Ордена Ленина
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В. Келдыша
Академии наук СССР

С.А. Романенко

РЕАЛИЗАЦИЯ РЕФАЛА - 2.

Москва

Ордена Ленина
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
им. М.В.Келдыша
Академии Наук СССР

С.А.Романенко

Реализация Рефала-2.

Москва
1987

УДК 681.3.06

Работа посвящена теоретическим и практическим аспектам реализации языка программирования Рефал-2, предназначенного для обработки символьной информации. Основными изобразительными средствами Рефала-2 являются синтаксическое отождествление (сопоставление с образцом), подстановка и рекурсия.

В главах 1 и 2 формулируются и доказываются некоторые свойства правила отождествления, принятого в Рефале-2, которые служат основой для алгоритмов компиляции и оптимизации рефал-программ.

В главе 3 описан язык сборки - промежуточный машинно-независимый язык низкого уровня, допускающий простую аппаратную и программную реализацию.

В главе 4 описан компилятор с рефала на язык сборки.

Монография предназначена для специалистов, занимающихся разработкой и реализацией языков программирования.

КЛЮЧЕВЫЕ СЛОВА И ФРАЗЫ: компилятор, обработка символьной информации, оптимизация программ, рефал, синтаксическое отождествление, сопоставление с образцом, функциональное программирование, язык программирования.

С О Д Е Р Ж А Н И Е

Введение 7

Обозначения 13

ГЛАВА 1. Спецификаторы 14
1.1. Пространства спецификаторов 14
1.2. Семантика спецификаторов 16
1.3. Свойства конкатенации спецификаторов 21
1.4. Достаточные условия полноты спецификаторов . . 23
1.5. Сужение спецификатора 24
1.6. Об'единение спецификаторов 27
1.7. Пересечение спецификаторов 29
1.8. Разность спецификаторов 31
1.9. Распознавание отношений между спецификаторами 32
1.10. Распознавание пустоты значения спецификатора 33
1.11. Спецификаторы как элементы спецификаторов . . . 35

ГЛАВА 2. Алгоритмы отождествления 38
2.1. Аксиоматический и процедурный подходы к синтаксическому отождествлению 38
2.2. Подстановки 40
2.3. Отождествления 42
2.4. Алгоритм отождествления 48
2.5. Сокращение перебора при отождествлении 61
2.6. Переупорядочение дыр 66
2.7. Отделение удлиняющегося выражения 70
2.8. Отделение максимальных выражений 78
2.9. Оценка множеств нуль-термов 84

ГЛАВА 3. Язык сборки 88
3.1. Язык сборки как система команд 88
3.2. Организация памяти 89
3.3. Форматы операторов языка сборки 92
3.4. Общая структура интерпретатора языка сборки . . 93
3.5. Таблица элементов 95
3.6. Переменные NEL, B1 И B2 97
3.7. Язык звеньев 98
3.8. Сопоставление об'ектных выражений 100
3.9. Сопоставление свободных переменных 107
3.10. Сопоставление открытых вхождений VE-переменных 115
3.11. Передача управления с одного предложения на другое 119
3.12. Оператор EQE(N) 122
3.13. Реализация спецификаторов 128
3.14. Операторы преобразования поля зрения 137
3.15. Список свободной памяти 138
3.16. Порождение об'ектных выражений 142
3.17. Порождение функциональных скобок 145
3.18. Копирование переменных 147
3.19. Пересадка кусков списка 148
3.20. Завершение очередного шага и подготовка следующего 151
3.21. Порождение операторов преобразования 154
3.22. Примеры перевода функций на язык сборки 159

ГЛАВА 4. Компилятор с рефала на язык сборки 161
4.1. Общая структура компилятора 161
4.2. Компиляция левой части предложения 162
4.3. Компиляция правой части предложения 174

Литература 177

Перечень функций и обозначений 179
Предметный указатель 184
Перечень операторов языка сборки 189
Коды операторов языка сборки 191

В В Е Д Е Н И Е

Данная работа посвящена теоретическим и практическим аспектам реализации языка программирования Рефал-2, предназначенного для обработки символьной информации. Основными изобразительными средствами Рефала-2 являются синтаксическое отождествление (сопоставление с образцом), подстановка и рекурсия [Т 1966], [Т 1968], [ТС 1969], [Т 1974], [БР 1977], [Р2ОВЯ 1987], [Р2ОБФ 1986].

Основные трудности, возникающие при реализации рефала на современных вычислительных машинах обусловлены следующими обстоятельствами.

- Об'ектом обработки для программ, написанных на рефале, являются символические выражения, размер и структура которых, вообще говоря, неизвестны до того, как рефал-программа начнет работать. Этим рефал разительно отличается, от таких операторных языков, как фортран.

- Рефал-программа представляет собой набор предложений, которые могут применяться совсем не в том порядке, в котором они написаны. На каждом шаге работы рефал-машина должна выбирать среди предложений то, которое следует применить. Этот поиск может требовать значительного перебора, что может, если не предпринимать серьезных мер по его ограничению и искоренению, сделать работу рефал-программ неприемлемо медленной.

- Системы команд существующих вычислительных машин предоставляют набор операций хорошо приспособленный для численных задач и задач обработки данных, однако эти операции не отражают специфики задач обработки

символьной информации. Поэтому, хотя рефал-программы, в принципе, и можно компилировать непосредственно в команды машины, на практике это приводит к катастрофическому разбуханию программы в результате компиляции, ибо даже те операции, которые являются элементарными с точки зрения символьной обработки, после компиляции превращаются в длинные последовательности команд машины.

Для преодоления этих трудностей в данной реализации рефала были использованы следующие средства:

- списковая организация памяти;
- оптимизации при компиляции рефал-программ;
- промежуточный язык (язык сборки), на который компилируются рефал-программы.

Рассмотрим эти средства более подробно.

Для представления рефал-выражений в памяти машины используется списковая организация памяти. При этом в данной реализации рефала каждый элемент рефал-выражения отображается в одно звено списка. Таким образом, для представления выражений выбрано одно, раз и навсегда зафиксированное представление, которое не зависит от структуры исполняемой рефал-программы. Конечно, это самое простое решение проблемы, но оно не является наилучшим ни с точки зрения расхода памяти под выражения, ни с точки зрения скорости исполнения рефал-программ. Поэтому будем надеяться, что в будущем появятся такие реализации рефала, в которых представление для рефал-выражений будет автоматически выбираться во время компиляции, на основе глубокого анализа структуры рефал-программы.

Хотя представление выражений в данной рефализации рефала и зафиксировано, это еще не означает, что после этого в рефал-программах уже нечего оптимизировать. Поэтому при компиляции рефал-программ на язык сборки компилятор стремится повысить скорость работы скомпилированных рефал-программ и уменьшить их размер.

Каждый шаг рефал-машины состоит из двух этапов:

синтаксического отождествления и преобразования поля зрения. В соответствии с этим и оптимизации, выполняемые при компиляции рефал-программ, распадаются на две большие группы: оптимизации, связанные с отождествлением, и оптимизации, связанные с преобразованием поля зрения.

Синтаксическое отождествление, как оно определяется в описании рефала, может требовать значительного комбинаторного перебора. Поэтому возникает задача разработки достаточно эффективного алгоритма отождествления, всегда дающего те же результаты, что и правило отождествления в описании рефала, но в то же время обходящегося без перебора в большинстве случаев, встречающихся в реальных рефал-программах. Корректность любого мало-мальски сложного алгоритма отождествления приходится доказывать, что представляет собой достаточно утомительную задачу. К счастью, с этой проблемой сталкиваются только реализаторы рефала, в то время как обычные пользователи могут знать только простое правило отождествления из описания языка.

Оптимизации, относящиеся к преобразованию поля зрения, значительно проще по своей сущности, чем преобразования, связанные с отождествлением. Основным источником экономии при выполнении преобразований является то, что в реальных рефал-программах левые и правые части предложений часто бывают похожи друг на друга и содержат совпадающие части. Поэтому можно формировать результат замены ведущего функционального терма не совсем заново, а используя готовые куски из аргумента функции.

Как уже было сказано, рефал-программы компилируются не непосредственно в команды машины, а на промежуточный язык (язык сборки), к которому при его разработке пред'являлись следующие требования:

- Адекватность рефалу. Программы на рефале должны сравнительно просто и естественно компилироваться на язык сборки.

- Простота реализации. Язык сборки должен допускать простую программную реализацию на существующих ЭВМ.

Это, до некоторой степени, может служить гарантией того, что язык сборки будет также допускать простую микропрограмную или аппаратную реализацию.

- Эффективность. Рефал-программы, скомпилированные на язык сборки, должны работать достаточно быстро. Это означает, что во-первых, операторы языка сборки должны допускать эффективную реализацию, а во-вторых, язык сборки должен давать достаточный простор для проведения различных оптимизаций во время компиляции рефал-программ на язык сборки.

- Машинная независимость. Язык сборки не должен отражать специфические особенности какой-либо конкретной ЭВМ.

Данная работа состоит из четырех глав.

В п е р в о й главе формулируются и доказываются различные свойства спецификаторов.

Спецификаторы служат в рефал-программах для формулировки ограничений, накладываемых на множество допустимых значений переменных.

Спецификаторы, изучаемые в главе 1, являются частью входного языка системы программирования Рефал-2 для БЭСМ-6 и ЕС ЭВМ [Р20ВЯ 1987]. В предыдущих реализациях рефала спецификаторов не было, поэтому не было и необходимости изучать их свойства.

Каждый спецификатор представляет собой синтаксический об'ект, который изображает некоторое множество об'ектных термов. В главе 1 вводится несколько операций над спецификаторами: об'единение, пересечение, разность и др., исследуются свойства этих операций и доказывается, что эти операции корректны относительно семантики спецификаторов. Например, операция пересечения спецификаторов дает спецификатор, изображающий множество, которое является пересечением множеств, изображаемых исходными спецификаторами. Рассматривается также вопрос о проверке различных теоретико-множественных соотношений между множествами, изображаемыми спецификаторами: включения, пустоты пересечения и др.

Операции над спецификаторами, введенные в главе 1, используются в компиляторе с рефала на язык сборки.

Во в т о р о й главе рассматриваются вопросы, связанные с синтаксическим отождествлением в языке рефал. Во всех рассмотрениях предполагается, что отождествление производится слева направо, однако все результаты, естественно, применимы после очевидной переформулировки и к отождествлению справа налево.

В описаниях рефала [БР 1977], [Р20ВЯ 1987], [Т 1966], [Т 1974] определение правила отождествления носит непроцедурный характер: описывается результат отождествления, но не рассматривается вопрос о методах его эффективного отыскания. Между тем, синтаксическое отождествление является основным средством, с помощью которого в рефал-программах осуществляется доступ к данным, проверка условий и организация управления. Поэтому скорость работы рефал-программ в значительной степени зависит от того, насколько эффективно будет осуществляться синтаксическое отождествление.

Между тем, непосредственная реализация правила отождествления, сформулированного в описании рефала, приводит к крайне неэффективному алгоритму отождествления. Поэтому в главе 2 формулируются и доказываются различные свойства правила отождествления, которые служат основой для алгоритма отождествления, описанного в этой же главе. Основная идея этого алгоритма, впрочем, уже неоднократно описывалась [БР 1977]. Далее в главе 2 рассматриваются различные способы уменьшения комбинаторного перебора возникающего в тех случаях, когда левая часть содержит вхождения VE-переменных. Эти способы, в основном, были описаны в [РКТ 1973]. Однако, в связи с появлением спецификаторов, появились и некоторые новые оптимизации, которые описываются в главе 2.

В т р е т ь е й главе описан язык сборки, интерпретатор языка сборки и методы компиляции с рефала на язык сборки.

После неформального описания каждого оператора языка сборки приводится его формальное описание в виде подпрограммы интерпретатора языка сборки, реализующей этот оператор. Таким образом, формальное описание семантики языка сборки дано через описание интерпретатора языка сборки.

Методы компиляции с рефала на язык сборки описаны в главе 3 неформально, с помощью примеров перевода рефал-предложений на язык сборки.

Тот вариант языка сборки, который описан в главе 3, отличается от ранее опубликованных [КРТ 1972], [БР 1973] тем, что он дает возможность использовать спецификаторы и отождествление справа налево.

Его важной особенностью является также то, что пересадка кусков списка в процессе замены выполняется не сразу, а откладывается на конец шага. Благодаря этому, рефал-машина всегда может вернуть поле зрения к тому состоянию, в котором оно было до начала шага, в том случае, если свободной памяти не хватает для завершения шага. Это усовершенствование было предложено Арк.В.Климовым и впервые опробовано на практике А.В.Веденовым, которым автор считает своим приятным долгом выразить благодарность.

В четвертой главе описан компилятор с рефала на язык сборки. Обсуждается общая структура компилятора, а затем приводится описание его наиболее сложных частей: алгоритма компиляции левой части предложения и алгоритма компиляции правой части предложения.

Алгоритм компиляции левой части включает различные оптимизации, уменьшающие комбинаторный перебор при отождествлении. Корректность этих оптимизаций следует из результатов глав 1 и 2.

В каком порядке следует читать данную работу? С формально-логической точки зрения ее следует читать подряд, начиная с главы 1, поскольку каждая глава опирается на результаты предыдущих. Однако, можно поступить и иначе: начать чтение с глав 3 и 4, время от времени обращаясь за раз'яснениями к предметному указателю и перечню функций и обозначений, находящимся в конце работы. Если при этом у читателя возникнут какие-то сомнения в корректности методов компиляции и оптимизации, описанных в главах 3 и 4, он всегда сможет вернуться к главам 1 и 2. При этом он получит ответы на вопросы, которые к тому времени уже успеют возникнуть, вместо того, чтобы пытаться угадывать вопросы, исходя из получаемых ответов.

ОБОЗНАЧЕНИЯ

При подготовке текста монографии использовался ограниченный набор специальных знаков, поэтому, в ряде случаев, обозначения отличаются от общепринятых.

Если U и V - множества, то через U+V, U*V и U-V обозначаются, соответственно, об'единение, пересечение и разность множеств U и V.

Пустое множество обозначается через <Ø>.

<* A1, A2,...,AN *> обозначает конечное множество, состоящее из элементов A1, A2,...,AN.

Если U - множество, то X IN U означает, что X - элемент U.

Если U и V - множества, то U SUBSET V означает, что U подмножество V (при этом не исключается случай U=V).

(A1,A2,...,AN) обозначает упорядоченную N-ку (кортеж), состоящую из элементов A1, A2,...,AN.

Пустой кортеж обозначается через <>.

ГЛАВА 1. СПЕЦИФИКАТОРЫ.

1.1. ПРОСТРАНСТВА СПЕЦИФИКАТОРОВ

В описании рефала содержится определение синтаксиса и семантики спецификаторов. В настоящей главе формулируются и доказываются различные свойства спецификаторов, а также вводится ряд операций над ними. Эти операции затем могут использоваться в компиляторе с рефала на язык сборки.

Для достижения этих целей оказывается необходимо описать семантику спецификаторов более подробно и в более абстрактном виде, чем это сделано в описании рефала.

О п р е д е л е н и е 1.1.

П р о с т р а н с т в о м с п е ц и ф и к а т о р о в называется упорядоченная тройка

(PSPEC,OTERM,YES), где

PSPEC - счетное множество об'ектов, именуемых п е р в и ч н ы м и с п е ц и ф и к а т о р а м и,

OTERM - счетное множество об'ектов, именуемых о б ' е к т н ы м и т е р м а м и,

YES - функция, которая каждому первичному спецификатору ставит в соответствие некоторое множество об'ектных термов.

При этом функция YES должна обладать следующими свойствами:

(1) для любых первичных спецификаторов P и P'

$$YES[P]=YES[P'] \Rightarrow P=P'.$$

(2) для любых первичных спецификаторов P и P' выполнено

хотя бы одно из следующих соотношений:

YES[P]*YES[P'] =<Ø>
YES[P] SUBSET YES[P']
YES[P'] SUBSET YES[P]

(3) для любого первичного спецификатора P

YES[P]¬=<Ø>.

Определим на множестве первичных спецификаторов отношение <= следующим образом

P<= P' ⇔ YES[P] SUBSET YES[P']

Легко видеть, что так определенное отношение является отношением частичного порядка.

О п р е д е л е н и е 1.2.
Отрицанием первичного спецификатора P называется конструкция (P), т.е. выражение, получающееся заключением P в круглые скобки. С п е ц и ф и к а т о р о м называется конструкция вида

Q1 Q2 ... QN

где каждое Q[I] - это либо первичный спецификатор, либо отрицание первичного спецификатора. Случай N=Ø не исключается, т.е. спецификатор может быть пустым.

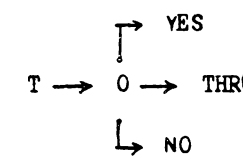
Пустой спецификатор будем в дальнейшем обозначать через <>.

I.2. СЕМАНТИКА СПЕЦИФИКАТОРОВ

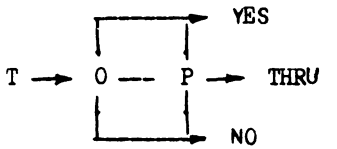
Семантика первичных спецификаторов задается функцией YES. А именно, для любого первичного спецификатора P, YES[P] - это множество тех об'ектных термов T, которые "удовлетворяют P". Мы будем также говорить, что первичный спецификатор P "изображает" множество об'ектных термов YES[P].

Семантика произвольного спецификатора определяется через семантику составляющих его первичных спецификаторов.

Согласно описанию Рефала-2 [Р2ОВЯ 1987], каждый спецификатор представляет собой как бы некоторое устройство, имеющее один вход и три выхода: YES, NO, THRU. На вход спецификатора можно подать произвольный об'ектный терм T. Спецификатор анализирует этот терм и либо выдает сигнал на выход YES (говорит "да"), либо подает сигнал на выход NO (говорит "нет"), либо не говорит ни "да", ни "нет", а пропускает этот терм сквозь себя на выход THRU. Таким образом, спецификатор Q можно наглядно изобразить следующим образом:



Спецификатор - это более сложное устройство, чем предикат, поскольку предикат имеет только два выхода: YES и NO, а у спецификатора еще есть и выход THRU. Наличие третьего выхода позволяет создавать из спецификаторов более сложные устройства, поскольку термы, попадающие на выход THRU, можно посылать на входы других спецификаторов. Например, два спецификатора Q и R можно соединить последовательно следующим образом:



В результате последовательного соединения опять получается устройство, которое имеет один вход и три выхода: YES, NO и THRU, и которое, таким образом, тоже является спецификатором.

Обозначим через YES[Q] множество тех об'ектных термов, для которых спецификатор Q говорит "да", через NO[Q] - множество об'ектных термов, для которых спецификатор Q говорит "нет", а через THRU[Q] - множество тех об'ектных термов, которые спецификатор Q пропускает сквозь себя, т.е. для которых он не говорит ни "да", ни "нет". Очевидно, что функции YES, NO и THRU полностью определяют поведение спецификаторов.

В описании Рефала-2 [Р20ВЯ 1987] семантика спецификатора, составленного из первичных спецификаторов и отрицаний первичных спецификаторов описана с помощью следующего алгоритма, который для любого об'ектного терма T и любого спецификатора Q позволяет узнать к какому из множеств YES[Q], NO[Q] и THRU[Q] принадлежит T.

Если Q=<>, т.е. Q - пустой спецификатор, то T принадлежит THRU[Q].

Если Q= P Q', т.е. Q начинается с первичного спецификатора P, то следует рассмотреть два случая. Если T принадлежит YES[P], то T принадлежит YES[Q]. Если же T не принадлежит YES[P], то следует применить тот же самый алгоритм к остатку спецификатора Q, т.е. к Q'.

Если Q= (P) Q', т.е. Q начинается с отрицания первичного спецификатора P, то следует рассмотреть два случая. Если T принадлежит YES[P], то T принадлежит NO[P]. Если же T не принадлежит YES[P], то следует применить тот же самый алгоритм к остатку спецификатора Q, т.е. к Q'.

Приведенные выше рассуждения об устройствах, выходах и сигналах хороши в качестве неформальных пояснений, однако

они не могут служить удобной основой для теоретического изучения и доказательства свойств спецификаторов. Поэтому ниже мы определим семантику спецификаторов заново. Проверка того, что предлагаемый формализм является адекватным уточнением приведенных выше рассуждений предоставляется читателю в качестве приятного упражнения.

Итак, семантика первичных спецификаторов задается функцией YES. Следующая наша задача - определить семантику произвольных спецификаторов. Для этого мы расширим область определения функции YES так, чтобы она был определена на любых спецификаторах.

Пусть P - произвольный первичный спецификатор, а Q и R - произвольные спецификаторы. Тогда значение функции YES определяется из следующих рекурсивных соотношений.

(1) YES[<>] = <∅>
(2) YES[P R] = YES[P] + YES[R]
(3) YES[(P) R] = YES[R] - YES[P]

С содержательной точки зрения функция YES каждому спецификатору Q ставит в соответствие то множество об'ектных термов YES[Q], которое он изображает.

Теперь определим с помощью рекурсивных соотношений функцию NO, которая каждому спецификатору Q ставит в соответствие некоторое множество об'ектных термов NO[Q].

(1) NO[<>] = <∅>
(2) NO[P R] = NO[R] - YES[P]
(3) NO[(P) R] = YES[P] + NO[R].

Если X - некоторое множество об'ектных термов, то через ¬X будем обозначать его дополнение до множества всех об'ектных термов. Таким образом,

¬X = OTERM - X.

Теперь определим функцию THRU, которая каждому спецификатору Q ставит в соответствие некоторое множество об'ектных

термов и выражается через функции YES и NO следующим образом

$$THRU[Q] = \neg[YES[Q] + NO[Q]]$$

Теперь определим с помощью рекурсивных соотношений унарную операцию $\neg$, которая каждому спецификатору Q ставит в соответствие спецификатор $\neg Q$, именуемый отрицанием Q.

(1) $\neg[<>] = <>$
(2) $\neg[P\ Q] = (P)\neg[Q]$
(3) $\neg[(P)\ Q] = P\neg[Q]$

Очевидно, что для любого спецификатора Q имеет место

$$\neg\neg Q = Q.$$

Используя введенную выше операцию $\neg$ над спецификаторами, мы можем выразить значение функции NO через значение функции YES и наоборот.

Т е о р е м а 2.1.

$$NO[Q] = YES[\neg Q]$$

Д о к а з а т е л ь с т в о.
Индукцией под длине спецификатора Q.
Случай Q=<>. Тогда $NO[<>]=YES[\neg<>]=<\emptyset>$.
Случай Q = P Q'. Тогда, по индуктивному предположению, $NO[Q']=YES[\neg Q']$. Поэтому $NO[P\ Q'] = NO[Q']-YES[P] = YES[\neg Q']-YES[P] = YES[(P)\neg Q'] = YES[\neg[P\ Q']] = YES[\neg Q]$.
Случай Q=(P)Q'. Тогда, используя индуктивное предположение, получаем $NO[(P)Q'] = YES[P]+NO[Q'] = YES[P]+YES[\neg Q'] = YES[P\neg Q'] = YES[\neg[(P)Q']] = YES[\neg Q]$.
Теорема доказана.

Следствия.

YES[Q] = NO[¬Q]
THRU[Q] = THRU[¬Q]

Теперь рассмотрим следующий вопрос. Верно ли, что для любого спецификатора Q его отрицание ¬Q изображает дополнение множества YES[Q]. Другими словами, верно ли, что для любого спецификатора Q имеет место YES[¬Q]=¬YES[Q]. Оказывается, что ответ на этот вопрос - отрицательный.

Определение 2.1.

Спецификатор Q называется полным, если NO[Q]=¬YES[Q].

Теорема 2.2.

NO[Q] SUBSET ¬YES[Q].

Доказательство.
Проводится индукцией по длине Q.
Случай Q=<>. <Ø> = NO[<>] SUBSET ¬YES[<>] = ¬<Ø> = OTERM.
Случай Q = P Q'. Тогда по индуктивному предположению NO[Q'] SUBSET ¬YES[Q'].
Отсюда NO[Q] = YES[¬Q] = YES[(P)¬Q'] = YES[¬Q']-YES[P]. ¬YES[Q] = ¬[YES[P]+YES[Q']] = ¬YES[P]*¬YES[Q'] = ¬YES[Q']-YES[P].
Поэтому YES[¬Q'] SUBSET ¬YES[Q'] ⇒ YES[¬Q']-YES[P] SUBSET ¬YES[Q']-YES[P] ⇒ YES[Q] SUBSET ¬YES[Q].
Случай Q = (P) Q'. Используя индуктивное предположение, получаем YES[¬Q] = YES[P]+YES[¬Q']. ¬YES[Q] = ¬[YES[Q']*¬YES[P]] = YES[P]+¬YES[Q'].
Поэтому YES[¬Q'] SUBSET ¬YES[Q'] ⇒ YES[P]+YES[¬Q'] SUBSET YES[P]+¬YES[Q'] ⇒ YES[¬Q] SUBSET ¬YES[Q].
Теорема доказана.

Следствия.

YES[Q] * NO[Q] = <Ø>
YES[Q] SUBSET ¬NO[Q]

Теорема 2.3.

Q – полный спецификатор ⟺ THRU[Q] = <Ø>.

Доказательство.
(⇒). Если Q – полный, то NO[Q] = ¬YES[Q]. Поэтому THRU[Q] = ¬[YES[Q]+NO[Q]] = ¬ [YES[Q]+¬YES[Q]] = ¬OTERM = <Ø>.
(⇐). Пусть THRU[Q] = <Ø>. Тогда ¬YES[Q]*¬NO[Q] = <Ø>. Поэтому ¬YES[Q] SUBSET NO[Q]. По теореме 2.2 всегда NO[Q] SUBSET ¬YES[Q]. Отсюда следует, что NO[Q] = ¬YES[Q].
Теорема доказана.

Теорема 2.4.
Если Q – полный спецификатор, то и ¬Q – полный спецификатор.
Доказательство.
По следствию к теореме 2.1, THRU[¬Q]=THRU[Q]. Q – полный, поэтому, по теореме 2.3, THRU[Q]=<Ø>. Таким образом, THRU[¬Q]=<Ø>, поэтому, по теореме 2.3, ¬Q – полный.

I.3.СВОЙСТВА КОНКАТЕНАЦИИ СПЕЦИФИКАТОРОВ

Пусть даны два спецификатора Q и R. Образуем их конкатенацию Q R, т.е. к спецификатору Q припишем справа спецификатор R. Оказывается, что значения функций YES, NO и THRU на спецификаторе Q R выражаются через значения этих функций на спецификаторах Q и R.

Т е о р е м а 3.1.
Для любых спецификаторов Q и R

$$YES[Q\ R] = YES[Q] + [YES[R] - NO[Q]].$$

Д о к а з а т е л ь с т в о.
Проводится индукцией по длине Q.
Случай Q = <>. YES[R] = YES[<>]+[YES[R]-NO[<>]].
Случай Q = P Q'. По индуктивному предположению YES[Q' R] = YES[Q'] + [YES[R]-NO[Q']]. Поэтому YES[Q R] = YES[P] + YES[Q' R] = YES[P] + YES[Q'] + [YES[R]-NO[Q']] = YES[Q] + YES[P] + [YES[R]-NO[Q']].
Теперь воспользуемся теоретико-множественным тождеством A+[B-C] = A+[B-[C-A]]. Получаем YES[P Q' R] = YES[Q'] + YES[P] + [YES[R] - [NO[Q']-YES[P]]]. Далее NO[Q']-YES[P] = YES[(P)¬Q'] = YES[¬Q] = NO[Q]. Отсюда YES[Q R] = YES[Q] + [YES[R]-NO[Q]].
Случай Q = (P)Q'. YES[Q R] = YES[Q' R] - YES[P] = [YES[Q'] + [YES[R] - NO[Q']]] - YES[P] = YES[(P)Q'] + [YES[R] - YES[¬[P Q']]] = YES[Q] + [YES[R]-NO[Q]].
Теорема доказана.

Т е о р е м а 3.2.

$$YES[Q\ R] = YES[Q] + [YES[R]*THRU[Q]].$$

Д о к а з а т е л ь с т в о.
YES[R] * THRU[Q] = YES[R] *¬YES[Q] *¬NO[Q] =¬YES[Q] * [YES[R]-NO[Q]]. Теперь воспользуемся теоретико-множественным тождеством A+B = A+[¬A*B] и теоремой 3.1. Получим YES[Q R] = YES[Q] + [YES[R]-NO[Q]] = YES[Q] +[¬YES[Q] * [YES[R]-NO[Q]]] = YES[Q] + [YES[R]*THRU[Q]].

Т е о р е м а 3.3.

$$NO[Q\ R] = NO[Q] + [NO[R]-YES[Q]] = NO[Q] + [NO[R]*THRU[Q]].$$

Доказательство.
NO[Q R] = YES[¬Q¬R] = YES[¬Q] + [YES[¬R]-NO[¬Q]] = NO[Q] + [NO[R]-YES[Q]]. Аналогично NO[Q R] = YES[¬Q¬R] = YES[¬Q] + [YES[¬R]*THRU[¬Q]] = NO[Q] + [NO[R]*THRU[Q]].

Теорема 3.4.

THRU[Q R] = THRU[Q] * THRU[R].

Доказательство.
THRU[Q R] = ¬[YES[Q R] + NO[Q R]] = ¬[YES[Q] + [YES[R]-NO[Q]] + NO[Q] + [NO[R]-YES[Q]]] = ¬[YES[Q] + YES[R] + NO[Q] + NO[R]]. С другой стороны, THRU[Q]*THRU[R] = ¬[YES[Q]+NO[Q]] * ¬[YES[R]+NO[R]] = ¬[YES[Q] + NO[Q] + YES[R] + NO[R]].

4.4.ДОСТАТОЧНЫЕ УСЛОВИЯ ПОЛНОТЫ СПЕЦИФИКАТОРОВ

Пусть W - первичный спецификатор, такой, что YES[W] совпадает с множеством всех об'ектных термов, т.е. YES[W]=OTERM.

Теорема 4.1.
Если спецификатор Q имеет вид Q = Q' W, то Q - полный спецификатор, т.е. NO[Q] = ¬YES[Q].
Доказательство.
По следствию из теоремы 2.2 имеем YES[Q'] SUBSET ¬NO[Q']. Поэтому YES[Q] = YES[Q'] + [YES[W]-NO[Q']] = YES[Q'] + ¬NO[Q'] = ¬NO[Q']. С другой стороны, NO[Q] = YES[¬[Q'](W)] = NO[Q'] + [YES[(W)]-YES[Q']] = NO[Q'].
Таким образом, NO[Q] = NO[Q'] = ¬YES[Q].
Теорема доказана.

Т е о р е м а 4.2.

Если спецификатор Q имеет вид Q = Q'(W), то Q – полный спецификатор, т.е. NO[Q] = ¬YES[Q].

Д о к а з а т е л ь с т в о.

По теореме 2.2 имеет место NO[Q'] SUBSET ¬YES[Q']. Поэтому NO[Q] = YES[¬[Q'] W] = NO[Q'] + [YES[W]-YES[Q']] = NO[Q'] + ¬YES[Q'] = ¬YES[Q']. С другой стороны, YES[Q] = YES[Q'(W)] = YES[Q'] + [YES[(W)]-NO[Q']] = YES[Q'].

Таким образом, NO[Q] = ¬YES[Q'] = ¬YES[Q].

Теорема доказана.

В рефал-программах используются только полные спецификаторы, ибо согласно описанию языка рефал, к каждому спецификатору, встречающемуся в тексте рефал-программы, перед его интерпретацией приписывается сзади либо – W, если этот спецификатор кончается на правую скобку ")", либо – (W), если этот спецификатор не кончается на правую скобку.

I.5. СУЖЕНИЕ СПЕЦИФИКАТОРА

О п р е д е л е н и е 5.1.

Пересечением двух первичных спецификаторов P и P' называется первичный спецификатор, который обозначается P*P' и удовлетворяет условиям

(1) если YES[P] * YES[P'] = <Ø>, то P*P' = < >.

(2) если YES[P] SUBSET YES[P'], то P*P' = P.

(3) если YES[P'] SUBSET YES[P], то P*P' = P'.

Заметим, что из определения 1.1 следует, что P*P' всегда определен и при том однозначно. Поскольку YES[P]=YES[P'] влечет P=P', получаем, что при YES[P]=YES[P'], P*P' = P = P'.

Т е о р е м а 5.1.

YES[P*P'] = YES[P] * YES[P'].

Доказательство. Очевидно из определения 4.1.

Определение 5.2.

Сужением спецификатора Q по первичному спецификатору P, называется спецификатор, который обозначается через CONTR[P,Q] и определяется следующими рекурсивными соотношениями

(1) CONTR[P,<>] = <>
(2) CONTR[P, Q P'] = CONTR[P,Q][P*P']
(3) CONTR[P, Q (P')] = CONTR[P,Q] ([P*P'])

Из определения 5.2 очевидно, что CONTR[P,¬Q] = ¬CONTR[P,Q].

Теорема 5.2.

YES[CONTR[P,Q]] = YES[P] * YES[Q].

Доказательство.

Проводится индукцией по длине Q.

Случай Q =<>. YES[<>] = YES[P]*YES[<>].

Случай Q = Q' P'. По индуктивному предположению YES[CONTR[P,Q']] = YES[P]*YES[Q']. Поэтому YES[P]*YES[Q] = YES[P]*YES[Q' P'] = YES[P] * [YES[Q'] + [YES[P'] -NO[Q']]] = [YES[P]*YES[Q']] + [YES[P] * [YES[P'] -NO[Q']]] = YES[CONTR[P,Q']] +[[YES[P]*YES[P']] - [YES[P]*NO[Q']]] = YES[CONTR[P,Q']] + [YES[P*P'] - [YES[P]*NO[Q']]].

С другой стороны, YES[CONTR[P,Q]] = YES[CONTR[P,Q'] [P*P']] = YES[CONTR[P,Q']] + [YES[P*P'] - NO[CONTR[P,Q']]].

Таким образом, осталось доказать, что NO[CONTR[P,Q']] = YES[P]*NO[Q']. Действительно, NO[CONTR[P,Q']] = YES[¬CONTR[P,Q']] = YES[CONTR[P,¬Q']] = YES[P]*YES[¬Q'] = YES[P]*NO[Q'].

Случай Q = Q' (P'). Имеем YES[P]*YES[Q] = YES[P]*YES[Q'(P')] = YES[P] * [YES[Q'] + [YES[P'] -NO[Q']]]= YES[P]*YES[Q'].

С другой стороны, YES[CONTR[P,Q]] = YES[CONTR[P,Q']([P*P'])] = YES[CONTR[P,Q']] + [YES[(P*P')] - NO[CONTR[P,Q']]] = YES[CONTR[P,Q']] = YES[P]*YES[Q'].

Теорема доказана.

Т е о р е м а 5.3.

NO[CONTR[P,Q]] = YES[P] * NO[Q].

Д о к а з а т е л ь с т в о.

NO[CONTR[P,Q]] = YES[¬CONTR[P,Q]] = YES[CONTR[P,¬Q]] = YES[P]*YES[¬Q] = YES[P]*NO[Q].

Т е о р е м а 5.4.

THRU[CONTR[P,Q]] = THRU[P] + THRU[Q].

Д о к а з а т е л ь с т в о.

THRU[P] = ¬YES[P], THRU[Q] = ¬[YES[Q]+NO[Q]] = ¬YES[Q]*¬NO[Q]. Поэтому THRU[P]+THRU[Q] = ¬YES[P] + [¬YES[Q]*¬NO[Q]] =[¬YES[P] + ¬YES[Q]] *[¬YES[P] +¬NO[Q]] = ¬[YES[P]*YES[Q]] *¬ [YES[P]*NO[Q]] = ¬YES[CONTR[P,Q]] * ¬NO[CONTR[P,Q]] = THRU[CONTR[P,Q]].

Т е о р е м а 5.5.

Если Q - полный спецификатор, то

THRU[CONTR[P,Q]] = THRU[P].

Д о к а з а т е л ь с т в о.

Если Q - полный, то, по теореме 2.3, THRU[Q]=<Ø>. Поэтому, THRU[CONTR[P,Q]] = THRU[P]+THRU[Q] = THRU[P].

1.6. ОБ'ЕДИНЕНИЕ СПЕЦИФИКАТОРОВ

О п р е д е л е н и е 6.1.

Об'единением спецификаторов R и Q называется спецификатор, который обозначается R+Q и определяется рекурсивными соотношениями

(1) $\langle\rangle + Q = \langle\rangle$

(2) $[P\ R] + Q = P[R+Q]$

(3) $[(P)\ R] + Q = CONTR[P,Q]\ [R+Q]$

где P – произвольный первичный спецификатор, а R и Q – произвольные спецификаторы.

Т е о р е м а 6.1.

Если Q – полный спецификатор, то

$$YES[R+Q] = YES[R] + [NO[R]*YES[Q]].$$

Д о к а з а т е л ь с т в о.

Проводится индукцией по длине R.

Случай $R = \langle\rangle$. $YES[R+Q] = YES[\langle\rangle] = \langle\emptyset\rangle = YES[\langle\rangle] + [NO[\langle\rangle]*YES[Q]]$.

Случай $R = P\ R'$. $YES[R+Q] = YES[[P\ R']+Q] = YES[P\ [R'+Q]] = YES[P] + YES[R'+Q] = YES[P] + YES[R'] + [NO[R']*YES[Q]] = YES[P] + YES[R'] + [NO[R']*\neg YES[P]*YES[Q]] = YES[P\ R'] + [NO[P\ R']*YES[Q]] = YES[R] + [NO[R]*YES[Q]]$.

Случай $R = (P)R'$. Так как Q – полный, по теореме 5.5 $THRU[CONTR[P,Q]] = THRU[P] = \neg YES[P]$. Отсюда $YES[R+Q] = YES[[(P)R']+Q] = YES[CONTR[P,Q]\ [R'+Q]] = YES[CONTR[P,Q]] + [YES[R'+Q]*\neg YES[P]] = [YES[P]*YES[Q]] + [[YES[R'] + [NO[R']*YES[Q]]]*\neg YES[P]] = [YES[P]*YES[Q]] + [[YES[R']*\neg YES[P]] + [NO[R']*YES[Q]*\neg YES[P]]] = [YES[P]*YES[Q]] + [YES[R']*\neg YES[P]] + [\neg YES[P]*NO[R']*YES[Q]]$.

С другой стороны, $YES[R] + [NO[R]*YES[Q]] = [YES[R']*\neg YES[P]] + [[YES[P]+NO[R']]\ *\ YES[Q]] = [YES[R']*\neg YES[P]] + [YES[P]*YES[Q]] + [NO[R']*YES[Q]]$.

Но ведь NO[R'] * YES[Q] = [NO[R'] * YES[Q] * YES[P]] + [NO[R'] * YES[Q] * ¬YES[P]]. Учитывая, что NO[R'] * YES[Q] * YES[P] SUBSET YES[P] * YES[Q], получаем YES[R] + [NO[R] * YES[Q]] = [YES[P] * YES[Q]] + [YES[R'] * ¬YES[P]] + [NO[R'] * YES[Q] * ¬YES[P]] = YES[R+Q].

Теорема доказана.

Т е о р е м а 6.2.

Если R и Q - полные спецификаторы, то

YES[R+Q] = YES[R] + YES[Q].

Д о к а з а т е л ь с т в о.

NO[R] = ¬YES[R], ибо R - полный. Отсюда YES[R+Q] = YES[R] + [NO[R] * YES[Q]] = YES[R] + [¬YES[R] * YES[Q]] = YES[R] + YES[Q].

Т е о р е м а 6.3.

Если Q - полный спецификатор, то

THRU[R+Q] = THRU[R].

Д о к а з а т е л ь с т в о.

Индукция по длине R.

Случай R = <>. THRU[<>+Q] = THRU[<>].

Случай R = P R'. THRU[R+Q] = THRU[[P R'] +Q] = THRU[P [R'+Q]] = THRU[P] * THRU[R'+Q] = THRU[P] * THRU[R'] = THRU[P R'] = THRU[R].

Случай R = (P)R'. THRU[R+Q] = THRU[[(P)R'] +Q] = THRU[CONTR[P,Q] [R'+Q]] = THRU[CONTR[P,Q]] * THRU[R'+Q] = THRU[P] * THRU[R'] = THRU[P R'] = THRU[R].

Теорема доказана.

Т е о р е м а 6.4.

Если R и Q - полные спецификаторы, то и R+Q полный специ-

фикатор.

Доказательство.

Если R – полный, то THRU[R]=<Ø>. Поэтому THRU[R+Q] = THRU[R] = <Ø>.

Теорема 6.5. Если R и Q – полные спецификаторы, то

NO[R+Q] = NO[R] * NO[Q].

Доказательство.

R и Q – полные, поэтому, по теореме 6.4, R+Q – полный. Следовательно, NO[R+Q] = ¬YES[R+Q] = ¬[YES[R]+YES[Q]] = ¬YES[R]*¬YES[Q] = NO[R]*NO[Q].

1.7. ПЕРЕСЕЧЕНИЕ СПЕЦИФИКАТОРОВ

Определение 7.1.

Пересечением спецификаторов R и Q называется спецификатор, который обозначается R*Q и определяется соотношением

R * Q = ¬[¬R + ¬Q]

Теорема 7.1.

Для операции пересечения спецификаторов * справедливы следующие рекурсивные соотношения

(1) <> * Q = <>
(2) [P R] * Q = CONTR[P,Q] [R*Q]
(3) [(P) R] * Q = (P) [R*Q]

где P – произвольный первичный спецификатор, а R и Q – произвольные спецификаторы.

Доказательство.

(1) $\langle\rangle * Q = \neg[\neg\langle\rangle + \neg Q] = \neg[\langle\rangle + \neg Q] = \neg\langle\rangle = \langle\rangle$.

(2) $[P\ R]*Q = \neg[\neg[P\ R] + \neg Q] = \neg[(P)\neg R + \neg Q] = \neg[CONTR[P,\neg Q]\ [\neg R + \neg Q]] = CONTR[P,Q]\neg[\neg R + \neg Q] = CONTR[P,Q]\ [R * Q]$.

(3) $[(P)R]*Q = \neg[\neg[(P)R] + \neg Q] = \neg[[P\neg R] + \neg Q] = \neg[P\ [\neg R + \neg Q]] = (P)\neg[\neg R + \neg Q] = (P)\ [R * Q]$.

Теорема 7.2.

Если Q – полный спецификатор, то

$$NO[R*Q] = NO[R] + [YES[R] * NO[Q]].$$

Доказательство.

Q – полный, поэтому, по теореме 2.4, $\neg Q$ – тоже полный. Поэтому, применяя теорему 6.1, получаем $NO[R*Q] = YES[\neg[R*Q]] = YES[\neg R + \neg Q] = YES[\neg R] + [NO[\neg R] * YES[\neg Q]] = NO[R] + [YES[R] * NO[Q]]$.

Теорема 7.3.

Если R и Q – полные спецификаторы, то и R*Q – полный спецификатор.

Доказательство.

$R*Q = \neg[\neg R + \neg Q]$. R и Q – полные, поэтому, по теореме 2.4, $\neg R$ и $\neg Q$ – полные. Следовательно, по теореме 6.4 $\neg R + \neg Q$ – полный. Применяя еще раз теорему 2.4, заключаем, что $\neg[\neg R + \neg Q] = R*Q$ – полный.

Теорема 7.4.

Если R и Q – полные спецификаторы, то

$$NO[R*Q] = NO[R] + NO[Q].$$

Доказательство.

Так как R – полный, $YES[R] = \neg NO[R]$. Поэтому, применяя теорему 7.2, получаем $NO[R*Q] = NO[R] + [YES[R] * NO[Q]] = NO[R] + [\neg NO[R] * NO[Q]] = NO[R] + NO[Q]$.

Т е о р е м а 7.5.
Если R и Q - полные спецификаторы, то

$$YES[R*Q] = YES[R] * YES[Q].$$

Д о к а з а т е л ь с т в о.
R и Q - полные, поэтому, по теореме 7.3, R*Q - полный. Следовательно, справедливы соотношения $YES[R]=\neg NO[R]$, $YES[Q]=\neg NO[Q]$, $YES[R*Q]=\neg NO[R*Q]$. Поэтому, применяя теорему 7.4, получаем $YES[R*Q] = \neg NO[R*Q] = \neg[NO[R]+NO[Q]] = \neg NO[R]*\neg NO[Q] = YES[R]*YES[Q]$.

1.8. РАЗНОСТЬ СПЕЦИФИКАТОРОВ

О п р е д е л е н и е 8.1.
Разностью спецификаторов R и Q называется спецификатор, который обозначается R-Q и определяется соотношением

$$R - Q = R * \neg Q.$$

Т е о р е м а 8.1.
Если Q - полный спецификатор, то

$$NO[R-Q] = NO[R] + [YES[R] * YES[Q]].$$

Д о к а з а т е л ь с т в о.
Используя теорему 7.2, получаем $NO[R-Q] = NO[R*\neg Q] = NO[R] + [YES[R]*NO[\neg Q]] = NO[R] + [YES[R]*YES[Q]]$.

Т е о р е м а 8.2.
Если R и Q - полные спецификаторы, то и R-Q - полный спецификатор.
Д о к а з а т е л ь с т в о.
R-Q = $R*\neg Q$. Q - полный, поэтому, по теореме 2.4, $\neg Q$ - полный. Следовательно, по теореме 7.3, $R*\neg Q$ - полный.

Т е о р е м а 8.3.
Если R и Q - полные, то

YES[R-Q] = YES[R] - YES[Q].

Д о к а з а т е л ь с т в о.
По теореме 8.2, R-Q - полный, поэтому YES[R-Q]=¬NO[R-Q]. Применяя теорему 8.1 и учитывая, что R - полный, получаем YES[R-Q] = ¬NO[R-Q] = ¬[NO[R] + [YES[R]*YES[Q]] = ¬[NO[R] + [¬NO[R]*YES[Q]]] = ¬[NO[R]+YES[Q]] = YES[R]*¬YES[Q] = YES[R]-YES[Q].

1.9. РАСПОЗНАВАНИЕ ОТНОШЕНИЙ МЕЖДУ СПЕЦИФИКАТОРАМИ

При оптимизации рефал-программ требуется распознавать истинность следующих соотношений:

(1) YES[R] * YES[Q] = <Ø>
(2) YES[R] SUBSET YES[Q]
(3) YES[R] = YES[Q]

при условии, что R и Q - полные спецификаторы.
Проверку этих соотношений можно свести к проверке истинности соотношения YES[Q]=<Ø>, где Q - полный спецификатор.

Т е о р е м а 9.1.

(1) YES[R]*YES[Q]=<Ø> ⇔ YES[R*Q]=<Ø>.
(2) YES[R] SUBSET YES[Q] ⇔ YES[R-Q]=<Ø>.
(3) YES[R]=YES[Q] ⇔ YES[R-Q]=<Ø> & YES[Q-R]=<Ø>.

Д о к а з а т е л ь с т в о.
(1) очевидно, ибо по теореме 7.5 YES[R]*YES[Q] = YES[R*Q].
(2) YES[R] SUBSET YES[Q] ⇔ YES[R]*¬YES[Q]=<Ø>. Но ¬YES[Q]=YES[¬Q], ибо Q - полный. Поэтому, по теореме 7.5, YES[R]*¬YES[Q] = YES[R]*YES[¬Q] = YES[R*¬Q] = YES[R-Q].

(3) YES[R]=YES[Q] ⟺ [YES[R] SUBSET YES[Q]] ∈ [YES[Q] SUBSET YES[R]].

Теорема доказана.

1.10. РАСПОЗНАВАНИЕ ПУСТОТЫ ЗНАЧЕНИЯ СПЕЦИФИКАТОРА

В предыдущем разделе было установлено, что для распознавания истинности некоторых отношений достаточно уметь распознавать истинность соотношения

YES[Q] = <Ø>.

В этом разделе задача проверки соотношения YES[Q]=<Ø> будет сведена к еще более простой.

О п р е д е л е н и е 10.1.

Будем говорить, что конечное множество первичных спецификаторов <* P1,P2,...,PN *>, где N>=2, является р а з л о ж е н и е м первичного спецификатора P, если

YES[P] = YES[P1] + YES[P2] + ... + YES[PN].

Очевидно, что если <* P1,P2,...,PN *> - разложение первичного спецификатора P, то YES[P[I]] SUBSET YES[P] для I=1,2,...,N.

Теперь покажем, что если мы умеем для любого первичного P и любого конечного множества первичных спецификаторов <* P1,P2,...,PN *> распознавать истинность соотношения YES[P] = YES[P1]+YES[P2]+...+YES[PN], то мы можем построить и алгоритм распознавания соотношения YES[Q]=<Ø>.

Алгоритм распознавания YES[Q]=<Ø>.

Если Q не содержит положительные члены, т.е. Q = (P1)(P2)...(PN), где N>=Ø, то YES[Q] = YES[<>] - YES[P1 P2 ... PN] = <Ø>.

Предположим теперь, что Q содержит хотя бы один положительный член. Тогда его можно представить в виде

Q = (P1)(P2)...(PN) P R.

Получаем YES[Q] = YES[(P1)(P2)...(PN)] + [YES[P R] - NO[(P1)(P2)...(PN)]] = YES[P R] - YES[P1 P2 ... PN] = [YES[P] - YES[P1 P2 ... PN]] + [YES[R] - YES[P1 P2 ... PN]] = [YES[P] - YES[P1 P2 ... PN]] + YES[(P1)(P2)...(PN)R].

Таким образом,

YES[Q] = <Ø> <=> [YES[P]-YES[P1 P2 ... PN] =<Ø>] & [YES[(P1)(P2)...(PN)R] = <Ø>].

Поскольку (P1)(P2)...(PN)R короче, чем Q, для распознавания истинности соотношения YES[(P1)(P2)...(PN)R] = <Ø> можно рекурсивно применить тот же самый алгоритм.

Займемся теперь соотношением YES[P] - YES[P1 P2 ... PN] = <Ø>.

Если существует хоть одно P[I], такое, что YES[P] SUBSET YES[P[I]], то заведомо YES[P] - YES[P1 P2 ... PN] = <Ø>.

Если YES[P[I]]*YES[P] = <Ø> для некоторого P[I], то можно вычеркнуть P[I] из P1 P2 ... PN.

В конце концов, приходим к задаче распознавания истинности соотношения YES[P] - YES[P1 P2 ... PN] = <Ø>, где для всех I=1,2,...,N верно YES[P[I]] SUBSET YES[P]. Но при таких ограничениях это условие равносильно условию YES[P] = YES[P1]+YES[P2]+...YES[PN], т.е. проверке того, что <* P1,P2,...,PN *> является разложением первичного спецификатора P. А эту проверку мы, по предположению, делать умеем.

З а м е ч а н и е.

Если рассмотреть множество первичных спецификаторов, имеющихся в языке рефал, то оказывается, что распознавание того, что <* P1,P2,...,PN *> является разложением P, не составляет труда, поскольку разложения имеют только первичные спецификаторы W, S, D, L. А именно

```
vES[W] = vES[S B]
vES[S] = vES[O F N R]
vES[D] = vES['0123456789']
vES[L] = vES['ABCDEFGHIJKLMNOPQRSTUVWXYZ'
            'БГДЖЗИЙЛПУФЦЧШЩЬЭЮЯ']
```

1.11. СПЕЦИФИКАТОРЫ КАК ЭЛЕМЕНТЫ СПЕЦИФИКАТОРОВ

В описании рефала разрешается строить спецификаторы не только из первичных спецификаторов, но и из других, ранее определенных спецификаторов. Такое расширение языка спецификаторов для нас удобнее всего описывается следующим образом.

Будем называть элементарным спецификатором либо первичный спецификатор, либо конструкцию вида <: R :>, где R - спецификатор. Отрицанием элементарного спецификатора будем называть конструкцию вида (R), где R - элементарный спецификатор. Спецификатором назовем произвольную последовательность

```
Q1 Q2 ... QN
```

где каждое Q[I] - либо элементарный спецификатор, либо отрицание элементарного спецификатора.

Определим функцию vES, ставящую в соответствие каждому спецификатору Q множество об'ектных термов vES[Q], посредством рекурсивных соотношений:

```
(1)  vES[<>] = <Ø>
(2)  vES[P R]  =  vES[P] + vES[R]
(3)  vES[(P) R]  =  vES[R] - vES[P]
(4)  vES[<: Q :> R]  =  vES[Q] + vES[R]
(5)  vES[(<: Q :>) R]  =  vES[R] - vES[Q]
```

Оказывается, что для любого спецификатора Q, содержащего конструкции вида <: R :>, можно построить спецификатор Q', такой, что vES[Q]=vES[Q'] и при этом Q' не содержит конструкции вида <: R :>. Таким образом, оказывается, что

конструкция <: R :> не увеличивает изобразительную силу языка спецификаторов.

Определим с помощью рекурсивных соотношений операцию PLAIN, которая каждому спецификатору Q ставит в соответствие спецификатор PLAIN[Q], не содержащий конструкций вида <: R :>.

(1) PLAIN[<>] = (W)
(2) PLAIN[P R] = [P (W)] + PLAIN[R]
(3) PLAIN[(P) R] = PLAIN[R] - [P (W)]
(4) PLAIN[<: Q :> R] = PLAIN[Q] + PLAIN[R]
(5) PLAIN[(<: Q :>) R] = PLAIN[R] - PLAIN[Q]

где P - первичный спецификатор, W - универсальный спецификатор, для которого YES[W] = OTERM, Q и R - спецификаторы.

Т е о р е м а 10.1.
Для любого спецификатора Q, PLAIN[Q] - полный спецификатор.

Д о к а з а т е л ь с т в о.
Согласно теореме 4.2, любой спецификатор, который кончается на (W), является полным. Согласно теоремам 6.4 и 8.2, об'единение и разность полных спецификаторов снова являются полными спецификаторами. Поэтому, применяя индукцию по длине Q, получаем утверждение теоремы.

Т е о р е м а 10.2. Для любого спецификатора Q

YES[Q] = YES[PLAIN[Q]]

Д о к а з а т е л ь с т в о.
Проводится индукцией по длине Q.
Используя теорему 10.1, теоремы 6.4 и 8.2 и индуктивное предположение, получаем.
(1) YES[PLAIN[<>]]= YES[(W)] = <Ø> = YES[<>].
(2) YES[PLAIN[P R] = YES[[P (W)] + PLAIN[R]] = YES[P (W)] + YES[PLAIN[R]] = YES[P]+YES[R] = YES[P R].
(3) YES[PLAIN[(P) R]] = YES[PLAIN[R] - [P (W)]] =

YES[PLAIN[R]] - YES[P (W)] = YES[R]-YES[P] = YES[(P) R].

(4) YES[PLAIN[<: Q :> R]] = YES[PLAIN[Q] + PLAIN[R]] = YES[PLAIN[Q]] + YES[PLAIN[R]] = YES[Q] + YES[R] = YES[<: Q :> R].

(5) YES[PLAIN[(<: Q :>) R]] = YES[PLAIN[R] - PLAIN[Q]] = YES[PLAIN[R]] - YES[PLAIN[Q]] = YES[R] - YES[Q] = YES[(<: Q :>) R].

Теорема доказана.

ГЛАВА 2. АЛГОРИТМЫ ОТОЖДЕСТВЛЕНИЯ

2.1. АКСИОМАТИЧЕСКИЙ И ПРОЦЕДУРНЫЙ ПОДХОДЫ К СИНТАКСИЧЕСКОМУ ОТОЖДЕСТВЛЕНИЮ

В описании рефала процесс синтаксического отождествления определяется следующим образом.

О п р е д е л е н и е 1.1.

Будем говорить, что об'ектное выражение M может быть синтаксически отождествлено как типовое выражение L, если переменные в L можно заменить на такие выражения, именуемые их значениями, что M совпадет с L. При этом значения переменных должны удовлетворять следующим требованиям:

(1) Значением S-переменной может быть только символ, значением W-переменной - только терм, значением V-переменной - непустое выражение, значением E-переменной - произвольное выражение (которое может быть пустым).

(2) Все вхождения одной и той же переменной должны принимать одинаковые значения.

(3) Если некоторое вхождение переменной имеет спецификатор, то все нуль-термы значения этой переменной (т.е. термы, входящие в ее значение на нулевом уровне скобок) должны удовлетворять этому спецификатору.

Например, если

L = E1 SX E2, M = 'A' ('*' ()) 'B',

то отождествление M как L можно произвести двумя способами. При первом способе отождествления переменные получают следующие значения (где < > обозначает пустое выражение)

E1 ← <>, SX ← 'A', E2 ← ('*'())'B'

При втором способе отождествления переменные получают следующие значения

E1 ← 'A'('*'()), SX ← 'B', E2 ← <>.

Если

L = SX E1, M = ('A')'B',

то отождествление M как L невозможно.

В дальнейшем будем обозначать через W*[L,M] множество всех способов отождествления M как L.

О п р е д е л е н и е 1.2. (Правило отождествления.)
Пусть W*[L,M]¬=<Ø>. Пусть X1, X2,...,XN - VE-переменные, входящие в L, перечисленные в том порядке, как они входят в L. Построим последовательность множеств WØ, W1,...,WN, обладающую следующими свойствами.

(1) WØ = W*[L,M].

(2) для всех способов отождествления, входящих в WK, значения переменных X1,X2,...,XK не зависят от способа отождествления.

(3) WK SUBSET W[K-1], причем в WK входят только такие способы отождествления из W[K-1], для которых XK принимает самое короткое значение.

Ясно, что WN содержит ровно один способ отождествления. Будем называть его с а м ы м л е в ы м в W*[L,M] и обозначать RCG#[L,M].

Таким образом, правило отождествления определяет частичную функцию RCG#[L,M], которая не определена при W*[L,M]=<Ø> и определена при W*[L,M]¬=<Ø>, причем RCG#[L,M] IN W*[L,M]. Впрочем, в некоторых случаях нам будет удобнее

считать, что функция RCG# всюду определена, но при W#[L,M]=<Ø> вырабатывает неопределенное значение UNDEF.

Описанное выше правило отождествления определяет RCG#[L,M] однозначно, поэтому оно вполне пригодно для теоретического описания рефала. Однако, при практической реализации рефала возникает необходимость разработать достаточно эффективный а л г о р и т м для вычисления RCG#[L,M]. Такой алгоритм мы будем в дальнейшем называть а л г о р и т м о м о т о ж д е с т в л е н и я.

Ясно, что можно придумать много различных алгоритмов отождествления, которые могут различаться по сложности и эффективности. Наша ближайшая задача - рассмотреть основные способы построения таких алгоритмов. Для этого нам прежде всего нужно произвести дальнейшее уточнение и формализацию введенных выше понятий.

2.2. ПОДСТАНОВКИ

Поскольку результатом работы RCG#[L,M] должен быть некоторый способ отождествления M как L, необходимо как-то "материализовать" понятие способа отождествления, т.е. изобразить способы отождествления в виде каких-то об'ектов.

Видимо, это можно сделать простейшим способом, представив отождествления в виде подстановок.

О п р е д е л е н и е 2.1.

Будем называть п о д с т а н о в к о й любое конечное множество упорядоченных пар вида (X,B), где X - переменная, а B - об'ектное выражение.

Если D - подстановка, то через VARS[D] будем обозначать множество таких переменных X, для которых существует упорядоченная пара вида (X,B), входящая в D.

Если L - типовое выражение, то через VARS[L] будем обозначать множество переменных, входящих в L.

О п р е д е л е н и е 2.2.
Подстановка D называется с о в м е с т н о й, если выполнены следующие условия.

(1) для любой переменной X из VARS[D] существует ровно одна пара вида (X,B), входящая в D.

(2) для любой пары (X,B), входящей в D, об'ектное выражение B является допустимым значением для переменной X.

Теперь определим результат применения подстановки D к типовому выражению L.
Пусть B - об'ектное выражение. Через ZT[B] будем обозначать множество составляющих его нуль-термов. Т.е. если B = T1 T2 ... TN, где T1,T2,...,TN - термы, то ZT[B] = <* T1, T2,...,TN *>.

О п р е д е л е н и е 2.3.
Будем говорить, что подстановка D применима к типовому выражению L, если выполнены следующие условия.

(1) D - совместна.
(2) VARS[L] SUBSET VARS[D].
(3) для любого вхождения переменной X в L, имеющего спецификатор Q, из того, что (X,B) IN D, следует, что все нуль-термы выражения B удовлетворяют этому спецификатору, т.е. что ZT[B] SUBSET YES[Q].

О п р е д е л е н и е 2.4.
Пусть подстановка D применима к типовому выражению L. Тогда результат применения D к L обозначается через SUBST[D,L] и определяется с помощью соотношений

(1) SUBST[D,<>] = <>.
(2) SUBST[D,Z] = Z.
(3) SUBST[D,A' A''] = SUBST[D,A'] SUBST[D,A''].
(4) SUBST[D,X] = B, ГДЕ (X,B) IN D.

где через A' и A'' обозначены произвольные типовые выражения, через Z - произвольный символ, через X - произвольная переменная, а через B - произвольное об'ектное выражение.

Таким образом, мы определили частичную функцию SUBST[D,L], которая определена тогда и только тогда, когда D применима к L. Впрочем, в некоторых случаях нам будет удобнее считать, что SUBST всюду определена, но если D не применима к L, принимает неопределенное значение UNDEF.

2.3. ОТОЖДЕСТВЛЕНИЯ

Будем говорить, что подстановка D является *отождествлением* об'ектного выражения M как типового выражения L, если SUBST[D,L]=M.

Поскольку SUBST[D,L] определена только если D применима к L, отождествлением M как L может быть только совместная подстановка, для которой VARS[L] SUBSET VARS[D].

Теперь мы можем более точно об'яснить смысл обозначения W#[L,M], введенного в 2.1. А именно, будем считать, что W#[L,M] - это множество таких подстановок D, для которых SUBST[D,L]=M и при этом VARS[D]=VARS[L].

Таким образом, в силу определения 2.1, RCG#[L,M] обозначает самое левое отождествление из числа входящих в W#[L,M].

Очевидно, что множество W#[L,M] конечно, поскольку для всех D из W#[L,M] множество VARS[D] = VARS[L] - конечно, а для любой пары (X,B) из D выражение B является некоторым подвыражением выражения M (не исключая случай B=M). Поэтому можно найти W#[L,M], а значит - и RCG#[L,M] полным перебором.

Следовательно, функция RCG#[L,M] - вычислима. Нас, однако, интересуют не любые алгоритмы отождествления, а только те, которые работают достаточно эффективно.

Как это часто случается в математике, оказывается, что вместо того, чтобы решать исходную задачу, удобно эту задачу обобщить, и исходную задачу рассматривать как частный случай

обобщенной задачи.

Первый шаг обобщения будет состоять в том, что мы будем решать задачу поиска отождествления при условии, что наложены дополнительные ограничения на множество искомых подстановок. А именно, будем считать, что значения некоторых переменных жестко зафиксированы. Это означает, что все искомые подстановки должны содержать в качестве подмножества некоторую подстановку C.

Второй шаг обобщения будет состоять в том, что вместо отождествления одного об'ектного выражения M как типового выражения L мы будем рассматривать отождествление упорядоченной N-ки (кортежа) об'ектных выражений (M1,M2,...,MN) как упорядоченной N-ки типовых выражений (L1,L2,...,LN). Это обобщение, впрочем, менее существенно, ибо отождествление кортежей эквивалентно отождествлению выражения (M1)(M2)...(MN) как (L1)(L2)...(LN).

Чтобы говорить об отождествлении кортежей, удобно расширить область определения функции SUBST, с тем, чтобы ее можно было применять не только к типовым выражениям, но и к кортежам из типовых выражений.

О п р е д е л е н и е 3.1

Пусть L - кортеж из типовых выражений L= (L1,L2,...,LN). Будем говорить, что подстановка D применима к кортежу L, если она применима к каждому из типовых выражений L1, L2,...,LN.

Если подстановка D применима к кортежу L, то результат применения D к L обозначается через SUBST[D,L] и определяется с помощью соотношения:

$$SUBST[D,L] = (SUBST[D,L1], SUBST[D,L2], \ldots, SUBST[D,LN])$$

О п р е д е л е н и е 3.2.

Пусть задана упорядоченная N-ка типовых выражений L=(L1,L2,...,LN), именуемых *д н р а м и*, и упорядоченная N-ка об'ектных выражений M=(M1,M2,...,MN), именуемых *о б р а з а м и*, а также подстановка G. Будем говорить, что подстановка D является отождествлением M как L, при

ограничении G, если выполнены условия

(1) D - СОВМЕСТНА.
(2) G SUBSET D.
(3) SUBST[D,L] = M.

Из этого определения следует, что D - совместная подстановка, для которой

VARS[L1 L2 ... LN] + VARS[G] SUBSET VARS[D].

Для дальнейшего удобно ввести следующие обозначения.

Для кортежей L=(L1,L2,...,LN) и M=(M1,M2,...,MN) через VARS[L] будем обозначать множество переменных, входящих в L, т.е. VARS[L1 L2 ... LN], а через L#M будем обозначать кортеж из упорядоченных пар

((L1,M1),(L2,M2),...,(LN,MN)).

Поскольку такая запись довольно громоздка, будем для L#M использовать сокращенное обозначение

(L1,M1)(L2,M2)...(LN,MN).

Через VARS[L#M] будем обозначать множество переменных, входящих в L, т.е. VARS[L#M]=VARS[L].

В дальнейшем нам будет удобно использовать операцию к о н к а т е н а ц и и (сцепления) кортежей. Если A=(A1,A2,...,AM) - кортеж длины M, а B=(B1,B2,...,BN) - кортеж длины N, то их конкатенация

A B = (A1,A2,...,AM,B1,B2,...,BN)

является кортежем длины M+N.

Теперь мы можем обобщить введенное ранее обозначение W#[L,M].

Через W[G,L#M] обозначим множество таких отождествлений D кортежа M как кортежа L при ограничении G, что

VARS[L]+VARS[G] = VARS[D].

Чтобы обобщить введенное ранее обозначение RCG#[L,M], мы должны определить, какой из элементов множества W[G,L#M] является "самым левым". Для этого необходимо ввести на множестве W[G,L#M] отношение линейного порядка.

О п р е д е л е н и е 3.3.

Пусть A - типовое выражение (которое, в частности, может быть и об'ектным выражением), а T1, T2,...,TN - такие термы, что A= T1 T2 ... TN. Тогда нуль-длиной выражения a называется целое число N.

Нуль-длину выражения A будем обозначать через ZL[A].

О п р е д е л е н и е 3.4.

Пусть L - кортеж типовых выражений, а D' и D'' - подстановки, применимые к L. Пусть X1, X2,...,XN - VE-переменные, входящие в L и выписанные в том порядке, как они входят в L.

Будем говорить, что D' эквивалентна D'' на L, если для всех i=1,2,...,N выполнено

ZL[SUBST[D',X[i]]] = ZL[SUBST[D'',X[i]]].

Утверждение о том, что D' эквивалентна D'' на L, будем записывать в виде D' EQU[L] D''.

Будем говорить, что D' левее, чем D'' на L, если существует такое целое K, что

ZL[SUBST[D',X[K]]] < ZL[SUBST[D'',X[K]]]

и при этом для всех i<K верно

ZL[SUBST[D',X[i]]] = ZL[SUBST[D'',X[i]]].

Утверждение о том, что D' левее, чем D'' на L, будем записывать в виде D'<[L] D''.

Утверждение о том, что D' EQU[L] D'' или D'< [L] D'' будем записывать в виде D'< =[L] D''.

Если L#M - кортеж из пар выражений, то EQU[L#M], <[L#M] и

<=[L#M] будут обозначать те же отношения, что и EQU[L], <[L] и <=[L] соответственно.

Т е о р е м а 3.1.

Пусть <* X1, X2,...,XN > множество VE-переменных, входящих в кортеж типовых выражений L. Пусть заданы целые числа K1, K2,...,KN. Тогда может существовать не более, чем одна подстановка D из W[G,L#M], такая, что для всех I=1,2,...,N выполнено ZL[SUBST[D,X[I]] = K[I].

Д о к а з а т е л ь с т в о.

Нуль-длины для значений всех VE-переменных заданы, а нуль-длины для значений всех S- и W-переменных всегда равны единице. Отсюда очевидно, что если сопоставить L и M, то можно однозначно найти для каждого вхождения каждой переменной в L то подвыражение из M, с которым это вхождение должно совпасть после применения подстановки.

Т е о р е м а 3.2.

Если D' и D'' принадлежат множеству W[G,L#M] и D' EQU[L] D'', то D'=D''.

Д о к а з а т е л ь с т в о.

Во-первых, заметим, что VARS[D'] = VARS[D''] = VARS[G]+VARS[L]. Во-вторых, из D' EQU[L] D'' следует, что для любой VE-переменной X из VARS[L] выполнено ZL[SUBST[D',X]] = ZL[SUBST[D'',X]]. Поэтому, применяя теорему 3.1, получаем, что D'=D''.

Т е о р е м а 3.3.

Отношение <=[L] является линейным порядком на множестве W[G,L#M].

Д о к а з а т е л ь с т в о.

Из определения 3.4 непосредственно следует, что отношение <=[L] рефлексивно и транзитивно. Покажем, что на множестве W[G,L#M] оно еще и антисимметрично.

Пусть D' и D'' принадлежат W[G,L#M]. Тогда D' и D'' заведомо применимы к L. Поэтому, из определения 3.4 следует, что обязательно истинно одно и только одно из следующих утверждений: D'<[L] D'', D' EQU[L] D'', D''<[L] D'.

Согласно теореме 3.2, из D' EQU[L] D'' следует, что D'=D''. Поэтому из одновременной истинности D' <=[L] D'' и D'' <=[L] D' следует, что D'=D''.

Теорема доказана.

О п р е д е л е н и е 3.5. (Правило отождествления.)
Пусть G - подстановка, а L#M - кортеж из пар выражений. Тогда функция RCG[G,L#M] определена следующими условиями.

(1) если W[G,L#M]=<Ø>, то RCG[G,L#M]=UNDEF.

(2) если W[G,L#M]¬=<Ø>, то RCG[G,L#M] равна самому левому отождествлению из W[G,L#M], точнее тому отождествлению, которое является наименьшим в смысле отношения линейного порядка <=[L].

Таким образом, RCG - функция, которая может вырабатывать значение UNDEF. В дальнейшем нам будет удобно считать, что UNDEF может употребляться в операндах теоретико-множественных операций об'единения, пересечения и разности. При этом, будем считать, что если хоть одним из операндов такой операции является UNDEF, то значением операции тоже является UNDEF. Например, если A - множество, то A+UNDEF = UNDEF+A = A*UNDEF = UNDEF*A = UNDEF.

Очевидно, что только что определенная функция RCG[G,L#M] является, в некотором смысле, обобщением введенной в п.2.1 функции RCG#, вычислявшей самое левое отождествление об'ектного выражения B как типового выражения A, поскольку

RCG#[A,B] = RCG[<Ø>, ((A,B))] .

2.4. АЛГОРИТМ ОТОЖДЕСТВЛЕНИЯ

Теперь мы можем изложить основные принципы, на которых основан эффективный (по сравнению с полным перебором) алгоритм отождествления.

Основная идея алгоритма отождествления заключается в сведении вычисления RCG[G,L#M] к вычислению RCG[G',L'#M'], где L'#M' содержит меньше элементов, чем L#M. В конце концов кортеж L#M становится пустым и работа алгоритма отождествления заканчивается, поскольку RCG[G,<>]=G (при условии, что G - совместная подстановка).

Процесс сведения задачи вычисления RCG[G,L#M] к более простым основан на следующих теоремах.

Т е о р е м а 4.1.

Пусть C - подстановка, A' и A'' - типовые выражения, B' и B'' - об'ектные выражения, а

$$L\#M = X\ (A'\ A'',\ B'\ B'')\ Y$$

кортеж из пар выражений. Тогда, если для любой подстановки D из W[G,L#M] выполнено SUBST[D,A']=B' и SUBST[D,A'']=B'', то

$$RCG[G,X\ (A'\ A'',\ B'\ B'')\ Y] = RCG[G,X\ (A',B')(A'',B'')\ Y].$$

Д о к а з а т е л ь с т в о.

Очевидно, что при сформулированных условиях W[G,X (A'A'',B'B'') Y] = W[G,X (A',B')(A'',B'') Y].

Кроме того, отношение <=[X (A'A'',B'B'') Y] равносильно отношению <=[X (A',B')(A'',B'') Y], ибо не изменяется порядок, в котором следуют друг за другом вхождения VE-переменных.

Т е о р е м а 4.2.

Пусть G - подстановка, B' и B'' - об'ектные выражения, L#M = X (B',B'') Y. Тогда, если B'=B'', то RCG[G,L#M] = RCG[G,X Y], а если B'¬=B'', то RCG[G,L#M] = UNDEF.

Д о к а з а т е л ь с т в о. Очевидно.

Т е о р е м а 4.3.

Пусть G - подстановка, A - типовое выражение, B - об'ектное выражение и L#M = X ((A),(B)) Y. Тогда

RCG[G,L#M] = RCG[G,X (A,B) Y].

Д о к а з а т е л ь с т в о. Очевидно.

Т е о р е м а 4.4.

Пусть G - подстановка, U(Q)X - вхождение переменной со спецификатором Q, B - об'ектное выражение и

L#M = X (U(Q)X,B) Y

Тогда, если ZT[B] SUBSET YES[Q], то

RCG[G,L#M] = RCG[G+<* (UX,B) *>, X Y]

В противном случае, RCG[G,L#M]=UNDEF.

Д о к а з а т е л ь с т в о.

Очевидно, если принять во внимание, что для любой несовместной подстановки G' и кортежа из пар выражений L'#M' верно RCG[G',L'#M'] =UNDEF.

Т е о р е м а 4.5.

Пусть G - подстановка, U - вхождение YE-переменной, A - типовое выражение, B - об'ектное выражение и

L#M = (U A,B) X.

Пусть T1, T2,...,TN - такие термы, что B = T1 T2 ... TN. Обозначим B[I]= T1 T2 ... T[I], C[I] = T[I+1] ... T[N],

D[İ] = RCG[G,(U,B[İ])(A,C[İ]) X].

Тогда, если для всех İ=Ø,1,2,...,N выполнено D[İ]=UNDEF, то RCG[G,L#M]=UNDEF. В противном случае, обозначим через K наименьшее целое число, для которого D[K]¬=UNDEF. Тогда RCG[G,L#M]=D[K].

Доказательство.

Очевидно из определения 3.5.

Символы, скобки и переменные, входящие в L и M будем называть элементами дыр и образов.

В процессе работы алгоритма отождествления элементы дыр постепенно сопоставляются с элементами образов.

Основной шаг алгоритма отождествления состоит в том, чтобы выбрать в какой-то дыре из L какой-то элемент и сопоставить ему часть образа соответствующей дыры, именуемую в дальнейшем образом этого элемента. После этого элемент удаляется из L, а его образ - из M. В результате этого размер кортежа L#M уменьшается. Если сопоставленный элемент из L является какой-то переменной X, а образом этого элемента является выражение B, то в C добавляется пара (X,B).

Пусть, например, требуется вычислить

RCG[<* (EX,'AA') *>, (EX SY EI, 'AABC')] .

Алгоритм отождествления "рассуждает" следующим образом.

Первая (и единственная) дыра начинается с E-переменной, значением которой должно быть 'AA'. В то же время, образ дыры тоже начинается с 'AA'. Следовательно, вхождение переменной EX для всех отождествлений должно сопоставляться с первыми двумя символами образа дыры. Поэтому задачу можно свести к вычислению

RCG[<* (EX,'AA') *>, (SY EI, 'BC')] .

Теперь оказывается, что дыра начинается с переменной SY, а образ дыры - с символа 'B'. Значением S-переменной SY может быть только символ, стало быть, для всех

отождествлений SY должна сопоставляться с символом 'B'. Поэтому задача сводится к вычислению

```
RCG[<* (EX,'AA'),(SY,'B') *>, (EI,'C')] .
```

Теперь выясняется, что для любого отождествления значением переменной EI может быть только 'C'. В результате, задача сводится к вычислению

```
RCG[<* (EX,'AA'),(SY,'B'),(EI,'C') *>,<>] ,
```

результатом которого является подстановка

```
<* (EX,'AA'),(SY,'B'),(EI,'C') *>
```

Итак мы видели, что в процессе отождествления элементы из L сопоставляются с элементами из M последовательно, один за другим. Поэтому элементы из L можно пронумеровать в том порядке, в котором выполняется их сопоставление. Эти номера мы будем называть п р о е к ц и о н н ы м и номерами.

Для только что рассмотренного примера элементы из L получают следующие проекционные номера:

```
1  2  3
EX SY EI
```

В дальнейшем мы будем рассматривать только такие алгоритмы отождествления, для которых очередность сопоставления элементов зависит от VARS[G] и L, но не зависит от M. Это означает, что проекционные номера однозначно определяются кортежем L и множеством VARS[G], но совершенно не зависят от M.

Если некоторая переменная входит в L несколько раз, то каждому ее вхождению будет соответствовать свой проекционный номер.

Если некоторая переменная не входит в VARS[G], то ее вхождение, имеющее наименьший проекционный номер, будем называть г л а в н ы м вхождением. Все вхождения

переменных, которые не являются главными, мы будем называть **повторными**.

Символы, скобки, вхождения S-переменных, W-переменных, а также повторные вхождения VE-переменных мы будем называть **жесткими** элементами.

Жесткие элементы обладают следующим свойством. Если такой элемент находится на левом или правом конце дыры, то его сопоставление с образом дыры либо вообще невозможно, либо выполнимо не более чем одним способом и, следовательно, должно быть одинаковым для всех отождествлений. Это свойство очевидно по отношению к символам, скобкам и S-переменным. Что касается вхождения W-переменной, то оно может быть сопоставлено либо с символом, либо с выражением в скобках. В последнем случае парная скобка находится в образе дыры однозначно по правилам скобочного синтаксиса. И наконец, повторное вхождение VE-переменной сопоставляется однозначно потому, что в момент сопоставления значение переменной уже известно.

Если главное вхождение VE-переменной в момент сопоставления находится одновременно и на левом, и на правом конце дыры (т.е. вся дыра состоит только из этой переменной), то это вхождение будет называться **закрытым**, в противном случае - **открытым**.

Жесткие элементы и закрытые вхождения VE-переменных мы будем называть **однозначно сопоставимыми** элементами. Все алгоритмы отождествления, которые мы будем рассматривать в дальнейшем, будут придерживаться следующих правил при определении очередности сопоставления элементов.

Будем говорить, что элемент **доступен** для сопоставления, если он находится на левом или на правом конце дыры.

Подвергаться сопоставлению могут только те элементы, которые доступны для сопоставления. Из этого правила есть только одно исключение, касающееся скобок.

Для каждой скобки, входящей в некоторое выражение, всегда можно однозначно найти парную к ней скобку. Можно считать, что парные скобки и в L, и в M, как бы жестко скреплены

между собой. Поэтому, как только мы сопоставим какую-то скобку из L со скобкой из M, мы можем сразу же отыскать в L и M парные скобки и сопоставить их между собой.

Именно так и будет поступать алгоритм отождествления: каждые две парные скобки из L будут одновременно сопоставляться с двумя парными скобками из M. Таким образом, будет происходить следующее сведение задачи к более простой:

```
RCG[G, X ((A')A'', (B')B'') Y] =
RCG[G, X (A'(A''), B'(B'')) Y] =
RCG[G, X (A',B') (A'',B'') Y]
```

где A',A'' - типовые выражения, B',B'' - об'ектные выражения, X,Y - кортежи из упорядоченных пар выражений.

При таком сведении задачи количество дыр (и их образов) увеличивается, однако при этом все же уменьшается количество элементов, входящих в L и M, и, в этом смысле, задача сводится к более простой.

Выбор очередного элемента из L для сопоставления производится следующим образом.

Сначала алгоритм отождествления находит все элементы из L доступные для сопоставления. Затем доступные элементы делятся на две группы очередности. В первую группу входят все однозначно сопоставимые элементы, а во вторую - открытые вхождения VE-переменных, т.е. вхождения VE-переменных, не входящих в VARS[G] и не стоящих одновременно на правом и левом концах дыры.

Алгоритм отождествления стремится в первую очередь сопоставлять элементы из первой группы, и только если их нет - сопоставляет элементы из второй группы.

Если в первой группе оказалось несколько элементов, может быть выбран любой из них. (Во всех примерах мы, для определенности, будем выбирать тот из них, который в L расположен левее, чем остальные.) Если же элементы принадлежат второй группе, то всегда выбирается самый левый из них.

Найдя нужный элемент, алгоритм отождествления пытается его сопоставить.

Если элемент - однозначно сопоставимый, то попытка его

сопоставить, либо терпит неудачу, либо проходит успешно. Если сопоставление проходит успешно, алгоритм отождествления вносит надлежащие изменения в C и L#M и переходит к сопоставлению следующих элементов. Если же сопоставление элемента терпит неудачу, то алгоритм отождествления вырабатывает результат UNDEF и прекращает работу.

Сложнее обстоит дело с открытыми вхождениями VE-переменных. Здесь алгоритм отождествления уже не может решить сразу, каким должно быть значение VE-переменной, поэтому ему приходится перебирать несколько возможностей.

Пусть, например, сопоставляемый элемент является открытым вхождением E-переменной. Тогда L#M непременно имеет следующий вид:

L#M = (E(Q)X A,B) X

Пусть B = T1 T2 ... TN, где T1 T2 ... TN - об'ектные термы. Обозначим B[K] = T1 T2 ... TK, C[K] = T[K+1] ... TN.

Алгоритм отождествления пытается подобрать наименьшее K, для которого

G[K] = RCG[G+<* (EX,B[K]) *>, (A,C[K]) X]

не равно UNDEF, и для которого ZT[B[K]] SUBSET YES[Q]. Если такое K существует, то результатом работы алгоритма будет G[K]. Если же не существует, то результатом работы алгоритма будет UNDEF.

Случай сопоставления открытого вхождения V-переменной рассматривается аналогично, но при этом рассматриваются только случаи когда K=1,2,...,N.

В результате описанного выше перебора всегда будет найдено самое левое отождествление, поскольку перебор происходит по самому левому вхождению VE-переменной в L, а возможные значения переменной рассматриваются в порядке возрастания их длин.

Рассмотрим теперь несколько примеров отождествления M как L при ограничении G=<Ø>. Во всех примерах кортеж дыр

L=(L1,L2,...,LN) для краткости будем записывать в виде L1,L2,...,LN.

Пусть L имеет вид

```
 1  2 4   5  6 8   9  7 3
'B' ( SI 'A' ( W2 'C' ) )
```

где над каждым элементом написан его проекционный номер.

Поскольку все элементы - жесткие, отождествление в целом терпит неудачу, как только терпит неудачу попытка сопоставления хотя бы одного элемента.

В процессе отождествления L меняется следующим образом:

```
'B' (SI 'A' (W2 'C'))
(SI 'A' (W2 'C'))
SI 'A' (W2 'C') , < >
'A' (W2 'C') , < >
(W2 'C') , < >
W2 'C' , < > , < >
'C' , < > , < >
< > , < >
< >
```

В следующее выражение

```
 1  4  3 5  6 8  7 9  10  11 2
'A' E1 ( SA ( E2 ) WB '=' E3 )
```

входят три E-переменные, но все они - закрытые, поэтому сопоставление, как и в предыдущем примере, проходит без всякого перебора. Время, затрачиваемое на отождествление, не зависит от M.

В процессе отождествления L меняется следующим образом:

```
'A' E1 (SA (E2) WB '=' E3)
E1 (SA (E2) WB '=' E3)
E1 , SA (E2) WB '=' E3
SA (E2) WB '=' E3
```

```
(E2) WB '=' E3
E2 , WB '=' E3
WB '=' E3
'=' E3
E3
```

Теперь рассмотрим выражение

```
I   2  3
E1 '+' E2
```

в котором переменная E1 - открытая, а E2 - закрытая. Алгоритм отождествления будет удлинять значение переменной E1 до тех пор, пока не подберет для E1 кратчайшее значение, при котором возможно отождествление.

В следующем выражении

```
I 5   6  7  2 8   9  I0 4 I1 I2  I3 3
( E1 '+' E2 ) E3 '+' E4 ( E5 '+' E6 )
```

имеется три открытых вхождения переменных E1, E3 и E5 и три закрытых вхождения переменных E2, E4 и E6.

В двух последних примерах время, затрачиваемое на отождествление, зависит от M, ибо необходим перебор при сопоставлении открытых вхождений E-переменных.

Рассмотрим еще один пример:

```
I0 I1  I2 2 3  4  6   7  5 9   8  I
E1 '+' E2 ( W3 ( '*' '*' ) E1 'B' )
```

На первый взгляд может показаться, что первое вхождение переменной E1 - открытое. В действительности же оно - повторное. Сопоставляя жесткие элементы, мы добираемся до вхождения E1, которое является закрытым (элемент N 9). Теперь элемент N I0 становится жестким и, следовательно, однозначно сопоставимым. Таким образом, отождествление производится без всякого перебора. Однако, время, затрачиваемое на отождествление, зависит от значения переменной E1,

ибо при сопоставлении повторного вхождения, приходится просматривать элемент за элементом образ этого вхождения.

В общем случае время, затрачиваемое на отождествление M как L, зависит как от L, так и от M, и может быть грубо оценено как C*(LM**N), где C - константа, LM - суммарная длина выражений в L, а N - количество открытых и повторных вхождений VE-переменных в L.

Теперь, после данных выше раз'яснений, мы можем описать алгоритм отождествления, т.е. алгоритм вычисления RCG[G,L#M].

Алгоритм вычисления RCG[G,L#M]

RCG. В момент обращения, алгоритм получает параметры:

G - ограничения на значения переменных (подстановка).
L = (L1,L2,...,LN) - кортеж дыр.
M = (M1,M2,...,MN) - кортеж образов.

(Будем считать для единообразия, что все вхождения переменных имеют спецификатор, ибо отсутствие спецификатора эквивалентно наличию универсального спецификатора W.)

Если G - несовместная подстановка, то RETURN[UNDEF].

Если N=Ø, т.е. L и M - пустые кортежи, то RETURN[G].

HSCH. (Поиск дыры.) Если все дыры имеют вид
U(Q')1 A U(Q'')2
где U1, U2 - VE-переменные, не входящие в VARS[G], а A - типовое выражение, то перейти к LVE, иначе найти дыру, которая не имеет указанного вида и перейти к HARD.

HARD. (Сопоставление жестких элементов.) Пусть L#M имеет вид

L#M = X (A,B) Y

где A - дыра, найденная в пункте HSCH, а B - ее образ.

Если A=<>, то перейти к NIL.

Если A = E(Q)X, где EX - E-переменная, не входящая в VARS[G], то перейти к CE.

Если A = V(Q)X, где VX - V-переменная, не входящая в VARS[G], то перейти к CV.

(В этом месте известно, что дыра A не может и начинаться, и кончаться на VE-переменные, не входящие в VARS[G].)

Выбрать такой конец дыры A, на котором не находится VE-переменная, не входящая в VARS[G]. Если выбран левый конец, то перейти к HARDL, если выбран правый конец - перейти к HARDR.

NIL. (Сопоставление пустой дыры.) Если B=<>, то RETURN[RCG[G,X Y]], иначе RETURN[UNDEF].

CE. (Сопоставление закрытого вхождения E-переменной.) Если ZT[B] SUBSET YES[Q], то RETURN[RCG[G+<* (EX,B) *>, X Y]], иначе RETURN[UNDEF].

CV. (Сопоставление закрытого вхождения V-переменной.) Если B=<>, то RETURN[UNDEF]. Если ZT[B] SUBSET YES[Q], то RETURN[RCG[G+<* (VX,B) *>, X Y]], иначе RETURN[UNDEF].

HARDL. (Жесткий элемент слева.)

Если A= (A')A'', то перейти к LB.

Если A= Z A, где Z - символ, то перейти к LSC.

Если A= S(Q)X A', где SX - S-переменная, то перейти к LS.

Если A= W(Q)X A', где WX - W-переменная, то перейти к LW.

Если A= E(Q)X A', где EX - E-переменная, то перейти к LED.

Если A= V(Q)X A', где VX - V-переменная, то перейти к LVD.

HARDR. (Жесткий элемент справа.)

Если A= A'(A''), то перейти к RB.

Если A= A Z, где Z - символ, то перейти к RSC.

Если A= A' S(Q)X, где SX - S-переменная, то перейти к RS.

Если A= A' W(Q)X, где WX - W-переменная, то перейти к RW.
Если A= A' E(Q)X, где EX - E-переменная, то перейти к RED.
Если A= A' V(Q)X, где VX - V-переменная, то перейти к RVD.

LB. (Скобка слева.) Если B не начинается с "(", то RETURN[UNDEF]. Иначе B= (B')B''. RETURN [RCG[G,X (A',B')(A'',B'') Y]].

LSC. (Символ-константа слева.) Если B не начинается с символа Z, то RETURN[UNDEF]. Иначе B= Z B'. RETURN [RCG[G,X (A',B') Y]].

LS. (S-переменная слева.) Если B не начинается с символа, то RETURN[UNDEF]. Иначе B= Z B'. Если Z IN YES[Q], то RETURN [RCG[G+<* (SX,Z) *>, X (A',B') Y]]. Иначе RETURN[UNDEF].

LW. (W-переменная слева.) Если B=<>, то RETURN[UNDEF]. Иначе B= T B', где T - терм. Если T IN YES[Q], то RETURN[RCG[G+<* (WX,T) *>, X (A',B') Y]]. Иначе RETURN[UNDEF].

LED. (Повторная E-переменная слева.) Пусть E=SUBST[G,EX]. Если B не начинается с E, то RETURN[UNDEF]. Иначе B= E B'. Если ZT[E] SUBSET YES[Q], то RETURN [RCG[G, X (A',B') Y]]. Иначе RETURN[UNDEF].

LVD. (Повторная V-переменная слева.) Пусть V=SUBST[G,VX]. Если B не начинается с V, то RETURN[UNDEF]. Иначе B= V B'. Если ZT[V] SUBSET YES[Q], то RETURN [RCG[G, X (A',B') Y]]. Иначе RETURN[UNDEF].

RB. (Скобка справа.) Если B не кончается на ")", то RETURN[UNDEF]. Иначе B= B'(B''). RETURN [RCG[G,X (A',B')(A'',B'') Y]].

RSC. (Символ-константа справа.) Если B не кончается на

символ Z, то RETURN[UNDEF]. Иначе B= B' Z. RETURN[RCG[G,X (A',B') Y]].

RS. (S-переменная справа.) Если B не кончается на символ, то RETURN[UNDEF]. Иначе B= B' Z. Если Z IN YES[Q], то RETURN[RCG[G+<* (SX,Z) *>, X (A',B') Y]]. Иначе RETURN[UNDEF].

RW. (W-переменная справа.) Если B=<>, то RETURN[UNDEF]. Иначе B= B' T, где T - терм. Если T IN YES[Q], то RETURN[RCG[G+<* (WX,T) *>, X (A',B') Y]]. Иначе RETURN[UNDEF].

RED. (Повторная E-переменная справа.) Пусть E=SUBST[G,EX]. Если B не кончается на E, то RETURN[UNDEF]. Иначе B= B' E. Если ZT[E] SUBSET YES[Q], то RETURN[RCG[G, X (A',B') Y]]. Иначе RETURN[UNDEF].

RVD. (Повторная V-переменная справа.) Пусть V=SUBST[G,VX]. Если B не кончается на V, то RETURN[UNDEF]. Иначе B= B' V. Если ZT[V] SUBSET YES[Q], то RETURN[RCG[G, X (A',B') Y]]. Иначе RETURN[UNDEF].

LVE. (Открытое вхождение VE-переменной.) В этом месте

L#E = (U(Q)X A,B) X

где UX - VE-переменная со спецификатором Q, не входящая в VARS[G].

Пусть B= T1 T2 ... TN, где T1, T2,...,TN - термы. Обозначим B[K] = T1 T2 ... TK, C[K] = T[K+1] ... TN, G[K] = RCG[G+<* (UX,B[K]) *>, (A,C[K]) X].

Если UX - E-переменная, то рассмотрим значения K=0,1,...,N, а если UX - V-переменная, то рассмотрим значения K=1,2,...,N, и найдем минамальное K, для которого C[K]¬=UNDEF, и при этом ZT[B[K]] SUBSET YES[Q]. Если такое K существует, то RETURN[G[K]], иначе RETURN[UNDEF].

2.5. СОКРАЩЕНИЕ ПЕРЕБОРА ПРИ ОТОЖДЕСТВЛЕНИИ

Описанный выше алгоритм отождествления действует достаточно эффективно в тех случаях, когда список дыр не содержит открытые вхождения VE-переменных. Однако, легко привести примеры, когда этот алгоритм делает лишнюю работу, пытаясь удлинить значение VE-переменной, хотя это заведомо не может привести к отождествлению.

Пусть, например, требуется вычислить

```
RCG[<Ø>, (E1 '+' E2 '*' E3, B)] ,
```

где B - некоторое об'ектное выражение.

Нетрудно показать, что если терпит неудачу удлинение значения переменной E2, то можно сразу же, не пытаясь удлинять значение переменной E1, сделать вывод, что отождествление невозможно. Отказ от удлинения E1 может значительно ускорить работу алгоритма отождествления. Пусть, например,

```
B = '+' '+' '+' ... '+'
```

где '+' повторяется N раз. Легко видеть, что неоптимизированный алгоритм проделает при отождествлении N удлинений для переменной E1 и

```
(N-1)+(N-2)+...+1
```

удлинений для переменной E2. Всего, таким образом, будет проделано N*(N+1)/2 длинений. Оптимизированный алгоритм проделает только N-1 удлинений для переменной E2, а до удлинения E1 дело вообще не дойдет. Таким образом, неоптимизированный алгоритм проделает примерно в N/2 раз больше удлинений, чем оптимизированный. При N=20 скорость работы двух алгоритмов будет различаться примерно в 10 раз.

В чем состоит сущность предлагаемой оптимизации можно более точно сформулировать следующим образом. Утверждается, что

```
RCG[<∅>, (E1 '+' E2 '*' E3, B)] =
    RCG[<* (E1,B1) *>, (E2 '*' E3, B2)]
```

где B1 и B2 - об'ектные выражения, которые вычисляются из соотношения

```
RCG[<∅>, (E1 '+' E2, B)] = <* (E1,B1), (E2,B2) *>
```

Таким образом, мы сначала вычисляем значение для переменной E1 (что может потребовать некоторого перебора), а затем, используя это значение, вычисляем значение переменной E2 (что тоже требует некоторого перебора). Однако, до оптимизации цикл перебора по E2 был вложен в цикл перебора по E1, а в результате оптимизации цикл перебора по E2 выносится из цикла перебора по E1.

В данном случае правильность оптимизации доказывается следующим рассуждением от противного. Пусть

```
RCG[<∅>, (E1 '+' E2 '*' E3, B)] =
    <* (E1,C1),(E2,C2),(E3,C3) *> = D'
```

и при этом

```
RCG[<∅>, (E1 '+' E2, B)] =
    <* (E1,B1),(E2,B2) *> = D''¬=
    <* (E1,C1),(E2,C2 '*' C3) *>
```

Из определения функции RCG следует, что

```
B = B1 '+' B2 = C1 '+' C2 '*' C3
```

и при этом, в силу того, что RCG находит самое левое отождествление, B1 короче, чем C1. Следовательно, существует такое выражение C, что C1 = B1 '+' C.

Теперь, исходя из подстановок D' и D'' сконструируем новую подстановку

```
D = <* (E1,B1),(E2,C '+' C2),(E3,C3) *>
```

Легко убедиться в том, что, во-первых, D является отождествлением и при этом более левым, чем D'. А это противоречит предположению о том, что D' – самое левое отождествление выражения B как E1 '+' E2 '*' E3. Полученное противоречие и доказывает наше утверждение.

Пусть теперь требуется вычислить

```
                I  2  3       4  5  6
RCG[<Ø>, (EA '+' EB, B1)(EX '+' EY, B2)].
```

Нетрудно доказать, что если удлинение для EX терпит неудачу, то отождествление невозможно. Другими словами

```
RCG[<Ø>, (EA '+' EB, B1)(EX '+' EY, B2)] = D = D'+D''
```

где D' и D'' вычисляются как

```
RCG[<Ø>, (EA '+' EB, B1)] = D'
RCG[<Ø>, (EX '+' EY, B2)] = D''.
```

Здесь рассуждением от противного также нетрудно показать, что D'+D'' является отождествлением, и при том – самым левым.

Теперь рассмотрим следующий пример. Пусть требуется вычислить

```
RCG[<Ø>, (EX E('A')Y E('B')Z, B)] = D
```

где B – некоторое об'ектное выражение.

Оказывается, что здесь вычисление подстановки D можно проделать следующим образом. Сначала нужно представить B в виде B= B' BZ, где все нуль-термы выражения BZ равны 'B', а B' кончается на терм, отличный от 'B', либо пусто. Затем B' следует представить в виде B'= BX BY, где все нуль-термы выражения BY равны 'A', а BX пусто или кончается на нуль-терм, отличный от 'A'. Затем в качестве D следует взять

```
D = <* (EX,BX),(EY,BY),(EZ,BZ) *>
```

Другими словами, нужно сначала "набрать E('B')Z справа по-максимуму", затем "набрать E('A')Y справа по-максимуму", а затем то, что останется, сделать значением переменной EX. Таким образом, перебор в процессе отождествления совсем не потребуется.

Докажем, что D - самое левое отождествление. То, что D - отождествление, очевидно. Теперь докажем, что если D - самое левое отождествление, то значением E('B')Z будет BZ определенное выше. Это доказывается рассуждением от противного.

Пусть D' - самое левое отождествление, и D'¬=D. Тогда

D' = <* (EX,B1),(EY,B2),(EZ,B3) *>

где B1, B2, B3 - некоторые об'ектные выражения. Ясно, что длина выражения B3 не может быть больше, чем длина выражения BZ, ибо в значение переменной E('B')Z не должны попадать нуль-термы, отличные от 'B'. Значит, B3 короче, чем BZ, и при этом

B1 B2 B3 = BX BY BZ.

Теперь рассмотрим два случая.

Если длина выражения B2 B3 не меньше, чем длина выражения BZ, то B2 можно представить в виде B2 = BA BB, где BB B3 = BZ. Рассмотрим подстановку

D'' = <* (EX,B1),(EY,BA),(EZ,BZ) *>.

Ясно, что D'' - отождествление, и при этом - более левое, чем D'.

Теперь рассмотрим случай, когда длина выражения B1 B2 B3 не меньше, чем длина выражения BZ, но длина выражения B2 B3 меньше, чем длина выражения BZ. Тогда B1 можно представить в виде B1= BA BB, где BB B2 B3 = BZ. Рассмотрим подстановку

D'' = <* (EX,BA),(EY,<>),(EZ,BZ) *>.

Ясно, что D'' - отождествление, и при этом – более левое, чем D'.

Итак, и в первом, и во втором случае было построено отождествление, более левое, чем D', что противоречит предположению о том, что D' - самое левое. Следовательно, значением переменной EZ должно быть выражение BZ.

Аналогичными рассуждениями доказывается, что значением переменной EY должно быть выражение BY.

Теперь рассмотрим следующий пример. Пусть требуется вычислить

```
RCG[<Ø>, (E('A')X 'B' EY, B)] = D,
```

где B - некоторое об'ектное выражение.

Оказывается, что здесь вычисление можно проделать следующим образом. Сначала нужно представить выражение B в виде B= BX B', где все нуль-термы выражения BX равны 'A', а B' пусто или начинается термом, отличным от 'A'. После этого D вычисляется из соотношения

```
D = RCG[<* (EX,BX) *>, ('B' EY,B')].
```

Другими словами, нужно "набрать E('A')X слева по-максимуму".

Действительно, значение переменной E('A')X не может быть короче, чем BX, ибо ни один нуль-терм из BX не может быть сопоставлен с 'B'. С другой стороны, значение переменной E('A')X не может быть длиннее, чем BX, ибо в этом случае в значение переменной E('A')X попал бы нуль-терм, отличный от 'A'.

Аналогичными рассуждениями доказывается, что при вычислении

```
RCG[<Ø>, (E('A')X E('B')Y '+' E('C')Z, B)]
```

значение переменной E('A')X следует "набрать слева по-максимуму".

Итак, мы рассмотрели несколько примеров, когда можно сократить перебор при отождествлении. Следующие разделы

посвящены формулировке достаточных условий, при которых выполнение таких оптимизаций является корректным.

2.6.ПЕРЕУПОРЯДОЧЕНИЕ ДНР

Т е о р е м а 6.1.

Пусть C - подстановка, а X и Y - кортежи пар выражений. Рассмотрим множество W[G,X], упорядоченное отношением <=[X] и выберем в нем самую левую подстановку DX из числа таких подстановок D, что W[D,Y]¬=<Ø>. Тогда

RCG[G,X Y] = RCG[DX,Y].

Д о к а з а т е л ь с т в о.

Пусть D IN W[G,X Y]. Представим D в виде D=D1+D2, где VARS[D1]= VARS[G]+VARS[X], VARS[D2]= VARS[G]+VARS[Y]. Легко видеть, что D1 IN W[G,X]. С другой стороны, D2 IN W[G,Y], откуда D2+D1 IN W[G+D1,Y], что равносильно D IN W[D1,Y].

Таким образом, множество W[G,X Y] можно представить в виде об'единения всех множеств вида W[D1,Y], где D1 IN W[G,X].

Пусть D=RCG[G,X Y]. Найдем такую подстановку D1, что D1 IN W[G,X] и D=RCG[D1,Y]. Докажем, что D1=DX. Действительно, если D1< [X] DX, то W[D1,Y]=<Ø>, что невозможно, ибо D IN W[D1,Y]. Предположим теперь, что DX< [X] D1. Рассмотрим D'=RCG[DX,Y]. Из того, что DX< [X] D1, следует, что D'< [X] D, ибо DX SUBSET D' и D1 SUBSET D. А из D'< [X] D следует, что D'< [X Y] D, что невозможно, в силу того, что D - самое левое отождествление в W[G,X Y]. Следовательно, D1=DX.

Теорема доказана.

Т е о р е м а 6.2.

Пусть G - подстановка, а X и Y - такие кортежи пар выражений, что

VARS[X] * VARS[Y] SUBSET VARS[G].

Тогда, если DX IN W[G,X] и DY IN W[G,Y], то DX+DY IN W[G,X Y].

Доказательство.

DX и DY - совместные подстановки. Поскольку G SUBSET DX и G SUBSET DY, G - тоже совместная. А поскольку все общие переменные из VARS[DX] и VARS[DY] входят в VARS[G], DX+DY тоже совместная. Отсюда очевидно утверждение теоремы.

Теорема 6.3.

Пусть G - подстановка, а X и Y - такие кортежи пар выражений, что

VARS[X] * VARS[Y] SUBSET VARS[G].

Тогда

RCG[G,X Y] = RCG[G,X] + RCG[G,Y].

Доказательство.

Сначала покажем, что области определения обеих частей равенства совпадают.

Очевидно, что если W[G,X]=<Ø> или W[G,Y]=<Ø>, то и W[G,X Y] = <Ø>. А по теореме 6.2 из W[G,X]¬=<Ø> и W[G,Y]¬=<Ø> следует, что W[G,X Y] ¬= <Ø>.

Таким образом, осталось рассмотреть случай, когда W[G,X]¬=<Ø>, W[G,Y]¬=<Ø> и W[G,X Y] ¬= <Ø>.

Обозначим DX=RCG[G,X], DY=RCG[G,Y]. Пусть D=DX+DY. По теореме 6.2, D IN W[G,X Y]. Докажем, что D = RCG[G,X Y].

Рассмотрим подстановку D' = RCG[G,X Y]. Поскольку D' - самая левая в W[G,X Y], заведомо выполнено D' < =[X Y] D. Представим D' в виде D'=DX'+DY', где DX' IN W[G,X] и DY' IN W[G,Y].

D' < =[X Y] D равносильно тому, что либо D'<[X Y] D, либо D' EQU[X Y] D.

Предположим, что D'<[X Y] D. Это равносильно тому, что либо D'<[X] D, либо D' EQU[X] D и при этом D'<[Y] D.

Предположим, что D'<[X] D. Тогда DX'<[X] DX, что невозможно в силу того, что DX=RCG[G,X]. Предположим, что D'

EQU[X] D и при этом D'< [Y] D. Тогда из D'< [Y] D следует DY'< [Y] DY, что невозможно в силу того, что DY=RCG[G,Y].

Таким образом, D'< [X Y] D невозможно. Поэтому D' EQU[X Y] D. Но тогда, по теореме 3.2, D'=D. Следовательно, D = RCG[G,X Y].

Теорема доказана.

Т е о р е м а 6.4.

Пусть G – подстановка, а X, Y и Z – такие кортежи пар выражений, что

VARS[X]*VARS[Y] SUBSET VARS[G]+VARS[Z].

Тогда

RCG[G,Z X Y] = RCG[G,Z Y X].

Д о к а з а т е л ь с т в о.

Очевидно, что W[G,Z X Y] = W[G,Z Y X], поэтому области определения обеих частей равенства в утверждении теоремы совпадают. Рассмотрим случай, когда W[G,Z X Y] = W[G,Z Y X] ¬= <Ø>.

По теореме 6.1 существуют такие подстановки DZ' и DZ'', что DZ' IN W[G,Z], DZ'' IN W[G,Z] и при этом

RCG[G,Z X Y] = RCG[DZ', X Y]
RCG[G,Z Y X] = RCG[DZ'',Y X].

Причем DZ' – самая левая из таких, что W[DZ',X Y]¬=<Ø>, а DZ'' – самая левая из таких, что W[DZ'',Y X]¬=<Ø>.

Очевидно, что W[G,X Y] = W[G,Y X], поэтому DZ'=DZ''. Обозначим их общее значение через DZ. Имеем по теореме 6.3

RCG[G,Z X Y] = RCG[DZ,X Y] =
RCG[DZ,X] + RCG[DZ,Y] =
RCG[DZ,Y] + RCG[DZ,X] =
RCG[DZ,Y X] = RCG[G,Z Y X].

Теорема доказана.

Т е о р е м а 6.5.
Пусть G - подстановка, а X1, X2,...,XN, Y1, Y2,...,YN - кортежи из пар выражений. Обозначим

```
X = X1 X2 ... XN
Y = Y1 Y2 ... YN
Z = X1 Y1 X2 Y2 ... XN YN
```

Тогда, если

```
VARS[X]*VARS[Y] SUBSET VARS[G]
```

то

```
RCG[G,Z] = RCG[G,X] + RCG[G,Y].
```

Д о к а з а т е л ь с т в о.
Проводится индукцией по N.
При N=1, по теореме 6.3, получаем

```
RCG[G,Z] = RCG[G,X1 Y1] =
RCG[G,X1] + RCG[G,Y1] = RCG[G,X] + RCG[G,Y].
```

При N>1 применяем два раза теорему 6.4, затем - индуктивное предположение, затем - теорему 6.3. Получаем

```
RCG[G,Z] =
RCG[G,X1 Y1 ... X[N-1] Y[N-1] XN YN] =
RCG[G,X1 Y1 ... X[N-1] Y[N-1] YN XN] =
RCG[G,X1 Y1 ... X[N-1] XN Y[N-1] YN] =
RCG[G,X Y] = RCG[G,X] + RCG[G,Y].
```

Теорема доказана.

2.7. ОТДЕЛЕНИЕ УДЛИНЯЮЩЕГОСЯ ВЫРАЖЕНИЯ

В дальнейшем нам понадобится рассматривать множества нуль-термов, входящих в об'ектные выражения, которые могут получаться из некоторого типового выражения в результате различных подстановок.

Расширим область определения функции ZT, введенной ранее в п.2.2, так, чтобы ее аргументом могли быть не только об'ектные, но и типовые выражения.

Если A - типовое выражение, то через ZT[A] будем обозначать об'единение всех множеств ZT[SUBST[D,A]], где D - некоторая подстановка, применимая к A. Очевидно, что при этом достаточно ограничиться рассмотрением только таких подстановок D, для которых VARS[D] = VARS[A].

Очевидно, что для любых типовых выражений A' и A'' справедливо

ZT[A' A''] SUBSET ZT[A'] + ZT[A''].

Равенство ZT[A' A''] = ZT[A']+ZT[A''] может и не иметь места. Например, пусть

A' = S('AB')X, A'' = S('BC')X

Тогда ZT[A' A''] = <* 'B' *>, ZT[A'] = <* 'A','B' *>, ZT[A''] = <* 'B','C' *>, ZT[A']+ZT[A''] = <* 'A','B','C' *>.

Т е о р е м а 7.1.
Пусть G - подстановка,

L#M = (A' A'', B) X

где A' и A'' - типовые выреия, B - об'ектное выражение, а X - кортеж из пар выражений. Пусть EZ - E-переменная, не входящая в множество VARS[G] + VARS[L] и при этом выполняются следующие условия.

(1) для любых двух подстановок D1 и D2, применимых к

A', и таких, что G SUBSET D1 и G SUBSET D2, из того, что

ZL[SUBST[D1,A']] < ZL[SUBST[D2,A']]

следует, что D1< [(A')] D2.

(2) VARS[A'] * [VARS[A'']+VARS[X]] SUBSET VARS[G].

(3) для любых об'ектных выражений C', C'' и C таких, что B= C' C C'', из того, что W[G,(A', C' C)] ¬= <Ø> и W[G,(A'',C'')] ¬= <Ø> следует, что W[G,(A'', C C'')]¬= <Ø>.

Обозначим D = RCG[G,(A' EZ, B)]. Тогда, если D=UNDEF, то и RCG[G,L#M]=UNDEF. Если же D¬=UNDEF, то RCG[G,L#M] можно вычислить следующим образом.

Представим D в виде

D = G + DØ +<* (EZ,B'') *>

где VARS[G], VARS[DØ] и <* EZ *> попарно не пересекаются. Тогда

RCG[G,L#M] = DØ + RCG[G, (A'', B'') X].

Д о к а з а т е л ь с т в о.

Обозначим

D' = RCG[G,L#M],

Если D'¬=UNDEF, то D' можно представить в виде

D' = G + DX + DZ,

где VARS[G], VARS[DX] и VARS[DZ] попарно не пересекаются и при этом

VARS[DX] = VARS[A'] - VARS[G],
VARS[DZ] = [VARS[A''] +VARS[X]] - VARS[G].

В случае, если D'¬=UNDEF, будем обозначать

SUBST[D', A'] = C',
SUBST[D', A''] = C''.

Докажем сначала, что если D=UNDEF, то и D'=UNDEF.

Действительно, предположим, что D'¬=UNDEF. Рассмотрим подстановку

D'' = G + DX + <* (EZ,C'') *>.

Нетрудно убедиться, что эта подстановка принадлежит множеству W[G,(A' EZ, C' C'')]. Следовательно, D¬=UNDEF. Поэтому не может одновременно выполняться D=UNDEF и D' =UNDEF.

Теперь рассмотрим случай, когда D¬=UNDEF. Обозначим

B' = SUBST[D, A'],
B'' = SUBST[D, A''].

Докажем, что B'=C'.

Сначала докажем, что B' не длиннее, чем C'. Рассмотрим подстановку

D'' = G + DX + <* (EZ,C'') *>.

Имеем

SUBST[D',A'] = SUBST[D'',A']

и при этом D и D'' принадлежат множеству W[G,(A' EZ, B)].

Предположим теперь, что B' длиннее, чем C'. Это означает, что SUBST[D,A'] длиннее, чем SUBST[D',A'] = SUBST[D'',A']. А это, согласно условию (I), означает, что D''< [(A')] D. Из этого следует, что D''< [(A' EZ)] D, что

невозможно в силу того, что D= RCG[G, (A' EZ, B)]. Из противоречия следует, что B' не длиннее, чем C'.

Теперь докажем, что B' не короче, чем C'.

Мы уже доказали, что B' не длиннее, чем C', поэтому существует такое об'ектное выражение C, что C' = B' C. Поскольку

$$D' = RCG[G, (A' A'', B' C C'') X],$$

имеем $W[G, (A', B' C)] \neg = \langle \emptyset \rangle$ и $W[G, (A'', C'') X] \neg = \langle \emptyset \rangle$. Поэтому, по условию (3), $W[G, (A'', C C'') X] \neg = \langle \emptyset \rangle$. Пусть DZ'= RCG[G, (A'', C C'') X] - G.

Рассмотрим подстановку

$$D''' = G + D\emptyset + DZ'.$$

Докажем, что эта подстановка принадлежит множеству W[G,L#M].

Действительно, множества VARS[G], VARS[DØ], VARS[A''] + VARS[X] попарно не пересекаются, стало быть, D''' - совместная подстановка. Далее,

$$SUBST[D''', A'] = B',$$
$$SUBST[D''', A''] = C C''$$

откуда

$$SUBST[D''', A' A''] = B' C C'' =$$
$$C' C'' = B = SUBST[D', A' A'']$$

Из этого очевидно, что D''' IN W[G,L#M].

Теперь докажем, что C=<>. Действительно, если $C \neg = \langle \rangle$, то B' короче, чем C'. Из этого следует, что SUBST[D''',A'] короче, чем SUBST[D',A']. Значит, согласно условию (1), D''' < [(A')] D', откуда D'''< [L#M] D', что невозможно, в силу того, что D'=RCG[G,L#M].

Итак, мы доказали, что B'=C', поэтому, по теореме 4.1,

RCG[G,L#M] = RCG[G,(A',B')(A'',B'') X]

Теперь, применяя теорему 6.3, получаем

RCG[G,L#M] = RCG[G,(A',B')] + RCG[G,(A'',B'') X].

Таким образом, осталось доказать, что

RCG[G, (A', B')] = G+DØ.

Заметим, что SUBST[G+DØ, A'] = B', поэтому G+DØ IN W[G, (A', B')].

Предположим, что существует такая подстановка DI, что

VARS[DI] = VARS[A'] - VARS[G],
G+DI IN W[G,(A', B')]

и при этом G+DI<[(A')] G+DØ. Тогда

G+DI+<* (EZ,B'') *> IN W[G, (A' EZ, B)].

При этом

G+DI+<* (EZ,B'') *><[(A' EZ)] G+DØ+<* (EZ,B'') *>,

что невозможно в силу того, что

G+DØ+<* (EZ,B'') *> = RCG[G, (A' EZ, B)].

Теорема доказана.

Т е о р е м а 7.2.
Пусть G - подстановка,

L#M = (A' A'', B) X

где A' и A'' - типовые выражения, B - об'ектное выражение, а X - кортеж из пар выражений. Пусть EZ - E-переменная, не

входящая в множество VARS[G] + VARS[L] и при этом выполняются следующие условия.

(1) A' имеет вид A'= U(Q)X AØ, где UX - VE-переменная, а AØ - типовое выражение, такое что каждая VE-переменная, входящая в AØ на нулевом уровне скобок, принадлежит множеству VARS[G] + <* UX *>.

(2) VARS[A'] * [VARS[A''] +VARS[X]] SUBSET VARS[G].

(3) A'' имеет вид A''= TI T2 ... TN U(R)Y AI, где AI - типовое выражение, UY - VE-переменная, а TI, T2,..., TN - типовые термы, которые не являются VE-переменными, такие, что

ZT[A'] SUBSET ZT[TI] SUBSET ZT[T2] SUBSET ...
ZT[TN] SUBSET YES[R],

и при этом попарно не пересекаются множества VARS[G] + VARS[A'] + VARS[AI] + VARS[X], VARS[TI], VARS[T2],..., VARS[TN], <* UY *>.

Обозначим D = RCG[G,(A' EZ, B) . Тогда, если D=UNDEF, то и RCG[G,L#M]=UNDEF. Если же D¬=UNDEF, то RCG[G,L#M] можно вычислить следующим образом.

Представим D в виде

D = G + DØ + <* (EZ,B'') *>

где VARS[G], VARS[DØ] и <* EZ *> попарно не пересекаются. Тогда

RCG[G,L#M] = DØ + RCG[G, (A'', B'') X].

Доказательство.

Достаточно показать, что из условий (1), (2), (3) следуют соответственно условия (1), (2), (3) теоремы 7.1.

(1) теоремы 7.1 следует из (1) в силу того, что ZL[SUBST[D,A']] есть линейная функция от ZL[SUBST[D,UX]] с положительным угловым коэффициентом.

(2) теоремы 7.1 совпадает с (2).

Таким образом, осталось показать, что (3) влечет (3) теоремы 7.1.

Пусть B= C' C C'' и при этом W[G, (A', C' C)] ¬= <∅> и W[G,(A'',C'') X]¬= <∅>. Тогда существуют такие подстановки D' и D'', что D' IN W[G, (A', C' C)] и D'' IN W[G, (A'', C'') X].

Обозначим CT= SUBST[D'', T1 T2 ... TN], CY= SUBST[D'',UY], CZ= SUBST[D'',A1]. Имеем C''= CT CY CZ. Рассмотрим теперь множество W[<∅>, (T1 T2 ... TN U(R)Y, C CT CY)] и докажем, что при ограничениях, наложенных на T1, T2,..., TN, U(R)Y, оно непусто.

Сначала заметим, что SUBST[D',A'] = C' C, поэтому ZT[C' C] SUBSET ZT[A'], откуда следует, что ZT[C] SUBSET ZT[A'] SUBSET ZT[T1] SUBSET ZT[T2] SUBSET ... SUBSET ZT[TN] SUBSET YES[R].

Поскольку все термы T1, T2,..., TN не являются VE-переменными, а SUBST[D'', T1 T2 ... TN] = CT, имеем ZL[CT] = N, т.е. CT составлено из N нуль-термов.

Обозначим через C1, C2,..., CN нуль-термы, составляющие выражение CT, а через B1, B2,..., BM - нуль-термы, составляющие выражение C. Таким образом, CT = C1 C2 ... CN, C = B1 B2 ... BM.

Очевидно, что SUBST[D'',T[I]] = C[I] при I=1, 2,..., N. Поэтому C[I] IN ZT[T[I]], а следовательно - ZT[C[I]] SUBSET ZT[T[I]] при I=1, 2,..., N.

Обозначим через H1, H2,..., H[M+N] нуль-термы, составляющие выражение C CT. Таким образом, H1=B1,..., H[M]=B[M], H[M+1]=C1,..., H[M+N]=C[N].

Поскольку множества VARS[G] + VARS[A'] + VARS[A1] + VARS[X], VARS[T1],..., VARS[TN], <* UY *> попарно не пересекаются, согласно теореме 6.2, для того, чтобы доказать,

что W[G, (A'', C C'') X] = <∅>, достаточно доказать, что

W[G, (AI, CZ) X]¬= <∅>,
W[G, (U(R)Y, H[N+I] ... H[N+M] CY)]¬= <∅>,
W[G, (T[İ], H[İ])]¬= <∅> при İ=I, 2,..., N.

Сначала заметим, что, по предположению, D'' IN W[G, (A'', C'') X] и при этом SUBST[D'',AI] = CZ. Поэтому, удалив из D'' лишние пары, мы можем построить подстановку из множества W[G, (AI, CZ) X], которое, следовательно, не пусто.

Теперь рассмотрим множество W[G, (U(R)Y, H[N+I] ... H[N+M] CY)]. Поскольку ZT[CY] SUBSET YES[R] и H[İ] SUBSET YES[R] при İ= N+I,..., N+M, а множество VARS[G] не содержит UY, подстановка

G + <* (UY, H[N+I] ... H[N+M] CY) *>

принадлежит множеству W[G, (U(R)Y, H[N+I] ... H[N+M] CY)], которое, следовательно, не пусто.

Осталось доказать, что W[G, (T[İ], H[İ])]¬= <∅> при İ=I, 2,..., N.

Ясно, что при любом İ=I, 2,...,N терм H[İ] либо является одним из термов BI, B2,..., BM, либо является одним из термов CI, C2,..., CN. В первом случае существует такое K, что H[İ]=B[K], а во втором случае существует такое J, что H[İ]=C[J] и при этом J<İ. Это можно наглядно изобразить следующей схемой:

```
        T T T T T
  B B B C C C C C
  T T T T T
```

Если H[İ]=B[K], то ZT[B[K]] SUBSET ZT[A'] SUBSET ZT[T[İ]], поэтому существует такая подстановка D[İ], что SUBST[D[İ], T[I]] = B[K]. Следовательно, W[<∅>, (T[İ], B[K])]¬= <∅>.

Если H[İ]=C[J], где J<İ, имеем ZT[C[J]] SUBSET ZT[T[J]] SUBSET ZT[T[İ]], поэтому существует такая подстановка D[İ],

что SUBST[D[I], T[I]] = C[J]. Следовательно, W[<∅>, (T[I], C[J])] ¬= <∅>.

Таким образом, W[<∅>, (T[I], H[I])] ¬= <∅>. Однако, G - совместная подстановка, а множества VARS[G] и VARS[T[I]] не пересекаются, поэтому и W[G, (T[I], H[I])] ¬= <∅>.

Теорема доказана.

2.8. ОТДЕЛЕНИЕ МАКСИМАЛЬНЫХ ВЫРАЖЕНИЙ

Пусть T - множество об'ектных термов, а B - об'ектное выражение. Рассмотрим множество выражений B', удовлетворяющих условиям

(1) существует выражение B'', такое, что B = B' B''.
(2) ZT[B'] SUBSET T.

Ясно, что это множество не пусто (ибо в него входит пустое выражение) и все входящие в него выражения имеют разную нуль-длину. Поэтому в этом множестве найдется единственное выражение, имеющее максимальную длину. Это выражение мы будем обозначать через ⌊MAX[T,B].

Аналогично, рассмотрим множество об'ектных выражений B'', удовлетворяющих условиям

(1) существует выражение B' такое такое, что B = B' B''.
(2) ZT[B''] SUBSET T.

Самое длинное выражение из этого множества обозначим через RMAX[T,B].

Т е о р е м а 8.1.
Пусть

⌊#M = X (A' A'', B) Y

и B = B' B'', где B' = ⌊MAX[ZT[A'],B]. Тогда значение выражения A' не может быть длиннее, чем выражение B', т.е. для

любого отождествления D из W[G,L#M] существует такое выражение C, что B' = C' C, где C' = SUBST[D,A'].

Доказательство.

Пусть для некоторого отождествления D, C' = SUBST[D,A'], C'' = SUBST[D,A'']. Тогда C' C'' = B = B' B'', причем ZT[C'] SUBSET ZT[A'], ZT[C''] SUBSET ZT[A''], ZT[B'] SUBSET ZT[A'].

Предположим, что C' не короче, чем B'. Тогда существует такое выражение C, что C' = B' C, B'' = C C''. Имеем ZT[B' C] = ZT[C'] SUBSET ZT[A']. Поэтому ZT[C] SUBSET ZT[A']. С другой стороны, ZT[B'] SUBSET ZT[A']. Отсюда ZT[C'] = ZT[B' C] SUBSET ZT[B']+ZT[C] SUBSET ZT[A']. Таким образом, ZT[B' C] SUBSET ZT[A']. Но, поскольку, B' = LMAX[ZT[A'],B], B' C не может быть длиннее, чем B'. Следовательно, C=<>, откуда C'=B'. Следовательно, C' не может быть длиннее, чем B'.

Теорема доказана.

Теорема 8.2.

Пусть

$$L\#M = X\ (A'\ A'',B)\ Y$$

и B = B' B'', где B'' = RMAX[ZT[A''],B]. Тогда значение выражения A'' не может быть длиннее, чем выражение B'', т.е. для любого отождествления D из W[G,L#M] существует выражение C такое, что B'' = C C'', где C'' = SUBST[D,A''].

Доказательство.

Доказательство "зеркально-симметрично" по отношению к доказательству теоремы 8.1.

Теорема 8.3.

Пусть

$$L\#M = X\ (A'\ A'',\ B)\ Y,$$
$$ZT[A'] * ZT[A''] = <\emptyset>$$

и B = B' B'', где B' = LMAX[ZT[A'],B]. Тогда

RCG[G,X (A' A'',B) Y] =
RCG[G,X (A',B')(A'',B'') Y].

Доказательство.

Согласно теореме 4.1, для того, чтобы доказать данную теорему, достаточно показать, что для любого отождествления D из W[G,L#M] выполнено SUBST[D,A'] = B' и SUBST[D,A''] = B''.

Пусть D IN W[G,L#M]. Обозначим C' = SUBST[D,A'], C'' = SUBST[D,A'']. Тогда C' C'' = B = B' B'', причем ZT[C'] SUBSET ZT[A'] и ZT[C''] SUBSET ZT[A''], ZT[B'] SUBSET ZT[A'].

Докажем, что C' = B' и C'' = B''.

Из теоремы 8.1 следует, что C' не может быть длиннее, чем B'. Поэтому существует такое C, что B' = C' C, C'' = C B''. Имеем ZT[C' C] = ZT[B'] SUBSET ZT[A']. Поэтому ZT[C] SUBSET ZT[A']. С другой стороны, ZT[C B''] = ZT[C''] SUBSET ZT[A'']. Поэтому ZT[C] SUBSET ZT[A'']. Следовательно, ZT[C] SUBSET ZT[A'] *ZT[A''] = <∅>. Итак, ZT[C] = <∅>, откуда следует, что C=<>. Поэтому B'=C', B''=C''.

Теорема доказана.

Теорема 8.4.

Пусть

L#M = X (A' A'',B) Y,
ZT[A'] *ZT[A''] = <∅>

и B = B' B'', где B'' = RMAX[ZT[A''],B]. Тогда

RCG[G,X (A' A'',B) Y] =
RCG[G,X (A',B')(A'',B'') Y].

Доказательство.

Доказательство теоремы "зеркально-симметрично" по отношению к доказательству теоремы 8.3.

Т е о р е м а 8.5.
Пусть

$$L\#M = X\ (A'\ A''\ A\emptyset,\ B)\ Y,$$
$$ZT[A'] * ZT[A''] = <\emptyset>$$

и B = B' B'', где B' = LMAX[ZT[A'],B]. Тогда, если для любой подстановки D из W[G,L#M] верно SUBST[D,A'']¬=<>, то

$$RCG[G,X\ (A'\ A''\ A\emptyset,\ B)\ Y] =$$
$$RCG[G,X\ (A',B')(A''\ A\emptyset,\ B'')\ Y].$$

Д о к а з а т е л ь с т в о.
Согласно теореме 4.1, для того, чтобы доказать данную теорему, достаточно показать, что для любой подстановки D из W[G,L#M] будет справедливо SUBST[D,A'] = B' и SUBST[D,A'' A∅] = B''.

Пусть D IN W[G,L#M]. Обозначим C' = SUBST[D,A'], C'' = SUBST[D,A''], C∅ = SUBST[D,A∅]. Тогда B' B'' = B = C' C'' C∅, причем C''¬=<>, ZT[C'] SUBSET ZT[A'], ZT[C''] SUBSET ZT[A''], ZT[B'] SUBSET ZT[A'].

Докажем, что C'=B' и C'' C∅ = B''.

Из теоремы 8.1 следует, что C' не может быть длиннее, чем B', поэтому существует такое C, что B' = C' C. При этом ZT[C' C] = ZT[B'] SUBSET ZT[A']. Поэтому, ZT[C] SUBSET ZT[A'].

Предположим теперь, что C' короче, чем B, т.е. что C¬=<>. Тогда выражение C начинается с некоторого терма T, T IN ZT[A']. С другой стороны, C'' = SUBST[D,A'] ¬=<>, и при этом C' C'' C∅ = B B'' = C' C B'', откуда C'' C∅ = C B''. Отсюда следует, что C'' тоже начинается с терма T. Однако, ZT[C''] SUBSET ZT[A''], поэтому T IN ZT[A'']. Следовательно, T IN ZT[A'] * ZT[A'']. Но, по условию теоремы ZT[A'] * ZT[A''] =<∅>. Таким образом, предположение о том, что C¬=<> ведет к противоречию. Следовательно B'=C'.

Теорема доказана.

Т е о р е м а 8.6.
Пусть

$$L\#M = X\ (A\emptyset\ A''\ A',\ B)\ Y,$$
$$ZT[A'] * ZT[A''] = \langle\emptyset\rangle$$

и B = B' B'', где B' = RMAX[ZT[A'],B]. Тогда, если для любой подстановки D из W[G,L#M] верно SUBST[D,A'']=<>, то

$$RCG[G,X\ (A\emptyset\ A''\ A',\ B)\ Y] = RCG[G,X\ (A\emptyset\ A'',B'')(A',\ B')\ Y].$$

Д о к а з а т е л ь с т в о.
Доказательство теоремы "зеркально-симметрично" по отношению к доказательству теоремы 8.5.

Т е о р е м а 8.7.
Пусть G - подстановка,

$$L\#M = X\ (E(Q1)1\ E(Q2)2\ \ldots\ E(QN)N\ U(R)A,\ B)\ Y,$$

где E(Q1)1, E(Q2)2,...,E(QN)N - вхождения E-переменных, а U(R)A - вхождение VE-переменной. Пусть каждая из переменных E1, E2,...,EN, UA входит в L только один-единственный раз и не входит в VARS[G]. Пусть B = B'' B', где B' = RMAX[VES[R],B]. Тогда

$$RCG[G,L\#M] = RCG[G,X\ (E(Q1)1\ E(Q2)2\ \ldots\ E(QN)N,\ B'')\ (U(R)A,\ B')\ Y].$$

Д о к а з а т е л ь с т в о.
Достаточно доказать, что если RCG[G,L#M] = D и B'' B' = B = C CA, где CA = SUBST[D,UA], то CA=B'.
По теореме 8.1, CA не может быть длиннее, чем B', поэтому остается доказать, что CA не может быть короче, чем B'.
Предположим, что CA короче, чем B'. Докажем, что при этом предположении можно построить подстановку D', которая,

во-первых, принадлежит W[G,L#M], и во-вторых, является более левой, чем D.

Представим D в виде D=DØ+D1, где

D1 = <* (E1,C1),(E2,C2),...,(EN,CN),(UA,CA) *>

и при этом VARS[DØ]*VARS[D1] = <Ø>.

Очевидно, что

B = B'' B' = C1 C2 ... CN CA

поэтому можно найти такое целое K, что

C[K] C[K+1] ... C[N] CA

не короче, чем B', но

C[K+1] ... C[N] CA

не длиннее, чем B'. Тогда найдутся такие выражения C' и C'', что C[K] = C' C'' и при этом

C'' C[K+1] ... C[N] CA = B'.

Построим подстановку

D2 = <* (E1,C1),(E2,C2),...,(E[K-1],C[K-1]),
(E[K],C'),(E[K+1],<>),...,(E[N],<>),
(UA, C'' C[K+1] ... C[N] CA) *>

Теперь рассмотрим подстановку D' = DØ+D2. Оказывается, что, во-первых, D' IN W[G,L#M], поскольку переменные E1, E2,..., EN, UA попарно различны, не входят в другие дыры и, таким образом, их новые значения не вступают в конфликт со значениями остальных переменных. Во-вторых, D'< [L#M] D, ибо значения всех VE-переменных, вхождения которых расположены левее, чем вхождение переменной E[K], не изменились, а значение переменной E[K] стало короче. Однако D'< [L#M] D

невозможно, ибо D = RCG[G,L#M]. Следовательно, CA не может быть короче, чем B' таким образом, остается только возможность CA=B'.

Теорема доказана.

2.9. ОЦЕНКА МНОЖЕСТВ НУЛЬ-ТЕРМОВ

В предшествующих разделах было доказано несколько теорем, в условиях которых используется функция ZT, применяемая к типовым выражениям. Если мы желаем применять эти теоремы для оптимизации рефал-программ, сразу же возникает проблема практического вычисления значений функции ZT. Более того, неясно даже, в каком виде следует изображать значения этой функции, ведь эти значения могут представлять собой бесконечные множества об'ектных термов.

Оказывается, что на практике вполне достаточно вместо точного вычисления значений функции ZT находить для этих значений только оценки сверху , и эти оценки - изображать в виде спецификаторов.

Определим с помощью рекурсивных соотношений функцию ZTS, которая каждому типовому выражению A ставит в соответствие спецификатор ZTS[A].

```
(1)  ZTS[<>] = (W)
(2)  ZTS[T A] = ZTS[T] + ZTS[A]
(3)  ZTS[Z] = Z (W)
(4)  ZTS[(A)] = B (W)
(5)  ZTS[S(Q)X] = [S (W)] * Q
(6)  ZTS[W(Q)X] = Q
(7)  ZTS[V(Q)X] = Q
(8)  ZTS[E(Q)X] = Q
```

где B и W - первичные спецификаторы, изображающие множество об'ектных термов вида (A) и всех об'ектных термов соответственно, A - произвольное типовое выражение, T - произвольный типовой терм, Z - произвольный символ, S(Q)X, W(Q)X, V(Q)X, E(Q)X - вхождения произвольной S-, W-, V- и E-переменной

соответственно со спецификатором Q.

Т е о р е м а 9.1.

Для любого типового выражения A, ZTS[A] – полный спецификатор.

Д о к а з а т е л ь с т в о.

Спецификаторы (W), Z (W) и B(W) – оканчиваются на (W), поэтому они – полные, согласно теореме 4.2 из главы 1. Если U(Q)X – вхождение переменной со спецификатором Q, то Q – полный, поскольку в типовых выражениях употребляются только полные спецификаторы. А по теоремам 6.4 и 7.3 из главы 1, операции + и *, примененные к полным спецификаторам, дают снова полный спецификатор. Поэтому, применив индукцию по длине аргумента функции ZTS, получаем утверждение теоремы.

Т е о р е м а 9.2.

Для любого типового выражения A справедливо

ZT[A] SUBSET YES[ZTS[A]].

Д о к а з а т е л ь с т в о.

Доказательство проводится индукцией по длине выражения A.

Рассмотрим определение функции ZTS. Если A имеет вид, указанный в пунктах (1), (3), (5), (6), (7), (8), то непосредственной проверкой убеждаемся, что ZT[A] = YES[ZTS[A]].

Если A имеет вид, указанный в пункте (4), т.е. A = (A'), то ZT[A] SUBSET YES[B (W)]. Поэтому ZT[(A')] SUBSET YES[B (W)] = YES[ZTS[(A')]].

Если A имеет вид, указанный в пункте (2), т.е. A = T A', то, по индуктивному предположению, ZT[T] SUBSET YES[ZTS[T]] и ZT[A'] SUBSET YES[ZTS[A']]. В то же время ZT[T A'] SUBSET ZT[T]+ZT[A'] (см. п.2.7). Таким образом,

ZT[T A'] SUBSET YES[ZTS[T]]+YES[ZTS[A']] =
YES[ZTS[T]+ZTS[A']] = YES[ZTS[T A']].

Теорема доказана.

Теперь, используя функцию ZTS и результаты главы I, мы можем предложить конструктивные способы проверки различных соотношений между значениями функции ZT.

В условиях теоремы 7.1 требуется проверять соотношение вида

```
ZT[A]  SUBSET  ZT[U(R)Y]
```

где A - типовое выражение, а U(R)Y - вхождение VE-переменной со спецификатором R. Заметим, что ZT[A] SUBSET YES[ZTS[A]], а ZT[U(R)Y] = YES[R]. Поэтому достаточным условием того, чтобы соотношение ZT[A] SUBSET ZT[U(R)Y] выполнялось, является справедливость соотношения

```
YES[ZTS[A]]  SUBSET  YES[R].
```

А это последнее соотношение, согласно теореме 9.1 из главы I, равносильно соотношению

```
YES[R-ZTS[A]]  = <∅>
```

которое можно проверить с помощью алгоритма, описанного в п.1.10.

В условиях теорем 8.1, 8.2, 8.3, 8.4, 8.5, 8.6 требуется вычислять выражения вида LMAX[ZT[A],B] и RMAX[ZT[A],B], где A - типовое выражение, а B - об'ектное выражение. На практике эти теоремы применяются, когда A - вхождение VE-переменной со спецификатором Q. Но в этом случае ZT[A]=YES[Q], поэтому фактически приходится вычислять только LMAX[YES[Q],B] и RMAX[YES[Q],B].

В условиях теорем 8.3, 8.4, 8.5 и 8.6 требуется проверять справедливость соотношений вида

```
ZT[A'] * ZT[A''] = <∅>
```

где A' и A'' - типовые выражения. Поскольку

```
ZT[A']  SUBSET  YES[ZTS[A']],
ZT[A'']  SUBSET  YES[ZTS[A'']],
```

достаточным условием справедливости проверяемого соотношения является справедливость соотношения

```
YES[ZTS[A']] * YES[ZTS[A'']]  =  <Ø>,
```

которое, согласно теореме 9.I из главы I, равносильно соотношению

```
YES[ZTS[A'] * ZTS[A'']]  =  <Ø>.
```

Это последнее соотношение можно проверить с помощью алгоритма, описанного в п.I.I0.

ГЛАВА 3. ЯЗЫК СБОРКИ

3.1. ЯЗЫК СБОРКИ КАК СИСТЕМА КОМАНД

В этом разделе описывается язык сборки для Рефала-2, который, фактически, представляет собой систему команд некоторой виртуальной машины.

Почему же мы называем этот язык "языком сборки", а не системой команд. Дело в том, что обычно различают систему команд некоторой машины и язык ассемблера для этой машины (ASSEMBLY LANGUAGE - язык сборки). Язык ассемблера отражает важнейшие особенности системы команд. Так, например, в языке ассемблера для каждой команды имеется ее мнемонический код, который определяет сколько операндов и каких типов требуется для этой команды. В то же время, язык ассемблера позволяет абстрагироваться от того, как представлены команды в памяти машины: какой двоичный код имеет каждая команда, в каких именно битах слова находится каждый операнд и т.д.

Система команд для рефала не представляет исключения. Мы называем ее языком сборки для рефала (REFAL ASSEMBLY LANGUAGE или, сокращенно, RAL) потому, что описываем ее в символическом виде, используя мнемонические названия операций, метки и т.д. При реализации языка сборки на различных машинах, его операторы можно представить в памяти машины по-разному. При этом должны учитываться специфические особенности каждой машины, как, например, разрядность слова и т.п.

Предполагается, что язык сборки должен реализовываться посредством интерпретации. Структура языка сборки хорошо приспособлена для этого. Каждый оператор начинается с кода операции, за которым следует фиксированное, определяемое кодом операции число операндов.

Интерпретатор языка сборки может реализовываться как программным, так и аппаратным путем.

Самый легкий путь - программная реализация. Он же - и единственный, если требуется реализовать интерпретатор языка сборки на ЭВМ с фиксированной архитектурой. При этом каждому

оператору языка сборки ставится в соответствие некоторая подпрограмма в интерпретаторе языка сборки. Каждая такая подпрограмма содержит, в среднем, 10-15 команд. Общий об'ем интерпретатора языка сборки составляет 1200-2000 команд.

Аппаратная реализация языка сборки - дело более сложное, и связанные с ним проблемы в данной работе не рассматриваются [ЭМСЭ 1974].

Наконец, следует отметить, что на крупных ЭВМ, имеющих большую виртуальную память, может оказаться целесообразным не интерпретировать операторы языка сборки, а компилировать их в команды машины. Скорость исполнения рефал-программы при этом может возрасти в 2-3 раза, но размер скомпилированной рефал-программы при этом может увеличиться в 5-10 раз.

В этой главе будет описан язык сборки, интерпретатор языка сборки, и на конкретных примерах будут показаны методы перевода программы с рефала на язык сборки, а также применяемые при этом оптимизации.

Семантика операторов языка сборки будет описана словесно, а также через соответствующие подпрограммы интерпретатора языка сборки, которые будут описаны формально, на некотором Фортрано-Алголо-ПЛ/1-подобном языке.

3.2. ОРГАНИЗАЦИЯ ПАМЯТИ

Во время работы рефал-программы обрабатываемая информация находится в поле зрения. Действия, совершаемые рефал-машиной над содержимым поля зрения, заключаются в удалении одних подвыражений и замене их другими. Поэтому, информацию в поле зрения нужно хранить таким образом, чтобы эти действия не приводили к преобразованию тех частей поля зрения, которые непосредственно не затрагиваются данной подстановкой. Очевидно, что для этого следует применить списковую организацию памяти.

Поскольку выражение в языке рефал - симметричный об'ект, в процессе работы с которым нужно уметь двигаться и слева направо, и справа налево, поле зрения следует представить в виде симметричного списка, каждый элемент которого содержит

ссылки на предшествующий и следующий элементы. Кроме того, для эффективного выполнения синтаксического отождествления, нужно уметь мгновенно находить для каждой скобки парную к ней скобку. Поэтому, элемент списка, изображающий скобку, должен содержать ссылку на соответствующую парную скобку.

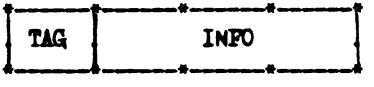
Исходя из этих соображений, будем считать, что об'ектные выражения представлены в виде списка, имеющего следующую структуру.

Основной единицей списка является звено. Память, отведенная под поле зрения, представляет собой совокупность некоторого числа звеньев. Звено - это группа машинных разрядов, допускающая прямую адресацию. В машинах с достаточно большой длиной слова основной памяти звеном может являться одно слово. Так обстоит дело, например, в случае машин БЭСМ-6, М-20 и др. В машинах серии ЕС ЭВМ в качестве звена может служить последовательность из трех слов (двенадцать байтов).

Каждое звено состоит из трех полей: CODE, PREV и NEXT.

Поля PREV и NEXT используются для связывания звеньев в линейную последовательность. Их размер должен быть таким, чтобы они могли содержать адрес любого звена, входящего в поле зрения. Поле NEXT всегда содержит адрес следующего звена, а поле PREV - адрес предыдущего звена.

Поле CODE состоит из двух подполей: TAG и INFO.



Содержимое поля CODE зависит от того, какому об'екту соответствует данное звено. Звено может изображать символ, либо структурную скобку (левую или правую), либо функциональную скобку (левую или правую).

Подполе TAG содержит признаки, позволяющие различать звенья разных типов. По нему всегда можно различить, что изображает звено: символ, левую скобку или правую скобку.

Рефал-машина оперирует с символами как с неделимыми об'ектами, природа которых для нее безразлична. Поэтому, единственно, что ей требуется - это уметь различать символ от скобки, а левую скобку от правой.

В настоящее время в Рефале-2 [Р2ОВЯ 1987] допускаются символы следующих типов: об'ектный знак (литера), метка, макроцифра, ссылка. При дальнейшем развитии рефала могут появиться символы других типов, однако, это никак не отражается на интерпретаторе языка сборки, ибо ему нет необходимости распознавать типы символов, а тождественность двух символов он распознает по полному совпадению содержимого полей CODE у соответствующих звеньев.

В зависимости от значения поля TAG, поле INFO содержит или код об'ектного знака (литеры), или адрес парной скобки, или адрес входа в функцию, или число, соответствующее макроцифре.

Таким образом, об'ектам каждого типа соответствует определенное числовое значение поля TAG. Какие именно числа сопоставить об'ектам различных типов - безразлично, поэтому в дальнейшем мы будем обозначать эти числа следующими идентификаторами:

TAGO - символ-литера (об'ектный знак);
TAGF - символ-метка (имя функции);
TAGN - символ-число (макроцифра);
TAGR - символ-ссылка (имя динамического ящика);
TAGLB - левая структурная скобка "(";
TAGRB - правая структурная скобка ")";
TAGK - левая функциональная скобка (знак "<" или "К");
TAGD - правая функциональная скобка (знак ">" или ".").

3.3.ФОРМАТЫ ОПЕРАТОРОВ ЯЗЫКА СБОРКИ

Программа на языке сборки представляет собой последовательность операторов. Каждый оператор имеет следующий вид:

МЕТКА: ИМЯ(АРГУМЕНТ1,АРГУМЕНТ2,...,АРГУМЕНТN);

то есть, оператор состоит из метки, мнемонического имени оператора и списка аргументов.

Метка оператора может отсутствовать. В этом случае опускается и следующее за ней двоеточие. Список аргументов может быть пустым. В этом случае обрамляющие его скобки опускаются.

Например:

```
TPL(6,2);
LW;
L1:SJUMP(L2);
```

Допускаются аргументы следующих типов:

N - целое число из интервала 0..255.
L - метка (адрес).
S - символ (в смысле языка рефал).

Имя оператора однозначно определяет сколько аргументов, каких типов и в какой последовательности требуются для данного оператора. Поэтому, все операторы языка сборки разбиваются на шесть групп, в соответствии с тем, какие аргументы требуются для этих операторов.

Г р у п п а 0. Для операторов этой группы аргументы не требуются. Например:

```
LS; EOS; PW;
```

Г р у п п а N. Требуется один аргумент типа N. Например:

```
LED(25); MULE(I5);
```

Г р у п п а NN. Требуются два аргумента типа N. Например:

```
SB(25,4);
```

Г р у п п а L. Требуется один аргумент типа L. Например:

```
SJUMP(LI); BLF(FUNC);
```

Г р у п п а S. Требуется один аргумент типа S. Например:

```
LSC('A'); NS(/PSI/); RSC(/I250/);
```

3.4. ОБЩАЯ СТРУКТУРА ИНТЕРПРЕТАТОРА ЯЗЫКА СБОРКИ

Чтобы программа на языке сборки могла выполняться с разумной эффективностью, перед исполнением ее необходимо перекодировать из "символической" формы в "машинное представление". (Этот процесс совершенно аналогичен переводу программы с языка ассемблера в команды машины.) При перекодировке мнемонические имена операторов заменяются на некоторые двоичные коды, метки заменяются на машинные адреса. Затем, программа, перекодированная в машинное представление , подается на вход интерпретатору языка сборки.

Интерпретатор языка сборки состоит из двух основных частей: дешифратора и набора подпрограмм, каждая из которых реализует действие отдельного оператора.

Дешифратор выполняет следующие действия. Для очередного оператора языка сборки дешифратор выбирает код этого оператора и определяет, к какой группе этот оператор принадлежит. Затем дешифратор выбирает из памяти аргументы оператора и загружает их в фиксированные переменные. Для операторов группы N - в переменную N, для операторов группы NN - в

переменные N и M, для группы L - в переменную L, для группы S - в переменную S. Затем дешифратор продвигает счетчик адреса на следующий оператор, по коду оператора определяет адрес входа в подпрограмму, реализующую данный оператор и передает управление по этому адресу. Затем работает подпрограмма, реализующая оператор, которая завершается передачей управления в дешифратор на метку NEXTOP.

Опишем дешифратор формально, используя следующие обозначения.

NMBL, LBLL, SMBL - длины аргументов типа N, L, S соответственно.

NMB, LBL, SMB - функции, которые выбирают по указанному адресу аргументы типа N, L, S соответственно. Предполагается, что код оператора занимает в памяти столько же места, сколько и аргумент типа N, поэтому для его выборки можно использовать функцию NMB.

VPC - виртуальный счетчик адреса. В момент обращения к дешифратору VPC должна содержать адрес очередного оператора языка сборки.

OPGROOP - переключатель, который по коду оператора вырабатывает одну из меток: GRP0, GRPN, GRPNN, GRPL, GRPS если оператор принадлежит к одной из групп 0, N, NN, L, S соответственно.

OPENTRY - переключатель, который по коду оператора вырабатывает адрес точки входа в подпрограмму, реализующую данный оператор.

Работа подпрограммы, реализующей оператор завершается передачей управления на метку NEXTOP.

При сделанных предположениях дешифратор описывается следующей программой:

```
*
* Выборка кода операции, операндов
* и ветвление на подпрограмму.
*
NEXTOP:  OPC=NMB(VPC);
         VPC=VPC+NMBL;
         GOTO OPGROOP(OPC);
```

```
GRPO:    GOTO OPENTRY(OPC);
GRPN:    N=NMB(VPC);
         VPC=VPC+NMBL;
         GOTO OPENTRY(OPC);
GRPNN:   N=NMB(VPC);
         VPC=VPC+NMBL;
         M=NMB(VPC);
         VPC=VPC+NMBL;
         GOTO OPENTRY(OPC);
GRPL:    L=LBL(VPC);
         VPC=VPC+LBLL:
         GOTO OPENTRY(OPC);
GRPS:    S=SMB(VPC);
         VPC=VPC+SMBL;
         GOTO OPENTRY(OPC);
```

Вход в дешифратор осуществляется передачей управления на метку NEXTOP.

Сколько памяти следует отводить под аргументы различных типов. Как мы увидим в дальнейшем, под аргумент типа N на практике будет достаточно отвести один байт, так как будут использоваться целые числа, не превосходящие 255. Длина аргумента типа L совпадает с длиной полей INFO, PREV и NEXT, а длина аргумента типа S совпадает с размером поля CODE.

3.5. ТАБЛИЦА ЭЛЕМЕНТОВ

Основная задача синтаксического отождествления - проанализировать выражение, стоящее в поле зрения внутри ведущего функционального терма и, в случае, если отождествление возможно, запомнить те адреса звеньев из поля зрения, которые могут потребоваться в дальнейшем, для замены левой части на правую.

Будем называть элементом предложения рефала любой входящий в него символ, скобку или переменную, а элементом поля зрения - любой символ или скобку, входящие в него.

В процессе работы интерпретатора языка сборки адреса

элементов поля зрения заносятся в одномерный массив, называемый таблицей элементов (ET). Чтобы не возникала терминологическая путаница, будем называть элементы массива ET строками таблицы элементов.

Строки таблицы элементов нумеруются начиная с единицы: ET[1], ET[2], ET[3],... В каждую строку помещается один адрес звена из 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 сводится к занесению некоторых адресов в ET, одному элементу левой части всегда будет соответствовать одна или две строки в ET.

Если элемент левой части - символ, скобка или S-переменная, в ET заносится адрес сопоставляемого ему звена из поля зрения.

Если элемент - W-переменная или VE-переменная, ему будет соответствовать две соседних строки в ET. В первую из них будет занесен адрес начального, а во вторую - конечного звена выражения из поля зрения, сопоставляемого данному элементу левой части.

Соответствие между строками ET и звеньями в поле зрения устанавливается динамически, в процессе синтаксического отождествления. Соответствие между элементами левой части и строками таблицы элементов устанавливается статически, во время компиляции программы с рефала на язык сборки.

3.6. ПЕРЕМЕННЫЕ NEL, B1 И B2

Номером элемента левой части рефал-предложения назовем номер соответствующей ему строки в ET. Если элементу соответствуют две соседних строки, номером элемента называется больший из номеров этих строк.

В процессе отождествления таблица элементов последовательно заполняется.

Во время работы интерпретатора языка сборки переменная NEL всегда содержит номер первой незаполненной строки в ET. Каждый оператор отождествления заполняет отведенное число строк в ET и увеличивает значение NEL на соответствующую величину, так, чтобы она снова указывала на первую свободную строку в ET.

Как правило, один оператор языка сборки осуществляет сопоставление одного элемента левой части предложения, но некоторые операторы сопоставляют сразу по несколько элементов.

В отличие от переменной NEL, которая в процессе отождествления движется по таблице элементов, переменные B1 и B2 движутся по полю зрения. В каждый момент отождествления, B1 и B2 содержат адреса каких-то двух звеньев из поля зрения.

Процесс отождествления организован так, что приступать к сопоставлению какого-либо элемента левой части можно лишь после того, как уже сопоставлен хотя бы один из двух его соседних элементов. При этом получается, что между уже сопоставленными элементами зияют еще не проанализированные дыры, состоящие из еще не сопоставленных элементов. Эти дыры постепенно стягиваются и исчезают. При сопоставлении пары скобок исходная дыра разбивается на две дыры меньшего размера.

В самом начале отождествления известно только три адреса: адреса ведущих знаков "<" и ">", а также адрес звена, содержащего имя функции и расположенного вслед за "<". Поэтому все пространство между именем функции и ">" представляет собой одну большую дыру.

В конце отождествления, когда сопоставлены уже все элементы левой части, дыр не остается вовсе.

Ясно, что в процессе отождествления каждой дыре соответствует некоторый участок поля зрения, который, в случае успешного завершения отождествления должен быть сопоставлен этой дыре. Этот участок будем называть образом дыры.

Будем называть текущим тот элемент левой части, сопоставление которого выполняется в данный момент, а текущей дырой - ту дыру, к которой принадлежит текущий элемент.

В процессе отождествления переменные B1 и B2 всегда установлены на левый и правый конец образа текущей дыры, а именно, B1 содержит адрес звена, предшествующего образу дыры, а B2 содержит адрес звена, следующего за образом дыры.

Сопоставление очередного элемента приводит к тому, что либо B1 сдвигается вправо, либо B2 сдвигается влево. Тем самым, образ дыры стягивается. Заметим, что при этом B1 всегда остается левее, чем B2, т.е. звено, на которое указывает B1, всегда предшествует звену, на которое указывает B2.

Перед началом отождествления в таблице элементов заполняются три строки. В ET[1] заносится адрес ведущей "<", в ET[2] - адрес ведущей ">", а в ET[3] - адрес звена, следующего за "<" и содержащего имя функции. Переменной NEL присваивается значение 4, а B1 и B2 устанавливаются на имя функции и на ">" соответственно. Все эти действия выполняет оператор EOS, который завершает очередной шаг и начинает следующий (его описание содержится в п.3.25).

3.7. ЯЗЫК ЗВЕНЬЕВ

Подпрограммы, соответствующие операторам языка сборки, мы будем описывать на так называемом языке звеньев. Он так назван потому, что позволяет описывать семантику операторов в терминах операций над звеньями.

Программа на языке звеньев представляет собой последовательность предложений. Каждое предложение записывается на отдельной строке и имеет вид:

метка: оператор;

Метка отделяется от оператора одним или несколькими пробелами. Она может быть опущена.

Существует три типа операторов языка звеньев:

1. Оператор присваивания:

левая часть = правая часть

2. Оператор перехода:

GOTO метка

3. Условный оператор:

IF (условие) оператор

Для работы с полями звена используется следующая конструкция:

имя поля (переменная)

где переменная является указателем на звено.

Таким образом, предложение

```
NEXT(B1) = INFO(B2);
```

означает, что в поле NEXT звена, адрес которого находится в переменной B1, засылается содержимое поля INFO звена, адрес которого находится в B2.

Далее, нам понадобится предикат BRA(X), который вырабатывает значение "истина", если переменная X содержит адрес звена, в котором находится скобка (левая или правая), и вырабатывает "ложь", если X указывает на звено, в котором находится символ.

Кроме того, в программах на языке звеньев будут употребляться следующие сокращения ("макрокоманды").

SHB1; - сдвиг B1 вправо. Сокращение для

```
BI=NEXT(BI);
IF(BI .EQ. B2) GOTO FAIL;
```

SHB2; - сдвиг B2 влево. Сокращение для

```
B2=PREV(B2);
IF(BI .EQ. B2) GOTO FAIL;
```

Используются и некоторые другие сокращения, которые будут вводиться по мере необходимости.

3.8. СОПОСТАВЛЕНИЕ ОБ'ЕКТНЫХ ВЫРАЖЕНИЙ

В этом разделе описаны операторы языка сборки, с помощью которых можно осуществить сопоставление любого выражения, не содержащего свободных переменных.

Оператор LSC(S); относится к группе S. Он сопоставляет символ S, с которого начинается некоторая дыра, с символом S, на левом конце образа дыры.

При обращении к LSC(S); границы BI и B2 должны быть установлены на края образа дыры. А именно, BI должен быть установлен на звено, предшествующее образу дыры, а B2 - на звено, следующее за образом дыры.

Оператор LSC(S); продвигает BI на одно звено вправо. Если в результате этого BI совпадет с B2, это означает, что образ дыры пуст. В этом случае сопоставление терпит неудачу, поскольку образ дыры не начинается с символа S.

Если же значение BI не совпадет со значением B2, LSC(S); проверяет, содержит ли звено, на которое теперь указывает BI, символ S. Если содержит - сопоставление успешно. Оператор заносит адрес этого звена в ET[NEL], а значение NEL увеличивает на единицу. Если же это звено не содержит символ S, то сопоставление неуспешно.

Всякий раз, когда некоторый оператор отождествления обнаруживает, что сопоставление потерпело неудачу, управление передается на программу, помеченную меткой FAIL (неудача). Что именно делает эта программа, будет описано позднее, в

п.3.10.

Ниже приведено описание оператора LSC(S); на языке звеньев.

```
LSC:    SHBI;
        IF(CODE(BI) .NE. S) GOTO FAIL;
        ET[NEL]=BI;
        NEL=NEL+1;
        GOTO NEXTOP;
```

Оператор NIL; относится к группе 0. Он не заполняет ET и не сдвигает NEL, BI и B2. Единственная его задача - проверить, что образ дыры пуст, т.е., что BI и B2 становлены на соседние звенья. Если между BI и B2 что-то есть, NIL; передает управление на метку FAIL. Таким образом, NIL; осуществляет сопоставление пустой дыры.

```
NIL:    IF(NEXT(BI) .NE. B2) GOTO FAIL;
        GOTO NEXTOP;
```

Имея только два оператора: LSC(S); и NIL; мы можем перевести на язык сборки любые левые части предложений, состоящие из одних символов. Например, рассмотрим предложение:

```
FUNC    'ЖАБА' =
```

Расставим над левой частью номера элементов. (При этом следует помнить, что имя функции всегда имеет номер 3, а - номер 2.) Получим:

```
  3      4   5   6   7  2
FUNC    'Ж' 'А' 'Б' 'А' =
```

Теперь ясно, что можно скомпилировать левую часть на язык сборки следующим образом:

```
LSC('Ж'); LSC('А'); LSC('Б'); LSC('А'); NIL;
```

Интересно отметить, что при работе этих операторов NEL,

B1 и B2 сами собой передвигаются надлежащим образом, причем обеспечивается правильное заполнение таблицы элементов.

Оператор RSC(S); относится к группе S. Он "зеркально симметричен" оператору LSC(S);.

```
RSC:    SHB2;
        IF(CODE(B2) .NE. S) GOTO FAIL;
        ET[NEL]=B2;
        NEL=NEL+1;
        GOTO NEXTOP;
```

Используя оператор RSC(S); можно по-другому скомпилировать последний пример

```
 3      5   7   6   4   2
FUNC   'Ж' 'А' 'Б' 'А'  =
```

в последовательность операторов:

```
RSC('А'); LSC('Ж'); RSC('Б'); LSC('А'); NIL;
```

При этом порядок отождествления уже другой, и соответствие между элементами левой части и строками таблицы элементов тоже другое.

Оператор LB; относится к группе 0. Он сопоставляет пару скобок, стоящую с левого конца дыры с парой скобок в образе дыры.

LB; сдвигает B1 на одно звено вправо, проверяет, не совпало ли B1 с B2, а затем проверяет наличие левой скобки в звене, на которое указывает B1. Если левая скобка есть, ее адрес заносится в ET[NEL]. Затем, по адресу, хранящемуся в поле INFO, LB; находит парную правую скобку и ее адрес заносит в ET[NEL+1]. После этого B2 станавливается на правую скобку, а NEL сдвигается на две единицы. Таким образом, LB; сопоставляет сразу два элемента левой части и заполняет две строки в ET.

```
LB:     SHB1;
        IF(.NOT. BRA(B1)) GOTO FAIL;
        B2=INFO(B1);
        ET[NEL]=B1;
        ET[NEL+1]=B2;
        NEL=NEL+2;
        GOTO NEXTOP;
```

Оператор RB; относится к группе 0. Он является "зеркальным отражением" оператора LB;.

```
RB:     SHB2;
        IF(.NOT. BRA(B2)) GOTO FAIL;
        B1=INFO(B2);
        ET[NEL]=B1;
        ET[NEL+1]=B2;
        NEL=NEL+2;
        GOTO NEXTOP;
```

Оператор SB(N,M); относится к группе NN. Он предназначен для перестановки границ B1 и B2 с образа одной дыры на образ другой. При этом в B1 заносится адрес из ET[N], а в B2 - адрес из ET[M].

```
SB:     B1=ET[N];
        B2=ET[M];
        GOTO NEXTOP;
```

Описанных операторов уже достаточно, чтобы скомпилировать любую левую часть, не содержащую переменных. Например:

```
  3      4  6  7 8 5  9   2
FUNC     ( 'X' ( ) ) 'Y'  =

    LB; LSC('X'); LB; NIL; SB(8,5); NIL;
    SB(5,2); LSC('Y'); NIL;
```

Если изменить очередность сопоставления элементов левой

части, получаем другой перевод на язык сборки:

```
  3      5 7  8 9 6  4    2
FUNC     ( 'X' ( ) ) 'Y'  =

     RSC('Y'); LB; LSC('X'); RB; NIL;
     SB(7,8); NIL; SB(6,4); NIL;
```

Из этих примеров видно, что довольно часто приходится производить установку границ оператором SB(N,M): из-за того, что операторы LB; и PB; ставят границы "не туда, куда надо". Чтобы уменьшить число таких случаев, в языке сборки предусмотрены два дополнительных оператора LBY; и RBY;. Они отличаются от LB; и RB; тем, что не входят в скобки , а "перепрыгивают через них".

```
LBY:     SHB1;
         IF(.NOT. BRA(B1)) GOTO FAIL;
         ET[NEL]=B1;
         B1=INFO(B1);
         ET[NEL+1]=B1;
         NEL=NEL+2;
         GOTO NEXTOP;

RBY:     SHB2;
         IF(.NOT. BRA(B2)) GOTO FAIL;
         ET[NEL+1]=B2;
         B2=INFO(B2);
         ET[NEL]=B2;
         NEL=NEL+2;
         GOTO NEXTOP;
```

Кроме того, довольно часто приходится сопоставлять пару скобок, внутри которой ничего нет. Для этих случаев предусмотрены два дополнительных оператора: LBNIL; и RBNIL;. LBNIL; заменяет последовательность операторов LB; NIL;, а RBNIL; - последовательность RB; NIL;. Кроме этого эти операторы перепрыгивают через скобки, что, как правило,

позволяет обойтись без явной установки границ на образ новой дыры. Таким образом, в этих случаях не приходится использовать оператор SB(N,M);.

```
LBNIL:   SHB1;
         IF(.NOT. BRA(B1)) GOTO FAIL;
         B0=B1;
         B1=INFO(B1);
         IF(NEXT(B0) .NE. B1) GOTO FAIL;
         ET[NEL]=B0;
         ET[NEL+1]=B1;
         NEL=NEL+2;
         GOTO NEXTOP;

RBNIL:   SHB2;
         IF(.NOT. BRA(B2)) GOTO FAIL;
         B0=B2;
         B2=INFO(B2);
         IF(NEXT(B2) .NE. B0) GOTO FAIL;
         ET[NEL]=B2;
         ET[NEL+1]=B0;
         NEL=NEL+2;
         GOTO NEXTOP;
```

Теперь левую часть предложения, рассмотренного выше, можно скомпилировать следующим образом:

```
  3      4  6  7 8 5  9   2
FUNC     ( 'X' ( ) ) 'Y'  =

     LB; LSC('X'); LBNIL; NIL;
     SB(5,2); LSC('Y'); NIL;
```

Довольно часто аргументами операторов LSC(S); и RSC(S); являются символы-литеры (об'ектные знаки). В этом случае можно уменьшить об'ем памяти, занимаемый программой на языке сборки, за счет того, что для хранения аргументов типа символ-литера достаточно расходовать только один байт, т.е.

столько же памяти, сколько для аргумента типа N.

В тех случаях, когда аргументами операторов LSC(S); и RSC(S); являются символы-литеры, эти операторы можно заменять на операторы LSC0(N); и RSC0(N); соответственно.

Операторы LSC0(N); и RSC0(N); относятся к группе N. Их аргументом должен быть код некоторого об'ектного знака. Они имеют следующее описание на языке звеньев.

```
LSC0:   SHB1;
        IF(TAG(B1) .NE. TAG0) GOTO FAIL;
        IF(INFO(B1) .NE. N) GOTO FAIL;
        ET[NEL]=B1;
        NEL=NEL+1;
        GOTO NEXTOP;

RSC0:   SHB2;
        IF(TAG(B2) .NE. TAG0) GOTO FAIL;
        IF(INFO(B2) .NE. N) GOTO FAIL;
        ET[NEL]=B2;
        NEL=NEL+1;
        GOTO NEXTOP;
```

В тех случаях, когда встречается подряд несколько операторов LSC0(N); или несколько операторов RSC0(N);, эти последовательности можно следующим образом заменить на операторы LTXT и RTXT соответственно. Пусть C1, C2,...,CN - символы-литеры, причем N не превышает 255. Тогда замена делается следующим образом:

```
LSC0(C1); ... LSC0(CN);   =   LTXT(N,C1,...,CN);
RSC0(C1); ... RSC0(CN);   =   RTXT(N,C1,...,CN);
```

Операторы LTXT(N,S1,S2,...,SN); и RTXT(N,S1,S2,...,SN); относятся к группе N, поскольку дешифратор операторов извлекает из памяти только их первый аргумент, а все прочие аргументы (каждый из которых занимает один байт) эти операторы извлекают самостоятельно.

```
LTXT:   SHB1;
        IF(TAG(B1) .NE. TAG0) GOTO FAIL;
        IF(INFO(B1) .NE. NMB(VPC)) GOTO FAIL;
        ET[NEL]=B1;
        NEL=NEL+1;
        VPC=VPC+NMBL;
        N=N-1;
        IF(N .NE. 0) GOTO LTXT;
        GOTO NEXTOP;

RTXT:   SHB2;
        IF(TAG(B2) .NE. TAG0) GOTO FAIL;
        IF(INFO(B2) .NE. NMB(VPC)) GOTO FAIL;
        ET[NEL]=B2;
        NEL=NEL+1;
        VPC=VPC+NMBL;
        N=N-1;
        IF(N .NE. 0) GOTO RTXT;
        GOTO NEXTOP;
```

3.9. СОПОСТАВЛЕНИЕ СВОБОДНЫХ ПЕРЕМЕННЫХ

При сопоставлении S-переменных не возникает никаких дополнительных проблем.

Оператор LS; относится к группе 0. Он сопоставляет главное вхождение S-переменной и работает аналогично оператору LSC(S);. Разница заключается только в том, что LS; проверяет наличие произвольного символа на левом конце образа дыры, в то время как LSC(S); проверяет наличие конкретного символа S.

```
LS:     SHB1;
        IF(BRA(B1)) GOTO FAIL;
        ET[NEL]=B1;
        NEL=NEL+1;
        GOTO NEXTOP;
```

Оператор RS: относится к группе 0. Он является зеркальным отражением оператора LS;.

```
RS:     SHB2;
        IF(BRA(B2)) GOTO FAIL;
        ET[NEL]=B2;
        NEL=NEL+1;
        GOTO NEXTOP;
```

Теперь можно скомпилировать такую левую часть:

```
  3      4  5   2
FUNC     SX SY  =

    LS; LS; NIL;
```

Если изменить порядок сопоставления, получаем другой перевод:

```
  3      5  4   2
FUNC     SX SY  =

    RS; LS; NIL;
```

Операторы LSD(N); и RSD(N); ("символ-дубликат") относятся к группе N. Они используются для сопоставления повторных вхождений S-переменных. Аргумент N - это номер главного вхождения S-переменной. Эти операторы работают аналогично LS; и RS;, но делают проверку на совпадение полей CODE у главного и повторного вхождений.

```
LSD:    SHB1;
        IF(CODE(B1) .NE. CODE(ET[N])) GOTO FAIL;
        ET[NEL]=B1;
        NEL=NEL+1;
        GOTO NEXTOP;
```

```
RSD:    SHB2;
        IF(CODE(B2) .NE. CODE(ET[N1])) GOTO FAIL;
        ET[NEL]=B2;
        NEL=NEL+1;
        GOTO NEXTOP;
```

Теперь можно скомпилировать такую левую часть:

```
  3     4  5   2
FUNC    SX SX  =

    LS; LSD(4); NIL;
```

Эту же левую часть можно скомпилировать и по-другому (сохраняя те же номера элементов, но не сохраняя вид движения B1 и B2):

```
    LS; RSD(4); NIL;
```

Теперь приступим к сопоставлению закрытых и повторных вхождений W-, E- и V-переменных.

Оператор CE: относится к группе 0. Он производит сопоставление закрытого вхождения E-переменной.

Для каждого вхождения E-переменной в ET отводится две строки, где запоминаются адреса начального и конечного звена того выражения, которое является значением вхождения переменной. Если значение выражения - пустое, то адреса запоминаются крест-накрест , т.е. в качестве адреса начала берется адрес звена, следующего за значением E-переменной, а в качестве адреса конца берется адрес звена, предшествующего значению E-переменной. Это странное соглашение удобно, так как в процессе отождествления не приходится по-особому рассматривать случаи, когда значение переменной - пустое.

Оператор CE; имеет следующее описание на языке звеньев:

```
CE:     ET[NEL]=NEXT(B1);
        ET[NEL+1]=PREV(B2);
        NEL=NEL+2;
        GOTO NEXTOP;
```

Теперь мы можем скомпилировать такую левую часть:

```
  3      4  6,7 5   2
FUNC     SX E1  SX  =

    LS; RSD(4); CE;
```

Здесь над E1 проставлены два номера: 6 и 7. Это вызвано тем, что E1 занимает две строки в ET. В ET[6] будет адрес начала значения E1, а в ET[7] - адрес конца.

Довольно часто встречаются главные вхождения E-переменных, заключенные в скобки, т.е. имеющие вид (EX). Для сопоставления таких вхождений предусмотрены операторы LBCE; и RBCE;, каждый из которых заменяет последовательность из двух операторов LB; CE; и FB; CE; соответственно. Операторы LBCE; и RBCE; перепрыгивают через пару скобок, что позволяет, как правило, не ставить после них оператор SB(N,M);.

LBCE; и RBCE; имеют следующее описание на языке звеньев:

```
LBCE:   SHB1;
        IF(.NOT. BRA(B1)) GOTO FAIL;
        B0=B1;
        B1=INFO(B1);
        ET[NEL]=B0;
        ET[NEL+1]=B1;
        ET[NEL+2]=NEXT(B0);
        ET[NEL+3]=PREV(B1);
        NEL=NEL+4;
        GOTO NEXTOP;

RBCE:   SHB2;
        IF(.NOT. BRA(B2)) GOTO FAIL;
        B0=B2;
```

```
        B2=INFO(B2);
        ET[NEL]=B2;
        ET[NEL+1]=B0;
        ET[NEL+2]=NEXT(B2);
        ET[NEL+3]=PREV(B0);
        NEL=NEL+4;
        GOTO NEXTOP;
```

Используя операторы LBCE; и RBCE; мы можем более кратко скомпилировать следующую левую часть:

```
  3      4 6,7 5   12,13   8 10,11 9   2
FUNC     ( EA )    E1      (  EB   )  =
```

```
    LBCE; RBCE; CE;
```

Оператор LED(N); ("выражение-дубликат") относится к группе N. Он сопоставляет повторное вхождение переменной выражения. Аргумент N является номером главного вхождения сопоставляемой переменной, аналогично оператору LSD(N);. Разница только в том, что выражение занимает две строки в ET, и в качестве аргумента используется номер второй из них.

Работает LED(N); следующим образом. Сначала из ET[N-1] и ET[N] он находит адреса начала и конца значения главного вхождения. Затем B1 начинает двигаться вправо. При этом каждое очередное звено сравнивается с соответствующим звеном главного вхождения. Если произойдет несовпадение хотя бы для одной пары звеньев, LED(N); терпит неудачу. Сопоставление считается успешно законченным, когда все звенья главного вхождения исчерпаны. В этот момент B1 как раз установлена на правый конец значения повторного вхождения.

```
LED:    ET[NEL]=NEXT(B1);
        B0=PREV(ET[N-1]);
LED1    IF(B0 .EQ. ET[N]) GOTO LED2;
        B0=NEXT(B0);
        SHB1;
        IF(CODE(B1) .EQ. CODE(B0)) GOTO LED1;
```

```
        IF(.NOT. BRA(B1)) GOTO FAIL;
        IF(TAG(B1) .EQ. TAG(B0)) GOTO LED1;
        GOTO FAIL;
LED2:   ET[NEL+1]=B1;
        NEL=NEL+2;
        GOTO NEXTOP;
```

Оператор RED(N); относится к группе N. Он является "зеркальным отражением" оператора LED(N);.

```
RED:    ET[NEL+1]=PREV(B2);
        B0=NEXT(ET[N]);
RED1:   IF(B0 .EQ. ET[N-1]) GOTO RED2;
        B0=PREV(B0);
        SHB2;
        IF(CODE(B2) .EQ. CODE(B0)) GOTO RED1;
        IF(.NOT. BRA(B2)) GOTO FAIL;
        IF(TAG(B2) .EQ. TAG(B0)) GOTO RED1;
        GOTO FAIL;
RED2:   ET[NEL]=B2;
        NEL=NEL+2;
        GOTO NEXTOP;
```

Теперь можно перевести на язык сборки такие левые части:

```
  3      4  6,7  5  8,9  2
FUNC     (  EA  )  EA   =

    LBCE; LED(7); NIL;

  3      4  6,7  5  10,11 8,9  2
FUNC     (  EA  )   E1   EA   =

    LBCE; RED(7); CE;
```

Операторы LW; и RW; относятся к группе 0. Они сопоставляют главное вхождение W-переменной. Их действие аналогично действию операторов LS; и RS;, однако, они заполняют

две строки в ET.

```
LW:     SHB1;
        ET[NEL]=B1;
        IF(BRA(B1)) B1=INFO(B1);
        ET[NEL+1]=B1;
        NEL=NEL+2;
        GOTO NEXTOP;

RW:     SHB2;
        ET[NEL+1]=B2;
        IF(BRA(B2)) B2=INFO(B2);
        ET[NEL]=B2;
        NEL=NEL+2;
        GOTO NEXTOP;
```

Теперь можно скомпилировать такую левую часть:

```
  3     4,5  8,9  6,7  2
FUNC    WX   EI   WY   =

    LW; RW; CE;
```

А как быть, если требуется сопоставить повторное вхождение терма. В этом случае можно использовать уже рассмотренные операторы LED(N); и RED(N);, поскольку информация, заносимая в ET для терма, ничем не отличается от информации, заносимой для выражения.

```
  3     4,5  8,9  6,7  2
FUNC    WX   EI   WX   =

    LW; RED(5); CE;
```

Теперь рассмотрим, как выполнять сопоставление для закрытых вхождений V-переменных.

Оператор NNIL; относится к группе 0. Он не передвигает B1 и B2 и ничего не заносит в ET. Единственная его задача - проверить, что ET[NEL-2] и ET[NEL-1] указывают на начало и

конец непустого выражения. Оператор NNIL; ставится после оператора CE;, чтобы проверить, что CE; занес в ET ссылки на непустое выражение.

```
NNIL:   IF(NEXT(ET[NEL-1]) .EQ. ET[NEL-2]) GOTO FAIL;
        GOTO NEXTOP;
```

Теперь мы можем скомпилировать такую левую часть:

```
  3      4   6,7  5   8,9  2
FUNC     (   VX   )   VY   =
```

```
LBCE; NNIL; CE; NNIL;
```

Пользуясь рассмотренными операторами, мы уже можем скомпилировать левую часть любой сложности, при условии, что она не содержит ни открытых вхождений VE-переменных, ни спецификаторов. Например:

```
  3      4 6   8,9  7   14,15  10  12,13  11  5
FUNC     ( (   E1   )   E2     (   E3     )  )  +

          16,17  18,19  20,21  25,26  23,24  22     2
          E1     WX     WX     E4     WX     SY     =
```

```
LB; LBCE; RBCE; CE;
SB(5,2); LED(9); LW; LED(19);
RS; RED(19); CE;
```

Сопоставление открытых вхождений VE-переменных представляет несколько более сложную задачу, так как тут уже не удается обойтись простым последовательным выполнением операторов языка сборки.

3.10. СОПОСТАВЛЕНИЕ ОТКРЫТЫХ ВХОЖДЕНИЙ VE-ПЕРЕМЕННЫХ

Чтобы обеспечить сопоставление открытых вхождений VE-переменных, интерпретатор языка сборки использует так называемый *стек переходов* (JS).

Стек переходов представляет собой одномерный массив, элементы которого нумеруются с нуля: JS[0], JS[1], JS[2],...

Имеется переменная - указатель стека переходов (JSP), которая в процессе интерпретации языка сборки указывает на первую свободную строку JS. Перед началом отождествления JSP присваивается значение 0. Каждая строка JS состоит из четырех полей: B1, B2, NEL и VPC. Поэтому нам удобнее изображать JS в виде четырех массивов: JSB1, JSB2, JSNEL и JSVPC. Каждое из полей B1, B2 и VPC должно вмещать произвольный аргумент типа L, а поле NEL - произвольный аргумент типа N.

Далее, в программах на языке звеньев нам понадобятся следующие сокращения.

PUTJS(L1,L2,N,L3); - запись в JS. Сокращение для

```
JSB1[JSP]=L1;
JSB2[JSP]=L2;
JSNEL[JSP]=N;
JSVPC[JSP]=L3;
```

GETJS(L1,L2,N,L3); - считывание из JS. Сокращение для

```
L1=JSB1[JSP];
L2=JSB2[JSP];
N=JSNEL[JSP];
L3=JSVPC[JSP];
```

Теперь мы можем описать на языке звеньев программу FAIL, на которую передают управление операторы отождествления, когда сопоставление терпит неудачу:

```
FAIL:   IF(JSP .EQ. 0) GOTO RCGIMP:
        JSP=JSP-1;
        GETJS(B1,B2,NEL,VPC);
        GOTO NEXTOP;
```

Отсюда видно, что последствия передачи управления на FAIL зависят от того, какая информация была ранее занесена в JS. В частности, в JS находится адрес того оператора языка сборки, который будет работать после FAIL. Если в момент обращения к FAIL стек переходов пуст, управление передается на программу RCGIMP, которая вызывает аварийный останов интерпретатора языка сборки: "отождествление невозможно".

Теперь, манипулируя стеком переходов, мы можем организовать проектирование открытых вхождений VE-переменных.

Открытое вхождение E-переменной сопоставляют операторы PLE; и LE;. Эти операторы относятся к группе 0. Оператор PLE; может употребляться только совместно с оператором LE; в следующей комбинации:

```
PLE; LE;
```

Действие этой пары операторов проще описать (и прочитать!) на языке звеньев, чем передать словами.

Оператор PLE; заносит в JS текущие значения B1, B2, NEL и VPC. Напоминаем, что в момент работы оператора PLE; виртуальный счетчик адреса VPC уже продвинут на следующий оператор, т.е. на оператор LE;. Поэтому в JS заносится адрес оператора LE;.

Затем PLE; заполняет ET[NEL] и ET[NEL+1] так, чтобы они соответствовали пустому выражению, продвигает VPC на оператор, следующий за LE; и возвращает управление в дешифратор.

Если один из последующих операторов передаст управление на FAIL, то FAIL восстановит B1, B2, NEL и VPC, в результате чего управление попадет в LE;.

При каждом входе в LE;, этот оператор так изменяет ET[NEL+1], чтобы к выражению, на которое указывают ET[NEL] и ET[NEL+1] добавился один терм справа. Если это возможно, LE; увеличивает глубину JS и возвращает управление в дешифратор,

если же невозможно - LE; терпит неудачу, т.е. передает управление на FAIL;.

Сопоставление открытого вхождения V-переменной осуществляется с помощью пары операторов

```
PLV; LE;
```

Оператор PLV; отличается от оператора PLE; только тем, что он не продвигает VPC на оператор, следующий за LE;. В результате этого, когда PLV; завершает работу, управление сразу же попадает на LE;. Таким образом, V-переменная сразу же получает непустое значение.

```
PLE:    PUTJS(B1,B2,NEL,VPC);
        JSP=JSP+1;
        ET[NEL]=NEXT(B1);
        ET[NEL+1]=B1;
        NEL=NEL+2;
        VPC=VPC+NMBL;
        GOTO NEXTOP;

PLV:    PUTJS(B1,B2,NEL,VPC);
        ET[NEL]=NEXT(B1);
        ET[NEL+1]=B1;
        GOTO NEXTOP;

LE:     B1=ET[NEL+1];
        SHB1;
        IF(BRA(B1)) B1=INFO(B1);
        JSP=JSP+1;
        ET[NEL+1]=B1;
        NEL=NEL+2;
        GOTO NEXTOP;
```

Теперь можно скомпилировать такие левые части:

```
  3      4,5  6    7,8  2
FUNC     E1   SX   E2   =

     PLE; LE; LS; CE;

  3      4,5  6    7,8  9   10,11  2
FUNC     E1   SX   V2   SX   E3    =

     PLE; LE; LS; PLV; LE; LSD(6); CE;
```

Операторы PLE;, PLV; и LE; используются, если производится отождествление слева направо. При отождествлении справа налево следует использовать их "зеркальные отражения" PRE;, PRV; и RE;.

```
PRE:    PUTJS(B1,B2,NEL,VPC);
        JSP=JSP+1;
        ET[NEL]=B2;
        ET[NEL+1]=PREV(B2);
        NEL=NEL+2;
        VPC=VPC+NMBL;
        GOTO NEXTOP;

PRV:    PUTJS(B1,B2,NEL,VPC);
        ET[NEL]=B2;
        ET[NEL+1]=PREV(B2);
        GOTO NEXTOP;

RE:     B2=ET[NEL];
        SHB2;
        IF(BRA(B2)) B2=INFO(B2);
        JSP=JSP+1;
        ET[NEL]=B2;
        NEL=NEL+2;
        GOTO NEXTOP;
```

3.II. ПЕРЕДАЧА УПРАВЛЕНИЯ С ОДНОГО ПРЕДЛОЖЕНИЯ НА ДРУГОЕ

Если в процессе сопоставления какой-то оператор языка сборки потерпел неудачу, управление передается либо на некоторый оператор LE; (а при отождествлении справа налево - на оператор RE;), либо на следующее рефал-предложение. Для управления переходами с одного предложения на другое используется оператор SJUMP(L): ("установка перехода").

Оператор SJUMP(L); относится к группе L. Он описывается на языке звеньев следующим образом:

```
SJUMP:   PUTJS(B1,B2,NEL,L);
         JSP=JSP+1;
         GOTO NEXTOP;
```

Простейший способ употребления оператора SJUMP(L); состоит в следующем. Допустим, что нам нужно перевести на язык сборки функцию FUNC, описание которой содержит N предложений. Тогда перевод функции FUNC на язык сборки может выглядеть следующим образом:

```
FUNC:    SJUMP(L1);   Перевод предложения 1
L1:      SJUMP(L2);   Перевод предложения 2
  . . . . . .
LN-2:    SJUMP(LN-1); Перевод предложения N-1
LN-1:                 Перевод предложения N
```

Оператор SJUMP(L); дает возможность оптимизировать программы на языке сборки за счет об'единения совпадающих частей различных предложений.

Поясним сказанное на конкретном примере. Рассмотрим функцию, состоящую из двух предложений.

```
  3      4,5  8,9   7    6    2
FUNC     WX   E1   'A'   SZ   =

  3      4,5  8,9   7    6    2
         WA   E2   'B'   SX   =
```

```
FUNC: SJUMP(LI);
      LW;              LI: LW;
      RS;                  RS;
      RSC('A');            RSC('B');
      CE;                  CE;
      ...                  ...
```

Видно, что перевод обоих предложений начинается операторами LW; RS;. Следовательно, когда управление попадает на второе предложение, операторы LW; RS; будут выполняться зря, ибо они будут повторять ту же работу, которую уже сделали операторы LW; RS; в первом предложении. Оператор SJUMP(L); позволяет провести оптимизацию программы на языке сборки.

```
FUNC: LW;
      RS;
      SJUMP(LI);
      RSC('A');        LI: RSC('B');
      CE;                  CE;
      ...                  ...
```

Оптимизированная программа меньше по размерам и работает быстрее. Если первое предложение терпит неудачу, управление передается не на начало второго предложения, а прямо в то место, с которого начинаются расхождения между предложениями.

В общем случае оптимизация производится следующим образом.

Пусть нам дана функция, состоящая из N предложений. Переведем каждое предложение на язык сборки независимо. Обозначим через A[I,J] J-й оператор языка сборки в переводе I-го предложения. Затем построим упорядоченное дерево с выделенным корнем 0. Каждая дуга дерева помечена некоторым оператором A[I,J].

Это дерево показывает, как должно передаваться управление при работе операторов отождествления.

Если оператор успешно произвел сопоставление, то нужно по

направлению соответствующей дуги спуститься до ближайшего узла, а затем перейти на самую левую дугу, выходящую из этого узла.

Если оператор потерпел неудачу, то нужно подняться до ближайшего узла против направления соответствующей дуги, а затем перейти на дугу, выходящую из этого узла и следующую за дугой, по которой мы только что пришли. Если такой дуги нет, то следует опять подняться до ближайшего сверху узла, и процесс повторяется.

Теперь мы следующим образом будем преобразовывать дерево.

Пусть из некоторого узла Q выходят две дуги, помеченные одним и тем же оператором R, причем между ними нет других дуг. Пусть эти дуги направлены в узлы Q1 и Q2, к которым подвешены поддеревья X1 и X2 соответственно.

Тогда, если оператор R не является одним из операторов PLE;, PLV;, LE;, PRE;, PRV;, RE;, мы можем слить два узла Q1 и Q2 в один узел Q', к которому подвешены поддеревья X1 и X2 без нарушения взаимного порядка, т.е. сначала идут дуги, принадлежащие X1, а потом - дуги, принадлежащие X2.

Будем применять описанное преобразование до тех пор, пока это возможно. Полученное дерево теперь нужно превратить в программу на языке сборки, надлежащим образом расставив метки и операторы SJUMP(L);.

Пусть из некоторого узла Q выходит N дуг, которые направлены в узлы Q1, Q2,...QN. Каждая дуга (Q,Qi) помечена оператором Ri, а к каждому из узлов Qi подвешено поддерево Xi.

Тогда перевод поддерева с корнем в узле Q выглядит следующим образом:

```
          SJUMP(L1);   R1;     Перевод поддерева X1
L1:       SJUMP(L2);   R2;     Перевод поддерева X2
  . . . . . . . . . . . . . . . . . . . . . . .
LN-2:     SJUMP(LN-1); RN-1;   Перевод поддерева XN-1
LN-1:                  RN;     Перевод поддерева XN
```

В частности, при N=1, эта конструкция вырождается в

```
CI;  Перевод поддерева XI
```

При описании оптимизации мы сделали важную оговорку, касающуюся операторов сопоставления открытых вхождений VE-переменных. Их об'единять нельзя, поскольку на них могут передать управление операторы, идущие вслед за ними. К этому вопросу мы еще вернемся в п.3.12.

3.12. ОПЕРАТОР EOE(N);

Оператор EOE(N); относится к группе N. Он имеет следующее описание на языке звеньев:

```
EOE:    JSP=JSP-N;
        GOTO NEXTOP;
```

В тех случаях, когда N=1, вместо оператора EOE(N); можно употреблять оператор EOE1;, который относится к группе 0 и имеет следующее описание на языке звеньев:

```
EOE1:   JSP=JSP-1;
        GOTO NEXTOP;
```

Чтобы понять назначение операторов EOE(N); и EOE1;, рассмотрим следующий пример:

```
  3      4,5   6    7,8   9     10,11 12
FUNC     E1   '+'   E2   '*'    E3    =
```

```
PLE; LE; LSC('+'); PLE; LE; LSC('*'); CE;
```

Пусть ведущий функциональный терм имеет вид:

```
<FUNC '+++ ... +'>
```

т.е. аргумент функции содержит знак '+', повторенный N раз.

Как будет происходить сопоставление? Сначала E1 примет

значение "пусто", а символ '+' в левой части сопоставится с первым символом '+' из аргумента функции. Затем значение переменной E2 будет удлиняться N-1 раз в поисках символа '*'. Когда дальнейшее удлинение станет невозможно, удлинится значение E1, после чего значение E2 будет удлиняться N-2 раз и т.д. Когда удлинение значения E1 станет невозможно, все предложение потерпит неудачу.

Очевидно, что второй оператор LE; проработает

$$(N-1)+(N-2)+\ldots+1 = N*(N-1)/2$$

раз, а первый LE; будет работать N раз. Поэтому общий об'ем работы можно оценить как (N*N)/2.

Между тем, совершенно очевидно, что если выражение не содержит символ '*', то и любая его часть не содержит '*'. Следовательно, удлинять E1 нет смысла. Поэтому, если второй LE; терпит неудачу, то можно сразу же об'явить, что все предложение терпит неудачу.

Оператор EOE(N); позволяет выразить это на языке сборки следующим образом:

```
PLE; LE; LSC('+'); EOE1;
PLE; LE; LSC('*'); CE;
```

Теперь первый LE; не будет исполняться ни разу (поскольку PLE; обойдет его), а второй LE; проработает только N-1 раз. Таким образом, об'ем работы можно оценить как N. Оптимизированная левая часть работает примерно в N/2 раза быстрее. При N=20 скорость ее работы возрастает на порядок.

Усложним пример. Пусть теперь функция FUNC содержит два предложения.

```
FUNC    E1   '+'   E2   '*'   E3    =
        E1   '+'   E2   'A'   E3    =
```

К каждому из этих предложений применимы рассуждения из предыдущего примера. Однако, теперь возможна дополнительная оптимизация, за счет об'единения операторов из различных

левых частей. Легко видеть, что перевод и первого, и второго предложения начинается с операторов

```
PLE; LE; LSC('+'); EOEI;
```

Эта группа операторов либо находит '+', либо вся в целом терпит неудачу. Если '+' найден, то последующие операторы, в случае неудачи, не возвращаются на LE;, а сразу переходят на следующее предложение: поэтому вся группа операторов

```
PLE; LE; LSC(+'); EOEI;
```

в целом обладает такими же свойствами, как "обычные" операторы языка сборки вроде LS; LW; и т.д. Ее можно рассматривать как "составной оператор" языка сборки.

Если рассматривать группы операторов PLE; LE;, PLV; LE;, PRE; RE;, PRV; RE; как "левые скобки", а оператор EOEI; как "правую скобку", то последовательность операторов отождествления можно рассматривать как "выражение", которое представляет собой последовательность "термов", где каждый "терм" - это либо оператор, отличный от "скобок", либо снова "выражение" заключенное в "скобки".

Теперь видно, как усовершенствовать алгоритм оптимизации, описанный в п.3.11. Раньше этот алгоритм не имел права об'единять "левые скобки", поскольку каждая дуга дерева была помечена одним оператором. Однако, можно слегка изменить дерево, чтобы каждая дуга была помечена "термом" в описанном выше смысле. Тогда оказывается, что можно об'единять дуги с одинаковыми пометками без ограничений.

Рассматриваемый пример следующим образом переведется на язык сборки:

```
FUNC: PLE; LE; LSC('+'); EOEI; SJUMP(L1);
      PLE; LE; LSC('*'); CE;
L1:   PLE; LE; LSC('A'); CE;
```

Некоторые комбинации, состоящие из операторов LE; (или RE;) с другими операторами, встречаются довольно часто,

поэтому предусмотрено несколько дополнительных операторов, каждый из которых по своему действию эквивалентен последовательности из нескольких операторов языка сборки. Ниже перечислены комбинации операторов и указаны соответствующие им операторы-сокращения.

```
PLE; LE; LSC(S);   ⇒  PLESC; LESC(S);
PLV; LE; LSC(S);   ⇒  PLVSC; LESC(S);
PRE; RE; RSC(S);   ⇒  PRESC; RESC(S);
PRV; RE; RSC(S);   ⇒  PRVSC; RESC(S);
PLE; LE; LSD(N);   ⇒  PLESC; LESD(N);
PLV; LE; LSD(N);   ⇒  PLVSC; LESD(N);
PRE; RE; RSD(N);   ⇒  PRESC; RESD(N);
PRV; RE; RSD(N);   ⇒  PRVSC; RESD(N);
PLE; LE; LB;       ⇒  PLEB; LEB;
PLV; LE; LB;       ⇒  PLVB; LEB;
PRE; RE; RB;       ⇒  PREB; REB;
PRV; RE; RB;       ⇒  PRVB; REB;
PLE; LE; LSC(S); EOEI;   =   LSRCH(S);
PRE; RE; RSC(S); EOEI:   =   RSRCH(S);
```

Дополнительные операторы языка сборки имеют следующее описание на языке звеньев:

```
PLESC:   PUTJS(BI,B2,NEL,VPC);
         ET[NEL]=NEXT(BI);
         ET[NEL+2]=BI;
         GOTO NEXTOP;

PLVSC:   PUTJS(BI,B2,NEL,VPC);
         ET[NEL]=NEXT(BI);
         SHBI;
         IF(BRA(BI)) BI=INFO(BI);
         ET[NEL+2]=BI;
         GOTO NEXTOP;
```

```
LESC:   B1=ET[NEL+2];
LESC1:  SHB1;
        IF(.NOT. BRA(B1)) GOTO LESC2;
        B1=INFO(B1);
        GOTO LESC1;
LESC2:  IF(CODE(B1) .NE.S) GOTO LESC1;
        JSP=JSP+1;
        ET[NEL+1]=PREV(B1);
        ET[NEL+2]=B1;
        NEL=NEL+3;
        GOTO NEXTOP;

PRESC:  PUTJS(B1,B2,NEL,VPC);
        ET[NEL+1]=PPEV(B2);
        ET[NEL+2]=B2;
        GOTO NEXTOP;

PRVSC:  PUTJS(B1,B2,NEL,VPC);
        ET[NEL+1]=PREV(B2);
        SHB2;
        IF(BRA(B2)) B2=INFO(B2);
        ET[NEL+2]=B2;
        GOTO NEXTOP;

RESC:   B2=ET[NEL+2];
RESC1:  SHB2;
        IF(.NOT. BRA(B2)) GOTO RESC2;
        B2=INFO(B2);
        GOTO RESC1;
RESC2:  IF(CODE(B2)) .NE.S) GOTO RESC1;
        JSP=JSP+1;
        ET[NEL]=NEXT(B2);
        ET[NEL+2]=B2;
        NEL=NEL+3;
        GOTO NEXTOP;

LESD:   S=CODE(ET[N]);
        GOTO LESC;
```

```
RESD:   S=CODE(ET[N]);
        GOTO RESC;

PLEB:   PUTJS(BI,B2,NEL,VPC);
        ET[NEL]=NEXT(BI);
        ET[NEL+3]=BI;
        GOTO NEXTOP;

PLVB:   PUTJS(BI,B2,NEL,VPC);
        ET[NEL]=NEXT(BI);
        SHBI;
        IF(BRA(BI)) BI=INFO(BI);
        ET[NEL+3]=BI;
        GOTO NEXTOP;

LEB:    BI=ET[NEL+3];
LEBI:   SHBI;
        IF(.NOT. BRA(BI)) GOTO LEBI;
        JSP=JSP+1;
        ET[NEL+1]=PREV(BI);
        ET[NEL+2]=BI;
        B2=INFO(BI);
        ET[NEL+3]=B2;
        NEL=NEL+4;
        GOTO NEXTOP;

PREB:   PUTJS(BI,B2,NEL,VPC);
        ET[NEL+1]=PREV(B2);
        ET[NEL+2]=B2;
        GOTO NEXTOP;

PRVB:   PUTJS(BI,B2,NEL,VPC);
        ET[NEL+1]=PREV(B2);
        SHB2;
        IF(BRA(B2)) B2=INFO(B2);
        ET[NEL+2]=B2;
        GOTO NEXTOP;
```

```
REB:    B2=ET[NEL+2];
REB1:   SHB2;
        IF(.NOT. BRA(B2)) GOTO REB1;
        JSP=JSP+1;
        ET[NEL]=NEXT(B2);
        ET[NEL+3]=B2;
        B1=INFO(B2);
        ET[NEL+2]=B1;
        NEL=NEL+4;
        GOTO NEXTOP;
```

3.13. РЕАЛИЗАЦИЯ СПЕЦИФИКАТОРОВ

Спецификаторы, в сущности, представляют собой некоторый самостоятельный язык, который никак не связан с языком, на котором пишутся рефал-предложения. С точки зрения рефал-предложения спецификатор - это некоторый предикат, который можно применить к любому звену из аргумента функции, и который в результате этого применения говорит "да" или "нет". При этом внутреннее устройство спецификатора совершенно безразлично для того, кто его вызывает.

В свете вышесказанного нет ничего удивительного в том, что спецификаторы компилируются в программы на некотором особом языке, который не имеет ничего общего с языком сборки рефала. Этот язык мы будем называть языком сборки спецификаторов.

Каждый спецификатор компилируется в отдельную программу на языке сборки спецификаторов. Эта программа помечается меткой, которая затем может использоваться в качестве аргумента в некоторых операторах языка сборки рефала. После перевода языка сборки в машинное представление эта метка, разумеется, превратится в машинный адрес - тот адрес, с которого начинается скомпилированный спецификатор.

Спецификаторы могут исполняться с помощью особого интерпретатора, который с точки зрения интерпретатора языка сборки рефала является функцией, обращение к которой имеет следующий вид:

```
SPC(L,B)
```

где L – метка, которой помечен некоторый спецификатор, а B – переменная, которая содержит адрес некоторого звена. Функция SPC(L,B) интерпретирует спецификатор и выдает значение "да", если звено в удовлетворяет спецификатору L, и значение "нет" в противном случае.

Компиляция спецификаторов сводится к простой перекодировке. При этом каждый элемент спецификатора заменяется на оператор следующим образом:

```
W   ⇒   SPCW;
S   ⇒   SPCS;
B   ⇒   SPCB;
F   ⇒   SPCF;
N   ⇒   SPCN;
R   ⇒   SPCR;
O   ⇒   SPCO;
D   ⇒   SPCD;
L   ⇒   SPCL;
```

Круглые скобки "(" и ")" заменяются на оператор SPCNG;. Каждый символ S (имеется в виду символ в смысле языка рефал) заменяется на оператор SPCSC(S);. Каждое имя спецификатора имеющее вид :L: заменяется на оператор вызова спецификатора SPCCLL(L);. В конце скомпилированного спецификатора ставится оператор SPCNGW;, который эквивалентен последовательности из двух операторов: SPCNG; SPCW;. Благодаря этому спецификатор выдает либо "да", либо "нет" для любого звена.

Например, спецификатор

```
/ALPHA/ (L) :PSI: S
```

скомпилируется в

```
SPCSC(/ALPHA/); SPCNG; SPCL; SPCNG;
SPCCLL(PSI); SPCS; SPCNGW;
```

В интерпретаторе спецификаторов используются следующие переменные.

SPCVPC - виртуальный счетчик адреса, который содержит адрес очередного оператора.
SPCS - стек спецификаторов, в котором запоминается информация при вызове одного спецификатора из другого.
SPCSP - указатель стека спецификаторов.
SPCPLS - признак того, что очередной элемент спецификатора должен интерпретироваться как положительный.
SPCWRK - рабочая переменная.

Стек спецификаторов имеет следующую структуру. Каждая строка стека спецификаторов состоит из двух полей: SPCSPLS и SPCSVPC. SPCSPLS служит для сохранения значения переменной SPCPLS, а SPCSVPC - для сохранения значения SPCVPC.

Ниже приведено описание интерпретатора спецификаторов на языке звеньев. В этом описании используются следующие сокращения.

PUTSS(SPCPLS,SPCVPC); - служит сокращением для

```
SPCSPLS[SPCSP]=SPCPLS;
SPCSVPC[SPCSP]=SPCVPC;
```

GETSS(SPCPLS,SPCVPC); - служит сокращением для

```
SPCPLS=SPCSPLS[SPCSP];
SPCVPC=SPCSPLS[SPCSP];
```

Ниже приведено описание интерпретатора спецификаторов на языке звеньев.

```
*
* Предикат SPC(L,B);
*
SPC:      SPCSP=Ø;
          SPCVPC=L;
          SPCPLS=.TRUE.;
          GOTO SPCNXT;
*
* Возврат из элемента спецификатора,
* если он сказал "да".
*
SPCRET:   IF(SPCSP .EQ. Ø) RETURN(SPCPLS);
          SPCSP=SPCSP-1;
          SPCWRK=SPCPLS;
          GETSS(SPCPLS,SPCVPC);
          IF(SPCWRK) GOTO SPCRET;
          SPCVPC=SPCVPC+LBLL;
*
* Возврат из элемента спецификатора,
* если он сказал "нет".
*
SPCNXT:   SPCOPC=NMB(SPCVPC);
          SPCVPC=SPCVPC+NMBL;
          GOTO SPCOP(SPCOPC);
*
* SPCCLL(L); - Вызов спецификатора из спецификатора.
*
SPCCLL:   PUTSS(SPCPLS,SPCVPC);
          SPCSP=SPCSP+1;
          SPCVPC=LBL(SPCVPC);
          SPCPLS=.TRUE.;
          GOTO SPCNXT;
*
* Элементы спецификатора переходят на
* SPCRET, если хотят сказать "да", и на
* SPCNXT, если хотят сказать "нет".
*
SPCW:     GOTO SPCRET;
```

```
SPCNG:   SPCPLS= .NOT. SPCPLS;
         GOTO SPCNXT;

SPCNGW:  SPCPLS= .NOT. SPCPLS;
         GOTO SPCRET;

SPCSC:   IF(CODE(B) .EQ. SMB(SPCVPC)) GOTO SPCRET;
         SPCVPC=SPCVPC+SMBL;
         GOTO SPCNXT;

SPCS:    IF(.NOT. BRA(B)) GOTO SPCRET;
         GOTO SPCNXT;

SPCB:    IF(BRA(B)) GOTO SPCRET;
         GOTO SPCNXT;

SPCF:    IF(TAG(B) .EQ. TAGF) GOTO SPCRET;
         GOTO SPCNXT;
*
* Совершенно аналогично устроены
*   SPCN, SPCR, SPCO, SPCD, SPCL.
*
* Замечание. Можно было бы изгнать переменную
* SPCPLS, но тогда пришлось бы иметь каждый элемент
* спецификатора в двух вариантах: положительном и
* отрицательном.
*
```

Для сопоставления переменных, имеющих спецификатор, предусмотрено несколько операторов языка сборки.

Если требуется сопоставить вхождение S- или W-переменной, либо закрытое или повторное вхождение VE-переменной, можно действовать в два этапа: сначала проделать сопоставление так, словно спецификатора у вхождения нет, а затем проверить, что полученное в результате сопоставления значение переменной удовлетворяет спецификатору. Для выполнения этой проверки используются два оператора: WSPC(L); и ESPC(L);.

Оператор WSPC(L); относится к группе L. Он проверяет, что

ET[NEL-1] указывает на звено, которое удовлетворяет спецификатору L. Таким образом, если этот оператор следует за оператором, производящим сопоставление S- или W-переменной, делается проверка на то, что значение этой переменной удовлетворяет спецификатору L. Если звено, на которое указывает ET[NEL-1], удовлетворяет спецификатору L, WSPC(L); прорабатывает успешно, в противном случае - терпит неудачу.

WSPC(L); имеет следующее описание на языке звеньев:

```
WSPC:   IF(.NOT. SPC(L,ET[NEL-1])) GOTO FAIL;
        GOTO NEXTOP;
```

Ниже приведен пример использования WSPC(L);.

```
 3       4       5,6      7       8,9     2
FUNC    S:X:A   W:Y:B    S:Z:A   W:X:B   =

    LS; WSPC(X); LW; WSPC(Y);
    LSD(4); WSPC(Z); LED(6); WSPC(X); NIL;
```

Оператор ESPC(L); относится к группе L. Он проверяет, что ET[NEL-2] и ET[NEL-1] указывают на начало и конец выражения, каждый терм которого, находящийся на нулевом уровне скобочной структуры, удовлетворяет спецификатору L. Если это условие выполнено, ESPC(L); прорабатывает успешно, в противном случае - терпит неудачу. Таким образом, если перед ESPC(L); находится оператор, который сопоставляет вхождение VR-переменной, ESPC(L); проверяет, что значение этого вхождения удовлетворяет спецификатору L.

ESPC(L); имеет следующее описание на языке звеньев:

```
ESPC:   BØ=PREV(ET[NEL-2]);
ESPCI:  IF(BØ .EQ. ET[NEL-1]) GOTO NEXTOP;
        BØ=NEXT(BØ);
        IF(BRA(BØ)) BØ=INFO(BØ);
        IF(.NOT. SPC(L,BØ)) GOTO FAIL;
        GOTO ESPCI;
```

Ниже приведен пример использования ESPC(L);.

```
  3      4    6,7     5    10,11   8,9     2
FUNC     (   E:X:A    )   V:Y:B   E:Z:A   =

    LBCE; ESPC(X); RED(7); ESPC(Z);
    CE; NNIL; ESPC(Y);
```

Оператор ESPC(L);, в принципе, позволяет осуществить сопоставление не только закрытых и повторных вхождений VE-переменных, но и сопоставление открытых вхождений VE-переменных. Для этого достаточно добавлять оператор ESPC(L); после операторов LE; и RE; в тех случаях, когда открытое вхождение имеет спецификатор. Однако, при этом сопоставление выполнялось бы неэффективно, ибо оператор ESPC(L); при каждом удлинении значения переменной повторно проверял бы одни и те же термы, в то время как достаточно проверять при удлинении только тот терм, который добавляется к значению переменной. Кроме того, если очередной добавляемый терм не удовлетворяет спецификатору, можно прекратить дальнейшие попытки удлинить значение этой переменной, так как при дальнейших удлинениях уже невозможно будет получить значение переменной, удовлетворяющее спецификатору. Поэтому вместо комбинации из операторов LE;, RE; и ESPC(L); заменяются на эквивалентные им (но более эффективные) комбинации операторов следующим образом:

```
PLE; LE; ESPC(L);   ⇒   PLESPC; LESPC(L);
PLV; LE; ESPC(L);   ⇒   PLV;    LESPC(L);
PRE; RE; ESPC(L);   ⇒   PRESPC; RESPC(L);
PRV; RE; ESPC(L);   ⇒   PRV;    RESPC(L);
```

Дополнительные операторы имеют следующее описание на языке звеньев:

```
PLESPC:  PUTJS(B1,B2,NEL,VPC);
         JSP=JSP+1;
         ET[NEL]=NEXT(B1);
```

```
        ET[NEL+1]=B1;
        NEL=NEL+2;
        VPC=VPC+NMBL+LBLL;
        GOTO NEXTOP;

LESPC:  B1=ET[NEL+1];
        SHB1;
        IF(BRA(B1)) B1=INFO(B1);
        IF(.NOT. SPC(L,B1)) GOTO FAIL;
        JSP=JSP+1;
        ET[NEL+1]=B1;
        NEL=NEL+2;
        GOTO NEXTOP;

PRESPC: PUTJS(B1,B2,NEL,VPC);
        JSP=JSP+1;
        ET[NEL+1]=PREV(B2);
        ET[NEL]=B2;
        NEL=NEL+2:
        VPC=VPC+NMBL+LBLL;
        GOTO NEXTOP;

RESPC:  B2=ET[NEL];
        SHB2;
        IF(BRA(B2)) B2=INFO(B2);
        IF(.NOT. SPC(L,B2)) GOTO FAIL;
        JSP=JSP+1;
        ET[NEL]=B2;
        NEL=NEL+2;
        GOTO NEXTOP;
```

В некоторых случаях оказывается, что можно выполнить отождествление, набрав максимально возможное значение VE-переменной. При этом можно воспользоваться операторами LMAX(L); и RMAX(L);.

Оператор LMAX(L); относится к группе L. Он продвигает границу B1 вправо, до тех пор, пока не встретит терм, который не удовлетворяет спецификатору L. После этого в ET[NEL]

и в ET[NEL+1] заносится адрес начала и адрес конца выражения, образованного термами, через которые прошла B1. Каждый из этих термов удовлетворяет спецификатору L.

Оператор RMAX(L); является зеркальным отражением оператора LMAX(L);.

LMAX(L); и RMAX(L); имеют следующее описание на языке звеньев:

```
LMAX:   ET[NEL]=NEXT(B1);
LMAX1:  B1=NEXT(B1);
        IF(B1 .EQ. B2) GOTO LMAX2;
        IF(.NOT. SPC(L,B1)) GOTO LMAX2;
        IF(BRA(B1)) B1=INFO(B1);
        GOTO LMAX1;
LMAX2:  B1=PREV(B1);
        ET[NEL+1]=B1;
        NEL=NEL+2;
        GOTO NEXTOP;

RMAX:   ET[NEL+1]=PREV(B2);
RMAX1:  B2=PREV(B2);
        IF(B1 .EQ. B2) GOTO RMAX2;
        IF(.NOT. SPC(L,B2)) GOTO RMAX2;
        IF(BRA(B2)) B2=INFO(B2);
        GOTO RMAX1;
RMAX2:  B2=NEXT(B2);
        ET[NEL]=B2;
        NEL=NEL+2;
        GOTO NEXTOP;
```

Рассмотрим пример использования операторов LMAX(L); и RMAX(L);. Пусть требуется скомпилировать левую часть

```
FUNC    E1   E:X:2   =
```

В соответствии с описанием рефала, при отождествлении требуется подобрать самое короткое значение для E1, при котором возможно отождествление. Таким образом,

прямолинейный перевод на язык делается следующим образом:

```
  3       4,5   6,7    2
FUNC      E1    E:X:2  =

    PLE; LE; CE; ESPC(X);
```

Такая программа на языке сборки будет, однако, работать неэффективно, так как оператор ESPC(X); будет после каждого удлинения повторно проверять одни и те же звенья на соответствие спецификатору. Можно, однако воспользоваться оператором RMAX(L); исходя из следующего рассуждения.

Самому короткому E1, при котором возможно отождествление, соответствует самое длинное E2, при условии, что оно состоит из термов, удовлетворяющих спецификатору X. Поэтому будет правильным (и эффективным) следующий перевод предложения на язык сборки:

```
  3       6,7   4,5    2
FUNC      E1    E:X:2  =

    RMAX(X); CE;
```

В этом варианте отождествление выполняется без всякого перебора.

3.14. ОПЕРАТОРЫ ПРЕОБРАЗОВАНИЯ ПОЛЯ ЗРЕНИЯ

До сих пор рассматривались операторы отождествления. Их отличительной чертой было то, что они не изменяли поле зрения, а только анализировали его, накапливая информацию в таблице элементов.

Начиная с этого раздела, рассматриваются операторы преобразования. Они (почти) ничего не анализируют в поле зрения, но зато преобразуют его, используя адреса, накопленные в таблице элементов.

В то время, как операторы отождествления использовались

при компиляции левой части рефал-предложения, операторы преобразования используются при компиляции его правой части.

Задача операторов преобразования - сформировать результат выполнения шага рефал-машины или, как мы будем говорить, результат замены левой части предложения на правую. Конечно, в действительности все преобразования выполняются над полем зрения, но в процессе компиляции, для наглядности, можно представлять себе дело так, будто преобразуется не поле зрения, а некоторое выражение, содержащее свободные переменные. Это выражение сначала является пустым, но в процессе выполнения преобразований постепенно превращается в правую часть предложения.

Такой подход оправдан в том смысле, что преобразования, выполняемые над выражением, содержащим свободные переменные, изображает в "алгебраическом" общем виде преобразования над полем зрения. Поэтому в дальнейшем мы будем говорить о перестановке, копировании и уничтожении переменных, хотя в действительности все эти операции интерпретатор языка сборки будет проделывать над значениями переменных.

3.15. СПИСОК СВОБОДНОЙ ПАМЯТИ

Когда отождествление окончено, начинается формирование результата замены. В момент окончания отождествления результат замены еще пуст, а к концу замены - он должен совпасть с правой частью предложения. Правая часть предложения состоит из символов, скобок (как структурных, так и функциональных) и переменных. Задача операторов замены - добавить недостающие элементы правой части в результат замены. Откуда же они могут взять эти элементы.

Недостающие элементы можно либо породить заново (взяв откуда-то дополнительные звенья), либо из'ять какие-то звенья из ведущего функционального терма. Например, если какая-то переменная входит в левую часть три раза, а в правую - только два раза, то нет необходимости создавать заново значение этой переменной, достаточно переставить два экземпляра ее значения из аргумента функции в результат

замены. Если же переменная входит в левую часть два раза, а в правую - три раза, то придется один раз выполнить копирование значения этой переменной.

Для формирования элементов правой части требуется откуда-то брать дополнительные звенья. Источником этих звеньев является с п и с о к с в о б о д н ы х з в е н ь е в (ССЗ).

Когда требуется вставить новые звенья в поле зрения, они берутся из ССЗ. Когда требуется убрать какие-то звенья из поля зрения, они присоединяются к ССЗ.

Список свободных звеньев организован как двусвязный симметричный списск. Каждое звено, входящее в ССЗ, в поле PREV содержит ссылку на предыдущее звено, а в поле NEXT - ссылку на следующее звено. Исключением являются только первое и последнее звено ССЗ, которые ссылаются на особое звено, именуемое г о л о в о й ССЗ. Первое звено содержит ссылку на голову в поле PREV, а последнее - в поле NEXT. Таким образом ССЗ "закручен в кольцо".

В процессе преобразования поля зрения может возникнуть ситуация, при которой результат замены уже наполовину сформирован, но продолжить работу невозможно, так как в ССЗ не хватает свободных звеньев. В этом случае следует либо добыть откуда-то дополнительную память для звеньев, либо собрать мусор, либо аварийно завершить работу всей рефал-программы. Интерпретатор языка сборки сам этими вопросами не занимается. Если возникает нехватка свободной памяти, он приводит поле зрения в то состояние, в котором оно было перед началом шага, и передает управление вызвавшей его программе, сообщив ей, что он остановился по нехватке свободной памяти. Если головной программе удается изыскать дополнительные звенья, она может снова вызвать интерпретатор языка сборки. При этом выполнение шага возобновится с самого начала (с отождествления), так, словно раньше и не было попытки его выполнить.

Таким образом, замена должна быть организована так, чтобы в случае нехватки памяти можно было вернуться к началу шага. Это достигается следующим образом. Результат замены всегда формируется на звеньях ССЗ. При этом сначала порождаются все

новые символы и скобки и выполняются все копирования переменных. При этом ведущий функциональный терм остается без изменения, если не считать того, что некоторые структурные скобки могут оказаться преобразованными в функциональные скобки. Только после того, как все порождения и копирования закончены, в результат переставляются куски из ведущего терма. При этом уже никаких неприятностей возникнуть не может, ибо дополнительные звенья не требуются. После этого ведущий функциональный терм удаляется из поля зрения и присоединяется к ССЗ, а результат замены удаляется из ССЗ и вставляется в поле зрения вместо удаленного функционального терма.

В процессе замены интерпретатор языка сборки использует следующие переменные:

FLHEAD - указатель на голову списка свободных звеньев. Значение этого указателя в процессе работы интерпретатора языка сборки не изменяется.
F - указатель на текущее звено свободной памяти.
LASTB - адрес последней порожденной левой скобки.
LASTK - адрес последнего активизированного знака "<".
QUASIK - указатель на фиктивный знак "<". Значение этого указателя в процессе работы интерпретатора языка сборки не изменяется.
TS - стек трансплантаций.
TSP - указатель стека трансплантаций.

Указатель F используется в интерпретаторе языка сборки следующим образом. Перед началом замены он устанавливается на голову ССЗ. Затем, по мере порождения элементов результата замены, F продвигается по ССЗ вправо (т.е. в направлении полей NEXT), при этом F всегда указывает на последнее израсходованное звено.

Если при очередной попытке продвинуть F вправо оказывается, что F наскочил на голову ССЗ, это означает, что звеньев не хватает и управление передается на программу, помеченную меткой LACK. Эта подпрограмма останавливает работу интерпретатора языка сборки как было описано выше.

При описании операторов замены используются следующие сокращения.

SHF; - сдвинуть указатель F. Является сокращением для

```
F=NEXT(F);
IF(F .EQ. FLHEAD) GOTO LACK;
```

LINK(X,Y); - связать звенья X и Y. Является сокращением для

```
NEXT(X)=Y;
PREV(Y)=X;
```

Для того, чтобы подготовить интерпретатор языка сборки к выполнению замены используется оператор EOR;.

Оператор EOR; относится к группе 0. Этим оператором должна заканчиваться последовательность операторов, полученная в результате компиляции левой части. Вслед за этим оператором располагаются операторы, полученные в результате компиляции правой части. EOR; имеет следующее описание на языке звеньев.

```
EOR:    F=FLHEAD;
        LASTK=QUASIK;
        LASTB=Ø;
        TSP=Ø;
        GOTO NEXTOP;
```

3.16. ПОРОЖДЕНИЕ ОБ'ЕКТНЫХ ВЫРАЖЕНИЙ

В этом разделе рассматриваются операторы, которые позволяют породить произвольное об'ектное выражение.

Оператор NS(S); относится к группе S. Он вставляет в результат замены новый символ S следующим образом. Сначала NS(S); продвигает F на следующее звено. Если после этого F указывает на голову ССЗ, это означает, что ССЗ исчерпан, и управление передается на метку LACK. В противном случае в звене, на которое указывает F, создается символ S.

```
NS:     SHF;
        CODE(F)=S;
        GOTO NEXTOP;
```

Оператор BL; относится к группе O. Он вставляет в результат замены левую структурную скобку.

Оператор BR; относится к группе O. Он вставляет в результат замены правую структурную скобку.

Каждая структурная скобка в поле зрения должна содержать в поле INFO ссылку на парную скобку. Об этом обязаны позаботиться операторы BL; и BR;.

Для связывания парных скобок используется следующий метод. Все неуравновешенные левые скобки, для которых в результат замены еще не вставлены парные правые скобки, связываются в односвязный список. Для этого используются поля INFO в этих скобках. В переменной LASTB хранится адрес начала этого списка.

Когда в результат замены вставляется левая скобка, она присоединяется к списку левых скобок. Когда вставляется новая правая скобка, то из списка левых скобок убирается одна левая скобка, которая связывается с вставляемой правой скобкой.

Операторы BL; и BR; имеют следующее описание на языке звеньев:

```
BL:     SHF;
        INFO(F)=LASTB;
        LASTB=F;
        GOTO NEXTOP;

BR:     SHF;
        INFO(F)=LASTB;
        FI=INFO(LASTB);
        INFO(LASTB)=F;
        TAG(LASTB)=TAGLB;
        TAG(F)=TAGRB;
        LASTB=FI;
        GOTO NEXTOP;
```

С помощью операторов NS(S);, BL; и BR; мы можем порождать любые об'ектные выражения. Например, выражение

```
'A' ( 'B' ( ) ) 'C'
```

порождается последовательностью операторов:

```
NS('A'); BL; NS('B'); BL; BR; BR; NS('C');
```

Для некоторых, часто встречающихся случаев предусмотрены дополнительные операторы. Пусть C, C1, C2,... обозначают символы-литеры (об'ектные знаки). Тогда можно следующим образом записать правила, по которым можно заменять комбинации из одних операторов другими:

```
BL; BR;  ⇒  BLR;
NS(C);   ⇒  NSO(C);
BL; NS(/L/);  ⇒  BLF(L);
NS(C1); ... NS(CN);  ⇒  TEXT(N,C1,...,CN);
```

Оператор BLR; относится к группе 0. Оператор NSO(N); относится к группе N. Его аргумент - об'ектный знак, который как раз помещается в один байт. Оператор BLF(L); относится к группе L. Поскольку его аргумент - символ-метка, не

требуется хранить в аргументе поле TAG, а поле INFO по размерам соответствует аргументу типа L. Оператор TEXT(N,C1,C2,...,CN); относится к группе N, поскольку дешифратор извлекает из памяти только его первый аргумент. Остальные аргументы этот оператор извлекает самостоятельно. Каждый из аргументов C1,C2,...,CN занимает столько же места, сколько аргумент типа N.

```
NSO:    SHF;
        TAG(F)=TAGO;
        INFO(F)=N;
        GOTO NEXTOP;

TEXT:   SHF;
        TAG(F)=TAGO;
        INFO(F)=NMB(VPC);
        VPC=VPC+NMBL;
        N=N-1;
        IF(N .NE. 0) GOTO TEXT;
        GOTO NEXTOP;

BLR:    SHF;
        F1=F;
        SHF;
        INFO(F1)=F;
        INFO(F)=F1;
        TAG(F1)=TAGLB;
        TAG(F)=TAGRB;
        GOTO NEXTOP;

BLF:    SHF;
        INFO(F)=LASTB;
        LASTB=F;
        SHF;
        INFO(F)=L;
        TAG(F)=TAGF;
        GOTO NEXTOP;
```

3.17. ПОРОЖДЕНИЕ ФУНКЦИОНАЛЬНЫХ СКОБОК

Согласно описанию языка рефал, в начале каждого шага рефал-машина должна просматривать поле зрения в поисках ведущего функционального терма. Ясно, что время такого ассоциативного поиска пропорционально размерам поля зрения, и при реализации рефала лучше постараться без него обойтись.

К счастью, можно придумать такое представление для "<" и ">", что время поиска ведущего функционального терма не будет зависеть от размеров поля зрения. Достигается это следующим образом.

Функциональные скобки занимают по одному звену каждая. Имя функции хранится в виде символа-метки, сразу же вслед за знаком "<", и занимает отдельное звено.

Звено, соответствующее знаку "<", содержит в поле TAG признак TAGK. Звено, соответствующее знаку ">", содержит в поле TAG признак TAGD.

Знак ">" в поле INFO содержит адрес парного к нему знака "<". Знак "<" в поле INFO содержит адрес скобки ">", которая станет ведущей, после полного вычисления данной области конкретизации. Если же такой скобки ">" не существует, знак "<" в поле INFO содержит нуль.

Таким образом, знаки "<" и ">" связаны в список в том порядке, в котором они будут становиться ведущими.

Интересно отметить, что скобки ">" расположены в поле зрения в точности в том порядке, в котором они будут становиться ведущими, т.е. самая левая скобка всегда является ведущей. В то же время нет никакой очевидной связи между расположением скобок "<" и порядком, в котором они будут становиться ведущими.

Порождение функциональных скобок выполняется в два этапа. Сначала в результат замены вставляются структурные скобки, а затем они активизируются, т.е. превращаются в функциональные.

Оператор BRACT; относится к группе 0. Он вставляет в результат замены новую правую скобку ")" и тут же активизирует ее вместе с парной к ней скобкой "(". При этом адрес парной левой скобки извлекается из переменной LASTB, а адрес

последней из созданных "<" - из переменной LASTK. Этот адрес нужен, ибо в скобку "<", находящуюся по этому адресу требуется, вставить ссылку на порождаемую скобку ">".

```
BRACT:   SHF;
         INFO(F)=LASTB;
         TAG(F)=TAGD;
         INFO(LASTK)=F;
         TAG(LASTK)=TAGK;
         LASTK=LASTB;
         LASTB=INFO(LASTB);
         GOTO NEXTOP;
```

В некоторых случаях правая и левая структурные скобки не создаются заново, а переносятся из ведущего функционального терма. Такую пару скобок можно активизировать с помощью оператора ACT(N);.

Оператор ACT(N); относится к группе N. Он активизирует правую скобку ")", адрес которой содержится в ET[N], а также парную к ней левую скобку "(".

```
ACT:     INFO(LASTK)=ET[N];
         TAG(LASTK)=TAGK;
         LASTK=INFO(ET[N]);
         TAG(ET[N])=TAGD;
         GOTO NEXTOP;
```

Например, выражение

```
'A'<X(<Y>)><Z>
```

можно породить с помощью операторов

```
NSO('A'); BLF(X); BL; BLF(Y); BRACT;
BR; BRACT; BLF(Z); BRACT;
```

Таким образом, оператор ACT(N); может преобразовать некоторые структурные скобки, входящие в ведущий функциональный

терм, в функциональные скобки. Поэтому, если выполнение шага прерывается из-за нехватки свободной памяти, и управление передается на программу LACK, необходимо привести эти скобки в исходное состояние. Поэтому программа LACK должна прежде всего выплнять следующие действия.

```
LACK:   INFO(LASTK)=Ø;
        LASTB=INFO(QUASIK);
LACK1:  IF(LASTB .EQ. Ø) GOTO LACK2;
        FØ=INFO(LASTB);
        TAG(LASTB)=TAGRB;
        F1=INFO(FØ);
        INFO(FØ)=LASTB;
        TAG(FØ)=TAGLB;
        LASTB=F1;
        GOTO LACK1;
LACK2:  INFO(QUASIK)=ET[2];
```

3.18. КОПИРОВАНИЕ ПЕРЕМЕННЫХ

Оператор MULS(N); относится к группе N. Он копирует ("размножает") значение переменной символа, ссылка на которое содержится в ET[N]. Копия значения переменной добавляется справа к результату замены.

```
MULS:   SHF;
        CODE(F)=CODE(ET[N]);
        GOTO NEXTOP;
```

Оператор MULE(N); относится к группе N. Он копирует значение W- или VE- переменной, на начало и конец которого ссылаются ET[N-1] и ET[N] соответственно.

```
MULE:   FØ=PREV(ET[N-1]);
MULE1:  IF(FØ .EQ. ET[N]) GOTO NEXTOP;
        FØ=NEXT(FØ);
        SHF;
```

```
        IF(BRA(FØ)) GOTO MULE2;
        CODE(F)=CODE(FØ);
        GOTO MULE1;
MULE2:  IF(TAG(FØ) .EQ. TAGRB) GOTO MULE3;
        INFO(F)=LASTB;
        LASTB=F;
        GOTO MULE1;
MULE3:  INFO(F)=LASTB;
        TAG(F)=TAGRB;
        F1=INFO(LASTB);
        INFO(LASTB)=F;
        TAG(LASTB)=TAGLB;
        LASTB=F1;
        GOTO MULE1;
```

3.19. ПЕРЕСАДКА КУСКОВ СПИСКА

Во многих случаях целесообразно не создавать элементы результата замены заново, а переносить куски списка из ведущего функционального терма в результат замены. В особенности это справедливо по отношению к значениям W- и VE-переменных, так как эти значения могут иногда состоять из нескольких тысяч звеньев.

Мы будем называть переставляемый участок списка трансплантатом , а саму операцию пересадки - трансплантацией .

Все трансплантации выполняются в самом конце замены, после того, как выполнены все порождения и копирования. Однако, информацию о тех местах, куда должны быть переставлены трансплантаты, удобно накапливать во время выполнения порождений и копирований. Поэтому для накопления этой информации в интерпретаторе языка сборки предусмотрен особый стек: с т е к т р а н с п л а н т а ц и й (TS).

Каждая строка стека трансплантаций состоит из трех полей: TSØ, TS1 и TS2, каждое из которых может содержать одну ссылку на некоторое звено списка. TSØ содержит ссылку на звено, после которого необходимо вставить трансплантат. TS1 и TS2 содержат ссылки на начало и конец трансплантата. Стек

TS может физически занимать то же место в памяти, что и стек переходов JS, ибо JS и TS никогда не нужны одновременно.

Занесение в стек трансплантаций сведений о трансплантате и о месте, куда его нужно перенести, мы будем называть п л а н и р о в а н и е м пересадки.

В дальнейшем описании будем использовать следующие сокращения.

PUTTS(Z,X,Y); - является сокращением для

```
TS0[TSP]=Z;
TS1[TSP]=X;
TS2[TSP]=Y;
```

GETTS(Z,X,Y); - является сокращением для

```
Z=TS0[TSP];
X=TS1[TSP];
Y=TS2[TSP];
```

Оператор TPL(N,M); относится к группе NN. Он предназначен для планирования пересадки, т.е. для занесения информации в стек TS. При этом предполагается, что ET[N] и ET[M] содержат ссылки на начало и конец трансплантата соответственно, а вставлять трансплантат нужно будет после звена, на которое указывает F. В том случае, если трансплантат пуст, в стек TS ничего не заносится, так как для пересадки пустого выражения не требуется выполнять какие-либо действия.

В некоторых случаях заведомо известно, что трансплантат не пуст, поэтому можно опустить проверку на пустоту трансплантата. Для таких случаев предусмотрен оператор TPLM(N,M);, который относится к группе NN.

```
TPL:    IF(NEXT(ET[M]) .EQ. ET[N]) GOTO NEXTOP;
TPLM:   PUTTS(F,ET[N],ET[M]);
        TSP=TSP+1;
        GOTO NEXTOP;
```

Для некоторых важных частных случаев предусмотрены особые операторы. Ниже показано, на какие операторы можно заменять TPL(N,M); и TPLM(N,M); и при каких условиях.

```
TPL(N-1,N);   ⇒   TPLE(N);
TPLM(N-1,N);  ⇒   TPLV(N);
TPLM(N,N);    ⇒   TPLS(N);
```

Ниже приведено описание операторов TPLE(N);, TPLV(N); и TPLS(N); на языке звеньев.

```
TPLE:   IF(NEXT(ET[N]) .EQ. ET[N-1]) GOTO NEXTOP;
TPLV:   PUTTS(F,ET[N-1],ET[N]);
        TSP=TSP+1;
        GOTO NEXTOP;

TPLS:   PUTTS(F,ET[N],ET[N]);
        TSP=TSP+1;
        GOTO NEXTOP;
```

Ниже приведен пример использования операторов копирования и пересадки.

```
  3      4,5  6   7,8  2   4,5  6   7,8
FUNC     E1  '+'  E2   =   E1  '+'  E2   E1

    LSRCH('+'); CE; EOR;
    TPLM(4,8); MULE(5); EOS;
```

Скомпилированное предложение заканчивается оператором EOS; (конец шага). Этот оператор описан в следующем разделе.

3.20. ЗАВЕРШЕНИЕ ОЧЕРЕДНОГО ШАГА И ПОДГОТОВКА СЛЕДУЮЩЕГО

Оператор EOS; относится к группе 0. Он завершает очередной шаг рефал-машины и начинает выполнение нового шага.

EOS; начинает свою работу с того, что "зашивает дырку" в списке функциональных скобок. А именно, в последнюю из активизированных скобок "<" вставляется ссылка на ту скобку ">", на которую указывает скобка "<", с которой начинается ведущий функциональный терм. Обратите внимание на то, что если в правой части предложения нет ни одной пары функциональных скобок, получится, что будет исправлен адрес, находящийся в поле INFO фиктивной скобки "<" (на которую указывает QUASIK).

После этого выполняются все запланированные трансплантации, информация о которых находится в стеке трансплантаций TS. Эти трансплантации выполняются в порядке, обратном к тому, в котором выполнялись операторы планирования трансплантаций. Почему это делается именно так, об'ясняется в следующем разделе.

Затем меняются местами сформированный результат замены и ведущий функциональный терм. Результат замены вставляется на место функционального терма, а ведущий функциональный терм присоединяется к списку свободной памяти.

На этом завершается выполнение очередного шага и начинается подготовка следующего шага.

Новый шаг начинается с того, что из INFO(QUASIK) извлекается адрес. Если этот адрес равен нулю, это означает, что в поле зрения не осталось ни одного функционального терма, поэтому работа рефал-машины успешно завершается. Если же этот адрес не равен нулю, он является адресом самой левой скобки ">" в поле зрения. Эта ">" является ведущей. Тогда проверяется, что вслед за ведущей "<" находится символ-метка. Если это не так, то следует рассмотреть отдельно случаи, когда вслед за "<" расположен символ-ссылка или что-то другое.

Если вслед за "<" находится символ-метка, в VPC из поля INFO этого символа загружается адрес начала соответствующей программы на языке сборки.

После этого заполняются ET[1], ET[2], ET[3], устанавливаются B1, B2, NEL и JSP, и управление передается в дешифратор операторов языка сборки.

```
*
* EOS;
*
EOS:    INFO(LASTK)=INFO(ET[1]);
        TAG(LASTK)=TAGK;
        NEXTR=NEXT(F);
*
* Выполнить запланированные пересадки.
*
EOS1:   IF(TSP .EQ. 0) GOTO INSRES;
        TSP=TSP-1;
        GETTS(F,F0,F1);
        LINK(PREV(F0),NEXT(F1));
        LINK(F1,NEXT(F));
        LINK(F,F0);
        GOTO EOS1;
*
* Вставить результат замены.
*
INSRES: IF(NEXT(FLHEAD) .EQ. NEXTR) GOTO INSRES1;
        LINK(PREV(NEXTP),NEXT(ET[2]));
        LINK(PREV(ET[1]),NEXT(FLHEAD));
        GOTO DELKD;
INSRES1: LINK(PREV(ET[1]),NEXT(ET[2]));
*
* Удалить "<" и ">".
*
DELKD:  LINK(ET[2],NEXTR);
        LINK(FLHEAD,ET[1]);
*
* Начало нового шага.
*
START:  B2=INFO(QUASIK);
        IF(B2 .EQ. 0) GOTO DONE;
```

```
          B0=INFO(B2);
          B1=NEXT(B0);
          VPC=INFO(B1);
          IF(TAG(B1) .NE. TAGF) GOTO REF;
          JSP=0;
          ET[1]=B0;
          ET[2]=B2;
          ET[3]=B1;
          NEL=4;
          GOTO NEXTOP;
*
* Выполнить шаг для функции-ссылки.
*
REF:      IF(TAG(B1) .NE. TAGR) GOTO RCGIMP;
          ET[1]=B0;
          ET[2]=B2;
          ET[3]=B1;
          GOTO SWAPREF;
*
* SWAP;
* (После кода оператора находится
*  голова статического ящика.)
*
SWAP:     IF(PREV(VPC) .NE. 0) SWAPREF;
          LINK(VPC,VPC);
          INFO(VPC)=REFAL.SVAR;
          TAG(VPC)=0;
          REFAL.SVAR=VPC;
SWAPREF:  INFO(QUASIK)=INFO(ET[1]);
          IF(NEXT(VPC) .EQ. VPC) GOTO SWAP2;
          LINK(PREV(VPC),NEXT(ET[2]));
          LINK(PREV(ET[1]),NEXT(VPC));
          GOTO SWAP3;
SWAP2:    LINK(PREV(ET[1]),NEXT(ET[2]));
SWAP3:    IF(NEXT(ET[3]) .EQ. ET[2]) GOTO SWAP4;
          LINK(PREV(ET[2]),VPC);
          LINK(VPC,NEXT(ET[3]));
          LINK(ET[3],ET[2]);
```

```
        GOTO SWAP5;
SWAP4:  LINK(VPC,VPC);
SWAP5:  LINK(ET[2],NEXT(FLHEAD));
        LINK(FLHEAD,ET[1]);
        GOTO START;
```

3.21. ПОРОЖДЕНИЕ ОПЕРАТОРОВ ПРЕОБРАЗОВАНИЯ

Выше были описаны операторы преобразования поля зрения, однако не рассматривался вопрос, каким образом следует их генерировать при компиляции правой части предложения.

Ясно, что генерация операторов порождения и копирования не составляет особого труда. Другое дело - операторы пересадки. При их генерации основная трудность состоит в том, что пересадки не выполняются сразу, а откладываются до конца шага, и это должен учитывать алгоритм компиляции правой части.

Если бы все трансплантации выполнялись сразу же, а не откладывались, компиляция правой части не составляла бы никакого труда.

Алгоритм компиляции правой части в качестве исходной информации имел бы два выражения: левую часть предложения [L] без "<" и "=" (вместе с номерами элементов, получившимися при компиляции левой части), а также правую часть предложения [R]. Результатом его работы была бы последовательность операторов преобразования.

Алгоритм состоял бы из следующих шагов:

1. Если [R] - пустое выражение, то породить EOS; и закончить работу. Иначе - перейти к пункту 2.

2. Попытаться подобрать в [L] трансплантат [T], т.е. представить [R] и [L] в виде [R]=[T]R1, [L]=[L1][T][L2]. Если это не удалось, то перейти к пункту 3. Иначе - породить оператор TPL(N,M);, где N - номер левого конца для первого элемента из [T], а M - номер правого конца для последнего элемента из [T].

Заменить [R] на [R1], а [L] - на [L1][L2] и перейти к пункту 1.

3. Найти элемент [Z], с которого начинается [R], т.е. представить [R] в виде [R]=[Z][R1]. Если [Z] равен символу или "(", или"<", или ")", или">", или S-переменной, или W- либо VE-переменной, то породить соответственно NS(S); или BL;, или BL;, или BR;, или BRACT;, или MULS(N);, или MULE(N);. Заменить [R] на [R1] и перейти к пункту 1.

В этом описании алгоритма опущены некоторые детали, которые нам пока не существенны. Так, например, при порождении трансплантаций нужно следить за тем, чтобы не нарушалась парность скобок, а также учитывать возможность активизации скобок оператором ACT(N);.

Главное в этом алгоритме то, что сразу после порождения TPL(N,M); трансплантат удаляется из левой части предложения. Например, пусть

[R] = 'CBAD' , [L] = 'ABCD'

Будем изображать упорядоченную пару из [R] и [L] в виде [R]*[L]. Тогда работу алгоритма компиляции можно изобразить следующим образом.

```
                      4   5   6   7   TPLS(6);
'C' 'B' 'A' 'D' * 'A' 'B' 'C' 'D' ----------->

                  4   5   7   TPLS(5);
'B' 'A' 'D' * 'A' 'B' 'D' ----------->

              4   7   TPL(4,7);
'A' 'D' * 'A' 'D' -----------> *
```

Однако, как мы знаем, пересадки не выполняются сразу, а только планируются операторами TPL(N,M);, поэтому описанный выше алгоритм компиляции кажется совершенно непригодным.

Тем не менее, как это ни удивительно, алгоритм компиляции дает правильный результат также и в том случае, когда пересадки откладываются. При этом решающую роль играет то обстоятельство, что отложенные пересадки выполняются в о б р а т н о м порядке.

Так, в только что рассмотренном примере, последовательность операторов

```
TPLS(6); TPLS(5); TPL(4,7);
```

приведет к следующим преобразованиям над полем зрения (если обозначать пару из результата замены [R] и аргумента функции через [R]*[L]).

```
   4   5   6   7   TPL(4,7);
* 'A' 'B' 'C' 'D' ---------->

 4   5   6   7     TPLS(5);
'A' 'B' 'C' 'D' * ---------->

 5   4   6   7     TPLS(6);
'B' 'A' 'C' 'D' * ---------->

 6   5   4   7
'C' 'B' 'A' 'D'
```

Как видно, в конце-концов получается правильный результат замены. Чем же это об'яснить?

Дело в том, что каждый трансплантат состоит из элементов двух типов: "полезных", которые действительно требуется переставить в указанное место результата замены, и посторонних , которые в этом месте не нужны. Например, если через "О" обозначить полезные элементы, а через "Х" - посторонние, трансплантат может иметь следующий вид:

О О Х Х Х О О О Х Х Х Х О

Лишние элементы, однако, рано или поздно исчезают из

трансплантата, ибо они сами являются трансплантатами для других пересадок, которые будут выполняться позже. Но ведь те пересадки, которые выполняются позже, планируются раньше. Поэтому, подыскивая в левой части предложения трансплантат, компилятор должен игнорировать те элементы, которые являются частью трансплантатов для уже порожденных к этому моменту операторов пересадки. Ибо, хотя эти посторонние элементы будут перенесены вместе с полезными в новое место, они обязательно будут удалены теми пересадками, которые были запланированы раньше, но будут исполняться позже.

Но ведь самый простой способ игнорировать посторонние элементы при поисках трансплантатов - это удалять их из левой части сразу же после порождения соответствующего оператора TPL(N,M);! Таким образом мы и приходим к алгоритму компиляции правой части, описанному выше.

Почему же интерпретатор языка сборки выполняет запланированные пересадки в обратном порядке. Ведь, казалось бы, их можно, с тем же результатом, выполнять и в прямом порядке.

Причина заключается в том, что элементами трансплантатов могут быть E-переменные, а E-переменные могут принимать пустые значения.

Пусть, например,

```
                                          4,5  6,7  8,9
[R] = 'A' E2 'B' E3 ,   [L] = E1   E2   E3
```

Ясно, что можно получить [R] из [L] с помощью операторов

```
NS('A'); TPLE(7); NS('B'); TPL(4,9);
```

Нетрудно убедиться, что при выполнении пересадок в обратном порядке результат получается правильный, несмотря на то, что переменные E1, E2 и E3 могут принимать пустые значения.

А что получится, если выполнять пересадки в прямом порядке. Тогда нетрудно найти случай, в котором получается неверный результат преобразования.

Предположим, что E2 не пусто, а E1 и E3 - пустые. Тогда оператор TPLE(7); и оператор TPL(4,9); запланируют

пересадки, ибо в момент исполнения оператора TPLE(7); не пуст трансплантат E2, а в момент исполнения оператора TPL(4,9); не пуст трансплантат E1 E2 E3.

Из-за того, что E1 и E3 пустые, а E2 - не пустое, справедливы соотношения

```
ET[4] = ET[6] ,   ET[7] = ET[9]
```

(Поскольку для пустых значений E-переменных адреса начала и конца запоминаются "крест-накрест").

Если пересадки выполняются в обратном порядке, то сначала трансплантат E1 E2 E3 пересаживается после 'B', а затем из него вынимается E2 и пересаживается после 'A'. Процесс преобразования выглядит следующим образом:

```
'A' 'B'  *  E1 E2 E3  ⟶
'A' 'B' E1 E2 E3  *   ⟶
'A' E2 'B' E1 E3  *
```

Таким образом, одно и то же выражение E2 = E1 E2 E3 пересаживается два раза, но это не приводит ни к какой беде.

Теперь предположим, что пересадки выполняются в прямом порядке. Тогда трансплантат E2 пересаживается после 'A', а затем трансплантат E1 E3 пересаживается после 'B'. Но ведь адрес начала E1 совпадает с адресом начала E2, а адрес конца E3 совпадает с адресом конца E2. Таким образом, фактически получается, что E2 будет пересажено после 'B'.

```
'A' 'B'  *  E1 E2 E3  ⟶
'A' E1 E2 E3 'B'  *   ⟶
'A' 'B' E1 E2 E3  *
```

Следовательно, результатом замены будет 'AB' E2 , что неверно.

При выполнении пересадок в обратном порядке пустые значения E-переменных нисколько не вредят в силу того факта, что в момент выполнения пересадки трансплантат всегда в точности тот же самый, что был при планировании, поскольку изменения

в него могут быть внесены только после того, как пересадка уже выполнена.

Если же выполнять пересадки в прямом порядке, то оказывается, что часто к моменту выполнения пересадки трансплантат уже изменил свое состояние по сравнению с тем, которое было в момент планирования пересадки. В результате возникает путаница, которая может приводить к катастрофическим результатам.

3.22. ПРИМЕРЫ ПЕРЕВОДА ФУНКЦИЙ НА ЯЗЫК СБОРКИ

Теперь для любой функции, описанной на рефале, мы можем дать ее полный перевод на язык сборки.

```
        4  6,7  5          4       6,7  5
FUNC    (  E1  )  =  (  <  FUNC    E1  >  )

        4  6,7  5 8  9,10          4 6,7  8  5  9,10
        (  E1  ) SX  E2   =  <  FUNC ( E1   SX ) E2  >

        4,5      4,5
        E1   =   E1
```

```
FUNC:SJUMP(L2); LBCE;
     SJUMP(L1); NIL; EOR; BL; TPLS(4); NS(/FUNC/);
                TPLM(6,5); ACT(5); BR: EOS;
L1:  LS; CE; EOR; BLF(FUNC); TPLM(4,7); TPLS(8);
     TPLM(5,10); BRACT; EOS;
L2:  CE; EOR; TPLE(5); EOS;
```

```
        4    5,6          5,6
REV     SX   E1   =  <  REV  E1  > SX

        4  6,7  5  8,9         8,9    4       6,7 5
        (  E1  )  E2  =  <  REV  E2     (<  REV  E1  > )
```

```
REV: SJUMP(L3); LS; CE; EOR;
     BLF(REV); TPLE(6); BRACT; TPLS(4); EOS;
L3:  SJUMP(L4); LBCE; CE; EOR;
     BLF(REV); TPLE(9); BRACT; BL; TPLS(4);
     NS(/REV/); TPLM(6,5); ACT(5); BR; EOS;
L4:  NIL; EOR; EOS;
```

ГЛАВА 4. КОМПИЛЯТОР С РЕФАЛА НА ЯЗЫК СБОРКИ

4.1. ОБЩАЯ СТРУКТУРА КОМПИЛЯТОРА

Программа, написанная на рефале, состоит из некоторого числа функций. Компиляция каждой функции происходит независимо от компиляции остальных функций.

Различные предложения, принадлежащие к одной функции, также компилируются независимо друг от друга.

Компиляция каждого предложения состоит из двух этапов: компиляции левой части предложения и компиляции правой части предложения. Информация, накопленная компилятором при компиляции левой части, используется при компиляции правой части.

Когда все предложения функции скомпилированы, производится расстановка операторов SJUMP(L);. При этом производится об'единение совпадающих частей различных предложений.

Таким образом, компилятор с рефала на язык сборки имеет следующую структуру.

```
WHILE( не все функции скомпилированы );
  начать компиляцию очередной функции;
  WHILE( не все предложения функции скомпилированы );
    начать компиляцию очередного предложения:
    скомпилировать левую часть;
    скомпилировать правую часть;
  END;
   расставить операторы SJUMP(L); и об'единить
   совпадающие операторы левых частей предложений:
   вывести результат компиляции функции:
END;
```

Алгоритм об'единения совпадающих операторов левых частей предложений был описан в главе 3.

Основные принципы, на которых основан алгоритм компиляции правой части, также описаны в главе 3.

В данной главе описывается самая сложная часть компилятора - алгоритм компиляции левой части предложения.

4.2.КОМПИЛЯЦИЯ ЛЕВОЙ ЧАСТИ ПРЕДЛОЖЕНИЯ

В этом разделе описан алгоритм компиляции левой части предложения, включая исключение лишних удлинений значений VE-переменных с помощью операторов EOE(N);.

Правильность предлагаемого алгоритма вытекает из результатов, изложенных в главах 1 и 2.

В оставшейся части этого раздела будем, для краткости, обозначать словом "компилятор" только ту часть рефал-компилятора, которая компилирует левую часть предложения.

Исходными данными для алгоритма компиляции является левая часть рефал-предложения LP (без "=" и без имени функции). Результатом компиляции является некоторая последовательность операторов отождествления.

Алгоритм компиляции левой части порождает операторы отождествления строго в том порядке, в котором они должны стоять в результирующей программе.

В процессе работы компилятор постоянно поддерживает упорядоченный список дыр

A[1], A[2], ... , A[N]

При этом, для каждой дыры A[I] запоминается ее "левая граница" P[I] и "правая граница" Q[I]. P[I] - это номер правого конца элемента, непосредственно предшествующего A[I] в LP, а Q[I] - номер левого конца элемента, непосредственно следующего за A[I] в LP.

Поэтому список дыр хранится в компиляторе в виде кортежа

(H[1], H[2], ... , H[N]),

где H[I] = (P[I],A[I],Q[I]).

Перед началом компиляции кортеж дыр имеет следующий вид:

((1, LP, 2))

то есть он содержит только одну дыру - левую часть

предложения, границы которой соответствуют имени функции и знаку ">".

Во время компиляции дыры могут появляться и исчезать. Сопоставление пустой дыры или закрытого вхождения VE-переменной приводит к исчезновению дыры. Сопоставление пары скобок приводит к раздроблению дыры на две новых дыры.

В тот момент, когда кортеж дыр становится пуст, компиляция левой части предложения заканчивается.

Помимо списка дыр, компилятор поддерживает следующую информацию.

VD - множество переменных, уже принявших значение. Кроме того, для каждой переменной X, входящей в VD, запоминается целое число VQ[X] - номер правого конца главного вхождения переменной X.

NEL - номер первой свободной строки в таблице элементов.

Перед началом компиляции устанавливаются начальные значения VD=<Ø>, NEL=4.

Для оптимизации удлинений значений VE-переменных посредством операторов EOE(N); компилятор использует дополнительную информацию.

Прежде всего он следит за текущей глубиной стека переходов с помощью переменной JSP.

Перед началом работы устанавливается JSP=Ø. Затем, каждый раз, когда порождается оператор LE; или LESPC(L);, значение JSP величивается на единицу. При каждом порождении оператора EOE(N); значение JSP уменьшается на N.

Для генерации операторов EOE(N); используются два метода: разложение кортежа дыр на независимые классы (теорема 6.5 из главы 2) и выделение удлиняющегося независимого выражения (теорема 7.1 из главы 2).

Разложение кортежа дыр на независимые классы производится следующим образом.

Пусть в какой-то момент работы компилятора образовался следующий список дыр:

(H[1], H[2], ... , H[N])

где H[I] = (P[I], A[I], Q[I]).

Будем говорить, что две дыры H[I] и H[J] **непосредственно зависимы**, если либо I=J, либо множество

VARS[A[I]] * VARS[A[J]]

содержит хоть одну переменную, не входящую в VD.

Отношение непосредственной зависимости рефлексивно и симметрично. Теперь рассмотрим транзитивное замыкание этого отношения, которое будем называть отношением **зависимости**. А именно, будем говорить, что две дыры H' и H'' зависимы, если существует такая последовательность дыр

H[K1], H[K2], ... , H[KM] (M>=1)

что H'=H[K1], H''=H[KM] и любые две дыры, соседние в этой последовательности, непосредственно зависимы.

Отношение зависимости дыр является отношением эквивалентности, поэтому оно разбивает список дыр на J непересекающихся классов.

Разложение списка дыр на независимые классы можно получить с помощью следующего простого алгоритма.

Алгоритм разложения на непересекающиеся классы.

Исходные данные: кортеж дыр (H[1], H[2],..., H[N]) и множество переменных VD.

Результат работы: J - число классов и массив CLASS[1], ..., CLASS[N]. Для каждой дыры H[I] значением CLASS[I] является номер класса, к которому принадлежит H[I]. При этом 1 <= CLASS[I] <= J.

CL1. Установить J:=0; CLASS[I]:=0 для 1<=I<=N.

CL2. Если CLASS[I]¬=0 для всех 1<=I<=N, то закончить работу. Иначе найти такое I, что CLASS[I]=0. Установить X:=VARS[A[I]]; J:=J+1; CLASS[I]:=J.

CL3. Если X*VARS[A[I]] SUBSET VD для всех 1<=I<=N, таких, что CLASS[I]=0, то перейти к шагу CL2. Иначе, найти такое I, что CLASS[I]=0 и X*VARS[A[I]] содержит хоть одну переменную, не входящую в VD, установить CLASS[I]:=J; X:=X+VARS[A[I]] и повторить шаг CL3 .

Компилятор следующим образом использует разложение на независимые классы дыр.

Пусть в некоторый момент времени кортеж дыр принял следующий вид:

(H[1], H[2], ... , H[N]),

где дыры H[I] разбиваются на J независимых классов, где J>=2. Тогда компилятор строит J новых кортежей дыр

C[1], C[2], ... , C[J]

где C[K] получается из исходного кортежа дыр вычеркиванием тех дыр H[I], которые не принадлежат к K-му классу.

Пусть текущее значение JSP=P. Теперь компилятор временно "забывает" о существовании кортежей C[2], C[3],..., C[J] (отложив их в стек) и компилирует кортеж дыр C[1]. Пусть в тот момент, когда компиляция кортежа C[1] закончена, JSP=Q. Тогда, если Q>P, компилятор порождает оператор EOE(Q-P); и восстанавливает значение JSP, равное P. После этого компилятор точно так же поступает с кортежами дыр C[2], C[3],..., C[J].

П р и м е р.

Пусть NEL=8, VD=<0>, JSP=0, а кортеж дыр имеет вид

((6, E1 SX E2 ,7),(7, EA '+' EB ,5),(5, E3 SX E4 ,2))

Компилятор разбивает кортеж дыр на два кортежа:

C[1] = ((6, E1 SX E2 ,7),(5, E3 SX E4 ,2))
C[2] = ((7, EA '+' EB ,5))

Текущее значение JSP=Ø. Компилируется C[1]:

SB(6,7); PLE; LE; LS; CE; SB(5,2);
PLE; LE; LSD(10); CE;

Теперь JSP=2, поэтому порождается EOE(2); и восстанавливается JSP=Ø. Затем компилируется C[2]:

SB(7,5); PLE; LE; LSC('+'); CE;

Теперь JSP=1, поэтому порождается EOE(1); и восстанавливается JSP=Ø.

Выделение независимого удлиняющегося выражения происходит следующим образом.

Пусть в какой-то момент работы компилятора образовался кортеж дыр

(H[1], H[2], ... , H[N]) где H[1] = (P,A,Q).

Компилятор вычисляет множество VARSH переменных, входящих в дыры H[2], H[3],..., H[N] и пытается представить дыру A в виде

A = U(RX) AØ U(RY)* A1

где UX и UY - VE-переменные, AØ и A1 - типовые выражения, и при этом

(1) каждая VE-переменная, входящая в AØ на нулевом уровне скобок, принадлежит множеству <* UX *> + VD.

(2) VARS[U(RX)X AØ] * [VARS[AI] + VARSH] SUBSET VD.
(3) UX не входит ни в VD, ни в U(RX)X AØ, ни в AI, ни в VARSH.
(4) YES[ZTS U(RX)X AØ] - RY] = <Ø>.

Если это удалось, то компилятор выполняет следующие действия.

Пусть текущее значение JSP=JI. Компилятор временно прекращает сопоставление каких-либо элементов выражений A[1], A[2],..., A[N] за исключением элементов выражения U(RX)X AØ. Затем компиляция продолжается до тех пор, пока выражение U(RX)X AØ не будет полностью сопоставлено.

Пусть после этого JSP=J2. Тогда, если J2>JI, компилятор порождает EQE(J2-JI); и восстанавливает JSP=JI. После этого запрет на сопоставление оставшихся элементов из списка дыр отменяется и компиляция продолжается.

Теперь мы можем завершить описание переменных, которые нужны компилятору для порождения операторов EQE(N);.

HS - стек кортежей дыр, который используется для сохранения кортежей дыр, полученных в результате разложения списка дыр на независимые классы.

JSPS - стек для сохранения и восстановления значений переменной JSP.

HSP - переменная, которая содержит текущую глубину стеков HS и JSPS.

MV - переменная, которая используется, чтобы помечать переменную UY из независимых удлиняющихся выражений U(RX) AØ U(RY)Y. MV содержит множество отмеченных VE-переменных. Кроме того, для каждой переменной X, входящей в MV запоминается число JSPMV X - значение JSP в тот момент, когда X заносится в MV.

Теперь мы можем привести полный алгоритм компиляции левой части предложения.

Алгоритм компиляции левой части.

Исходные данные: LP - левая часть предложения.

Результат работы: последовательность операторов языка сборки.

(Нам будет удобно считать, что в LP все вхождения всех переменных имеют спецификатор, поскольку отсутствие спецификатора эквивалентно наличию универсального спецификатора W.)

CMPLP. (Начальные установки.) Установить HL:=((3,LP,2)); HSP:=Ø; NEL:=4; JSP:=Ø; VD:=<Ø>; MV:=<Ø>; B1:=3; B2:=2.

HSCH. (Искать дыру, из которой будет сопоставляться очередной элемент.) В этот момент HL=(H[1],H[2],...,H[N]), где N>=Ø. Установить I:=1.

HSCH1. Если I>N, то перейти к HSCH4, иначе рассмотреть дыру H[I]=(P,A,Q). Если A = U(R)Y A', где UY - VE-переменная и UY IN MV, то перейти к HSCH2, иначе перейти к HSCH3.

HSCH2. (Вытолкнуть дыры H[I], H[I+1],..., H[N] в стек HS.) Установить HSP:=HSP+1; HS[HSP]:=(H[I],H[I+1],...,H[N]); JSPS[HSP]:=JSPMV[UY]; VM:=MV - <* UY *>; HL:= (H[1],H[2],...,H[I-1]). Перейти к HSCH4.

HSCH3. Если A = U(RX)X A' U(RY)Y, где UX и UY - VE-переменные, не входящие в VD, то установить I:=I+1 и перейти к HSCH1. Иначе - перейти к ELMATCH.

HSCH4. В этот момент HL= (H[1],H[2],...,H[N]). Если N>Ø, то перейти к HLDECOMP. Если N=Ø и HSP>Ø, то породить ECE(JSP-JSPS[HSP]);, установить JSP:=JSPS[HSP]; HL:=HS[HSP]; HSP:=HSP-1 и перейти к HSCH.

Наконец, если N=Ø и HSP=Ø, то все элементы из LP сопоставлены и компиляция левой части предложения закончена.

ELMATCH. (Сопоставление элемента.) В этот момент H[I]=(P,A,Q).
Если P¬=B1 или Q¬=B2, то породить SB(P,Q);.
Если A=<>, то перейти к GNIL.
Если A = U(R)X, где UX – VE–переменная, не входящая в VD, то перейти к GCVE.
Если A = (A')A'', то перейти к GLB.
Если A = Z A', где Z – символ, то перейти к GLSC.
Если A = S(R)X A', то перейти к GLS.
Если A = W(R)X A', то перейти к GLW.
Если A = U(R)X A', где UX VE–переменная, входящая в VD, то перейти к GLVED.
Если A = A'(A''), то перейти к GRB.
Если A = A' Z, где Z – символ, то перейти к GRSC.
Если A = A' S(R)X, то перейти к GRS.
Если A = A' W(R)X, то перейти к GRW.
Если A = A' U(R)X, где UX – VE–переменная, входящая в VD, то перейти к GRVED.

(Теперь описывается сопоставление элементов всех видов, за исключением открытых VE–переменных.)

GNIL. Породить NIL;. Удалить H[I] из HL и перейти к HSCH.

GCVE. Породить CE;. Если R¬=W, то перейти к GLMAX. Если UX – V–переменная, то породить NNIL;. Удалить H[I] из HL и установить VD:= VD + <* UX *>; VQ[UX]:=NEL+1; NEL:=NEL+2. Перейти к HSCH.

GLB. Породить LB;. Заменить H[I] в HL на две дыры: (NEL,A',NEL+1), (NEL+1,A'',Q). Установить B1:=NEL; B2:=NEL+1; NEL:=NEL+2 и перейти к HSCH.

GLSC. Породить LSC(Z); и перейти к L1.

GLS. Если SX не входит в VD, то породить LS;, установить VD := VD + <* SX *>, VQ[SX]:=NEL. Иначе, породить LSD(VQ[SX]). Если R¬=W, то породить WSPC(R);. Перейти к L1.

GLW. Если WX не входит в VD, то породить LW;, установить VD := VD + <* WX *>, VQ[WX]:=NEL+1. Иначе породить LED(VQ[WX]);. Если R¬=W, то породить WSPC(R);. Перейти к L2.

GLVED. Породить LED(VQ[UX]);. Если R¬=W, то породить ESPC(R);. Перейти к L2.

GRB. Породить RB;. Заменить H[I] в HL на две дыры: (P,A',NEL), (NEL,A'',NEL+1) и установить B1:=P; B2:=NEL; NEL:=NEL+2. Перейти к HSCH.

GRSC. Породить RSC(Z);. Перейти к R1.

GRS. Если SX не входит в VD, то породить RS;, установить VD := VD + <* SX *>; VQ[SX]:=NEL. Иначе, породить RSD(VQ[SX]). Если R¬=W, то породить WSPC(R);. Перейти к R1.

GRW. Если WX не входит в VD, то породить RW;, установить VD := VD + <* WX *>; VQ[WX]:=NEL+1. Иначе породить RED(VQ[WX]);. Если R¬=W, то породить WSPC(R);. Перейти к P2.

GRVED. Породить RED(VQ[UX]);. Если R¬=W, то породить ESPC(R);. Перейти к P2.

L1. Заменить H[I] в HL на (NEL,A',Q), установить B1:=NEL; B2:=Q; NEL:=NEL+1. Перейти к HSCH.

L2. Заменить H[I] в HL на (NEL+1,A',Q), установить B1:=NEL+1; B2:=Q; NEL:=NEL+2. Перейти к HSCH.

R1. Заменить H[I] в HL на (P,A',NEL), установить B1:=P; B2:=NEL; NEL:=NEL+1. Перейти к HSCH.

R2. Заменить H[I] в HL на (P,A',NEL). Установить B1:=P; B2:= NEL; NEL:=NEL+2. Перейти к HSCH.

HLDECOMP. (Разложение списка дыр на независимые классы.)
В этот момент HL= (H[1],H[2],...,H[N]). Разбить дыры из HL на непересекающиеся классы по отношению зависимости. Пусть получилось J классов.
Если J=1, то перейти к FRSTHL.
Если J =2, построить J кортежей дыр C[1], C[2],...,C[J], где C[K] получается из HL удалением всех H[I], не входящих в K-й класс. Установить HS[HSP+I]:=C[I+1], JSPS[HSP+1]:=JSP для всех 1 <= I <= J. Затем установить JSP:=JSP+J-1; HL:=C[1].

FRSTHL. (Обработка первой дыры.)
В этот момент HL= (H[1], H[2],...,H[N]), H[1]= (P,A,Q).
Если B1¬=P или B2¬=Q, то породить SB(P,Q); и установить B1:=P; B2:=Q.

TH8P7. (Проверка условий теоремы 8.7 из главы 2.) Проверить, что

(1) A = E(R1)1 E(R2)2 ... E(RM)M U(R)X, где UX - VE-переменная и R¬=W.
(2) E1, E2,..., EM, EX - попарно различны и не входят ни в множество VD, ни в дыры H[2], H[3],..., H[N].

Если эти условия выполнены, то перейти к GRMAX.

TH8P3. (Проверка условий теоремы 8.3 из главы 2.) Проверить, что

(1) A = U(R)X A', где UX - VE-переменная и R¬=W.
(2) VES[R•ZTS[A']] = <Ø>.

Если эти условия выполнены, то перейти к GLMAX.

TH8P5. (Проверка условий теоремы 8.5 из главы 2.) Попытаться представить дыру A в виде

A = U(R)X A' AØ

где UX – VE–переменная, R¬=W и при этом

(1) A'¬=<> и хотя бы один из нуль–термов выражения A' не является вхождением E–переменной.
(2) YES[R*ZTS[A']] = <Ø>.

Если это удалось, то перейти к GLMAX.

TH8P4. (Проверка условий теоремы 8.4 из главы 2.) Проверить, что

(1) A = A' U(R)X, где UX – VE–переменная и R¬=W.
(2) YES[R*ZTS[A']] = <Ø>.

Если эти условия выполнены, то перейти к GRMAX.

TH8P6. (Проверка условий теоремы 8.6 из главы 2.) Попытаться представить дыру A в виде

A = AØ A' U(R)X

где UX – VE–переменная, R¬=W и при этом

(1) A'¬=<> и хотя бы один из нуль–термов выражения A' не является вхождением E–переменной.
(2) YES[R*ZTS[A']] = <Ø>.

Если это удалось, то перейти к GRMAX.

TH7P2. (Проверка условий теоремы 7.2 из главы 2.) Вычислить множество переменных, входящих в дыры H[2], H[3],..., H[N] и обозначить это множество через VARSH. Попытаться представить дыру A в виде

A = U(RX)X AØ U(RY)Y A1

где UX и UY – VE–переменные, AØ и A1 – типовые выражения, и при этом

(1) каждая VE-переменная, входящая в A0 на нулевом уровне скобок, принадлежит множеству <* UX *> + VD.
(2) VARS[U(RX)X A0] * [VARS[A1] + VARSH] SUBSET VD.
(3) UY не входит ни в VD, ни в U(RX)X A0, ни в A1, ни в VARSH.
(4) YES[ZTST[U(RX)X A0] - RY] =<0>.

Если это удалось, то установить MV:= MV + <* UY *>; JSPMV[UY]:=JSP.

GLVE. (Сопоставление открытого вхождения VE-переменной.) В этот момент

H[1] = (P, U(R)X A' , Q)

где UX - VE-переменная, не входящая в VD.

Если R¬=W, то перейти к GLVESPC.
Если UX - E-переменная, то породить PLE;. Иначе - породить PLV;. Породить LE;. Установить JSP:=JSP+1. Перейти к GLVEL2.

GLVESPC. Если UX - E-переменная, то породить PLESPC;. Иначе породить PLV;. Породить LESPC(P);. Установить JSP:=JSP+1. Перейти к GLVEL2.

GLMAX. (Набрать по-максимуму слева.) В этот момент

H[1] = (P, U(R)X A' , Q),

где UX - VE-переменная, не входящая в VD, и R¬=W.
Породить LMAX(R);. Если UX - V-переменная, то породить NNIL;. Перейти к GLVEL2.

GLVEL2. Установить VD := VD + <* UX *>, VQ[UX]:=NEL+1. Заменить H[1] в HL на (NEL+1,A',Q). Установить BI:=NEL+1; NEL:=NEL+2. Перейти к HSCH.

GRMAX. (Набрать по-максимуму справа.) В этот момент

H[1] = (P, A' U(R)X , Q),

где UX - VE-переменная, не входящая в VD, и R¬=W.

Породить RMAX(R);. Если UX - V-переменная, то породить NNIL;.

Установить VD := VD + <* UX *>; VQ[UX]:=NEL+1. Заменить H[1] в HL на (P,A',NEL). Установить B2:=NEL; NEL:=NEL+2. Перейти к HSCH.

4.3.КОМПИЛЯЦИЯ ПРАВОЙ ЧАСТИ ПРЕДЛОЖЕНИЯ

Основные принципы, на которых основан алгоритм компиляции правой части, описаны в главе 3. Однако, чтобы показать, как используется информация, накопленная при компиляции левой части для компиляции правой части, в этой главе приводится полное описание простейшего алгоритма компиляции правой части.

Правая часть компилируется в последовательность операторов языка сборки, преобразующих поле зрения. Как отмечалось в главе 3, компилятор может порождать операторы преобразования так, как если бы пересадки не планировались операторами трансплантации, а выполнялись немедленно.

Действия, которые во время исполнения рефал-программы производятся над значениями переменных, во время компиляции можно изображать как действия над переменными. С этой точки зрения, задача компилятора - породить правую часть предложения RP, используя куски левой части LP. Поэтому будем называть ту часть правой части, которая еще не порождена - ц е л ь ю.

В начале компиляции правой части цель совпадает с RP, а в конце компиляции - становится пустой.

Алгоритм компиляции правой части.

Исходные данные: левая часть предложения LP, правая часть предложения RP. Кроме того, для каждой переменной X, входящей в LP, - номер правого конца ее главного вхождения VQ[X], а для каждого вхождения X в LP - номер правого конца этого вхождения.

Результат работы: последовательность операторов преобразования.

REPL. Породить EOR;. Установить OBJ:=LP; TRG:=RP.

REPLE. Если TRG=<>, то породить EOS; и закончить работу.

Если TRG = Z A, где Z - символ, то породить NS(Z);, установить TRG:=A и перейти к REPLE.

Если TRG = (A, то породить BL;, установить TRG:=A и перейти к REPLE.

Если TRG =) A, то породить BR;, установить TRG:=A и перейти к REPLE.

Если TRG = < A, то породить BL;, установить TRG:=A и перейти к REPLE.

Если TRG = > A, то породить BRACT;, установить TRG:=A и перейти к REPLE.

Если TRG = SX A, то перейти к REPLSV.

Если TRG = WX A, то перейти к REPLWV.

Если TRG = VX A, то перейти к REPLVV.

Если TRG = EX A, то перейти к REPLEV.

REPLSV. Установить TRG:=A. Породить MULS(VQ[SX]);. Перейти к REPLE.

REPLWV. Установить TRG:=A. Если OBJ содержит хоть одно вхождение переменной WX и N - номер правого конца этого вхождения, то удалить это вхождение и породить TPLV(N);. Иначе породить MULE(VQ[WX]);. Перейти к REPLE.

REPLVV. Установить TRG:=A. Если OBJ содержит хоть одно вхождение переменной VI и N - номер правого конца этого вхождения, то удалить это вхождение и породить TPLV(N);. Иначе породить MALE(VQ[VI]);. Перейти к REPLE.

REPLEV. Установить TRG:=A. Если OBJ содержит хоть одно вхождение переменной EI и N - номер правого конца этого вхождения, то удалить это вхождение и породить TPLE(N);. Иначе породить MALE(VQ[EI]);. Перейти к REPLE.

Л И Т Е Р А Т У Р А

[БР 1977]
Базисный рефал и его реализация на вычислительных машинах. М., ЦНИПИАСС, 1977.

[ЗМСЭ 1974]
И.Б.Задыхайло, А.Н.Мямлин, В.К.Смирнов, Л.К.Эйсымонт. Об эффективной аппаратной реализации языка для описания об'ектов на уровне понятий и символьных преобразований. Искусственный интеллект. Итоги, перспективы. Семинар МДНТП им.Дзержинского, М.,1974, с.157-164.

[КПРТР 1975]
Ан.В.Климов, Л.В.Проворов, С.А.Романенко, Е.В.Травкина. Рефал в мониторной системе Дубна БЭСМ-6. Входной язык компилятора и запуск программ. Препринт ИПМ АН СССР N 8, М., 1975.

[КР 1975]
Ан.В.Климов, С.А.Романенко. Рефал в мониторной системе "Дубна" БЭСМ-6. Интерфейс рефала и фортрана. ИПМ АН СССР, М., 1975.

[КРТ 1972]
Ан.В.Климов, С.А.Романенко, В.Ф.Турчин. Компилятор с языка рефал. ИПМ АН СССР, М., 1972.

[КРТР 1974]
Ан.В.Климов, С.А.Романенко, Е.В.Травкина. Инструкция по работе с мониторной системой "Рефал" для БЭСМ-6. ИПМ АН СССР, М., 1974.

[Р2ОБФ 1986]

С.А.Романенко. Система программирования Рефал-2 для ЕС ЭВМ. Описание библиотеки функций. Препринт ИПМ им.М.В.Келдыша АН СССР N 200, М.,1986.

[Р2ОВЯ 1987]

С.А.Романенко. Система программирования Рефал-2 для ЕС ЭВМ. Описание входного языка. ИПМ им.М.В.Келдыша АН СССР, М. 1987.

[РКТ 1973]

С.А.Романенко, Ан.В.Климов, В.Ф.Турчин. Теоретические основы синтаксического отождествления в языке рефал. Препринт ИПМ АН СССР N 13, М., 1973.

[РТ 1970]

С.А.Романенко, В.Ф.Турчин. Рефал-компилятор. В сб. "Труды 2-й всесоюзной конференции по программированию". Новосибирск, 1970.

[Т 1966]

В.Ф.Турчин. Метаязык для формального описания алгоритмических языков. В сб. "Цифровая вычислительная техника и программирование". Сов. Радио, 1966, с.116-124.

[Т 1968]

В.Ф.Турчин. Метаалгоритмический язык. Кибернетика, N 4, 1968, с.45-54.

[Т 1974]

В.Ф.Турчин. Базисный рефал. Описание языка и основные приемы программирования. М.,ЦНИПИАСС, 1974.

[ТС 1969]

В.Ф.Турчин, В.И.Сердобольский. Язык рефал и его использование для преобразования алгебраических выражений. Кибернетика, N 3, 1969, с.58-62.

ПЕРЕЧЕНЬ ФУНКЦИЙ И ОБОЗНАЧЕНИЙ

<Ø> - пустое множество (II).

<* A1, A2, ... , AN *> - конечное множество, состоящее из элементов A1, A2,..., AN (II).

< > - пустой кортеж, пустой спецификатор, пустое выражение (II,I3,37).

<: R :> - спецификатор R, входящий в качестве элементарного спецификатора в другой спецификатор (33).

(P) - отрицание первичного спецификатора P (I3).

¬X - дополнение множества об'ектных термов до OTERM. Другими словами ¬X = OTERM-X (I6).

¬Q - отрицание спецификатора Q (I7).

U+V - об'единение множеств U и V (II).

R+Q - об'единение спецификаторов R и Q (25).

U*V - пересечение множеств U и V (II).

P*P' - пересечение первичных спецификаторов P и P' (22).

R*Q - пересечение спецификаторов R и Q (27).

U-V - разность множеств U и V (II).

R-Q - разность спецификаторов R и Q (29).

L#M - кортеж, составленный из пар выражений следующим образом (42). Пусть L и M - кортежи выражений,

имеющие одинаковую длину: L= (L1, L2,..., LN) и M= (M1, M2,..., MN). Тогда L#M= ((L1,M1), (L2,M2),..., (LN,MN)). В сокращенной записи L#M изображается в виде (L1,M1) (L2,M2) ... (LN,MN).

D'< [L] D'' - утверждение о том, что подстановка D' левее, чем подстановка D'' на кортеже L (43).

D'< [L#M] D'' - утверждение, равносильное D' < [L] D'' (43).

D'< =[L] D'' - утверждение о том, что либо D'< [L] D'', либо D' EQU[L] D'' (43).

D'< =[L#M] D'' - утверждение, равносильное D'< =[L] D'' (43).

B - первичный спецификатор, такой, что YES[B] состоит из всех термов вида (B), где B - об'ектное выражение (82).

CONTR[P,Q] - сужение спецификатора Q по первичному спецификатору P (23).

D' EQU[L] D'' - утверждение о том, что подстановка D' эквивалентна подстановке D'' на кортеже L (43).

D' EQU[L#M] D'' - утверждение, эквивалентное D' EQU[L] D'' (43).

X IN U - утверждение о том, что X принадлежит множеству U (11).

LMAX[T,B] - максимальный левый конец об'ектного выражения B, все нуль-термы которого принадлежат множеству об'ектных термов T (76).

NO[Q] - множество об'ектных термов, для которых

спецификатор Q говорит "нет" (16).

OTERM - множество всех об'ектных термов (12).

PLAIN[Q] - полный спецификатор, который не содержит конструкции вида <: P :> и для которого YES[PLAIN[Q]] = YES[Q] (34).

PSPEC - множество первичных спецификаторов (12).

RCG#[A,B] - самое левое отождествление из множества W#[A,B] (37,40).

RCG[G,L#M] - самое левое отождествление из множества W[G,L#M] (45).

RMAX[T,B] - максимальный правый конец об'ектного выражения B, все нуль-термы которого принадлежат множеству об'ектных термов T (76).

U SUBSET V - утверждение о том, что U является подмножеством V (U=V не исключается) (11).

SUBST[D,A] - результат применения подстановки D к типовому выражению A (39).

SUBST[D,L] - результат применения подстановки D к кортежу типовых выражений L (41).

THRU[Q] - множество термов, которые "проходят сквозь" спецификатор Q, т.е. для которых Q не говорит ни "да", ни "нет" (17).

UNDEF - особое значение "неопределено", которое вырабатывается функцией, в том случае, если применение ее к некоторому аргументу не имеет смысла (37).

VARS[A] - множество переменных, входящих в типовое

выражение A (38).

VARS[D] - множество переменных, входящих в подстановку D, т.е. множество таких переменных X, для которых существует об'ектное выражение B такое, что (X,B) IN D (38).

VARS[L] - множество переменных, входящих в кортеж типовых выражений L (42).

VARS[L#M] - множество переменных, входящих в кортеж из пар типовых и об'ектных выражений, т.е. множество VARS[L] (42).

W - первичный спецификатор, такой, что YES[W]=OTERM (2I).

W#[A,B] - множество таких отождествлений D об'ектного выражения B как типового выражения A, что VARS[D] = VARS[A] (37,40).

W[G,L#M] - множество таких отождествлений D кортежа об'ектных выражений M как кортежа типовых выражений L при ограничении G, что VARS[D] = VARS[G] + VARS[L] (42).

YES[P] - множество об'ектных термов, обозначаемое первичным спецификатором P (I2).

YES[Q] - множество об'ектных термов, для которых спецификатор говорит "да" (I6).

ZL[A] - нуль-длина типового выражения A, т.е. количество составляющих его нуль-термов (43).

ZT[A] - множество нуль-термов, входящих в об'ектные выражения, которые получаются из типового выражения A применением к нему всех подстановок, применимых к

A (68).

ZT[B] - множество нуль-термов, составляющих об'ектное выражение A (39).

ZTS[A] - для любого типового выражения A, ZTS[A] - полный спецификатор, такой, что ZT[A] SUBSET YES[ZTS[A]] (82).

П Р Е Д М Е Т Н Ы Й У К А З А Т Е Л Ь

Алгоритм
 - компиляции левой части I60
 - компиляции правой части I72
 - отождествления 38,46,55
 - распознавания пустоты значения спецификатора 33
Аргумент
 - оператора языка сборки 90

Вхождение переменной
 - главное 49
 - жесткое 49
 - закрытое 50
 - открытое 50
 - повторное 49
Выделение удлиняющегося выражения 68,I64
Выражение
 - максимальное 76,I33
 - удлиняющееся 68,I64

Дерево из левых частей II7
Дешифратор операторов языка сборки 9I
Дополнение
 - множества об'ектных термов I6
Дыра 4I,95

Звено списка 87,88

Имя поля 87,96
Интерпретатор языка сборки 86,9I

Компилятор
 - с рефала на язык сборки I59
Конкатенация
 - кортежей 42
 - спецификаторов I9

Копирование переменных 145
Кортеж 11
 - дыр 42,160
 - образов дыр 42

Метка
 - оператора языка звеньев 96
 - оператора языка сборки 90
Множество
 - нуль-термов 64,39,82
 - об'ектных термов 12
 - отождествлений
 - - кортежа об'ектных выражений как кортежа
       типовых выражений 42
 - - об'ектного выражения как типового 37,40
 - переменных входящих
 - - в кортеж пар выражений 42
 - - в кортеж типовых выражений 42
 - - в подстановку 38
 - - в типовое выражение 38

Номер
 - проекционный 49
 - элемента левой части предложения 95
Нуль-длина типового выражения 43

Об'единение спецификаторов 25
Образ дыры 41,95
Оператор
 - завершения и начала шага 149,150
 - завершения отождествления 139
 - копирования переменных 145
 - отождествления 93
 - пересадки 146
 - порождения
 - - символов 140
 - - структурных скобок 140
 - - функциональных скобок 143

- преобразования поля зрения I35
- языка звеньев 96
- языка сборки 90
Отношение
- линейного порядка на множестве
  отождествлений 43,44
- эквивалентности подстановок 43
Отождествление
- кортежа об'ектных выражений как кортежа
  типовых выражений 4I
- об'ектного выражения как типового выражения 36,40
- самое левое 37,40,45
Отрицание
- первичного спецификатора I3
- спецификатора I7

Пересадка кусков списка I46
Пересечение
- первичных спецификаторов 22
- спецификаторов 27
Переупорядочение дыр 64
Подстановка 38
- применимая к типовому выражению 39
- совместная 39
Поле звена 88
Порождение
- об'ектных выражений I40
- операторов преобразования I52
- функциональных скобок I43
Правило отождествления 45
Преобразование поля зрения I35
Примеры
- перевода функций на язык сборки I57
Пространство спецификаторов I2

Разложение
- кортежа дыр на независимые классы I6I
- первичного спецификатора 3I

Разность спецификаторов 29

Сопоставление
 - об'ектных выражений 98
 - открытых вхождений VE-переменных 113
 - переменных 105
 - - имеющих спецификатор 130
Спецификатор 13,126
 - как элемент другого спецификатора 33
 - первичный 12
 - полный 18
Список 87
 - свободных звеньев 136
Стек
 - переходов 113,117
 - трансплантаций 146
Строка таблицы элементов 93
Сужение спецификатора 23
Счетчик адреса виртуальный 93

Таблица элементов 93
Терм
 - об'ектный 12
Трансплантация 146

Управление в рефал-программе 91,117
Условия
 - полноты спецификатора 21
 - пустоты значения спецификатора 31

Цель 172

Шаг рефал-машины 149
 - завершение 149
 - начало 149

Элемент выражения 48
 - доступный для сопоставления 50

- жесткий 49
- однозначно сопоставимый 50

Язык
- звеньев 96
- сборки 86

ПЕРЕЧЕНЬ ОПЕРАТОРОВ ЯЗЫКА СБОРКИ

ACT(N) I44
BL I4I
BLF(L) I42
BLR I42
BR I4I
BRACT I44
CE I08
EOE(N) I20
EOEI I20
EOR I39
EOS I50
ESPC(L) I3I
FAIL II4
LB I0I
LBCE I08
LBNIL I03
LBY I02
LE II5
LEB I25
LED(N) I09
LESC(S) I24
LESD(N) I24
LESPC(L) I33
LMAX(L) I34
LS I05
LSC(S) 99
LSD(N) I06
LSCO(N) I04
LTXT(N,C1,C2,...,CN) I05
LW III
MULE(N) I45
MULS(N) I45
NIL 99
NNIL II2
NS(S) I40

NSO(N) I42
PLE II5
PLEB I25
PLESC I23
PLESPC I32
PLV II5
PLVB I25
PLVSC I23
PRE II6
PREB I25
PRESC I24
PRESPC I33
PRV II6
PRVB I25
PRVSC I24
RB I0I
RBCE I08
RBNIL I03
RBY I02
RE II6
REB I26
RED(N) II0
RESC(S) I24
RESD(N) I25
RESPC(L) I33
RMAX(L) I34
RS I06
RSC(S) I00
RSD(N) I07
RSCO(N) I04
RTXT(N,C1,C2,...,CN) I05
RW III
SB(N,M) I0I
SJUMP(L) II7
SPCB I27
SPCCLL(L) I27
SPCD I27
SPCF I27

SPCL I27
SPCN I27
SPCNG I27
SPCNGW I27
SPCO I27
SPCR I27
SPCS I27
SPCSC(S) I27
SPCW I27
SWAP I5I
TEXT(N,C1,C2,...,CN) I42
TPL(N,M) I47
TPLE(N) I48
TPLM(N,M) I47
TPLS(N) I48
TPLV(N) I48
WSPC(L) I3I

*** ТАБЛИЦА КОДОВ ОПЕРАТОРОВ ЯЗЫКА СБОРКИ ***

I	00	10	20	30	40	I
0	I	LS	PRE	REB	TEXT	I 0
1	I SJUMP	RS	PRV	EOE	BL	I 1
2	I FAIL	LW	RE	EOEI	BR	I 2
3	I SB	RW	PLESC	LSRCH	BLR	I 3
4	I LSC	LBNIL	PLVSC	PSRCH	BRACT	I 4
5	I RSC	RBNIL	LESC	WSPC	ACT	I 5
6	I LSCO	LBCE	PRESC	ESPC	MULS	I 6
7	I RSCO	RBCE	PRVSC	PLESPC	MULB	I 7
8	I LSD	NIL	RESC	LESPC	TPL	I 8
9	I RSD	CE	LESD	PRESPC	TPLM	I 9
A	I LTXT	LED	RESD	RESPC	TPLE	I A
B	I RTXT	RED	PLEB	LMAX	TPLV	I B
C	I LB	NNIL	PLVB	RMAX	TPLS	I C
D	I LBV	PLE	LEB	EOR	EOS	I D
E	I RB	PLV	PREB	NS	SWAP	I E
F	I RBV	LE	PRVB	NSO	BLF	I F
	I 00	10	20	30	40	I
	I					I

CLL	00	F	07
W	01	N	08
NG	02	R	09
NGW	03	O	0A
SC	04	D	0B
S	05	L	0C
B	06		

Романенко Сергей Анатольевич " Реализация Рефала – 2."
Редактор Штаркман В.С. Корректор Климов А.В.

Подписано к печати 12.05.87г. № Т-05144. Заказ № 216.
Формат бумаги 60X90 1/16. Тираж 280 экз.
Объём 8,2 уч.-изд.л. Цена 70 коп.

055 (02)2

Отпечатано на ротапринтах в Институте прикладной математики АН СССР
Москва, Миусская пл. 4.

Ссылки на издание рекомендуется делать по следующей форме: и.о., фамилия, название, препринт Ин. прикл. матем. им. М.В. Келдыша АН СССР, год, №.

Распространение: препринты института продаются в магазинах Академкниги г. Москвы, а также распространяются через Библиотеку АН СССР в порядке обмена.

Адрес: СССР, 125047, Москва-47, Миусская пл. 4, Институт прикладной математики им. М.В. Келдыша АН СССР, ОНТИ.

The references should be typed by the following form: initials, name, title, preprint, Inst.Appl.Mathem., the USSR Academy of Sciences, year, N(number).

Distribution. The preprints of the Keldysh Institute of Applied Mathematics, the USSR Academy of Sciences are sold in the bookstores "Academkniga", Moscow and are distributed by the USSR Academy of Sciences Library as an exchange.

Adress: USSR, I25047, Moscow A-47, Miusskaya Sq.4, the Keldysh Institute of Applied Mathematics, Ac.of Sc., the USSR, Information Bureau.

Цена 70 коп.