором надо кончить. Например:

```
500, 2300
1,325
```

Если надо задать только начальный номер шага, набивается одно число. Например:

```
600
9500
```

б) прокрутка по детерминативам. С первой позиции перфокарты набивается знак равенства и вплотную к нему имя функции, работу которой надо прокрутить. Например:

```
=MET2
=MUL
=DET1
=SIGMA
```

Таких карт должно быть не более десяти одновременно;

в) прокрутка со знаками "больше" (>) и "меньше" (<).

С первой позиции набивается знак > или <, вплотную к нему - имя функции. Например:

```
>EBK
<KOMM
```

Если есть одна карта типа > или <,к ней можно добавить только карту с указанием диапазона шагов.

Управляющие карты прокрутки кладутся в любом порядке перед именем выполняемой программы во входном потоке SYSIN!

Например:

```
//SYSIN DD *
300,850
=MAPE
=ALFA
=PSI
=PROG
....      <входная информация для процедуры CARD >
/*
```

Другой пример (при использовании процедуры REFALG):

```
//GO.SYSIN    DD *
<TRT

750
GO
.... <входная информация для процедуры CARD>
/*
```

6. Диагностические сообщения и коды возврата

При компиляции или при счете рефал-программ ИЯС может по-разному оканчивать работу. Чтобы можно было управлять прохождением следующих шагов задания, ИЯС выдает при различных окончаниях разные коды возврата. Ниже они описываются вместе с появляющимися при этом сообщениями о причине окончания работы.

1. Код возврата-0. Нормальный конец работы, вызванный тем, что в поле зрения не осталось знаков конкретизации. Появляется сообщение: NORMAL END OF JOB.

2. Код возврата-4. Отождествление невозможно. Появляется сообщение: REFAL ERROR - RECOGNITION IMPOSSIBLE. Распечатывается сообщение: EXPRESSION UNDER CONCRETIZATION (конкретизируемое выражение) и затем в метакоде - Б распечатывается ведущая область конкретизации. Распечатка ограничивается слева и справа тройками звездочек.

3. Код возврата-8. Второе обращение к кончившемуся входному набору данных SYSIN при использовании машинной процедуры CARD (см.главуУI). Появляется сообщение SYSIN - END OF DATA SET.

4. Код возврата-12. Не открылся набор данных с именем DD-предложения SYSIN, LIB, SYSUT1 или SYSUT2. Появляется сообщение: SYSIN FAILED TO OPEN (соответственно LIB ..., SYSUT1 или SYSUT2). Если не открылся набор данных SYSPRINT, соответствующее сообщение появляется на пульте оператора, так как REFAL в этом случае распечатать сообщение не сможет. Возможные ошибки: не описано DD-предложение или при описании DD-предложения указаны неверные параметры DCB.

5. Код возврата-16. Недостаточно памяти для поля зрения. Надо учитывать специфическое представление информации для рефал-программ при оценке размера памяти, которую следует отвести задаче (каждый символ или скобка в поле зрения занимает 8 байтов). Память под поле зрения отводится динамически: в начале работы отводится 2 килобайта (256 звеньев). Если в процессе работы этого оказывается недостаточно, добавляется еще один килобайт,

затем еще и т.д. Если операционная система не сможет удовлетворить очередной запрос на килобайт памяти, задача заканчивается аварийно. Появляется сообщение: INSUFFICIENT STORAGE FOR VIEW – FIELD, печатается имя функции, которая работала последней.

6. Код возврата-20. Переполнен стэк переходов. Появляется сообщение: STACK OVERFLOW.

7. Код возврата-24. Ошибка в управляющих картах прокрутки. Появляется сообщение: ERROR IN A DEBUGGING CONTROL CARD и распечатывается ошибочная карта.

8. Код возврата-28. Блок синтаксического контроля рефал-компилятора обнаружил синтаксические ошибки. Компиляция не производится. Печатается диагностика ошибок и сообщение: COMPILATION WAS NOT EXECUTED BECAUSE OF SYNTAX ERRORS

9. Код возврата-32. При компиляции обнаружено, что левые части предложения слишком большие, т.е. не выполнено ограничение на количество объектных знаков, скобок и вхождений переменных в левой части предложения, которое было введено в разд.2 главы У1. Появляется сообщение: LEFT PART TOO GREAT. На этом компиляция прекращается.

10. При работе рефал-программ может возникнуть еще одна аварийная ситуация, которую ИЯС не в состоянии обработать самостоятельно – ошибку обнаруживает операционная система. Если вы забыли подложить карту с именем загружаемой рефал-программы, или указали имя неверно, или не записали предварительно нужную рефал-программу в библиотеку шага (STEPLIB), произойдет системный авост при попытке выполнить макрокоманду LOAD (загрузить).

Сообщив о причине окончания работы, ИЯС обычно распечатывает полезную дополнительную информацию:

1. Распечатывается содержимое поля зрения (VIEW – FIELD) в момент окончания работы. Распечатка производится в метакоде-Б и ограничивается по краям тройками звездочек. Если причина окончания работы – нехватка памяти, распечатать все поле зрения не удается, так как внутри конкретизируемого выражения списковая структура оказалась испорченной. Поэтому распечатывается сначала часть поля зрения слева от конкретизируемого выражения (VIEW – FIELD TO THE LEFT FROM EXPRESSION UNDER CONCRETIZATION), затем – справа (THE SAME TO THE RIGHT).

2. Список закопанной информации (копилка), если он не пуст. Печатается слово BURIED и содержимое копилки в метакоде-Б.

3. Максимальное число знаков k в процессе работы.

4. Память, отведенная под поле зрения (в килобайтах).

5. Число шагов рефал-машины.

Эта информация, как правило, необходима при отладке программы, но может оказаться лишней при многократном использовании

уже отлаженной программы. Чтобы заблокировать распечатку этой информации, надо передать программе REFAL параметр NOPRINT

```
//EXEC PGM = REFAL, PARM = NOPRINT
```

Если параметр NOPRINT отсутствует (или ошибочный), информация распечатается. При запуске компилятора обычно указывается параметр NOPRINT.

7. Примеры

Приведем пример запуска на компиляцию и счет рефал-программы. Пример иллюстрирует типичную обработку и контроль входных данных, а также распечатку результатов счета в удобном виде.

Программа реализует алгоритм Евклида (нахождение наибольшего общего делителя двух чисел). Информация задается в таком виде: с первой позиции перфокарты набивается первое число в десятичной системе, затем через запятую - второе. Программа читает карту, печатает ее с начала строки и затем обрабатывает. Результат обработки печатается в следующей строке (отступя от начала строки на три позиции). Если информация на карте ошибочная, результатом обработки будет слово ERROR (ошибка), если верная - наибольший общий делитель в десятичной системе счисления. Обработав карту, программа перейдет к следующей и повторит с ней описанные действия. Кончится работа по концу входного файла. Работу каждой функции программы объясним ниже.

```
//EUQLID JOB 24, PETROV, MSGLEVEL = 1
//        EXEC REFALG
//REF.SYSIN DD *
RFC
          START
          ENTRY TASK
          EXTRN CARD, PRINT
          EXTRN ADD, MUL, DR, NUMB, SYMB
          EXTRN TYPE
TASK = K /CYC/ K /PRINT/  K/CARD/ ...
CYC  =
     E1 = K /TEST/E1.K/TASK/.
TEST E1 ',' E2 '␣' E3 =                          1
            K/GQ/ K /TRT/E1. K /TRT /E2 . .
     E1 = K /ERROR /.
TRT  =
```

```
      EA = K /TRT1 /(  ) K /TYPE / EA . .
TRT 1 (E 1) 'D' SX E2 = K /TRT 1 /(E1 SX)                          1
             K/TYPE / E2 . .
      (E1) '* ' = (K /CVB / E 1 .)
      (E1) E2 =
GQ    W1 W2 = K /OUT / K /PRINT / '␣␣␣' K / CVD /                  1
             K /EUWL /W1W2 . . . .
      EA = K /ERROR /.
EUQL (EA) E1 '(/0/)     = EA
      (EA) E1 (EB) = K /EUQL / (EB) K /DR / (EA) EB . .
CVB   SX E1 = K /CVB1/(K /NUMB / SX .) E1 .
CVB1 (E1) SX E2 = K /CVB1/(K /ADD/                                 1

             (K /MUL / (E1) /10 /.) K /NUMB / SX .) E2 .
      (E1) = E1
CVD   E1 = K / CVD 1 / K /DR /(E1) /10 /. .
CVD1  /0 / (SX) = K /SYMB / SX  .
      E1 (SX) = K /CVD 1 / K /DR /(E 1) /10/. .                    1
                   K /SYMB / SX .
OUT   EA =
ERROR = K /OUT / K /PRINT / '␣␣␣ ERROR ' . .
      END  TASK
//LKED. SYSIN  DD *
      INCLUDE  REFLIB (RINOUT , ARITHM , SYNTAX )
//GO. SYSIN  DD  *
< TRT
15,200
GO
2652834791268,1246
256,32768
0,149
124,72886
126.112
15674,
/*
```

TAS K – входная точка программы. Эта функция работает первой. Она вызывает машинные процедуры чтения и печати, а также функ-

цию СУС, которая организует цикл чтения, проверку конца файла и подает карту на дальнейшую обработку – функцию TEST. Функция TEST выделяет первый и второй операнды (отыскивая опорные символы: запятую и пробел). Если она не сможет этого сделать, она передает управление функции ERROR, которая печатает сообщение об ошибке.

Функция TRT, к которой обращается функция TEST, производит окончательный синтаксический контроль входной информации: т.е. выясняет, не пусто ли число, все ли его символы являются цифрами. Если найдется ошибка, TRT выдаст пустой результат. Если ошибки нет, TRT вызовет функцию CVB (перевести в "двоичную"). Окончательным результатом работы TRT окажется взятое в скобки число в 2^{31}-ичной системе счисления.

Функция GQ по наличию двух термов убедится, что оба операнда верные, и вызовет новую последовательность функций: сначала она обратится к функции EUQL, которая по алгоритму Евклида найдет НОД и выдаст его в 2^{31}-ичной системе счисления, затем функция CVD переведет результат в десятичную систему, PRINT напечатает этот результат и OUT уберет его из поля зрения.

Функция CVB производит перевод по схеме Горнера.

CVD осуществляет перевод в десятичную систему по известному алгоритму (деление с остатком и запись остатков справа налево).

В программе используются стандартные машинные процедуры: CARD, PRINT, TYPE,ADD,MUL, DR, SYMB, NUMB. Об их работе см.главу У1.

Прокрутка производится с 15 по 200 шаг, пока не встретится функция TRT. В этот момент прокрутка прекращается, пока не вычислится полностью функция TRT. Пусть, например, на TRT была подана последовательность объектных знаков '2147278850' ($2^{31}+2$). Прокрутка напечатает:

```
        STEP ...
    ***K/TRT/ '2147278850'.***
    и как результат замены:
    ***(/1/ /2/) ***
```

Две последние карты ошибочные, программа напечатает сообщение: ERROR.

Запуск процедуры REFALG обычно применяется при отладке. Если программа уже отлажена, не нужно каждый раз заново компилировать ее. Запишите отлаженную программу в библиотеку и вызывайте ее на счет всякий раз, когда это понадобится.

Запишем описанную нами программу в библиотеку REFAL.LOAD под именем EUQLID при помощи следующего пакета управляющих карт:

```
    //      EXEC REFAL
```

```
//REF.SYSIN DD *
RFC
        START
        .
        .
        .
        END TASK
//LKED. SYSIN DD *
        INCLUDE REFLIB (RINOUT, SYNTAX, ARITHM)
//LKED. SYSLMOD DD DSN = REFAL. LOAD (EUQLID)
//          DISP = OLD, UNIT = 2311, VOL = SER = AUTOL
```

(Библиотека находится на диске типа 2311, серийный номер AUTOL).

Теперь запустить эту программу на счет можно таким образом:

```
//          EXEC PGM = REFAL, PARM = NOPRINT
//STEPLIB DD DSN = REFAL. LOAD, DISP = SHR,
//               UNIT = 2311, VOL = SER = AUTOL
//SYSPRINT DD SYSOUT = A
//SYSIN DD *
EUQLID
2456749810765872,5674287
1000000000000000,65555555
15,455
/*
```

(Сервисные распечатки заблокированы, счет проводится без прокрутки).

Если программа разбита на несколько частей, которые надо компилировать отдельно, применять каталогизированные процедуры нельзя (при компиляции первых кусков программы), так как редактировать связи можно только после компиляции всех частей. Последовательность действий в этом случае такая: запускаете компилятор, затем с помощью каталогизированной процедуры ASMFC транслируете получившийся символьный модуль, и записываете результат трансляции в какую-нибудь библиотеку объектных модулей. Таким же образом поступаете с остальными частями программы, после чего объединяете все части редактором связей.

```
//COMP EXEC PGM = REFAL, PARM = NOPRINT
//SYSIN DD *
RFC
                 ...     < программа >
//SYSPRINT DD SYSOUT = A
```

```
//LIB DD DSN = &LIB, SPACE = (400, (1000, 50)),
//         DISP = (NEW, PASS), UNIT = SYSDA
//ASM EXEC ASMFC, PARM.ASM = 'NOLIST, LOAD'
//ASM.SYSGO DD DSN = LIB.OBJ (PART 1), DISP = OLD,
//              UNIT = 2311, VOL = SER = 231103
//ASM.SYSIN DD DSN = &LIB, DISP = (OLD, DELETE)
```

УШ. ЗАПУСК ПРОГРАММ НА МАШИНЕ БЭСМ-6

Компиляция, загрузка и выполнение рефал-программ происходит под управлением мониторной системы "Дубна". В этой главе содержатся только самые необходимые сведения о мониторной системе "Дубна". Более подробную информацию можно почерпнуть из работ [18-22].

1. Местонахождение рефал-системы

Рефал-система является набором загрузочных модулей, которые должны храниться в личной библиотеке пользователя рефала (PERSONAL LIBRARY). Поэтому вам нужно записать рефал-систему на свою магнитную ленту в PERSONAL LIBRARY. Этот набор модулей включает в себя административную часть, компилятор, систему отладки и машинные процедуры. Там же можно хранить и ваши программы, странслированные с рефала и других языков.

В паспорте задачи не забудьте указать ленту с PERSONAL LIBRARY, подложив такую карту:

ЛЕНТЫ␣67 (*yyy*- 3П)‾

где *yyy* - десятичный номер бобины.

2. Простейший пакет

Теперь мы рассмотрим пример трансляции и счета рефал-программы, приведенной на рис. 6.2.

Нужно составить следующий пакет перфокарт:

```
*NAME USEROV
*PERSONAL LIBRARY
*CALL REFAL
```

```
ЬДЕЛО СТАРТ                 ⎫
    ВХОДН  ДЕЛО             ⎪
    ВНЕШН  КАРТА,П          ⎬ рефал-программа (см.рис.6.2)
    .                       ⎪
    .                       ⎪
    .                       ⎪
    ФИНИШ                   ⎭
*READ DRUM
*FORTRAN
          PROGRAM DELO
          EXTERNAL ДЕЛО
          CALL  RFВЫП (ДЕЛО)
          END
*EXECUTE
A(B*C)( )(C*B)A  }  исходные данные для рефал-про-
*END FILE            граммы
```

Мониторная система "Дубна" выполнит этот пакет следующим образом.

Сначала прочитается карта *NAME. С этой карты обязательно должен начинаться любой пакет. На ней указывается фамилия пользователя в позициях 7-24.

Затем монитор последовательно прочитает и исполнит остальные карты пакета.

Прочитав карту *PERSONAL LIBRARY, монитор перепишет содержимое личной библиотеки на барабан и сформирует на нем временную библиотеку. Следующая карта *CALL REFAL означает, что нужно вызвать компилятор с рефала. Затем идет программа на рефале, после нее обязательно должна идти карта *READ DRUM. Необходимо запомнить, что она всегда должна стоять после карты ФИНИШ. Рефал-компилятор производит синтаксический контроль программы, печатает исходный текст программы и сообщения об ошибках. Если ошибок в программе нет, то программа транслируется, формируется загрузочный модуль, который затем переписывается во временную библиотеку на барабане.

Дальше в пакете лежит карта *FORTRAN, которая указывает, что следующие за ней карты представляют собой программу на ФОРТРАНе. Откуда же взялась программа на ФОРТРАНе и зачем она нужна?

Дело в том, что рефал-программа способна работать только под управлением административной системы. Поэтому просто передать ей управление нельзя, и она запускается с помощью административ-

ной подпрограммы RFВЫП. RFВЫП имеет один параметр – входную метку рефал-программы $\mathscr{D}$ и всегда формирует начальное поле зрения вида:

K $\mathscr{D}$.

Чтобы вызвать RFВЫП и передать ей параметр, пишется маленькая программка на ФОРТРАНе. Во всех картах фортранной программы информация размещается, начиная с 7 позиции. Карта PROGRAM DELO означает, что программа имеет имя DELO. Имя должно состоять не более чем из шести латинских букв и цифр и начинаться с буквы. Карта EXTERNAL ДЕЛО означает, что нас интересует входная точка ДЕЛО в одном из рефал-модулей. Карта CALL RF ВЫП (ДЕЛО) означает, что надо вызвать подпрограмму RFВЫП, которая сформирует начальное поле зрения

K /ДЕЛО/.

и запустит рефал-программу. Карта END означает конец программы на ФОРТРАНе.

Знать смысл программки на ФОРТРАНе необязательно, достаточно переписать ее чисто механически, вставив вместо DELO любое имя, а вместо ДЕЛО – интересующую вас входную метку.

Дальше в пакете лежит карта *EXECUTE. Эта карта вызовет загрузку и исполнение только что странслированной фортранной программы. При этом в память вместе с ней загрузится административная система, рефал-модуль с входной меткой ДЕЛО и нужные машинные процедуры.

Затем управление передастся программе DELO, которая обратится к RFВЫП, и наконец ваша рефал-программа начнет работать.

После карты *EXECUTE следует положить карты с исходной информацией для рефал-программы, которые можно прочитать посредством машинной процедуры КАРТА. В данном случае исходные данные состоят из единственной карты

A (B * C) ( ) (C * B) A

После карт с исходными данными идет карта *END FILE, которая служит указанием на конец пакета и которой должен заканчиваться любой пакет.

Таков простейший пакет. Если же вы желаете собирать более сложные пакеты, то вам нужно ознакомиться с более подробным описанием.

3. Вызов рефал-компилятора

Запуск рефал-компилятора имеет следующий вид:

```
*CALL REFAL
 < имя модуля >   СТАРТ  \
                    :      } рефал-модуль
                  ФИНИШ  /
*READ DRUM
```

По карте *CALL REFAL вызывается рефал-компилятор. Производится синтаксический контроль программы. Если в программе есть ошибки, то компиляция не производится. Печатаются диагностические сообщения об ошибках вместе с неправильными картами, после чего карта *READ DRUM пропускается и выполнение пакета продолжается.

Если ошибки не обнаружены, рефал-компилятор транслирует рефал-модуль на автокод БЕМШ и помещает его на барабан. Затем картой *READ DRUM вызывается транслятор с БЕМШ, который формирует загрузочный модуль и записывает его во временную библиотеку на барабане.

Если установлен стандартный режим распечатки программ при трансляции, то рефал-компилятор распечатывает текст исходного рефал-модуля. Можно отменить печать с помощью управляющей карты *NO LIST. Для этого следует поместить ее перед очередной картой *CALL REFAL. Все рефал-модули, расположенные после нее, не будут распечатываться, пока не будет восстановлен стандартный режим печати. Восстановить его можно картой *STANDARD LIST, подложив ее перед одной из карт *CALL REFAL . Карта *NAME всегда устанавливает режим *STANDARD LIST, поэтому, как правило, не приходится специально заботиться о распечатке программы.

Таким образом, если вы хотите странслировать два рефал-модуля с именами МОД1 и МОД2, причем первый не печатать, а второй печатать, то это можно сделать, подложив следующие карты:

```
*NO LIST
*CALL REFAL
МОД1    СТАРТ  \
          :     } Первый рефал-модуль
        ФИНИШ  /
```

```
*READ DRUM
*STANDARD LIST
*CALL  REFAL

МОД2      СТАРТ    )
           .       |
           .       } Второй рефал-модуль
           .       |
          ФИНИШ    )
*READ DRUM
```

Иногда может потребоваться проделать только синтаксический контроль программы без компиляции и получения загрузочного модуля. Для этого следует подложить такие карты:

```
*CALL RCHECK
<имя модуля>   СТАРТ
                 .
                 .
                 .
               ФИНИШ
```

В этом случае карту *READ DRUM после карты ФИНИШ класть не надо! Производится синтаксический контроль рефал-модуля и печатаются сообщения об ошибках. Управление режимом печати производится с помощью карт *NO LIST и *STANDARD LIST точно так же, как при запуске компилятора.

4. Запуск готовой программы

Запуск готовой рефал-программы производится с помощью административной подпрограммы RFВЫП. Эта подпрограмма имеет один параметр – входную метку некоторого рефал-модуля, которая является детерминативом некоторой функции $\mathscr{D}$. Обращение к RFВЫП из ФОРТРАНа имеет вид:

CALL RFВЫП ($\mathscr{D}$)

при этом RFВЫП формирует начальное поле зрения вида

K $\mathscr{D}$.

и запускает "рефал-машину".

Метка $\mathscr{D}$ должна быть описана в программе на ФОРТРАНе как внешняя с помощью оператора EXTERNAL.. Напоминаем, что эта метка должна быть описана как входная в одном из рефал-модулей.

Таким образом, простейшая подпрограмма на ФОРТРАНе, вызывающая рефал-программу, имеет вид:

SUBROUTINE $\mathscr{P}$

EXTERNAL $\mathcal{D}$

CALL RFВЫП ($\mathcal{D}$)

END

где $\mathcal{P}$ – имя подпрограммы, т.е. метка, длиной не более шести символов.

Теперь, чтобы выполнить рефал-программу, при начальном поле зрения

K$\mathcal{D}$.

в конце пакета следует подложить следующие карты:

*MAIN $\mathcal{P}$

*EXECUTE

исходные данные для рефал-программы

*END FILE

В карте *MAIN с 7-й позиции указывается имя подпрограммы, которую следует выполнить. Имя $\mathcal{P}$, указанное в карте *MAIN, не должно содержать русских букв, не совпадающих с латинскими. Затем следует карта *EXECUTE . Прочитав эту карту, мониторная система загрузит в память подпрограмму $\mathcal{P}$ и все модули, к которым она обращается непосредственно или через другие модули. Таким образом, автоматически загрузятся все модули административной системы, нужные рефал-модули и машинные процедуры, которые используются в данной программе. После этого управление передастся программе $\mathcal{P}$, которая обращается к RFВЫП, после чего начинает работать рефал-программа.

После карты *EXECUTE лежат карты с исходными данными для рефал-программы, которые можно читать с помощью машинной процедуры КАРТА. За ними следует карта *END FILE.

Конечно, если программа ничего не читает с помощью машинной процедуры КАРТА, то и исходных данных подкладывать не нужно. В этом случае сразу же после карты *EXECUTE следует *END FILE.

Если программа не знает заранее, сколько карт займут исходные данные, то нужно предусмотреть какой-то признак конца и распознавать его в программе. Если программа прочитает все исходные данные, а потом попробует прочесть карту *END FILE, то мониторная система сразу же прекратит работу. В качестве признака конца исходных данных можно использовать стандартную карту

//␣ END

прочитав которую, машинная процедура КАРТА выдает в качестве результата замены пустое выражение, что легко проверить в программе.

Подпрограмму $\mathscr{P}$, которая обращается к RFВЫП, можно хранить в PERSONAL LIBRARY. Тогда ее достаточно протранслировать только один раз. Если же вы транслируете ее при каждом запуске заново, то конец пакета будет иметь следующий вид:

*FORTRAN
SUBROUTINE $\mathscr{P}$
EXTERNAL $\mathscr{D}$
CALL RFВЫП ($\mathscr{D}$)
END
*MAIN $\mathscr{P}$
*EXECUTE

исходные данные для рефал-программы

*END FILE

Можно обойтись без карты *MAIN . В этом случае в фортранной подпрограмме вместо оператора SUBROUTINE $\mathscr{P}$ следует поместить оператор PROGRAM $\mathscr{P}$. Тогда конец пакета будет иметь следующий вид:

*FORTRAN
PROGRAM $\mathscr{P}$
EXTERNAL $\mathscr{D}$
CALL RFВЫП ($\mathscr{D}$)
END
*EXECUTE

исходные данные для рефал-программы

*END FILE

5. Средства отладки

Административная подпрограмма RFВЫП не содержит средств отладки. Ею обычно пользуются для запуска уже отлаженных рефал-программ. Если вы хотите отлаживать рефал-программу, следует воспользоваться административной подпрограммой RFОТЛ. Обращение к ней ничем не отличается от обращения к RFВЫП. Достаточно только в вызывающей фортранной программе вместо оператора

CALL RFВЫП ($\mathscr{D}$)

написать оператор

CALL RFOTЛ ($\mathscr{D}$)

Средства отладки в основном заключаются в том, что вы можете организовать пошаговую печать или, как говорят, прокрутку. Прокрутка дает возможность на каждом шаге рефал-машины получить следующую информацию:

1) номер шага;

2) ведущий терм (конкретизируемое выражение);

3) результат замены.

Рефал-программа запускается на счет примерно так же, как и при использовании административной подпрограммы RFВЫП. Для этого следует подложить следующие карты:

```
*FORTRAN
          SUBROUTINE 𝒫
          EXTERNAL 𝒟
          CALL RFOTЛ ( 𝒟 )
          END
*MAIN 𝒫
*EXECUTE
```

управляющие карты прокрутки

*

исходные данные для рефал-программы

*END FILE

Отличие состоит в том, что после карты *EXECUTE сначала идут управляющие карты прокрутки, затем карта со звездочкой в первой позиции, и только потом - исходные данные для рефал-программы.

Карты прокрутки имеют следующий вид: начиная с первой позиции записывается ключевое слово, а затем через один или несколько пробелов следует информация. С помощью карт прокрутки задаются условия прокрутки. Если нет ни одной карты прокрутки, то печать шагов не производится. Опишем, как задается каждое из четырех условий прокрутки.

1. Условие "по шагам". Начальный шаг прокрутки задается картой с ключевым словом С в виде

С $\mathscr{N}$

где $\mathscr{N}$ - номер начального шага в десятичной системе.

Конечный шаг прокрутки задается картой с ключевым словом ДО в виде

ДО $\mathcal{N}$

где $\mathcal{N}$ – номер конечного шага прокрутки в десятичной системе.

Например, прокрутка с 100 по 1000 шаг задается двумя картами:

С 100

ДО 1000

Если карта С опущена, то подразумевается

С 1

Если опущена карта ДО, то прокрутка ведется до конца работы рефал-программы.

Если опущены обе карты С и ДО, но имеется хотя бы одна из описанных ниже карт прокрутки, то предполагается, что задан диапазон с первого шага до конца.

2. Условие "по детерминативам" задается на одной или нескольких картах с ключевым словом ПО. На одной карте указывается один детерминатив в виде

ПО $\mathcal{D}$

где $\mathcal{D}$ – детерминатив.

Например, прокрутка шагов с детерминативами АЛЬФА, БЕТА, ГАММА задается на трех картах:

ПО АЛЬФА

ПО БЕТА

ПО ГАММА

3. Условие " > " задается на одной или нескольких картах с ключевым словом ПО. На одной карте указывается один детерминатив в виде

ПО >$\mathcal{D}$

где $\mathcal{D}$ – детерминатив.

Например, условие " > " с двумя детерминативами ПСИ и ФИ задается так:

ПО >ПСИ

ПО >ФИ

4. Условие " < " задается аналогично условию " > ", только вместо знака " > " на картах ПО ставится знак " < ".

Карты прокрутки всех четырех видов подкладываются в любом порядке.

Среди карт прокрутки можно подложить карту останова по номеру шага и карту сборки мусора.

Карта останова по номеру шага имеет вид

СТОП $\mathcal{N}$

где $\mathcal{N}$ – номер шага в десятичной системе. Если подложить такую карту среди карт прокрутки, то программа прекратит работу, закон-

чив указанный шаг. При этом печатается состояние рефал-машины в момент останова.

Карта сборки мусора имеет вид

СМУС

Если подложить такую карту среди карт прокрутки, то при каждой сборке мусора будет печататься следующая информация:

1) номер шага, во время которого произошла сборка мусора;

2) количество ячеек памяти, освободившихся в результате сборки мусора.

Пример. Пусть нужно запустить программу, приведенную на рисунке 6.2. При этом требуется распечатать шаги с детерминативом СПАР, начиная с 10-го шага. Для этого нужно подложить следующие карты:

```
*FORTRAN
         PROGRAM DELO
         EXTERNALI ДЕЛО
         CALL RF ОТЛ (ДЕЛО)
         END
*EXECUTE
С 10
ПО СПАР
*
A (B * C) ( ) (C * B) A
* END FILE
```

Вместо ключевых слов С,ДО,ПО,СТОП на картах прокрутки и останова можно использовать слова FROM, TO, IF, STOP соответственно.

При выполнении рефал-программы возможны аварийные остановы "отождествление невозможно" и "свободная память исчерпана". При этом печатается информация, необходимая для обнаружения ошибки в рефал-программе.

6. Работа с личной библиотекой

Как уже говорилось, рефал-система и странслированные программы пользователя находятся в личной библиотеке на магнитной ленте. С другой стороны, все трансляторы работают непосредственно только с временной библиотекой, которая создается только на время выполнения одного пакета и находится на магнитном барабане.

Поэтому, чтобы изменить личную библиотеку, ее сначала нужно перенести на барабан, образовав на нем временную библиотеку. Затем произ-

водится несколько трансляций, и во временную библиотеку заносится несколько загрузочных модулей. После этого, если мы хотим зафиксировать происшедшие изменения в личной библиотеке, следует переписать содержимое временной библиотеки на ту же ленту, сформировав на ней заново личную библиотеку.

Если встречается управляющая карта *PERSONAL LIBRARY, мониторная система переписывает содержимое личной библиотеки на барабан. Если встречается управляющая карта *TO PERSONAL LIBRARY, мониторная система переписывает содержимое временной библиотеки на ленту.

Пример. Пусть надо странслировать два рефал-модуля МОД1 и МОД2, причем МОД1 надо сохранить в PERSONAL LIBRARY для дальнейших запусков, а МОД2 нужен только для данного запуска. Тогда следует собрать пакет, имеющий такое начало:

```
* NAME ПЕТРОВ
* PERSONAL LIBRARY
* CALL REFAL
МОД1    СТАРТ
          :
        ФИНИШ
* READ DRUM
* TO PERSONAL LIBRARY
* CALL REFAL
МОД2     СТАРТ

         :
        ФИНИШ
* READ DRUM
         :
```

Каждый раз, когда во временную библиотеку заносится новый модуль, его входные метки сравниваются с входными метками других модулей, уже находящихся в библиотеке. При этом, если в новом модуле есть входная метка, которая совпадает с входной меткой старого модуля, то у старого модуля соответствующая входная метка исчезает. Если у какого-то модуля не останется ни одной входной метки, он удаляется из библиотеки. Таким образом, автоматически удаляются старые варианты модулей, которые остались от предыдущих трансляций.

Такая система удобна, но с ней сопряжены и некоторые опасности, так как неосторожно выбранная входная метка может стереть в библиотеке нужный модуль. Поэтому советуем распечатать каталог временной библиотеки, подложив в пакет карту

* CALL TCATALOG

и посмотреть, какие модули с какими входными метками в ней содержатся.

Имена машинных процедур вам известны. Имена административных подпрограмм всегда начинаются либо с букв RF, либо с буквы L и обязательно содержат хоть одну русскую букву, отличную от латинской.

7. Связь с ФОРТРАНом

В некоторых случаях в программе, написанной на рефале, желательно использовать подпрограммы, написанные на ФОРТРАНе. Кроме того, хорошо иметь возможность писать машинные процедуры на ФОРТРАНе.

С этой целью во входной язык рефал-компилятора введены карты SCALL и FCALL. Они имеют вид

SCALL метка-1, метка-2, . . . , метка-n

FCALL метка-1, метка-2, . . . , метка-n

Кроме того, эти карты могут иметь еще один формат:

метка-1 SCALL .метка-2

метка-1 FCALL .метка-2

Легко видеть, что форматы карт SCALL и FCALL полностью совпадают с форматами карты ВНЕШН. И это не случайно. Карты SCALL и FCALL дают еще один способ описания внешних меток, но при этом компилятору сообщается, что эти внешние функции описаны не на рефале, а на ФОРТРАНе.

Карта SCALL применяется, когда нужно вызвать фортранную подпрограмму без параметров, которая специально не предназначалась для того,чтобы ее вызывали из рефала. Естественно, что такая подпрограмма ничего "не знает" о рефале. Поэтому она заведомо не берет никакой информации из поля зрения и ничего не выдает в поле зрения. Естественно считать, что после обращения к такой подпрограмме в поле зрения останется пустое выражение в качестве результата замены.

П р и м е р. Пусть подпрограмма, к которой мы хотим обращаться, называется SUBR . Тогда ее следует определить как внешнюю метку с помощью карты SCALL :

SCALL SUBR

Если теперь выполнить конкретизацию

K/SUBR /.

то произойдет обращение к подпрограмме SUBR, и она начнет работать. Затем SUBR возвратит управление вызвавшей ее программе. После этого шаг заканчивается, результат замены – "пусто".

Карта FCALL применяется, когда нужно вызвать машинную процедуру, написанную на ФОРТРАНе. С точки зрения ФОРТРАНа она представляет собой обычную подпрограмму без параметров, написанную с учетом того, что ее будут вызывать из рефала. Поэтому она сама "несет ответственность" за выборку информации из ведущей области конкретизации и формирование результата замены. Использование таких машинных процедур ничем не отличается от использования других внешних функций, кроме того, что они описываются не картами ВНЕШН, а картами FCALL.

Опишем некоторые полезные подпрограммы, которые всегда есть в личной библиотеке, так как входят в состав рефал-системы.

Подпрограмма PTIME. При обращении к этой подпрограмме печатается время процессора, затраченное к этому моменту на решение задачи. В качестве примера приведем программу, которая печатает текущее время после каждых 10000 прибавлений единицы к макроцифре.

```
ЬПРГ    СТАРТ
        ВХОДН ПРГ
        ВНЕШН P1
        SCALL  PTIME
ПРГ     = K/PTIME/. . K/ПРГ1/ /0/.
ПРГ1    /10000/ = K/ПРГ/.
        SN  = K/ПРГ1/ K/P1/SN ..
        ФИНИШ
```

Подпрограмма RECXCФ и RFBCCФ. Эти подпрограммы используются в тех случаях, когда мы читаем исходные данные с помощью машинной процедуры КАРТА, и при этом хотим прочесть одни и те же карты не один раз, а два или вообще несколько раз.

Машинная процедура КАРТА при каждом обращении к ней выдает очередную карту с исходными данными. Но как она узнает, какую карту нужно прочесть в очередной раз? Можно представить себе, что вдоль массива карт с исходными данными движется некоторый указатель – "стрелка файла". Стрелка файла все время указывает на ту карту, которую нужно прочесть при очередном обращении к машинной процедуре КАРТА. Каждое обращение к функции КАРТА продвигает стрелку файла на одну карту вперед. Таким образом, она начинает указывать на следующую карту и т.д.

Можно запомнить текущее положение стрелки файла, обратившись к подпрограмме RFCXCФ. Затем, прочитав несколько карт, можно обратиться к подпрограмме RFBCCФ, которая передвинет стрелку файла назад, в то положение, в котором она находилась в момент обращения к подпрограмме RFCXCФ. Если теперь мы начнем читать карты, то получится, что мы прочитаем второй раз те карты, которые еще не были прочитаны к моменту обращения к RFCXCФ.

В качестве примера рассмотрим функцию ДВЕ, которая каждую очередную карту входной информации прочитывает два раза подряд.

```
        ВНЕШН КАРТА
        SCALL RFCXCФ, RFBCCФ
    ДВЕ = К/RFЖХСФ/. К/КАРТА/. +
          К/RFBCCФ/. К/КАРТА/.
```

Эта функция эквивалентна следующей функции:

```
        ВНЕШН КАРТА
    ДВЕ = К/ДВЕ1/.К/КАРТА/..
    ДВЕ1    ЕА = ЕА ЕА
```

Повторное чтение исходных данных может понадобиться, например, при написании многопроходных трансляторов.

Подпрограмма RFOШТР . Эта подпрограмма может использоваться при написании на рефале трансляторов. Если ваш транслятор обнаружит ошибки в обрабатываемой программе, то следует обратиться к фортранной подпрограмме RFOШТР . Эта подпрограмма проинформирует мониторную систему "Дубна" о том, что были ошибки при трансляции. Если после этого в пакете встретится управляющая карта *EXECUTE, то эта карта выполняться не будет и мониторная система сразу же прекратит работу.

8. Машинные процедуры для работы с символьными файлами

Машинные процедуры OPEN, PUT, CLOSE предназначены для записи данных в символьные файлы на магнитную ленту или барабан.

Перед тем как начать запись данных в файл, нужно выполнить конкретизацию:

K/OPEN/<адрес файла> .

где указывается адрес файла. Адрес файла представляет собой последовательность из пяти объектных знаков вида НМ333, где

Н – математический номер направления;

М – математический номер устройства;

333 – восьмеричный номер зоны или тракта, начиная с которого будет располагаться символьный файл.

Например:

K/OPEN/'67100'.

означает, что файл расположится на магнитной ленте 67, начиная с зоны 100_8. А конкретизация

K/OPEN/'03000'.

означает, что файл расположится на магнитном барабане 03, начиная с тракта 000_8.

В качестве результата замены функция OPEN выдает пустое выражение.

Запись данных в файл осуществляется функцией PUT, обращение к которой имеет вид:

K /PUT / ℰ.

где ℰ –произвольное выражение, не содержащее составных символов. Структурные скобки, содержащиеся в ℰ, записываются в файл как объектные знаки '('и')'.

Функция PUT разбивает выражение ℰ на записи по 80 символов. При необходимости последняя запись дополняется справа до 80 символов пробелами. Эти записи помещаются в файл. При следующем обращении к функции PUT новые записи добавляются к предыдущим.

Если на функцию PUT подано пустое выражение

K /PUT/.

то в файл помещается одна запись, состоящая из 80 пробелов.

Результатом замены при конкретизации

K /PUT / ℰ.

является выражение ℰ без изменения.

По окончании формирования символьного файла нужно обратиться к функции CLOSE:

K /CLOSE /.

В качестве результата замены функция CLOSE выдает пустое выражение.

Если рефал-программа завершится без обращения к функции CLOSE (аварийно или нормально), то файл будет потерян.

После обращения к функции CLOSE вы можете снова обратиться к функции OPEN и сформировать еще один файл.

Пример. Рефал-программа

```
БЗАПФ СТАРТ
       ВХОДН ЗАПФ
       ВНЕШН OPEN, PUT, CLOSE
       ВНЕШН КАРТА,П
ЗАПФ   = К/ЗАПФ1/ К/КАРТА/. .
ЗАПФ1  EA' ' EB = К/ OPEN / EA. +
                  К/ЗАПФ2/ К/КАРТА/. .
ЗАПФ2  =        К /CLOSE/.
       EX = К/П/ К/ PUT / EX . . +
            К/ЗАПФ2/ К/КАРТА/ . .
       ФИНИШ
```

если к ней обратиться, подложив исходные данные

```
65Ø4Ø
В ЛЕСУ РОДИЛАСЬ ЕЛОЧКА,
В ЛЕСУ ОНА РОСЛА.
//␣ END
```

отпечатает две строчки

В ЛЕСУ РОДИЛАСЬ ЕЛОЧКА,
В ЛЕСУ ОНА РОСЛА.

и на ленте 65, начиная с зоны 40_8, образует файл из двух записей:

В ЛЕСУ РОДИЛАСЬ ЕЛОЧКА,
В ЛЕСУ ОНА РОСЛА.

по 80 объектных записей в каждой.

Машинные процедуры OPENG, GET, CLOSG предназначены для чтения данных из символьных файлов на магнитной ленте или барабане.

Перед тем как начать чтение данных из файла, нужно выполнить конкретизацию

K /OPENG / <адрес файла>.

где адрес файла указывается точно так же, как при обращении к функции OPEN. В качестве результата замены OPENG выдает пустое выражение.

Чтение данных из файла осуществляется функцией GET, обращение к которой имеет вид

K /GET /.

Результатом замены является очередная запись из файла, состоящая из 80 объектных знаков. При первом обращении к GET выдается первая запись, при втором – вторая и т.д. Если будут прочитаны все записи из файла, а мы все-таки обратимся к функции GET, то результатом замены будет пустое выражение. Если после этого еще раз обратиться к GET, то произойдет авост "отождествление невозможно".

По окончании чтения символьного файла нужно обратиться к функции CLOSG:

K /CLOSG /.

В качестве результата замены функция CLOSG выдает пустое выражение. Чтобы прекратить чтение файла, необязательно дожидаться, пока он кончится, т.е. к функции CLOSG можно обратиться в любой момент, если только перед этим уже было обращение к функции OPENG. После обращения к функции CLOSG вы можете снова обратиться к функции OPENG и прочитать еще один файл.

Пример. Рассмотрим рефал-программу, которой в качестве исходных данных подаются адреса нескольких файлов. Первый адрес – это адрес файла, который нужно сформировать. Остальные адреса – это адреса файлов, из которых берется информация для формирования нового файла. Содержимое нового файла представляет собой результат приписывания (конкатенацию) содержимых остальных файлов в том порядке, как они перечислены.

```
ЬКОНК   СТАРТ
        ВХОДН  КОНК
        ВНЕШН  OPEN, PUT, CLOSE
```

```
            ВНЕШН OPENG, GET, CLOSG
            ВНЕШН  КАРТА
     КОНК   = К/OPEN / К/АДРЕС/. . +
              К/КОНК1/ К/АДРЕС/. . +
              К/CLOSE /.
     КОНК1  =
            EA = К/OPENG / EA. +
                 К/ЗАП/ К/GET/. . К/CLOSG /. +
                 К/КОНК1/ К/АДРЕС/ . .
     ЗАП    =
            EX = К/СТЕР/ К/PUT / EX . . +
                 К/ЗАП/ К/GET / . .
     СТЕР   EX =
     АДРЕС  = К/АДРЕС1/ К/КАРТА/ . .
     АДРЕС1 =
            EA '␣' EB = EA
            ФИНИШ
```

Если теперь обратиться к этой программе, подложив в качестве исходных данных карты

```
    Ø5ØØØ
    67140
    671ØØ
    6532Ø
    //␣END
```

то программа сформирует файл на барабане 05, начиная с тракта 000_8. Содержимое этого файла будет составлено из содержимых трех файлов на лентах. Первые два из них находятся на ленте 67, начиная с зон 140_8 и 100_8, а третий файл на ленте 65, начиная с зоны 320_8.

С помощью описанных машинных процедур можно читать символьные файлы, сформированные другими программами мониторной системы "Дубна", например редактором текстов. Точно так же символьные файлы, сформированные машинными процедурами, воспринимаются другими программами мониторной системы "Дубна".

9. Обменные функции и символы-ссылки

Машинные процедуры ЗК, ВК, СЧ, ЗП обладают недостатком: чтобы найти закопанное выражение по его имени, нужно просмотреть копилку. Конечно, если закопано немного выражений, скажем, 10-15, то такой ассоциативный поиск занимает приемлемое время. Однако если мы хотим работать с несколькими сотнями или тысячами выражений, имеющих различные имена, то результат будет неудовлетворительным. А такие задачи встречаются. Например, задачи, связанные с обработкой графов, постро-

ением интерпретаторов языков операторного типа, задачи искусственного интеллекта. Здесь рассматриваются средства, обеспечивающие адресный доступ к информации.

В теоретическом описании рефала предполагалось, что рефал-машина состоит из двух запоминающих устройств: поля памяти, в котором находится набор предложений, и поля зрения. Кроме этих устройств, мы уже ввели дополнительное запоминающее устройство – копилку. Теперь будем считать, что имеется еще потенциально бесконечное множество запоминающих устройств, называемых *ящиками*. Каждый ящик содержит произвольное объектное выражение, которое может изменяться в процессе работы. Это выражение будем называть *содержимым ящика*.

Каждому ящику соответствует функция, с помощью которой можно получить доступ к содержимому ящика. Эти функции будем называть *обменными*. Таким образом, имеется взаимно-однозначное соответствие между множеством обменных функций и множеством ящиков.

Детерминатив обменной функции будем также называть именем соответствующего ей ящика.

Наглядно это можно изобразить так:

$$\mathcal{D} : \boxed{\mathcal{E}}$$

Здесь $\mathcal{D}$ – детерминатив обменной функции, а $\mathcal{E}$ – содержимое ящика.

Обменные функции работают следующим образом.

После выполнения конкретизации

$$\text{К}\,\mathcal{D}\,\mathcal{E}'.$$

в поле зрения останется выражение $\mathcal{E}$ – содержимое ящика с именем $\mathcal{D}$, а выражение $\mathcal{E}'$ окажется в ящике:

$$\mathcal{D} : \boxed{\mathcal{E}'}$$

Таким образом происходит обмен информацией между полем зрения и ящиком (откуда и произошло название обменных функций).

Все ящики делятся на *статические* и *динамические*. Статические ящики существуют в течение всего времени выполнения программы и не могут ни порождаться, ни уничтожаться во время работы. Напротив, динамические ящики порождаются только во время работы и могут уничтожаться.

Все статические ящики должны быть описаны в программе. Для этого используются карты ОБМЕН, которые выглядят так:

ОБМЕН метка-1, метка-2,...,метка - n

т.е., пропустив несколько позиций слева, записываете слово ОБМЕН, а затем через один или несколько пробелов перечисляете детерминативы обменных функций через запятую.

Таким образом, ящики описываются одновременно со своими обменными функциями. Перед началом работы программы все статические ящики содержат пустые выражения.

Вместо слова ОБМЕН можно использовать его латинские эквиваленты SWAP и SWOP.

Пример. Рассмотрим следующий фрагмент программы:

```
              ОБМЕН X1,X2
    ВАСЯ      =K/X1/'A'.K/X2/'B'.K/ПЕРЕ//X1/X2/.
    ПЕРЕ      SX SY=K SX K SY K SX...
```

В процессе выполнения конкретизации

K/ВАСЯ/.

в ящики /X1/ и /X2/ сначала занесутся символы 'A' и 'B' соответственно, а затем в результате выполнения конкретизации

K/ПЕРЕ/ /X1/ /X2/.

содержимое ящиков поменяется местами. То есть в ящике /X1/ окажется символ 'B', в ящике /X2/ – символ 'A', а в поле зрения останется пустое выражение.

Динамические ящики порождаются в процессе работы программы машинной процедурой NEW. В результате конкретизации

K /NEW/$\mathscr{E}$.

создается новый ящик и в него помещается выражение $\mathscr{E}$. Одновременно порождается новый символ $\mathscr{D}$, который остается в поле зрения в качестве результата замены. Символ $\mathscr{D}$ является детерминативом новой обменной функции, которая обеспечивает доступ к созданному ящику.

Имена динамических ящиков являются символами нового типа – символами-ссылками, в отличие от имен статических ящиков, которые являются обычными составными символами-метками. Символы-ссылки не имеют графического представления, что находится в полном согласии с тем, что их невозможно задать в программе. Тем не менее при отладке программы возникает необходимость как-то печатать символы-ссылки. Они печатаются в виде

/%*nnnnn*/

где *nnnnn* – пять восьмеричных цифр, обладающих следующими свойствами:

1) различным символам-ссылкам соответствуют различные числа;

2) один и тот же символ-ссылка имеет одно и то же число на протяжении одного запуска программы.

Эти загадочные свойства объясняются просто: восьмеричное число *nnnnn* является адресом ящика в памяти машины.

Пример. Рассмотрим следующий фрагмент программы, аналогичный предыдущему примеру:

```
    ВНЕШН NEW
    КОЛЯ = K/ПЕРЕ/ K/NEW/'A'.K/NEW/'B'..
    ПЕРЕ SX SY =K SXK SYK SX...
```

Выполним теперь конкретизацию

K/КОЛЯ/.

В результате двух обращений к функции NEW образуются два ящика с именами, имеющими, например, такое изображение: /%70613/ и

/%53Ø24/. Ящик /%7Ø613/ будет содержать символ 'A' , а ящик /%53Ø24/ - символ 'B' . Затем функция ПЕРЕ воспользуется символами-ссылками, оставленными в поле зрения, и поменяет местами содержимое этих ящиков.

Обратите внимание на любопытный факт: выполнение конкретизации

K/КОЛЯ/.

привело к появлению двух новых ящиков. Можно ли теперь как-нибудь извлечь содержимое из ящиков? Очевидно, что нельзя. Ни в поле зрения, ни в копилке, ни в других ящиках не сохранилось имен этих ящиков. Поэтому дальнейшая работа программы не изменится, если эти ящики уничтожить.

Ясно, что если обращаться к функции КОЛЯ много раз, то память рефал-машины будет забиваться ненужными ящиками. Конечно, для абстрактной рефал-машины это не беда, ибо ее память потенциально бесконечна, но конечная память БЭСМ-6 рано или поздно исчерпается.

Что же делать в этом случае? Очевидно, удалить ненужные ящики. Для этой цели в данную реализацию рефала встроен аппарат с б о р к и м у с о р а.

Сборка мусора автоматически запускается каждый раз, когда исчерпается свободная память. При этом обнаруживаются и удаляются все ящики, к которым нельзя добраться прямо или косвенно из поля зрения, копилки или статических ящиков.

На рис.8.1. изображены поле зрения, копилка и динамические ящики, пронумерованные цифрами от 1 до 8. Символы-ссылки показаны в виде

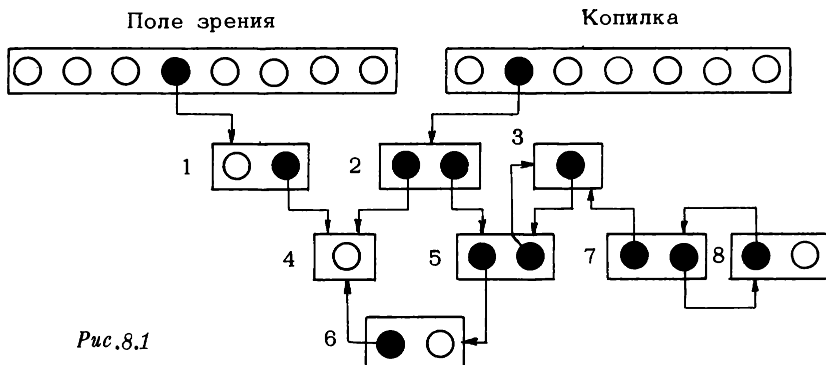


Рис.8.1

темных кружков. Видно, что по ссылке из поля зрения доступен ящик 1, а из него - ящик 4. Из копилки можно непосредственно добраться до ящика 2 и косвенно (через ящик 2) до ящиков 4,5, 6,3. Таким образом, нет способа извлечь информацию из ящиков 7 и 8. Если в этот момент запустить процесс сборки мусора, то ящики 7 и 8 будут уничтожены.

Обратите внимание, что если убрать символ-ссылку из поля зрения, то станет недоступным еще ящик 1. Если же символ-ссылку в поле зрения оставить, но убрать ссылку из копилки, то окажутся ненужными все ящики, кроме 1 и 4.

Опишем теперь пять машинных процедур, которые, хотя и не дают ничего принципиально нового, часто оказываются удобными.

Машинная процедура GTR (взять по ссылке) извлекает содержимое ящика. Результатом замены при конкретизации

K /GTR /$\mathcal{D}$.

где $\mathcal{D}$ – имя статического или динамического ящика, является содержимое ящика. После этого в ящике остается "пусто".

Машинная процедура RDR (прочитать по ссылке), как и функция GTR, , выдает содержимое ящика в поле зрения, но ящик при этом не меняется, т.е. происходит размножение информации.

Машинная процедура PTR (положить по ссылке) добавляет в ящик новую информацию. В результате конкретизации

K /PTR /$\mathcal{D}\mathcal{E}$.

где $\mathcal{D}$ – имя ящика, $\mathcal{E}$ – произвольное выражение, ящик меняется так:

$$\mathcal{E}_0 \to \mathcal{E}_0\mathcal{E}$$

где $\mathcal{E}_0$ – старое содержимое ящика. При этом в поле зрения ничего не остается.

Машинная процедура WTR (записать по ссылке) помещает в ящик новую информацию, при этом старая выкидывается. То есть при конкретизации

K /WTR / $\mathcal{D}\mathcal{E}$.

где $\mathcal{D}$ – имя ящика, $\mathcal{E}$ – произвольное выражение, ящик меняется так:

$$\mathcal{E}_0 \to \mathcal{E}$$

где $\mathcal{E}_0$ – старое содержимое ящика. Результатом замены является пустое выражение.

Машинная процедура SWR (обменять по ссылке) записывает в ящик новую информацию, а старую выдает в поле зрения, т.е. происходит такое преобразование:

поле зрения: K /SWR /$\mathcal{D}\mathcal{E}$. $\longrightarrow \mathcal{E}_0$
ящик: $\mathcal{E}_0 \longrightarrow \mathcal{E}$

где $\mathcal{D}$ – имя ящика, $\mathcal{E}$ и $\mathcal{E}_0$ – выражения.

Эти машинные процедуры можно было бы описать и на рефале следующим образом:

```
GTR SD = KSD.
RDR SD = K/RDR1/SDKSD. .
```

```
RDR1 SDE1 = E1KSDE1.
PTR  SDE1 = KSDKSD.E1.
WTR  SDE1 = K /WTR1/KSDE1..
WTR1 E1 =
SWR  SDE1 = KSDE1.
```

Надо сказать, что область определения этих функций, написанных на рефале, шире, чем у соответствующих машинных процедур, так как они не проверяют, что SD – имя ящика, и поэтому им можно подать в качестве SD детерминатив любой функции. А машинные процедуры проверяют, что SD– имя ящика.

10. Управление размером скомпилированной программы

Отдельной задаче на БЭСМ-6 предоставляется не более 32768 ячеек. Когда исполняется программа на рефале, то примерно 4000 ячеек занято программами административной системы (если используется прокрутка, то 6000 ячеек). В оставшейся памяти располагаются рефал-программа и поле зрения. Если вы напишете очень большую программу, то памяти, оставшейся под поле зрения, может оказаться недостаточно. Вы можете уменьшить объем скомпилированной программы в 3–4 раза за счет уменьшения скорости работы примерно в два раза. Рефал-компилятор может выдавать программу в двух видах: так называемом "упакованном" и "распакованном". Упакованный вид занимает меньше места, но требует дополнительных затрат времени при исполнении программы.

Для управления размером скомпилированной программы служат карты PACK и UNPACK. На этих картах записывается по одному слову PACK или UNPACK в любом месте во 2–71 позициях:

```
PACK
UNPACK
```

Карты PACK и UNPACK могут находиться в любом месте между описаниями функций. Встретив карту PACK , компилятор начинает выдавать следующие за ней функции в упакованном виде до тех пор, пока не встретит карту UNPACK. Встретив карту UNPACK, компилятор выдает следующие за ней функции в распакованном виде.

До того, как встретится первая карта PACK или UNPACK , компилятор выдает программу в распакованном виде, т.е. подразумеваемый режим UNPACK. Таким образом, можно выдать в распакованном виде те функции, которые часто работают, а на остальных функциях сэкономить память, выдав их в упакованном виде.

В упакованном виде одно предложение занимает в среднем 10–15 ячеек в распакованном – 40–60 ячеек. В поле зрения один символ или скобка занимает одну ячейку. Эта информация дает возможность оценить количество памяти, необходимое для работы рефал-программы.

IX. ЗАПУСК ПРОГРАММ НА МАШИНЕ «МИНСК-32»

Компиляция, загрузка и выполнение рефал-программ происходит под управлением мониторной системы СЕКАЧ. Сведения о мониторной системе СЕКАЧ можно почерпнуть из работы [23]. Здесь они будут предполагаться известными.

1. Общие сведения о рефал-системе

Процедура запуска задачи в рефал-системе состоит из трех основных функциональных частей:

- ввод текстов рефал-программ с перфокарт на магнитную ленту;
- компиляция программ в язык сборки с записью результата на магнитную ленту в виде загрузочных модулей;
- загрузка рабочих модулей программы и исполнение ее.

Каждый из этих трех этапов может быть оформлен как отдельным пакетом, так и в рамках одного общего пакета.

Инструкцию по выполнению программы ввода см. в работе [23]. Здесь будут подробно описаны этапы компиляции и исполнения программ.

2. Компиляция программ

Для трансляции программы, изображенной на рис.6.3, нужно составить такой пакет:

```
             //ЗАКР  СИБИБ; МОБИБ,2
             //ВЫП  ВКОРМ,ЛС
           ┌*** НБПРИМ
Этап       │ ↑↑↑↑↑                          ┐
ввода      │           СТАРТ                │
           │           ВНЕШН   КАРТА,П      │   Рефал-программа
           │           ВХОДН   ДЕЛО         │
           │           .                    │
           │           .                    │
           │           .                    │
           │           ФИНИШ                ┘
           │*** КБ
           └ КОНЕЦ

Этап       ┌//ЗГР   РФК,ЛС
компи-     │//ПАРМ  1,2
ляции     ─│//ПАРМ   ПРИМ, ПРИМР
           │//ИП
           └
```

На этапе ввода программа ПРИМ помещается на ленту символьных модулей СИБИБ. Затем загружается РФК. Он должен находиться на ленте системы, так же как и все остальные модули рефала.

На картах //ПАРМ компилятор получает информацию о транслируемых модулях. На первой карте – индексы исходной и результирующей МЛ через запятую. На каждой из последующих карт //ПАРМ – имя исходного текста (первый параметр) и имя результирующего модуля (второй параметр). Последнее можно опустить, если оно совпадает с первым.

На картах с именами модулей можно указывать третьим параметром цифрой от 0 до 7 количество распечаток транслируемого текста с сообщениями об ошибках, если они есть. Для получения одной распечатки этот параметр опускается.

Аналогично можно составить пакет только для распечатки исходных текстов с сообщениями об имеющихся в них ошибках. Для этого нужно опустить второй параметр на первой карте //ПАРМ (индекс МЛ результата).

3. Загрузка и выполнение программ

Для запуска оттранслированной программы ПРИМР нужно поместить такое продолжение пакета:

```
//ЗГР  РЕФАЛ,ЛС
//ПАРМ ПЧК
//ПАРМ ПРИМР,2
//ИП
ПО СПАР        ]  Карты задания на прокрутку
  *
 А(В*С) ()(С*В)А  ] Исходные данные
КОНЕЦ
//
```

Программа РЕФАЛ, загружаемая с ЛС, является одновременно и интерпретатором языка сборки и своеобразным монитором, управляющим работой рефал-программ. В лежащих вслед за картой //ЗГР картах //ПАРМ указывается первым параметром имя загружаемого модуля – машинной процедуры или модуля на языке сборки, вторым – индекс МЛ, с которой загружается модуль. Для стандартных машинных процедур, перечисленных в конце этой главы, второй параметр опускается. На картах //ПАРМ должны быть указаны имена всех модулей, участвующих в запуске. Совокупность указанных модулей образует поле памяти рефал-машины. Оно формируется перед началом интерпретации и в процессе ее изменяться не может. Для запуска рефал-машины с другим полем памяти следует повторить загрузку программы РЕФАЛ и передать ей новые параметры.

Загрузив все модули, перечисленные в картах //ПАРМ, программа РЕФАЛ редактирует внешние связи, формирует начальное поле зрения

К/ДЕЛО/,

и выходит в режим чтения своих управляющих карт.

Управляющие карты начинаются с имени приказа, далее – через один или несколько пробелов – следуют параметры, имеющие свой формат для каждого приказа. Пустая колонка и пробивка "␣" в управляющих картах не различаются. Несколько подряд идущих карт можно объединить в одну через разделитель ";". Так, например, карта

СО;ВХОД УВ; ПМ *

эквивалента последовательности из трех карт:

СО
ВХОД УВ
ПМ *

Исполнение каждого приказа происходит сразу же после его поступления.

Запуск рефал-программы на счет осуществляется после прочтения и выполнения всех приказов на карте, последним информационным символом которой является * (звездочка). Этот символ вы просто приписываете в конце карты, после которой хотите запустить программу. В середине информационной части (даже перед ";") ставить звездочку нельзя.

Программа РЕФАЛ может принимать приказы и с пультовой машинки ПМ. Находясь в этом режиме, она запрашивает оператора сообщением ЖДУ, на что оператор отвечает:

-ААААА; < текст управляющей карты > ◇

Такую директиву оператор может набрать и в процессе работы рефал-программы независимо от того, был ли ввод перед этим с ПМ или с ПК. Тогда происходит принудительное (по окончанию очередного шага) переключение в режим ввода с ПМ и обработка поступивших приказов. Исполнение рефал-программы будет продолжено после карты со звездочкой.

При появлении ошибки в управляющих картах всегда происходит переключение в режим ввода с ПМ.

Когда рефал-программа кончает свою работу, происходит переключение в режим ввода очередных приказов. При этом в случае нормального останова приказы будут поступать с того же устройства, что и раньше, а в случае аварийного – с ПМ.

Перейдем теперь к описанию приказов для программы РЕФАЛ.

Приказ ПМ задает переключение в режим ввода с ПМ.

Приказ ПК или ВК задает переключение в режим ввода с перфокарт.

Приказ ВХОД $\mathscr{D}$ задает начальный детерминатив. Параметр – метка, описанная в одном из модулей как входная. По этому приказу в поле зрения слева от находящегося в нем выражения вставляется терм:

$$K\mathscr{D}.$$

Если к моменту поступления этого приказа не было сделано еще ни одного шага рефал-машины, счетчик числа шагов равен 0, а в поле зрения – К/ДЕЛО/., то нового терма не формируется, а детерминатив $\mathscr{D}$ подставляется вместо детерминатива /ДЕЛО/.

Пример.

ВХОД БНОРМ

Приказ ШАГ задает печать текущего номера шага на ПМ и АЦПУ.

Приказ ПЗ задает распечатку поля зрения на АЦПУ.

Приказ ВТ задает распечатку ведущего терма $K\mathscr{E}.$ на АЦПУ.

Приказ ЗК задает распечатку всех выражений, закопанных при помощи функций ЗК и ЗП.

Приказ СО задает печать информации о состоянии рефал-машины. Он эквивалентен последовательности приказов:

ШАГ;ПЗ;ВТ;ЗК

Приказ СБРОС приводит рефал-машину в начальное состояние, аналогичное состоянию перед вводом первого приказа. Счетчик числа шагов сбрасывается на 0, все ранее поступившие условия на прокрутку (см.ниже) отменяются. В поле памяти – прежняя рефал-программа, в поле зрения – К/ДЕЛО/.

Для задания условий на прокрутку используйте следующие приказы.

Приказы С $\mathscr{N}$ и ДО $\mathscr{N}$ где $\mathscr{N}$ – десятичное число, задают условие "по шагам" (см.главу УI), соответственно начало и конец диапазона. До поступления этих приказов считается заданным диапазон от 0 до бесконечности.

Пример.

С 50;ДО 1000

Приказ СТОП $\mathscr{N}$ или СТОП $\mathscr{D}$ задают останов по номеру шага или по детерминативу. Здесь и ниже $\mathscr{N}$ – число, а $\mathscr{D}$ – произвольный детерминатив. При появлении ведущего терма с указанным детерминативом или при совпадении номера шага с указанным числом происходит переключение в состояние ввода управляющей информации с устройства, с которого происходил ввод ее в последний раз. Шаг, на котором произошел останов, завершается.

Приказ НА $\mathscr{N}$ задает продвижение на $\mathscr{N}$ шагов вперед. Вычисляется число $\mathscr{N}_1$ как сумма параметра приказа и текущего номера шага, после чего выполняется последовательность приказов

ДО $\mathscr{N}_1$;СТОП $\mathscr{N}_1$

Пример.

НА 15

Приказ ПО $\mathcal{D}$ задает условие "по детерминативу"(см.главу У1).

Приказ ПО $<\mathcal{D}$ задает условие "< ".

Приказ ПО $>\mathcal{D}$ задает условие " >",

Прокрутка будет "включена", если поступит хотя бы один из приказов с именами С, ДО, НА, СТОП, ПО. Поэтому, например, для организации полной прокрутки достаточно задать условие

С 1

Каждый из приказов СТОП по номеру шага, С, ДО отменяет предыдущий приказ с таким же именем. Приказы ПО можно отменить приказом УПО.

Для того чтобы временно отключить прокрутку, а затем вновь восстановить все условия, используйте соответственно приказы ОТО и ВКЛО.

Пустой приказ игнорируется. Однако если он был с ПМ, когда рефал-машина работала (например, вы набрали директиву -ААААА;◊), то происходит прерывание с переключением в режим ввода с ПМ.

По приказу КОНЕЦ управление возвращается монитору. Теперь вы можете снова вызвать программу РЕФАЛ и продолжить работу с другим полем памяти.

4. Библиотека машинных процедур

Все машинные процедуры, описанные в главе У1, реализованы в нашей системе без изменений. Они хранятся на ленте системы в следующих модулях:

модуль ЗКВК - ЗК, ВК, ЗП, СЧ, ВКВСЕ
модуль АРИФМ - SYMB, NUMB, ADD, SUB, MUL, DR
модуль ВСПМ - TYPE, FIRST, LAST, MULTE, LENGW, LENGR

модуль ПЧК - КАРТА, П, ПЕЧ

X. ЗАПУСК ПРОГРАММ НА МАШИНАХ ТИПА М-220

Настоящая глава содержит минимум сведений о рефал-системе РС4/220, которые необходимы для правильной подготовки рефал-программы и выполнения ее на любой из машин указанного типа.

Основные отличия РС4/220 от реализаций рефала на ЭВМ БЭСМ-6, "Минск-32", ЕС ЭВМ заключаются в следующем:

- при вводе рефал-программы в машину используется метакод-А;
- начальное поле зрения допускает любую структуру вложенности конкретизаций.

1. Перфокарта. Бланк

В машинах типа М-220 ввод перфокарт производится по широкой стороне, т.е. строками. Каждая строка, содержащая информацию, должна обязательно иметь пробивку в 18-й позиции (так называемый маркер). Перфорация карт для рассматриваемого класса машин производится на устройствах подготовки перфокарт (УПП) в коде ГОСТ. На одной строке перфокарты может быть до 6 символов. Всего на перфокарте может разместиться до 12 строк, т.е. на одной перфокарте можно набить текст, не превышающий 72-х символов.

Для набивки рефал-программы, начального поля зрения, а также управляющих директив системы необходимо записывать нужный текст на бланках, разграфленных на строки по 72 позиции в каждой. Одна строка бланка соответствует одной перфокарте.

Необходимо помнить, что в РС4/220 каждая перфокарта, содержащая информацию, должна иметь ровно 12 маркеров (т.е. на каждой строке). Если даже текст на данной перфокарте заканчивается раньше, например на четвертой строке, нужно пробить маркеры на всех остальных строках этой перфокарты, нажимая для этого клавишу "Исполнение" на УПП.

Любое рефал-предложение можно записывать на произвольном количестве строк бланка, т.е. набивать на одной или нескольких перфокартах. Каждое предложение можно начинать с любой позиции в строке. Разрешается прервать запись предложения в любой позиции и продолжить с прерванного места в любой позиции следующей строки. Дело в том, что при анализе каждой перфокарты в системе РС4/220 происходит удаление незначащих нулей и пробелов слева и справа от текстовой информации. Заметим, что под символом "пробел" здесь и далее в этой главе понимается символ ␣ ("корыто"), имеющий на УПП код 017_8. Другие коды пробела (например, 176_8) в РС4/220 не допускаются. Так сделано потому, что некоторые типы алфавитно-цифровых печатающих устройств (АЦПУ) такие коды воспринимают как признаки прогона бумаги. Слева от текстовой информации удаляются все нули или пробелы до первого отличного от них символа. Справа удаляются все подряд идущие нули до первого ненулевого символа, но если этот символ есть пробел, то он тоже удаляется. Таким образом, если информация на перфокарте оканчивается значащим пробелом, необходимо отперфорировать его дважды, если же информация оканчивается значащим нулем, после него нужно набить один пробел.

Признаком конца рефал-программы является перфокарта, на которой набиты два подряд идущих ромба.

Большая рефал-программа должна быть разбита на порции, не превышающие 170 перфокарт. Признаком конца порции является перфокарта, в первой позиции которой пробит символ / (слеш).

Правила перфорации начального поля зрения совпадают с правилами перфорации рефал-программ. Отличие состоит лишь в том, что признаком конца поля зрения является карта, в первых позициях которой пробиты два символа /*.

Разбиение начального поля зрения на порции не допускается.

Правила перфорации управляющих директив будут описаны ниже, после описания языка управления заданиями.

2. Метакод-А

В системе РС4/220 для идентификации начала рефал-предложения используется символ "ромб" (вместо символа "параграф", которого нет в числе кодов АЦПУ).

В метакоде-А для выделения собственных знаков рефала используется специальный символ - надчеркивание. Надчеркивание используется для выделения:

- свободных переменных (например, ¯S1, ¯WA, ¯E2 и т.д.);
- конкретизационных скобок ¯K и ¯.;
- знака, разделяющего левую и правую части рефал-предложения ¯>.

Скобки записываются одним символом: левая - (, а правая -).

В метакоде-А все объектные знаки делятся на три группы:

1. Простые символы - это буквы, цифры, а также символы +, -, x, * /, ', · , ␣ , 10, ↑ , = , ; , [,] , ≠, < ,> , :
Никакие другие символы в РС4/220 не допускаются.

2. Составные символы - это любая последовательность простых символов, заключенная в апострофы (в РС4/220 роль апострофов выполняют открывающая и закрывающая кавычки, которые для системы равнозначны). Составной символ не должен включать более чем 50 символов (считая и апострофы), причем составной символ только из двух апострофов не допускается. Примеры составных символов :

'АЛЬФА' , 'ДЛИННЫЙ СИМВОЛ', ' ↑↑↑ВЕТА↑↑↑ ' и т.д.

3. Символы - числа или макроцифры - это натуральные десятичные числа, заключенные в апострофы: '142 ', 'Ø' и т.п. Одна макроцифра не может быть больше величины $2^{18}-1$ (262143). Если требуется записать с помощью макроцифр число большее, чем $2^{18}-1$, то необходимо перевести его в систему счисления с основанием 2^{18}, например число $262144=2^{18}$ нужно записывать так: '1' '0' - т.е. две макроцифры.

В общем виде рефал-предложение в метакоде-А выглядит следующим образом:

◊ <комментарий> <левая часть> ¯> <правая часть>

<комментарий> - это любая последовательность простых символов. Здесь, например, можно указывать номер рефал-предложения, некоторые пояснения к данному предложению и т.п.

В отличие от вышеописанных реализаций рефала на ЭВМ БЭСМ-6, "Минск-32" и ЕС ЭВМ, использующих метакод-Б, в метакоде-А внутри рефал-предложений не может быть никаких пропусков. В этом случае пропуски будут восприниматься как объектные знаки: нули или пробелы.

В отличие от метакода-Б в метакоде-А левая часть каждого рефал-предложения должна начинаться со знака конкретизации ¯К и детерминатива. Детерминатив в РС4/220 - это последовательность не более чем из пяти букв и цифр, начинающаяся с буквы и ограниченная апострофами.

В качестве примера напишем в метакоде-А программу, определяющую, является ли исследуемая строка фразой-перевертышем (такое название имеют фразы естественного языка, сохраняющие свое значение при чтении их справа налево.).

```
◊1 ¯K'ПЕРЕВ'¯ >ДА
◊2 ¯K'ПЕРЕВ'¯SA¯>ДА
◊3 ¯K'ПЕРЕВ'¯S1¯EX¯S1¯>¯K'ПЕРЕВ'
◊4 ¯K'ПЕРЕВ' ¯E1¯>НЕТ
◊◊
```

В РС4/220 допускаются строки-комментарии, которые идентифицируются символом * в первой позиции строки. Строка-комментарий может содержать любые конструкции языка рефал.

Перечислим некоторые ограничения, которые накладываются на рефал-программу в РС4/220:

1. Детерминатив не может быть пустым (состоять только из двух апострофов).

2. На длину левой части предложения накладывается такое ограничение: если n -количество скобок, объектных знаков, составных символов и вхождений переменной символа в левой части, а m - количество вхождений переменных терма и выражения, то должно выполняться условие:

$$n + 2m < 128$$

3. В качестве идентификаторов свободных переменных можно использовать только буквы и цифры.

4. В рефал-программе должно быть не более 256 различных детерминативов.

В РС4/220 поле зрения не должно превышать 4000 объектных знаков.

3. Управление заданиями

Решение задачи в системе РС4/220 состоит из нескольких этапов:

- проверки синтаксической правильности рефал-программы и начального поля зрения;
- компиляции программы на язык сборки;
- собственно выполнения скомпилированной программы.

По желанию пользователя некоторые этапы работы могут опускаться.

Для автоматизации управления прохождением задачи в системе РС4/220 используется простой язык управления заданиями, который представляет собой набор определенных директив. В настоящее время используются следующие директивы: РАБОТА, ВЫПОЛНИТЬ, ДАННЫЕ, ПРОКРУТКА, МАШИННЫЕ ОПЕРАЦИИ, КОНЕЦ РАБОТЫ.

Директива РАБОТА открывает пакет управляющих директив и служит для идентификации задания. Формат этой директивы следующий:

//РАБОТА <комментарий>

Комментарий является произвольным текстом, который не должен превосходить 60 символов. Обычно в комментарии указывается название программы, фамилия автора, дата написания и т.п.

Директива ВЫПОЛНИТЬ предназначена для организации последовательного выполнения требующихся модулей рефал-системы и имеет следующий формат:

//ВЫП␣ <имя модуля>

где <имя модуля>: := БСК.А␣|КОМП␣␣| ИНТЯС ␣ |РЕФАЛ␣|
ЯУЗА

Использование параметра <имя модуля> позволяет пользователю вызывать в оперативную память и исполнять как отдельные модули рефал-системы (блок синтаксического контроля, компилятор, интерпретатор языка сборки), так и последовательность модулей (РЕФАЛ). Использование в качестве названия модуля имени РЕФАЛ позволяет с помощью одной директивы задать такую последовательность выполнения модулей: БСК.А, КОМП, ИНТЯС. Употребление директивы //ВЫП␣ РЕФАЛ␣ удобно при работе с отлаженной рефал-программой.

Директива ДАННЫЕ предназначается для описания устройств, на которых подготовлена входная информация для работы данного модуля рефал-системы, а также устройств, на которые будет выдаваться результат работы этого модуля. Директива имеет следующие форматы:

//ДАН ␣ ВХПР␣␣<входной носитель>
//ДАН ␣ ВЫХПР␣ <выходной носитель>
//ДАН ␣ ВХПЗ␣␣ <входной носитель>
//ДАН ␣ ВЫХПЗ␣ <выходной носитель>

где <входной носитель> : :=ЧУ␣␣␣␣|МЛАА. ␣ ББББ␣␣
<выходной носитель> : :=ПФ␣␣␣␣|АЦПУ␣␣ |МЛАА. ␣ББББ␣␣

ЧУ означает, что данные подготовлены на перфокартах;
ПФ – информация будет выдана на перфокарты;
АЦПУ – информация будет распечатана; в описании носителя магнитной ленты (МЛ) использованы обозначения:

АА – программный номер магнитофона (например, 03, 01 и т.д.);

ББББ – восьмеричный номер зоны на ленте, начиная с которой записана информация (например, 1022). Номер зоны обязательно должен быть четырехзначным числом, т.е. использование зон с номерами $< 1000_8$ не допускается.

Параметры ВХПР и ВЫХПР дают системе информацию о входных и выходных носителях для рефал-программы, а параметры ВХПЗ и ВЫХПЗ – о входных и выходных носителях для поля зрения.

Для модуля КОМП директивы ДАННЫЕ с параметрами ВХПЗ и ВЫХПЗ не имеют смысла и поэтому не допускаются. Для модуля ИНТЯС не имеет смысла директива ДАННЫЕ с параметром ВЫХПР.

Необходимо подчеркнуть, что рефал-программа может подаваться на вход блока КОМП только после прохождения блока БСК.А. Дело в том, что блок БСК.А, кроме проверки синтаксической правильности программы, производит некоторые преобразования ее с целью подготовки к компиляции. Можно, конечно, запомнить результат работы блока БСК.А на каком-нибудь внешнем носителе, например, на магнитной ленте, а потом начинать работу непосредственно с блока КОМП, указывая в директиве //ДАН␣ВХПР тот носитель, на котором записан результат работы (ВЫХПР) блока БСК.А.

Таким образом, после каждой директивы ВЫПОЛНИТЬ может следовать до четырех директив ДАННЫЕ. Если какая-либо из директив ДАННЫЕ отсутствует, то система использует в качестве носителя соответствующей этой директиве информации магнитный барабан. Кроме того, на магнитный барабан всегда выводятся результаты работы любого блока, независимо от того, какие носители описаны в качестве выходных.

При описании выходных носителей в одной директиве ДАННЫЕ может быть указано несколько устройств, при этом носитель на магнитной ленте должен указываться в такой директиве последним.

При употреблении директивы //ВЫП␣*ЯУЗА* на АЦПУ выдается инструкция по использованию языка управления заданиями в РС4/220. После этой директивы ставить директивы ДАННЫЕ не нужно.

С помощью директивы МАШИННЫЕ ОПЕРАЦИИ пользователь сообщает системе, используются ли данной задачей личные машинные операции, а также определяется их носитель. Формат этой директивы следующий:

//МОП␣ЛИЧН :␣<входной носитель>

<входной носитель> описывается так же, как было сказано выше.

Директива ПРОКРУТКА используется для задания режима отладки рефал-программы. Она имеет два формата:

//ПРN␣<параметр>

//ПРD␣<параметр>

N означает прокрутку по номеру шага; D – по детерминативу.

Если параметр пустой, то происходит выполнение рефал-программы с выдачей на АЦПУ информации о выполнении каждого шага (печатается номер шага и конкретизируемое выражение из поля зрения).

Если параметр имеет вид NNNNNN ,то происходит прокрутка, начиная с шага с таким номером (например, параметр 000010 означает, что печать начнется с десятого шага работы рефал-программы).

Если параметр имеет вид; ,KKKKK ␣ , то прокрутка будет вестись с первого шага до шага с номером KKKKK.

Наконец, если параметр имеет вид NNNNN, KKKKK␣, то прокрутка будет происходить, начиная с шага с номером NNNNN и закончится на шаге с номером KKKKK.

То же относится и к прокрутке по детерминативам, только в этом случае указываются не номера шагов, а детерминативы рефал-предложений. При этом детерминатив пишется без апострофов, прижимается влево в соответствующем формате параметра, а недостающие позиции заполняются нулями. Например, если требуется произвести прокрутку программы, начиная с детерминатива 'A', до выполнения рефал-предложений с детерминативом 'BETA', то необходимо дать следующую директиву:

//ПРD␣A0000, BETA0␣

Заметим, что при таком режиме прокрутки, если после того как отработает рефал-предложение с детерминативом 'BETA' в процессе дальнейшей работы программы внсвь будет работать предложение с детерминативом 'A', то снова начнется прокрутка программы до выполнения рефал-предложения с детерминативом 'BETA' и т.д. Это можно использовать для исследования работы определенной функции, указывая в директиве в качестве начального и конечного один и тот же детерминатив, при этом будет происходить печать конкретизируемого выражения на каждом шаге работы предложений с данным детерминативом.

Директива КОНЕЦ РАБОТЫ закрывает пакет управляющих директив. Ее формат:

/.

При составлении пакета управлящих директив нужно помнить следующее:

- пакет начинается директивой РАБОТА;

- после директивы РАБОТА, в случае использования личных машинных операций, следует директива МАШИННЫЕ ОПЕРАЦИИ;

- директива ПРОКРУТКА ставится после директивы МАШИННЫЕ ОПЕРАЦИИ, а в случае отсутствия последней - после директивы РАБОТА;

- остальная часть пакета управляющих директив вплоть до директивы КОНЕЦ РАБОТЫ представляет собой пункты задания, каждый из которых имеет такую структуру:

1) директива ВЫПОЛНИТЬ, указывающая наименование блока рефал-системы;

2) до четырех директив ДАННЫЕ, указывающих устройства для входной и выходной информации.

В РС4/220 производится контроль пакета управляющих директив. Пакет директив всегда распечатывается. В случае обнаружения ошибок в управляющих директивах анализ их прекращается, оставшиеся директивы распечатываются, но не анализируются, выдается сообщение о типе ошибки и работа системы прекращается.

Для правильной перфорации управляющих директив необходимо выполнение правил:

- каждая директива набивается на отдельной перфокарте;
- каждая директива должна начинаться с первой позиции;
- перфокарты с управляющими директивами должны иметь ровно по 12 маркеров, т.е. на каждой строке перфокарты.

Пример 1.

Пусть нужно выполнить такую задачу:

- проверить синтаксическую правильность рефал-программы и начального поля зрения, подготовленных на перфокартах;
- скомпилировать эту программу и результат компиляции запомнить на магнитной ленте, начиная с 1000_8 зоны;
- выполнить скомпилированную программу с начальным полем зрения;
- результаты работы по каждому пункту распечатывать на АЦПУ.

Такой задаче соответствует пакет управляющих директив:

```
//РАБОТА ПРИМЕР ПАКЕТА ДИРЕКТИВ
//ВЫП␣БСК.А␣
//ДАН␣ВХПР␣␣ЧУ␣␣␣␣
//ДАН␣ВЫХПР␣АЦПУ␣␣
//ДАН␣ВХПЗ␣␣ЧУ␣␣␣␣
//ДАН␣ВЫХПЗ␣АЦПУ␣␣
//ВЫП␣КОМП␣␣
//ДАН␣ВЫХПР␣АЦПУ␣␣МЛ03.␣1000␣␣
//ВЫП␣ИНТЯС␣
//ДАН␣ВЫХПЗ␣АЦПУ␣␣
/.
```

Отсутствие директивы //ДАН␣ВХПР в пункте задания //ВЫП ␣КОМП объясняется тем, что перед ним описан пункт задания БСК.А, результат работы которого как раз является информацией ВХПР для КОМП. Но выше было сказано, что результат работы любого блока всегда записывается на магнитный барабан. Так как носитель ВХПР для блока КОМП не указан, что система автоматически примет в качестве него магнитный барабан, где как раз и находится нужная информация.

По этой же причине в пункте задания //ВЫП␣ ИНТЯС отсутствует директива //ДАН␣ ВХПР, потому что результат работы блока КОМП является информацией ВХПР для блока ИНТЯС.

Директива //ДАН␣ ВХПЗ может появляться либо на этапе работы БСК.А, либо ИНТЯС. Если начальное поле зрения вводится на этапе БСК.А, то оно запоминается на магнитном барабане и может использоваться на этапе ИНТЯС. В этом случае директиву //ДАН␣ ВХПЗ в пункте задания //ВЫП␣ ИНТЯС ставить не нужно - система автоматически считает начальное поле зрения с магнитного барабана.

Пример 2.

Допустим, что требуется рефал-программу, которая использовалась в предыдущем примере, пропустить с другим начальным полем зрения, подготовленным на перфокартах.

Используем то обстоятельство, что скомпилированная программа записана на магнитной ленте - ее не нужно будет проверять и компилировать заново.

Пакет директив может быть таким:

```
//РАБОТА ПРИМЕР
//ВЫП␣ИНТЯС␣␣
//ДАН␣ ВХПР␣␣ МЛ03.␣1000␣␣
//ДАН␣ ВХПЗ␣␣ ЧУ␣␣␣␣
//ДАН␣ ВЫХПЗ␣ АЦПУ␣␣
/.
```

Обратите внимание, что в этом примере не используется БСК.А, поэтому необходимо указывать носитель для начального поля зрения ВХПЗ.

4. Библиотека машинных процедур

РС4/220 располагает библиотекой машинных процедур (МП), которые делятся (конечно, условно) на системные и личные. К системным относятся те МП, которые широко используются в различных задачах. Они поставляются вместе со всей системой и доступны всем пользователям данного экземпляра системы.

При необходимости каждый пользователь может заводить свои личные МП, которыми библиотека дополняется только на время решения его задачи. Существует и такой режим работы системы, когда личные МП включаются в систему и остаются в ней после решения задачи (т.е. становятся системными). Это дает возможность пользователю расширять библиотеку системных МО и компоновать ее по своему усмотрению.

Детерминативы машинных операций нельзя использовать в качестве детерминативов рефал-предложений. Это означает, что, работая

с определенной версией системы РС4/220, нужно знать, какие детерминативы зарезервированы в этой версии как системные машинные операции и не использовать их в другом смысле. Ниже описаны системные машинные операции, которые существуют в РС4/220 к настоящему времени.

5. Машинные процедуры печати и перфорации

¯К' ВЫВ'ℰ¯.

Эта МП выводит на АЦПУ выражение ℰ, располагая его в строки, состоящие из 114 символов. После выполнения этой МО выражение ℰ в поле зрения не остается. Составные символы и макроцифры печатаются в кавычках.

¯К' ПЧ'ℰ¯.

Эта МП подобна предыдущей, но после ее выполнения выражение остается без изменений в поле зрения.

¯К'ПЧЧ'ℰ¯.

Эта МП аналогична 'ПЧ', т.е. тоже оставляет выражение ℰ в поле зрения, а отличается она от 'ПЧ' тем, что при печати нескольких подряд идущих макроцифр в этой МП производится перевод чисел из 2^{18}-ичной системы счисления в десятичную.

Например, если в поле зрения появится:

¯К'ПЧЧ''1' '2'¯.

то АЦПУ отпечатает:

'262146' (так как 2^{18}x1+2=262146),

а в поле зрения останутся две макроцифры:

'1' '2'

Обращение к МП перфорации имеет вид:

¯К'ПФ'ℰ¯.

Перфорация происходит по широкой стороне по 6 символов в строке. Составные символы и макроцифры перфорируются с кавычками. Если число перфорируемых символов не кратно 6, то последняя перфорируемая строка дополняется нулями.

6. Машинные процедуры ЗК, ВК, ПАМ

Обращение к МП закапывания имеет вид:

¯К 'ЗК' <имя> = ℰ¯.

где <имя> - имя закапывания; это любой символ (может быть составной или макроцифра).

При повторном обращении к закапыванию по тому же самому имени произойдет затирание информации, закопанной ранее.

Для того чтобы затирания закопанной информации не происходило, можно воспользоваться таким обращением:

¯К 'ЗК' <имя> ℰ¯.

При этом выражение $\mathcal{E}$ будет "прикопано" (приписано справа) к выражению, закопанному по данному имени ранее.

После выполнения МП закапывания на месте обращения к ней в поле зрения ничего не остается.

Общее количество имен, по которым в данный момент есть закопанная информация, не должно превышать 20_8, в противном случае работа системы прекращается и выдается сообщение:

21 ИМЯ ЗАКАПЫВАНИЯ

Обращение к МП выкапывания следующее:

-К 'ВК' <имя> -.

После его выполнения в поле зрения появится выкопанное выражение. Если ранее по данному имени ничего не закапывалось, то будет выкопано пустое выражение.

МП обмена с магнитным барабаном имеет следующий формат обращения:

-К 'ПАМ' <число страниц> <имя> <знак> $\mathcal{E}$-.

где <число страниц> - это макроцифра;

<имя>- последовательность не более чем из 5 символов, не начинающаяся с символа * и не содержащая внутри себя символов = или ≠ (могут употребляться составные символы и макроцифры);

<знак> = означает запись на барабан со стиранием информации, записанной по данному имени ранее;

<знак> ≠ означает приписывание информации справа к тому, что было записано по этому имени ранее.

Информация записывается на магнитный барабан в упакованном виде страницами по 200_8 кодов.

При первом обращении к МП 'ПАМ' с данным именем для этого имени будет отведено на барабане указанное число страниц. При повторных обращениях информация о числе страниц игнорируется. Если в первом обращении к МП 'ПАМ' с данными именем число страниц не указано, то этому имени будет отведена одна страница на барабане.

Считывание информации с магнитного барабана в поле зрения производится по обращению:

-К 'ПАМ' <имя> -.

при этом информация на барабане сохраняется.

Для того чтобы исключить имя из списка задействованных имен, и тем самым освободить занимаемые страницы на барабане, нужно воспользоваться обращением:

-К 'ПАМ *<имя> -.

Всего в распоряжении пользователя имеется 52 страницы.

Если при обращении на запись по какому-либо новому имени нет того числа подряд идущих страниц на барабане, которое затребова-

но в обращении с этим именем, то работа системы прекращается и выдается сообщение:

НЕТ ПАМЯТИ ДЛЯ: <имя>

Если информация, записываемая по данному имени не помещается в отведенной для этого имени памяти, то работа системы прекращается и выдается сообщение:

ВЕЛИКА ИНФ. В ПАМ: <имя>

7. Машинные процедуры арифметики

В РС4/220 имеются две арифметические машинные процедуры

1) 'ВЧСЛ' - МП целочисленной арифметики, работающая с числами, представленными в виде последовательности цифр - простых символов;

2) 'ВЧС' - МП рациональной арифметики, работающая с числами, представленными в виде последовательности макроцифр.

Обращение к МП целочисленной арифметики следующее:

¯К'ВЧСЛ' <знак> (<1 число>) <2 число> ¯.

знак "+" означает сложение; "-" - вычитание; "х"-умножение; "/" - деление (выполняется нацело с избытком).

Число должно состоять из одного или нескольких простых символов, идущих подряд (например, 1124 или 8) и не должно превышать 2^{18}=262144. Числу может предшествовать знак плюс или минус (при отсутствии знака число считается положительным). Число может быть представлено также одной макроцифрой (со знаком или без него).

Результат арифметической операции остается в поле зрения на месте обращения к этой МП и представляет собой последовательность цифр - простых символов. Результат любой арифметической операции по модулю не должен превосходить 2^{18}= 262144. Если результат отрицательный, ему предшествует знак минус, у положительных и нулевых результатов знак не ставится. При делении на нуль работа системы прекращается и выдается сообщение:

ВЫЧСЛ: ДЕЛЕНИЕ НА НУЛЬ

Примеры выполнения МП 'ВЧСЛ':

¯К 'ВЧСЛ'+ (-20)+'35' ¯. результат: 15
¯К 'ВЧСЛ'х (20)0¯. результат: 0
¯К 'ВЧСЛ'/ (-'20') 60¯. результат: 0
¯К 'ВЧСЛ'- (-20) 35¯. результат: -55

Обращение к МП рациональной арифметики следующее:

¯К 'ВЧС' <знак> (<1 число>) <2 число> ¯.

<знак> имеет тот смысл, что и в МП 'ВЧСЛ', но деление в МП 'ВЧС' происходит без округления.

Результатом любой операции является несократимая дробь со знаком или целое число со знаком (без знака выдается только нулевой результат).

Общий вид числа, допустимого в качестве операнда этой МП, следующий:

[<знак>] < макроцифры > [/< макроцифры>]

(здесь в квадратные скобки заключены конструкции, которые не являются обязательными).

Макроцифры до символа слеш (/) образуют числитель дроби, и макроцифры после символа слеш - знаменатель.

Таким образом, число может быть либо целым, либо рациональной дробью.

При делении на нуль работа системы прекращается и выдается сообщение:

ДЕЛИМ НА Ø

Как операнды, так и результат в МП 'ВЧС' не должны превосходить по модулю величины 2^{720}, т.е. около 10^{220}.

Примеры выполнения МО 'ВЧС':

```
¯К'ВЧС'+ (-'2Ø')+'35'¯.        результат: + '15'
¯К'ВЧС'х ('2Ø')'Ø'¯.           результат:   'Ø'
¯К'ВЧС'/ (-'2Ø')'6Ø'¯.         результат: -'1' / '3'
¯К'ВЧС'/ (-'2Ø'/'7') -.
          -'5'/'3'¯.           результат: +'12'/'7'
¯К'ВЧС'х ('1Ø24')'256'¯        результат: +'1''Ø'
```

(в последнем примере результат = 262144, т.е. 2^{18}, а это число представляется уже двумя макроцифрами).

8. Машинные процедуры сравнения, обмена и счетчик

МП сравнения чисел, записанных в виде последовательности макроцифр, имеет следующий формат обращения:

¯К'СР'(<1 число>) <2 число> ¯.

Число может задаваться как со знаком, так и без него, в последнем случае оно считается положительным.

Результат действия МП 'СР':

<знак> (<1 число>) <2 число>

Числа в результате остаются теми же самыми, которые были в обращении в этой МП. Знак + ставится, если первое число больше второго (с учетом знака числа, конечно); знак - ставится, если второе число больше; знак = ставится, если оба числа равны.

Использование в этой МП чисел, заданных в виде последовательности простых символов (цифр) не допускается, т.е. недопустима, например, такая запись:

¯К'СР'(9)-188¯.

в то время как обращение:

¯К'СР'('9') -'188'¯.

правильно.

В РС4/220 имеется машинная процедура посимвольного сравнения. Обращение к ней в общем виде следующее:

К'СРН'(<1 группа символов>) <2 группа символов> ¯.
Результат действия этой МП:

<знак> (<1 группа символов>) <2 группа символов >

Сравнение символов происходит следующим образом.

Коды крайних справа символов обеих групп сравниваются между собой, при этом запоминается знак +, если код символа из первой группы больше (как восьмеричное число) символа из второй группы, знак минус - если наоборот. Затем точно так же сравниваются следующие символы из обеих групп, т.е. сравнение идет посимвольно справа налево. Если в результате такого сравнения двух каких-то символов выяснится, что они одинаковы, то сохраняется знак плюс или минус, который был получен на предыдущем шаге.

Если число символов в обеих группах одинаково, то в результате в поле зрения ставится последний полученный знак.

Если в группах разное число символов, то, независимо от самих символов, принимается, что больше та группа, в которой больше символов.

Знак = в результате ставится тогда и только тогда, когда обе группы символов совпадают.

МП 'СРН' применяется для сравнения целых положительных чисел, записанных в виде последовательности простых символов (цифр).

Примеры выполнения МП 'СРН':

¯К'СРН'(123)5¯. результат: +(123)5
¯К'СРН'(2А)3А¯. результат: -(2А)3А
¯К'СРН'(12АБ)12АБ¯. результат: =(12АБ)12АБ

В РС4/220 имеется МП, позволяющая заменять выполняемую рефал-программу на другую, предварительно скомпилированную и записанную на магнитную ленту. Эта МП используется для совместной отладки отдельно написанных рефал-программ, а также при работе с большими программами - когда не хватает места для размещения всей программы на магнитном барабане и программа поэтому разбивается на отдельные модули. С помощью МП 'ОБМЕН' осуществляется связь между модулями в процессе их работы.

Обращение в общем виде следующее:

¯К 'ОБМЕН' ℰ (<детерминатив> : <№ МЛ> <№ зоны>) ¯.

В результате выполнения МП 'ОБМЕН'с указанного магнитофона, начиная с заданной зоны, будет считана и записана на магнитный барабан рефал-программа (которая, естественно, должна была быть заранее скомпилирована на ленту), а информация ℰ окажется в области действия указанного детерминатива.

Например:

¯К 'ОБМЕН' АБВ (СИГМА: 3.1000)¯.

(Обратите внимание, что детерминатив записывается в обращении к МП без апострофов).

В результате выполнения такого обращения, начиная с зоны 1000_8 магнитной ленты, находящейся на магнитофоне с программным номером 3, на магнитный барабан будет считана рефал-программа, а следующим шагом рефал-машины будет выполнение конкретизации:

¯К 'СИГМА' АБВ¯.

Очевидно, что считанная рефал-программа должна иметь рефал-предложения с детерминативом 'СИГМА'.

Машинная операция 'СЧЕТЧИК' имеет следующий формат обращения:

¯К 'СЧ' <имя> <знак> <макроцифра> ¯.

где имя - это простой или составной символ, имя счетчика (может быть употреблена здесь и макроцифра);

знак указывает операцию, совершаемую со счетчиком:

"=" означает присваивание счетчику значения (с затиранием того, что было там раньше);

"+" означает сложение значения счетчика с данными именем с числом, задаваемым макроцифрой;

"-" означает вычитание из значения счетчика указанного числа;

"x" означает умножение значения счетчика на указанное число.

Для правильной работы счетчика необходимо, чтобы число, присваиваиваемое счетчику или получаемое в результате арифметической операции с содержимым счетчика, не превосходило величины 2^{18}= = 262144.

Для считывания значения счетчика в поле зрения необходимо обращение:

¯К 'СЧ' <имя> ¯.

При этом в поле зрения в виде макроцифры появится значение, которое в данный момент имеет счетчик с указанным именем.

Заметим, что числа от Ø до 9 можно задавать как макроцифрами, так и простыми символами - цифрами.

Например, в результате выполнения конкретизаций:

¯К 'СЧ'А=9¯. ¯К 'СЧ'Аx5Ø'¯. ¯К'СЧ'А¯.

в поле зрения появится макроцифра '45Ø', при этом значение счетчика А сохранится и может использоваться в дальнейшем.

Предполагается, что в самом начале работы рефал-системы все счётчики занулены.

Общее число различных счетчиков, имеющих в данный момент ненулевые значения, не должно превышать 20_8, в противном случае прекращается работа системы и выдается сообщение:

21 ИМЯ

Если при вычитании значение счетчика меньше вычитаемого числа, работа системы прекращается и выдается сообщение:

ВЫЧИТАНИЕ НЕВОЗМОЖНО

9. Запуск задачи в РС4/220

Вызов системы РС4/220 осуществляется с помощью специальной карты вызова, содержащей команды:

```
0  50  0023  0500  7770
0  70  0020  0001  0000
0  16  0000  0600  0000
1  56  0043  1301  7770   к Σ
```

В РС4/220 имеется возможность работы с вариантом компилятора, производящего значительную оптимизацию рефал-программ. Для того чтобы при компиляции использовался оптимизирующий компилятор, карта вызова системы должна быть такой:

```
0  50  0023  0502  7770
0  70  0020  0001  0000
0  16  0000  0600  0000
1  56  0043  1303  7770   к Σ
```

Для запуска задачи в системе РС4/220 пакет перфокарт формируется следующим образом:

1. Карта вызова системы.
2. Две чистые карты.
3. Пакет управляющих директив.
4. Карта с нулевой контрольной суммой.
5. Две чистые карты.
6. Рефал-программа.
7. Карта с нулевой контрольной суммой
8. Две чистые карты.
9. Начальное поле зрения.
10. Карта с нулевой контрольной суммой.

Если рефал-программа разбивается на порции, то в конце порции должна быть карта с нулевой контрольной суммой и две чистые карты.

Если в качестве носителя рефал-программы используется магнитная лента, то в пакете задачи рефал-программа, естественно, отсутствует.

Для запуска описанного выше пакета на ЭВМ БЭСМ-4, М-220, и М-222 оператор должен выполнить следующие действия:

1. Установить на магнитофон с программным номером Ø3 ленту с рефал-системой.

2. Клавишу "Блокировка КЗУ" отжать (для БЭСМ-4 и М-222) или переключить вверх (для М-220).

3. Установить пакет рефал-задачи на устройство ввода.

4. Нажать кнопку "ввод".

10. Эксплуатационные характеристики РС4/220

Скорость компиляции рефал-программы, а также ее быстродействие существенно зависят в РС4/220 от того, какая компилирующая программа использовалась.

Для малого компилятора, в котором не производится значительная оптимизация рефал-программы, скорость компиляции составляет около 400 операторов языка сборки в минуту или 17-20 рефал-предложений в минуту. Быстродействие рефал-программы, скомпилированной на малом компиляторе, составляет 10-15 шагов в секунду.

Для оптимизирующего компилятора скорость компиляции составляет около 45 операторов языка сборки в минуту или 2-3 рефал-предложения в минуту.

Таким образом, скорость компиляции уменьшается здесь по сравнению с малым компилятором в 6-8 раз. Однако быстродействие рефал-программы, скомпилированной на оптимизирующем компиляторе, достигает 25 шагов в секунду, к тому же программа на языке сборки занимает на 15-20% памяти меньше, чем программа, полученная на малом компиляторе.

Приведенные характеристики показывают, что оптимизирующий компилятор имеет смысл использовать только тогда, когда рефал-программа уже отлажена на малом компиляторе и встает вопрос о повышении ее быстродействия. Допускаемая длина рефал-программы составляет 450-500 рефал-предложений для малого компилятора и 550-600 предложений для оптимизирующего компилятора.

Реализация языка рефал, описанная в данных методических рекомендациях, является первым этапом на пути использования его в системном программировании. Однако уже существующие реализации позволяют в большой степени облегчить труд программистов при решении ряда прикладных задач.

В качестве конкретных областей применения рефала в разработке автоматизированных систем проектирования и управления в строительстве следует отметить, в частности, создание специализированных языков общения с ЭВМ, автоматическую генерацию программ в системах переработки информации, перенос программ на языках высокого уровня и их адаптация при переходе от одного типа ЭВМ к другому.

Литература

1. Цифровая вычислительная техника и программирование. М., "Советское радио", 1966, стр.116.

2. "Кибернетика", №4, 1968.

3. Алгоритмический язык рекурсивных функций (РЕФАЛ). – Препринт Ин-та прикладной математики АН СССР, М., 1968.

4. "Кибернетика", №3, 1969.

5. J. Mc Carthy, Recursive Functions of Symbolic Expression and Their Computation by Machine. – Comm. ACM, Apr. 1960.

6. J. Mc Carthy et al. LISP 1,5 Programmer s Manual.

7. V. Ynagve, COMIT, – Comm. ACM, 5, № 1, 1962.

8. D. J. Farber, R.E. Griswold, I.P. Polonsky. SNOBOL, a String Manipulation Language T. ACM 11, № 1, 1964.

9. Труды Матем. Инст. им.Стеклова, т.42 М., Изд-во АН СССР, 1954.

10. Современное программирование. М., "Советское радио", 1966.

11. Современное программирование. М., "Советское радио", 1966.

12. Труды 1-й Всесоюзной конференции по программированию. Киев, ИК АН УССР, 1968.

13. Эффективный интерпретатор для языка РЕФАЛ. – Препринт Ин-та прикладной математики АН СССР. М., 1969.

14. Языки программирования и методы их реализации. Киев. ИК АН УССР, 1973.

15. Труды 2-й Всесоюзной конференции по программированию. Новосибирск. ВЦ СО АН СССР, 1970.

16. Компилятор с языка РЕФАЛ. – Препринт Ин-та прикладной математики АН СССР. М., 1972.

17. Инструкция пользователю по работе с операционной системой ДИСПАК для БЭСМ-6. – Препринт Ин-та прикладной математики АН СССР. М., 1972.

18. Руководство по работе в мониторной системе "Дубна", М., 1970.

19. Язык ФОРТРАН. Дубна, ИОЯИ, 1969.

20. Краткая инструкция по составлению пакета перфокарт для решения задач в мониторной системе "Дубна", БЭСМ-6. М., 1972.

21. Краткая инструкция по работе с фортранными программами в мониторной системе "Дубна" на ЭВМ БЭСМ-6. М., 1974.

22. Транслятор с автокода БЭМШ в мониторной системе "Дубна". - Препринт Ин-та прикладной математики АН СССР. М., 1973.

23. Мониторная система СЕКАЧ. Описание и инструкция. М., ЦНИПИАСС, 1976.

24. Программирование на языке РЕФАЛ. - Препринты ИПМ АН СССР №№ 41,43-44, 48-49, М., 1971.

25. Труды ЦНИПИАСС, вып.6, М., 1974.

содержание

Введение 1
1. Описание языка 3
  1. Универсальный метаязык программирования 3
  2. Понятие конкретизации 4
  3. Знаки, символы, выражения 6
  4. Предложения 10
  5. Свободные переменные 13
  6. Рекурсивные функции 17
  7. Алгоритм проектирования 24
  8. Общие принципы реализации рефала 29
  9. Закапывание и выкапывание 37
  10. Формальное описание базисного рефала 39
II. Приемы программирования 42
  1. Открытые и закрытые переменные. Форматы функции 42
  2. Размерность просмотра 47
  3. Размножение переменных 50
  4. Разветвления и циклы 52
  5. Разделение алгоритма на функции 58
  6. Действия над многочленами от одной переменной 61
  7. Сквозные просмотры 67
  8. Решения задач 74
III. Техника использования рефала 76
  1. Трансляционные задачи 76
  2. Действия над полиномами 96
  3. Обработка графов 105
1У. Реализация рефала. Язык сборки 108
  1. Промежуточный язык 108
  2. Организация памяти 109
  3. Таблица элементов 110
  4. Переменные НЭЛ, Г1 и Г2 111
  5. Язык звеньев 113
  6. Отождествление объектных выражений 114
  7. Отождествление свободных переменных 118
  8. Отождествление открытых переменных выражения 124
  9. Передача управления с одного предложения на другое 127
  10. Оператор КУД, n; 130
  11. Операторы замены 133
  12. Необходимость ограничения "свободы выбора" . 134
  13. Роль переменной Г в процессе замены 134
  14. Порождение объектных выражений 136
  15. Размножение свободных переменных 139

16. Перестановка участков объекта 140
17. Оператор OUT, n; 142
18. Оператор УГ, n; 143
19. Пример преобразования объекта в цель 143
20. Проблема пустых выражений 143
21. Решение проблемы пустых выражений 145
22. Устранение лишних коррекций 148
23. Детерминативы 149
24. Области конкретизаций 150
25. Активизация скобок 152
26. Завершение очередного и начало нового шага . 155
27. Дополнительные операторы 156
28. Примеры 160
29. Словарь понятий языка звеньев 163
30. Список операторов языка сборки 166
У. Компилятор с рефала на язык сборки 167
1. Предварительные замечания 167
2. Внутренняя структура данных 167
У1. Входной язык для ЭВМ типа ЕС ЭВМ, БЭСМ-6, "Минск-32" 175
1. Перфокарта. Бланк 175
2. Метакод-Б 179
3. Модули 182
4. Начальное поле зрения 185
5. Машинная процедура КАРТА (CARD) 186
6. Печать 186
7. Машинные процедуры ЗК, ВК, СЧ, ЗП 187
8. Машинные процедуры арифметики 189
9. Машинные процедуры для лексического анализа 192
10. Прокрутка 194
УП. Запуск программ на машинах серии ЕС ЭВМ . . . 196
1. Общая схема работы с рефал-программой . . 197
2. Наборы данных и DD-предложения 200
3. Каталогизированные процедуры REFAL и REFALG 201
4. Машинные процедуры 202
5. Прокрутка 205
6. Диагностические сообщения и коды возврата . . 206
7. Примеры 208
УШ. Запуск программ на машине БЭСМ-6 212
1. Местонахождение рефал-системы 212
2. Простейший пакет 212
3. Вызов рефал-компилятора 215
4. Запуск готовой программы 216

5. Средства отладки 218
6. Работа с личной библиотекой 222
7. Связь с ФОРТРАНом 223
8. Машинные процедуры для работы с символьными файлами 225
9. Обменные функции и символы-ссылки 228
10. Управление размером скомпилированной программы 233
X. Запуск программ на машине "Минск-32" 234
1. Общие сведения о рефал-системе 234
2. Компиляция программ 234
3. Загрузка и выполнение программ 235
4. Библиотека машинных процедур 238
X. Запуск программ на машинах типа М-220 238
1. Перфокарта. Бланк 239
2. Метакод-А 240
3. Управление заданиями 241
4. Библиотека машинных процедур 246
5. Машинные процедуры печати и перфорации . . . 247
6. Машинные процедуры ЗК, ВК, ПАМ 247
7. Машинные процедуры арифметики 249
8. Машинные процедуры сравнения, обмена и счетчик 250
9. Запуск задачи в РС4/220 253
10. Эксплуатационные характеристики РС4/220 . . 254
Литература 255

БАЗИСНЫЙ РЕФАЛ И ЕГО РЕАЛИЗАЦИЯ НА ВЫЧИСЛИТЕЛЬНЫХ МАШИНАХ (методические рекомендации). М., ЦНИПИАСС 258 с. (Фонд алгоритмов и программ для ЭВМ в отрасли "Строительство". Вып.У-40).

Л-91371. Подписано к печати 3/Ш-77 г. Формат 60х84/16.
Объем 16 п.л. Тир.800. Зак.273. Цена 2 руб.17 коп.

ОТРД ЦНИПИАСС
117393, ГСП-1, Москва, В-393, Новые Черемушки, квартал 28, корпус 3

цнипиасс

Цена 2 руб. 17 коп.