С.А. РОМАНЕНКО

МАШИННО-НЕЗАВИСИМЫЙ КОМПИЛЯТОР С ЯЗЫКА РЕКУРСИВНЫХ ФУНКЦИЙ

Д и с с е р т а ц и я
на соискание ученой степени
кандидата физико-математических наук
(Специальность 01.01.10 - математическое
обеспечение вычислительных машин и систем)

Научный руководитель:
кандидат физико-математических наук
В.С.Штаркман

Москва
1978г.

Оглавление

Введение **5**

1 Описание языка рефал **11**

1.1 Неформальное описание языка рефал . . . 11
1.1.1 Назначение языка . . . 11
1.1.2 Понятие конкретизации . . . 12
1.1.3 Знаки, символы, выражения . . . 13
1.1.4 Предложения . . . 15
1.1.5 Свободные переменные . . . 17
1.1.6 Рекурсивные функции . . . 19
1.1.7 Снятие неоднозначности при отождествлении . . . 20
1.1.8 Типы составных символов . . . 21
1.2 Формальное описание языка рефал . . . 22
1.2.1 Синтаксис . . . 22
1.2.2 Синтаксическое отождествление . . . 23
1.2.3 Рефал-машина . . . 23

2 Алгоритмы отождествления **26**

2.1 Алгоритмический подход к синтаксическому отождествлению . . 26
2.2 Использование различных правил отождествления в языке рефал 32
2.3 Преимущества и недостатки аксиоматического и алгоритмического подхода к отождествлению . . . 35
2.4 Оптимизация удлинений открытых вхождений е-переменных . . 37
2.5 Специфика синтаксического отождествления в языке рефал . . . 39

3 Язык сборки для базисного рефала **45**

3.1 Язык сборки как система команд . . . 45
3.2 Организация памяти . . . 46
3.3 Форматы операторов языка сборки . . . 50
3.4 Общая структура интерпретатора языка сборки. Эмулятор . . . 51
3.5 Таблица элементов . . . 53
3.6 Переменные `НЭЛ, Г1` и `Г2` . . . 53
3.7 Язык звеньев . . . 55

3.8 Отождествление объектных выражений 56
3.9 Проектирование свободных переменных 60
3.10 Проектирование открытых вхождений переменных выражения . 64
3.11 Передача управления с одного предложения на другое 67
3.12 Оператор `КУД,N;` 70
3.13 Операторы преобразования поля зрения 72
3.14 Роль переменной Г в процессе замены 73
3.15 Список свободной памяти 75
3.16 Порождение объектных выражений 76
3.17 Копирование свободных переменных 77
3.18 Перестановка участков объекта 78
3.19 Удаление участков объекта 80
3.20 Принудительное продвижение границы 80
3.21 Пример преобразования объекта в цель 80
3.22 Проблема пустых выражений 81
3.23 Представление конкретизационных скобок в поле зрения 83
3.24 Порождение конкретизационных скобок 85
3.25 Завершение очередного шага и подготовка следующего 87
3.26 Дополнительные операторы 87
3.27 Примеры перевода функций на язык сборки 88

4 Компилятор с рефала на язык сборки 90
4.1 Общая структура компилятора 90
4.2 Компиляция левой части предложения 90
4.3 Компиляция правой части предложения 97

5 Реализация интерпретатора языка сборки и компилятора с рефала 103
5.1 Основные этапы реализации 103
5.2 Первый вариант языка сборки и компилятора 104
5.3 Переработка языка сборки и создание оптимизирующего компилятора 105
5.4 Перенос компилятора на другие машины 106

Заключение 107

Литература 108

A Теоретический анализ правила отождествления 112
A.1 Строки и подстановки 112
A.2 Отождествление строк 113
A.3 Алгоритм отождествления 114
A.4 Монотонные предикаты $D[\Delta]$ 115
A.5 Отождествление векторов строк 117
A.6 Теоремы о разложении $W[\Delta,\overline{x}]$ 120
A.7 Правила отождествления 122

A.8 Упорядочение множества $W[\Delta, \overline{x}]$ 124
A.9 Левосторонние правила отождествления 129
A.10 Теоремы о разложении $L[\Delta, \overline{x}]$ 129
A.11 Независимые множества символов 131
A.12 Использование независимости множеств символов для разложения $L[\Delta, \overline{x}]$ 133
A.13 Отождествление левого конца строки 135
A.14 Удлиняющиеся строки 136
A.15 Строки, термы и выражения в языке рефал 138
A.16 Правильные подстановки в языке рефал 139
A.17 Жесткие термы и выражения 140
A.18 Независимые множества символов в языке рефал 143
A.19 Удлиняющиеся выражения 146

Введение

В первые годы своего существования ЭВМ применялись, главным образом, для численных задач, а затем - задач обработки больших массивов экономической информации. Для этих областей были созданы такие общепризнанные языки программирования как ФОРТРАН и КОБОЛ.

В 60-е годы ЭВМ быстро проникали в другие области, одна из которых известна под названием "обработка символьной информации".

До сих пор не существует точного и общепринятого определения того, что именно следует включить в понятие "обработка символьной информации", но как правило, предполагают, что к этой области относятся такие виды деятельности как перевод с одного искусственного или естественного языка на другой (трансляция), выполнение аналитических выкладок на ЭВМ, исследования в области искусственного интеллекта и др.

Для задач символьной обработки было создано большое количество разнообразных, резко отличающихся друг от друга, языков программирования, среди которых прежде всего следует отметить такие языки как C0MIT[4,5], LISP[6,7], SNOBOL[8,9,10], AMBIT[12,13], CONVERT [14], LIMP [16], РЕФАЛ [22,23,24,25,31,33,35,36].

Уже само обилие и разнообразие языков для символьной обработки и отсутствие среди них стандартного, удовлетворяющего подавляющее число пользователей, говорит о тех трудностях, с которыми сопряжены разработка и реализация таких языков.

Трудности эти, в принципе, возникают при создании любого языка высокого уровня и вытекают из самой сущности таких языков. С одной стороны, язык программирования, ориентированный на решение задач из какой-то определенной области, должен предоставлять такие средства записи алгоритмов, которые удобны и естественны для человека. Следовательно, он должен быть построен на основе формализации ряда понятий, терминов, представлений, важных и характерных для этой конкретной области. С другой стороны, алгоритмы, записанные на этом языке, должны достаточно эффективно исполняться на ЭВМ. Следовательно, язык должен как-то учитывать особенности существующих машин.

Таким образом, всякий язык высокого уровня имеет двойственную природу и представляет собой более или менее удачный компромисс между "человеческими" и "машинными" требованиями.

В области языков для символьной обработки эта двойственность проявляется особенно наглядно, поскольку существующие ЭВМ хорошо приспособлены для задач численного характера и для задач обработки данных и гораздо хуже – для задач обработки символьной информации.

Недостатки аппаратуры приходится преодолевать программными средствами. Вследствие этого, языки для символьной обработки часто реализуются с помощью интерпретаторов, и даже в тех случаях, когда создаются компиляторы, скомпилированная программа обычно не в состоянии работать самостоятельно, без поддержки значительной административной системы. Все это резко снижает скорость работы программ и усложняет реализации языков.

Пропасть, образовавшаяся между языками для символьной обработки и существующими ЭВМ хорошо отражена в следующем высказывании Д.Кнута [21, стр.563]:

"Появилось также несколько систем для манипуляций с цепочками (или строками); в этих системах первостепенное значение имеют операции над цепочками переменной длины, содержащими буквенную информацию (поиск вхождений некоторых подцепочек и т.д.). В историческом плане наиболее важными из такого рода систем были КОМИТ... и СНОБОЛ... Хотя системы для манипуляций с цепочками широко использовались и хотя они состоят в основном из алгоритмов, с которыми мы встречались в этой главе, они играют довольно скромную роль в истории методов представления информационных структур; пользователей этих систем почти не интересовали подробности фактических внутренних процессов, происходящих в машине".

Для выхода из сложившейся ситуации необходимо создание процессоров, специально приспособленных для символьной обработки. Однако, совершенно очевидно, что приступать к разработке такого процессора невозможно до тех пор, пока не выделен некоторый набор элементарных операций, хорошо отражающий специфику задач символьной обработки и допускающий относительно простую аппаратную реализацию.

При разработке этого набора операций опасно полагаться на одну лишь интуицию или удачу. Нам нужны какие-то гарантии того, что выбранный набор операций действительно адекватен задачам символьной обработки и достаточно прост, чтобы имело смысл реализовывать его аппаратно.

В связи с этим представляется разумным обратиться к существующим языкам для обработки символьной информации, так как эти языки вобрали в себя богатый опыт по программированию задач из этой области.

Предлагается следующий подход: выбрать в качестве отправного пункта некоторый язык высокого уровня и разработать систему команд, удобную для компилятора с этого языка. При этом, если исходный язык отражает специфику задач символьной обработки, можно надеяться, что и получившийся набор операций будет отражать существенные, важные особенности обработки символьной информации.

По убеждению автора, одним из языков, подходящих для этой цели, является язык рефал [22,23,25,31,35]. Это убеждение основано на следующих особенностях языка рефал.

1. Рефал построен на основе небольшого числа основных понятий “высокого уровня”: синтаксическое отождествление (поиск по образцу - pattern matching), подстановка, рекурсивная функция. Благодаря этому рефал сочетает компактность (формальное описание языка занимает 3-4 страницы) и выразительность. В то же время, понятия, на которых построен рефал - по существу традиционны и восходят к алгорифмам Маркова [2,3] и формализации рекурсивной арифметики по Эрбрану-Гёделю [1,3]. В рефале произведен удачный синтез этих понятий в единое целое.

2. Рефал резко отличается от языков, предназначенных для решения численных задач (Фортран) и задач обработки данных (кобол). В нем отсутствуют такие “классические” понятия как “переменная”, “метка”, “оператор присваивания”, “оператор перехода”, “условный оператор” и т.д. В этом отношении рефал заходит гораздо дальше,чем, например, LISP и SNOBOL ,в которых мы обнаруживаем метки, переменные, операторы присваивания и перехода (в языке LISP – благодаря механизму PROG). Между тем,чем меньше в языке заимствований из фортрана, тем нагляднее в нем проступают как раз те черты, которые специфически связаны с символьной обработкой.

3. Рефал - машинно-независимый язык. Первоначально он создавался с теоретическими целями и не предназначался для использования в качестве практического языка программирования. Поэтому в язык не были внесены ради облегчения реализации какие-либо черты, продиктованные особенностями имеющихся ЭВМ, а не назначением языка. Таким образом, рефал машинно-независим в силу того, что он не ориентируется на существующие ЭВМ, в то время как фортран машинно-независим в силу того, что отражает наиболее общие, характерные черты имеющихся ЭВМ.

4. Рефал прошел достаточно длительную проверку в качестве практического языка программирования. Он применялся как универсальный макропроцессор, для написания компиляторов, для производства аналитических выкладок на ЭВМ, лингвистических задач, машинного доказательства теорем и т.д. Поэтому, эффективная реализация рефала представляет не только теоретический, но и практический интерес.

Исходя из вышеизложенного, основная цель данной работы была сформулирована следующим образом: разработать язык, называемый в дальнейшем “язык сборки для рефала” (refal assembly language), удовлетворяющий следующим требованиям:

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

2. 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 микропрограммную или аппаратную реализацию.

3. 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-программы на язык сборки.

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

Очевидно, что задача создания языка сборки, удовлетворяющего перечисленным требованиям, может быть успешно решена только в том случае, если одновременно с ней будут решаться следующие задачи:

1. Разработка программной реализации языка сборки.

2. Разработка компилятора с рефала на язык сборки.

3. Разработка методов оптимизации рефал-программ.

В результате решения этих задач, мы попутно получим эффективную программную реализацию рефала, обладающую, по сравнению с реализациями на основе рефал-интерпретаторов [26,27], следующими достоинствами:

1. Упрощается логическая структура рефал-транслятора, поскольку он разбивается на две независимые части: интерпретатор языка сборки и компилятор с рефала на язык сборки.

2. Самую сложную и значительную по объему часть рефал-транслятора – компилятор, можно написать на рефале, который весьма удобен для этой цели,

3. Компилятор с рефала на язык сборки, написанный на рефале, машинно-независим, что значительно повышает портативность реализации рефала.

4. Скорость работы скомпилированных рефал-программ может быть повышена за счет оптимизаций, выполняемых во время компиляции.

Диссертационная работа состоит из введения, пяти глав, заключения, списка литературы и приложения.

В первой главе содержится описание языка рефал, на основе работ [22,23,24, 25,31,35,36] . Описание рефала включено в диссертацию вследствие того, что различные описания рефала используют различную терминологию, и по-разному определяют правило синтаксического отождествления. Терминология, введенная в первой главе, последовательно используется в последующих разделах диссертации.

Во второй главе рассматриваются вопросы, связанные с синтаксическим отождествлением в языке рефал.

Синтаксическое отождествление (поиск по образцу) является основным средством, с помощью которого в рефал-программах осуществляется доступ к данным, проверка условий и организация управления. Поэтому, скорость работы рефал-программ зависит, главным образом, от того, насколько эффективно будет осуществляться синтаксическое отождествление.

Непосредственная реализация правила отождествления, сформулированного в описании рефала, приводит к крайне неэффективному алгоритму отождествления. В связи с этим, в диссертации уделяется много внимания вопросам, связанным с ускорением процесса синтаксического отождествления.

Во второй главе рассматриваются различные формулировки правила отождествления, которые использовались в разных описаниях рефала. Сравниваются два способа определения правила отождествления: аксиоматический и алгоритмический. Обсуждаются достоинства и недостатки этих способов определения, а также методы оптимизации, позволяющие сократить комбинаторный перебор, возникающий в процессе синтаксического отождествления. Производится сравнение синтаксического отождествления в языке рефал с поиском по образцу в других языках.

В третьей 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 в третьей главе описаны неформально, с помощью примеров перевода рефал-предложений на язык сборки.

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

Алгоритм компиляции левой части содержит алгоритм исключения лишних удлинений е-переменных, благодаря чему уменьшается перебор во время синтаксического отождествления.

Алгоритм компиляции правой части использует эвристические методы, позволяющие уменьшить число преобразований, выполняемых над полем зрения во время замены ведущей области конкретизации.

В пятой главе описана история работы над рефал-компилятором и указан вклад участников этой работы.

Приложение содержит теоретический анализ правила отождествления, сформулированного в описании рефала. Доказываются свойства этого правила отождествления, на которых основан алгоритм компиляции левой части предложения, описанный в четвертой главе.

Основные результаты диссертации докладывались на Второй Всесоюзной Конференции по Программированию в г.Новосибирске в 1970 году [29] , а также на Всесоюзном Симпозиуме по вопросам обработки символьной информа-

ции в г.Тбилиси в 1970 году ([30]). По теме диссертации опубликованы работы [29,30,32,34,37,38,39].

Глава 1

Описание языка рефал

1.1 Неформальное описание языка рефал

1.1.1 Назначение языка

Язык рефал (алгоритмический язык рекурсивных функций) был создан в качестве абстрактного матаалгоритмического языка предназначенного для формализации семантики алгоритмических языков [22] . Различные описания языка [23,24,25,31,35] отличаются в некоторых деталях. Здесь будет описан вариант ре- фала соответствующий [35] , который получил название “базисный рефал”. Именно этот вариант был реализован в рефал-компиляторе.

Основные черты рефала определяются тем, что он был задуман как средство для описания семантики других языков.

Формальное описание семантики алгоритмического языка – это либо описание интерпретатора этого языка, либо описание компилятора с этого языка на другой язык (для которого уже имеется формальное описание семантики). Поэтому, язык, предназначенный для описания семантики – это язык, на котором можно описывать интерпретаторы и компиляторы [31]. Таким образом, метаалгоритмический язык сам должен быть алгоритмическим языком. Каким же требованиям должен удовлетворять этот язык?

Ясно, что, в принципе, на любом универсальном алгоритмическом языке можно описать семантику любого другого алгоритмического языка, поэтому выбор того или иного алгоритмического языка в качестве метаязыка – вопрос удобства. Однако, “неудобство”, практике, часто означает “невозможность”, поэтому, чтобы метаязык был пригоден для практического употребления, он должен удовлетворять следующим требованиям [31,35].

1. Метаязык предназначен для записи программ, которые обрабатывают тексты на каких-то формализованных языках. Поэтому он должен позволять описывать сколь угодно сложные преобразования одного текста в другой, без каких-либо ограничений на характер преобразования.

2. Метаязык должен быть удобен для человека. Текст на метаязыке должен представляться не в виде сложной и запутанной программы, которая каким то неведомым образом осуществляет трансляцию, а в виде предложений, выражающих понятия этого языка через более простые понятия.

3. Метаязык должен допускать эффективную реализацию на вычислительных машинах.

Сформулированные требования показывают, что метаязык представляет собой некоторый язык для обработки символьной информации (symbol manipulation language), поэтому,помимо описания семантики алгоритмических языков, он может иметь и другие, не менее важные применения. В первую очередь, это машинное выполнение громоздких аналитических выкладок в теоретической физике и прикладной математике, перевод с естественных языков, машинное доказательство теорем, моделирование целенаправленного поведения и т.п. Общим для всех этих применений является то, что мы заставляем машину совершать сложные преобразования над объектами, определенными в некоторых формализованных языках (алгоритмические языки, язык алгебры, язык исчисления предикатов и т.д.),

1.1.2 Понятие конкретизации

Каким же должен быть метаалгоритмический язык, чтобы он был удобен для человека?

Машинно-независимые алгоритмические языки (фортран, кобол, ПЛ/1, симула и т.д.) удобны для записи задач из определенной области вследствие того, что они строятся на основе формализации ряда понятий, важных и характерных для данной специальной области. Нам же нужен язык, который не просто фиксирует определенный набор понятий, а позволяет описывать любые языки и понятия (метаязык). Такой метаязык будет удобен для человека только в том случае, если он будет отражать какие-то чрезвычайно общие, и, в то же время, важные особенности естественных языков и их продолжения – формализованных языков математики [23, 31, 35].

Важнейшей чертой этих языков является наличие в них иерархии понятий. Возьмем какой-нибудь языковый объект, например, слово и зададим вопрос: что значит понимать это слово. Очевидно, что внешний вид этого слова сам по себе не имеет никакого значения, имеют значение лишь связи этого слова с другими словами и, в конечном счете, с элементами чувственного опыта. Поэтому понимать слово – значит уметь пройти в обратном направлении путь его построения. Понимать абстрактное понятие – значит уметь его конкретизировать в каждой заданной ситуации, понимать сложное понятие – значит уметь свести его к ряду более простых. И то, и другое означает замену языкового объекта, занимающего более высокое положение в иерархии понятий на ряд объектов, занимающих в ней более низкое положение. Эту операцию мы будем называть конкретизацией языкового объекта [31, 35].

Операция конкретизации является основной и единственной операцией в языке рефал. Для обозначения этой операции будут использоваться два знака: k и $\perp$, или функциональные скобки. Знак k мы будем также называть "знаком конкретизации", а знак $\perp$ – "конкретизационной точкой".

Языковый объект,подлежащий конкретизации, должен быть заключен в конкретизационные скобки. Так, например, объект $k\ 28+7\perp$ будет рано или поздно заменен на объект 35 (если, конечно + обозначает операцию сложения).

Очевидно, что явное введение конкретизационных скобок непосредственно вытекает из назначения языка рефал. В специализированных алгоритмических языках, опирающихся на фиксированную иерархию понятий, обычно придумывают такую систему обозначений, чтобы для каждого языкового объекта по его внешнему виду (и по контексту) можно было распознать его положение в иерархии понятий. Так, в фортране, объект 2.7183 обозначает число, а объект $X0$ обозначает переменную, которая может принять (в числе других) значение 2.7183. В нашем же случае, когда мы проектируем метаязык, рассчитанный на произвольные системы понятий и обозначений, нужно с помощью специальных знаков указывать, какие объекты уже полностью вычислены, а какие подлежат конкретизации.

1.1.3 Знаки, символы, выражения

Язык рефал является одномерным знаковым языком, т.е. его объекты – это последовательности некоторых знаков. Под знаком мы понимаем минимальную синтаксическую единицу, не расчленимую на составные части.

Так как язык рефал является метаязыком и должен применяться для описания других языков, не следует сильно ограничивать набор знаков, которые будут в нем использоваться. В то же время нам, очевидно, потребуется какое-то число специфических знаков с фиксированными значениями. Поэтому мы объявим, что все знаки языка делятся на собственные знаки и объектные знаки. Набор объектных знаков не фиксируется, предполагается только, что он конечен. Что касается собственных знаков, мы будем вводить их по мере дальнейшего изложения. Пока что нам известно два собственных знака: k и $\perp$.

Минимальной семантической единицей, нерасчленимой на составные части в процессе работы рефал-программы, является символ, который представляет из себя либо объектный знак, либо последовательность объектных знаков, заключенную в апострофы.Например:

```
Z
=
'ЕСЛИ'
'ФИ25'
```

Знак '(апостроф) является собственным знаком рефала.

Синтаксические средства, применяемые в формализованных языках для конструирования сложных объектов из простых, могут быть самыми разнообразными. Поэтому, мы не можем предполагать для объектов рефала какой-то

определенный, специализированный синтаксис. Тем не менее, есть одна черта, общая для всех синтаксических схем, которую нельзя не учесть при построении метаязыка: это тот факт, что синтаксический анализ всегда порождает определенное дерево языковых объектов (см.рис.1.1). Простейший и наиболее привычный способ изображения такой структуры в виде линейной последовательности знаков – это использование скобок. Поэтому скобки рассматриваются в рефале, как собственные знаки (следовательно – не являются символами).

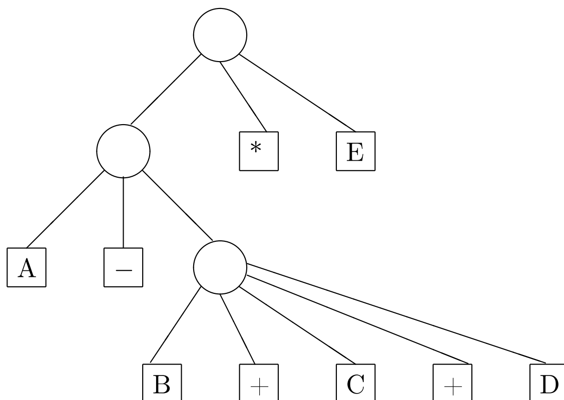


Рис. 1.1: Дерево, соответствующее структуре алгебраического выражения $(A - (B + C + D)) * E$

Круглые скобки, придающие структуру объектам работы, мы будем называть структурными, в отличие от конкретизационных скобок, указывающих на необходимость выполнения конкретизации.

Теперь мы введем понятие выражения, которое можно определить как последовательность символов и скобок, соответствующую следующему определению.

1. Пустой объект есть выражение.

2. Символ есть выражение. Свободная переменная есть выражение (о свободных переменных см. п.2.1.5).

3. Последовательность выражений есть выражение.

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

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

Примеры выражений:

ABC()
A = (ABC)/2
k 'ЕСЛИ' A $<$ B 'ТО' A 'ИНАЧЕ' B $\perp$
$k\ \ k\, X \perp (\, k\, Y \perp)\perp$

Терм определяется как выражение, которое представляет из себя либо символ, либо выражение, заключенное в структурные или конкретизационные скобки.

Таким образом, всякое выражение есть последовательность из некоторого (быть может нулевого) числа термов.

Информация, обрабатываемая рефал-программой, всегда представляет из себя некоторое выражение.

1.1.4 Предложения

Алгоритмические языки можно разделить на две группы. Первую группу образуют языки, которые мы назовем языками операторного типа. Элементарными единицами программы являются здесь операторы, то есть приказы, выполнение которых сводится к четко определенному изменению четко определенной части памяти машины. Фортран, например, является типичным операторным языком. Языки второй группы мы назовем языками сентенциального типа. В сентенциальных языках программа представляется в виде набора предложений (соотношений, формул, правил) порядок применения которых не совпадает, вообще говоря, с порядком их расположения. Машина, выполняющая алгоритм, должна не только применять предложения, но также и определяясь, какое именно предложение должно использоваться на каждом конкретном шаге. Примером сентенциального языка, созданного с теоретическими целями, может служить язык нормальных алгоритмов А.А. Маркова[2,3]

Язык рефал является сентенциальным в своей основе и по своей природе, ибо он был задуман как язык для описания связей, соотношений между понятиями.

Программа на языке рефал представляет собой набор предложений (правил конкретизации) и имеет следующий вид:

$\S\, \mathcal{K}_1\ \ k\, \mathcal{L}_1\ \sim\ \mathcal{R}_1$
$\S\, \mathcal{K}_2\ \ k\, \mathcal{L}_2\ \sim\ \mathcal{R}_2$
$\dots$
$\S\, \mathcal{K}_n\ \ k\, \mathcal{L}_n\ \sim\ \mathcal{R}_n$

Собственный знак § (параграф) служат для отделения предложений друг от друга.

Каждое правило конкретизации

$$\S\, \mathcal{K}_i\ \ k\, \mathcal{L}_i\ \sim\ \mathcal{R}_i$$

является указанием для замены одного языкового объекта на другой, поэтому оно состоит из левой части $\mathcal{L}_i$ (заменяемый объект) и правой части $\mathcal{R}_i$ (объект,

заменяющий левую часть). Синтаксически $\mathcal{L}_i$ и $\mathcal{R}_i$. являются выражениями. Для разделения левой и правой части используется собственный знак $\sim$ (тильда). $\mathcal{K}_i$ – комментарий, который обычно используется для нумерации предложений и является произвольной последовательностью объектных знаков.

Выполнение рефал-программы производится с помощью абстрактной рефал-машины. Рефал-машина имеет два запоминающих устройства: поле памяти и поле зрения.

Перед началом работы в поле памяти заносится набор предложений, а в поле зрения – выражение, подлежащее конкретизации.

Пусть, например, в поле памяти находится единственное предложение

$$\S 1.1\ k\ x\ \sim\ 137$$

а в поле зрения – выражение

$$k\ x \perp$$

Тогда рефал-машина заменит содержимое поля зрения на выражение

$$137$$

и остановится, поскольку в поле зрения не осталось ни одного знака конкретизации.

А что будет, если занести в поле памяти два предложения:

$$\S 1.1\ \ k\ x\ \sim\ 137$$
$$\S 1.2\ \ k\ x\ \sim\ 274$$

Теперь для выполнения конкретизации $k\ x \perp$ пригодно не одно, а два предложения. Неоднозначность устраняется следующим образом. Рефал-машина просматривает предложения в том порядке, в котором они стоят в поле памяти и применяет первое из них, которое окажется подходящим.

Поле зрения рефал-машины может содержать сколько угодно конкретизационных скобок, которые могут быть как угодно вложены друг в друга. Поэтому, нужно до говориться, каким образом рефал-машина будет выбирать выражение, с которого надо начинать процесс конкретизации.

Назовем областью действия знака k выражение, заключенное между этим знаком k и парной к нему конкретизационной точной $\perp$. Ведущим назовем самый левый из таких знаков k , в области действия которых нет других знаков k . Теперь мы можем описать работу рефал-машины.

Работа рефал-машины разбивается на шаги. В начале каждого шага рефал-машина находит ведущий знак k и соответствующую ему область действия (ведущую область конкретизации). Затем, рефал-машина сравнивает ведущую область конкретизации с левыми частями предложений, стоящих в поле памяти. Найдя подходящее предложение она делает соответствующую замену в поле зрения и переходит к исполнению следующего шага.

Пусть, например, поле памяти содержит набор предложений:

§1.1 $k\,X\ \sim\ 137$
§1.2 $k\,X\ \sim\ 274$
§1.3 $k\,Y\ \sim\ 2$
§1.4 $k\,137+2\ \sim\ 139$

а поле зрения содержит выражение:

$k\ \ k\,X\perp\ +\ k\,Y\perp\ \perp$

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

$k\,137+\ \ k\,Y\perp\ \perp$

Теперь подлежит конкретизации выражение $k\,Y\perp$. Применяется третье предложение. Поле зрения принимает вид:

$k\,137+2\perp$

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

139

которое уже не содержит знаков k.

1.1.5 Свободные переменные

Во всех рассмотренных до сих пор примерах, предложения, которые оказывались применимыми для выполнения конкретизации, имели левые части, в точности совпадавшие с конкретизируемым выражением. Чтобы заставить рефал-машину выполнить сложение $137+2 = 139$, пришлось в поле памяти занести особое предложение. Ясно, что мы так далеко не уйдем. Нужно уметь записывать предложения, применимые более чем к одному конкретизируемому выражению. Для этого нужно ввести в предложения свободные переменные, которые при различных применениях предложения могут принимать различные значения.

Языковые объекты, с которыми имеет дело рефал-машина, это всегда выражения, которые могут быть, в частности, термами и символами. Поэтому мы будем использовать переменные трех типов, соответственно синтаксическому типу тех объектов, которые могут быть их значениями. Для изображения свободных переменных служат три собственных знака - признака типа свободных переменных: e – признак выражения, t – признак терма и s – признак символа. Свободная переменная изображается парой знаков, состоящей из признака типа и следующего за ним произвольного объектного знака - индекса свободной переменной.

Так, $e1$, $e2$, eA - свободные переменные выражения (e-переменные). e-переменная может принять в качестве значения любое выражение.

$t1$, $t2$, tA – свободные переменные терма (t-переменные). Значением t-переменной может быть любой терм, например, A или (AB), но никак не AB.

$s1$, $s2$, sA – свободные переменные символа (s-переменные). Значением s-переменной может быть только символ.

В дальнейшем, для большей компактности записи, мы будем использовать следующее сокращение: индекс переменной будем помещать в нижнюю позицию, одновременно заменяя заглавную букву на строчную. Например, вместо eA, $t1$, sX будем писать e_a, t_1, s_x.

Разрешив использовать в левых и правых частях предложений свободные переменные, мы получаем мощное изобразительное средство. Теперь, чтобы решить, применимо ли данное предложение к конкретизируемому выражению, рефал-машина должна определить, можно ли вместо переменных в левой части предложения подставить такие значения, чтобы левая часть совпала с конкретизируемым выражением. При этом, разные вхождения одной и той же переменной должны заменяться на одно и то же значение. Это действие рефал-машины мы будем называть синтаксическим отождествлением.

В случае успешного синтаксического отождествления, ведущая область конкретизации заменяется на правую часть соответствующего предложения, в которую вместо свободных переменных подставлены значения, которые получили эти переменные в результате синтаксического отождествления. В правой части предложения разрешается использовать только такие переменные, которые входят в левую часть. Если же синтаксическое отождествление невозможно, предложение неприменимо.

Рассмотрим несколько примеров. Допустим, что мы хотим описать понятие "первый символ выражения". Это значит, что мы хотим, чтобы, например, результатом выполнения конкретизации

k'ПЕРВСИМ' Z(AB) + F $\perp$

являлся символ Z , а конкретизации

k'ПЕРВСИМ' + $\perp$

символ + и т.д.

Для этого достаточно ввести в поле памяти рефал-машины предложение

§2.1 k'ПЕРВСИМ' $s_a\, e_x \;\sim\; s_a$

Аналогично можно описать понятие "последний символ выражения" с помощью предложения

§3.1 k'ПОСЛСИМ' $e_x\, s_a \;\sim\; s_a$

1.1.6 Рекурсивные функции

Для того, чтобы правильно описать на рефале какой-либо алгоритм, нужно совершенно ясно представлять, как будет работать рефал-машина с некоторым набором предложений в поле памяти. Это ставит человека в трудное положение, ибо, имитируя работу рефал-машины, он должен на каждом шаге просматривать все предложения, стоящие в поле памяти. Следовательно, необходим какой-то способ, позволяющий на каждом шаге исключить из рассмотрения все лишние предложения, заведомо не имеющие отношения к данному акту конкретизации. Надо как-то разбить предложения на небольшие, независимые группы.

Мы потребуем, чтобы в левой части каждого предложения непосредственно за знаком k следовал какой-то символ. Этот символ мы будем называть детерминативом предложения.

Тем самым, все предложения разбиваются на группы, каждая из которых состоит из предложений, имеющих один и тот же детерминатив.

Если два предложения имеют различные детерминативы, они не могут оказаться применимыми одновременно. Поэтому порядок расположения в памяти групп предложений с различными детерминативами несущественен.

Пусть у нас есть группа с детерминативом $\mathcal{F}$. Тогда она определяет функцию, определенную на некотором множестве выражений и принимающую значения из множества выражений.

Действительно, занесем в поле памяти эту группу предложений, а в поле зрения - выражение

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

где: $\mathcal{E}$ - некоторое выражение. Запустим рефал-машину. Если она придет в состояние нормальной,остановки, то будем считать, что значением функции $\mathcal{F}$ на выражении $\mathcal{E}$ является то выражение, которое осталось в поле зрения. В противном случае будем считать, что значение $\mathcal{F}$ для выражения $\mathcal{E}$ не определено.

Таким образом, обычной в математике функциональной записи $\mathcal{F}(\mathcal{E})$ соответствует в языке рефал запись $k\;\mathcal{F}\;\mathcal{E}\;\perp$. Это различие вызвано необходимостью отличать конкретизационные скобки от обычных, структурных скобок.

Детерминатив $\mathcal{F}$ в дальнейшем мы будем называть именем функции.

Теперь мы можем объяснить, почему рефал является языком рекурсивных функций. Для этого рассмотрим следующий пример.

Опишем на рефале функцию `'REV'`, которая определена для всех выражений. Значением этой функции является "перевернутое" исходное выражение. Так, после выполнения конкретизации

k`'REV'` A(B(CD)F) $\perp$

в поле зрения останется выражение

(F(DC)B)A

функция 'REV' описывается тремя предложениями:

$\S 4.1\ k\texttt{'REV'}\ e_1\ s_x\ \sim\ s_x\ k\texttt{'REV'}\ e_1\ \perp$
$\S 4.2\ k\texttt{'REV'}\ e_1(e_x)\ \sim\ (\ k\texttt{'REV'}\ e_x)\ k\texttt{'REV'}\ e_1\ \perp$
$\S 4.3\ k\texttt{'REV'}\ \ \sim$

Видно, что в правых частях предложений стоят обращения к самой же функции 'REV'.

Другой пример. Функция 'SYMM' принимает значение T для симметричных выражений (которые не изменяются после применения к ним функции 'REV') и значение F – для всех остальных.

$\S 5.1\ \ k\texttt{'SYMM'}\ \ \sim\ \ \mathrm{T}$
$\S 5.2\ \ k\texttt{'SYMM'}\ \ s_x\ \sim\ \ \mathrm{T}$
$\S 5.3\ \ k\texttt{'SYMM'}\ \ s_x\, e_a\, s_x\ \sim\ \ k\texttt{'SYMM'}\ \ e_a\ \perp$
$\S 5.4\ \ k\texttt{'SYMM'}\ \ (e_a)\ \sim\ \ k\texttt{'SYMM'}\ \ e_a\ \perp$
$\S 5.5\ \ k\texttt{'SYMM'}\ \ (\,)\, e_a\, (\,)\ \sim\ \ k\texttt{'SYMM'}\ \ e_a\ \perp$
$\S 5.6\ \ k\texttt{'SYMM'}\ \ (t_x\, e_1)\, e_a\, (e_2\, t_y)\ \sim\ \ k\texttt{'SYMM'}\ \ t_x(e_1)\, e_a\, (e_2)t_y\ \perp$
$\S 5.7\ \ k\texttt{'SYMM'}\ \ e_a\ \sim\ \mathrm{F}$

1.1.7 Снятие неоднозначности при отождествлении

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

$\S 6.1\ \ k\texttt{'F'}\ e_1\ ;\ e_2\ \sim\ \ k\texttt{'G'}\ e_1\ \perp\ k\texttt{'G'}\ e_2\ \perp$

а в поле зрения - выражение

k A1:=A2;BI:=B2; 'GOTO'L $\perp$

Оно может быть отождествлено с левой частью таким образом, что переменные e_1 и e_2 примут следующие значения;

$e_1 \leftarrow$ A1:=A2
$e_2 \leftarrow$ B1: = B2;'GOTO'L

Однако, возможен и второй вариант отождествления, при котором переменные примут такие значения:

$e_1 \leftarrow$ A1:=A2;B1=B2
$e_2 \leftarrow$ 'GOTO'L

Следовательно, необходимо договориться, как поступает рефал-машина при наличии такой неоднозначности. Мы примем следующее соглашение: рефал-машина выбирает тот вариант отождествления, при котором первая слева е-переменная принимает наиболее короткое значение, а если это не устраняет

неоднозначности, то такой же отбор производится по значению второй e-переменной, затем третьей и т.д. В нашем примере будет выбран первый вариант отождествления.

Принятое соглашение широко используется при программировании на рефале. В качестве примера рассмотрим функцию 'MSET'.

$$\S 7.1\ \ k\texttt{'MSET'}\ e_1\, t_x\, e_2\, t_x\, e_3\ \sim\ e_1\, k\texttt{'MSET'}\ e_2\, t_x\, e_3 \perp$$
$$\S 7.2\ \ k\texttt{'MSET'}\ e_1\ \sim\ e_1$$

Эта функция просматривает выражение слева направо терм за термом. Для очередного терма проверяется, не стоит ли справа от него точно такой же терм. Если да, то очередной терм вычеркивается, в противном случае – оставляется. Оставшееся выражение обладает тем свойством, что составляющие его термы попарно различны. Например, результатом выполнения конкретизации

$$k\texttt{'MSET'}\ \text{AAACBDBEAAF} \perp$$

будет выражение

CDBEAF

Можно описать функцию `'MSET'` так, чтобы при отождествлении не возникло неоднозначностей:

$$\S 8.1\ \ k\texttt{'MSET'}\ \sim$$
$$\S 8.2\ \ k\texttt{'MSET'}\ t_a\, e_1\ \sim\ k\texttt{'MSET1'}\ t_a\,(\,)\, e_1 \perp$$
$$\S 8.3\ \ k\texttt{'MSET1'}\ t_a\,(e_1)\, t_a\, e_2\ \sim\ k\texttt{'MSET'}\ e_1\, t_a\, e_2 \perp$$
$$\S 8.4\ \ k\texttt{'MSET1'}\ t_a\,(e_1)\, t_b\, e_2\ \sim\ k\texttt{'MSET1'}\ t_a\,(e_1\, t_b)\, e_2 \perp$$
$$\S 8.5\ \ k\texttt{'MSET1'}\ t_a\,(e_1)\ \sim\ t_a\, k\texttt{'MSET'}\ e_1 \perp$$

Видно, что первое описание короче и нагляднее, чем второе. Кроме того, во втором описании пришлось использовать вспомогательную функцию `'MSET1'`.

1.1.8 Типы составных символов

В формальном описании рефала составной символ определяется как последовательность объектных знаков, заключенная в апострофы. Эта последовательность объектных знаков называется “телом составного символа”.

Множество составных символов можно разбить на непересекающиеся классы, в соответствии с тем, какой вид имеют их тела. В дальнейшем мы будем рассматривать символы-метки и символы-числа.

Символами-метками мы будем называть составные символы, тело которых является идентификатором. Например:

'ALPHA' 'L2A4'.

Символами-числами будем называть составные символы, тело которых представляет собой целое неотрицательное число, т.е. последовательность десятичных цифр. Например:

'0' '1' '512' '23'.

В дальнейшем, мы будем считать, что в качестве детерминативов всех предложений допускаются только символы-метки. Таким образом, символы-метки будут служить в качестве имен функций.

Символы-числа будут служить для изображения чисел.

Различные реализации рефала накладывают свои ограничения на допустимый вид символов-меток и символов-чисел, а также могут допускать составные символы других типов [37, 38, 39, 40, 41]. В частности, накладывается ограничение на максимальную величину символов-чисел.

1.2 Формальное описание языка рефал

1.2.1 Синтаксис

Значительную часть синтаксиса мы опишем в бэкусовской нормальной форме.

1. Знаки.

   <знак>::=<собственный знак>|<объектный знак>
   <собственный знак>::= § | ' | ~ |<признак типа переменной>
   | <скобка>
   <скобка>::=<структурная скобка> | <конкретизационная скобка>
   <структурная скобка>::= (|)
   <конкретизационная скобка>::= k | ⊥
   <признак типа переменной>::= s | t | e

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

2. Символы и выражения.

   <символ>::=<объектный знак>|<составной символ >
   <составной символ>::='<тело составного символа>'
   <тело составного символа>::=<объектная цепочка>
   <объектная цепочка>::=<объектный знак>
   | <объектная цепочка> <объектный знак >
   <выражение>::=<пусто>| <терм> <выражение>
   <пусто>::=
   <терм>::= <символ>
   |<свободная переменная>
   | (<выражение>) | k <выражение> ⊥
   <свободная переменная>::=
   <признак типа переменной> <индекс>
   <индекс>::=<объектный знак>

   Выражение, которое не содержит знаков k и ⊥ называется типовым выражением. Выражение, которое не содержит свободных переменных, назы-

вается рабочим выражением. Выражение, которое не содержит ни знаков k , и $\perp$, ни свободных переменных, называется объектным выражением.

3. Предложения.

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

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

   <левая часть>::= k < типовое выражение > $\sim$

   <правая часть>::=<выражение>

   В правую часть предложения могут входить только те переменные, которые входят в левую часть.

4. Набор предложений.

   <набор предложений>::=<предложение>
   | <предложение> <набор предложений>

1.2.2 Синтаксическое отождествление

1. Говорят, что объектное выражение $\mathcal{E}$ может быть отождествлено как типовое выражение $\mathcal{L}$, если свободные переменные в $\mathcal{L}$ могут быть заменены с соблюдением указанных ниже правил на такие выражения, называемые их значениями, что $\mathcal{E}$ совпадает с $\mathcal{L}$. Правила таковые:

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

   1.2. Все вхождения одной и той же переменной заменяются на одно и то же значение.

2. Если существует несколько вариантов придания значений свободным переменным, приводящих к совпадению $\mathcal{L}$ с $\mathcal{E}$, то эта неоднозначность снимается следующим способом. Из всех вариантов отождествления выбирается тот, при котором самая левая свободная переменная выражения в $\mathcal{L}$ принимает наикратчайшее значение. Если это не устраняет неоднозначности, то такой же отбор производится по второй слева переменной и т.д.

3. Под отождествлением терма $k\,\mathcal{E} \perp$ как левой части предложения $k\,\mathcal{L} \sim$ мы понимаем отождествление $\mathcal{E}$ как $\mathcal{L}$.

1.2.3 Рефал-машина

Рефал-машина - это вычислительное устройство, состоящее из двух потенциально бесконечных запоминающих устройств: поля памяти и поля зрения, а также - процессора, преобразующего содержимое поля зрения.

В каждый момент времени поле памяти содержит некоторый набор предложений, а поле зрения – некоторое рабочее выражение.

Областью действия знака k называется выражение, начинающееся непосредственно за этим знаком k , и кончающееся перед парной ему конкретизационной точкой. Ведущим знаком k, в некотором выражении называется самый левый знак k , в области действия которого не содержится ни одного знака k . Терм, начинающийся с ведущего знака k , называется конкретизируемым термом.

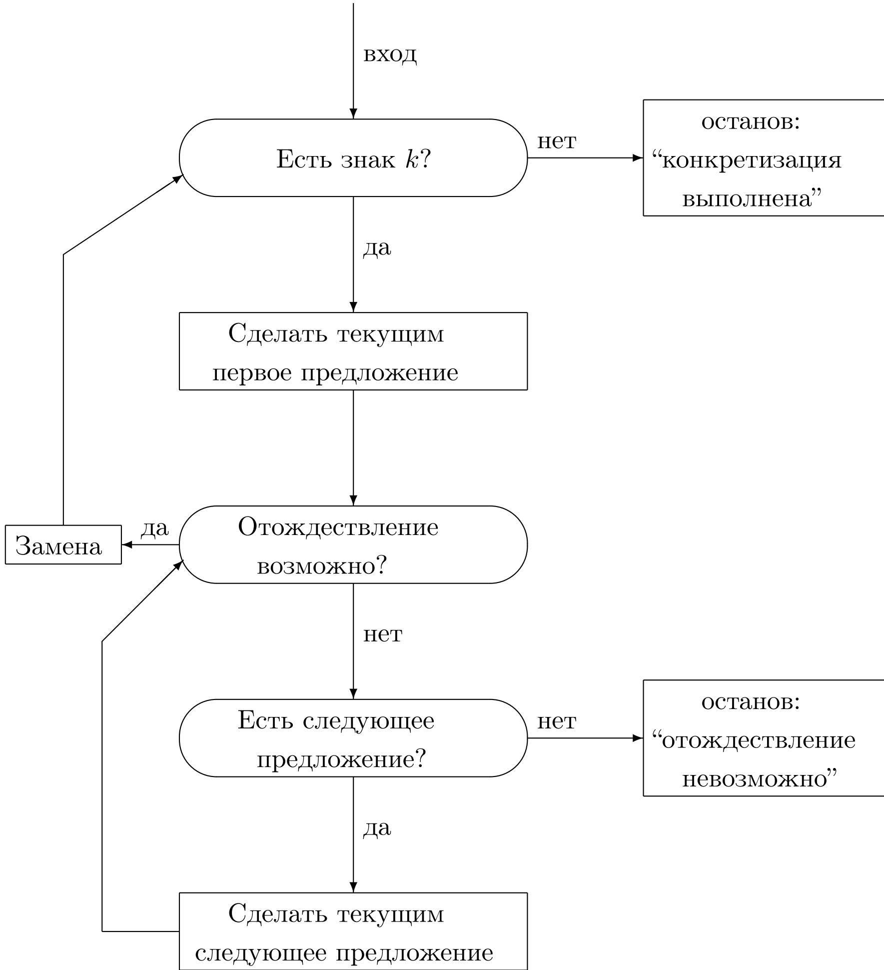


Рис. 1.2: Блок-схема рефал-машины

Работа рефал-машины представляет из. себя последовательность однотипных действий, называемых шагами (рис.1.2).

Выполнение шага начинается с поиска в поле зрения ведущего знака k .

Если в поле зрения нет знаков k , рефал-машина приходит в состояние нормальной остановки: “конкретизация выполнена”.

Если в поле зрения есть знаки k , то рефал-машина находит конкретизируемый терм и сравнивает его с левой частью первого, второго и т.д. предложения в поле памяти, ища подходящее предложение, то есть такое, что конкретизируемый терм может быть отождествлен как его левая часть. Найдя первое подходящее предложение, машина заменяет в поле зрения конкретизируемый терм на правую часть предложения, в которую вместо свободных переменных подставлены значения, присвоенные им в процессе отождествления. Если подходящего предложения не нашлось, происходит аварийная остановка: “отождествление невозможно”.

Глава 2

Алгоритмы отождествления

2.1 Алгоритмический подход к синтаксическому отождествлению

Назовем *правилом отождествления* частичную функцию $F[\mathcal{L}, \mathcal{E}]$ которая для каждого типового выражения $\mathcal{L}$ и объектного выражения $\mathcal{E}$ либо вырабатывает некоторый вариант отождествления $\mathcal{E}$ как $\mathcal{L}$ (см. п.1.2.2), либо не определена.

Согласно этому определению, $F[\mathcal{L}, \mathcal{E}]$ заведомо не определена, если отождествление $\mathcal{E}$ как $\mathcal{L}$ невозможно, более того, она может быть не определена даже в том случае, когда возможно отождествление $\mathcal{E}$ как $\mathcal{L}$.

Будем говорить, что правило отождествления является полным, если $F[\mathcal{L}, \mathcal{E}]$ не определена только в тех случаях, когда отождествление $\mathcal{E}$ как $\mathcal{L}$ невозможно.

Таким образом, задача полного правила отождествления сводится к тому, чтобы выбрать какой-то один вариант отождествления, когда можно отождествить $\mathcal{E}$ как $\mathcal{L}$ несколькими способами.

Поскольку существует бесконечно много таких $\mathcal{E}$ и $\mathcal{L}$, для которых при отождествлении возникает неоднозначность, существует бесконечно много различных правил отождествления.

Правила отождествления могут задаваться различными способами.

В формальном описании рефала, в п.1.2.2, сформулировано правило, которое указывает, какой вариант отождествления следует предпочесть, если возникнет неоднозначность. Тем самым, задано некоторое полное правило отождествления. Но, хотя результат отождествления определен однозначно, через описание свойств, которым он должен удовлетворять, ничего не сказано о том, как отыскать его фактически. Такой способ задания правила отождествления мы будем называть "аксиоматическим".

Между тем, при описании рефал-машины, можно было бы опреде- лить результат отождествления $\mathcal{E}$ как $\mathcal{L}$, описав некоторый алгоритм, который при любых заданных $\mathcal{L}$ и $\mathcal{E}$ либо объявляет, что он "потерпел неудачу", либо на-

ходит какой-то вариант отождествления $\mathcal{E}$ как $\mathcal{L}$. Тем самым, этот алгоритм определяет некоторое правило отождествления, Такой способ задания правила отождествления мы будем называть “алгоритмическим”.

При алгоритмическом подходе выражение $\mathcal{L}$ рассматривается как изощренно закодированная “программа отождествления”, а $\mathcal{E}$ – исходные данные для этой программы.

При аксиоматическом подходе приходится проявлять особую заботу о том, чтобы результат отождествления был определен однозначно. При алгоритмическом подходе эта проблема решается автоматически: алгоритм отождествления выдает “первый попавшийся” вариант отождествления.

В этом разделе мы опишем алгоритм отождествления, который можно было бы использовать при описании рефал-машины. Как выяснится в дальнейшем, этот алгоритм отождествления задает то же правило отождествления, которое было аксиоматически определено в п.1.2.2, хотя заранее это вовсе не очевидно.

Итак, пусть нам даны типовое выражение $\mathcal{L}$ и объектное выражение $\mathcal{E}$. Опишем алгоритм отождествления $\mathcal{E}$ как $\mathcal{L}$.

Будем называть элементами выражений $\mathcal{L}$ и $\mathcal{E}$ символы, скобки и свободные переменные, входящие в $\mathcal{L}$ и $\mathcal{E}$. Промежутки между элементами будем называть *узлами.* Два узла, ограничивающих элемент, будем называть его левым и правым концами.

Работа алгоритма отождествления заключается в том, что узлам из $\mathcal{L}$ ставятся в соответствие некоторые узлы из $\mathcal{E}$.

Пусть узлу из $\mathcal{L}$ сопоставлен узел из $\mathcal{E}$. Тогда будем говорить, что первый узел *спроектирован* на второй и называть второй узел *проекцией* первого. Проекции узлов из $\mathcal{L}$ должны удовлетворять следующим требованиям K1, K2 и K3.

K1. Каждый узел $\mathcal{K}$ в $\mathcal{L}$ может иметь не более, чем одну проекцию в $\mathcal{E}$.

K2. Если в $\mathcal{L}$ узел $\mathcal{K}_1$ расположен левее, чем узел $\mathcal{K}_2$ то в $\mathcal{P}$ проекция $\mathcal{P}_1$ узла $\mathcal{K}_1$ не может находиться правее, чем проекция $\mathcal{P}_2$ узла $\mathcal{K}_2$ •

Свойства K1 и K2 позволяют дать следующее определение. Пусть $\mathcal{P}_1$ и $\mathcal{P}_2$ – проекции концов некоторого элемента, тогда выражение, заключенное между $\mathcal{P}_1$ и $\mathcal{P}_2$ называется проекцией этого элемента. Теперь мы можем сформулировать требование K3.

K3. Проекции свободных переменных должны удовлетворять требованиям, предъявляемым к их значениям, т.е. проекцией s-переменной может быть только символ, t-переменной – только терм, а e-переменной – только выражение. Различные вхождения одной и той же переменной должны иметь одинаковые проекции. Проекции остальных элементов (символов, скобок) должны быть тождественны самим элементам.

На рис.2.1 показан пример проектирования $\mathcal{L}$ на $\mathcal{E}$, удовлетворяющего требованиям K1, K2 и K3.

Все сказанное дает основание употреблять термин “проектирование $\mathcal{L}$ на $\mathcal{E}$” наравне с термином “отождествление $\mathcal{E}$ как $\mathcal{L}$” и также “алгоритм проектирования $\mathcal{L}$ на $\mathcal{E}$”.

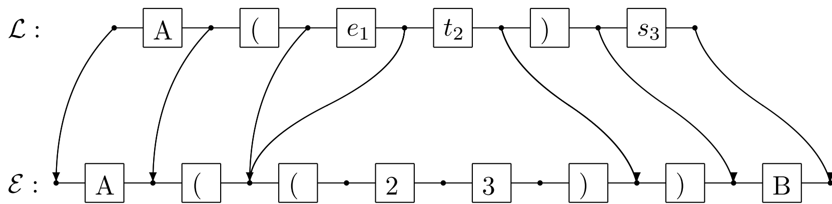


Рис. 2.1: Пример проектирования типового выражения $\mathcal{L}$ на объектное выражение $\mathcal{E}$.

Работа алгоритма проектирования начинается с того, что левый конец $\mathcal{L}$ проектируется на левый конец $\mathcal{E}$, а правый конец $\mathcal{L}$ – на правый конец $\mathcal{E}$.

Каждое последующее действие алгоритма отождествления - это либо *проектирование элемента* из $\mathcal{L}$ либо *перестройка.*

Правила проектирования элементов таковы, что они однозначно определяют порядок, в котором будут проектироваться элементы. Поэтому мы можем пронумеровать все элементы $\mathcal{L}$ в том порядке, в котором они должны проектироваться, присвоив каждому элементу из $\mathcal{L}$ определенный *проекционный номер.* Проекционные номера зависят только от вида выражения $\mathcal{L}$, но не от $\mathcal{E}$, и не меняются при перестройке.

Если некоторая свободная переменная входит в $\mathcal{L}$ несколько раз, то каждому ее вхождению будет соответствовать свой проекционный номер. Вхождение переменной, которое имеет наименьший проекционный номер, будем называть *главным,* а остальные вхождения этой переменной – *повторными.* Значением переменной будет считаться проекция ее главного вхождения.

Символы, скобки, вхождения s-переменных и t-переменных, а также повторные вхождения e-переменных назовем *жесткими* элементами.

Жесткие элементы обладают следующим свойством. Если у такого элемента спроектирован один конец, то второй конец либо проектируется однозначно, либо вообще не может быть спроектирован без нарушения требований К1, К2 и К3. Это очевидно по отношению к символам, скобкам и вхождениям s-переменных. Что касается вхождения t-переменной, то оно может проектироваться либо на символ, либо на выражение в скобках. В последнем случае парная скобка находится в выражении однозначно, по правилам скобочного синтаксиса. Повторное вхождение e-переменной проектируется однозначно потому, что в момент его проектирования нам уже известно значение переменной.

Если в момент проектирования главного вхождения e-переменной уже спроектированы оба конца, мы будем называть это вхождение *закрытым,* в противном случае – открытым.

Жесткие элементы и закрытые вхождения e-переменных мы будем называть *однозначно проектируемыми элементами.*

Теперь мы сформулируем правила, согласно которым производится проек-

тирование элементов.

Будем говорить, что элемент *доступен для проектирования*, если у него уже спроектирован хотя бы один конец, но сам он еще не спроектирован.

Подвергаться проектированию могут только те элементы, которые доступны для проектирования. Обращаем внимание, что если у элемента уже спроектированы оба конца, то это еще не значит, что спроектирован сам элемент: необходимо еще произвести дей- ствие проектирования элемента, которое заключается в проверке того, что проекция элемента удовлетворяет условию КЗ.

Только что было сказано, что можно проектировать только такие элементы, у которые уже спроектирован хотя бы один конец.

Однако, из этого правила есть единственное исключение, касающееся скобок.

Для каждой скобки, входящей в $\mathcal{L}$ или $\mathcal{E}$ всегда можно однозначно найти парную скобку. Можно считать, что парные скобки (и в $\mathcal{L}$, и в $\mathcal{E}$) как бы жестко скреплены между собой. Поэтому, как только мы спроектируем какую-то скобку, мы можем сразу же отыскать скобку, парную к проекции этой скобки и спроектировать первую на вторую (рис.2.2). Именно так и поступает алгоритм отождествления: каждые две парные скобки из $\mathcal{L}$ проектируются одновременно на две парные скобки из $\mathcal{E}$.

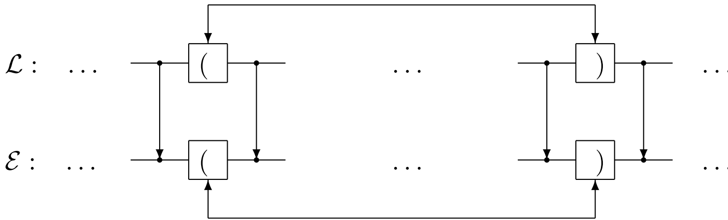


Рис. 2.2: Проектирование пары скобок из $\mathcal{L}$ на пару скобок из $\mathcal{E}$.

Выбор очередного элемента для проектирования производится следующим образом.

Сначала алгоритм отождествления находит все элементы,доступные для проектирования.Затем,доступные элементы делятся на две группы очередности. В первую группу входят однозначно проектируемые элементы, а во вторую – открытые вхождения е-переменных, т.е. вхождения е-переменных, для которых еще не спроектировано ни одно вхождение и еще не спроектированы оба конца.

Алгоритм отождествления стремится в первую очередь проектировать элементы из первой группы, и только если их нет – проектирует элементы из второй группы.

Если в одной группе оказалось несколько элементов, предпочтение отдается тому из них, который расположен в $\mathcal{L}$ левее, чем остальные.

Найдя нужный элемент, алгоритм отождествления пытается его спроектировать. Попытка проектирования элемента либо проходит успешно, либо терпит неудачу. Если проектирование элемента прошло успешно, оба конца элемента получают проекции (или подтверждаются ранее установление проекции). Если же проектирование элемента терпит неудачу, то возникает ситуация *тупика*, когда очередной элемент нельзя спроектировать без перепроектирования уже спроектированных элементов.

Если проектирование скобки проходит успешно, то сразу производится проектирование парной скобки.

Если проектируемый элемент является открытым вхождением е-переменной, то его левый конец уже спроектирован, а правый – нет. Проектирование такого элемента делается так, чтобы он получил пустую проекцию, т.е. его правый конец проектируется на проекцию его левого конца. Проектирование открытого вхождения всегда проходит успешно.

Если проектирование очередного элемента прошло успешно, алгоритм отождествления переходит к проектированию других элементов, если же оно потерпело неудачу, производится перестройка.

Перестройка делается следующим образом. Допустим, что потерпело неудачу проектирование элемента с проекционным номером i_0. Тогда алгоритм отождествления находит открытое вхождение е-переменной с ближайшим номером i_1 , не превышающим i_0 и аннулирует результаты проектирования элементов с номерами $i_1 < i < i_0$. Затем, значение открытой переменной с номером i_1 удлиняется, т.е. проекция ее правого конца переносится на один терм вправо (если это не противоречит условиям К1, К2 и К3) и проектирование продолжается, начиная с элемента номер $i_1 + 1$. Если же удлинение переменной номер i_1 невозможно, делается попытка удлинить открытую переменную с ближайшим проекционным номером i_2, не превышающим i_1, и т.д. Если невозможно удлинить ни одной открытой переменной, процесс синтаксического отождествления заканчивается с результатом "отождествление невозможно".

Если спроектированы все элементы из $\mathcal{L}$, отождествление считается успешно выполненным (рис.2.3).

Рассмотрим теперь несколько примеров проектирования $\mathcal{L}$ на $\mathcal{E}$. Пусть $\mathcal{L}$ имеет вид:

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

где над каждым элементом надписан его проекционный номер. Поскольку все элементы $\mathcal{L}$ – жесткие, все отождествление в целом терпит неудачу, как только терпит неудачу проектирование, хотя бы одного элемента.

В следующее выражение

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

входит три е-переменных, однако, все они – закрытые, поэтому проектирование, как и в предыдущем примере, происходит без всякого перебора. Время, затрачиваемое на успешное отождествление, не зависит от $\mathcal{E}$.

Рис. 2.3: Блок-схема алгоритма отождествления $\mathcal{E}$ как $\mathcal{L}$

Теперь рассмотрим выражение

$$\overset{1}{e_1}\ \overset{2}{+}\ \overset{3}{e_2}$$

в котором переменная e_1 открытая, а e_2 – закрытая. Алгоритм проектирования будет производить удлинение e_1 до тех пор, пока не подберет для e_1 кратчайшее значение, при котором возможно отождествление.

В следующем выражении

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

имеется три открытых переменных e_1, e_3 и e_5 и три закрытых переменных e_2, e_4, e_6.

В двух последних примерах, время затрачиваемое на отождествление, зависит от $\mathcal{E}$, ибо необходим перебор при проектировании открытых вхождений е-переменных.

Рассмотрим еще один пример:

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

На первый взгляд может показаться, что первое вхождение переменной e_1 - открытое. В действительности же оно повторное. Проектируя жесткие элементы, мы добираемся до вхождения e_1, которое является закрытым (элемент №9). Теперь элемент №10 становится жестким и, следовательно, однозначно проектируемым. Таким образом, отождествление производится без всякого перебора. Однако, время, затрачиваемое на отождествление, зависит от значения переменной e_1 , ибо при проектировании повторного вхождения, мы вынуждены просмотреть элемент за элементом проекцию этого вхождения.

В общем случае, время, затрачиваемое на проектирование $\mathcal{L}$ на $\mathcal{E}$, зависит как от $\mathcal{L}$, так и от $\mathcal{E}$ и может быть грубо оценено как $O(l^n)$, где l – длина выражения $\mathcal{E}$, а n – количество открытых и повторных вхождений е-переменных в $\mathcal{L}$.

2.2 Использование различных правил отождествления в языке рефал.

В различных описаниях и реализациях языка рефал использовались различные правила отождествления, которые задавались как аксиоматическим, так и алгоритмическим способом.

В первых описаниях рефала [22,23,24] правило отождествления задавалось аксиоматически и совпадало с правилом, сформулированным в п.1.2.2.

В дальнейшем,в процессе работы над рефал-интерпретатором [26,27,28], был разработан алгоритм отождествления, который давал другое правило отождествления. В описании рефала [25] была сделана попытка сформулировать

это правило отождествления, однако, она не вполне удалась из-за запутанности алгоритма отождествления, принятого в рефал-интерпретаторе.

В процессе работы над рефал-компилятором [29,30,32] был разработан алгоритм отождествления, изложенный в п.2.1. Этот алгоритм использовался для определения правила отождествления в описании рефала [31] , однако было неясно, всегда ли правила отождествления, описанные в п.1.2.2 и п.2.1, дают одинаковый результат.

В работе [34] был проделан теоретический анализ синтаксического отождествления, который показал, что правила отождествления, изложенные в п.1.2.2 и п.2.1 эквивалентны. В связи с этим, в описании рефала [35] вновь было использовано самое раннее [29,30,32] аксиоматическое определение правила отождествления.

Рассмотрим теперь алгоритм отождествления, принятый в рефал-интерпретаторе [26,27]. Этот алгоритм можно переформулировать в тех же терминах, в которых был описан алгоритм проектирования, изложенный в п.2.1. При этом оказывается, что все различие между этими алгоритмами заключено в том, как назначаются проекционные номера для элементов типового выражения $\mathcal{L}$.

Рассмотрим следующий пример. Пусть

$$\mathcal{L} = \overset{1}{e_1} \ \overset{2}{(} \ \overset{9}{e_a} \ \overset{10}{+} \ \overset{11}{e_b} \ \overset{3}{)} \ \overset{4}{e_2} \ \overset{5}{(} \ \overset{8}{e_x} \ \overset{6}{)} \ \overset{7}{e_3}$$

где над элементами надписаны их проекционные номера, в соответствии с тем, как это делает рефал-интерпретатор. Видно, что сначала рефал-интерпретатор спроектирует элементы

$$e_1 \, (\ \ldots\ ) \, e_2 \, (\ \ldots\ ) \, e_3$$

“не заглядывая” внутрь скобок. После этого задача сводится к проектированию двух внутренних подвыражений: $e_a + e_b$ и e_x

Рефал-интерпретатор спроектирует сначала подвыражение e_x а затем – подвыражение $e_a + e_b$, руководствуясь следующим правилом: в первую очередь следует проектировать “закрытые” подвыражения, т.е. подвыражения, которые на нулевом уровне скобочной структуры содержат не более чем одно вхождение е-переменной. Из двух закрытых подвыражений первым проектируется подвыражение, которое расположено в $\mathcal{L}$ левее.

Теперь посмотрим, как будет происходить проектирование выражения $\mathcal{L}$ на объектное выражение:

$$\mathcal{E} = \underbrace{(\mathrm{A})(\mathrm{A})\ldots(\mathrm{A})}_{n \text{ раз}}$$

Сначала переменная e_1 примет пустое значение, а переменная e_2 будет удлиняться $n-1$ раз. При этом, после каждого удлинения e_2 будет делаться одна и та же работа: поиск символа + внутри скобок, с помощью удлинения переменной e_a. Между тем, ясно, что если мы не нашли +, то удлинять e_2 бессмысленно, а нужно сразу же удлинить e_1. Наконец, когда удлинение e_2 станет невозможно,

e_1 удлинится на один терм, после чего e_2 будет удлиняться $n-2$ раза. Таким образом, в процессе отождествления будет проделано n удлинений переменной e_1,

$$(n-1)+(n-2)+\ldots+1=n(n-1)/2$$

удлинений переменной e_2 и

$$(n-2)+(n-3)+\ldots+1=(n-1)(n-2)/2$$

удлинений переменной e_a. Всего потребуется $(n-1)^2$ удлинений.

Теперь посмотрим, как будет работать в этом случае алгоритм проектирования, изложенный в п.2.1. Ему будет соответствовать следующий порядок проектирования:

$$\mathcal{L} = \overset{1}{e_1}\ \overset{2}{(}\ \overset{4}{e_a}\ \overset{5}{+}\ \overset{6}{e_b}\ \overset{3}{)}\ \overset{7}{e_2}\ \overset{8}{(}\ \overset{10}{e_x}\ \overset{9}{)}\ \overset{11}{e_3}$$

из чего следует, что при проектировании $\mathcal{L}$ на $\mathcal{E}$ будет проделано n удлинений переменной e_1 , n удлинений переменной e_a, а до проектирования e_2 дело вообще не дойдет. Всего потребуется $2n$ удлинений.

Таким образом, оба алгоритма отождествления обнаружат, что отождествление $\mathcal{E}$ как $\mathcal{L}$ невозможно, но алгоритму проектирования из п.1.3 потребуется для этого $O(n)$ удлинений, а рефал-интерпретатору – $O(n^2)$ удлинений.

Итак, мы разобрали, какое влияние оказывают скобки на порядок, в котором рефал-интерпретатор проектирует подвыражения из $\mathcal{L}$. Теперь рассмотрим, как производится проектирование каждого выражения.

Интерпретатор движется по подвыражению слева направо, не заходя внутрь скобок, до тех пор, пока он не дойдет до конца подвыражения или не встретит "закрытое" вхождение е-переменной, т.е. такое, которое не является повторным и вслед за которым больше нет ни одного вхождения е-переменной. В последнем случае интерпретатор меняет направление движения на противоположное, т.е. находит правый конец подвыражения и начинает двигаться по нему справа налево. Например:

$$\mathcal{L} = \overset{1}{s_a}\ \overset{2}{e_1}\ \overset{3}{+}\ \overset{4}{e_2}\ \overset{5}{(}\ \overset{10}{e_z}\ \overset{6}{)}\ \overset{9}{e_3}\ \overset{8}{t_y}\ \overset{7}{\mathrm{A}}$$

Здесь вхождение переменной e_3 – закрытое, направление движения меняется в тот момент когда мы доходим до e_3.

Ясно, что такой порядок проектирования элементов приводит к неэффективной работе алгоритма отождествления даже во многих тривиальных и часто встречающихся случаях. Пусть, например,

$$\begin{aligned} \mathcal{L} &= \overset{1}{e_1}\ \overset{2}{\mathrm{A}}\ \overset{4}{e_2}\ \overset{3}{\mathrm{B}} \\ \mathcal{E} &= \underbrace{\mathrm{A\ A\ \ldots\ A}}_{n\ \text{раз}} \end{aligned}$$

Здесь отождествление $\mathcal{E}$ как $\mathcal{L}$ невозможно, но чтобы обнаружить это, рефал-интерпретатор проделает n удлинений переменной e_1. В тоже время,

алгоритм проектирования из п.2.1 делает проектирование в следующем порядке:

$$\mathcal{L} = \overset{2}{e_1} \ \overset{3}{\mathrm{A}} \ \overset{4}{e_2} \ \overset{1}{\mathrm{B}}$$

поэтому он обнаружит, что отождествление невозможно, как только попытается спроектировать символ B. Таким образом, не потребуется ни одного удлинения.

Во всех до сих пор разобранных примерах оба алгоритма отождествления давали одинаковый результат, хотя для этого им потребовалось разное время. Однако нетрудно найти такие $\mathcal{L}$ и $\mathcal{E}$, для которых эти алгоритмы дадут разный результат (рис.2.4 и 2.5). Таким образом, эти алгоритмы не эквивалентны.

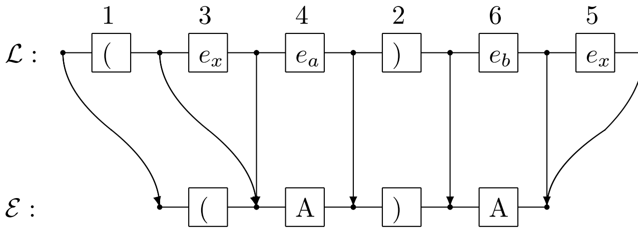


Рис. 2.4: Результат проектирования для алгоритма из п.2.1

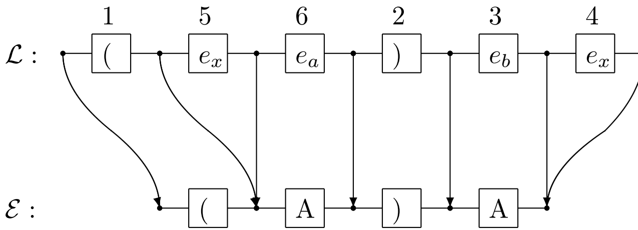


Рис. 2.5: Результат проектирования для рефал-интерпретатора

2.3 Преимущества и недостатки аксиоматического и алгоритмического подхода к отождествлению.

Центральное место в описании рефала занимает правило отождествления. Его можно определить как аксиоматически, так и алгоритмически, и оба способа определения имеют свои достоинства и недостатки.

С одной стороны, описание языка предназначено для тех, кто будет его использовать, с другой стороны – для его реализаторов. Оно должно, по возможности, удовлетворять интересам как тех, так и других.

Аксиоматическое определение правила отождествления гораздо короче и идейно проще, чем алгоритмическое. Оно однозначно указывает, каков результат отождествления для любых $\mathcal{L}$ и me, но не навязывает конкретный способ его нахождения. Это вполне соответствует интересам пользователей языка. Реализатор же языка должен изучить свойства этого правила отождествления и на этой основе разработать более или менее эффективный алгоритм отождествления. Ясно, что при этом ни один из разработанных алгоритмов отождествления не должен рассматриваться как наилучший из возможных, ибо более тщательный анализ правила отождествления наверняка позволит разработать более сложный, но и более эффективный алгоритм.

Алгоритмическое определение правила отождествления освобождает реализатора языка от обязанности разрабатывать свой алгоритм отождествления. Достаточно просто реализовать тот алгоритм, который указан в описании языка. Если же алгоритм отождествления будет работать плохо, то виноват в этом будет не реализатор языка, а автор описания языка. Вполне естественно, что из-за этого в описание языка придется включить достаточно сложный, пригодный для практического использования алгоритм. Чем совершеннее будет этот алгоритм, тем больше усложнится описание языка, и тем труднее будет предугадать результат работы алгоритма для конкретных $\mathcal{L}$ и $\mathcal{E}$.

Таким образом, алгоритмическое определение правила отождествления не соответствует интересам пользователей языка. Однако оказывается, что оно не столь уж удобно и для реализатора языка. В самом деле, когда реализатор языка замечает, что имеющийся алгоритм плохо работает в каких-то случаях, у него возникает желание улучшить этот алгоритм. При этом, улучшенный алгоритм должен работать неправильно", т.е. для любых $\mathcal{L}$ и $\mathcal{E}$ давать тот же результат, что и алгоритм, сформулированный в описании языка. Таким образом, возникает необходимость доказывать эквивалентность различных алгоритмов отождествления. Это трудная задача, ибо работа алгоритмов протекает во времени, человеку же гораздо легче мыслить в статических понятиях объектов, множеств, отношений и т.д. Поэтому, реализатору языка придется изучить свойства алгоритма отождествления и, фактически, сформулировать некоторое аксиоматическое определение для правила отождествления, задаваемого этим алгоритмом, а затем, на этой основе, разработать свой алгоритм отождествления. Это сложный и противоестественный путь. Гораздо проще начать прямо с некоторого аксиоматического определения правила отождествления и на его основе разработать алгоритм отождествления.

Исходя из сказанного, мы примем за основу правило отождествления, аксиоматически сформулированное в п.1.2.2. Это правило отождествления эквивалентно алгоритму проектирования $\mathcal{L}$ на $\mathcal{E}$ из п.2.1 (см.[34] и приложение А)

В следующем разделе обсуждаются некоторые улучшения алгоритма проектирования за счет исключения лишних удлинений открытых вхождений е-переменных. Достаточные условия, при которых возможны эти оптимизации,

сформулированы в главе 4 (доказательство их справедливости см.[34] и приложение A).

В главе 3 описан язык сборки, на котором для любого типового выражения $\mathcal{L}$ можно записать алгоритм проектирования $\mathcal{L}$ на произвольное объектное выражение $\mathcal{E}$.

2.4 Оптимизация удлинений открытых вхождений e-переменных.

Согласно описанию алгоритма проектирования (п.2.1), когда удлинение открытого вхождения e-переменной с проекционным номером i невозможно, следует перейти к удлинению другого открытого вхождения, имеющего максимальный проекционный номер i' такой, что $i' < i$. Если же такого вхождения нет, все проектирование $\mathcal{L}$ на $\mathcal{E}$ терпит неудачу.

Однако, легко привести примеры, когда удлинение переменной i' заведомо не приведет к отождествлению, поэтому, как только потерпит неудачу удлинение переменной с номером i, можно либо сразу заключить, что все проектирование потерпело неудачу, либо удлинять не переменную с номером i', а какую-то другую переменную с номером i'' таким, что $i'' < i' < i$.

Таким образом, мы “перепрыгиваем” через удлинение переменной с номером i' , что приводит к ускорению работы алгоритма проектирования.

Пусть, например,

$$\mathcal{L} = \overset{1}{e_1} \; \overset{2}{+} \; \overset{3}{e_2} \; \overset{4}{*} \; \overset{5}{e_3}$$

Нетрудно показать, что если терпит неудачу удлинение переменной e_2, то можно сразу же, не пытаясь удлинять e_1, сделать вывод, что проектирование $\mathcal{L}$ на $\mathcal{E}$ терпит неудачу. В самом деле, пусть удлинение e_2 потерпело неудачу. Тогда $\mathcal{E}$ можно представить в виде $\mathcal{E} = \mathcal{E}_1 + \mathcal{E}_2$, где $\mathcal{E}_1$ – часть $\mathcal{E}$, на которую спроектирована e_1. $\mathcal{E}_2$ не содержит * на нулевом уровне скобок, поскольку удлинение e_2 потерпело неудачу. Ясно, что если удлинить e_1 , то после этого $e_2 * e_3$ будет проектироваться на часть $\mathcal{E}_2$, которая заведомо не будет содержать * на нулевом уровне скобок, следовательно, проектирование $e_2 * e_3$ опять потерпит неудачу. Значит, удлинять e_1 бесполезно.

Отказ от удлинения e_1 может значительно ускорить работу алгоритма отождествления. Пусть, например,

$$\mathcal{E} = \underbrace{+ + \ldots +}_{n \text{ раз}}$$

Легко видеть, что неоптимизированный алгоритм проделает при проектировании $\mathcal{L}$ на $\mathcal{E}$ n удлинений переменной e_1 и

$$(n-1) + (n-2) + \ldots + 1$$

удлинений переменной e_2. Всего, таким образом, будет проделано $n(n+1)/2$ удлинений. Оптимизированный алгоритм проделает $n-1$ удлинение переменной e_2, а до удлинения e_1 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 следующем случае

$$\begin{array}{cccccccccc} & & 1 & 3 & 4 & 5 & 2 & 6 & 7 & 8 \\ \mathcal{L} & = & (& e_a & + & e_b &) & e_x & + & e_y \end{array}$$

нетрудно доказать, что если удлинение e_x терпит неудачу, то проектирование $\mathcal{L}$ на $\mathcal{E}$ невозможно. Действительно, если удлинение e_x потерпело неудачу, то $\mathcal{E}$ можно представить в виде $\mathcal{E} = (\mathcal{E}_1)\mathcal{E}_2$, где $e_a + e_b$ спроектировано на $\mathcal{E}_1$, а проектирование $e_x + e_y$ на $\mathcal{E}_2$ невозможно. Значит, $\mathcal{E}_2$ не содержит $+$ на нулевом уровне скобок и положение никак не изменится от того, что мы удлиним e_a.

Оценим выгоды от оптимизации. Пусть $\mathcal{E}_1$ состоит из n термов, среди которых p термов являются символом $+$, а $\mathcal{E}_2$ состоит из m термов, среди которых нет ни одного символа $+$. Тогда, в процессе проектирования $\mathcal{L}$ на $\mathcal{E}$, неоптимизированный алгоритм проделает $n + p \times m$ удлинений переменных e_a и e_x , а оптимизированный – не больше, чем $(n-p)+m$ удлинений. Если считать, что p пропорционально n , т.е. $p = a \times n, a < 1$, то получаем для оптимизированного алгоритма $n \times (1-a) + m$ удлинений, а для неоптимизированного $n + a \times n \times m$ удлинений. В первом случае время работы растет линейно с ростом размеров выражения $\mathcal{E}$, а во втором случае – квадратично.

Приведем теперь пример, когда оптимизация, описанного выше типа, невозможна. Пусть

$$\begin{array}{cccccccccc} & & 1 & 3 & 4 & 5 & 2 & 6 & 7 & 8 \\ \mathcal{L} & = & (& e_1 & s_x & e_2 &) & e_3 & s_x & e_4 \end{array}$$

Здесь, если удлинение e_3 потерпит неудачу, мы обязаны попытаться удлинить e_1. В этом легко убедиться, рассмотрев проектирование $\mathcal{L}$ на выражение

$$\mathcal{E} = (+ *) * * *$$

Проектирование будет происходить так. Сначала e_1 примет пустое значение, а значением s_x станет $+$. После этого проектирование $e_3\ s_x\ e_4$ на $*\ *\ *$ потерпит неудачу. e_1 примет значение $+$, а s_x – значение $*$, после чего проектирование успешно закончится. Очевидно, что здесь оптимизация невозможна из-за того, что подвыражения $e_1\ s_x\ e_2$ и $e_3\ s_x\ e_4$ содержат общую переменную s_x. Точно так же невозможна оптимизация для выражения $e_1\ s_x\ e_2\ s_x\ e_3$

Однако, можно привести примеры, когда наличие общей переменной не является помехой для оптимизации:

$$\begin{array}{ccccccccccc} & & 1 & 2 & 4 & 5 & 6 & 3 & 7 & 8 & 9 \\ \mathcal{L} & = & s_x & (& e_1 & s_x & e_2 &) & e_3 & s_x & e_4 \end{array}$$

$$\begin{array}{cccccccc} & & 2 & 3 & 4 & 5 & 6 & 1 \\ \mathcal{L} & = & e_1 & s_x & e_2 & s_x & e_3 & s_x \end{array}$$

До сих пор мы рассматривали случаи, когда при оптимизации удлинений не приходилось изменять очередность проектирования элементов. Теперь мы рассмотрим более сложный пример.

$$\mathcal{L} = \overset{1}{(}\ \overset{5}{e_1}\ \overset{6}{s_x}\ \overset{7}{e_2}\ \overset{2}{)}\ \overset{8}{e_a}\ \overset{9}{+}\ \overset{10}{e_b}\ \overset{4}{(}\ \overset{11}{e_3}\ \overset{12}{s_x}\ \overset{13}{e_4}\ \overset{3}{)}$$

Здесь алгоритм проектирования будет делать лишнюю работу, повторяя заново проектирование подвыражения $e_a + e_b$ после каждого удлинения переменной e_1 . Между тем, можно один раз спроектировать $e_a + e_b$, а потом взяться за проектирование подвыражений $e_1\, s_x\, e_2$ и $e_3\, s_x\, e_4$. Таким образом, получим следующие проекционные номера:

$$\mathcal{L} = \overset{1}{(}\ \overset{8}{e_1}\ \overset{9}{s_x}\ \overset{10}{e_2}\ \overset{2}{)}\ \overset{5}{e_a}\ \overset{6}{+}\ \overset{7}{e_b}\ \overset{4}{(}\ \overset{11}{e_3}\ \overset{12}{s_x}\ \overset{13}{e_4}\ \overset{3}{)}$$

Причем, если удлинение e_1 терпит неудачу, мы сразу объявляем, что проектирование $\mathcal{L}$ на $\mathcal{E}$ невозможно.

Мы могли бы поступить и иначе, спроектировав сначала $e_1\, s_x\, e_2$ и $e_3\, s_x\, e_4$, а затем $e_a + e_b$.

$$\mathcal{L} = \overset{1}{(}\ \overset{5}{e_1}\ \overset{6}{s_x}\ \overset{7}{e_2}\ \overset{2}{)}\ \overset{11}{e_a}\ \overset{12}{+}\ \overset{13}{e_b}\ \overset{4}{(}\ \overset{8}{e_3}\ \overset{9}{s_x}\ \overset{10}{e_4}\ \overset{3}{)}$$

Здесь, если удлинение e_1 невозможно, мы сразу объявляем, что проектирование $\mathcal{L}$ на $\mathcal{E}$ невозможно.

Эффект , который дает оптимизация удлинений, особенно наглядно проявляется при рассмотрении следующего примера.

$$\mathcal{L} = e_1\, e_2\, \ldots\, e_n\, *\, e_0$$

$$\mathcal{E} = \underbrace{\text{A A ... A}}_{l\ \text{раз}}$$

Здесь $\mathcal{L}$ содержит n открытых е-переменных, а $\mathcal{E}$ имеет длину l. Можно показать, что при проектировании $\mathcal{L}$ на $\mathcal{E}$ потребуется $O(l^n)$ удлинений. Теперь проделаем оптимизацию. Несложными рассуждениями можно показать, что если удлинение невозможно, то проектирование всего $\mathcal{L}$ терпит неудачу. Таким образом, после оптимизации достаточно всего $O(l)$ удлинений. В главе 4 будут сформулированы достаточные условия, при которых возможна оптимизация удлинений открытых переменных.

2.5 Специфика синтаксического отождествления в языке рефал.

Синтаксическое отождествление (syntactic recognition) занимает в языке рефал важное, если не центральное место, однако, нет никаких оснований считать его достоянием одного рефала. Оно используется во многих алгоритмических

языках, где известно также под названием “поиск по образцу” или “сопоставление с образцом” (pattern matching) и “отождествление с образцом” (pattern recognition).

Обычно в синтаксическом отождествлении участвуют две строки. Одна из этих строк может содержать переменные, ее называют “образцом” (pattern). Другую строку, не содержащую переменных, называют “объектом” (subject). В результате успешного отождествления переменные образца получают некоторые значения.

В неформализованном виде синтаксическое отождествление использовалось в математике еще задолго до появления алгоритмических языков. Когда, например, мы записываем закон коммутативности сложения для натуральных чисел в виде

$$a + b = b + a$$

мы подразумеваем, что это равенство, содержащее “свободные переменные” a и b изображает бесконечное множество равенств

$$0 + 0 = 0 + 0$$
$$0 + 1 = 1 + 0$$
$$25 + 13 = 13 + 25$$

каждое из которых получается, если вместо a и b поставить некоторые натуральные числа. Задача, состоящая в распознавании того, принадлежит ли некоторое равенство, например,

$$1 + 1036 = 1036 + 5$$

к этому множеству - это и есть задача синтаксического отождествления.

Интуитивные представления об отождествлении были уточнены в формализации понятия вычислимой функции Эрбрана и Геделя [1,3]. В этой формализации рассматриваются функции, определенные посредством систем рекурсивных соотношений. Аргументами функций являются натуральные числа, которые представляются следующим образом.

Каждому числу n ставится в соответствие его представление $\overline{n}$, согласно следующим правилам:

1. $\overline{0}$ есть 0
2. $\overline{n+1}$ есть $(\overline{n})'$

Например, $\overline{3}$ есть $(((0)')')'$.

Каким способом определяются функции нетрудно понять, рассмотрев следующее определение функции сложения:

$$+(x_1, 0) = x_1$$
$$+(x_1, (x_2)') = (+(x_1, x_2))'$$

В дальнейшем, язык, на котором описываются рекурсивные функции в формализации Эрбрана-Геделя, мы будем называть сокращенно языком ЭГ.

Определения функций, написанные на языке ЭГ, легко компилируются в программы на рефале. Для этого нужно, прежде всего, договориться, как кодировать натуральные числа с помощью рефал-выражений. Обозначим через $\overline{\overline{n}}$ представление числа n в виде рефал-выражения. Выберем следующее представление:

1. $\overline{\overline{0}}$ есть 0
2. $\overline{\overline{n+1}}$ есть $(\overline{\overline{n}})$

Теперь нетрудно понять, как произвести компиляцию с ЭГ на рефал. Для этого нужно в ЭГ программе каждую переменную x_i заменить на t_i, конструкции вида $(...)'$ заменить на $(...)$, а функциональные термы вида $\mathcal{F}(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n)$ заменить на $k\ \mathcal{F}\mathcal{A}_1\mathcal{A}_2\ldots\mathcal{A}_n\ \perp$. После такого преобразования получится программа на рефале. Так, например, определение сложения примет следующий вид:

$$\S\, k\ +t_1\,0\ \sim\ t_1$$
$$\S\, k\ +t_1\,(t_2)\ \sim\ (\ k\ +t_1\,t_2\perp)$$

Интересно, что после компиляции с ЭГ получаются рефал-программы, в которых не используются ни s-переменные, ни e-переменные, однако, важную роль играют скобки. Получающиеся левые части предложений таковы, что синтаксическое отождествление всегда может быть произведено не более чем одним способом. При отождествлении не требуется перебор.

Другим языком, в котором использовано синтаксическое отождествление, является язык нормальных алгорифмов А.А.Маркова [2,3]. В дальнейшем мы будем сокращенно называть этот язык языком АМ.

Программа, записанная на языке АМ, представляет из себя последовательность *формул подстановки*, каждая из которых имеет вид $\alpha \to \beta$, либо $\alpha \to. \beta$, где α и β – некоторые строки. Такую программу легко скомпилировать в программу на рефале. Для этого достаточно каждую формулу подстановки вида $\alpha \to \beta$ заменить на рефал-предложение

$$\S\, k\, e_1\, \alpha\, e_2\ \sim\ \ k\, e_1\, \beta\, e_2 \perp$$

а каждую формулу подстановки вида $\alpha \to. \beta$ заменить на рефал-предложение

$$\S\, k\, e_1\, \alpha\, e_2\ \sim\ e_1\, \beta\, e_2$$

Например, алгоритм

$$A11 \to A1$$
$$A1 \to. 1$$
$$\to A$$

переведется в следующую программу на рефале:

$$\S\, k\, e_1\ A11\, e_2\ \sim\ \ k\, e_1\ A1\, e_2 \perp$$
$$\S\, k\, e_1\ A1\, e_2\ \sim\ e_1\ 1\ e_2$$
$$\S\, k\, e_1\, e_2\ \sim\ \ k\, e_1\ A\, e_2 \perp$$

Интересно, что после компиляции с АМ получаются рефал-программы, в которых не используются ни s-переменные, ни t-переменные, ни скобки, но зато используются открытые е-переменные. Кроме того, знаки k, используются несущественно, поскольку в правой части каждого предложения находится не более, чем один знак k. Таким образом, мы видим, что программы, написанные на ЭГ и АМ легко компилируются в программы на рефале. Причина этого состоит в том, что рефал, в сущности, просто содержит в себе ЭГ и АМ как подмножества. В то же время, рефал является синтезом этих языков.

Хорошо видно, что t-переменные, скобки и знаки k, рефал позаимствовал из языка ЭГ, а открытые е-переменные из языка АМ.

Нормальные алгорифмы А.А.Маркова послужили отправной точкой не только при создании языка рефал, но и при разработке языка COMIT[4,5], а также таких языков для обработки строк как SNOBOL [8,9,10], AXLE[11], AMBIT [12, I3], макропроцессора LIMP [16] .

Почти все языки, возникшие из алгориамов А.А.Маркова, предназначены для обработки строк, т.е. линейных последовательностей литер, не имеющих явно выраженной синтаксической структуры. Таким образом, скобки в строках не играют особой, выделенной роли; их представление в памяти машины ничем не отличается от представления остальных литер. Правда, в языке SN0BOL предусмотрены переменные, значением которых может быть только строка, сбалансированная по скобкам, однако, поскольку обрабатываемые строки не обязаны быть сбалансированы по скобкам, скобки никак не используются для ускорения отождествления.

В языке AMBIT[12,13] обрабатываемые объекты являются деревьями, как и в языке рефал. При отождествлении скобки играют ключевую роль. Однако, в языке допускаются только образцы такого вида, для которых отождествление может быть произведено не более, чем одним способом. Подмножество рефала, описанное в [33,36], допускает только такие левые части рефал-предложений, которые соответствуют образцам языка AMBIT. Эти ограничения при- водят к тому, что левые части предложений не содержат открытых е-переменных.

Язык ЭГ послужил одним из источников при создании языка LISP [6,7]. Объектом обработки в языке LISP являются бинарные деревья.

LISP оказался весьма неудобен для обработки текстов (редактирование, компиляция и т.д.), поскольку в нем не предусмотрено синтаксическое отождествление [14,15]. Поэтому, были созданы системы CONVERT[14] и БЛИСС[15], которые позволяли в программах на языке LISP использовать фрагменты, написанные на языках CONVERT и БЛИСС. БЛИСС, фактически, является подмножеством языка SNOBOL и обрабатывает линейные строки. Тем самым, он игнорирует тот факт, что LISP позволяет обрабатывать бинарные деревья, а не линейные строки. В языке CONVERT при синтаксическом отождествлении и объект и образец – деревья, представленные в виде бинарных деревьев. Поэтому, в процессе отождествления возможно движение только слева направо, но не справа налево. Отождествление при этом делается менее эффективно, чем когда

можно двигаться в обоих направлениях. Например, в следующем выражении

$$\mathcal{L} = \overset{1}{(} \ \overset{3}{e_a} \ \overset{2}{)} \ \overset{5}{e_1} \ \overset{4}{e_a}$$

переменная e_1 – закрытая. Если же движение справа налево запретить, то придется делать проектирование следующим образом:

$$\mathcal{L} = \overset{1}{(} \ \overset{3}{e_a} \ \overset{2}{)} \ \overset{4}{e_1} \ \overset{5}{e_a}$$

и переменная e_1 станет открытой.

Интересно проследить, каким способом аксиоматическим или алгоритмическим определено правило отождествления в различных языках. Как мы уже видели в п.2,3, за время существования рефала в этом вопросе наблюдались колебания, т.е. применялся и тот и другой подход.

В большинстве алгоритмических языков не проводится последовательно аксиоматическая или алгоритмическая точка зрения, а используется смешанный подход: правило отождествления описывается с помощью алгоритма, многие части которого описаны аксиоматически, т.е. указано, что они делают, но не указано – как.

В языках ЭГ, АМ и AXLE применяется чисто аксиоматический подход. В языках COMIT, SNOBOL и CONVERT подход смешанный.

Интересно заметить, что в серии языков SNOBOL происходила постепенная эволюция в сторону алгоритмического подхода. Если в языках SNOBOL1[8] и SNOBOL3[9] описание правила отождествления еще близко к аксиоматическому, то в SNOBOL4[10] окончательно восторжествовала алгоритмическая точка зрения. В SNOBOL4 образец превратился в сложную программу, действие которой подчас нелегко понять. Выполнение образца может привести, например, к зацикливанию программы.

В языке AMBIT используется алгоритмическое определение правила отождествления.

Итак, на основании вышеизложенного, мы можем сделать вывод, что синтаксическое отождествление в языке рефал обладает следующими особенностями:

1. И образец, и объект симметричные, т.е. в процессе отождествления возможно движение как слева направо, так и справа налево. Эта черта рефала – общая с языками ЭГ и AMBIT.

2. И образец, и объект являются деревьями, которые изображаются с помощью скобок. Таким образом, скобки играют выделенную роль, служат для выделения синтаксической структуры объектов, что используется для ускорения отождествления. Эта черта рефала – общая с языками ЭГ, AMBIT и CONVERT.

3. Существуют такие пары образец-объект, для которых отождествление возможно несколькими различными способами. Из-за этого, в процессе отождествления может потребоваться комбинаторный перебор. Эта черта рефала – общая с языками АМ, COMIT, SNOBOL, AXLE, CONVERT.

4. Правило отождествления определяется аксиоматически, как и в языках ЭГ и АМ.

Глава 3

Язык сборки для базисного рефала

3.1 Язык сборки как система команд

В этой главе описывается язык сборки для базисного рефала [29,32], который, фактически, представляет собой некоторую систему команд. Автор надеется, что эта система команд столь же хорошо подходит для реализации рефала, как системы команд большинства существующих ЭВМ для реализации Фортрана.

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 и отражено сколько операндов и каких 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 составляет исключения. Мы называем ее языком сборки для рефала (refal assembly language) 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.

Интересно отметить, что язык сборки не сразу приобрел такой вид. Ранний язык сборки [29] своим внешним видом наводил на мысль, что его следует компилировать, а не интерпретировать. Только в процессе дальнейшей работы кристаллизовалась концепция языка сборки как системы команд.

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 уже имеющейся ЭВМ. 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 объем интерпретатора языка сборки составляет 600-1000 команд.

Интерпретатор языка сборки был реализован программным путем на машинах БЭСМ-6 [37, 38, 39] и МИНСК-32, а также на машинах серии М-220 [40, 41] и серии ЕС ЭВМ.

Путь аппаратной реализации языка сборки – гораздо более трудоемкий. Однако, он позволяет повысить скорость работы рефал-программ примерно на порядок [42,43]. В настоящее время работы по аппаратной реализации интерпретатора языка сборки ведутся в ИПМ АН СССР под руководством И.Б.Задыхайло, А.Н.Мямлина и В.К.Смирнова.

В этой главе будет описан язык сборки, интерпретатор языка сборки и на конкретных примерах будут показаны методы перевода программ с рефала на язык сборки и 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 фортраноподобном языке.

3.2 Организация памяти

Во время работы рефал-программы обрабатываемая информация находится в поле зрения (см.п.1.2). 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 объект, 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, мы будем считать, что поле зрения представлено в виде списка, имеющего следующую структуру.

Основной единицей списка является звено. Память, отведенная под поле зрения, представляет из себя 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 восьми или двенадцати байтов.

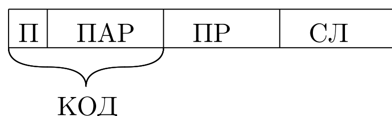


Рис. 3.1: Структура звена

Каждое звено состоит из трех полей: `КОД`, `ПР` и `СЛ`. Поле `КОД` подразделяется на два подполя: `П` и `ПАР` (рис.3.1).

Размер полей `ПР` и `СЛ` должен быть таким, чтобы они могли содержать адрес любого звена, входящего в поле зрения. Поля `ПР` и `СЛ` служат для связывания звеньев в линейную последовательность. Поле `ПР` всегда содержит адрес предыдущего звена в этой последовательности, а поле `СЛ` содержит адрес следующего звена.

Содержимое поля `КОД` зависит от того, какому объекту соответствует данное звено. Звено может изображать символ, либо скобку (левую или правую). Представление знака конкретизации и конкретизационной точки в поле зрения будет описано в п.3.2.3.

Содержимое поля `КОД` в описании языка сборки полностью не детализируется. Однако, предполагается, что по нему всегда можно различить, что изображает звено: символ, левую скобку или правую скобку. При этом, для скобки поле `КОД` должно содержать в себе адрес соответствующей парной скобки.

Отсюда следует, что размер поля `КОД` должен быть больше, чем размер полей `ПР` и `СЛ`. Поэтому поле `КОД` можно подразделить на два подполя: `П` и `ПАР`, где размер поля `ПАР` совпадает с размером полей `ПР` и `СЛ`, а размер поля `П` достаточен, чтобы по нему можно было отличить символ от скобки, а левую скобку от правой.

Больше ничего о содержимом поля `КОД` не предполагается. Это находится в полном соответствии с тем, что рефал-машина оперирует с символами как с неделимыми объектами, природа которых ей безразлична.

В настоящее время символ может быть объектным знаком, меткой или числом. 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

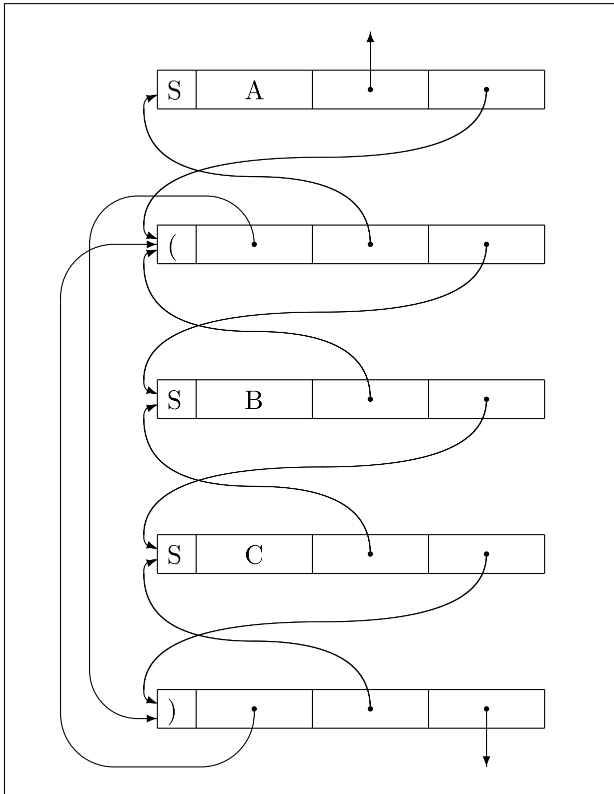


Рис. 3.2: Представление выражения A(BC) в виде списка

он распознает по полному совпадению содержимого полей КОД у соответствующих звеньев.

Поясним сказанное на примере машины БЭСМ-6.

Звено в машине БЭСМ-6 занимает одно слово (48 разрядов). Размер полей ПР, СЛ и ПАР равен длине адреса – 15 разрядов. Под поле П остается 3 разряда. Таким образом, длина поля КОД – 18 разрядов.

Для объектов различных типов поле П принимает следующие значения:

001 (

011)

000 объектный знак

010 символ-метка

100 символ-число

В зависимости от значения признака П в поле ПАР записывается или адрес парной скобки, или адрес, равный значению метки, или соответствующее число.

На рис.3.2 изображено представление выражения A(BC) в виде списка.

В дальнейшем, мы будем использовать более компактные обозначения, при

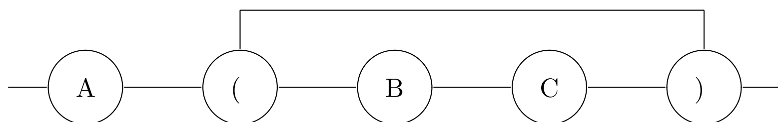


Рис. 3.3: Более компактное изображение списка, приведенного на рис.3.2.

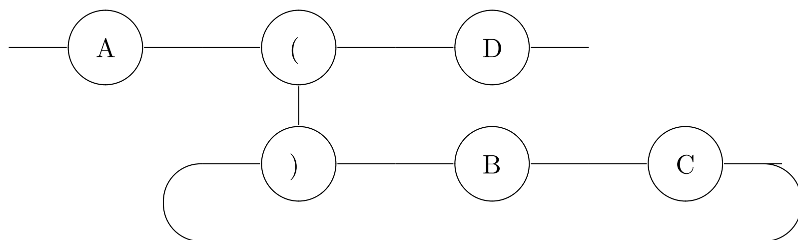


Рис. 3.4: Список, соответствующий выражению A(BC)D, если выбрано другое представление для скобок.

изображении списков на рисунках (см. рис.3.3). Каждое звено будем рисовать в виде кружка. Линии, выходящие из кружка горизонтально, изображают ссылки, находящиеся в полях `ПР` и `СЛ`. Внутри кружка записан объект, который содержит звено. Если поле `ПАР` содержит ссылку на другое звено, эта ссылка изображается с помощью линии, выходящей из кружка вертикально.

Списки такой структуры применялись ранее в рефал-интерпретаторе [26,27] для представления поля зрения и в реализации языка AMBIT [12,13]. Интересно отметить, что можно было бы использовать другую организацию списка для представления скобок. Выбранное нами представление скобок точно соответствует тому, как выражения рефала записываются на бумаге. Выражение при этом мыслится как линейная, одноуровневая строка, хотя и сбалансированная по скобкам. Однако, выражение можно рассматривать как дерево. При этом мы получаем другое представление для скобок (рис.3.4). При этом выражение представлено как линейная последовательность термов. Для каждого терма вида $(\mathcal{E})$ в списке содержится звено, которое в поле `П` содержит признак "левой скобки", а в поле `ПАР` – ссылку на другое звено – "голову" подвыражения $\mathcal{E}$. В "голове" в поле `П` содержится признак "правой скобки", а в поле `ПАР` – обратная ссылка на "левую скобку". Поля `ПАР` и `СЛ` используются, чтобы присоединить к "голове" выражение $\mathcal{E}$, которое образует совместно с головой симметричный циклический список.

Такая структура списка сходна с той, которая применялась в системе SLIP [17]. Отличие состоит только в том, что в системе SLIP разрешено использо-

вать один и тот же подсписок в нескольких местах. Поэтому "голова" каждого подсписка в поле ПАР содержит счетчик числа ссылок на данный подсписок.

Возникает, естественно, вопрос, какое представление скобок удобнее выбрать при реализации языка рефал? С точки зрения синтаксического отождествления, оба представления одинаково удобны. Однако, оказывается, что при выполнении преобразования ведущей области конкретизации, удобно рассматривать выражения как линейные строки, не делая принципиального различия между скобками и символами. Поэтому, в дальнейшем, для скобок будет использоваться первое из рассмотренных представлений.

Наконец, необходимо отметить следующий принципиальный недостаток выбранного нами представления поля зрения. (Впрочем, недостаток этот типичен для универсальных систем обработки списков). А именно, всегда используется некоторое стандартное представление информации, которое зафиксировано раз и навсегда и не зависит от программы на рефале. В принципе, можно разработать компилятор, который будет выбирать такое представление поля зрения, которое соответствует конкретному алгоритму на основе глубокого анализа исходной рефал-программы. Однако, в данной работе такая задача не ставилась.

3.3 Форматы операторов языка сборки

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

[метка:] имя оператора [,аргумент]...;

То есть, оператор обязательно содержит мнемоническое имя оператора и заканчивается точкой с запятой. Перед именем оператора может стоять метка, которая отделяется двоеточием.

Вслед за именем оператора могут идти один или несколько аргументов, разделенных запятыми. Например:

```
TPL,6,2; ТЕРМ; М1:УПЕР,М2;
```

Допускаются аргументы трех типов.

Аргумент типа N - это целое неотрицательное число. Аргумент типа L - это некоторая метка. Аргумент типа S – произвольный 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.

Группа О. Для операторов из этой группы аргументы не требуются. Например:

```
СИМ; EST; ТЕРМЯ;
```

Группа N. Требуется один аргумент типа N. Например:

```
СТВ,25; MULE,15;
```

Группа NN. Требуется два аргумента типа N. Например:

```
УГР,25,4;
```

Группа L. Требуется один аргумент типа L. Например:

```
УПЕР,M1; АКТО,FUNC;
```

Группа S. Требуется один аргумент типа S. Например:

```
ЗНАЧ,A; NS,'ПСИ1'; ЗНАЧЯ,'1250';
```

Группа NL. Требуется один аргумент типа N и один аргумент типа L. Например:

```
АКТ,10,FUNC;
```

3.4 Общая структура интерпретатора языка сборки. Эмулятор

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 этом, 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: эмулятора и набора подпрограмм, каждая из которых реализует действие отдельного оператора.

Эмулятор выполняет следующие действия. Для очередного оператора языка сборки, эмулятор выбирает код этого оператора и определяет, к какой группе этот оператор принадлежит. Затем эмулятор 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`, а для группы NL – в переменные `N` и `L`. Затем эмулятор 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 эмулятор на метку `NEXTOP`.

Опишем эмулятор формально.

Будем считать, что в переменной СЧА перед обращением к эмулятору находится адрес очередного оператора языка сборки. Обозначим через NMBL, LBLL и SMBL длину аргументов типа N, L и S, соответственно. Через NMB, LBL и SMB обозначим функции, которые выбирают по указанному адресу аргументы типа N, L и S, соответственно. Будем полагать, что код оператора занимает в памяти столько же места, сколько аргумент типа N.

Функция OPGROOP определяет по коду оператора, к какой группе он принадлежит и вырабатывает один из адресов GRPO, GRPN, GRPNN, GRPL, GRPS или GRPNL, если оператор принадлежит к одной из групп O, N, NN, L, S или NL, соответственно. Функция OPENTRY по коду оператора вырабатывает адрес входа в подпрограмму, реализующую данный оператор.

При сделанных предположениях эмулятор описывается следующей программой:

```
NEXTOP   OPC = NMB(СЧА)
         СЧА = СЧА + NMBL
         GOTO  OPGROOP(OPC)
GRPO     GOTO  OPENTRY(OPC)
GRPN     N = NMB(СЧА)
         СЧА = СЧА + NMBL
         GOTO OPENTRY(OPC)
GRPNN    N = NMB(СЧА)
         СЧА = СЧА + NMBL
         M = NMB(СЧА)
         СЧА = СЧА + NMBL
         GOTO OPENTRY(OPC)
GRPL     L = LBL(СЧА)
         СЧА = СЧА + LBLL
         GOTO OPENTRY(OPC)
GRPS     S = SMB(СЧА)
         СЧА = СЧА + SMBL
         GOTO OPENTRY(OPC)
GRPNL    N = NMB(СЧА)
         СЧА = СЧА + NMBL
         L = LBL(СЧА)
         СЧА = СЧА + LBLL
         GOTO OPENTRY(OPC)
```

Вход в эмулятор 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 ПАР, ПР и СЛ, а длина аргумента типа S совпадает с размером поля КОД (см.п.3.2).

3.5 Таблица элементов

Основная задача синтаксического отождествления – проанализировать выражение, стоящее в поле зрения в области действия ведущего знака конкретизации и, в случае, если отождествление возможно, запомнить те адреса звеньев из поля зрения, которые могут потребоваться в дальнейшем, для замены левой части на правую.

Будем называть элементом предложения рефала любой входящий в него символ, скобку или свободную 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 (`ТЭ`). Чтобы не возникала терминологическая путаница, мы будем называть элементы массива `ТЭ` строками таблицы элементов.

Строки таблицы элементов нумеруются, начиная с нуля: `ТЭ[0]`, `ТЭ[1]`, `ТЭ[2]`,... 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 `ТЭ`, одному элементу левой части всегда будет соответствовать одна или две строки в `ТЭ`.

Если элемент левой части – символ, скобка или переменная символа, то в `ТЭ` заносится адрес сопоставляемого ему звена из поля зрения.

Если элемент – переменная терма или выражения, ему будет соответствовать две соседних строки в `ТЭ`. 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 в `ТЭ` 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 `НЭЛ`, `Г1` и `Г2`

Номером элемента левой части рефал-предложения назовем номер соответствующей ему строки в `ТЭ`. Если элементу соответствует 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 `НЭЛ` всегда содержит номер первой незаполненной строки в `ТЭ`. Каждый оператор отождествления заполняет определенное число строк в `ТЭ` и увеличивает значение `НЭЛ` на

соответствующую величину, так, чтобы она снова указывала на первую свободную строку в ТЭ.

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

В отличие от переменной НЭЛ, которая в процессе отождествления движется по таблице элементов, переменные Г1 и Г2 в процессе отождествления движутся по полю зрения. В каждый момент отождествления Г1 и Г2 содержат адреса каких-то двух звеньев из поля зрения.

Процесс отождествления организован так, что приступать к проектированию какого-либо элемента левой части можно лишь после того, как уже спроектирован хотя бы один из двух его соседних элементов. При этом получается, что между спроектированными элементами зияют еще не проанализированные дыры (рис.3.5), которые постепенно стягиваются и исчезают. При проектировании пары скобок исходная дыра всегда разбивается на две дыры меньшего размера.

В самом начале отождествления нам известно только два адреса: а именно, адреса ведущих знаков k и $\perp$. Поэтому все пространство между ними представляет собой одну большую дыру.

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

В процессе отождествления переменные Г1 и Г2 всегда установлены на левый и правый конец той дыры, в которую в данный момент проектируются элементы левой части (рис.3.5).

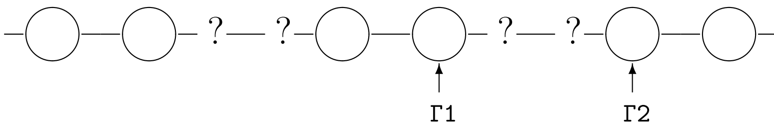


Рис. 3.5: Переменные Г1 и Г2, установленные на края дыры

Проектирование очередного элемента приводит к тому, что либо Г1 сдвигается вправо, либо Г2 сдвигается влево. Тем самым, дыра стягивается. Заметим, что при этом Г1 всегда остается левее, чем Г2.

Перед началом отождествления ведущие k и $\perp$ превращаются в структурные скобки. В таблице элементов заполняются три строки. В ТЭ[0] заносится адрес звена, предшествующего k, в ТЭ[1] – адрес k, а в ТЭ[2] – адрес $\perp$. Переменной НЭЛ присваивается значения 3. Г1 и Г2 устанавливаются на k и $\perp$ соответственно (рис.3.6). (Все эти действия выполняет оператор, который завершает очередной шаг и подготавливает следующий. Его описание содержится в п.3.25).

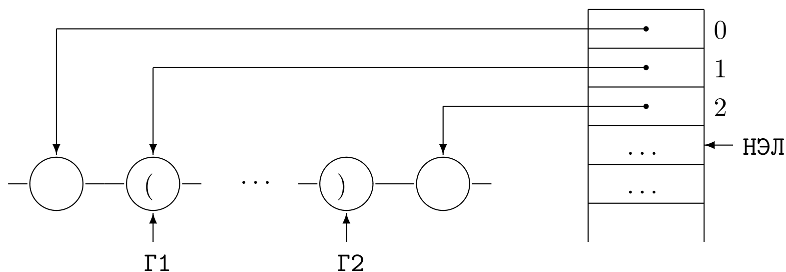


Рис. 3.6: Состояние ТЭ, НЭЛ, Г1 и Г2 перед началом шага

3.7 Язык звеньев

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

Программа на языке звеньев представляет из себя 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 содержит адрес звена.

Таким образом, предложение

```
СЛ(Г1) = ПАР(Г2)
```

означает, что в поле `СЛ` звена, адрес которого находится в переменной `Г1`, засылается содержимое поля `ПАР` звена, адрес которого находится в `Г2`.

Далее, нам понадобится предикат `СК(X)`, который вырабатывает значение “истина”, если переменная `X` содержит адрес звена, которое содержит скобку (левую или правую), и вырабатывает “ложь”, если `X` указывает на символ.

Аналогично, предикаты `СКЛ(X)` и `СКП(X)` распознают левую и правую скобки, соответственно.

Кроме того, в программах на языке звеньев будут употребляться следующие сокращения (“макрокоманды”):

`СДГ1` – сдвиг `Г1`, сокращение для

```
Г1 = СЛ(Г1)
IF(Г1 EQ Г2) GOTO НЕОТ
```

`СДГ2` – сдвиг `Г2`, сокращение для

```
Г2 = ПР(Г2)
IF(Г1 EQ Г2) GOTO НЕОТ
```

`RETURN` – возврат в эмулятор, сокращение для

```
GOTO NEXTOP
```

3.8 Отождествление объектных выражений

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

Оператор `ЗНАЧ,S;` относится к группе S. Он проектирует символ `S`, стоящий в левой части предложения на символ `S` в поле зрения.

При обращении к оператору `ЗНАЧ,S;` границы `Г1` и `Г2` должны уже быть установлены на края какой-то дыры (рис.3.7).

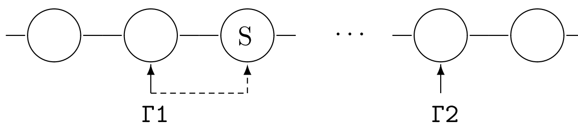


Рис. 3.7: Положение границ до применения оператора `ЗНАЧ,S;` и после (пунктирная стрелка)

Оператор `ЗНАЧ,S;` пытается передвинуть `Г1` на одно звено вправо. Если при этом `Г1` совпадет с `Г2`, это означает, что проектирование символа `S` потерпело неудачу, поскольку между `Г1` и `Г2` – пустое выражение.

Если значение `Г1` не совпадает с `Г2`, то `ЗНАЧ,S;` проверяет, содержит ли звено, на которое теперь указывает `Г1`, символ `S` или нет. Если содержит – проектирование успешно. Оператор заносит в `ТЭ[НЭЛ]` адрес этого звена, а значение `НЭЛ` увеличивает на единицу. Если не это звено не содержит символ `S`, то проектирование неуспешно.

Всякий раз, когда некоторый оператор отождествления обнаруживает, что проектирование потерпело неудачу, управление передается на метку `НЕОТ` (неотождествление). Что делает `НЕОТ` будет описано позднее, в п.3.10.

Ниже приведено описание оператора `ЗНАЧ,S;` на языке звеньев.

```
ЗНАЧ   СДГ1
       IF(КОД(Г1) NE S) GOTO НЕОТ
       ТЭ[НЭЛ] = Г1
       НЭЛ = НЭЛ + 1
       RETURN
```

Оператор `ПРОВ;` относится к группе O. Он не заполняет `ТЭ` и не сдвигает `НЭЛ`, `Г1` и `Г2`. Единственная его задача – проверить, что `Г1` и `Г2` установлены на соседние звенья, т.е. что между ними ничего нет. Если между `Г1` и `Г2` что-то есть, `ПРОВ;` передает управление на метку `НЕОТ`. Таким образом, `ПРОВ;` производит проектирование пустого выражения.

```
ПРОВ   IF(СЛ(Г1) NE Г2) GOTO НЕОТ
       RETURN
```

Имея только два оператора: `ЗНАЧ,S;` и `ПРОВ;` мы уже можем перевести на язык сборки левые части следующего вида:

§ k'F' А ∼ ...
§ k'F' Ж А Б А ∼ ...

В самом деле, расставляя над второй левой частью номера элементов, мы получим:

```
 1       3  4  5  6   2
§ k 'F'  Ж  А  Б  А   ∼     ...
```

(Название функции 'F' не занимает в поле зрения ни одного звена, поэтому при отождествлении не учитывается).

Проектирование произведет такая последовательность операторов:

```
ЗНАЧ,Ж; ЗНАЧ,А; ЗНАЧ,Б; ЗНАЧ,А; ПРОВ;
```

Интересно отметить, что при работе этих операторов `НЭЛ`, `Г1` и `Г2` "сами собой" передвигаются, нужным образом, и "автоматически" заполняются нужные строки в `ТЭ`.

Оператор `ЗНАЧЯ,S;` относится к группе S. Он "зеркально-симметричен" оператору `ЗНАЧ,S;`.

```
ЗНАЧЯ СДГ2
      IF(КОД(Г2) NE S) GOTO НЕОТ
      ТЭ[НЭЛ] = Г2
      НЭЛ = НЭЛ + 1
      RETURN
```

Используя оператор ЗНАЧЯ,S; можно по-другому скомпилировать последний пример

$$\S\, k\ \texttt{'F'}\ \overset{4}{\text{Ж}}\ \overset{6}{\text{А}}\ \overset{5}{\text{Б}}\ \overset{3}{\text{А}}\ \overset{2}{\sim}\quad \ldots$$

(над знаком § стоит номер 1)

в последовательность операторов:

```
ЗНАЧЯ,А; ЗНАЧ,Ж; ЗНАЧЯ,Б; ЗНАЧ,А; ПРОВ;
```

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

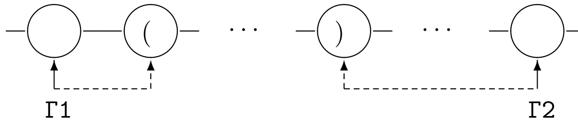


Рис. 3.8: Движение границ при работе оператора СКОБ;

Оператор СКОБ; относится к группе 0. Он проектирует пару скобок, стоящую в левой части предложения на пару скобок в поле зрения (рис.3.8).

СКОБ; сдвигает Г1 на одно звено вправо, проверяет, не "занято" ли уже оно Г2, а затем проверяет наличие в этом звене левой скобки. Если левая скобка есть, ее адрес заносится в ТЭ[НЭЛ]. Затем, по адресу, хранящемуся в поле ПАР, СКОБ; находит парную правую скобку и ее адрес заносит в ТЭ[НЭЛ + 1]. После этого Г2 устанавливается на правую скобку, а НЭЛ сдвигается на две единицы. Таким образом, СКОБ; проектирует сразу два элемента и заполняет две строки в ТЭ.

```
СКОБ  СДГ1
      IF(NOT CK(Г1)) GOTO НЕОТ
      Г2 = ПАР(Г1)
      ТЭ[НЭЛ] = Г1
      ТЭ[НЭЛ + 1] = Г2
      НЭЛ = НЭЛ + 2
      RETURN
```

Оператор СКОБЯ; относится к группе 0. Он осуществляет отождествление пары скобок, стоящей перед Г2 (рис.3.9).

```
СКОБЯ СДГ2
      IF(NOT CK(Г2)) GOTO НЕОТ
      ТЭ[НЭЛ + 1] = Г2
      Г2 = ПАР(Г2)
      ТЭ[НЭЛ] = Г2
      НЭЛ = НЭЛ + 2
      RETURN
```

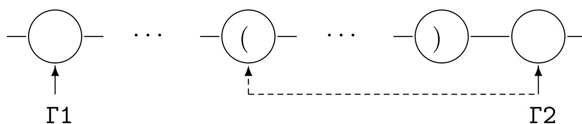


Рис. 3.9: Движение границ при работе оператора СКОБЯ;

Видно, что оператор СКОБЯ; не является “зеркальным” отражением оператора СКОБ;. Оба оператора разбивают исходную дыру на две дыры меньшего размера. Асимметричность СКОБ; и СКОБЯ; проявляется в том, что оба оператора устанавливают Г1 и Г2 на левую из образовавшихся дыр, адрес левой скобки заносят в ТЭ[НЭЛ], а адрес правой – в ТЭ[НЭЛ + 1]. Как мы увидим, это окажется удобным в дальнейшем.

Оператор УГР,N,M; относится к группе NN. Он предназначен для того, чтобы переставлять границы Г1 и Г2 с одной дыры на другую. При этом в Г1 заносится адрес из ТЭ[N] , а в Г2 заносится адрес из ТЭ[M].

```
УГР  Г1 = ТЭ[N]
     Г2 = ТЭ[M]
     IF(ТЭ[M] NE 0) RETURN
     Г2 = СЛ(ТЭ[M + 1])
     RETURN
```

В приведенном описании УГР,N,M; на языке звеньев, пока непонятны третье и четвертое предложения. Зачем они нужны станет ясно в п.3.9.

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

```
 1      3 5 6 7 4 8  2
§ k 'F' ( X ( ) ) Y  ~   ...

СКОБ; ЗНАЧ,X; СКОБ; ПРОВ; УГР,7,4;
ПРОВ; УГР,4,2; ЗНАЧ,Y; ПРОВ;
```

Если изменить очередность проектирования элементов левой части, получаем другой перевод на язык сборки:

```
 1      4 6 7 8 5 3  2
§ k 'F' ( X ( ) ) Y  ~   ...

ЗНАЧЯ,У; СКОБ; ЗНАЧ,X; СКОБЯ;
ПРОВ; УГР,7,8; ПРОВ; УГР,5,3; ПРОВ;
```

3.9 Проектирование свободных переменных

При проектировании свободных переменных символа не возникает никаких дополнительных проблем.

Оператор `СИМ;` относится к группе 0. Он проектирует главное вхождение переменной символа и работает аналогично оператору `ЗНАЧ,S;`. Различие заключается только в том, что `СИМ;` проверяет наличие после `Г1` произвольного символа, в то время как `ЗНАЧ,S;` проверял наличие конкретного символа `S`.

```
СИМ   СДГ1
      IF(СК(Г1)) GOTO НЕОТ
      ТЭ[НЭЛ] = Г1
      НЭЛ = НЭЛ + 1
      RETURN
```

Оператор `СИМЯ;` относится к группе 0. Он является “зеркальным” отражением оператора `СИМ;`.

```
СИМЯ  СДГ2
      IF(СК(Г2)) GOTO НЕОТ
      ТЭ[НЭЛ] = Г2
      НЭЛ = НЭЛ + 1
      RETURN
```

Теперь можно скомпилировать такую левую часть:

$$\S\ \overset{1}{k}\ \texttt{'F'}\ \ \overset{3}{s_x}\ \ \overset{4}{s_y}\ \ \overset{2}{\sim}\quad \dots$$

```
СИМ; СИМ; ПРОВ;
```

Если изменить очередность проектирования, получаем другой перевод:

$$\S\ \overset{1}{k}\ \texttt{'F'}\ \ \overset{4}{s_x}\ \ \overset{3}{s_y}\ \ \overset{2}{\sim}\quad \dots$$

```
СИМЯ; СИМ; ПРОВ;
```

Операторы `СТС,N;` и `СТСЯ,N;` (“старый символ”) относятся к группе N. Они используются для проектирования повторных вхождений переменных символа. Аргумент `N` – это номер главного вхождения переменной символа. Эти операторы работают аналогично `СИМ;` и `СИМЯ;`, но делают проверку на совпадение полей `КОД` у главного и повторного вхождений.

```
СТС   СДГ1
      IF(КОД(Г1) NE КОД(ТЭ[N])) GOTO НЕОТ
      ТЭ[НЭЛ] = Г1
      НЭЛ = НЭЛ + 1
      RETURN
```

```
СТСЯ  СДГ2
      IF(КОД(Г2) NE КОД(ТЭ[N])) GOTO НЕОТ
      ТЭ[НЭЛ] = Г2
      НЭЛ = НЭЛ + 1
      RETURN
```

Теперь можно скомпилировать такую левую часть:

$$\overset{1}{\S\, k}\ \texttt{'F'}\ \overset{3}{s_x}\ \overset{4}{s_x}\ \overset{2}{\sim}\quad \ldots$$

```
СИМ; СТС,3; ПРОВ;
```

Однако, ее можно скомпилировать и так (сохраняя те же номера элементов, но не сохраняя вид движения Г1 и Г2):

```
СИМ; СТСЯ,3; ПРОВ;
```

Теперь приступим к проектированию закрытых и повторных вхождений переменных выражения и переменных терма.

Оператор ЗАКР; относится к группе 0. Он производит проектирование закрытой переменной выражения.

Для каждого вхождения переменной выражения в ТЭ отводится две строки, где запоминаются адреса начального и конечного звена выражения. Если выражение пустое, то вместо адреса начала в ТЭ записывается нуль, а вместо адреса конца – адрес звена, предшествующего выражению.

Оператор ЗАКР; учитывает различие между пустым и непустым выражением.

```
ЗАКР    IF(СЛ(Г1) EQ Г2) GOTO ЗАКР1
        ТЭ[НЭЛ] = СЛ(Г1)
        ТЭ[НЭЛ + 1] = ПР(Г2)
        НЭЛ = НЭЛ + 2
        RETURN
ЗАКР1   ТЭ[НЭЛ] = 0
        ТЭ[НЭЛ + 1] = Г1
        НЭЛ = НЭЛ + 2
        RETURN
```

Другие операторы тоже должны учитывать различие между пустыми и непустыми выражениями. Теперь можно вернуться к описанию оператора УГР,N,M; в п.3.8 и разобраться, зачем в него вставлена проверка: ТЭ[M] EQ 0.

Теперь мы можем скомпилировать такую левую часть:

$$\overset{1}{\S\, k}\ \texttt{'F'}\ \overset{3}{s_x}\ \overset{5,6}{e_1}\ \overset{4}{s_x}\ \overset{2}{\sim}\quad \ldots$$

```
СИМ; СТСЯ,3; ЗАКР;
```

Здесь над e_1 проставлены два номера: 5 и 6. Это вызвано тем, что e_1 занимает две строки в ТЭ. В ТЭ[5] будет адрес начала e_1, а в ТЭ[6] – адрес конца.

Оператор СТВ,N; ("старое выражение") относится к группе N. Он проектирует повторное вхождение переменной выражения.

Аргумент N является номером главного вхождения проектируемой переменной, аналогично оператору СТС,N;. Разница только в том, что выражение занимает две строки в ТЭ, и в качестве аргумента используется номер второй из них.

Работает СТВ,N; следующим образом. Сначала из ТЭ[N - 1] и ТЭ[N] он находит адреса начала и конца значения главного вхождения. Затем Г1 начинает двигаться вправо. При этом каждое очередное звено сравнивается с соответствующим звеном главного вхождения. Если произойдет несовпадение хотя бы для одной пары звеньев, СТВ,N; терпит неудачу. Проектирование считается успешно законченным, когда все звенья главного вхождения исчерпаны. В этот момент Г1 как раз установлен на правый конец повторного вхождения (рис.3.10).

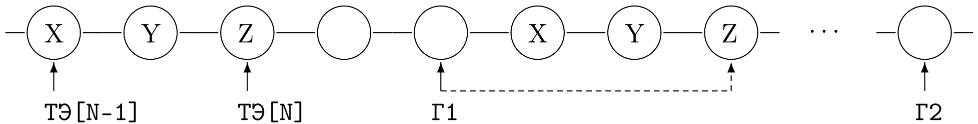


Рис. 3.10: Действие оператора СТВ,N;

```
СТВ   IF(ТЭ[N-1] EQ 0) GOTO СТВ2
      ТЭ[НЭЛ] =СЛ(Г1)
      АСТ = ПР(ТЭ[N - 1])
СТВ1  IF(АСТ EQ ТЭ[N]) GOTO СТВ3
      АСТ = СЛ(АСТ)
      СДГ1
      IF(КОД(Г1) EQ КОД(АСТ)) GOTO СТВ1
      IF(NOT СК(Г1)) GOTO НЕОТ
      IF(П(Г1) EQ П(АСТ)) GOTO СТВ1
      GOTO НЕОТ
СТВ2  ТЭ[НЭЛ] = 0
СТВ3  ТЭ[НЭЛ + 1] = Г1
      НЭЛ = НЭЛ + 2
      RETURN
```

Оператор СТВЯ,N; относится к группе N. Он является "зеркальным отражением" оператора СТВ,N;.

```
СТВЯ   ТЭ[НЭЛ + 1] = ПР(Г2)
       IF(ТЭ[N - 1] EQ 0) GOTO СТВЯ2
```

```
        АСТ = СЛ(ТЭ[N])
СТВЯ1   IF(АСТ = ТЭ[N - 1]) GOTO СТВЯ3
        АСТ = ПР(АСТ)
        СДГ2
        IF(КОД(Г2) EQ КОД(АСТ)) GOTO СТВЯ1
        IF(NOT СК(Г2)) GOTO НЕОТ
        IF(П(Г2) EQ П(АСТ)) GOTO  СТВЯ1
        GOTO  НЕОТ
СТВЯ2   ТЭ[НЭЛ] = 0
        НЭЛ = НЭЛ + 2
        RETURN
СТВЯ3   ТЭ[НЭЛ] = Г2
        НЭЛ = НЭЛ + 2
        RETURN
```

Теперь можно перевести на язык сборки такие левые части:

$$\overset{1}{\S\, k}\ \text{'F'}\ \overset{3}{(}\ \overset{5,6}{e_a}\ \overset{4}{)}\ \overset{7,8}{e_a}\ \overset{2}{\sim}\quad \ldots$$

```
СКОБ; ЗАКР; УГР,4,2; СТВ,6; ПРОВ;
```

$$\overset{1}{\S\, k}\ \text{'F'}\ \overset{3}{(}\ \overset{5,6}{e_a}\ \overset{4}{)}\ \overset{9,10}{e_1}\ \overset{7,8}{e_a}\ \overset{2}{\sim}\quad \ldots$$

```
СКОБ; ЗАКР; УГР,4,2; СТВЯ,6; ЗАКР;
```

Операторы ТЕРМ; и ТЕРМЯ; относятся к группе 0. Они проектируют главное вхождение переменной терма. Их действие аналогично действию операторов СИМ; и СИМЯ;, однако они заполняют две строки в ТЭ.

```
ТЕРМ    СДГ1
        ТЭ[НЭЛ] = Г1
        IF(NOT СК(Г1))GOTO ТЕРМ1
        Г1 = ПАР(Г1)
ТЕРМ1   ТЭ[НЭЛ + 1] = Г1
        НЭЛ = НЭЛ + 2
        RETURN

ТЕРМЯ   СДГ2
        ТЭ[НЭЛ + 1] = Г2
        IF(NOT СК(Г2)) GOTO ТЕРМЯ1
        Г2 = ПАР(Г2)
ТЕРМЯ1  ТЭ[НЭЛ] = Г2
        НЭЛ = НЭЛ + 2
        RETURN
```

Теперь можно скомпилировать такую левую часть:

$$\begin{array}{cccccccc} 1 & & 3,4 & 7,8 & 5,6 & 2 & \\ \S\, k & \texttt{'F'} & t_x & e_1 & t_y & \sim & \ldots \end{array}$$

```
ТЕРМ; ТЕРМЯ; ЗАКР;
```

А как быть, если требуется спроектировать повторное вхождение терма? В этом случае можно использовать уже рассмотренные операторы `СТВ,N;` и `СТВЯ,N;`, поскольку информация, заносимая в `ТЭ` для терма ничем не отличается от информации, заносимой для выражения.

$$\begin{array}{cccccccc} 1 & & 3,4 & 7,8 & 5,6 & 2 & \\ \S\, k & \texttt{'F'} & t_x & e_1 & t_x & \sim & \ldots \end{array}$$

```
ТЕРМ; СТВЯ,4; ЗАКР;
```

Пользуясь рассмотренными операторами мы уже можем скомпилировать левую часть любой сложности, лишь бы она не содержала открытых переменных выражения.

Например:

$$\begin{array}{cccccccccccccccccc} 1 & & 3 & 5 & 7,8 & 6 & 11,12 & 9 & 13,14 & 10 & 4 & 15,16 & 17,18 & 19,20 & 24,25 & 22,23 & 21 & 2 & \\ \S\, k & \texttt{'F'} & (& (& e_1 &) & e_2 & (& e_3 &) &) & e_1 & t_x & t_x & e_4 & t_x & s_y & \sim & \ldots \end{array}$$

```
СКОБ; СКОБ; ЗАКР; УГР,6,4;
СКОБЯ; ЗАКР; УГР,9,10; ЗАКР;
УГР,4,2; СТВ,8; ТЕРМ; СТВ,18;
СИМЯ; СТВЯ,18; ЗАКР;
```

Проектирование открытых переменных выражения представляет собой несколько более сложную задачу, так как тут уже не удастся обойтись простым последовательным выполнением операторов языка сборки.

3.10 Проектирование открытых вхождений переменных выражения

Чтобы обеспечить проектирование открытых вхождений переменных выражения, интерпретатор использует так называемый стек переходов (`СП`). Стек переходов представляет из себя одномерный массив, элементы которого нумеруются начиная с нуля: `СП[0]`, `СП[1]`, `СП[2]` и т.д. Имеется переменная `ГП` ("Глубина перехода"), которая в процессе интерпретации языка сборки указывает на первую свободную строку `СП`. Перед началом отождествления `ГП` присваивается значение 0. Каждая строка `СП` состоит из четырех полей: `Г1`, `Г2`, `НЭЛ` и `СЧА` (рис.3.11), поэтому нам удобнее представлять `СП` в виде четырех массивов: `СПГ1`, `СПГ2`, `СПНЭЛ` и `СПСЧА`. Каждое из полей `Г1`, `Г2` и `СЧА` должно вмещать произвольный аргумент типа L, а поле `НЭЛ` – произвольный аргумент типа N.

Дальше, в программах на языке сборки нам понадобятся следующие сокращения:

`ЗПСП(L1,L2,N,L3)` – запись в `СП`, сокращение для

Рис. 3.11: Структура стека переходов

```
СПГ1[ГП] = L1
СПГ2[ГП] = L2
СПНЭЛ[ГП] = N
СПСЧА[ГП] = L3
```

СЧСП(L1,L2,N,L3) – считывание из СП, сокращение для

```
L1 = СПГ1[ГП]
L2 = СПГ2[ГП]
N = СПНЭЛ[гП]
L3 = СПСЧА[ГП]
```

Теперь мы можем описать на языке звеньев подпрограмму НЕОТ, на которую передают управление операторы отождествления, когда проектирование терпит неудачу:

```
НЕОТ  IF(ГП EQ 0) GOTO RCGIMP
      ГП = ГП - 1
      СЧСП(Г1,Г2,НЭЛ,СЧА)
      RETURN
```

Отсюда видно, что последствия передачи управления на НЕОТ зависят от того, какая информация была ранее занесена в СП. В частности, в СП находится адрес того оператора, который будет работать после НЕОТ. Если в момент обращения к НЕОТ ГП равно 0, то управление передается на подпрограмму RCGIMP, которая вызывает аварийный останов интерпретатора языка сборки: "отождествление невозможно".

Теперь манипулируя с СП, мы можем организовать проектирование открытых вхождений переменных.

Открытое вхождение переменной проектируют операторы ПУД; ("подготовить удлинение") и УД; ("удлинение"). Эти операторы относятся к группе 0 и употребляются только совместно, в следующей комбинации:

```
ПУД; УД;
```

Действие их проще описать (и прочитать!) на языке звеньев, чем передать словами.

Оператор ПУД; заносит в СП текущие значения Г1, Г2, НЭЛ и СЧА. Напоминаем, что в момент работы оператора СЧА уже продвинут на следующий оператор, поэтому в СП окажется адрес оператора УД;!

Затем ПУД; заполняет ТЭ[НЭЛ] и ТЭ[НЭЛ + 1] так, чтобы они соответствовали пустому выражению, продвигает СЧА на оператор, следующий за УД; и возвращает управление в эмулятор. Таким образом, дальше будет исполняться не УД;, а следующий за ним оператор.

Если один из последующих операторов передаст управление на НЕОТ, то НЕОТ восстановит Г1, Г2, НЭЛ и СЧА, в результате чего управление попадет в УД;.

При каждом входе в УД;, этот оператор так изменяет ТЭ[НЭЛ + 1], чтобы к выражению, на которое указывают ТЭ[НЭЛ] и ТЭ[НЭЛ + 1] добавился один терм справа. Если это возможно, УД; увеличивает глубину СП и возвращает управление в эмулятор, если же невозможно, УД; терпит неудачу, т.е. передает управление на НЕОТ.

```
ПУД   ЗПСП(Г1,Г2,НЭЛ,СЧА)
      Г1 = ГП + 1
      ТЭ[НЭЛ] = 0
      ТЭ[НЭЛ + 1] = Г1
      НЭЛ = НЭЛ + 2
      СЧА = СЧА + NMBL
      RETURN

УД    IF(ТЭ[НЭЛ] NE 0) GOTO  УД1
      ТЭ[НЭЛ] = СЛ(Г1)
УД1   Г1 = ТЭ[НЭЛ + 1]
      СДГ1
      IF(NOT СК(Г1)) GOTO УД2
      Г1 = ПАР(Г1)
УД2   ГП = ГП + 1
      ТЭ[НЭЛ + 1] = Г1
      НЭЛ = НЭЛ + 2
      RETURN
```

Теперь можно скомпилировать такие левые части:

$$\overset{1}{\S\, k}\ \texttt{'F'}\ \overset{3,4}{e_1}\ \overset{5}{s_x}\ \overset{6,7}{e_2}\ \overset{2}{\sim}\ \ \ldots$$

ПУД;УД;СИМ;ЗАКР;

$$\overset{1}{\S\, k}\ \texttt{'F'}\ \overset{3,4}{e_1}\ \overset{5}{s_x}\ \overset{6,7}{e_2}\ \overset{8}{s_x}\ \overset{9,10}{e_3}\ \overset{2}{\sim}\ \ \ldots$$

ПУД; УД; СИМ; ПУД; УД; СТС,5; ЗАКР;

3.11 Передача управления с одного предложения на другое

Если в процессе проектирования какой-то оператор языка сборки потерпел неудачу, управление передается, либо на некоторый оператор `УД;`, либо на следующее рефал-предложение. Для управления переходами с одного предложения на другое используется оператор `УПЕР,L;` (“установка перехода”).

Оператор `УПЕР,L;` относится к группе L. Он описывается на языке звеньев следующим образом:

```
УПЕР  ЗПСП(Г1,Г2,НЭЛ,L)
      ГП = ГП + 1
      RETURN
```

Простейший способ употребления оператора `УПЕР,L;` состоит в следующем. Допустим, что нам нужно перевести на язык сборки функцию `'FUNC'`, которая содержит n предложений. Тогда перевод функции `'FUNC'` можно сделать следующим образом:

`FUNC:` `УПЕР,`L_1; перевод предложения 1
L_1: `УПЕР,`L_2; перевод предложения 2
. . .
L_{n-2}: `УПЕР,`L_{n-1}; перевод предложения $n-1$
L_{n-1}: перевод предложения n

Но можно функцию `'FUNC'` перевести и другим способом:
`FUNC:` `УПЕР,`L_{n-1}; `УПЕР,`L_{n-2}; . . . `УПЕР,`L_1; перевод предложения 1
L_1: перевод предложения 2
...
L_{n-2}: перевод предложения n-1
L_{n-1}: перевод предложения n

Второй способ перевода хуже, чем первый, ибо он зря забивает стек переходов. Кроме того, если отождествление получится уже в первом предложении, окажется, что операторы `УПЕР,`L_{n-1}; `УПЕР,`L_{n-2}; . . .`УПЕР,`L_2; выполнялись зря.

Оператор `УПЕР,L;` дает возможность оптимизировать программы на языке сборки за счет объединения совпадающих частей различных предложений.

Поясним сказанное на конкретном примере. Рассмотрим функцию, состоящую из двух предложений.

$$\S \overset{1}{k} \texttt{'F'} \;\overset{3,4}{t_x}\; \overset{7,8}{e_1}\; \overset{6}{\text{A}}\; \overset{5}{s_z}\; \overset{2}{\sim} \quad \ldots$$

$$\S \overset{1}{k} \texttt{'F'} \;\overset{3,4}{t_x}\; \overset{7,8}{e_1}\; \overset{6}{\text{B}}\; \overset{5}{s_z}\; \overset{2}{\sim} \quad \ldots$$

```
F:    УПЕР,L1;
      ТЕРМ;            L1: ТЕРМ;
```

```
СИМЯ;              СИМЯ;
ЗНАЧЯ,A;           ЗНАЧЯ,B;
ЗАКР;              ЗАКР;
...                ...
```

Видно, что перевод обеих предложений начинается операторами `ТЕРМ; СИМЯ;`. Следовательно, когда управление попадает на второе предложение, операторы `ТЕРМ; СИМЯ;` будут выполняться зря, ибо будут повторять ту работу, которую уже сделали операторы `ТЕРМ; СИМЯ;` в первом предложении. Оператор `УПЕР,L;` позволяет провести оптимизацию программы на языке сборки.

```
F: ТЕРМ;
   СИМЯ;
   УПЕР,L1;
   ЗНАЧЯ,A;         L1: ЗНАЧЯ,B;
   ЗАКР;                ЗАКР;
   ...                  ...
```

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.

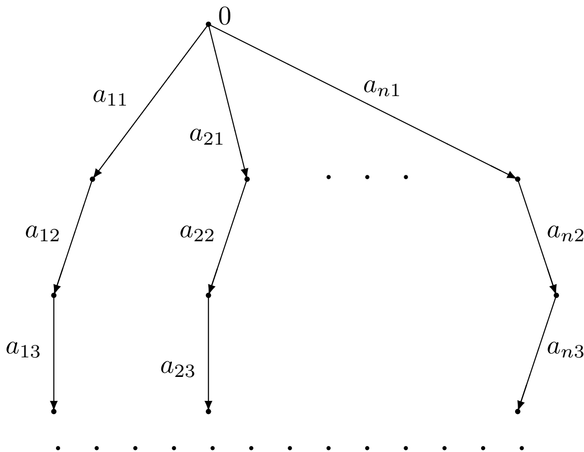


Рис. 3.12: Представление функции на языке сборки в виде дерева

Пусть нам дана функция, состоящая из n предложений. Переведем каждое предложение на язык сборки независимо. Обозначим через a_{ij} j-тый оператор

языка сборки в переводе i-го предложения. Затем построим упорядоченное дерево с выделенным корнем 0, изображенное на рис.3.12. Каждая дуга дерева помечена некоторым оператором a_{ij}

Это дерево показывает, как должно передаваться управление при работе операторов отождествления.

Если оператор успешно произвел проектирование, 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 преобразуем дерево, изображенное на рис. 3.12.

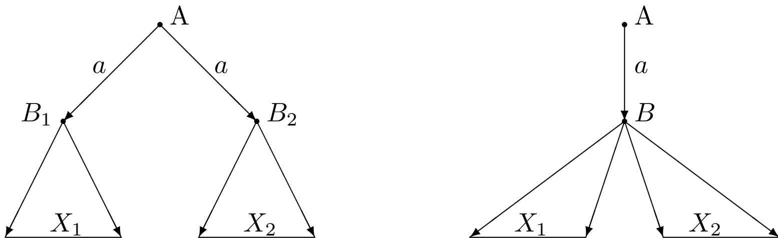


Рис. 3.13: Объединение совпадающих операторов языка сборки

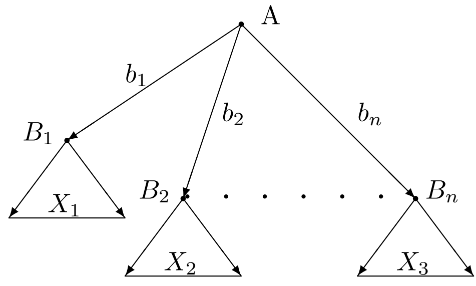


Рис. 3.14: Общий вид вершины дерева после оптимизации

Пусть из некоторого узла A выходят две дуги, намеченные одним и тем

же оператором a, причем между ними нет других дуг (рис.3.13). Пусть эти дуги направлены в узлы B_1 и B_2, к которым "подвешены" поддеревья X_1 и X_2, изображенные треугольниками.

Тогда, если оператор a не является оператором `ПУД;` или `УД;`, мы можем слить два узла B_1 и B_2 в один узел B, к которому подвешены поддеревья X_1 и X_2 без нарушения взаимного порядка, т.е. сначала идут дуги, принадлежащие X_1, а потом – дуги, принадлежащие X_2.

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 `УПЕР,L;`.

Пусть из некоторого узла A выходит n дуг, которые направлены в узлы $B_1, B_2, \ldots, B_n$. Каждая дуга AB_i помечена оператором b_i, а к каждому узлу B_i подвешено поддерево X_i.

Тогда перевод поддерева, с корнем в узле A выглядит следующим образом:

	`УПЕР,`L_1;	b_1;	перевод поддерева X_1
L_1:	`УПЕР,`L_2;	b_2;	перевод поддерева X_2
...			
L_{n-2} :	`УПЕР,`L_{n-1};	$b_{n-1;}$	перевод поддерева X_{n-1}
L_{n-1} :		b_n;	перевод поддерева X_n

В частности, при $n = 1$, эта конструкция вырождается в b_1; перевод поддерева X_1.

При описании оптимизации мы сделали важную оговорку, касающуюся операторов `ПУД;` и `УД;`. Их объединять 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 `КУД,N;`

Оператор `КУД,N;` относится к группе N. Он имеет следующее описание на языке звеньев:

```
КУД  ГП = ГП - N
     RETURN
```

Чтобы понять его назначение, рассмотрим следующий пример:

```
 1       3,4  5  6,7  8  9,10   2
§ k 'F'  e1   +  e2   *  e3     ~     ...

ПУД; УД; ЗНАЧ,+; ПУД; УД; ЗНАЧ,*; ЗАКР;
```

Пусть ведущая область конкретизации имеет вид:

k`'F'` $\underbrace{+ + + \ldots +}_{n \text{ раз}} \perp$

Как будет происходить отождествление? Сначала e_1 примет значение "пусто", а символ + в левой части спроектируется на первый + в ведущей области конкретизации. Затем e_2 будет удлиняться $n-1$ раз в поисках символа *. Когда дальнейшее удлинение e_2 станет невозможно, удлинится e_1, после чего e_2 будет удлиняться $n-2$ раз и т.д. Кода удлинение e_1 станет невозможно, все предложение терпит неудачу.

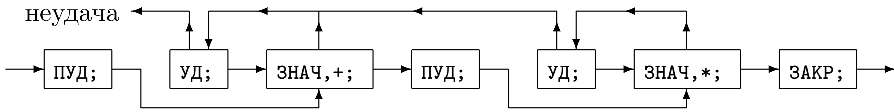


Рис. 3.15: Схема переходов между операторами языка сборки.

На рис.3.15 изображена схема переходов между операторами. Стрелки, выходящие из операторов горизонтально, показывают, куда происходит переход в случае успеха, а выходящие вертикально – куда происходит переход в случае неудачи.

Очевидно, что второй оператор `УД;` проработает

$$(n-1)+(n-2)+\ldots+1 = n(n-1)/2$$

раз, а первый `УД;` будет работать n раз. Поэтому общий объем работы можно оценить как $\approx n^2/2$.

Между тем, совершенно очевидно, что если выражение не содержит символ *, то и любая его часть не содержит *. Следовательно, удлинять e_1 нет смысла. Поэтому, если второй `УД;` терпит неудачу, то можно сразу же объявить, что все предложение терпит неудачу.

Оператор `КУД,N;` позволяет выразить это на языке сборки следующим образом:

```
ПУД; УД; ЗНАЧ,+; КУД,1;
ПУД; УД; ЗНАЧ,*; ЗАКР;
```

Получается схема переходов, изображенная на рис.3.16.

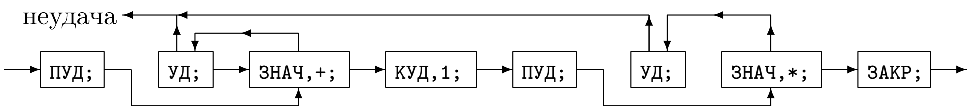


Рис. 3.16: Схема переходов после добавления оператора `КУД,1;`.

Теперь первый `УД;` не будет исполняться ни разу, а второй `УД;` проработает только $n-1$ раз. Таким образом, объем работы можно оценить как n. Оптимизированная программа работает примерно в $n/2$ раз быстрее. При $n = 20$ скорость работы возрастает на порядок.

Усложним пример. Пусть теперь функция `'F'` содержит два предложения.

k`'F'` e_1 + e_2 $*$ e_3 ∼ ...
k`'F'` e_1 + e_2 $*$ A ∼ ...

К каждому из этих предложений применимы рассуждения из предыдущего примера. Однако, теперь возможна дополнительная оптимизация, за счет объединения операторов из различных левых частей. Легко видеть, что перевод и первого и второго предложения начинается с операторов

```
ПУД; УД; ЗНАЧ,+; КУД,1;
```

Эта группа операторов либо находит +, либо вся в целом терпит неудачу, Если + найден, то последующие операторы, в случае неудачи, не возвращаются на УД; а сразу переходят на следующее предложение. Поэтому вся группа

```
ПУД; УД; ЗНАЧ,+; КУД,1;
```

в целом обладает такими же свойствами, как "обычные" операторы вроде `СИМ;` `ТЕРМ;` и т.д. Ее можно рассматривать как "составной оператор" языка сборки.

Если рассматривать группу `ПУД; УД;` как "левую скобку", а оператор `КУД,1;` как "правую скобку", то последовательность операторов отождествления можно рассматривать как "выражение", которое представляет собой последовательность "термов", где каждый "терм" это либо оператор, отличный от операторов `ПУД;`, `УД;` и `КУД,1;`, либо снова "выражение" заключенное в "скобки" `ПУД; УД;` и `КУД,1;`.

Теперь видно, как усовершенствовать алгоритм оптимизации, описанный в п.3.11. Раньше он не мог объединять операторы `ПУД; УД;`, поскольку каждая дуга дерева была помечена одним оператором. Однако, можно слегка изменить дерево, чтобы каждая дуга была помечена "термом" в описанном выше смысле. Т.е. последовательность дуг

мы заменим на одну дугу, помеченную "составным оператором". Теперь можно объединять дуги с одинаковыми пометками без ограничений.

Рассматриваемый пример следующим образом переведется на язык сборки:

```
F:   ПУД; УД; ЗНАЧ,+; КУД,1; УПЕР,L1;
     ПУД; УД; ЗНАЧ,*; ЗАКР;
L1:  ПУД; УД; ЗНАЧ,A; ЗАКР;
```

3.13 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 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.

Все операции, проводимые операторами преобразования над полем зрения, сводятся к тому, чтобы некоторый участок поля зрения либо породить, либо скопировать, либо переставить в другое место, либо уничтожить.

Хотя все указанные преобразования производятся над полем зрения, мы можем, для наглядности, представлять себе дело так, будто преобразуется не поле зрения, а некоторое выражение, содержащее свободные переменные, которое мы будем называть "объектом". Сначала объект совпадает с левой частью рефал-предложения, затем, подвергаясь преобразованиям, он постепенно переходит в правую часть соответствующего рефал-предложения. Эту правую часть мы будем называть "целью".

Такой подход оправдан в том смысле, что преобразования, производимые над объектом изображают в "алгебраическом" виде соответствующие операции над полем зрения. Поэтому в дальнейшем мы будем говорить о перестановке, копировании и уничтожении свободных переменных, хотя в действительности все эти операции интерпретатор языка сборки будет проделывать над их значениями.

3.14 Роль переменной Г в процессе замены

Перед началом замены ведущей области конкретизации объект полностью совпадает с левой частью рефал-предложения. В процессе замены объект проходит ряд промежуточных состояний, постепенно преобразуясь. В конце замены он должен полностью совпасть с правой частью предложения.

В каждый момент замены число элементарных преобразований, имеющихся в нашем распоряжении, сравнительно невелико. Все они сводятся к созданию, копированию, перестановке или уничтожению элементов. Но зато уж очень велик произвол в их применении. Один и тот же объект можно преобразовать в заданную цель, используя самые различные последовательности преобразований.

Если представить компилятору полную свободу в выборе преобразований, будет очень трудно различать, какое преобразование приближает объект к цели, какое удаляет от нее. Поэтому желательно наложить на применяемые преобразования такое ограничение, чтобы каждое примененное преобразование заведомо хоть на сколько-то приближало нас к цели.

Ограничение, которое мы наложим, будет заключаться в следующем. Мы потребуем, чтобы в любой момент замены в объекте можно было указать положение некоторой границы, обладающей следующими свойствами.

Та часть объекта, которая стоит слева от границы, должна полностью сов-

падать с левым концом цели. Эта часть объекта в процессе преобразования уже не меняется. Та часть объекта, которая находится справа от границы, еще не совпадает с целью, и будет преобразовываться.

Каждое элементарное преобразование должно продвигать границу хотя бы на один элемент вправо. Это гарантирует нам “сходимость” процесса, ибо если цель содержит n элементов, процесс преобразования потребует не более чем n элементарных преобразований.

Переменная `Г` (“граница”), как раз и играет роль такой разграничительной черты. Она всегда установлена на правый конец того элемента объекта, которым заканчивается часть объекта, совпадающая с целью (рис.3.17).

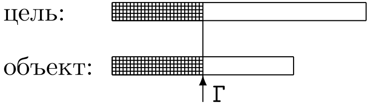


Рис. 3.17: Положение переменной `Г`. Совпадающие части объекта и цели заштрихованы.

Поясним высказанные положения конкретным примером, в котором, чтобы отвлечься от несущественных деталей, элементы объекта и цели будут обозначаться латинскими буквами.

Пусть нам дан объект

$a\ b\ c\ d\ e\ f$

который нужно преобразовать в цель

$b\ c\ a$

цель:	f	b	c	a		
объект:	a	b	c	d	e	f
	f	a	b	c	d	e
	f	b	c	a	d	e
	f	b	c	a		

Рис. 3.18: Преобразование объекта в цель.

На рис.3.18 показан один из возможных способов проделать это преобразование.

Видно, что в самом начале преобразования граница находится перед самым первым элементом объекта, ибо цель начинается с f , а объект с a , т.е. уже первые элементы не совпадают, Первое преобразование заключается в том, что мы переставляем f в начало объекта, что можно истолковать и как перемену

местами смежных строк $a\ b\ c\ d\ e$ и f. После этого у объекта и цели совпали первые элементы и граница сдвигается вправо на один элемент. Затем, мы переставляем участок $b\ c$, вставляя его перед границей (меняем местами a и $b\ c$). Теперь с целью совпали целых четыре элемента $f\ b\ c\ a$, поэтому продвигаем границу на целых три элемента. Теперь вся цель совпала с левым концом объекта, поэтому последнее преобразование заключается в том, что мы уничтожаем лишние элементы $d\ e$, после чего объект полностью совпадает с целью.

3.15 Список свободной памяти

В каждый момент работы интерпретатора языка сборки все звенья, не использованные в поле зрения, связаны в список, называемый списком свободной памяти (ССП).

Когда требуется вставить новые звенья в поле зрения, они берутся из ССП. Когда требуется убрать какие-то звенья из поля зрения, они присоединяются к ССП.

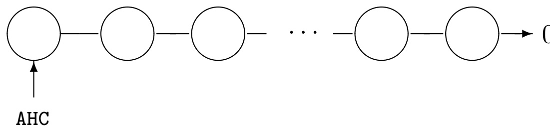


Рис. 3.19: Организация списка свободной памяти.

Список свободных звеньев (рис.3.19) организован как симметрический список. Каждое звено, входящее в ССП, в поле ПР содержит ссылку на предшествующее звено, а в поле СЛ – на следующее. Исключением являются только первое и последнее звено ССП. Для первого звена значение поля ПР не определено. Последнее звено в поле СЛ содержит нуль, что служит признаком конца ССП. Значение поля КОД для всех звеньев, входящих в ССП, не определено.

Переменная АНС используется в интерпретаторе языка сборки для того, чтобы постоянно указывать на начальное звено ССП. Выборка звеньев из ССП производится посредством “макрокоманды” ВСВ.

ВСВ – выборка свободного звена, сокращение для:

```
АНС = СЛ(АНС)
IF(АНС EQ 0) GOTO FLLE
```

где FLLE – вход в подпрограмму, которая вызывает аварийный останов интерпретатора “свободная память исчерпана”.

В дальнейшем,при описании операторов преобразования, нам понадобятся следующие сокращения.

СШИВ(X,Y) – сшить список, сокращение для:

```
P1 = X
P2 = Y
СЛ(P1) = P2
ПР(P2) = P1
```

где P1 и P2 – рабочие переменные. СШИВ можно истолковывать и как замкнутую подпрограмму, в которой параметры вызываются по значению.

СВЯЗ(X,Y) – связать пару скобок, сокращение для:

```
P1 = X
P2 = Y
П(P1) = ПСКЛ
П(P2) = ПСКП
ПАР(P1) = P2
ПАР(P2) = P1
```

где ПСКЛ и ПСКП – значения поля П, которые соответствуют левой и правой скобке.

3.16 Порождение объектных выражений

В этом разделе рассматриваются операторы, которые позволяют породить и вставить в поле зрения произвольное объектное выражение.

Оператор NS,S; относится к группе S. Он вставляет новый символ S в поле зрения следующим образом. Сначала NS,S; берет одно звено из списка свободной памяти, затем заносит S в поле КОД этого звена и вставляет это звено в поле зрения сразу вслед за тем звеном, на которое указывает Г, после чего Г устанавливается на вставленное звено.

```
NS     КОД(АНС) = S
ILINK  ПРК = СЛ(Г)
       СШИВ(Г,АНС)
       ВСВ
       СШИВ(Г,ПРК)
       RETURN
```

Оператор BL; относится к группе O. Он вставляет в поле зрения левую скобку после Г, затем Г передвигается на эту скобку.

Оператор BR; относится к группе 0. Действует он так же как BL;, но вставляет в поле зрения не левую, а правую скобку.

Каждая скобка в поле зрения должна содержать в поле ПАР 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 поле зрения еще не вставлены парные правые скобки, связываются в однонаправленный список (рис.3.20). Для этого используются поля ПАР в этих скобках. В переменной АСК постоянно хранится адрес начала этого списка.

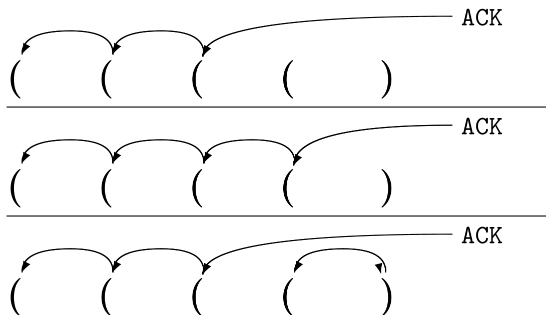


Рис. 3.20: Связывание парных скобок.

Когда в поле зрения 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.

Описанный метод связывания скобок лучше, чем тот, который использовался в рефал-интерпретаторе [26,27], поскольку не расходуется дополнительная память под стек и не накладывается никаких ограничений на глубину вложенности скобок.

Описание BL; и BR; на языке звеньев использует метку ILINK, определенную в описании оператора NS,S;.

```
BL     ПАР(АНС) = АСК
       АСК = АНС
       GOTO ILINK

BR     СЛСК = ПАР(АСК)
       СВЯЗ(АСК,АНС)
       АСК = СЛСК
       GOTO ILINK
```

С помощью операторов NS,S;, BL; и BR; мы можем порождать любые объектные выражения. Например выражение:

```
A ( B ( ) ) C
```

порождается последовательностью операторов

```
NS,A; BL; NS,B; BL; BR; BR; NS,C;
```

3.17 Копирование свободных переменных

Оператор MULS,N; относится к группе N. Он копирует (“размножает”) значение переменной символа, которой соответствует строка N в таблице элементов.

Копия значения переменной вставляется сразу после Г, затем Г передвигается на вставленное звено.

```
MULS КОД(АНС) = КОД(ТЭ[N)
     GOTO ILINK
```

Оператор MULE,N; относится к группе N. Он работает аналогично оператору MULS,N;, однако, предназначен для копирования значений переменных выражения и терма. При этом, в ТЭ[N-1] находится адрес начала копируемого выражения, а в ТЭ[N] – адрес его конца.

```
MULE    IF(ТЭ[N-1] EQ 0) GOTO RETURN
        АСТ = ТЭ[N-1]
        ПРК = СЛ(Г)
        СШИВ(Г,АНС)
        GOTO MULE2
MULE1   АСТ = СЛ(АСТ)
MULE2   Г = АНС
        ВСВ
        IF(СК(АСТ)) GOTO MULE4
        КОД(Г) = КОД(АСТ)
MULE3   IF(АСТ NE ТЭ[N]) GOTO MULE1
        СШИВ(Г,ПРК)
        RETURN
MULE4   IF(СКП(АСТ)) GOTO MULE5
        ПАР(Г) = АСК
        АСК = Г
        GOTO MULE1
MULE5   СЛСК = ПАР(АСК)
        СВЯЗ(АСК,Г)
        АСК = СЛСК
        GOTO MULE3
```

3.18 Перестановка участков объекта

Оператор TPL,N,M; относится к группе NN. Он производит перестановку ("трансплантацию") участка объекта в другое место (рис.3.21).

Представляемый участок называется "трансплантантом". Начало и конец трансплантанта задаются с помощью аргументов N и M. ТЭ[N] указывает на звено, предшествующее первому звену трансплантанта, а ТЭ[M] указывает на последнее звено трансплантанта. Если ТЭ[N] равно ТЭ[M], считается, что трансплантант пуст.

Трансплантант удаляется из поля зрения и вставляется на новое место, сразу после Г, после чего Г передвигается на правый конец трансплантанта.

```
TPL  IF(ТЭ[N] EQ ТЭ[M]) RETURN
```

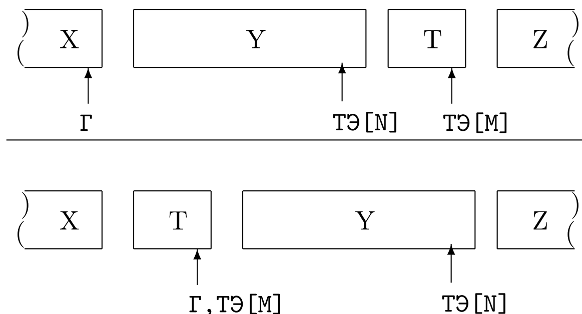


Рис. 3.21: Трансплантация участка Т.

```
ПРК = СЛ(ТЭ[N])
СШИВ(ТЭ[N],СЛ(ТЭ[M]))
СШИВ(ТЭ[M],СЛ(Г))
СШИВ(Г,ПРК)
Г = ТЭ[M]
RETURN
```

В тех случаях, когда трансплантант состоит только из одного элемента, удобнее использовать операторы `TPLS,N;` и `TPLE,N;`, которые относятся к группе N.

Оператор `TPLS,N;` применим в тех случаях, когда трансплантант является символом, скобкой или переменной символа.

```
TPLS    СШИВ(ПР(ТЭ[N]),СЛ(ТЭ[N]))
        СШИВ(ТЭ[N],СЛ(Г))
        СШИВ(Г,ТЭ[N])
        Г = ТЭ[N]
        RETURN
```

Оператор `TPLE,N;` применим в тех случаях, когда трансплантант является переменной выражения или терма.

```
TPLE  IF(ТЭ[N-1] EQ 0) RETURN
      СШИВ(ПР(ТЭ[N-1]),СЛ(ТЭ[N]))
      СШИВ(ТЭ[N-1],СЛ(Г))
      СШИВ(Г,ТЭ[N-1])
      Г = ТЭ[N]
      RETURN
```

3.19 Удаление участков объекта

Оператор `OUT,N;` относится к группе N. Он уничтожает участок объекта, который начинается после `Г` и заканчивается звеном, на которое указывает `ТЭ[N]`. Граница `Г` остается на месте.

```
OUT   IF(Г EQ ТЭ[N]) RETURN
      ПРК = СЛ(ТЭ[N])
      СШИВ(ТЭ[N],АНС)
      АНС = СЛ(Г)
      СШИВ(Г,ПРК)
      RETURN
```

Этот оператор можно многократно использовать во время замены. Однако, компилятор с рефала предпочитает `OUT,N;` в середине преобразования объекта в цель не использовать. Преобразование организовано так, что ненужные участки постепенно оттесняются в правый конец объекта, откуда они удаляются в самом конце преобразования объекта в цель. Это хорошо видно в примере, который разобран в п.3.14.

3.20 Принудительное продвижение границы

Все рассмотренные до сих пор операторы преобразования некоторым образом продвигали границу `Г` вправо. Обычно получается так, что каждый оператор продвигает `Г` как раз в ту позицию, которая требуется, чтобы применить следующий оператор. Однако, в некоторых случаях требуемое положение границ не совпадает с тем, в которое она была установлена последним оператором.

Оператор `УГ,N;` относится к группе N. Он загружает в переменную `Г` адрес, хранящийся в `ТЭ[N]`.

```
УГ    Г = ТЭ[N]
      RETURN
```

3.21 Пример преобразования объекта в цель

Теперь мы можем вернуться к примеру преобразования объекта в цель, рассмотренному в п. 3.14. Подставим вместо букв $a\,b\,c\,d\,e\,f$, конкретные элементы s_z, A, B, (,), t_x, которым присвоим номера 3, 4, 5, 6, 7 и 9. Таким образом, теперь мы имеем:

	3	4	5	6	7	8,9
объект:	s_z	A	B	(	)	t_x
цель:	t_x	A	B	s_z		

Легко убедиться, что требуемое преобразование проделает следующая последовательность операторов:

```
TPLE,9; TPL,3,5; УГ,3; OUT,7;
```

3.22 Проблема пустых выражений

Пусть нам дан объект (e_1) X и цель ( ) e_1. Очевидно, что преобразование объекта в цель можно выполнить следующим образом:

$$\overset{3}{(}\ \overset{5,6}{e_1}\ \overset{4}{)}\ \overset{7}{\mathrm{X}} \rightarrow \overset{3}{(}\ \overset{4}{)}\ \overset{5,6}{e_1}\ \overset{7}{\mathrm{X}} \rightarrow \overset{3}{(}\ \overset{4}{)}\ \overset{5,6}{e_1}$$

Этим преобразованиям соответствует следующая программа на языке сборки:

```
УГ,3; TPLS,4; УГ,6; OUT,7;
```

Если значение e_1 – непустое, эта программа работает правильно. А что получится, если e_1 – пустое?

Тогда, как мы договорились еще в п.3.9, вместо адреса конца выражения в ТЭ будет занесен адрес конца предшествующего элемента, а вместо адреса начала – нуль. Такое соглашение обеспечивает правильную работу операторов `УГ,N;` и `TPL,N,M;`, ибо их аргументами всегда являются номера правых концов элементов. К сожалению, когда мы начинаем переставлять элементы, среди которых есть выражения с пустыми значениями, это соглашение может нарушиться.

В нашем конкретном случае пустота e_1 приведет к тому, что в `ТЭ[6]` будет нуль, а в `ТЭ[6]` тот же адрес, что в ТЭ[3]. После применения операторов `УГ,3; TPLS,4;` значения `ТЭ[5]` и `ТЭ[6]` остаются прежними, поэтому выходит, будто e_1 осталось на старом месте, после левой скобки. Оператор `УГ,6;` установит границу на левую скобку, вместо того, чтобы установить ее на правую. Следовательно, `OUT,7;` уничтожит элементы) X, в результате чего в поле зрения останется последовательность элементов.

(e_1

где левая скобка ссылается на список свободной памяти.

Таким образом, следует особо позаботиться, чтобы в процессе преобразования поля зрения не нарушалось соглашение, принятое для пустых выражений. Этого можно добиться, если в подходящие моменты времени "подправлять" адреса, хранящиеся в ТЭ.

Оператор `CORA,N,M;` относится к группе NN. Он предназначен для коррекции правого конца выражения, в том случае, если это выражение пустое.

```
CORA  IF(ТЭ[M-1] NE 0) RETURN
      ТЭ[M] = ТЭ[N]
      RETURN
```

Таким образом, если выражение с номером `M` – пустое, адресом его правого конца становится адрес, хранящийся в `ТЭ[N]`.

Оператор `CORAO,N;` относится к группе N. Он предназначен для коррекции адреса правого конца выражения в том случае, если граница `Г` установлена на правый конец предшествующего элемента.

```
CORA0  IF(TЭ[N-1] NE 0) RETURN
       TЭ[N] = Г
       RETURN
```

Теперь, пользуясь оператором `CORA,N,M;` мы можем дать правильный перевод на язык сборки для рассматриваемого примера:

```
УГ,3; TPLS,4; CORA,4,6; УГ,6; OUT,7;
```

Рассмотрим другой пример, в котором потребуется оператор `CORA0,N;`

$$\overset{3}{X}\ \overset{4,5}{e_1}\ \overset{6}{Y} \to \overset{3}{X}\ \ A\ \overset{4,5}{e_1}\ \overset{6}{Y} \to \overset{3}{X}\ \ A\ \overset{4,5}{e_1}$$

```
УГ,3; NS,A; CORA0,5; УГ,5; OUT,6;
```

Здесь символу A не соответствуют какие-либо строки в `TЭ`, ибо он порождается оператором `NS,A;`. Но это не беда, ибо новосозданный символ сразу же оказывается слева от границы `Г` и в дальнейших преобразованиям не участвует.

Небольшое усложнение возникает, если нужно скорректировать адрес конца у переменной выражения, которой предшествует другая переменная выражения,

$$\overset{3}{X}\ \overset{4,5}{e_1}\ \overset{6,7}{e_2}\ \overset{8,9}{e_3}\ \overset{10}{Y} \to \overset{3}{X}\ \ A\ \overset{4,5}{e_1}\ \overset{6,7}{e_2}\ \overset{8,9}{e_3}\ \overset{10}{Y} \to \overset{3}{X}\ \ A\ \overset{4,5}{e_1}\ \overset{6,7}{e_2}\ \overset{8,9}{e_3}$$

```
УГ,3; NS,A; CORA0,5; CORA,5,7; CORA,7,9; УГ,9; OUT,10;
```

В этом случае, коррекция адресов сначала производится для предшествующей переменной выражения.

Сформулируем общий принцип, согласно которому следует вставлять операторы `CORA,N,M;` и `CORA0,N;` в текст на языке сборки.

Коррекцию адреса для некоторого вхождения переменной выражения производится, если непосредственно предшествующий элемент заменился на другой элемент, либо была произведена коррекция адреса для непосредственно предшествующего элемента. Если коррекцию адреса требуется произвести для двух элементов, то она делается прежде для того элемента, который стоит левее. Коррекция адреса производится только для тех элементов, которые стоят справа от границы `Г`.

Например, рассмотрим преобразование

$$\overset{3}{A}\ \overset{4,5}{e_1}\ \overset{6,7}{e_2}\ \overset{8,9}{e_3} \to \overset{3}{A}\ \overset{6,7}{e_2}\ \overset{4,5}{e_1}\ \overset{8,9}{e_3}$$

```
УГ,3; TPLE,7; CORA0,5; CORA,5,9;
```

Здесь, после применения оператора `TPLE,7;` изменился предшествующий элемент сразу у трех элементов: e_1, e_2 и e_3. Для e_2 коррекцию адреса производить не нужно, ибо e_2 стоит слева от `Г`. Для e_1 и e_3 коррекцию адреса делать нужно. Поскольку e_1 стоит левее, делаем коррекцию для e_1. Теперь для

e_3 нужно делать коррекцию сразу по двум причинам: изменился предшествующий элемент, и сделана коррекция адреса для предшествующего элемента. Если механически пользоваться сформулированным принципом, то коррекцию адресов придется делать чуть ли не после каждого преобразования поля зрения. Однако, если учесть, какие строки в `ТЭ` понадобятся в дальнейшем, а какие – нет, можно удалить из программы на языке сборки избыточные операторы `CORA,N,M;` и `CORAO,N;`, резко сократив их количество.

Будем говорить, что `ТЭ[`m`]` используется в операторе языка сборки, если этот оператор имеет следующий вид: `УГ,`m`;`, `TPL,`m,n`;`, `TPL,`n,m`;`, `OUT,`m`;` или `CORA,`n,m`;`, где n – произвольное число.

Оператор `CORA,`n,m`;` (`CORAO,`m`;`) можно удалить в одном из следующих случаев.

1. Если в операторах, следующих за `CORA,`n,m`;` (`CORAO,`m`;`) не используется `ТЭ[`m`]`, то этот оператор можно удалить.

2. Если в операторах, заключенных между `CORA,`n,m`;` (`CORAO,`m`;`) и `CORA,`n',m`;` или `CORAO,`m`;` не используется `ТЭ[`m`]`, то первый из этих операторов можно удалить.

Алгоритм оптимизации заключается в том, что мы просматриваем программу на языке сборки от конца к началу до оператора `CORA,`n,m`;` или `CORAO,`m`;` и пытаемся исключить этот оператор. Затем просмотр продолжается.

3.23 Представление конкретизационных скобок в поле зрения

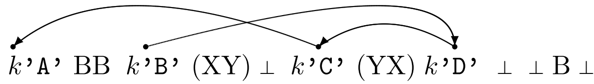
Согласно формальному описанию языка рефал, в начале каждого шага рефал-машина должна просматривать поле зрения в поисках ведущего знака конкретизации k . Ясно, что время такого ассоциативного поиска пропорционально размерам поля зрения, и при реализации рефала лучше постараться без него обойтись.

К счастью, можно придумать такое представление для k и $\perp$, что время поиска ведущей области конкретизации не будет зависеть от размеров поля зрения. Достигается это следующим образом.

Для знаков k , , входящих в поле зрения, можно ввести отношение линейного порядка, которое мы будем называть отношением старшинства.

Самым младшим мы объявим ведущий знак k . Следующим по старшинству будет тот знак k , который станет ведущим, если удалить из поля зрения ведущий знак k вместе с парной к нему точкой $\perp$. Продолжая этот процесс, распределим по старшинству все знаки конкретизации.

Таким образом, все знаки k , находящиеся в поле зрения, можно связать в однонаправленный список так, чтобы каждый знак k содержал ссылку на следующий за ним по старшинству. Например:



Этот список можно представить в машине по-разному.

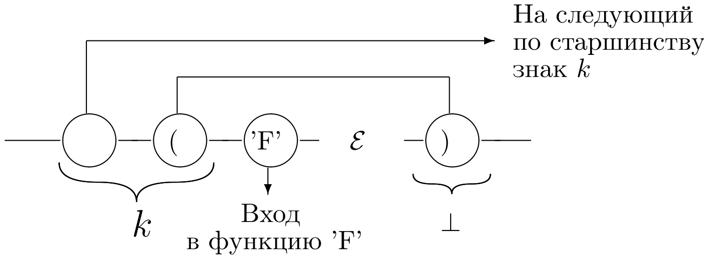


Рис. 3.22: Представление k`'F'` $\mathcal{E}\perp$ в рефал-интерпретаторе.

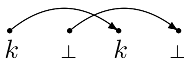
В рефал-интерпретаторе [26,27] каждый знак k представлялся с помощью двух звеньев (рис.3.22). Первое звено в поле `ПАР` содержало ссылку на следующий по старшинству знак k, а второе звено – ссылку на парную точку $\perp$. Точка представлялась в виде одного звена и содержала ссылку на парный знак k. Звено, следующее за знаком k, содержало в поле `ПАР` адрес начала некоторой функции `'F'`. Таким образом, под k, $\perp$ и `'F'` расходовалось 4 звена.

Во время замены ведущей области конкретизации в поле зрения могут вставляться новые знаки k. В этот момент их нужно сразу же связывать в однонаправленный список. Поскольку во время замены мы знаем адреса всех вставляемых знаков k мы можем связать их в список за время, которое зависит лишь от вида правой части предложения, но не от значений свободных переменных.

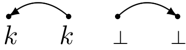
К сожалению, список, в который связаны знаки k, имеет запутанную структуру. Каждый знак k может ссылаться как вперед, т.е. на знак k, стоящий после него, так и назад, на знак k, стоящий перед ним. Поэтому, в рефал-интерпретаторе [26,27] использовался довольно сложный алгоритм связывания знаков k, для которого требовался стек.

А что получится, если рассмотреть отношение очередности не для знаков k, а для точек $\perp$? Будем считать, что точки упорядочены так же, как упорядочены парные к ним знаки k.

Легко доказать, что для любых двух точек $\perp$, стоящих в поле зрения, старшей является та точка, которая стоит правее. Действительно, рассмотрим любые две области конкретизации. Тогда они либо не перекрываются

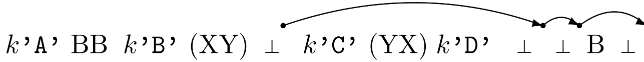


либо одна из них вложена в другую



И в том и в другом случае более правая точка соответствует старшему знаку k .

Таким образом, оказывается, что мы получим более простую картину, если свяжем в однонаправленный список не знаки, а точки $\perp$. Например:



Следующее улучшение заключается в том, чтобы снизить расход памяти на хранение k , $\perp$ и `'F'` в два раза: с четырех звеньев до двух. Мы можем сделать это, заметив, что за пределами ведущей области конкретизации можно временно нарушить "правильность" списка, ибо некоторые ссылки являются избыточными.

Действительно, имея адрес точки $\perp$, мы можем найти по содержимому поля `ПАР` парный знак k . Если отказаться от требования, чтобы знак k ссылался на парную точку, у нас освободится поле `ПАР` в знаке k , и мы можем поместить в него ссылку на функцию `'F'`.

Аналогично, имея адрес точки $\perp$, мы можем найти звено, следующее за ней, и в поле `ПР` этого звена поместить ссылку на следующую точку (рис.3.23).

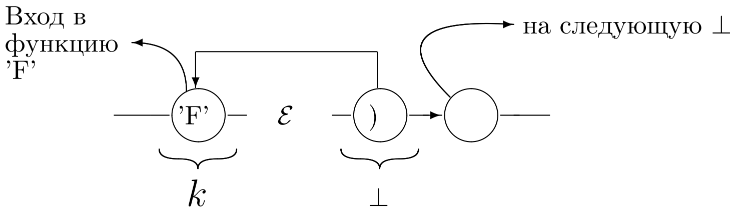


Рис. 3.23: Представление k`'F'` $\mathcal{E}\perp$ в интерпретаторе языка сборки

Перед началом очередного шага ведущие k и $\perp$ превращаются в обычные структурные скобки. При этом в поле `ПАР` звена, изображающего k и в поле `ПР` звена, следующего за $\perp$ помещаются ссылки на $\perp$. Тем самым, "правильность" списка восстанавливается.

3.24 Порождение конкретизационных скобок

Конкретизационные скобки порождаются в два приема. Сначала в поле зрения вставляются структурные скобки, а затем они активизируются, т.е. превращаются в конкретизационные скобки.

Оператор AKTO,L; относится к группе L. В момент применения этого оператора Г должна быть установлена на некоторую правую структурную скобку. Действие оператора выглядит так, как будто правая структурная скобка превращается в точку $\perp$, парная к ней левая скобка – в знак k, а после знака k вставляется символ-метка 'L'.

Так, например, выражение

k'F' A k'G' B $\perp$ C $\perp$

можно породить с помощью операторов:

```
BL; NS,A; BL; NS,B; BR; AKTO,G; NS,C; BR; ACTO,F;
```

Оператор AKT,N,L; относится к группе NL. Он работает аналогично оператору AKTO,L;, но активизирует правую скобку, адрес которой содержится в ТЭ[N].

Так, например, предложение:

§ $\overset{1}{k}$ 'F' $\overset{2}{\sim}$ k'ALPHA' k'BETA' $\perp$ $\perp$

можно следующим образом перевести на язык сборки:

```
ПРОВ; УГ,1; BL; BR; AKTO,BETA; AKT,2,ALPHA; EST;
```

Последний оператор EST; будет описан в п.3.25.

Для того, чтобы связывать точки $\perp$ в список, используются переменные ПРЕДТ и СЛЕДТ. Перед началом очередного шага в переменную СЛЕДТ загружается адрес точки, следующей за ведущей. Переменная ПРЕДТ содержит адрес точки, которая является предыдущей по отношению к точке, активизируемой в данный момент.

Возникает вопрос, какой адрес должна содержать ПРЕДТ в момент активизации самой первой точки? Ведь у нее нет предшествующей точки! Чтобы преодолеть это затруднение, интерпретатор языка сборки использует два особых звена: КВЗТ ("квазиточка") и СЛКВЗТ ("следующее за КВЗТ"). В поле СЛ звена КВЗТ постоянно хранится адрес звена СЛКВЗТ. Перед началом шага в переменную ПОСЛ загружается адрес звена КВЗТ.

```
AKTO  ПАР(ПАР(Г)) = L
      ПР(СЛ(ПРЕДТ)) = Г
      ПРЕДТ = Г
      RETURN

AKT   ПАР(ПАР(ТЭ[N])) = L
      ПР(СЛ(ПРЕДТ)) = ТЭ[N]
      ПРЕДТ = Г
      RETURN
```

3.25 Завершение очередного шага и подготовка следующего

Оператор `EST;` относится к группе 0. Он завершает очередной шаг рефал-машины и начинает выполнение нового шага.

Работа `EST;` начинается с того, что зашивается “дырка” в списке конкретизационных точек. После этого в поле `ПР` звена `СЛКВЗТ` оказывается адрес ведущей точки. Если этот адрес нулевой это означает, что в поле зрения не осталось ни одной точки Происходит нормальный останов рефал-машины “конкретизация выполнена”. Если адрес ведущей точки не нулевой, подготавливается следующий шаг.

`П` и `Г2` устанавливаются на ведущие k и $\perp$. `СЧА` устанавливается на начало функции, `СЛЕДТ` устанавливается на точку, следующую за ведущей, а `ПРЕДТ` – на звено `КВЗТ`.

Затем k и $\perp$ превращаются в структурные скобки, `ГП` устанавливается в начало стека переходов.

Наконец, заполняются `ТЭ[0]`, `ТЭ[1]` и `ТЭ[2]`, `НЭЛ` присваивается значение 3 и управление возвращается в эмулятор.

```
EST  ПР(СЛ(ПРЕДТ)) = СЛЕДТ
     Г2 = ПР(А((СЛКВЗТ))
     IF(Г2 EQ 0) GOTO ENDOFJOB
     Г1 = ПАР(Г2)
     СЧА = ПАР(Г1)
     СЛЕДТ = ПР(СЛ(Г1))
     ПРЕДТ = А(КВЗТ)
     ПР(СЛ(Г2)) = Г2
     ПАР(Г1) = Г2
     ГП = 0
     ТЭ[0] = ПР(Г1)
     ТЭ[1] = Г1
     ТЭ[2] = Г2
     НЭЛ = 3
     RETURN
```

3.26 Дополнительные операторы

Для удобства программирования, а также для экономии времени и памяти во время работы программы введены дополнительные операторы языка сборки.

Каждый из дополнительных операторов заменяет некоторую последовательность основных операторов языка сборки.

1. `ПУДЗН; УДЗН,S;` эквивалентно `ПУД; УД; ЗНАЧ,S;`

2. `ПУДСК; УДСК;` эквивалентно `ПУД; УД; СКОБ;`

3. ПУДЗН; УДСТС,N; эквивалентно ПУД; УД; СТС,N;

4. ИСК,S; эквивалентно ПУД; УД; ЗНАЧ,S; КУД,1;

5. BLR; эквивалентно BL; BR;

Каждому из этих операторов соответствует особая подпрограмма в интерпретаторе языка сборки, поэтому их выполнение требует меньше времени, чем выполнение эквивалентной последовательности основных операторов.

Поскольку дополнительные операторы не содержат ничего принципиально нового, мы опустим их описание на языке звеньев.

3.27 Примеры перевода функций на язык сборки

Теперь, для любой функции, описанной на рефале, мы можем дать ее полный перевод на язык сборки.

```
  1       3 5,6 4   2   1 3      5,6  4  2
§ k 'F'   (  e1  )  ~   ( k 'F'  e1   ⊥  )

  1       3 5,6 4  7  8,9   2   1       3 5,6  7  4 8,9  2
§ k 'F'   (  e1  ) sx  e2   ~   k 'F'   (  e1  sx ) e2   ⊥

  1       3,4   2  3,4
§ k 'F'   e1    ~  e1
```

```
На языке сборки после оптимизации получим:
F:  УПЕР,L2; СКОБ; ЗАКР; УГР,4,2;
    УПЕР,L1; ПРОВ; АКТ,4,F; EST;
L1: СИМ; ЗАКР; УГ,6; TPLS,7; АКТ,2,F; EST;
L2: ЗАКР; УГ,0; TPLE,4; OUT,2; EST;
```

Рассмотрим другой пример:

```
  1            3  4,5   2   1          4,5  2  3
§ k 'REV'      sx  e1   ~   k 'REV'    e1   ⊥  sx

  1          3 5,6 4 7,8   2   1          7,8  2  3           5,6     4
§ k 'REV'    (  e1 )  e2   ~   k 'REV'    e2   ⊥  (  k'REV'   e1    ⊥ )

  1          2
§ k 'REV'    ~
```

```
REV: УПЕР,L3; СИМ; ЗАКР; УГ,1; TPL,3,2;
     AKTO,REV; EST;

L3:  УПЕР,L4; СКОБ; ЗАКР; УГР,4,2; ЗАКР;
     УГ,1; TPL,4,2; AKTO,REV; УГ,3; BL;
     CORAO,6; УГ,6; BR; AKTO,REV; EST;
L4:  ПРОВ; УГ,0; OUT,2; EST;
```

Следует отметить, что оптимизация дает больший эффект не для таких простых примеров, а для сложных и громоздких предложений, где, как правило, есть большие участки, которые без (или почти без) изменения переходят из левой части в правую и где левые части различных предложений функции похожи друг на друга. В качестве примера можно привести функцию ПДР из рефал-компилятора.

$$
\begin{array}{l}
\S\, k\text{'ПДР'} \; \overset{3}{(} \; \overset{5,6}{e_c} \; \overset{4}{)} \; \overset{7}{s_h} \; \overset{8}{s_k} \; \overset{9}{(} \; \overset{11}{\text{'E'}} \; \overset{12}{s_x} \; \overset{15,16}{e_1} \; \overset{14}{\text{'E'}} \; \overset{13}{s_y} \; \overset{10}{)} \; \overset{19,20}{e_b} \; \overset{17}{(} \; \overset{21,22}{e_2} \; \overset{23}{s_x} \; \overset{24,25}{e_3} \; \overset{18}{)} \; \sim \\
\qquad (e_c)\; s_h\; s_k\; (\text{'E'}\; s_x\; e_1\; \text{'E'}\; s_y)\; e_b\; (e_2\; s_x\; e_3) \\
\S\, k\text{'ПДР'}\; (e_c)\; s_h\; s_k\; (\text{'E'}\; s_x\; e_1\; \text{'E'}\; s_y)\; e_b (e_2\; s_y\; e_3)\; \sim \\
\qquad (e_c)\; s_h\; s_k\; (\text{'E'}\; s_x\; e_1\; \text{'E'}\; s_y)\; e_b\; (e_2\; s_y\; e_3) \\
\S\, k\text{'ПДР'}\; (e_c)\; s_h\; s_k\; (\text{'E'}\; s_x\; e_1\; \text{'E'}\; s_y)\; e_2\; \sim \\
\qquad k\text{'ПДР'}\; (e_c\; s_h\; s_k\; (\text{'E'}\; s_x\; e_1\; \text{'E'}\; s_y)\;)\; e_2 \perp \\
\S\, k\text{'ПДР'}\; e_1\; \sim\; e_1
\end{array}
$$

После перевода на язык сборки и оптимизации получается:

```
ПДР:  УПЕР,L7; СКОБ; ЗАКР; УГР,4,2; СИМ; СИМ;
      СКОБ; ЗНАЧ,'E'; СИМ; СИМЯ; ЗНАЧЯ,'E';
      ЗАКР; УГР,10,2;
      УПЕР,L6; СКОБЯ; ЗАКР; УГР,17,18;
      УПЕР,L5; УДСТС,12; ЗАКР;
      УГ,0; TPL,1,18; OUT,2; EST;
L5:   УДСТС,13; ЗАКР;
      УГ,0; TPL,1,18; OUT,2; EST;
L6:   ЗАКР; УГ,6; TPL,4,10; АКТ,2,ПДР; EST;
L7:   ЗАКР; УГ,0; TPL,1,3; OUT,2; EST;
```

Хорошо видно, что хотя у всех предложений громоздкие правые части, замена делается быстро и просто. Левые части у всех предложений тоже громоздкие, однако, они похожи друг на друга, поэтому длинный перевод левой части получился только у первого предложения.

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 `УПЕР,L;` при этом производится объединение совпадающих частей различных предложений.

Общая структура компилятора показана на рис.4.1.

Алгоритм объединения совпадающих операторов левых частей предложений был описан в Главе 3.

Алгоритмы компиляции левой и правой частей предложения описываются в этой главе.

4.2 Компиляция левой части предложения

В этом разделе описан алгоритм компиляции левой части рефал- предложения на язык сборки, включая исключение лишних удлинений е-переменных с помощью операторов `КУД,N;`.

Этот алгоритм компиляции реализует алгоритм отождествления, который всегда дает те же результаты, что и правило отождествления, аксиоматически

определенное в формальном описании рефала (см. п.1.2), что следует из результатов работы [34] (см.приложение A, теоремы 10.1, 10.2, 10.3, 10.4, 10.5, 17.4, 17.5, 17.6). Алгоритм порождения операторов `КУД,N;` также опирается на результаты работы [34] (см.приложение A, теоремы 18.4, 19.4).

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 $\mathcal{L}$ (без k , $\sim$ и имени функции). Результатом компиляции является некоторая последовательность операторов отождествления.

Задача компилятора состоит в том, чтобы для каждого элемента $\mathcal{L}$, породить соответствующий оператор языка сборки.

Компилятор порождает операторы языка сборки строго в том порядке, в котором они должны стоять в результирующей программе. Поэтому в каждый момент работы компилятора можно указать, для каких элементов из $\mathcal{L}$ уже порождены соответствующие им операторы языка сборки, а для каких – еще нет.

Будем говорить, что некоторый элемент из $\mathcal{L}$ *спроектирован*, если для него уже порожден оператор отождествления, а под *проектированием* элемента из $\mathcal{L}$ будем подразумевать процесс порождения оператора, соответствующего этому элементу. (Таким образом, в этом разделе понятие "проектирование" обозначает действие производимое компилятором, а не интерпретатором языка сборки!).

Также будем говорить, что некоторая переменная из $\mathcal{L}$ *приняла значение*, если хотя бы одно ее вхождение в $\mathcal{L}$ уже спроектировано.

Когда часть элементов из $\mathcal{L}$ уже спроектирована, дальнейшая работа компилятора зависит от взаимного расположения только тех элементов $\mathcal{L}$, которые еще не спроектированы. Поэтому, компилятору нет необходимости хранить всю левую часть $\mathcal{L}$ целиком: достаточно иметь только те куски $\mathcal{L}$, которые еще не спроектированы.

Некоторое подвыражение $\mathcal{L}$ будем называть *дырой*, если оно обладает следующими свойствами:

а) все элементы этого выражения еще не спроектированы;

б) элементы, непосредственно примыкающие к этому выражению слева и справа, уже спроектированы.

Ясно, что две дыры не могут взаимно перекрываться, поскольку между ними обязательно находится по крайней мере один уже спроектированный элемент.

В процессе работы компилятор постоянно поддерживает упорядоченный список дыр, входящих в $\mathcal{L}$:

$$\mathcal{E}_1,\ \mathcal{E}_2,\ \ldots,\ \mathcal{E}_n$$

При этом, для каждой дыры $\mathcal{E}_i$, запоминается ее "левая граница" L_i и "правая граница" R_i. L_i - это номер *правого* конца элемента, непосредственно *предшествующего* $\mathcal{E}_i$ в $\mathcal{L}$, а R_i – это номер *левого* конца элемента, непосредственна *следующего* за $\mathcal{E}_i$ в $\mathcal{L}$.

Поэтому список дыр хранится в компиляторе в следующем виде:

$(\mathcal{D}_1,\ \mathcal{D}_2,\ \ldots,\ \mathcal{D}_n)$, где $\mathcal{D}_i = (L_i,\ \mathcal{E}_i,\ R_i)$

Перед началом компиляции список дыр имеет следующий вид:

$(\ (\ 1,\ \mathcal{L},\ 2\ )\ )$

то есть он содержит только одну дыру $\mathcal{L}$ – левую часть предложения, границы которой соответствуют знакам k и $\perp$, обрамляющим $\mathcal{L}$.

Во время компиляции дыры могут появляться и исчезать. Проектирование закрытой е-переменной приводит к исчезновению дыры, проектирование пары скобок приводит к раздроблению дыры на две дыры.

В тот момент, когда список дыр окажется пуст, компиляция левой части заканчивается.

Помимо списка дыр, компилятор поддерживает следующую информацию:

`ПП` - множество переменных, принявших значение. Кроме того, для каждой переменной $x \in$ `ПП` запоминается число `НП(`x`)` – номер правого конца главного вхождения переменной x.

`НЭЛ` – номер первой свободной строки в таблице элементов.

Перед началом компиляции устанавливаются начальные значения `ПП = 0`, `НЭЛ = 3`.

Для оптимизации удлинений е-переменных посредством операторов `КУД,N;` компилятор использует дополнительную информацию.

Прежде всего он следит за текущей глубиной стека переходов с помощью переменной `ГП`.

Перед началом работы устанавливается `ГП = 0`. Затем, каждый раз, когда порождается оператор `УД;` , значение `ГП` увеличивается: `ГП:= ГП + 1`. При каждом порождении оператора `КУД,`m`;` значение `ГП` уменьшается: `ГП:= ГП -` m.

Для генерации операторов `КУД,N;` используются два метода: разложение списка дыр на независимые классы (приложение А, теорема 18.4) и выделение удлиняющегося независимого выражения (приложение А, теорема 19.4).

Для изложения этих методов нагл потребуется следующее обозначение: пусть $\mathcal{E}$ – типовое выражение, тогда через $\|\mathcal{E}\|$ будет обозначаться множество переменных, входящих в $\mathcal{E}$. Например:

$\|s_x\ e_a\ (s_x\ e_1)\ t_b\| = \{s_x,\ e_a,\ e_1,\ t_b\}$

Разложение списка дыр на независимые классы производится следующим образом.

Пусть в какой-то момент работы компилятора образовался следующий список дыр:

$(\mathcal{D}_1,\ \mathcal{D}_2,\ \ldots\ \mathcal{D}_n)$, где $\mathcal{D}_i = (L_i,\ \mathcal{E}_i,\ R_i)$

Будем говорить, что две дыры $\mathcal{D}_i$ и $\mathcal{D}_j$ *непосредственно* зависимы, если либо $i < j$, либо $i \neq j$ и $\|\mathcal{E}_i\| \cap \|\mathcal{E}_j\| \nsubseteq$ `ПП`.

Отношение непосредственной зависимости рефлексивно и симметрично. Теперь рассмотрим транзитивное замыкание этого отношения, которое будем называть отношением зависимости. А именно, будем говорить, что две дыры $\mathcal{D}_i$ и $\mathcal{D}_j$ зависимы, если существует такая последовательность дыр: $\mathcal{D}_{k_1}$, $\mathcal{D}_{k_2}$, ..., $\mathcal{D}_{k_m}$ $(m \geq 1)$, что $\mathcal{D}_{k_i}$ непосредственно зависима от $\mathcal{D}_{k_{i-1}}$ для всех $i: \ 2 \leq i \leq m$

Отношение зависимости дыр является отношением эквивалентности. Поэтому оно разбивает список дыр на r непересекающихся классов:

$$\mathcal{K}_1, \ \mathcal{K}_2, \ \ldots, \ \mathcal{K}_r$$

Разложение списка дыр на независимые классы можно получить с помощью следующего простого алгоритма.

Алгоритм разложения на непересекающиеся классы.

Исходные данные: список дыр $(\mathcal{D}_1, \ \mathcal{D}_2, \ \ldots \mathcal{D}_n)$ и множество переменных ПП.

Результат работы: `N` – число классов и массив `КЛАСС(1)`, ..., `КЛАСС(`n`)`. Для каждой дыры $\mathcal{D}_i$ `КЛАСС(`i`)` дает номер класса, к которому принадлежит $\mathcal{D}_i$, $(i \leq$ `КЛАСС`$(i) \leq N)$.

К1. Установить `N:= 0; КЛАСС(`i`) = 0` для $1 \leq i \leq n$.

К2. Если `КЛАСС(`i`)` $\neq 0$ для всех $1 \leq i \leq n$, то закончить работу. Иначе, найти такое j , что `КЛАСС(`j`) = 0`. Установить `X:=` $\|\mathcal{E}_j\|$`; N:= N + 1; КЛАСС(`j`):= N`.

К3. Если `X`$\cap\|\mathcal{E}_i\| \subseteq$ `ПП` для всех $1 \leq i \leq n$, то перейти к шагу К2. Иначе, найти такое i, что `X` $\cup \|\mathcal{E}_i\| \not\subseteq$ `ПП`, установить `КЛАСС(`i`):= N; X:= X`$\cup\|\mathcal{E}_i\|$ и повторить шаг К3.

Компилятор следующим образом использует разложение на независимые классы дыр.

Пусть в некоторый момент компиляции список дыр принял следующий вид:

$$(\mathcal{D}_1, \ \mathcal{D}_2, \ \ldots, \ \mathcal{D}_n)$$

где дыры $\mathcal{D}_i$ разбиваются на r независимых классов $\mathcal{K}_1, \ \mathcal{K}_2, \ \ldots, \ \mathcal{K}_r$, где $r \geq 2$. Тогда компилятор строит r новых списков дыр

$$\mathcal{C}_1, \ \mathcal{C}_2, \ \ldots, \ \mathcal{C}_r$$

где $\mathcal{C}_k$ получается вычеркиванием из исходного списка дыр тех $\mathcal{D}_i$ для которых $\mathcal{D}_i \notin \mathcal{K}_k$.

Пусть текущее значение `ГП =` p. Теперь компилятор временно "забывает" о существовании списков $\mathcal{C}_2, \ \ldots, \ \mathcal{C}_r$ (отложив их в стек) и компилирует список дыр $\mathcal{C}_1$. Пусть после этого `ГП =` q. Тогда, если $q > p$, компилятор порождает оператор `КУД,`$q - p$`;` и восстанавливает `ГП =` p. После этого компилятор точно так же поступает со списками дыр $\mathcal{C}_2, \ \ldots, \ \mathcal{C}_r$.

Пример. `Пусть НЭЛ = 7, ПП = 0, ГП = 0`, а список дыр имеет вид:

$$(\ (5, \ e_1 \ s_x \ e_2, \ 6), \ (6, \ e_a \ + \ e_b, \ 4), \ (4, \ e_3 \ s_x \ e_4, \ 2) \ )$$

Компилятор разбивает список дыр на два списка:

$$\mathcal{C}_1 = (\ (5, \ e_1 \ s_x \ e_2, \ 6), \ (4, \ e_3 \ s_x \ e_4, \ 2) \ )$$
$$\mathcal{C}_2 = (6, \ e_a \ + \ e_b, \ 4)$$

Текущее значение `ГП = 0`. Компилируется $\mathcal{C}_1$:

```
УГР,5,6; ПУД; УД; СИМ; ЗАКР; УГР,4,2;
ПУД; УД; СТС,9; ЗАКР;
```

Теперь `ГП = 2`, поэтому порождается `КУД,2;` и восстанавливается `ГП = 0`. Затем компилируется $\mathcal{C}_2$:

```
УГР,6,4; ПУД; УД; ЗНАЧ,+; ЗАКР;
```

Теперь `ГП = 1`, поэтому порождается `КУД,1;` и восстанавливается `ГП = 0`.

Выделение независимого удлиняющегося выражения происходит следующим образом.

Пусть в какой-то момент работы компилятора образовался следующий список дыр:

$$(\mathcal{D}_1,\ \mathcal{D}_2,\ \ldots,\ \mathcal{D}_n),\ \text{где}\ \mathcal{D}_i = (L_i,\ \mathcal{E}_i,\ R_i)$$

Пусть при этом $\mathcal{E}_1 = e_x\ \mathcal{E}_a\ e_y\ \mathcal{E}_b$, где e_x, $\mathcal{E}_a$ и e_y удовлетворяют следующим условиям:

1. Каждая е-переменная, входящая в $\mathcal{E}_a$ на нулевом уровне скобок, принадлежит к множеству $\{e_x\} \cup$ ПП.

2. $$\|e_x\ \mathcal{E}_a\| \cap \|\mathcal{E}_b\ \mathcal{E}_2\ \mathcal{E}_3 \ldots \mathcal{E}_n\| \subseteq \text{ПП},$$

   т.е. переменные, входящие в $e_x\ \mathcal{E}_a$ и не принявшие значение не входят в выражения $\mathcal{E}_b,\ \mathcal{E}_2,\ \mathcal{E}_3, \ldots, \mathcal{E}_n$

3. $$e_y \notin \text{ПП} \cup \|e_x\ \mathcal{E}_a\| \cup \|\mathcal{E}_b\ \mathcal{E}_1\ \mathcal{E}_2\ \ldots \mathcal{E}_n\|,$$

   т.е. e_y входит в левую часть предложения $\mathcal{L}$ ровно один раз

Тогда будем говорить, что $\mathcal{E}_1$ начинается с независимого удлиняющегося выражения $e_x\ \mathcal{E}_a\ e_y$.

Компилятор следующим образом использует независимое удлиняющееся выражение.

Пусть текущее значение `ГП` = p. Компилятор временно прекращает проектирование каких либо элементов выражений $\mathcal{E}_1,\ \mathcal{E}_2,\ \ldots,\ \mathcal{E}_n$ за исключением элементов выражения $e_x\ \mathcal{E}_a$. Затем компиляция продолжается до тех пор, пока выражение $e_x\ \mathcal{E}_a$ не будет полностью спроектировано.

Пусть после этого `ГП` = q. Тогда, если $q > p$, компилятор порождает `КУД,`$q - p$`;` и восстанавливает `ГП` = p. После этого запрет на проектирование оставшихся элементов из списка дыр отменяется и компиляция продолжается.

Теперь мы можем закончить описание переменных, которые нужны компилятору для порождения операторов `КУД,N;`.

Для сохранения списков дыр, полученных в результате разложения списка дыр на независимые классы, используется стек списков дыр `ССД`. Для сохранения и восстановления значений переменной `ГП` используется стек `СГП`. Переменная `ГС` указывает текущую глубину стеков `ССД` и `СГП`.

Переменная ОП используется для того, чтобы "помечать" переменные e_y из независимых удлиняющихся выражений $e_x\ \mathcal{E}_a\ e_y$. ОП содержит множество "отмеченных" е-переменных. Кроме того, для каждой переменной e_x, входящей в ОП запоминается число ГПОП(e_x)) – значение ГП в тот момент, когда мы заносим e_x в ОП.

Теперь мы можем привести полный алгоритм компиляции левой части рефал-предложения.

Алгоритм компиляции левой части предложения.

Исходные данные: $\mathcal{E}$ – левая часть предложения.

Результат работы: последовательность операторов языка сборки.

КОМПЛЧ. (Начальные установки)
Установить СД:= ((1, $\mathcal{L}$, 2)); ГС:= 0; НЭЛ:= 3; ГП:= 0; ПП:= 0; ОП:= 0; Г1:= 1; Г2:= 2.

ИСД. (Искать дыру, из которой будет проектироваться очередной элемент). В этот момент СД $= (\mathcal{D}_1,\ \mathcal{D}_2,\ \ldots,\mathcal{D}_n)$, где $n \geq 0$. Установить i:= 1.

ИСД1. Если $i > n$, то перейти к ИСД4, иначе, рассмотреть дыру $\mathcal{D}_i = (L_i,\ \mathcal{E}_i,\ R_i)$.
Если $\mathcal{E}_i = e_y\ \mathcal{E}'_i$, где $e_y \in$ ОП, то перейти к ИСД2, иначе перейти к ИСД3.

ИСД2. (Вытолкнуть дыры $\mathcal{D}_i,\ \mathcal{D}_{i+1},\ \ldots,\ \mathcal{D}_n$ в ССД). Установить ГС:= ГС+1;
ССД(ГС):= $(\mathcal{D}_i,\ \mathcal{D}_{i+1},\ \ldots,\ \mathcal{D}_n)$; СГП(ГС):= ГПОП($e_y$); ОП:= ОП - $\{e_y\}$; СД:= $\mathcal{D}_1,\ \mathcal{D}_2,\ \ldots,\ \mathcal{D}_{i-1}$. Перейти к ИСД4.

ИСД3. Если $\mathcal{E}_i = e_x\ \mathcal{E}'_i\ e_y$, где $e_x \notin$ ПП и $e_y \notin$ ПП, то установить i:= $i+1$ и перейти к ИСД1, иначе перейти к ПРЭ.

ИСД4. В этот момент СД $=((\mathcal{D}_1,\ \mathcal{D}_2,\ \ldots,\ \mathcal{D}_n))$. Если $n > 0$, то перейти к РЗЛСД. Если $n = 0$ и ГС > 0, то породить КУД,ГП-СГП(ГС);, установить ГП:= СГП(ГС); СД:= ССД(ГС); ГС:= ГС-1; и перейти к ИСД.
Наконец, если $n = 0$ и ГС $= 0$, то все элементы $\mathcal{L}$ спроектированы и компиляция левой части предложения закончена.

ПРЭ. (Проектирование элемента). В этот момент $\mathcal{D}_i = (L_i,\ \mathcal{E}_i,\ R_i)$.
Если $L_i \neq$ Г1 или $R_i \neq$ Г2, то породить УГР,L_i, R_i.
Если $\mathcal{E}_i$ – пустое выражение, то перейти к ППУС.
Если $\mathcal{E}_i = e_x$, где $e_x \notin$ ПП, то перейти к ПЗАК.
Если $\mathcal{E}_i = (\mathcal{E}'_i)\ \mathcal{E}''_i$, то перейти к ПСКЛ.
Если $\mathcal{E}_i =$ Z $\mathcal{E}'_i$, где Z – символ, то перейти к ПСЛ.
Если $\mathcal{E}_i = s_x\ \mathcal{E}'_i$, то перейти к ППСЛ.
Если $\mathcal{E}_i = t_x\ \mathcal{E}'_i$, то перейти к ППТЛ.
Если $\mathcal{E}_i = e_x\ \mathcal{E}'_i$ и $e_x \in$ ПП, то перейти к ППВЛ.

Если $\mathcal{E}_i = \mathcal{E}'_i$ ($\mathcal{E}''_i$), то перейти к ПСКП.

Если $\mathcal{E}_i = \mathcal{E}'_i$ Z , где Z – символ, то перейти к ПСП.

Если $\mathcal{E}_i = \mathcal{E}'_i\ s_x$, то перейти к ППСП.

Если $\mathcal{E}_i = \mathcal{E}'_i\ t_x$, то перейти к ППТП.

Если $\mathcal{E}_i = \mathcal{E}'_i\ e_x$ и $e_x \in$ ПП, то перейти к ППВП.

(Теперь описывается проектирование элементов всех видов, за исключением открытых e-переменных).

ППУС. Породить `ПРОВ`; удалить $\mathcal{D}_i$ из СД и перейти к ИСД.

ПЗАК. Породить `ЗАКР`; удалить $\mathcal{D}_i$ из СД; установить ПП:=ПП $\cup \{e_x\}$; НП(e_x):= НЭЛ + 1; НЭЛ:= НЭЛ + 2; и перейти к ИСД.

ПСКЛ. Породить `СКОБ`; заменить $\mathcal{D}_i$ в СД на две дыры: (НЭЛ, $\mathcal{E}'_i$, НЭЛ + 1), (НЭЛ + 1, $\mathcal{E}''_i$, R_i); установить Г1:= НЭЛ; Г2:= НЭЛ + 1; НЭЛ:= НЭЛ + 2; перейти к ИСД.

ПСЛ. Породить `ЗНАЧ,Z`; перейти к Л1.

ППСЛ. Если $s_x \notin$ ПП, то породить `СИМ`; установить ПП:= ПП$\cup\{s_x\}$; НП(s_x):= НЭЛ. Иначе, породить `СТС,НП(`s_x`)`. Перейти к Л1.

ППТЛ. Если $t_x \notin$ ПП, то породить `ТЕРМ`; установить ПП:= ПП$\cup\{t_x\}$; НП(t_x):= НЭЛ иначе породить `СТВ,НП(`t_x`)`. Перейти к Л2.

ППВЛ. Породить `СТВ,НП(`e_x`)`; перейти к Л2.

ПСКП. Породить `СКОБЯ`; заменить $\mathcal{D}_i$ в СД на две дыры: (L_i, $\mathcal{E}'_i$, НЭЛ), (НЭЛ, $\mathcal{E}''_i$, НЭЛ + 1) установить Г1:= L_i; Г2:= НЭЛ; НЭЛ:= НЭЛ + 2; перейти к ИСД.

ПСП. Породить `ЗНАЧЯ,Z`; перейти к П1.

ППСП. Если $s_x \notin$ ПП, то породить `СИМЯ`; установить ПП:= ПП$\cup\{s_x\}$; НП(s_x): = НЭЛ иначе породить `СТСЯ,НП(`s_x`)`. Перейти к П1.

ППТП. Если $t_x \notin$ ПП, то породить `ТЕРМЯ`; установить ПП:= ПП$\cup\{t_x\}$; НП(t_x):= НЭЛ иначе породить `СТВЯ,НП(`t_x`)`. Перейти к П2.

ППВП. Породить `СТВЯ,НП(`e_x`)`; перейти к П2.

Л1. Заменить $\mathcal{D}_i$ в СД на (НЭЛ, $\mathcal{E}'_i$, R_i); установить Г1:= НЭЛ; Г2:= R_i; НЭЛ:= НЭЛ + 1; перейти к ИСД.

Л2. Заменить $\mathcal{D}_i$ в СД на (НЭЛ + 1, $\mathcal{E}'_i$, R_i); установить Г1:= НЭЛ + 1; Г2:= R_i; НЭЛ:= НЭЛ + 2; перейти к ИСД.

П1. Заменить $\mathcal{D}_i$ в СД на (L_i, $\mathcal{E}'_i$, НЭЛ); установить Г1:= L_i; Г2:= НЭЛ; НЭЛ:= НЭЛ + 1; перейти к ИСД.

П2. Заменить $\mathcal{D}_i$ в СД на (L_i, $\mathcal{E}'_i$, НЭЛ); установить Г1:= L_i; Г2:= НЭЛ; НЭЛ:= НЭЛ + 2; перейти к ИСД.

РЗЛСД. (Разложение списка дыр на независимые классы). В этот момент СД $=$ $(\mathcal{D}_1,\ \mathcal{D}_2,\ \ldots,\mathcal{D}_n)$. Разбить дыры из СД на непересекающиеся классы по отношению зависимости. Пусть получилось r классов: $\mathcal{K}_1,\ \mathcal{K}_2,\ \ldots,\ \mathcal{K}_r$. Если $r = 1$, то перейти к ВДЛНУВ. Если $r \geq 2$, построить r списков дыр $\mathcal{C}_1,\ \mathcal{C}_2,\ \ldots,\ \mathcal{C}_r$, где $\mathcal{C}_k$ получается из СД в результате удаления из СД всех $\mathcal{D}_i$, для которых $\mathcal{D}_i \notin \mathcal{K}_k$. Установить ССД(ГС + i):= $\mathcal{C}_{i+1}$; СГП(ГС + i):= ГП для $1 \leq i \leq r-1$. Затем установить ГС:= ГС + $r-1$; СД:= $\mathcal{C}_1$.

ВДЛНУВ. (Выделение независимого удлиняющегося выражения). В этот момент СД $=(\mathcal{D}_1,\ \mathcal{D}_2,\ \ldots,\mathcal{D}_n)$. Рассмотреть $\mathcal{D}_1 = (L_1,\ \mathcal{E}_1,\ R_1)$ и проверить, имеет ли $\mathcal{E}_1$ вид $\mathcal{E}_1 = e_x\ \mathcal{E}_a\ e_y\ \mathcal{E}_b$, где $e_x\ \mathcal{E}_a\ e_y$ – независимое, удлиняющееся выражение. Если это так, то установить ОП:= ОП$\cup\{e_y\}$; ГПОП(e_y):= ГП.

ППВО. (Проектирование открытой переменной выражения). В этот момент $\mathcal{D}_1 = (L_1,\ e_x\ \mathcal{E}'_1,\ R_1)$, где $e_x \notin$ПП.

Если Г1$\neq L_1$ или Г2$\neq R_1$, то породить УГР,L_1, R_1.

Породить ПУД; УД; установить ГП:= ГП + 1; ПП:= ПП$\cup\{e_x\}$; НП(e_x):= НЭЛ + 1. Заменить $\mathcal{D}_1$ в СД на дыру (НЭЛ + 1, $\mathcal{E}'_1$, R_1). Установить Г1:= НЭЛ + 1; Г2:= R_1; НЭЛ:= НЭЛ + 2 и перейти к ИСД.

4.3 Компиляция правой части предложения

Правая часть предложения компилируется в последовательность операторов языка сборки, преобразующих поле зрения.

Как отмечалось в главе 3, действия, которые производятся над *значениями переменных*, во время компиляции можно изобразить в виде действий, выполняемых над *самими переменными*.

Таким образом, изменения, которые претерпевает поле зрения, в компиляторе изображены как изменения, происходящие с некоторым выражением, содержащим свободные переменные. Это выражение мы будем называть *объектом*.

Первоначально объект совпадает с левой частью предложения $\mathcal{L}$, и компилятор должен подобрать такую последовательность преобразований, которая преобразует объект в *цель* $\mathcal{R}$ – правую часть предложения.

Для этого можно использовать следующий алгоритм.

Алгоритм ЗАМ.

Исходные данные: левая часть предложения – $\mathcal{L}$, правая часть предложения - $\mathcal{R}$. Кроме того, для каждой переменной x, входящей в $\mathcal{L}$ – номер правого конца ее главного вхождения НП(x), а для каждого вхождения x в $\mathcal{L}$ – номер правого конца этого вхождения.

Результат работы: последовательность операторов преобразования.

ЗАМ. Установить ОБ:= $\mathcal{L}$; Ц:= $\mathcal{R}$; ГК:= 0. Породить УГ,0;

ЗАМЭ. Если Ц – пусто, то породить OUT,2; EST; и закончить работу.

Если Ц $=$ Z α где Z – символ, то породить NS,Z;, установить Ц: = α и перейти к ЗАМЭ.

Если Ц $=$ (α, то породить BL; установить Ц: = α и перейти к ЗАМЭ.

Если Ц $=$) α, то породить BR; установить Ц:= α и перейти к ЗАМЭ.

Если Ц $=$ k Z α, то породить BL; установить Ц:= α; ГК:= ГК + 1; СЗК(ГК):= Z и перейти к ЗАМЭ.

Если Ц $=$ $\perp$ α, то породить BR; AKTO,СЗК(ГК); установить Ц:= α; ГК:= ГК - 1 и перейти к ЗАМЭ.

Если Ц $= s_x\ \alpha$, то перейти к ЗАМПС.

Если Ц $= t_x\ \alpha$, то перейти к ЗАМПТ.

Если Ц $= e_x\ \alpha$, то перейти к ЗАМПВ.

ЗАМПС. Установить Ц:= α. Если ОБ содержит хоть одно вхождение s_x и n – номер этого вхождения, то удалить это вхождение из ОБ и породить TPLS,n; иначе породить MULS,НП(s_x). Перейти к ЗАМЭ.

ЗАМПТ. Установить Ц:= α. Если ОБ содержит хоть одно вхождение t_x и n – номер правого конца этого вхождения, то удалить это вхождение из ОБ и породить TPLE,n; иначе породить MULE,НП(t_x);. Перейти к ЗАМЭ.

ЗАМПВ. Установить Ц:= α. Если ОБ содержит хоть одно вхождение e_x и n – номер правого конца этого вхождения, то удалить это вхождение из ОБ и породить TPLE,n; иначе породить MULE,НП(e_x);. Перейти к ЗАМЭ.

Способ преобразования $\mathcal{L}$ в $\mathcal{R}$, которым пользуется алгоритм ЗАМ, аналогичен способу, который применялся в рефал-интерпретаторе [26,27]. Часто этот способ дает программу на языке сборки, которая не является наилучшей как в смысле ее длины, так и в смысле скорости ее выполнения.

Происходит это потому, что в реальных рефал-программах типичной является ситуация, когда $\mathcal{L}$ и $\mathcal{R}$ похожи друг на друга, т.е. $\mathcal{L}$ содержит такие участки, которые без изменений переходят в $\mathcal{R}$. В этих случаях, эти участки можно не формировать заново посредством операторов NS,S; BL; BR; TPLS,N; TPLE,N;, как это делает алгоритм ЗАМ, а взять их из $\mathcal{L}$ в готовом виде, целиком, посредством операторов TPL,N,M; (см.примеры в главе 3).

Возникает проблема: как найти наилучшую последовательность преобразований?

В принципе, эта задача разрешима, ибо операторы преобразования сконструированы так, что применение каждого оператора приближает нас к цели, и в каждый момент процесс преобразования можно продолжить только конечным числом способов. Таким образом, для заданных $\mathcal{L}$ и $\mathcal{R}$, число способов преобразования $\mathcal{L}$ в $\mathcal{R}$ конечно. Поэтому, наилучшую последовательность операторов можно найти конечным перебором.

К сожалению, такое "решение" невозможно использовать в компиляторе, ибо слишком велик возникающий комбинаторный перебор, и необходимо искать более быстрые алгоритмы.

В литературе указанная проблема известна как "проблема коррекции строки в строку" (string-to-string correction problem) и формулируется следующим

образом.

Рассмотрим множество строк Σ^* над алфавитом Σ, а также множество *операций редактирования* S. Каждая операция $s \in S$ является частичной функцией $s : \Sigma^* \to \Sigma^*$, т.е. для всякой строки $A \in \Sigma^*$, либо $s(A)$ не определено, либо $s(A) = B$, где B – некоторая строка $B \in \Sigma^*$. Каждой операции $s \in S$ приписывается целое число $\gamma(s)$ – ее стоимость. Рассмотрим теперь некоторую последовательность операций редактирования $\sigma = s_1\ s_2 \ldots s_n$. Назовем *стоимостью* $C(\sigma)$ последовательности σ число

$$C(\sigma) = \sum_{k=1}^{n} \gamma(s_k)$$

Пусть теперь заданы две строки $A, B \in \Sigma^*$. Будем говорить, что последовательность операций σ переводит A в B, если

$$B = \sigma(A) = s_n\ (\ldots s_2\ (s_1\ (A)\ ) \ldots)$$

Проблема коррекции строки в строку теперь ставится следующим образом. Пусть заданы две строки $A, B \in \Sigma^*$. Среди всех последовательностей операций, переводящих A в B, найти последовательность σ, имеющую минимальную стоимость.

Ясно, что решение проблемы коррекции строки в строку самым непосредственным образом зависит от рассматриваемого набора операций S.

В работе [18] предложен алгоритм, который решает проблему коррекции строки в строку за время, пропорциональное произведению длин строк, при условии, что набор операций S включает следующие операции: замена одного символа строки на другой символ, вставление символа в строку, удаление символа из строки.

В работе [19] результат работы [18] обобщается на тот случай, когда к S добавляется операция взаимной перестановки двух соседних символов, а стоимости операций удовлетворяют условию:

$$2W \geq W_D + W_I$$

где W_D – стоимость операции удаления, W_I – операции вставления, а W_S – операции взаимной перестановки двух соседних символов.

К сожалению, эти результаты нельзя использовать в компиляторе с рефала, поскольку основной выигрыш от оптимизации получается от применения оператора `TPL,N,M;`, который меняет местами два соседних участка выражения произвольной длины, причем затрачиваемое время не зависит от длины переставляемых участков.

В связи с вышеизложенным, пришлось пойти на компромиссное решение и применить в компиляторе с рефала эвристический алгоритм, который дает удовлетворительное качество компиляции и, в то же время, работает достаточно быстро.

Чтобы не вдаваться в несущественные технические подробности, мы опишем этот алгоритм в упрощенном виде, для случая, когда $\mathcal{L}$ и $\mathcal{R}$ не содержат скобок

и переменных. При этих условиях задача сводится к проблеме коррекции строки в строку с набором операций $S = S_I \cup S_S \cup S_D$, где S_I – множество операций вставления одного символа, S_S – множество операций взаимной перестановки двух соседних участков строки, а S_D – множество операций удаления участка строки.

Пусть нам заданы две строки A и B. Будем говорить, что строка C – их общий левый конец, если существуют такие строки A' и B', что $A = CA'$ и $B = CB'$. Теперь для двух строк A и B определим *сходство* этих строк $r(A, B)$ следующим образом. Если $A = B$, то $r(A, B) = +\infty$, а если $A \neq B$, то $r(A, B) =$ длине максимального общего левого конца A и B

Алгоритм коррекции строки в строку теперь описывается следующим образом.

Алгоритм `КОРР`.

Исходные данные: две строки A и B.

Результат работы: последовательность операций редактирования, которая переводит A в B.

`КОРР1`. Установить X`:=` A.

`КОРР2`. Если $X = B$, то конец работы.

`КОРР3`. Установить

$$S_o := \{\ s \in S \mid r(s(X)\ , B) > r(X, B)\ \};$$
$$M := \max_{s \in S_o} \{\ r(s(X), B)\ \};$$
$$S_M := \{\ s \in S_o \mid r(s(X)\ , B) = M\ \};$$

`КОРР4`. Установить $S_{MI} := S_I \cap S_M$

Если $S_{MI} \neq 0$, то выбрать $s_o \in S_{MI}$ и перейти к `КОРР5`. Установить $S_{MS} := S_S \cap S_M$

Если $S_{MS} \neq \emptyset$, то выбрать $s_o \in S_{MS}$ и перейти к `КОРР5`.

Выбрать $s_o \in S_M$ и перейти к `КОРР5`.

`КОРР5`. Выдать s_o в качестве очередной операции редактирования. Установить $X := s_o(X)$ и перейти к `КОРР2`.

Алгоритм `КОРР` всегда заканчивает работу за конечное число шагов. Это следует во-первых из того, что множество

$$S_o = \{s \in S \mid r(s(X), B) > r(X, B)\ \}$$

конечно для любых строк X и B, а во-вторых, из того, что для любой строки B не может существовать бесконечная последовательность строк A_1, A_2, ..., A_k, ...такая, что

$$r(A_i, B) < r(A_{i+1}, B) \text{ для всех } i = 1, 2, 3, \ldots$$

Пример. Пусть $A = a\ b\ c\ d\ e\ f$, $B = f\ b\ c\ a$. Тогда алгоритм `КОРР` построит следующую последовательность преобразований:

$$a\ b\ c\ d\ e\ f \xrightarrow{I} \underline{f}\ b\ c\ d\ e\ f \xrightarrow{S} \underline{f\ b\ c\ a}\ d\ e\ f \xrightarrow{D} \underline{f\ b\ c\ a}$$

где для каждой строки подчеркнут ее максимальный, общий с B левый конец.

Алгоритму `ЗАМ` соответствовала бы следующая последовательность преобразований:

$$a\ b\ c\ d\ e\ f \xrightarrow{I} f\ a\ b\ c\ d\ e\ f \xrightarrow{I} f\ b\ a\ b\ c\ d\ e\ f \xrightarrow{I}$$
$$f\ b\ c\ a\ b\ c\ d\ e\ f \xrightarrow{I} f\ b\ c\ a\ a\ b\ c\ d\ e\ f \xrightarrow{D} f\ b\ c\ a$$

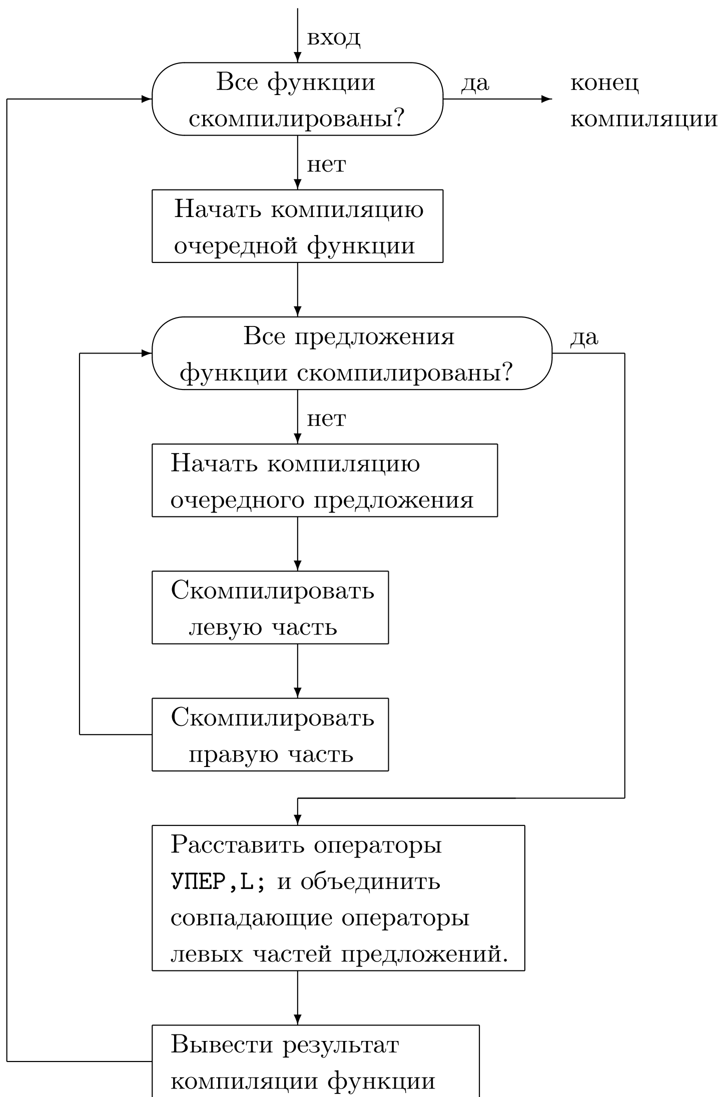


Рис. 4.1: Общая структура компилятора с рефала на язык сборки

Глава 5

Реализация интерпретатора языка сборки и компилятора с рефала

5.1 Основные этапы реализации

К моменту начала работы над рефал-компилятором уже существовал рефал-интерпретатор [26,27] для ЭВМ БЭСМ-6. Это дало возможность вести работу, придерживаясь следующей схемы:

1. Программным путем реализуется интерпретатор языка сборки для БЭСМ-6.

2. Пишется на рефале простейший компилятор с рефала на язык сборки, который отлаживается с помощью рефал-интерпретатора. Этот компилятор будем в дальнейшем называть "малым компилятором".

3. Малый компилятор компилирует сам себя на язык сборки. После этого он может работать уже без помощи рефал-интерпретатора.

4. Пишется на рефале более сложный, оптимизирующий компилятор, который в дальнейшем будем называть "большим компилятором". Большой компилятор отлаживается с помощью малого.

5. Большой компилятор компилирует сам себя на язык сборки. После этого мы получаем оптимизированный оптимизирующий компилятор на языке сборки.

Таким образом мы получаем компилятор с рефала в машинно- независимом виде. Поэтому, чтобы перенести реализацию рефала на другую машину, теперь достаточно реализовать интерпретатор языка сборки для этой машины.

Конечно, в принципе, интерпретатор языка сборки можно раз и навсегда описать на каком-либо машинно-независимом языке и получить полностью машинно-независимую реализацию рефала, однако, это вряд ли целесообразно по следующим причинам:

1. Скорость работы рефал-программ непосредственно зависит от эффективности интерпретации языка сборки. Поэтому при реализации языка сборки для некоторой ЭВМ желательно иметь возможность максимально использовать особенности этой ЭВМ для повышения эффективности.

2. Поскольку имеется формальное описание интерпретатора языка сборки, его реализация сводится к ручной трансляции этого описания в машинный код. Интерпретатор языка сборки достаточно прост. Его объем – 600-1000 команд.

3. Средства отладки рефал-программ, предоставляемые в различных реализациях, сильно зависят от окружающей среды, поэтому их вряд ли имеет смысл описывать в машинно-независимом виде.

Наконец, надо отметить, что наличие рефал-интерпретатора не является необходимым условием для реализации рефал-компилятора.

Поскольку малый компилятор имеет небольшой объем, его можно перевести на язык сборки вручную. А имея работающий малый компилятор, мы можем запустить большой компилятор.

5.2 Первый вариант языка сборки и компилятора

Работа над рефал-компилятором была начата автором в ИПМ АН СССР в 1969 году.

Первый вариант рефал-компилятора и интерпретатора языка сборки был закончен к ноябрю 1970 года.

Интерпретатор языка сборки был написан для ЭВМ БЭСМ-6 на автокоде. Компилятор был написан на рефале и отлажен с помощью рефал-интерпретатора. Затем компилятор перевел сам себя на язык сборки.

В январе 1971 года была начата работа по переносу рефал-компилятора с БЭСМ-6 на машины серии ЕС ЭВМ, в которой принимали участие Ан.В.Климов, Е.В.Травкина и И.Б.Щенков.

Интерпретатор языка сборки для ЕС ЭВМ был написан Ан.В.Климовым и отлажен Е.В.Травкиной. Затем Ан.В.Климов выдал рефал-компилятор, переведенный на язык сборки, на перфокарты, в виде констант ассемблера ЕС ЭВМ. Компилятор в таком виде был передан И.Б.Щенкову, который ввел его в ЕС ЭВМ и исправил некоторые ошибки, которые были допущены при кодировании операторов языка сборки константами ассемблера.

В октябре 1971 года работа по переносу компилятора на ЕС ЭВМ была закончена.

На начальном этапе работы над рефал-компилятором основная трудность заключалась в том, чтобы выделить набор элементарных операций, необходимых и удобных для компиляции рефал-программ. Были опробованы различные варианты языка сборки. В связи с этим интерпретатор языка сборки и компилятор приходилось многократно переделывать. Один из ранних вариантов описан в работе [29].

5.3 Переработка языка сборки и создание оптимизирующего компилятора

В середине 1971 года автор разработал новый вариант языка сборки, который имел перед предыдущим вариантом следующие преимущества:

1. Позволял реализовать более эффективный, чем использовавшиеся ранее в реализациях рефала, алгоритм отождествления. Это достигалось введением двух подвижных границ `Г1` и `Г2` вместо одной.

2. Позволял производить объединение совпадающих операторов отождествления из различных предложений. Это было достигнуто за счет введения стека переходов и оператора `УПЕР,L;`.

3. Позволял реализовать исключение лишних удлинений е-переменных. Это достигалось с помощью стека переходов и оператора `КУД,N;`.

Этот вариант языка сборки (описанный в [32]) почти не отличался от окончательного варианта, который описан в главе 3.

В октябре 1971 года описание нового варианта языка сборки было передано Е.В.Травкиной, которая написала на языке ассемблера и отладила соответствующий интерпретатор языка сборки для ЕС ЭВМ.

Затем автор написал на рефале малый компилятор с рефала на язык сборки, объемом приблизительно в 70 предложений. Е.В.Травкина перевела его вручную на язык сборки и отладила. Эта работа была закончена в январе 1972 года.

В январе-феврале 1972 года автором был написан на рефале оптимизирующий компилятор с рефала на язык сборки. В отличие от малого компилятора, он выполнял следующие оптимизации:

1. Объединение совпадающих операторов отождествления из различных предложений (см.главу 3).

2. Исключение лишних удлинений е-переменных (см.главу 4).

3. Оптимизацию преобразования левой части предложения в правую (см. главу 4).

Большой компилятор отлаживался на ЕС ЭВМ с помощью малого компилятора в феврале-марте 1972 года.

5.4 Перенос компилятора на другие машины

В апреле 1972 года Ан.В.Климов написал новый интерпретатор языка сборки для БЭСМ-6.

Для создания среды, необходимой для нормальной работы компилятора, в мае 1972 года Ан.В.Климовым была начата разработка мониторной системы "Рефал" [37]. Помимо монитора в эту систему в дальнейшем были включены: автокод БЕМШ, редактор связей, компилятор с рефала, ряд утилит. Работа над мониторной системой "Рефал" продолжалась вплоть до середины 1973 года и осуществлялась Ан.В.Климовым и Е.В.Травкиной.

В начале 1974 года было принято решение отказаться от мониторной системы "Рефал" и перевести рефал-компилятор в мониторную систему "Дубна", так как поддержание собственной мониторной системы порождало неудобства при использовании рефала и приводило к бесполезному расходу сил.

В марте-апреле 1974 года рефал-компилятор был переведен в мониторную систему "Дубна". В этой работе кроме автора принимали участие Ан.В.Климов, Л.В.Проворов и Е.В.Травкина.

При переходе в мониторную систему "Дубна" в реализацию рефала были внесены средства, позволяющие организовать взаимодействие и обмен информации между программами, написанными на рефале и программами, написанными на фортране [38,39].

В 1973году рефал-компилятор был перенесен на ЭВМ серии М-220 В.Ф.Хорошевским и А.Г.Красовским [40,41], а в 1975 году – на ЭВМ МИНСК-32 Арк.В.Климовым.

Работы по аппаратной реализации интерпретатора языка сборки ведутся в ИПМ АН СССР под руководством И.Б.Задыхайло, А.Н.Мямлина и В.К.Смирнова [42,43].

Заключение

В результате проделанной работы получены следующие основные результаты:

1. Создан машинно-независимый язык, предназначенный для обработки символьной информации – язык сборки.

2. На основании теоретического анализа правила отождествления, сформулированного в формальном описании рефала, разработаны алгоритмы оптимизации, которые позволяют уменьшить комбинаторный перебор, возникающий в процессе синтаксического отождествления. Правильность этих алгоритмов доказана теоретически.

3. Предложен алгоритм оптимизации, позволяющий уменьшить число преобразований, выполняемых над полем зрения, во время замены ведущей области конкретизации.

4. Разработан и реализован машинно-независимый оптимизирующий компилятор с рефала на язык сборки.

5. Разработан интерпретатор языка сборки и его формальное описание. Тем самым, реализация интерпретатора языка сборки на различных ЭВМ может быть получена путем ручной компиляции формального описания в коды конкретной машины.

6. Предложенная реализация рефала обладает высокой портативностью, что позволило перенести ее на ЭВМ БЭСМ-6 серии ЕС, М-220 и МИНСК-32. При этом, скорость работы скомпилированных рефал-программ повышена, по сравнению с реализациями на основе рефал-интерпретаторов, в 3-5 раз.

7. В настоящее время ведутся работы по аппаратной реализации языка сборки.

Литература

1. Kleene S.C. Introduction to Metamathematics.
   Van Nostrand, Princeton, 1952

   (Русский перевод: Клини С.К. Введение в метаматематику.
   ИЛ, 1957)

2. Марков А.А. Теория алгорифмов. Труды Матем. ин-та АН СССР им. В.А.Стеклова, XLII, Из-во АН СССР, 1954.

3. Mendelson E. Introduction to Mathematical Logic.
   Van Nostrand, Princeton, 1963

   (Русский перевод: Мендельсон Э. Введение в математическую логику. "Мир", М., 1971)

4. Yngve V.N. A programming language for mechanical translation.
   Mechanical Translation, 5(July 1958), №1, 25-41

   (Русский перевод: Ингве В. Язык для программирования задач машинного перевода. Кибернетический сб. 6, ИЛ, М., 1963)

5. Yngve V.N. COMIT as an information retrieval language,
   Comm. ACM, Vol. 5, №1 (January 1962), 19-28.

6. McCarthy J., Recursive functions of symbolic expressions and their computation by machine, Part I. Comm. ACM, Vol. 3, №4 (April 1960), 184-195.

7. McCarthy J. et al. LISP 1.5 Programmer's Manual. MIT Press,
   Cambridge, Mass., 1962.

8. Farber D.J., Griswold R.E., Polonsky I.P. SNOBOL, a string manipulation language. J.ACM, Vol. 11, №2 (January 1964), 21-30.

9. Farber D.J., Griswold R.E., Polonsky I.P. The SNOBOL3.
   Programming language. Bell System Tech. J. (July-August 1966).

10. Gimpel J.F. A theory of discrete patterns and their implementation in SNOBOL4. Comm. ACM, Vol. 16, №2 (February 1973), 91-100.

11. Cohen K., Wegstein J.H. AXLE; an axiomatic language for string transformations. Comm. ACM, Vol. 8, №11 (November 1965), 657-661.

12. Christensen C On the implementation of AMBIT, a language for symbol manipulation. Comm. ACM, Vol. 9, №8

    (August 1966), 570-573.

13. Christensen C. AMBIT/2 (programming language). Wakefield, Applied Data Research, Mass., 1970.

14. Guzman A., McIntosh H.V. CONVERT. Comm. ACM, Vol. 9, №8 (August 1966), 604-615.

15. Пильщиков В.Н. Строчные преобразования в языке ЛИСП. В сб.“Алгоритмы и алгоритмические языки”. Выпуск 6, М.,ВЦ АН СССР, 1973 .

16. Waite W.M. A language-independent macro processor. Comm. ACM, Vol. 10, №7 (July 1967).

17. Weizenbaum J. Symmetric list processor. Comm. ACM, Vol. 6, №9 (September 1963), 524-536.

18. Wagner R.A., Fischer M.J. The string-to-string correction problem. J.ACM, Vol. 21, №1 (January 1974), 168-173.

19. Lowrance R., Wagner R.A. An extension of the string-to-string correction problem. J.ACM, Vol. 22, №2 (April 1975), 177-183.

20. Floyd R. Nondeterministic Algorithms. J.ACM, Vol. 14, №4 (October 1967), 636-644.

21. Knuth D.E. The Art of Computer Programming. Vol.1: Fundamental Algorithms. Addison-Wesley, Reading, Mass.,1968.

    (Русский перевод: Кнут Д. Искусство программирования для ЭВМ. т.1: Основные алгоритмы. М., “Мир”, 1976).

22. “Цифровая вычислительная техника и программирование”. Сов. Радио, 1966, стр. 116-124.

23. “Кибернетика”, №4, 1968, с.45-54.

24. “Кибернетика”, №3, 1969, с.58-62.

25. Алгоритмический язык рекурсивных функций (РЕФАЛ). ИПМ АН СССР, М., 1968.

26. Флоренцев С.Н., Олюнин В.Ю. и др. Рефал-интерпретатор. В сб.“Труды 1-й Всесоюзной конференции по программированию”, Киев, ноябрь, 1968.

27. Флоренцев С.Н. Олюнин В.Ю. и др. Эффективный интерпретатор для языка РЕФАЛ. Препринт ИПМ АН СССР № 29, М., 1969.

28. Флоренцев С.Н., Олюнин В.Ю., Романенко С.А. и др.
Описание системы программирования "РЕФАЛ"(Инструкция для пользователей). Препринт ИПМ АН СССР № 30, М., 1969.

29. Романенко С.А. и др. Рефал-компилятор. В сб. "Труды 2-й Всесоюзной конференции по программированию". Новосибирск, 1970.

30. Романенко С.А. и др. Алгоритм перевода текста на РЕФАЛе в текст на машинно-ориантированном языке. В сб. "Тезисы докладов симпозиума по вопросам обработки символьной информации". Тбилиси, 3-5 ноября 1970.

31. Программирование на языке РЕФАЛ. Препринты ИПМ АН СССР №№ 41, 43, 46, 47, 49, М., 1971.

32. Климов А.В., Романенко С.А. и др. Компилятор с языка РЕФАЛ. ИПМ АН СССР, М., 1972.

33. Теория языков и методы построения систем программирования. Труды симпозиума. Киев-Алушта, 1972, с. 31-42.

34. Романенко С.А., Климов А.В. и др. Теоретические основы синтаксического отождествления в языке РЕФАЛ. Препринт ИПМ АН СССР № 13, М., 1973.

35. Базисный рефал. Описание языка и основные приемы программирования. М., ЦНИПИАС, 1974.

36. Труды ЦНИПИАСС. Вып. 6. М., 1974, с. 36-68.

37. Климов А.В., Романенко С.А., Травкина Е.В. Инструкция по работе с мониторной системой "РЕФАЛ"для БЭСМ-6. ИПМ АН СССР, М., 1974.

38. Климов А.В., Проворов Л.В., Романенко С.А., Травкина Е.В. Рефал в мониторной системе "Дубна"БЭСМ-6. Входной язык компилятора и запуск программ. Препринт ИПМ АН СССР № 8, М., 1975.

39. Климов А.В., Романенко С.А. Рефал в мониторной системе "Дубна" БЭСМ-6. Интерфейс рефала и фортрана. ИПМ АН СССР, М.,1975.

40. Кузин Л.Т., Флоренцев С.Н., Хорошевский В.Ф., Красовский А.Г. Система программирования на базе языка РЕФАЛ для ЭВМ типа БЭСМ-4, М-220, М-222. Отчёт МИФИ № 72008046, М., 1973,

41. Флоренцев С.Н., Хорошевский В.Ф., Красовский А.Г.

Принципы реализации языка РЕФАЛ на малых машинах.
В сб."Некоторые вопросы кибернетики". Вып. 2, МИФИ, М., 1975.

42. Задыхайло И.Б., Мямлин А.Н., Смирнов В.К., Эйсымонт Л.К.
Об эффективной аппаратной реализации языка для описания объектов на уровне понятий и символьных преобразований. Искусственный интеллект. Итоги, перспективы. Семинар МДНТП им.Дзержинского, М., 1974, стр. 157-164.

43. Ковалев Э.С., Меламед В.И., Мямлин А.Н., Смирнов В.К.
Микропрограммная реализация символьного процессора.
В сб.“Вопросы синтеза логики ЦВМ. Часть III” Каунасский политехнический институт, Каунас, 1976, стр.98-101.

Приложение A

Теоретический анализ правила отождествления

A.1 Строки и подстановки

Пусть Σ - конечное или счетное множество символов. Итерация его Σ^* - это множество строк над алфавитом Σ.

Пустую строку будем обозначать через $\square$.

Длину строки α будем обозначать через $|\alpha|$. i-й символ строки α будем обозначать как α_i. Таким образом, любую строку α можно представить в виде:

$$\alpha = \alpha_1 \alpha_2 \dots \alpha_{|\alpha|}$$

Через $\|\alpha\|$ мы будем обозначать неупорядоченное множество символов, входящих в строку α. Таким образом,

$$\|\alpha\| = \bigcup_{k=1}^{|\alpha|} \{\alpha_k\}$$

Через $\|\alpha_{i:j}\|$ мы будем обозначать строку $\alpha_i \alpha_{i+1} \dots \alpha_j$, если $1 \leq i \leq j \leq |\alpha|$. Если же $j < i$, мы будем считать, что $\alpha_{i:j} = \square$.

Назовем *подстановкой* любое конечное множество упорядоченных пар вида (s, α), где $s \in \Sigma$ а $\alpha \in \Sigma^*$.

Определение A.1.1 *Подстановка Δ называется* противоречивой, *если существуют две таких пары (s, α) и (s, β), что $(s, \alpha) \in \Delta$ и $(s, \beta) \in \Delta$, и при этом $\alpha \neq \beta$. Подстановку, которая не является противоречивой, мы будем называть* непротиворечивой.

Обозначим через $D_0[\Delta]$ предикат, который распознает непротиворечивость подстановки Δ. То есть

$$D_0[\Delta] = \begin{cases} \text{истина,} & \text{если } \Delta \text{ - непротиворечива} \\ \text{ложь,} & \text{если } \Delta \text{ - противоречива} \end{cases}$$

Очевидно, что D_0 - вычислимый предикат.

Обозначим через $\|\Delta\|$ неупорядоченное множество таких символов s, для которых существует такая строка α, что $(s, \alpha) \in \Delta$. Будем говорить, что символы из $\|\Delta\|$ "входят в подстановку Δ".

Обозначим результат применения подстановки Δ к строке α через $\Delta//\alpha$. Определим его следующим образом.

Если $\|\alpha\| \not\subseteq \|\Delta\|$ или неверно $D_0[\Delta]$, то результат $\Delta//\alpha$ не определен.

Если $\alpha = \Box$, то $\Delta//\alpha = \Box$.

Если $\alpha \neq \Box$, то $\Delta//\alpha = \Delta//[\alpha_1\alpha_2...\alpha_{\|\alpha\|}] = [\Delta//\alpha_1][\Delta//\alpha_2]...[\Delta//\alpha_{\|\alpha\|}]$.

Если подстановка Δ применяется к символу s, то $\Delta//s = \beta$, где $(s, \beta) \in \Delta$. Поскольку мы предположили, что $D_0[\Delta]$ и $\|\alpha\| \subseteq \|\Delta\|$, то пара (s, β) существует и единственна.

Введем теперь понятие *правильной* подстановки. Будем считать, что задан некоторый вычислимый предикат $D[\Delta]$, обладающий тем свойством, что если $D[\Delta]$ истинно, то истинно и $D_0[\Delta]$. То есть $D[\Delta] \Rightarrow D_0[\Delta]$. *Правильной* подстановкой будем называть любую подстановку, для которой верно $D[\Delta]$.

Из определения следует, что множество правильных подстановок является подмножеством множества непротиворечивых подстановок.

Будем говорить, что предикат D *невырожденный*, если существует хоть одна подстановка Δ, такая, что истинно $D[\Delta]$.

A.2 Отождествление строк

Определение A.2.1 *Пусть заданы две строки α и β. Тогда* отождествлением *строки β как строки α мы будем называть любую подстановку Δ, для которой $D[\Delta]$, $\|\alpha\| \subseteq \|\Delta\|$ и $\Delta//\alpha = \beta$*

Задача нахождения отождествления строки β как строки α является, по существу, обратной задачей по отношению к вычислению $\Delta//\alpha$.

В общем случае может существовать бесконечное число отождествлений β как α[1], но может не существовать и ни одного.

Теперь мы несколько обобщим предыдущее определение.

Определение A.2.2 *Пусть заданы две строки α и β и подстановка Δ. Тогда* отождествлением *строки β как строки α при Δ мы будем называть любую подстановку Γ, для которой $D[\Gamma]$, $\|\alpha\| \cup \|\Delta\| \subseteq \|\Gamma\|$, $\Delta \subseteq \Gamma$ и $\Gamma//\alpha = \beta$.*

Обозначим через $W[\Delta, (\alpha, \beta)]$ множество таких отождествлений β как α при Δ, что $\|\Gamma\| = \|\alpha\| \cup \|\Delta\|$[2].

Теорема A.2.1 *Множество $W[\Delta, (\alpha, \beta)]$ конечно.*

[1] Просто в силу того, что Δ может приписывать значения "лишним" переменным. Ведь определение не требует, чтобы $\|\alpha\| = \|\Delta\|$

[2] Т.е. множество $W[\Delta, (\alpha, \beta)]$ содержит подстановки, в которых "достаточно" информации, но и "не слишком много"

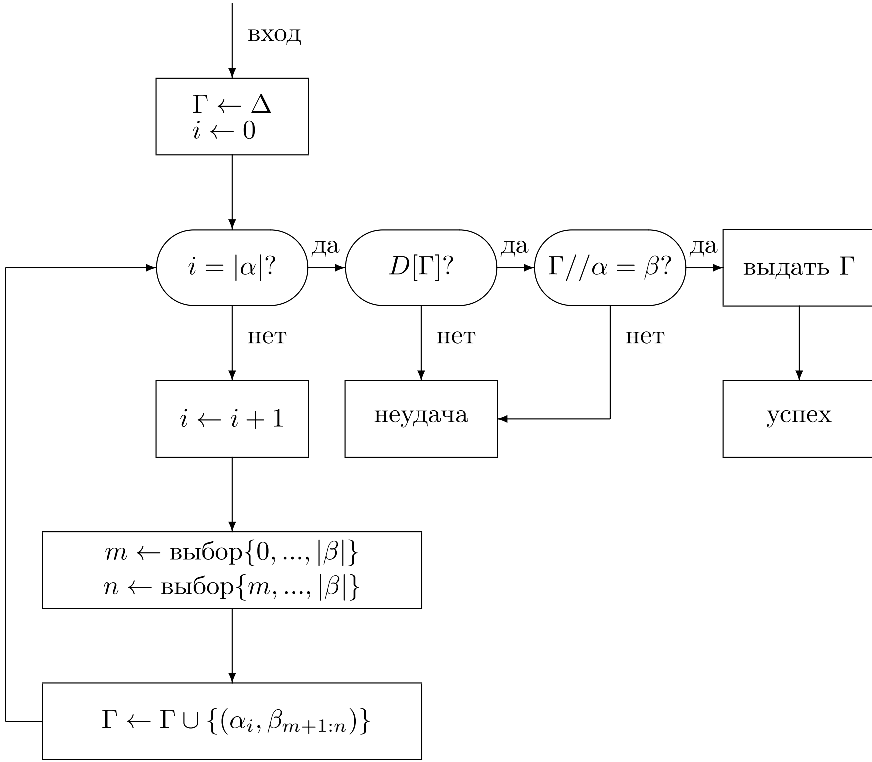


Рис. A.1: Простейший алгоритм отождествления

Доказательство. Пусть $\Gamma \in W[\Delta, (\alpha, \beta)]$. Тогда для любой пары $(s, \gamma) \in \Gamma$ имеем либо $(s, \gamma) \in \Delta$, либо $(s, \gamma) \in \Gamma \setminus \Delta$. Пар, для которых $(s, \gamma) \in \Delta$ только конечное число. Рассмотрим теперь пары, для которых $(s, \gamma) \in \Gamma \setminus \Delta$. Для каждой такой пары имеем $s \in \|\alpha\| \setminus \|\Delta\|$, $\|\gamma\| \subseteq \|\beta\|$ и $|\gamma| \leq \|\beta\|$. Поэтому таких пар может быть только конечное число. Теорема доказана.

A.3 Алгоритм отождествления

Теорема A.2.1 подсказывает нам, как построить алгоритм, который за конечное число шагов построит все множество $W[\Delta, (\alpha, \beta)]$. Поскольку этот алгоритм содержит комбинаторный перебор, его удобно представить в виде *недетерминированной* программы (рис.A.1). В дальнейшем мы будем использовать недетерминированные программы для представления подобных алгоритмов, т.к. они позволяют выделить в них главное, отвлекаясь от несущественных деталей [20].

Видно, что алгоритм, приведенный на рис.А.1, крайне неэффективен. Время работы его $O(|\beta|^{2|\alpha|})$.

Можно улучшить его, если принять во внимание, что

$$\sum_{i=1}^{|\alpha|} |\Gamma//\alpha_i| = |\beta|$$

Ясно, что каждому $\Gamma \in W[\Delta, (\alpha, \beta)]$ соответствует набор целых неотрицательных чисел $(z_1, z_2, ..., z_{|\alpha|})$ таких, что $|\Gamma//\alpha_i| = z_i$ и $z_1 + z_2 + ... + z_{|\alpha|} = |\beta|$. И обратно, каждому набору $(z_1, z_2, ..., z_{|\alpha|})$ может соответствовать не более, чем одно $\Gamma \in W[\Delta, (\alpha, \beta)]$.

На рис.А.2 приведен улучшенный алгоритм отождествления, в котором учтены сделанные замечания. Время работы этого алгоритма $O(|\beta|^{|\alpha|})$.

А.4 Монотонные предикаты $D[\Delta]$

Определение А.4.1 *Предикат D называется* монотонным, *если для любых подстановок Δ_1 и Δ_2 таких, что $\Delta_2 \subseteq \Delta_1$, из $D[\Delta_1]$ следует $D[\Delta_2]$. Т.е. $D[\Delta_1]$ и $\Delta_2 \subseteq \Delta_1 \Rightarrow D[\Delta_2]$*

Теорема А.4.1 *Для любого монотонного предиката D, $\Delta_2 \subseteq \Delta_1$ и ложно $D[\Delta_2] \Rightarrow$ ложно $D[\Delta_1]$.*

Доказательство. Пусть $\Delta_2 \subseteq \Delta_1$ и ложно $D[\Delta_2]$. Предположим, что $D[\Delta_1]$ истинно. Тогда, по определению А.4.1, $D[\Delta_1]$ и $\Delta_2 \subseteq \Delta_1 \Rightarrow D[\Delta_2]$. Однако, $D[\Delta_2]$ ложно по условию теоремы. Следовательно, $D[\Delta_1]$ не может быть истинным.

Теорема А.4.2 *Пусть D - монотонный, невырожденный предикат. Тогда $D[\emptyset]$ истинно.*

Доказательство. Раз D - невырожденный, значит существует такая подстановка Δ, что $D[\Delta]$ истинно. Однако, $\emptyset \subseteq \Omega$, откуда по определению А.4.1, получаем $D[\emptyset]$.

Теорема А.4.3 *Если D - монотонный предикат, и $D[\Delta]$ ложно, то, для любых α и β, $W[\Delta, (\alpha, \beta)] = \emptyset$.*

Доказательство. Согласно определению А.2.2 для любой $\Gamma \in W[\Delta, (\alpha, \beta)]$ выполнено $\Delta \subseteq \Gamma$. Однако, по условию теоремы, $D[\Delta]$ ложно. Поэтому, по теореме А.4.1, $D[\Gamma]$ ложно. Следовательно $\Gamma \notin W[\Delta, (\alpha, \beta)]$. Полученное противоречие доказывает, что $W[\Delta, (\alpha, \beta)] = \emptyset$. Теорема доказана.

Если известно, что $D[\Delta]$ - монотонный предикат, можно улучшить алгоритм отождествления, приведенный на рис. А.2. Этот алгоритм был основан на том, что мы сначала строим подстановку Γ всю целиком, а потом проверяем ее на правильность. Однако, для монотонного $D[\Delta]$ не обязательно дожидаться, пока подстановка будет построена полностью. Если в процессе работы алгоритма

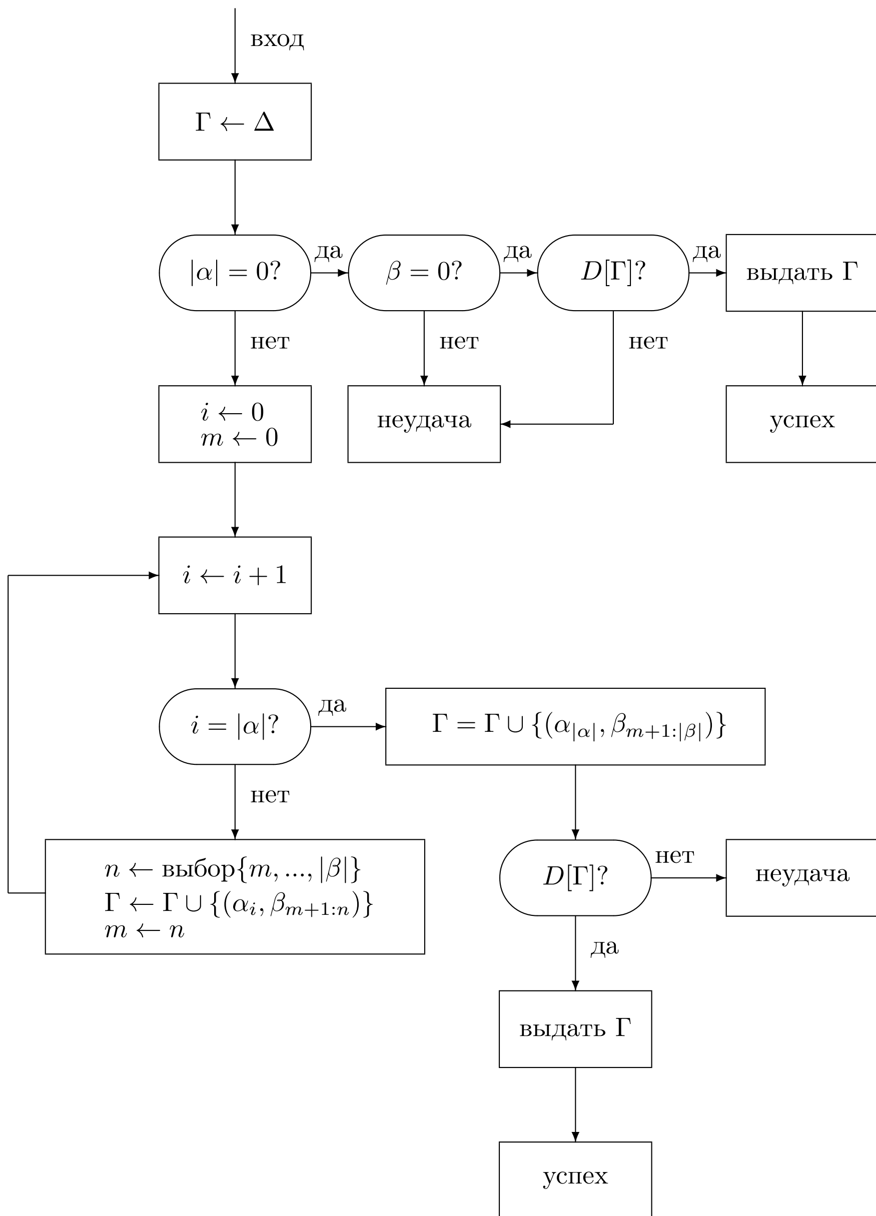


Рис. А.2: Улучшенный алгоритм отождествления

окажется вдруг, что $D[\Gamma]$ ложно, то продолжать построение Γ бессмысленно, ибо, добавляя к подстановке Γ новые пары (s, γ), мы будем все время получать неправильные подстановки.

На рис. А.3 приведен алгоритм отождествления, в котором учтено сделанное замечание.

В дальнейшем мы будем использовать только монотонные, невырожденные $D[\Delta]$. Поэтому, начиная с этого места, мы всюду будем предполагать, что $D[\Delta]$ - это произвольный предикат, определенный на всем множестве подстановок и удовлетворяющий следующим требованиям:

1. Непротиворечивость:
   $D[\Delta] \Rightarrow D_0[\Delta]$

2. Невырожденность:
   $D[\emptyset]$

3. Монотонность:
   $D[\Delta_1]$ и $\Delta_2 \subseteq \Delta_1 \Rightarrow D[\Delta_2]$

Обратим внимание, что предикат $D_0[\Delta]$ очевидным образом удовлетворяет всем поставленным требованиям. Поэтому, в частности, может оказаться $D[\Delta] \Leftrightarrow D_0[\Delta]$. В то же время, $D[\Delta] \Rightarrow D_0[\Delta]$. Поэтому из всех допустимых предикатов $D[\Delta]$ предикат $D_0[\Delta]$ накладывает на подстановки Δ наименее жесткие ограничения.

A.5 Отождествление векторов строк

Для дальнейшего исследования алгоритмов отождествления нам потребуется обобщить некоторые из ранее введенных понятий.

Пусть нам задано n строк: α^1, α^2,..., α^n. Назовем *вектором строк* упорядоченную n-ку $\overline{\alpha} = (\alpha^1, \alpha^2, \ldots, \alpha^n)$.

Пустой вектор строк будем обозначать через $\emptyset$.

Пусть теперь заданы два вектора строк: $\overline{\alpha} = (\alpha^1, \alpha^2, \ldots, \alpha^n)$ и $\overline{\beta} = (\beta^1, \beta^2, \ldots, \beta^m)$. Назовем их *конкатенацией* вектор строк

$$\overline{\alpha}\overline{\beta} = (\alpha^1, \alpha^2, \ldots, \alpha^n, \beta^1, \beta^2, \ldots, \beta^m)$$

Пусть задан вектор строк $\overline{\alpha} = (\alpha^1, \alpha^2, \ldots, \alpha^n)$. обозначим через $\|\overline{\alpha}\|$ множество символов

$$\|\overline{\alpha}\| = \|\alpha^1\| \cup \|\alpha^2\| \cup \ldots \cup \|\alpha^n\|$$

Про символы из множества $\|\overline{\alpha}\|$ мы будем говорить, что они *входят* в вектор строк $\overline{\alpha}$.

Пусть Δ - непротиворечивая подстановка, а $\overline{\alpha} = (\alpha^1, \alpha^2, \ldots, \alpha^n)$. Тогда результат применения подстановки Δ к вектору строк $\overline{\alpha}$ мы определим как

$$\Delta//\overline{\alpha} = (\Delta//\alpha^1, \Delta//\alpha^2, \ldots, \Delta//\alpha^n)$$

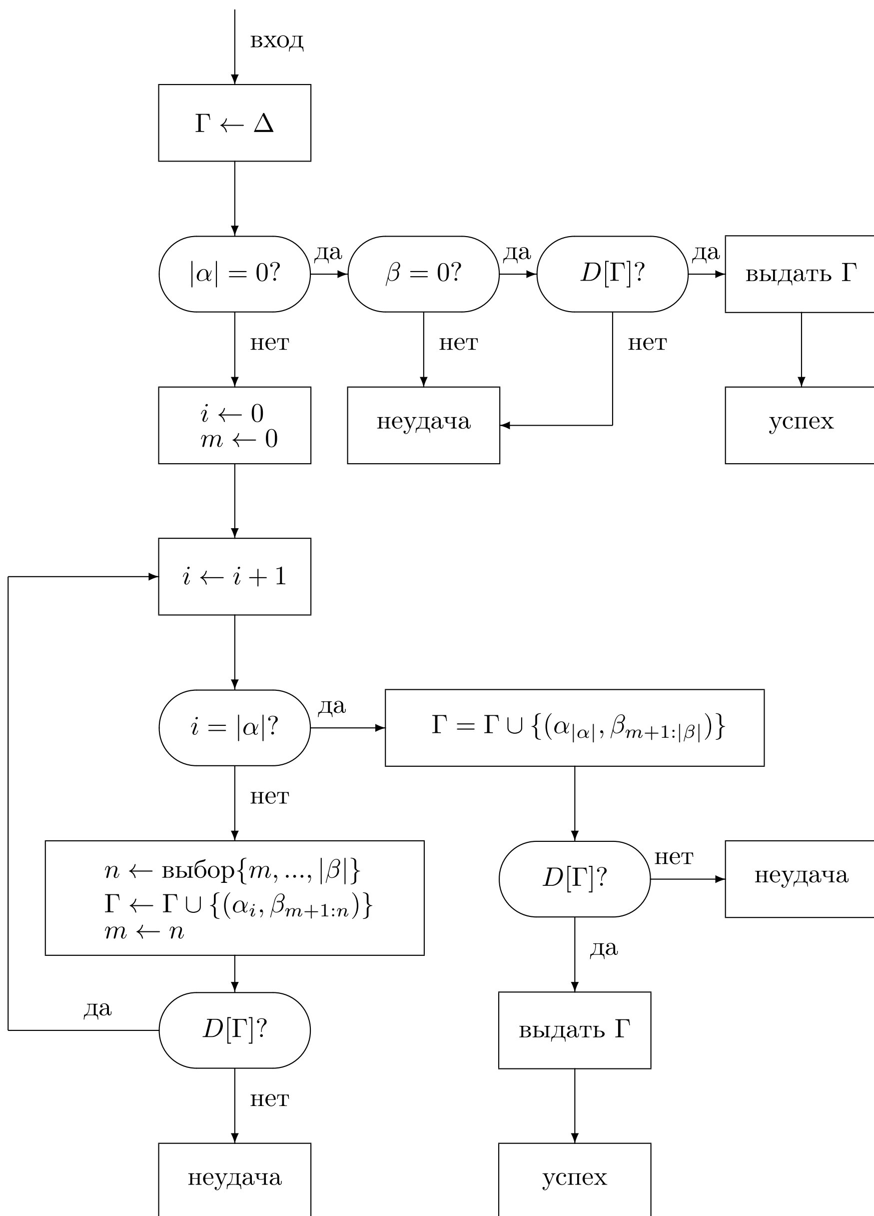


Рис. А.3: Алгоритм отождествления, который учитывает монотонность $D[\Gamma]$.

Пусть теперь задано два вектора строк:

$$\overline{\alpha} = (\alpha^1, \alpha^2, \dots, \alpha^n), \quad \overline{\beta} = (\beta^1, \beta^2, \dots, \beta^n)$$

Обозначим через $\overline{\alpha} * \overline{\beta}$ упорядоченную n-ку упорядоченных пар:

$$\overline{\alpha} * \overline{\beta} = ((\alpha^1, \beta^1), (\alpha^2, \beta^2), \dots, (\alpha^n, \beta^n))$$

которую будем называть *вектором пар строк.*

Для сокращения записи будем записывать $\overline{\alpha} * \overline{\beta}$ в виде

$$\overline{\alpha} * \overline{\beta} = (\alpha^1, \beta^1)(\alpha^2, \beta^2) \dots (\alpha^n, \beta^n)$$

Пустой вектор пар строк будем обозначать через $\emptyset$.

Пусть нам заданы два вектора пар строк $\overline{x} = \overline{\alpha} * \overline{\beta}$ и $\overline{y} = \overline{\gamma} * \overline{\delta}$, где $\overline{x} = (\alpha^1, \beta^1)(\alpha^2, \beta^2) \dots (\alpha^n, \beta^n)$, а $\overline{y} = (\gamma^1, \delta^1)(\gamma^2, \delta^2) \dots (\gamma^m, \delta^m)$. Тогда *конкатенацией* $\overline{x}\,\overline{y}$ вектора пар строк $\overline{x}$ и вектора пар строк $\overline{y}$ мы назовем вектор пар строк

$$\overline{x}\,\overline{y} = (\alpha^1, \beta^1)(\alpha^2, \beta^2) \dots (\alpha^n, \beta^n)(\gamma^1, \delta^1)(\gamma^2, \delta^2) \dots (\gamma^m, \delta^m)$$

Пусть $\overline{x} = \overline{\alpha} * \overline{\beta}$ - вектор пар строк. Тогда через $\|\overline{x}\|$ обозначим множество символов, входящих в $\overline{\alpha}$. Т.е.

$$\|\overline{x}\| = \|\overline{\alpha} * \overline{\beta}\| = \|\overline{\alpha}\|$$

Определение A.5.1 *Пусть задан вектор пар строк $\overline{\alpha} * \overline{\beta}$ и подстановка Δ. Тогда* отождествлением *вектора строк $\overline{\beta}$ как вектора строк $\overline{\alpha}$ при Δ мы будем называть любую подстановку Γ, для которой*

$$D[\Gamma],\ \|\overline{\alpha}\| \cup \|\Delta\| \subseteq \|\Gamma\|,\ \Delta \subseteq \Gamma \text{ и } \Gamma // \overline{\alpha} = \overline{\beta}$$

Через $W[\Delta, \overline{\alpha} * \overline{\beta}]$ обозначим множество таких отождествлений $\overline{\beta}$ как $\overline{\alpha}$ при Δ, что

$$D[\Gamma],\ \|\Gamma\| = \|\overline{\alpha}\| \cup \|\Delta\|,\ \Delta \subseteq \Gamma \text{ и } \Gamma // \overline{\alpha} = \overline{\beta}$$

Теорема A.5.1 *Пусть $\overline{\alpha} = (\alpha^1, \alpha^2, \dots, \alpha^n)$, $\overline{\beta} = (\beta^1, \beta^2, \dots, \beta^n)$, $\alpha = \alpha^1 \alpha^1 \dots \alpha^n$ и $\beta = \beta^1 \beta^1 \dots \beta^n$. Тогда $W[\Delta, \overline{\alpha} * \overline{\beta}] \subseteq W[\Delta, (\alpha, \beta)]$.*

Доказательство. Если $\Gamma \in W[\Delta, \overline{\alpha} * \overline{\beta}]$, то $\Gamma // \alpha^i = \beta^i$ для $i = 1, 2, \dots, n$. Отсюда следует, что $\Gamma // \alpha = \beta$, что и доказывает теорему.

Теорема A.5.2 *Множество $W[\Delta, \overline{\alpha} * \overline{\beta}]$ конечно.*

Доказательство. Очевидно из теорем A.2.1 и A.5.1.

A.6 Теоремы о разложении $W[\Delta, \overline{x}]$

Лемма A.6.1 *Пусть* Γ *- подстановка, а* $\overline{x}$ *и* $\overline{y}$ *- векторы пар строк, такие, что*

$$\overline{x} = \overline{\alpha}_x * \overline{\beta}_x,\ \overline{y} = \overline{\alpha}_y * \overline{\beta}_y,\ \|\overline{x}\,\overline{y}\| \subseteq \|\Gamma\| \text{ и } D_0[\Gamma]$$

Тогда $\Gamma//\overline{\alpha}_x\overline{\alpha}_y = \overline{\beta}_x\overline{\beta}_y$ *равносильно тому, что* $\Gamma//\overline{\alpha}_x = \overline{\beta}_x$ *и* $\Gamma//\overline{\alpha}_y = \overline{\beta}_y$.

Доказательство. Является тривиальным следствием определения $\Gamma//\overline{\alpha}$.

Лемма A.6.2 *Пусть* Δ *и* Γ *подстановки, а* $\overline{\alpha}$ *вектор строк, такие, что*

$$D[\Gamma],\ \Delta \subseteq \Gamma \text{ и } \|\overline{\alpha}\| \cup \|\Delta\| \subseteq \|\Gamma\|$$

Тогда существует единственная подстановка Ω *такая, что* $\Delta \subseteq \Omega \subseteq \Gamma$, $\|\Omega\| = \|\Delta\| \cup \|\overline{\alpha}\|$ *и при этом* $\Gamma//\overline{\alpha} = \Omega//\overline{\alpha}$.

Доказательство. Из $D[\Gamma]$ и $\Delta \subseteq \Gamma$ следует, что $D[\Delta]$. Теперь образуем из Γ новую подстановку следующим образом. Удалим из Γ все пары $(s, \gamma) \in \Gamma$ такие, что $s \notin \|\Delta\| \cup \|\overline{\alpha}\|$ и обозначим эту подстановку через Ω. Ясно, что $\Omega \subseteq \Gamma$, откуда $D[\Omega]$.

Далее заметим, что $\Gamma//\overline{\alpha}$ зависит только от таких пар $(s, \gamma) \in \Gamma$, в которых $s \in \|\Delta\| \cup \|\overline{\alpha}\|$. Но такие пары входят и в Γ, и в Ω. Поэтому $\Omega//\overline{\alpha} = \Gamma//\overline{\alpha}$.

Лемма A.6.3 *Пусть* $\overline{\alpha}$ *вектор строк, а* Δ *подстановка, такие, что* $\|\overline{\alpha}\| \subseteq \|\Delta\|$. *Тогда для любой подстановки* Γ *такой, что* $D[\Gamma]$ *и* $\Delta \subseteq \Gamma$, *имеет место* $\Delta//\overline{\alpha} = \Gamma//\overline{\alpha}$.

Доказательство. Из $\Delta \subseteq \Gamma$ и $\|\overline{\alpha}\| \subseteq \|\Delta\|$ следует, что $\|\overline{\alpha}\| \subseteq \|\Gamma\|$. Поэтому $\Gamma//\overline{\alpha}$ определена. Теперь применяем лемму A.6.2 и получаем, что $\Gamma//\overline{\alpha} = \Delta//\overline{\alpha}$.

Теорема A.6.1 *Пусть* $\Gamma \in W[\Delta, \overline{x}\,\overline{y}\,\overline{z}]$. *Тогда* Γ *единственным образом представляется в виде* $\Gamma = \Gamma_y \cup \Gamma_{xz}$, *где* $\Gamma_y \in W[\Delta, \overline{y}]$, *а* $\Gamma_{xz} \in W[\Delta, \overline{x}\,\overline{z}]$

Доказательство. Пусть $\overline{x} = \overline{\alpha}_x * \overline{\beta}_x$, $\overline{y} = \overline{\alpha}_y * \overline{\beta}_y$, $\overline{z} = \overline{\alpha}_z * \overline{\beta}_z$. Имеем $\Gamma//\overline{\alpha}_x\overline{\alpha}_y\overline{\alpha}_z = \overline{\beta}_x\overline{\beta}_y\overline{\beta}_z$, откуда по лемме A.6.1 получаем $\Gamma//\overline{\alpha}_x = \overline{\beta}_x$, $\Gamma//\overline{\alpha}_y = \overline{\beta}_y$, $\Gamma//\overline{\alpha}_z = \overline{\beta}_z$. Отсюда, в свою очередь, $\Gamma//\overline{\alpha}_x\overline{\alpha}_z = \overline{\beta}_x\overline{\beta}_z$ и $\Gamma//\overline{\alpha}_y = \overline{\beta}_y$. Теперь, по лемме A.6.2 существуют и единственны такие подстановки Γ_y и Γ_{xz}, что

$$\Delta \subseteq \Gamma_y \subseteq \Gamma,\ \|\Gamma_y\| = \|\Delta\| \cup \|\overline{\alpha}_y\|,\ \Gamma_y//\overline{\alpha}_y = \overline{\beta}_y$$

$$\Delta \subseteq \Gamma_{xz} \subseteq \Gamma,\ \|\Gamma_{xz}\| = \|\Delta\| \cup \|\overline{\alpha}_x\overline{\alpha}_z\|,\ \Gamma_{xz}//\overline{\alpha}_x\overline{\alpha}_z = \overline{\beta}_x\overline{\beta}_z$$

откуда видно, что $\Gamma_y \in W[\Delta, \overline{y}]$, а $\Gamma_{xz} \in W[\Delta, \overline{x}\,\overline{z}]$. Теорема доказана.

Следствие A.6.1.1 *Если* $W[\Delta, \overline{y}] = \emptyset$ *или* $W[\Delta, \overline{x}\,\overline{z}] = \emptyset$, *то и* $W[\Delta, \overline{x}\,\overline{y}\,\overline{z}] = \emptyset$

Следствие A.6.1.2 $W[\Delta,\overline{x}\,\overline{y}\,\overline{z}] = \cup\,\{W[\Delta\cup\Gamma,\overline{x}\,\overline{z}] \mid \Gamma\in W[\Delta,\overline{y}]\}$

Следующие теоремы дают нам рекурсивные соотношения, которые позволяют вычислять $W[\Delta,\overline{x}]$.

Теорема A.6.2 $W[\Delta,\emptyset] = \begin{cases} \{\Delta\} & \textit{при } D[\Delta] \\ \emptyset & \textit{при не } D[\Delta] \end{cases}$

Доказательство. Пусть $\Gamma\in W[\Delta,\emptyset]$. Тогда, согласно определению A.5.1, имеем $\|\Gamma\| = \|\emptyset\|\cup\|\Delta\| = \|\Delta\|$ и $\Delta\subseteq\Gamma$. Поэтому $\Gamma=\Delta$. Таким образом, $W[\Delta,\emptyset]$ может содержать только Δ.

Если $D[\Delta]$ ложно, то $\Delta\notin W[\Delta,\emptyset]$ и $W[\Delta,\emptyset]=\emptyset$.

Если $D[\Delta]$, то $\Delta//\emptyset=\emptyset$ и $\Delta\in W[\Delta,\emptyset]$.

Теорема доказана.

Теорема A.6.3 $W[\Delta,\overline{x}(\Box,\beta)\overline{y}] = \begin{cases} W[\Delta,\overline{x}\,\overline{y}] & \textit{при } \beta=\Box \\ \emptyset & \textit{при } \beta\neq\Box \end{cases}$

Доказательство. Для любой подстановки Γ, такой, что $D[\Gamma]$, имеем $\Gamma//\Box=\Box$. Поэтому $\Gamma//\Box=\beta$ выполнено для любой Γ при $\beta=\Box$ и не выполнено ни для какой Γ при $\beta\neq\Box$.

Теорема A.6.4 $W[\Delta,\overline{x}(s,\beta)\overline{y}] = W[\Delta\cup\{(s,\beta)\},\overline{x}\,\overline{y}]$

Доказательство. Пусть $\Gamma\in W[\Delta,\overline{x}(s,\beta)\overline{y}]$. Тогда $\Gamma//s=\beta$, откуда следует, что $(s,\beta)\in\Gamma$. Поэтому $\Gamma\in W[\Delta\cup\{(s,\beta)\},\overline{x}(s,\beta)\overline{y}] = W[\Delta\cup\{(s,\beta)\},\overline{x}\,\overline{y}]$.

Обратно, пусть $\Gamma\in W[\Delta\cup\{(s,\beta)\},\overline{x}\,\overline{y}]$. Тогда $\Gamma//s=\beta$, откуда следует, что $\Gamma\in W[\Delta\cup\{(s,\beta)\},\overline{x}(s,\beta)\overline{y}] = W[\Delta,\overline{x}(s,\beta)\overline{y}]$ Теорема доказана.

Теорема A.6.5 $W[\Delta,\overline{x}(\alpha\gamma,\beta)\overline{y}] = \bigcup_{k=0}^{|\beta|} W[\Delta,\overline{x}(\alpha,\beta_{1:k})(\gamma,\beta_{k+1:|\beta|})\overline{y}]$

Доказательство. Для любого $\Gamma\in W[\Delta,\overline{x}(\alpha\gamma,\beta)\overline{y}]$ имеем $\Gamma//\alpha\gamma=\beta$. Однако, $\Gamma//\alpha\gamma = [\Gamma//\alpha][\Gamma//\gamma]=\beta$. Поэтому существует такое k, что $\Gamma//\alpha=\beta_{1:k}$ и $\Gamma//\gamma=\beta_{k+1:|\beta|}$. Отсюда вытекает, что

$$\Gamma\in W[\Delta,\overline{x}(\alpha,\beta_{1:k})(\gamma,\beta_{k+1:|\beta|})\overline{y}]$$

И наоборот, пусть $\Gamma\in W[\Delta,\overline{x}(\alpha,\beta_{1:k})(\gamma,\beta_{k+1:|\beta|})\overline{y}]$. Тогда $\Gamma//\alpha=\beta_{1:k}$ и $\Gamma//\gamma=\beta_{k+1:|\beta|}$. Отсюда $\Gamma//\alpha\gamma=\beta$ и $\Gamma\in W[\Delta,\overline{x}(\alpha\gamma,\beta)\overline{y}]$. Теорема доказана.

Следствие A.6.5.1 *Пусть для каждого $k=0,1,\ldots,|\beta|$ имеет место $W[\Delta,\overline{x}(\alpha,\beta_{1:k})\overline{y}]=\emptyset$ или $W[\Delta,\overline{x}(\gamma,\beta_{k+1:|\beta|})\overline{y}]$. Тогда $W[\Delta,\overline{x}(\alpha\gamma,\beta)\overline{y}]=\emptyset$*

Определение A.6.1 *Пусть заданы строки α', α'', β', β'' и подстановка Δ. Мы будем говорить, что α' и α''* однозначно сопоставляются *с β' и β'' при Δ, если $W[\Delta,(\alpha'\alpha'',\beta'\beta'')] = W[\Delta,(\alpha',\beta')(\alpha'',\beta'')]$.*

Лемма A.6.4 *Пусть α' и α'' однозначно сопоставляются с β' и β'' при Δ. Пусть Ω любая подстановка, такая, что $\Delta \subseteq \Omega$. Тогда α' и α'' однозначно сопоставляются с β' и β'' при Ω.*

Доказательство. Нам нужно показать, что

$$W[\Omega, (\alpha', \beta')(\alpha'', \beta'')] = W[\Omega, (\alpha'\alpha'', \beta'\beta'')]$$

Однако, по теореме A.6.5, всегда

$$W[\Omega, (\alpha', \beta')(\alpha'', \beta'')] \subseteq W[\Omega, (\alpha'\alpha'', \beta'\beta'')]$$

Поэтому достаточно доказать, что

если $\Gamma \in W[\Omega, (\alpha'\alpha'', \beta'\beta'')]$, то $\Gamma \in W[\Omega, (\alpha', \beta')(\alpha'', \beta'')]$.

Пусть $\Gamma \in W[\Omega, (\alpha'\alpha'', \beta'\beta'')]$. Имеем $\Gamma//\alpha'\alpha'' = \beta'\beta''$. Применяя лемму A.6.2, найдем такую $\Gamma \in W[\Delta, (\alpha'\alpha'', \beta'\beta'')]$, что $\Delta \subseteq \Gamma' \subseteq \Gamma$. Имеем $\Gamma'//\alpha'\alpha'' = \beta'\beta''$. Однако, $W[\Delta, (\alpha'\alpha'', \beta'\beta'')] = W[\Delta, (\alpha', \beta')(\alpha'', \beta'')]$. Поэтому $\Gamma//\alpha' = \beta'$ и $\Gamma//\alpha'' = \beta''$. Отсюда $\Gamma \in W[\Omega, (\alpha', \beta')(\alpha'', \beta'')]$. Лемма доказана.

Теорема A.6.6 *Пусть α' и α'' однозначно сопоставляются с β' и β'' при Δ. Тогда*

$$W[\Delta, \overline{x}(\alpha'\alpha'', \beta'\beta'')\overline{y}] = W[\Delta, \overline{x}(\alpha', \beta')(\alpha'', \beta'')\overline{y}]$$

Доказательство. По следствию A.6.1.2 получаем

$$\begin{aligned} W[\Delta, \overline{x}(\alpha'\alpha'', \beta'\beta'')\overline{y}] &= \cup\{W[\Gamma, \overline{x}\,\overline{y}] \mid \Gamma \in W[\Delta, (\alpha'\alpha'', \beta'\beta'')]\} \\ &= \cup\{W[\Gamma, \overline{x}\,\overline{y}] \mid \Gamma \in W[\Delta, (\alpha', \beta')(\alpha'', \beta'')]\} \\ &= W[\Delta, \overline{x}(\alpha', \beta')(\alpha'', \beta'')\overline{y}] \end{aligned}$$

Теорема доказана.

Теорема A.6.7 $W[\Delta, \overline{x}\,\overline{u}\,\overline{v}\,\overline{y}] = W[\Delta, \overline{x}\,\overline{v}\,\overline{u}\,\overline{y}]$.

Доказательство. Пусть $\overline{u} = \overline{\alpha}_u * \overline{\beta}_u$, $\overline{v} = \overline{\alpha}_v * \overline{\beta}_v$. По лемме A.6.1 $\Gamma//\overline{\alpha}_u\overline{\alpha}_v = \overline{\beta}_u\overline{\beta}_v \Leftrightarrow \Gamma//\overline{\alpha}_u = \overline{\beta}_u$ и $\Gamma//\overline{\alpha}_v = \overline{\beta}_v \Leftrightarrow \Gamma//\overline{\alpha}_v\overline{\alpha}_u = \overline{\beta}_v\overline{\beta}_u$. Отсюда следует утверждение теоремы.

A.7 Правила отождествления

До сих пор мы изучали свойства множества $W[\Delta, \overline{x}]$. Однако, в языках программирования при синтаксическом отождествлении обычно не требуется находить все множество $W[\Delta, \overline{x}]$. Вместо этого формулируются какие-то правила, согласно которым для каждой подстановки Δ и вектора пар строк $\overline{x} = \overline{\alpha} * \overline{\beta}$ либо обнаруживается какое-то из отождествлений, либо объявляется, что отождествление потерпело неудачу. Мы формализуем эту ситуацию с помощью следующего определения.

Определение A.7.1 Правилом отождествления *называется такая частичная функция $F[\Delta, \overline{x}]$, которая для каждой подстановки Δ и ряда пар строк $\overline{x}$ либо не определена, либо в качестве значения вырабатывает некоторую подстановку. Причем, если $F[\Delta, \overline{x}]$ определена, то $W[\Delta, \overline{x}] \neq \emptyset$ и $F[\Delta, \overline{x}] \in W[\Delta, \overline{x}]$.*

Из этого определения вытекает, что если $W[\Delta, \overline{x}] = \emptyset$, то $F[\Delta, \overline{x}]$ заведомо не определена. Однако, если $W[\Delta, \overline{x}] \neq \emptyset$, это еще не означает, что $F[\Delta, \overline{x}]$ определена.

Определение A.7.2 *Правило отождествления называется* полным*, если из того, что $W[\Delta, \overline{x}] \neq \emptyset$ следует, что $F[\Delta, \overline{x}]$ определена.*

В языках программирования встречаются как полные, так и неполные правила отождествления.

Уточним теперь, что мы будем подразумевать под алгоритмом отождествления.

Определение A.7.3 *Алгоритмом отождествления для правила отождествления $F[\Delta, \overline{x}]$ называется такой алгоритм, который для каждого $\overline{x}$ и Δ либо вырабатывают некоторую подстановку Γ, либо сообщает, что он потерпел неудачу. Причем, если $F[\Delta, \overline{x}]$ не определена, то алгоритм терпит неудачу, а если $F[\Delta, \overline{x}]$, то $\Gamma = F[\Delta, \overline{x}]$.*

Ясно, что для любого правила отождествления может существовать много разных алгоритмов отождествления, более или менее сложных и более или менее эффективных.

Возникает естественный вопрос: каким способом можно задать $F[\Delta, \overline{x}]$?

Для этого обычно используются два основных метода. Первый способ - это прямо определить $F[\Delta, \overline{x}]$ через некоторый алгоритм отождествления. Таким образом, правило отождествления задано в описании полного [31] рефала. Для базисного рефала в рефал-интерпретаторе используется некоторый алгоритм отождествления, который принимается за определение правила отождествления [26, 27].

Второй способ - это некоторым “аксиоматическим” способом определить $F[\Delta, \overline{x}]$, не описывая алгоритм отождествления явно. Так и сделано в описании базисного рефала (п.1.2.2). В этом случае реализатор языка должен изучить свойства функции $F[\Delta, \overline{x}]$ и на их основе разработать, по возможности, наиболее эффективный алгоритм отождествления.

Цель этой работы состоит в том, чтобы исследовать некоторый класс правил отождествления и на основе изученных свойств разработать алгоритм отождествления.

Определение A.7.4 *Допустим, что на каждом множестве $W[\Delta, \overline{x}]$ задано некоторое отношение линейного порядка. Назовем $F[\Delta, \overline{x}]$* упорядочивающим *правилом отождествления, если оно определено как*

$$F[\Delta, \overline{x}] = \min\{\Gamma \mid \Gamma \in W[\Delta, \overline{x}]\}$$

Очевидно, что любое упорядочивающее правило отождествления является полным. Действительно, если $W[\Delta,\overline{x}] = \emptyset$, то $F[\Delta,\overline{x}]$ не определено, а если $W[\Delta,\overline{x}] \neq \emptyset$, то $F[\Delta,\overline{x}]$ определено и $F[\Delta,\overline{x}] \in W[\Delta,\overline{x}]$.

Правило отождествления, так, как оно определено в базисном рефала, является упорядочивающим.

Частным случаем упорядочивающих правил отождествления являются *левосторонние* правила отождествления, которые определяются в следующем пункте.

A.8 Упорядочение множества $W[\Delta,\overline{x}]$

Определим отношения $\underset{\alpha}{\sim}$ и $\underset{\alpha}{<}$ следующим образом. Пусть α некоторая строка. Обозначим через $M[\alpha]$ множество подстановок

$$M[\alpha] = \{\Delta \mid D[\Delta] \text{ и } \|\alpha\| \subseteq \|\Delta\|\}$$

Определение A.8.1 $\Delta_1 \underset{\alpha}{\sim} \Delta_2$ *равносильно тому, что* $\Delta_1, \Delta_2 \in M[\alpha]$ *и* $|\Delta_1//\alpha_i| = |\Delta_2//\alpha_i|$ *для* $i = 1, 2, \ldots, |\alpha|$.

Определение A.8.2 $\Delta_1 \underset{\alpha}{<} \Delta_2$ *равносильно тому, что* $\Delta_1, \Delta_2 \in M[\alpha]$ *и существует такое* $k \leq |\alpha|$, *что* $|\Delta_1//\alpha_i| = |\Delta_2//\alpha_i|$ *для* $i = 1, 2, \ldots, k-1$ *и* $|\Delta_1//\alpha_k| < |\Delta_2//\alpha_k|$.

Определение A.8.3 $\Delta_1 \underset{\alpha}{\leq} \Delta_2$ *если* $\Delta_1 \underset{\alpha}{\sim} \Delta_2$ *или* $\Delta_1 \underset{\alpha}{<} \Delta_2$.

Теорема A.8.1 *Для любых* $\Delta, \Delta_1, \Delta_2, \Delta_3 \in M[\alpha]$ *справедливы следующие утверждения:*

1. *Либо* $\Delta_1 \underset{\alpha}{\sim} \Delta_2$, *либо* $\Delta_1 \underset{\alpha}{<} \Delta_2$, *либо* $\Delta_2 \underset{\alpha}{<} \Delta_1$.
2. *Всегда* $\Delta_1 \underset{\Box}{\sim} \Delta_2$.
3. *Всегда ложно* $\Delta_1 \underset{\Box}{<} \Delta_2$.
4. $\Delta \underset{\alpha}{\sim} \Delta$.
5. $\Delta_1 \underset{\alpha}{\sim} \Delta_2 \Rightarrow \Delta_2 \underset{\alpha}{\sim} \Delta_1$.
6. $\Delta_1 \underset{\alpha}{\sim} \Delta_2$ *и* $\Delta_2 \underset{\alpha}{\sim} \Delta_3 \Rightarrow \Delta_1 \underset{\alpha}{\sim} \Delta_3$.
7. $\Delta_1 \underset{\alpha}{<} \Delta_2$ *и* $\Delta_2 \underset{\alpha}{<} \Delta_3 \Rightarrow \Delta_1 \underset{\alpha}{<} \Delta_3$.
8. $\Delta \underset{\alpha}{\leq} \Delta$.
9. $\Delta_1 \underset{\alpha}{\leq} \Delta_2$ *и* $\Delta_2 \underset{\alpha}{\leq} \Delta_3 \Rightarrow \Delta_1 \underset{\alpha}{\leq} \Delta_3$.

10. $\Delta_1 \underset{\alpha}{\leq} \Delta_2$ *и* $\Delta_2 \underset{\alpha}{\leq} \Delta_1 \Rightarrow \Delta_1 \underset{\alpha}{\sim} \Delta_2$.

Доказательство. Справедливость этих свойств 1–9 очевидна из определений A.8.1, A.8.2 и A.8.3. Свойство 10 вытекает из свойства 1 и определения A.8.3.

Следствие A.8.1.1 *Отношение* $\underset{\alpha}{\sim}$ *есть отношение эквивалентности на множестве* $M[\alpha]$.

Теорема A.8.2 *Пусть даны строки* α' *и* α'' *и подстановки* Δ_1 *и* Δ_2. *Тогда выполнены свойства:*

1. $\Delta_1 \underset{\alpha'\alpha''}{\sim} \Delta_2 \Leftrightarrow \Delta_1 \underset{\alpha'}{\sim} \Delta_2$ *и* $\Delta_1 \underset{\alpha''}{\sim} \Delta_2$
2. $\Delta_1 \underset{\alpha'\alpha''}{<} \Delta_2 \Leftrightarrow$ *либо* $\Delta_1 \underset{\alpha'}{<} \Delta_2$, *либо* $\Delta_1 \underset{\alpha'}{\sim} \Delta_2$ *и* $\Delta_1 \underset{\alpha''}{<} \Delta_2$.

Доказательство. Очевидно из определений A.8.1 и A.8.2.

Дальше нам понадобится следующая очевидная лемма, которую мы сформулируем без доказательства.

Лемма A.8.1 *Пусть* $\alpha', \alpha'', \beta', \beta''$ *такие строки, что* $\alpha'\beta' = \alpha''\beta''$. *Тогда, если* $|\alpha'| = |\alpha''|$, *то* $\alpha' = \alpha''$ *и* $\beta' = \beta''$. *А если* $|\alpha'| < |\alpha''|$, *то существует такая строка* $\delta \neq \Box$, *что* $\alpha'\delta = \alpha''$.

Лемма A.8.2 *Пусть* $\Delta_1, \Delta_2 \in M[\alpha]$ *и* $\Delta_1//\alpha = \Delta_2//\alpha$. *Тогда из* $\Delta_1 \underset{\alpha}{\sim} \Delta_2$ *следует, что*

$$\Delta_1//\alpha_i = \Delta_2//\alpha_i \text{ для } i = 1, 2, \ldots, |\alpha|$$

Доказательство. Проведем индукцией по длине строки α. Сначала заметим, что при $|\alpha| = 0$ лемма тривиально истинна. Пусть теперь она истинна для всех случаев строк γ таких, что $|\gamma| < |\alpha|$. Тогда по теореме A.8.2 получаем:

$\Delta_1 \underset{\alpha}{\sim} \Delta_2 \Rightarrow \Delta_1 \underset{\alpha_1}{\sim} \Delta_2$ и $\Delta_1 \underset{\alpha_{2:|\alpha|}}{\sim} \Delta_2$.

Из $\Delta_1 \underset{\alpha_1}{\sim} \Delta_2$ следует, что $|\Delta_1//\alpha_1| = |\Delta_2//\alpha_1|$.

Из $\Delta_1 \underset{\alpha_{2:|\alpha|}}{\sim} \Delta_2$ следует, что $|\Delta_1//\alpha_{2:|\alpha|}| = |\Delta_2//\alpha_{2:|\alpha|}|$.

При этом $[\Delta_1//\alpha_1][\Delta_1//\alpha_{2:|\alpha|}] = [\Delta_2//\alpha_1][\Delta_2//\alpha_{2:|\alpha|}]$. Поэтому, применяя лемму A.8.1 получаем

$$\Delta_1//\alpha_1 = \Delta_2//\alpha_1 \text{ и } \Delta_1//\alpha_{2:|\alpha|} = \Delta_2//\alpha_{2:|\alpha|}$$

Теперь, применяя индуктивное предположение, получаем

$$\Delta_1//\alpha_i = \Delta_2//\alpha_i \text{ при } i = 2, \ldots, |\alpha|$$

Теорема доказана.

Лемма A.8.3 *Пусть* $\Delta_1, \Delta_2 \in M[\alpha]$ *и* $\Delta_1//\alpha = \Delta_2//\alpha$. *Тогда из* $\Delta_1 \underset{\alpha}{<} \Delta_2$ *следует, что существует такое* k, *и строка* $\delta \neq \Box$, *что*

$$\Delta_1//\alpha_i = \Delta_2//\alpha_i \text{ для } i = 1, 2, \ldots, k-1$$
$$[\Delta_1//\alpha_k]\delta = \Delta_2//\alpha_k$$

Доказательство. Из $\Delta_1 \underset{\alpha}{<} \Delta_2$ вытекает, что существует такое $k \leq |\alpha|$, что

$$|\Delta_1//\alpha_i| = |\Delta_2//\alpha_i| \text{ для } i = 1, 2, \ldots, k-1$$
$$|\Delta_1//\alpha_k| < \Delta_2//\alpha_k$$

Отсюда $|\Delta_1//\alpha_{1:k-1}| = |\Delta_2//\alpha_{1:k-1}|$. Поскольку $\Delta_1//\alpha = \Delta_2//\alpha$, имеем $[\Delta_1//\alpha_{1:k-1}][\Delta_1//\alpha_{k:|\alpha|}] = [\Delta_2//\alpha_{1:k-1}][\Delta_2//\alpha_{k:|\alpha|}]$. Поэтому, по лемме A.8.1, $\Delta_1//\alpha_{1:k-1} = \Delta_2//\alpha_{1:k-1}$ и $\Delta_1//\alpha_{k:|\alpha|} = \Delta_2//\alpha_{k:|\alpha|}$. Отсюда следует, что $\Delta_1 \underset{\alpha_{1:k-1}}{\sim} \Delta_2$ и $\Delta_1//\alpha_{1:k-1} = \Delta_2//\alpha_{1:k-1}$. Поэтому, применяя лемму A.8.2, получаем

$$\Delta_1//\alpha_i = \Delta_2//\alpha_i \text{ для } i = 1, 2, \ldots, k-1$$

Теперь заметим, что

$$[\Delta_1//\alpha_k][\Delta_1//\alpha_{k+1:|\alpha|}] = [\Delta_2//\alpha_k][\Delta_2//\alpha_{k+1:|\alpha|}]$$

и при этом $|\Delta_1//\alpha_k| < |\Delta_2//\alpha_k|$. Отсюда, по лемме A.8.1 получаем, что существует $\delta \neq \Box$, такое, что

$$[\Delta_1//\alpha_k]\delta = \Delta_2//\alpha_k$$

Лемма доказана.

Теорема A.8.3 *Пусть* $\Delta_1, \Delta_2 \in W[\Delta, (\alpha, \beta)]$. *Тогда* $\Delta_1 \underset{\alpha}{\sim} \Delta_2 \Rightarrow \Delta_1 = \Delta_2$.

Доказательство. Поскольку $\Delta_1//\alpha = \Delta_2//\alpha = \beta$, по лемме A.8.2 имеем

$$\Delta_1//\alpha_i = \Delta_2//\alpha_i \text{ для } i = 1, 2, \ldots, |\alpha|$$

Заметим сначала, что в силу $D[\Delta_1]$, $D[\Delta_2]$ и $\|\Delta_1\| = \|\Delta_2\| = \|\Delta\| \cup \|\alpha\|$ для каждого $s \in \|\Delta_1\|$ существует ровно одна строка γ, такая, что $(s, \gamma) \in \Delta_1$. То же самое верно и для Δ_2.

Пусть теперь $(s, \gamma) \in \Delta_1$. Покажем, что тогда $(s, \gamma) \in \Delta_2$ В самом деле, если $s \in \|\Delta\|$, то $(s, \gamma) \in \Delta_2$ в силу того, что $(s, \gamma) \in \Delta$ и $\Delta \subseteq \Delta_2$. А если $s \notin \|\Delta\|$, то $s \in \|\alpha\| \setminus \|\Delta\|$. Поэтому существует такое j, что $\Delta_1//\alpha_j = \gamma$. Однако, $\Delta_1//\alpha_j = \Delta_2//\alpha_j$. Откуда следует, что $(s, \gamma) \in \Delta_2$.

Аналогично доказывается, что, если $(s, \gamma) \in \Delta_2$, то $(s, \gamma) \in \Delta_1$

Теорема доказана.

Следствие A.8.3.1 $\Delta_1 \in W[\Delta, (\alpha, \beta)]$ *и* $\Delta_2 \in W[\Delta, (\alpha, \beta)]$ *и* $\Delta_1 \underset{\alpha}{\leq} \Delta_2$ *и* $\Delta_2 \underset{\alpha}{\leq} \Delta_1 \Rightarrow \Delta_1 = \Delta_2$

Следствие A.8.3.2 *Отношение* $\underset{\alpha}{\leq}$ *на множестве* $W[\Delta, (\alpha, \beta)]$ *является отношением линейного порядка.*

Теперь обобщим полученные результаты на случай векторов строк.

Пусть $\overline{\alpha} = (\alpha^1, \alpha^2, \ldots, \alpha^n)$ вектор строк. Обозначим через $M[\overline{\alpha}]$ множество подстановок

$$M[\overline{\alpha}] = \{\Delta \mid D[\Delta] \text{ и } \|\overline{\alpha}\| \subseteq \|\Delta\|\}$$

Очевидно, что $M[\overline{\alpha}] = M[\alpha^1\alpha^2 \ldots \alpha_n]$

Определение A.8.4 $\Delta_1 \underset{\overline{\alpha}}{\sim} \Delta_2$ *равносильно тому, что* $\Delta_1 \underset{\alpha}{\sim} \Delta_2$, *где* $\alpha = \alpha^1\alpha^2 \ldots \alpha^n$.

Определение A.8.5 $\Delta_1 \underset{\overline{\alpha}}{<} \Delta_2$ *равносильно тому, что* $\Delta_1 \underset{\alpha}{<} \Delta_2$, *где* $\alpha = \alpha^1\alpha^2 \ldots \alpha^n$.

Определение A.8.6 $\Delta_1 \underset{\overline{\alpha}}{\leq} \Delta_2$ *равносильно тому, что* $\Delta_1 \underset{\alpha}{\leq} \Delta_2$, *где* $\alpha = \alpha^1\alpha^2 \ldots \alpha^n$.

Теорема A.8.4 *Для любых подстановок* Δ, Δ_1, Δ_2, Δ_3 *справедливы утверждения:*

1. *Либо* $\Delta_1 \underset{\overline{\alpha}}{\sim} \Delta_2$, *либо* $\Delta_1 \underset{\overline{\alpha}}{<} \Delta_2$, *либо* $\Delta_2 \underset{\overline{\alpha}}{<} \Delta_1$
2. $\Delta \underset{\overline{\alpha}}{\sim} \Delta$
3. $\Delta_1 \underset{\overline{\alpha}}{\sim} \Delta_2 \Rightarrow \Delta_2 \underset{\overline{\alpha}}{\sim} \Delta_1$
4. $\Delta_1 \underset{\overline{\alpha}}{\sim} \Delta_2$ *и* $\Delta_2 \underset{\overline{\alpha}}{\sim} \Delta_3 \Rightarrow \Delta_1 \underset{\overline{\alpha}}{\sim} \Delta_3$
5. $\Delta_1 \underset{\overline{\alpha}}{<} \Delta_2$ *и* $\Delta_2 \underset{\overline{\alpha}}{<} \Delta_3 \Rightarrow \Delta_1 \underset{\overline{\alpha}}{<} \Delta_3$
6. $\Delta \underset{\overline{\alpha}}{\leq} \Delta$
7. $\Delta_1 \underset{\overline{\alpha}}{\leq} \Delta_2$ *и* $\Delta_2 \underset{\overline{\alpha}}{\leq} \Delta_3 \Rightarrow \Delta_1 \underset{\overline{\alpha}}{\leq} \Delta_3$
8. $\Delta_1 \underset{\overline{\alpha}}{\leq} \Delta_2$ *и* $\Delta_2 \underset{\overline{\alpha}}{\leq} \Delta_1 \Rightarrow \Delta_1 \underset{\overline{\alpha}}{\sim} \Delta_2$

Доказательство. Справедливость этих свойств вытекает из результатов теоремы A.8.1 и определений A.8.4, A.8.5, A.8.6.

Следствие A.8.4.1 *Отношение* $\underset{\overline{\alpha}}{\sim}$ *есть отношение эквивалентности на множестве* $M[\overline{\alpha}]$.

Теорема A.8.5 *Пусть даны векторы строк $\overline{\alpha'}$ и $\overline{\alpha''}$ и подстановки Δ_1 и Δ_1. Тогда выполнены свойства:*

1. $\Delta_1 \underset{\overline{\alpha'}\,\overline{\alpha''}}{\sim} \Delta_2 \Leftrightarrow \Delta_1 \underset{\overline{\alpha'}}{\sim} \Delta_2$ *и* $\Delta_1 \underset{\overline{\alpha''}}{\sim} \Delta_2$

2. $\Delta_1 \underset{\overline{\alpha'}\,\overline{\alpha''}}{<} \Delta_2 \Leftrightarrow$ *либо* $\Delta_1 \underset{\overline{\alpha'}}{\leq} \Delta_2$, *либо* $\Delta_1 \underset{\overline{\alpha'}}{\sim} \Delta_2$ *и* $\Delta_1 \underset{\overline{\alpha''}}{\leq} \Delta_2$

Доказательство. Справедливость свойств вытекает из определений A.8.4 и A.8.5 и теоремы A.8.1.

Теорема A.8.6 *Пусть* $\Delta_1, \Delta_2 \in W[\Delta, \overline{\alpha} * \overline{\beta}]$. *Тогда* $\Delta_1 \underset{\overline{\alpha}}{\sim} \Delta_2 \Leftrightarrow \Delta_1 = \Delta_2$

Доказательство. Пусть $\overline{\alpha} = (\alpha^1, \alpha^2, \ldots, \alpha^n)$, $\overline{\beta} = (\beta^1, \beta^2, \ldots, \beta^n)$, $\alpha = \alpha^1 \alpha^2 \ldots \alpha^n$, $\beta = \beta^1 \beta^2 \ldots \beta^n$.

Тогда, по теореме A.5.1, $W[\Delta, \overline{\alpha} * \overline{\beta}] \subseteq W[\Delta, (\alpha, \beta)]$, а по определению A.8.4 $\Delta_1 \underset{\overline{\alpha}}{\sim} \Delta_2 \Leftrightarrow \Delta_1 \underset{\alpha}{\sim} \Delta_2$.

Поэтому можно применить теорему A.8.3, откуда $\Delta_1 = \Delta_2$.

Следствие A.8.6.1 $\Delta_1, \Delta_2 \in W[\Delta, \overline{\alpha} * \overline{\beta}]$ *и* $\Delta_1 \underset{\overline{\alpha}}{\leq} \Delta_2$ *и* $\Delta_2 \underset{\overline{\alpha}}{\leq} \Delta_1 \Rightarrow \Delta_1 = \Delta_2$.

Следствие A.8.6.2 *Отношение* $\underset{\overline{\alpha}}{\leq}$ *на множестве* $W[\Delta, \overline{\alpha} * \overline{\beta}]$ *есть отношение линейного порядка.*

Лемма A.8.4 *Пусть* $\Delta_1, \Delta_2, \Omega_1, \Omega_2 \in M[\alpha]$, *причем* $\Delta_1 \subseteq \Omega_1$ *и* $\Delta_2 \subseteq \Omega_2$. *Тогда*

1. $\Delta_1 \underset{\alpha}{\sim} \Delta_2 \Leftrightarrow \Omega_1 \underset{\alpha}{\sim} \Omega_2$

2. $\Delta_1 \underset{\alpha}{<} \Delta_2 \Leftrightarrow \Omega_1 \underset{\alpha}{<} \Omega_2$

Доказательство. По лемме A.6.3 имеем $\Delta_1 // \alpha_i = \Omega_1 // \alpha_i$ и $\Delta_2 // \alpha_i = \Omega_2 // \alpha_i$ при $i = 1, 2, \ldots, |\alpha|$. Отсюда немедленно следует справедливость леммы.

Лемма A.8.5 *Пусть* $\Delta_1, \Delta_2, \Omega_1, \Omega_2 \in M[\overline{\alpha}]$, *причем* $\Delta_1 \subseteq \Omega_1$ *и* $\Delta_2 \subseteq \Omega_2$. *Тогда*

1. $\Delta_1 \underset{\overline{\alpha}}{\sim} \Delta_2 \Leftrightarrow \Omega_1 \underset{\overline{\alpha}}{\sim} \Omega_2$

2. $\Delta_1 \underset{\overline{\alpha}}{\leq} \Delta_2 \Leftrightarrow \Omega_1 \underset{\overline{\alpha}}{\leq} \Omega_2$

Доказательство. Немедленно следует из определений A.8.4, A.8.5 и леммы A.8.4.

A.9 Левосторонние правила отождествления

Определение A.9.1 *Пусть на множестве* $W[\Delta, \overline{\alpha} * \overline{\beta}]$ *введено отношение линейного порядка* $\underset{\overline{\overline{\alpha}}}{\leq}$ *. Тогда левосторонним правилом отождествления будет называться частичная функция* $L[\Delta, \overline{\alpha} * \overline{\beta}]$*, определенная следующим образом:*

$$L[\Delta, \overline{\alpha} * \overline{\beta}] = \min\{\Omega | \Omega \in W[\Delta, \overline{x}]\}$$

Очевидно, что левостороннее правило отождествления является упорядочивающим и полным в смысле определений A.7.2 и A.7.4.

Наша дальнейшая задача – изучать свойства $L[\Delta, \overline{\alpha} * \overline{\beta}]$ и на этой основе разработать алгоритм отождествления для левосторонних правил отождествления.

Свойства $L[\Delta, \overline{x}]$ мы будем формулировать в виде равенств, которые всюду будут истолковываться следующим образом. Если одна из частей равенства определена, то определена и его другая часть и их значения совпадают. Если же не определена одна из частей равенства, то не определена и другая часть.

A.10 Теоремы о разложении $L[\Delta, \overline{x}]$

Теорема A.10.1 $$L[\Delta, \emptyset] = \begin{cases} \Delta, & \textit{при } D[\Delta] \\ \textit{не определено,} & \textit{при не } D[\Delta] \end{cases}$$

Доказательство. По теореме A.6.2 $W[\Delta, \overline{x}] = \emptyset$, если не $D[\Delta]$, и $W[\Delta, \overline{x}] = \Delta$, если $D[\Delta]$. Отсюда следует утверждение теоремы.

Теорема A.10.2 $$L[\Delta, \overline{x}(\square, \beta)\ \overline{y}] = \begin{cases} L[\Delta, \overline{x}\ \overline{y}] & \textit{при } \beta = \square \\ \textit{не определено,} & \textit{при } \beta \neq \square \end{cases}$$

Доказательство. Пусть $\beta \neq \square$. Тогда по теореме A.6.3 $W[\Delta, \overline{x}(\square, \beta)\ \overline{y}] = \emptyset$. Следовательно $L[\Delta, \overline{x}\ \overline{y}]$ не определено.

Пусть теперь $\beta = \square$. Тогда по теореме A.6.3 $W[\Delta, \overline{x}(\square, \square)\ \overline{y}] = w[\Delta, \overline{x}\ \overline{y}]$. Таким образом области определения $L[\Delta, \overline{x}(\square, \square)\ \overline{y}]$ и $L[\Delta, \overline{x}\ \overline{y}]$ совпадают. Поэтому осталось рассмотреть случай, когда $W[\Delta, \overline{x}(\square, \square)\ \overline{y}] = w[\Delta, \overline{x}\ \overline{y}] \neq \emptyset$.

Пусть $\overline{x} = \overline{\alpha_x} * \overline{\beta_x}$, $\overline{y} = \overline{\alpha_y} * \overline{\beta_y}$. Тогда из определения A.8.5 вытекает, что для любых $\Omega_1, \Omega_2 \in W[\Delta, \overline{x}\ \overline{y}]$ имеет место

$$\Omega_1 \underset{\overline{\alpha_x}(\square)\overline{\alpha_y}}{<} \Omega_2 \Leftrightarrow \Omega_1 \underset{\overline{\alpha_x}\ \overline{\alpha_y}}{<} \Omega_2$$

Отсюда $L[\Delta, \overline{x}(\square\square)\overline{y}] = L[\Delta, \overline{x}\ \overline{y}]$. Теорема доказана.

Теорема A.10.3 *Пусть s – некий символ. Тогда*

$$L[\Delta, \overline{x}(s, \beta)\ \overline{y}] = L[\Delta \cup \{(s, \beta)\},\ \overline{x}\ \overline{y}]$$

Доказательство. По теореме A.6.4 имеем

$$W[\Delta, \overline{x}(s,\beta)\ \overline{y}] = W[\Delta \cup \{(s,\beta)\},\ \overline{x}\ \overline{y}]$$

из чего следует, что области определения $L[\Delta, \overline{x}(s,\beta)\overline{y}]$ и $L[\Delta \cup \{(s,\beta)\},\ \overline{x}\ \overline{y}]$ совпадают. Поэтому осталось рассмотреть случай, когда

$$W[\Delta, \overline{x}(s,\beta)\ \overline{y}] = W[\Delta \cup \{(s,\beta)\},\ \overline{x}\ \overline{y}] \neq \emptyset$$

Пусть $\overline{x} = \overline{\alpha_x} * \overline{\beta_x}$, $\overline{y} = \overline{\alpha_y} * \overline{\beta_y}$. Рассмотрим две подстановки $\Omega_1, \Omega_2 \in W[\Delta, \overline{x}(s,\beta)\ \overline{y}]$. Ясно, что $\Omega_1 \underset{\alpha}{\sim} \Omega_2$, ибо $\Omega_1//s = \Omega_2//s = \beta$. Поэтому не может быть $\Omega_1 \underset{s}{<} \Omega_2$. Отсюда, по теореме A.8.5 $\Omega_1 \underset{\overline{\alpha}_x(s)\overline{\alpha}_y}{<} \Omega_2 \Leftrightarrow$ либо $\Omega_1 \underset{\overline{\alpha}_x}{<} \Omega_2$, либо $\Omega_1 \underset{\overline{\alpha}_x}{\sim} \Omega_2$ и $\Omega_1 \underset{\overline{\alpha}_y}{<} \Omega_2 \Leftrightarrow \Omega_1 \underset{\overline{\alpha}_x\ \overline{\alpha}_y}{<} \Omega_2$.

Отсюда $L[\Delta, \overline{x}(s,\beta)\ \overline{y}] = L[\Delta \cup \{(s,\beta)\},\ \overline{x}\ \overline{y}]$. Теорема доказана.

Теорема A.10.4 $L[\Delta, (s\ \alpha, \beta)\overline{y}] = L[\Delta \cup \{(s, \beta_{1:k})\}, (\alpha, \beta_{k+1:|\beta|})\ \overline{y}]$, *где*

$$k = \min\{i | W[\Delta \cup \{(s, \beta_{1:i})\},\ (\alpha, \beta_{i+1:|\beta|})\ \overline{y}] \neq \emptyset\}$$

Доказательство. Обозначим $W = W[\Delta, (s\ \alpha, \beta)\ \overline{y}]$,

$$W_i = W[\Delta \cup \{(s, \beta_{1:i})\},\ (\alpha, \beta_{i+1:|\beta|})\ \overline{y}]$$

Тогда по теоремам A.6.5 и A.6.4 имеем

$$W = \bigcup_{i=0}^{|\beta|} W[\Delta, (s, \beta_{1:i})(\alpha, \beta_{i+1:|\beta|})\ \overline{y}] = \bigcup_{i=0}^{|\beta|} W_i$$

Если $W_i = 0$ для $i = 0, 1, \ldots, |\beta|$, то $W = \emptyset$. В то же время $k = \min\{i | W_i \neq \emptyset\}$ не определено. Поэтому области определения обеих частей равенств в условиях теоремы совпадают.

Рассмотрим случай, когда $W \neq \emptyset$.

Пусть Ω_1 и Ω_2 такие подстановки, что $\Omega_1 \in W_i$, а $\Omega_2 \in W_j$. Покажем, что если $i < j$, то $\Omega_1 \underset{(s\ \alpha)\overline{\alpha}_y}{<} \Omega_2$.

В самом деле $|\Omega_1//s| = |\beta_{1:i}| = i, |\Omega_2//s| = |\beta_{1:j}| = j$. Поэтому, если $i < j$, то $\Omega_1 \underset{s}{<} \Omega_2$. Отсюда $\Omega_1 \underset{s\ \alpha}{<} \Omega_2$ и $\Omega_1 \underset{(s\ \alpha)\overline{\alpha}_y}{<} \Omega_2$

Пусть $L[\Delta, (s\ \alpha, \beta)\ \overline{y}] \in W_i$. Тогда для любого $\Omega \in W_j$ имеет место $L[\Delta, (s\ \alpha, \beta)\ \overline{y}] \underset{(s\ \alpha)\overline{\alpha}_y}{<} \Omega$, откуда $i \leq j$, что и доказывает теорему.

Теорема A.10.5 *Пусть α' и α'' однозначно сопоставляются с β' и β'' при Δ. Тогда*

$$L[\Delta, \overline{x}(\alpha'\ \alpha'', \beta'\ \beta'')\ \overline{y}] = L[\Delta, \overline{x}(\alpha', \beta')(\alpha'', \beta'')\ \overline{y}]$$

Доказательство. По теореме A.6.6

$$W[\Delta, \overline{x}(\alpha'\ \alpha'', \beta'\ \beta'')\ \overline{y}] = W[\Delta, \overline{x}(\alpha', \beta')(\alpha'', \beta'')\ \overline{y}].$$

Отсюда следует, что области определения обеих частей равенства в утверждении теоремы совпадают. Поэтому осталось рассмотреть случай, когда $W[\Delta, \overline{x}(\alpha'\ \alpha'', \beta'\ \beta'')\ \overline{y}] \neq \emptyset$.

Пусть $\overline{x} = \overline{\alpha}_x * \overline{\beta}_x, \overline{y} = \overline{\alpha}_y * \overline{\beta}_y$, где $\overline{\alpha}_x = (\alpha_x^1, \alpha_x^2, \ldots, \alpha_x^n)$, $\overline{\alpha}_y = (\alpha_y^1, \alpha_y^2, \ldots, \alpha_y^m)$. Обозначим через δ строку $\delta = \alpha_x^1\ \alpha_x^2 \ldots\ \alpha_x^n\ \alpha'\ \alpha''\ \alpha_y^1\ \alpha_y^2 \ldots\ \alpha_y^m$. Тогда, вследствие определения A.8.5 для любых $\Omega_1, \Omega_2 \in W[\Delta, \overline{x}(\alpha'\ \alpha'', \beta'\ \beta'')\ \overline{y}]$ получаем

$$\Omega_1 \underset{\overline{\alpha}_x(\alpha',\alpha'')\overline{\alpha}_y}{<} \Omega_2 \Leftrightarrow \Omega_1 \underset{\delta}{<} \Omega_2 \Leftrightarrow \Omega_1 \underset{\overline{\alpha}_x(\alpha'\ \alpha'')\overline{\alpha}_y}{<} \Omega_2$$

Отсюда следует, что множества

$$W[\Delta, \overline{x}(\alpha'\ \alpha'', \beta'\ \beta'')\ \overline{y}] \text{ и } W[\Delta, \overline{x}(\alpha', \beta')(\alpha'', \beta'')\ \overline{y}]$$

упорядочены одинаково. Теорема доказана.

Теорема A.10.6 $L[\Delta, \overline{x}\ \overline{y}] = L[\Gamma_x, \overline{y}]$, *где*

$$\Gamma_x = \min\{\Gamma \in W[\Delta, \overline{x}] \mid W[\Gamma, \overline{y}] \neq \emptyset\}$$

Доказательство. По следствию A.6.1.2 $W[\Delta, \overline{x}\ \overline{y}] = \cup\{W[\Gamma, \overline{y}] | \Gamma \in W[\Delta, \overline{x}]\}$. Поэтому, если для всех $\Gamma \in W[\Delta, \overline{x}]$ имеет место $W[\Gamma, \overline{y}] = \emptyset$, то $W[\Delta, \overline{x}\ \overline{y}] = \emptyset$ и Γ_x не определена. Поэтому области определения обеих частей равенства $L[\Delta, \overline{x}\ \overline{y}] = L[\Gamma_x, \overline{y}]$ совпадают.

Рассмотрим случай, когда $W[\Delta, \overline{x}\ \overline{y}] \neq \emptyset$. Обозначим $\Gamma_{xy} = L[\Gamma_x, \overline{y}]$, $\Omega = L[\Delta, \overline{x}\ \overline{y}]$. Тогда по теореме A.6.1 $\Gamma_{xy} = \Gamma_x \cup \Gamma_y$, $\Omega_x \cup \Omega_y$, где

$$\Gamma_y \in W[\Delta, \overline{y}],\ \ \Omega_x \in W[\Delta, \overline{x}],\ \ \Omega_y \in W[\Delta, \overline{y}]$$

Докажем, что $\Gamma_{xy} = \Omega$. Пусть $\overline{x} = \overline{\alpha}_x * \overline{\beta}_x$, $\overline{y} = \overline{\alpha}_y * \overline{\beta}_y$. Предположим, от противного, что $\Gamma_{xy} \neq \Omega$. Тогда $\Omega \underset{\overline{\alpha}_x\ \overline{\alpha}_y}{<} \Gamma_{xy}$. Тогда $\Omega \underset{\overline{\alpha}_x\ \overline{\alpha}_y}{<} \Gamma_{xy}$. Откуда по теореме A.8.5 либо $\Omega \underset{\overline{\alpha}_x}{<} \Gamma_{xy}$, либо $\Omega \underset{\overline{\alpha}_x}{\sim} \Gamma_{xy}$ и $\Omega \underset{\overline{\alpha}_y}{<} \Gamma_{xy}$.

Пусть $\Omega \underset{\overline{\alpha}_x}{<} \Gamma_{xy}$. Тогда по лемме A.8.4 $\Omega_x \underset{\overline{\alpha}_x}{<} \Gamma_x$, что невозможно по определению Γ_x. Поэтому $\Omega \underset{\overline{\alpha}_x}{\sim} \Gamma_{xy}$, откуда по лемме A.8.4 $\Omega_x \underset{\overline{\alpha}_x}{\sim} \Gamma_x$, откуда по теореме A.8.6 $\Omega_x = \Gamma_x$. Т.о. должно быть $\Omega = \Gamma_x \cup \Omega_y$, $\Omega \in W[\Gamma_x, \overline{y}]$, $\Omega \underset{\overline{\alpha}_y}{<} \Gamma_{xy}$. Однако, $\Omega \underset{\overline{\alpha} y}{<} \Gamma_{xy}$ невозможно, так как $\Gamma_{xy} = L[\Gamma_x, \overline{y}]$. Т.о. $\Gamma = \Omega$. Теорема доказана.

A.11 Независимые множества символов

Определение A.11.1 *Пусть* $Y, X_1, X_2, \ldots, X_n$ *некоторые множества символов. Будем говорить, что* $X_1, X_2, \ldots, X_n$ *независимы по* Y*, если для любых подстановок* $\Omega, \Delta_1, \Delta_2, \ldots, \Delta_n$ *таких, что* $\|\Omega\| = Y$ *и* $\|\Delta_i\| \subseteq X_i$ *для* $i =$

$1, 2, \ldots, n$ *из истинности* $D[\Omega \cup \Delta_i]$ *при* $i = 1, 2, \ldots, n$ *вытекает истинность* $D[\Omega \cup \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n]$.

Тот факт, что $X_1, X_2, \ldots, X_n$ *независимы по* Y*, мы будем записывать как* $Y \bot (X_1, X_2, \ldots, X_n)$

Теорема А.11.1 *Отношение независимости обладает следующими свойствами:*

1. $Y \bot (X_1)$
2. $Y \bot (X_1, \ldots, X_m, X_{m+1}, \ldots, X_n) \Rightarrow Y \bot (X_1, \ldots, X_{m+1}, X_m, \ldots, X_n)$
3. $X'_1 \subseteq X_1, X'_2 \subseteq X_2, \ldots, X'_n \subseteq X_n$ *и*
   $Y \bot (X_1, X_2, \ldots, X_n) \Rightarrow Y \bot (X'_1, X'_2, \ldots, X'_n)$
4. $Y \bot (X_1, X_2, \ldots, X_n) \Rightarrow Y \bot (X_2, \ldots, X_n)$

Доказательство. Свойства непосредственно вытекают из определения А.11.1

Теорема А.11.2 $Y \bot (X_1, X_2, \ldots, X_n) \Rightarrow Y \bot (X_1 \cup X_2, \ldots, X_n)$

Доказательство. Пусть $\Omega, \Delta_{1\,2}, \Delta_3, \ldots, \Delta_n$ такие подстановки, что $\|\Omega\| = Y$, $\|\Delta_{1\,2}\| \subseteq X_1 \cup X_2,\ D[\Omega \cup \Delta_{1\,2}]$ и $\|\Delta_i\| \subseteq X_i,\ D[\Omega \cup \Delta_i]$ при $i = 3, 4, \ldots, n$.

Докажем, что тогда $D[\Omega \cup \Delta_{1\,2} \cup \Delta_3 \cup \ldots \cup \Delta_n]$.

Для этого представим $\Delta_{1\,2}$ в виде $\Delta_{1\,2} = \Delta_1 \cup \Delta_2$, где $\|\Delta_1\| \subseteq X_1,\ \|\Delta_2\| \subseteq X_2$.

Тогда, принимая во внимание монотонность $D[\Delta]$ и истинность $D[\Omega \cup \Delta_{1\,2}]$, получаем, что $D[\Omega \cup \Delta_1]$ и $D[\Omega \cup \Delta_2]$.

Т.о. $\|\Omega\| = Y$ и $\|\Delta_i\| \subseteq X_i,\ D[\Omega \cup \Delta_i]$ при $i = 1, 2, \ldots, n$. Откуда, используя $Y \bot (x_1 \ldots X_n)$ получаем $D[\Omega \cup \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n]$ Однако $\Delta_1 \cup \Delta_2 = \Delta_{1\,2}$, поэтому $D[\Omega \cup \Delta_{1\,2} \cup \Delta_3 \cup \ldots \cup \Delta_n]$, откуда следует утверждение теоремы.

Теорема А.11.3 *Пусть* $Y_0 \subseteq Y \subseteq Y_0 \cup X_1 \cup X_2 \cup \ldots \cup X_n$ *и* $Y_0 \bot (X_1, X_2, \ldots, X_n)$. *Тогда* $Y \bot (X_1, X_2, \ldots, X_n)$

Доказательство. Пусть $\Omega, \Delta_1, \Delta_2, \ldots, \Delta_n$ такие подстановки, что $\|\Omega\| = Y$ и $\|\Delta_i\| \subseteq X_i$ для $i = 1, 2, \ldots, n$. Покажем, что тогда из $D[\Omega \cup \Delta_i]$ при $i = 1, 2, \ldots, n$ вытекает $D[\Omega \cup \Delta_1 \cup \ldots \cup \Delta_n]$.

Для этого представим Ω в виде $\Omega = \Omega_0 \cup \Omega_1 \cup \ldots \cup \Omega_n$, где $\|\Omega_0\| = Y_0,\ \|\Omega_i\| \subseteq \|\Delta_i\|$ для $i = 1, 2, \ldots, n$.

Тогда $D[\Omega \cup \Delta_i] \Rightarrow D[\Omega_0 \cup \Omega_1 \cup \ldots \cup \Omega_n \cup \Delta_i] \Rightarrow D[\Omega_0 \cup \Omega_i \cup \Delta_i]$ вследствие монотонности $D[\Delta]$. Теперь $\|\Omega_i \cup \Delta_i\| \subseteq X_i,\ \|\Omega_0\| = Y$ и $D[\Omega_0 \cup [\Omega_i \cup \Delta_i]]$. Поэтому в силу $Y_0 \bot (X_1, \ldots, X_n)$ получаем

$$D[\Omega_0 \cup [\Omega_1 \cup \Delta_1] \cup \ldots \cup [\Omega_n \cup \Delta_n]\,] \Rightarrow$$
$$D[\,[\Omega_0 \cup \Omega_1 \cup \ldots \cup \Omega_n] \cup \Delta_1 \cup \ldots \cup \Delta_n] \Rightarrow$$
$$D[\Omega \cup \Delta_1 \ldots \cup \Delta_n]$$

Теорема доказана.

A.12 Использование независимости множеств символов для разложения $L[\Delta, \overline{x}]$

Теорема A.12.1 *Пусть* $\|\Delta\| \perp (\|\overline{x}\|, \|\overline{y}\|)$. *Тогда* $\Gamma_x \in W[\Delta, \overline{x}]$ *и* $\Gamma_y \in W[\Delta, \overline{y}] \Rightarrow \Gamma_x \cup \Gamma_y \in W[\Delta, \overline{x}\,\overline{y}]$.

Доказательство. Из $\|\Delta\| \perp (\|\overline{x}\|, \|\overline{y}\|)$ вытекает, что $D[\Delta \cup \Gamma_x \cup \Gamma_y]$. По лемме A.6.3 $\Gamma_x // \overline{\alpha}_x = \overline{\beta}_x \Rightarrow \Gamma_x \cup \Gamma_y // \overline{\alpha}_x = \overline{\beta}_x$, а также $\Gamma_y // \overline{\alpha}_y = \overline{\beta}_y \Rightarrow \Gamma_x \cup \Gamma_y // \overline{\alpha}_y = \overline{\beta}_y$. Отсюда $\Gamma_x \cup \Gamma_y \in W[\Delta, \overline{x}\,\overline{y}]$. Теорема доказана.

Теорема A.12.2 *Пусть* $\|\Delta\| \perp (\|\overline{x}\|, \|\overline{y}\|)$. *Тогда* $L[\Delta, \overline{x}\,\overline{y}] = L[\Delta, \overline{x}] \cup L[\Delta, \overline{y}]$.

Доказательство. Сначала покажем, что области определения обеих частей равенства совпадают.

По теореме A.12.1 $W[\Delta, \overline{x}] \neq \emptyset$ и $W[\Delta, \overline{y}] \neq \emptyset \Rightarrow W[\Delta, \overline{x}\,\overline{y}] \neq \emptyset$.

А по следствию A.6.1.1 $W[\Delta, \overline{x}] = \emptyset$ или $W[\Delta, \overline{y}] = \emptyset \Rightarrow W[\Delta, \overline{x}\,\overline{y}] = \emptyset$.

Таким образом, осталось рассмотреть случай, когда $W[\Delta, \overline{x}] \neq \emptyset, W[\Delta, \overline{y}] \neq \emptyset$ и $W[\Delta, \overline{x}\,\overline{y}] \neq \emptyset$

Обозначим $\Gamma_x = L[\Delta, \overline{x}],\ \Gamma_y = L[\Delta, \overline{y}],\ \Gamma = \Gamma_x \cup \Gamma_y$. По теореме A.12.1 $\Gamma_x \cup \Gamma_y \in W[\Delta, \overline{x}\,\overline{y}]$. Докажем, что $\Gamma_x \cup \Gamma_y = L[\Delta, \overline{x}\,\overline{y}]$. Предположим, от противного, что существует $\Omega \in W[\Delta, \overline{x}\,\overline{y}]$ такая, что $\Omega \underset{\overline{\alpha}_x\,\overline{\alpha}_y}{<} \Gamma$. Здесь $\overline{x} = \overline{\alpha}_x * \overline{\beta}_x$, $\overline{y} = \overline{\alpha}_y * \overline{\beta}_y$.

По теореме A.6.1 $\Omega = \Omega_x \cup \Omega_y$, где $\Omega_x \in W[\Delta, \overline{x}]$, а $\Omega_y \in W[\Delta, \overline{y}]$.

По теореме A.8.5 $\Omega \underset{\overline{\alpha}_x\,\overline{\alpha}_y}{<} \Gamma \Leftrightarrow$ либо $\Omega \underset{\overline{\alpha}_x}{<} \Gamma$, либо $\Omega \underset{\overline{\alpha}_x}{\sim} \Gamma$ и $\Omega \underset{\overline{\alpha}_y}{<} \Gamma$.

Пусть $\Omega \underset{\overline{\alpha}_x}{<} \Gamma$. Тогда по лемме A.8.4 $\Omega_x \underset{\overline{\alpha}_x}{<} \Gamma$. Однако, это невозможно, потому что $\Gamma_x = L[\Delta, \overline{x}]$.

Пусть $\Omega \underset{\overline{\alpha}_x}{\sim} \Gamma$ и $\Omega \underset{\overline{\alpha}_y}{<} \Gamma$. Тогда по лемме A.8.4 $\Omega_x \underset{\overline{\alpha}_x}{\sim} \Gamma_x$ и $\Omega_y \underset{\overline{\alpha}_y}{<} \Gamma_y$. Однако $\Omega_y \underset{\overline{\alpha}_y}{<} \Gamma_y$ невозможно потому, что $\Gamma_y = L[\Delta, \overline{y}]$.

Т.о. не может быть $\Omega \underset{\overline{\alpha}_x\,\overline{\alpha}_y}{<} \Gamma$. Поэтому $\Gamma = L[\Delta, \overline{x}\,\overline{y}]$. Теорема доказана.

Теорема A.12.3 *Пусть* $\|\Delta\| \cup \|\overline{z}\| \perp (\|\overline{x}\|, \|\overline{y}\|)$. *Тогда* $L[\Delta, \overline{z}\,\overline{x}\,\overline{y}] = L[\Delta, \overline{z}\,\overline{y}\,\overline{x}]$.

Доказательство. По теореме A.6.7 $W[\Delta, \overline{z}\ \overline{x}\ \overline{y}] = W[\Delta, \overline{z}\ \overline{y}\ \overline{x}]$. Поэтому области определения обеих частей равенства в условии теоремы совпадают.

Рассмотрим теперь случай, когда $W[\Delta, \overline{z}\,\overline{x}\,\overline{y}] = W[\Delta, \overline{z}\,\overline{y}\,\overline{x}] \neq \emptyset$.

Тогда по теореме A.10.6

$$L[\Delta, \overline{z}\,\overline{x}\,\overline{y}] = L[\Gamma'_z, \overline{x}\,\overline{y}],\ \text{где } \Gamma'_z = \min\{\Gamma \in W[\Delta, \overline{z}] \mid W[\Gamma, \overline{x}\,\overline{y}] \neq \emptyset\}$$
$$L[\Delta, \overline{z}\,\overline{y}\,\overline{x}] = L[\Gamma''_z, \overline{x}\,\overline{y}],\ \text{где } \Gamma''_z = \min\{\Gamma \in W[\Delta, \overline{z}] \mid W[\Gamma, \overline{y}\,\overline{x}] \neq \emptyset\}$$

По теореме A.6.7 $W[\Gamma, \overline{x}\ \overline{y}] = W[\Gamma, \overline{y}\ \overline{x}]$. Поэтому $\Gamma'_z = \Gamma''_z$. Обозначим их общее значение через Γ_z. Имеем

$$L[\Delta, \overline{z}\,\overline{x}\,\overline{y}] = L[\Gamma_z, \overline{x}\,\overline{y}]$$
$$L[\Delta, \overline{z}\,\overline{y}\,\overline{x}] = L[\Gamma_z, \overline{y}\,\overline{x}]$$

Однако, по теореме А.12.2

$$L[\Gamma_z, \overline{x}\,\overline{y}] = L[\Gamma_z, \overline{x}] \cup L[\Gamma_z, \overline{y}] = L[\Gamma_z, \overline{y}\,\overline{x}]$$

Отсюда следует, что $L[\Delta, \overline{z}\,\overline{x}\,\overline{y}] = L[\Delta, \overline{z}\,\overline{y}\,\overline{x}]$. Теорема доказана.

Теорема А.12.4 *Пусть* $\|\Delta\| \perp (\|\overline{x}\|, \|\overline{y}\|)$, *где* $\overline{x} = \overline{u}_1\,\overline{u}_2 \dots \overline{u}_n$, $\overline{y} = \overline{v}_1\,\overline{v}_2 \dots \overline{v}_n$. *Обозначим* $\overline{z} = \overline{u}_1\,\overline{v}_1\,\overline{u}_2\,\overline{v}_2 \dots \overline{u}_n\,\overline{v}_n$. *Тогда* $L[\Delta, \overline{z}] = L[\Delta, \overline{x}] \cup L[\Delta, \overline{y}]$.

Доказательство. Проведем доказательство индукцией по n.

При $n = 1$ по теореме А.12.2 получаем

$$L[\Delta, \overline{z}] = L[\Delta, \overline{u}_1\,\overline{v}_1] = L[\Delta, \overline{u}_1] \cup L[\Delta, \overline{v}_1] = L[\Delta, \overline{x}] \cup L[\Delta, \overline{y}]$$

Предположим теперь, что теорема верна для $n = m$. Докажем, что тогда она верна и для $n = m + 1$.

Обозначим $T = \|\Delta\| \cup \|\overline{u}_1\,\overline{u}_2 \dots \overline{u}_m\| \cup \|\overline{v}_1\,\overline{v}_2 \dots \overline{v}_m\|$. Тогда по теореме А.11.3 и теореме А.11.1

$$\|\Delta\| \perp (\|\overline{x}\|, \|\overline{y}\|) \Rightarrow T \perp (\|\overline{x}\|, \|\overline{y}\|) \Rightarrow (\|\overline{u}_{m+1}\|, \|\overline{v}_{m+1}\|)$$

Поэтому можно применить теорему А.12.3

$$L[\Delta, \overline{u}_1\,\overline{v}_1\ \dots \overline{u}_m\,\overline{v}_m\,\overline{u}_{m+1}\,\overline{v}_{m+1}] = L[\Delta, \overline{u}_1\,\overline{v}_1\ \dots \overline{u}_m\,\overline{v}_m\,\overline{v}_{m+1}\,\overline{u}_{m+1}]$$

Мы поменяли местами $\overline{u}_{m+1}$ и $\overline{v}_{m+1}$. Теперь аналогичным приемом поменяем местами $\overline{v}_m\,\overline{v}_{m+1}$ и $\overline{u}_{m+1}$. Получается:

$$L[\Delta, \overline{z}] = L[\Delta, \overline{u}_1\,\overline{v}_1 \dots [\overline{u}_m\,\overline{u}_{m+1}]\,[\overline{v}_m\,\overline{v}_{m+1}]\,]$$

Считая, что теорема верна при $n = m$, отсюда находим, что $L[\Delta, \overline{z}] = L[\Delta, \overline{x}] \cup L[\Delta, \overline{y}]$. Теорема доказана.

Теорема А.12.5 *Пусть* $\|\Delta\| \perp (X_1, X_2, \dots, X_n)$, *а*

$$x = \overline{\alpha} * \overline{\beta},\ \textit{где}\ \overline{\alpha} = (\alpha^1, \alpha^2, \dots, \alpha^m),\ \overline{\beta} = (\beta^1, \beta^2, \dots, \beta^m)$$

Пусть существуют такие целые числа $p_1, p_2, \dots, p_m$, *что* $1 \le p_i \le n$ *и* $\|\alpha^i\| \subseteq X_{p_i}$ *при* $i = 1, 2, \dots, m$. *Тогда* $L[\Delta, \overline{x}] = \bigcup_{k=1}^{n} L[\Delta, \overline{x}_k]$, *где* $\overline{x}_k$ *получается из* $\overline{x}$ *вычеркиванием таких пар* (α^i, β^i), *что* $p_i \ne k$.

Доказательство. Проводится индукцией по n. Для $n = 1$ теорема очевидным образом справедлива.

Пусть теперь теорема верна для $n = q$. Докажем, что тогда она верна и для $n = q + 1$.

Обозначим через $\overline{x}'$ вектор пар строк, который получается из $\overline{x}$ вычеркиванием таких пар (α^i, β^i), что $p_i = q + 1$.

Через $\overline{x}''$ обозначим вектор пар строк, который получается из $\overline{x}$ вычеркиванием таких пар (α^i, β^i), что $p_i \ne q + 1$.

Тогда по теореме A.11.2 и по теореме A.11.1

$$\|\Delta\|\perp(X_1, X_2, \ldots, X_q, X_{q+1}) \Rightarrow \|\Delta\|\perp(X_1 \cup X_2 \cup \ldots \cup X_q, X_{q+1}) \Rightarrow$$
$$\|\Delta\|\perp(\|\overline{x}'\|, \|\overline{x}''\|)$$

ибо $\|\overline{x}'\| \subseteq X_1 \cup X_2 \cup \ldots \cup X_q$ и $\|\overline{x}''\| \subseteq X_{q+1}$.

Таким образом, можно применить теорему A.12.4, откуда

$$L[\Delta, \overline{x}] = L[\Delta, \overline{x}'] \cup L[\Delta, \overline{x}'']$$

Заметим, что $\overline{x}'' = \overline{x}_{q+1}$, поэтому $L[\Delta, \overline{x}''] = L[\Delta, \overline{x}_{q+1}]$.

Теперь по теореме A.11.1

$$\|\Delta\|\perp(X_1, X_2, \ldots, X_q, X_{q+1}) \Rightarrow \|\Delta\|\perp(X_1, X_2, \ldots, X_q)$$

Поэтому к $L[\Delta, \overline{x}']$ можно применить гипотезу индукции и представить его в виде $L[\Delta, \overline{x}'] = \bigcup_{k=1}^{q} L[\Delta, \overline{x}_k]$.

Отсюда следует, что $L[\Delta, \overline{x}] = \bigcup_{k=1}^{q+1} L[\Delta, \overline{x}_k]$. Теорема доказана.

A.13 Отождествление левого конца строки

Определение A.13.1 *Пусть заданы две строки α и β и подстановка Δ. Тогда мы будем называть* отождествлением *левого конца β как α при Δ любую подстановку Γ, для которой $D[\Gamma]$, $\|\alpha\| \cup \|\Delta\| \subseteq \|\Gamma\|$, $\Delta \subseteq \Gamma$ и существует строка γ такая, что $[\Gamma//\alpha]\gamma = \beta$.*

Обозначим через $G[\Delta, (\alpha, \beta)]$ множество отождествлений левого конца β как α при Δ таких, что $\|\Gamma\| = \|\alpha\| \cup \|\Delta\|$, а через $G_k[\Delta, (\alpha, \beta)]$ – множество $W[\Delta, (\alpha, \beta_{1:k})]$.

Теорема A.13.1 $G[\Delta, (\alpha, \beta)] = \bigcup_{k=0}^{|\beta|} G_k[\Delta, (\alpha, \beta)]$, *причем*

$$G_i[\Delta, (\alpha, \beta)] \cap G_j[\Delta, (\alpha, \beta)] = \emptyset \text{ при } i \neq j$$

Доказательство. Согласно определению A.13.1, если $\Gamma \in G[\Delta, (\alpha, \beta)]$, то существует строка γ такая, что $[\Gamma//\alpha]\ \gamma = \beta$. Отсюда следует, что $[\Gamma//\alpha] = \beta_{1:k}$, где $k = |\Gamma//\alpha|$. Поэтому $\Gamma \in W[\Delta, (\alpha, \beta_{1:k})] = G_k[\Delta, (\alpha, \beta)]$. И обратно, если $\Gamma \in W[\Delta, (\alpha, \beta_{1:k})]$, то согласно определению A.13.1 $\Gamma \in G[\Delta, (\alpha, \beta)]$. Отсюда

$$G[\Delta, (\alpha, \beta)] = \bigcup_{k=0}^{|\beta|} G_k[\Delta, (\alpha, \beta)].$$

Пусть теперь $\Gamma_1 \in G_i[\Delta, (\alpha, \beta)]$, $\Gamma_2 \in G_j[\Delta, (\alpha, \beta)]$, где $i \neq j$. Тогда $\Gamma_1//\alpha = \beta_{1:i}$ и $\Gamma_2//\alpha = \beta_{1:j}$. Однако $\beta_{1:i} \neq \beta_{1:j}$, следовательно $\Gamma_1 \neq \Gamma_2$. Теорема доказана.

Теорема A.13.2 *Пусть $\Delta_1, \Delta_2 \in G[\Delta, (\alpha, \beta)]$. Тогда $\Delta_1 \underset{\alpha}{\sim} \Delta_2 \Rightarrow \Delta_1 = \Delta_2$.*

Доказательство. $\Delta_1 \underset{\alpha}{\sim} \Delta_2 \Rightarrow |\Delta_1//\alpha| = |\Delta_2//\alpha| = k$. Следовательно, $\Delta_1, \Delta_2 \in W[\Delta, (\alpha, \beta_{1:k})]$. Поэтому можно применить теорему А.8.3 и получить $\Delta_1 = \Delta_2$.

Следствие А.13.2.1 *Отношение* $\underset{\alpha}{\leq}$ *на множестве* $G[\Delta, (\alpha, \beta)]$ *является отношением линейного порядка.*

А.14 Удлиняющиеся строки

Определение А.14.1 *Будем говорить, что* α *удлиняется по* β *при* Δ*, если для любых подстановок* Γ_1 *и* Γ_2 *таких, что* $\Gamma_1 \in G_i[\Delta, (\alpha, \beta)]$ *и* $\Gamma_2 \in G_j[\Delta, (\alpha, \beta)]$ *из* $i < j$ *следует* $\Gamma_1 \underset{\alpha}{<} \Gamma_2$.

Теорема А.14.1 *Пусть* α *удлиняется по* β *при* Δ. *Тогда*

$$L[\Delta, (\alpha\, \gamma, \beta)\, \overline{y}] = L[\Delta, (\alpha, \beta_{1:k})(\gamma, \beta_{k+1,|\beta|})\, \overline{y}]$$

где $k = \min\{i | W[\Delta, (\alpha, \beta_{1:i})(\gamma, \beta_{i+1:|\beta|})\, \overline{y}] \neq \emptyset\}$.

Доказательство. Введем обозначения

$$\begin{aligned}
&W = W[\Delta, (\alpha\, \gamma, \beta)\, \overline{y}],\\
&\lambda = L[\Delta, (\alpha\, \gamma, \beta)\, \overline{y}],\\
&W_i = W[\Delta, (\alpha\, \beta_{1:i})(\gamma, \beta_{i+1:|\beta|})\, \overline{y}],\\
&\lambda_i = L[\Delta, (\alpha\, \beta_{1:i})(\gamma, \beta_{i+1:|\beta|})\, \overline{y}]
\end{aligned}$$

Кроме того, пусть $\overline{y} = \overline{\alpha}_y * \overline{\beta}_y$.

Тогда по теореме А.6.5 $W = \bigcup_{i=0}^{|\beta|} W_i$. Поэтому, если $W_i = \emptyset$ для $i = 0, 1 \ldots, |\beta|$, то $W = \emptyset$. При этом $k = \min\{i | W_i \neq \emptyset\}$ не определено и обе части равенства $\lambda = \lambda_k$ не определены. Если же $W \neq \emptyset$, то определены и λ и k и λ_k.

Т.о. области определения обеих частей равенства $\lambda = \lambda_k$ совпадают. Рассмотрим теперь случай, когда $W \neq \emptyset$.

Для тех λ_i, которые определены через λ'_i, обозначим такую подстановку, что $\|\lambda'_i\| = \|\Delta\| \cup \|\alpha\|$, $\Delta \subseteq \lambda'_i \subseteq \lambda_i$. По лемме А.6.2 λ'_i существует, единственна и при этом

$$\lambda'_i \in G_i[\Delta, (\alpha, \beta)]$$

Поскольку $\lambda \in W$ и $W = \bigcup_{i=0}^{|\beta|} W_i$, существует такое l, что $\lambda \in W_l$. Покажем, что при этом $\lambda = \lambda_l$.

В самом деле, по определению λ, для любой $\Omega \in W$ имеем $\lambda \underset{(\alpha\,\gamma)\overline{\alpha}_y}{\leq} \Omega$, откуда $\lambda \underset{(\alpha\,\gamma)\overline{\alpha}_y}{\leq} \Omega$ для всех $\Omega \in W_l$. Однако, $\lambda \underset{(\alpha\,\gamma)\overline{\alpha}_y}{\leq} \Omega \Leftrightarrow \lambda \underset{(\alpha\,\gamma)\overline{\alpha}_y}{\leq} \Omega$. Следовательно $\lambda = \lambda_l$.

Теперь докажем, что $\lambda_l = \lambda_k$. По определению k невозможно $k < l$. Поэтому $k \geq l$. Предположим, что $k > l$. Имеем: $\lambda'_l \in G_l[\Delta, (\alpha, \beta)]$, $\lambda'_k \in G_k[\Delta, (\alpha, \beta)]$.

Поскольку α удлиняется по β при Δ, а $l > k$, имеем $\lambda'_k \underset{\alpha}{<} \lambda'_l$. Отсюда, по лемме A.8.4 $\lambda_k \underset{\alpha}{<} \lambda_l$, а по теореме A.8.5 $\lambda_k \underset{(\alpha,\gamma)\overline{\alpha}_y}{<} \lambda_l$. Однако, по определению это невозможно.

Следовательно, $l = k$. Теорема доказана.

Теорема A.14.2 *Пусть α удлиняется по β при Δ и $\|\Delta\| \bot (\|\alpha\|, \|\gamma\| \cup \|\overline{y}\|)$. Тогда*

$$L[\Delta, (\alpha\,\gamma, \beta)\,\overline{y}] = L[\Delta, (\alpha, \beta_{1:k})] \cup L[\Delta, (\gamma, \beta_{k+1:|\beta|})\,\overline{y}],$$

где $k = \min\{i | W[\Delta, (\alpha, \beta_{1:i})] \neq \emptyset$ *и* $W[\Delta, (\gamma, \beta_{i+1:|\beta|}\ y] \neq \emptyset\}$

Доказательство. Из следствия A.6.1.1 получаем

$$W[\Delta, (\alpha, \beta_{1:i})] = \emptyset \text{ или } W[\Delta, (\gamma, \beta_{i+1:|\beta|})\,\overline{y}] = \emptyset \Rightarrow$$
$$W[\Delta, (\alpha, \beta_{1:i})(\gamma, \beta_{i+1:|\beta|})\,\overline{y}] = \emptyset$$

По теореме A.12.1 учитывая $\|\Delta\| \bot (\|\alpha\|, \|\gamma\| \cup \|\overline{y}\|)$, получаем

$$W[\Delta, (\alpha, \beta_{1:i})] \neq \emptyset \text{ и } W[\Delta, (\alpha, \gamma_{i+1:|\beta|})\,\overline{y}] \neq \emptyset \Rightarrow$$
$$W[\Delta, (\alpha, \beta_{1:i})(\gamma, \beta_{i+1:|\beta|})\,\overline{y}] \neq \emptyset$$

Поэтому

$$k = \min\{i | W[\Delta, (\alpha, \beta_{1:i})] \neq \emptyset \text{ и } W[\Delta, (\gamma, \beta_{i+1:|\beta|})\,\overline{y}] \neq \emptyset\} =$$
$$\min\{i | W[\Delta, (\alpha, \beta_{1:i})(\alpha, \gamma_{i+1:|\beta|})\,\overline{y}] \neq \emptyset\}$$

Теперь применяем теоремы A.14.1 и A.12.2

$$L[\Delta, (\alpha\,\gamma, \beta)\,\overline{y}] = L[\Delta, (\alpha, \beta_{1:k})(\gamma, \beta_{k+1:|\beta|})\,\overline{y}] =$$
$$L[\Delta, (\alpha, \beta_{1:k})] \cup L[\Delta, (\gamma, \beta_{k+1:|\beta|})\,\overline{y}]$$

Теорема доказана.

Теорема A.14.3 *Пусть α удлиняется по β при Δ и $\|\Delta\| \bot (\|\alpha\|, \|\gamma\| \cup \|\overline{y}\|)$.*

Пусть для любых i и j таких, что $W[\Delta, (\alpha, \beta_{1:i})] \neq \emptyset$ и $W[\Delta, (\alpha, \beta_{1:j})] \neq \emptyset$ как только $i < j$ и $W[\Delta, (\gamma, \beta_{j+1:|\beta|})\ \overline{y}] \neq \emptyset$ выполнено $W[\Delta, (\gamma, \beta_{i+1:|\beta|})\,\overline{y}] \neq \emptyset$.

Тогда
$L[\Delta, (\alpha\,\gamma, \beta)\,\overline{y}] = L[\Delta, (\alpha, \beta_{1:k})] \cup L[\Delta, (\gamma, \beta_{k+1:|\beta|})\,\overline{y}]$, *где*
$k = \min\{i | W[\Delta, (\alpha, \beta_{1:i})] \neq \emptyset\}$

Доказательство. Обозначим
$l = \min\{i | W[\Delta, (\alpha, \beta_{1:i})] \neq \emptyset$ и $W[\Delta, (\gamma, \beta_{i+1:|\beta|})\,\overline{y}] \neq \emptyset\}$

В силу теоремы A.14.2 достатоно доказать, что $k = l$.

В силу определения k и l , $l \geq k$. Пусть $l > k$. Тогда, по определению l, $W[\Delta, (\alpha, \beta_{1:l})] \neq \emptyset$ и $W[\Delta, (\gamma, \beta_{l+1:|\beta|})\,\overline{y}] \neq \emptyset$. А по определению k, $W[\Delta, (\alpha, \beta_{1:k})] \neq \emptyset$. Отсюда получаем, что $W[\Delta, (\gamma, \beta_{k+1:|\beta|})\ \overline{y}] \neq \emptyset$, что противоречит определению l.

Следовательно, $l = k$. Теорема доказана.

A.15 Строки, термы и выражения в языке рефал

Теперь наша задача состоит в том, чтобы результаты, полученные в предыдущих разделах, применить к языку рефал.

Для этого, введенные ранее общие понятия строки и правильной подстановки будут конкретизированы.

Будем считать, что множество символов Σ состоит из двух непересекающихся частей: $\Sigma = \Sigma_c \cup \Sigma_v,\ \Sigma_c \cap \Sigma_v = \emptyset$.

Множество Σ_c будем называть множеством констант, а множество Σ_v – множеством переменных.

Будем считать, что $\Sigma_c = \Sigma_a \cup \Sigma_b,\ \Sigma_a \cap \Sigma_b = \emptyset$, где Σ_a есть множество объектных знаков и составных символов в смысле языка рефал, а
$\Sigma_b = \{(,)\}$, т.е. состоит из двух элементов: левой и правой структурной скобки.

В свою очередь, $\Sigma_v = \Sigma_s \cup \Sigma_t \cup \Sigma_e$, где $\Sigma_s \cap \Sigma_t = \Sigma_s \cap \Sigma_e = \Sigma_t \cap \Sigma_e = \emptyset$.
Σ_s – множество s-переменных, Σ_t – множество t-переменных, Σ_e – множество e-переменных.

Объектной строкой мы будем называть любую строку $\alpha \in \Sigma^*$, для которой $\|\alpha\| \subseteq \Sigma_c$.

Теперь мы определим множество термов $T \subseteq \Sigma^*$ и множество выражений $E \subseteq \Sigma^*$ следующим образом.

Определение A.15.1 *Множество термов $T \subseteq \Sigma^*$ и множество выражений $E \subseteq \Sigma^*$ есть наименьшие подмножества Σ^*, удовлетворяющие условиям:*

1. $\Sigma_a \cup \Sigma_v \subseteq T$
2. *Для любого $\mathcal{E} \in E$ имеем $(\mathcal{E}) \in T$.*
3. *Для любых $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n \in T$ имеем $\mathcal{T}_1\ \mathcal{T}_2 \ldots \mathcal{T}_n \in E$.*

Определение A.15.1 можно переформулировать более кратко, сказав, что T и E есть языки над алфавитом Σ, являющиеся решением системы уравнений

$$\begin{cases} T = \Sigma_a \cup \Sigma_v \cup (E) \\ E = T^* \end{cases}$$

Определение A.15.2 *Строка α является:*

1. *выражением, если $\alpha \in E$*
2. *термом, если $\alpha \in T$*
3. *объектным выражением, если $\alpha \in E$ и $\|\alpha\| \subseteq \Sigma_c$*
4. *объектным термом, если $\alpha \in T$ и $\|\alpha\| \subseteq \Sigma_c$.*

Множество объектных выражений обозначим через E_o, а множество объектных термов обозначим через T_o. Очевидно, что
$E_o = E \cap \Sigma_c^*,\ T_o = T \cap \Sigma_c^*$.

Теорема А.15.1 *Любое* $\mathcal{E} \in E$ *единственным образом представляется в виде* $\mathcal{E} = \mathcal{T}_1\ \mathcal{T}_2 \ldots \mathcal{T}_n$, *где* $n \geq 0$, *а* $\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_n \in T$.

Доказательство. Очевидно, из определения А.15.1 и ассоциативности конкатенации.

Определение А.15.3 *Пусть* $\mathcal{E} \in E$ *представлено в виде* $\mathcal{E} = \mathcal{T}_1\ \mathcal{T}_2 \ldots \mathcal{T}_n$, *где* $\mathcal{T}_1\ \mathcal{T}_2 \ldots \mathcal{T}_n \in T$. *Тогда термы* $\mathcal{T}_1\ \mathcal{T}_2 \ldots \mathcal{T}_n$ *называются нуль-термами выражения* $\mathcal{E}$, *а число* n *– его нуль-длиной. Нуль-длину выражения* $\mathcal{E}$ *будем обозначать через* $< \mathcal{E} >$.

Теорема А.15.2 *Для любых* $\mathcal{E}, \mathcal{E}', \mathcal{E}'', \alpha, \beta \in \Sigma^*$ *выполнены следующие свойства:*

1. $\mathcal{E}' \in E$ *и* $\mathcal{E}'' \in E \Rightarrow \mathcal{E}'\ \mathcal{E}'' \in E$.
2. *Если* $\mathcal{E} \in E$, *то* $\alpha \in E \Leftrightarrow \alpha\ \mathcal{E} \in E$.
3. *Если* $\mathcal{E} \in E$, *то* $\alpha \in E \Leftrightarrow \mathcal{E}\ \alpha \in E$.
4. *Если* $\mathcal{E}', \mathcal{E}'' \in E$, *то* $\alpha\ \mathcal{E}'\ \beta \in E \Leftrightarrow \alpha\ \mathcal{E}''\beta \in E$.

Доказательство. Свойство 1 следует из определения А.15.1 и ассоциативности конкатенации. Свойства 2 и 3 следуют из определения А.15.1 и теоремы А.15.1. Свойство 4 доказывается индукцией по числу скобок в $\alpha\ \beta$.

А.16 Правильные подстановки в языке рефал

Определение А.16.1 *Пусть подстановка* Δ *содержит ровно одну пару* (σ, β), *то есть* $\Delta = \{(\sigma, \beta)\}$. *Будем говорить, что* Δ *правильная подстановка, если она удовлетворяет условиям:*

1. *Если* $\sigma \in \Sigma_c$, *то* $\beta = \sigma$.
2. *Если* $\sigma \in \Sigma_s$, *то* $\beta \in \Sigma_a$.
3. *Если* $\sigma \in \Sigma_t$, *то* $\beta \in T_o$.
4. *Если* $\sigma \in \Sigma_e$, *то* $\beta \in E_o$.

Определение А.16.2 *Пусть* Δ *произвольная подстановка* $\Delta \subseteq \Sigma \times \Sigma^*$. *Тогда* Δ *– правильная, если она непротеворечива и для каждой пары* $(\sigma, \Delta) \in \Delta$ *подстановка* $\{(\sigma, \beta)\}$ *– правильная. Другими словами*

$$D[\Delta] \Leftrightarrow \wedge\{D[\{(\sigma, \beta)\}]|(\sigma, \beta) \in \Delta\} \wedge D_o[\Delta]$$

Теорема А.16.1 *Предикат* $D[\Delta]$, *определенный в определении А.16.2 удовлетворяет условиям:*

1. $D[\Delta] \Rightarrow D_o[\Delta]$

2. $D[\emptyset]$

3. $D[\Delta]$ *и* $\Delta_2 \subseteq \Delta_1 \Rightarrow D[\Delta_2]$.

Доказательство. Очевидно из определения A.16.2.

Таким образом, мы видим, что предикат $D[\Delta]$ как он определен в определении A.16.2 удовлетворяет всем ограничениям, сформулированным в пункте A.4. Поэтому, понятие правильной подстановки в смысле языка рефал является частным случаем общего понятия правильной подстановки.

Теорема A.16.2 *Пусть* $\alpha \in \Sigma^*$ *и* $D[\Delta]$. *Тогда* $\Delta//\alpha \in \Sigma_c^*$. *Т.е., правильная подстановка, будучи применена к строке, дает объектную строку.*

Доказательство. Очевидно из определений A.16.1 и A.16.2.

Теорема A.16.3 *Пусть* $\mathcal{E} \in E$ *и* $D[\Delta]$. *Тогда* $\Delta//\mathcal{E} \in E_o$. *Т.е. результатом применения правильной подстановки к выражению является объектное выражение.*

Доказательство. Очевидно из теоремы A.16.2 и определений A.16.1 и A.16.2.

Теорема A.16.4 *Пусть* Δ *– подстановка, а* $\overline{\alpha} * \overline{\beta}$ *– вектор пар строк, где* $\overline{\alpha} = (\alpha^1, \alpha^2, \ldots, \alpha^n)$, *а* $\overline{\beta} = (\beta^1, \beta^2, \ldots, \beta^n)$. *Тогда, если* $W[\Delta, \overline{\alpha} * \overline{\beta}] \neq \emptyset$, *то* $\beta^i \in \Sigma_c^*$ *для* $i = 1, 2, \ldots, n$. *Кроме того,* $\beta^i \in E_o$ *для всех* i, *для которых* $\alpha^i \in E$.

Доказательство. Следует из теорем A.16.2 и A.16.3.

Из теоремы A.16.4 следует, что вектор строк $\overline{\beta}$ может быть отождествлен как вектор строк $\overline{\alpha}$, только, если строки, составляющие $\overline{\beta}$ не содержат переменных. Причем, если $\overline{\alpha}$ составлен из выражений, то $\overline{\beta}$ должен быть составлен из объектных выражений.

Поэтому, в дальнейшем, при изучении отождествления $\overline{\beta}$ как $\overline{\alpha}$ мы всегда будем предполагать, что $\|\overline{\beta}\| \in \Sigma_c$.

A.17 Жесткие термы и выражения

Определение A.17.1 *Терм* $\mathcal{T} \in T$ *называется жестким при* Δ, *если* $\mathcal{T} \notin \Sigma_e$, *либо* $\mathcal{T} \in \Sigma_e \cap \|\Delta\|$. *Выражение* $\mathcal{E} \in E$ *называется жестким при* Δ, *если* $\mathcal{E} = \mathcal{T}_1\, \mathcal{T}_2 \ldots \mathcal{T}_n$, *где каждый* $\mathcal{T}_i$ *есть терм, жесткий при* Δ.

Теорема A.17.1 *Пусть* $\mathcal{E}$ *– выражение, жесткое при* Δ. *Тогда существует такое число* k, *что для любой подстановки* Γ, *такой, что* $D[\Gamma]$, $\Delta \subseteq \Gamma$ *и* $\|\mathcal{E}\| \subseteq \|\Gamma\|$ *имеет место* $< \Gamma//\mathcal{E} >= k$.

Доказательство. Если Δ – неправильная, то в качестве k можно выбрать любое число. Пусть теперь верно $D[\Delta]$. Представим $\mathcal{E}$ в виде последовательности нуль-термов $\mathcal{E} = \mathcal{T}_1\ \mathcal{T}_2 \dots \mathcal{T}_n$. Тогда,
$< \Gamma//\mathcal{E} >=< \Gamma//\mathcal{T}_1 > + < \Gamma//\mathcal{T}_2 > + \ldots + < \Gamma//\mathcal{T}_n >$. Т.о. достаточно доказать, что каждое слагаемое $< \Gamma//\mathcal{T}_i >$ не зависит от Γ. Для этого заметим, что каждый терм $\mathcal{T}_i$-жесткий. Поэтому, если $\mathcal{T}_i \notin \Sigma_e$, то
$< \Gamma//\mathcal{T}_i >= 1$ и от Γ не зависит. Если же $\mathcal{T}_i \in \Sigma_e \cap \|\Delta\|$, то
$< \Gamma//\mathcal{T}_i >=< \Delta//\mathcal{T}_i >$ и тоже не зависит от Γ. Теорема доказана.

Обозначим через $< \mathcal{E} >_\Delta$ число k, фигурирующее в условии теоремы А.17.1. При этом, если $\mathcal{E}$ – выражение, жесткое при Δ, и Δ – правильная подстановка, $< \mathcal{E} >_\Delta$ определено однозначно. Если же Δ – неправильная, то, для определенности, будем считать, что $< \mathcal{E} >_\Delta = 0$.

Теорема А.17.2 *Пусть $\mathcal{E}$, $\gamma \in E$, $\beta \in E_o$ и $\mathcal{E}$ – жесткое при Δ. Тогда, если $< \mathcal{E} >_\Delta\ >\ < \beta >$, то*

$$W[\Delta, \overline{x}(\mathcal{E}\ \gamma, \beta)\ \overline{y}] = W[\Delta, \overline{x}(\gamma\ \mathcal{E}, \beta)\ \overline{y}] = \emptyset$$

Доказательство. Пусть $\Gamma \in W[\Delta, \overline{x}(\mathcal{E}\ \gamma, \beta)\ \overline{y}]$. Тогда $\Gamma//\mathcal{E}\ \gamma = \beta$. Отсюда $< \Gamma//\mathcal{E}\ \gamma >=< \beta >$. Однако $< \Gamma//\mathcal{E}\ \gamma >=< \Gamma//\mathcal{E} > + < \Gamma//\gamma >$. Поэтому $< \Gamma//\mathcal{E} >\ \leq\ < \beta >$. Однако $< \Gamma//\mathcal{E} >=< \mathcal{E} >_\Delta$. Следовательно, $< \mathcal{E} >_\Delta\ \leq\ < \beta >$. Но по условию теоремы $< \mathcal{E} >_\Delta\ >\ < \beta >$. Следовательно, $W[\Delta, \overline{x}(\mathcal{E}\ \gamma, \beta)\ \overline{y}] = \emptyset$. Аналогично, $W[\Delta, \overline{x}(\gamma\ \mathcal{E}, \beta)\ \overline{y}] = \emptyset$.

Теорема А.17.3 *Пусть $\mathcal{E}, \gamma \in E$, $\beta', \beta'' \in E_o$ и $\mathcal{E}$ – жесткое при Δ. Тогда, если $< \mathcal{E} >_\Delta =< \beta' >$, то $\mathcal{E}$ и γ однозначно сопоставляются с β' и β'' при Δ, а γ и $\mathcal{E}$ однозначно сопоставляются с β'' и β'.*

Доказательство. Нам нужно доказать, что

$$W[\Delta, (\mathcal{E}\gamma, \beta'\beta'')] = W[\Delta, (\mathcal{E}, \beta')(\gamma, \beta'')] = W[\Delta, (\gamma\mathcal{E}, \beta''\beta')]$$

Обозначим $\beta = \beta'\beta''$, $W_k = W[\Delta, (\mathcal{E}, \beta_{1:k})(\gamma, \beta_{k+1:|\beta|})]$. По теореме А.6.5 $W[\Delta, (\mathcal{E}\gamma, \beta)] = \bigcup_{k=0}^{|\beta|} W_k$. Пусть $W_k \neq \emptyset$. Тогда по теореме А.16.4 $\beta_{1:k} \in E_o$ и $\beta_{k+1:|\beta|} \in E_o$. Кроме того, для любой $\Gamma \in W_k$ имеем $\Gamma//\mathcal{E} = \beta_{1:k}$. Отсюда $< \Gamma//\mathcal{E} >=< \beta_{1:k} >$. Однако $< \Gamma//\mathcal{E} >=< \mathcal{E} >_\Delta$, ибо $\mathcal{E}$ – жесткое при Δ. Следовательно, $< \mathcal{E} >_\Delta =< \beta_{1:k} >=< \beta' >$. Отсюда $\beta_{1:k} = \beta'$. Поэтому $W_k \neq \emptyset$ возможно только для единственного k такого, что $\beta_{1:k} = \beta'$. Таким образом,

$$W[\Delta, (\mathcal{E}\gamma, \beta'\beta'')] = W[\Delta, (\mathcal{E}, \beta')(\gamma, \beta'')]$$

Аналогично

$$W[\Delta, (\gamma\mathcal{E}, \beta''\beta')] = W[\Delta, (\gamma, \beta'')(\mathcal{E}, \beta')] = W[\Delta(\mathcal{E}, \beta')(\gamma, \beta'')]$$

Теорема доказана.

Теорема A.17.4 *Пусть $\mathcal{E}, \gamma \in E$, $\beta \in E_o$, и $\mathcal{E}$ – жесткое при Δ. Тогда, если $< \mathcal{E} >_\Delta \; > \; < \beta >$, то $L[\Delta, \overline{x}(\mathcal{E}\ \gamma, \beta)\ \overline{y}]$ и $L[\Delta, \overline{x}(\gamma\ \mathcal{E}, \beta)\ \overline{y}]$ не определены. Если же $< \mathcal{E} >_\Delta \; \leq \; < \beta >$, то*

$$L[\Delta, \overline{x}(\mathcal{E}\ \gamma, \beta)\ \overline{y}] = L[\Delta, \overline{x}(\mathcal{E}, \beta_{1:k})(\gamma, \beta_{k+1:|\beta|})\ \overline{y}]$$
$$L[\Delta, \overline{x}(\gamma\ \mathcal{E}, \beta)\ \overline{y}] = L[\Delta, \overline{x}(\gamma, \beta_{1:l})(\mathcal{E}, \beta_{l+1:|\beta|})\ \overline{y}]$$

где k и l однозначно определяются из условий:

$$\beta_{1:k} \in E_o,\ < \beta_{1:k} > = < \mathcal{E} >_\Delta,\ \beta_{l+1:|\beta|} \in E_o,\ < \beta_{l+1:|\beta|} > = < \mathcal{E} >_\Delta$$

Доказательство. Если $< \mathcal{E} >_\Delta \; > \; < \beta >$, то по теореме A.17.2
$W[\Delta, \overline{x}(\mathcal{E}\ \gamma, \beta)\ \overline{y}] = W[\Delta, \overline{x}(\gamma\ \mathcal{E}, \beta)\ \overline{y}] = \emptyset$
Поэтому, $L[\Delta, \overline{x}(\mathcal{E}\ \gamma, \beta)\ \overline{y}]$ и $L[\Delta, \overline{x}(\gamma\ \mathcal{E}, \beta)\ \overline{y}]$ не определены.

Если $< \mathcal{E} >_\Delta \; \leq \; < \beta >$, то существуют единственные k и l, указанные в условии теоремы. Поскольку $< \beta_{1:k} > = < \mathcal{E} >_\Delta$ и $< \beta_{l+1:|\beta|} > = < \mathcal{E} >_\Delta$, по теореме A.17.3 $\mathcal{E}$ и γ однозначно сопоставляются с $\beta_{1:k}$ и $\beta_{k+1:|\beta|}$, а γ и $\mathcal{E}$ однозначно сопоставляются с $\beta_{1:l}$ и $\beta_{l+1:|\beta|}$. Поэтому, применяя теорему A.10.5, получаем соотношения, указанные в утверждении теоремы.

Теперь обозначим через B подстановку $B = \{(\sigma, \sigma) | \sigma \in \Sigma_b\}$, которая состоит из двух пар: $((, ()$ и $(),))$. Ясно, что $\|B\| = \Sigma_b, D[B]$ и $[B//(] = (, [B//)] =)$.

Лемма A.17.1 *Пусть $\alpha \in E$, $\beta \in E_o$. Тогда*

$$L[\Delta, \overline{x}((\alpha), (\beta))\ \overline{y}] = L[\Delta \cup B, \overline{x}(\alpha, \beta)\ \overline{y}].$$

Доказательство. Пусть $\Gamma \in W[\Delta, ((\alpha), (\beta))]$. Тогда

$$\Gamma//(\alpha) = [\ \Gamma//(\ ]\ [\ \Gamma//\alpha\ ]\ [\ \Gamma//)\ ] = (\ [\Gamma//\alpha]\ ) = (\beta).$$

Таким образом, $\Gamma//\alpha = \beta$, $[\Gamma/(] = (,\ [\ \Gamma//)\ ] =)$. Поэтому (и α) однозначно сопоставляются с (и β), а α и) однозначно сопоставляются с β и). Теперь, дважды применяя теоремы A.10.5 и A.10.3, получаем:

$$L[\Delta, \overline{x}((\alpha), (\beta))\ \overline{y}] = L[\Delta \cup B, \overline{x}(\alpha, \beta)\ \overline{y}].$$

Лемма доказана.

Теорема A.17.5 *Пусть $\alpha', \alpha'' \in E$ и $\beta', \beta'' \in E_o$. Тогда*

$$L[\Delta, \overline{x}((\alpha')\alpha'', (\beta')\beta'')\ \overline{y}] = L[\Delta \cup B, \overline{x}(\alpha', \beta')(\alpha'', \beta'')\ \overline{y}],$$
$$L[\Delta, \overline{x}(\alpha''(\alpha'), \beta''(\beta'))\ \overline{y}] = L[\Delta \cup B, \overline{x}(\alpha'', \beta'')(\alpha', \beta')\ \overline{y}].$$

Доказательство. Терм (α') является жестким. $< (\alpha') >_\Delta = < (\beta') > = 1$. Поэтому, по теореме A.17.4 получаем:

$$L[\Delta, \overline{x}((\alpha')\alpha'', (\beta')\beta'')\ \overline{y}] = L[\Delta, \overline{x}((\alpha'), (\beta'))(\alpha'', \beta'')\ \overline{y}],$$
$$L[\Delta, \overline{x}(\alpha''(\alpha'), \beta''(\beta'))\ \overline{y}] = L[\Delta, \overline{x}(\alpha'', \beta'')((\alpha'), (\beta'))\ \overline{y}].$$

Теперь применяем лемму A.17.1 и получаем утверждение теоремы.

Теорема A.17.6 *Пусть* $\beta \in E_o$, $\beta = \mathcal{T}_1\ \mathcal{T}_2 \ldots \mathcal{T}_n$, *где* $\mathcal{T}_i \in T_o$ *для* $i = 1, 2, \ldots, n$, *а* $l \in \Sigma_l$. *Обозначим через* $\mathcal{E}'_i$ *и* $\mathcal{E}''_i$ *выражения* $\mathcal{E}'_i = \mathcal{T}_1\ \mathcal{T}_2 \ldots \mathcal{T}_i$, $\mathcal{E}''_i = \mathcal{T}_{i+1}\ \mathcal{T}_{i+2} \ldots \mathcal{T}_n$. *Тогда*

$$L[\Delta, (e\ \alpha, \beta)\ \overline{y}] = L[\Delta \cup \{(e, \mathcal{E}'_k)\}, (\alpha, \mathcal{E}''_k)\ \overline{y}], \text{ где}$$
$$k = \min\{i | W[\Delta \cup \{(e, \mathcal{E}'_i)\}, (\alpha, \mathcal{E}''_i)\ \overline{y}] \neq \emptyset\}.$$

Доказательство. Обозначим $W_j = W[\Delta \cup \{(e, \beta_{1:j})\}, (\alpha, \beta_{j+1:|\beta|})\ \overline{y}]$, $l = \min\{j | W_j \neq \emptyset\}$. Тогда по теореме A.10.4

$$L[\Delta, (e\ \alpha, \beta)\ \overline{y}] = L[\Delta \cup \{(e, \beta_{1:l})\}, (\alpha, \beta_{l+1:|\beta|})\ \overline{y}].$$

Однако, поскольку $e \in \Sigma_l$, из $W_j \neq \emptyset$ следует, что $\beta_{1:j} \in E_o$. А всякому j, такому, что $\beta_{1:j} \in E_o$, соответствует единственное i, такое, что $\mathcal{E}'_i = \beta_{1:j}$. При этом, если $\beta_{1:j_1} \in E_o$, $\beta_{1:j_2} \in E_o$, $\mathcal{E}'_{i_1} = \beta_{1:j_1}$, $\mathcal{E}'_{i_2} = \beta_{1:j_2}$, то из $j_1 < j_2$ следует, что $i_1 < i_2$. Отсюда вытекает утверждение теоремы.

A.18 Независимые множества символов в языке рефал

В пункте 11 изучалось отношение независимости для подмножеств Σ, которое определялось через предикат $D[\Delta]$. Теперь мы изучим те свойства отношения независимости, которые справедливы в том случае, когда предикат $D[\Delta]$ задан в соответствии с определениями A.16.1 и A.16.2.

Лемма A.18.1 *Пусть* $\Delta_1, \Delta_2, \ldots \Delta_n$ *такие подстановки, что* $D[\Delta_i]$ *истинно для* $i = 1, 2, \ldots, n$. *Обозначим* $\Delta = \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$. *Тогда, если подстановка* Δ *непротиворечива, то* $D[\Delta]$ *истинно, а если* Δ *– противоречива, то* $D[\Delta]$ *ложно.*

Доказательство. Пусть $(\sigma, \beta) \in \Delta$. Тогда существует такое i, что $(\sigma, \beta) \in \Delta_i$. Поскольку $D[\Delta_i]$ истинно, из монотонности $D[\Omega]$ получаем, что $D[\{(\sigma, \beta)\}]$ истинно. Т.о., для любой пары $(\sigma, \beta) \in \Delta$ истинно $D[\{(\sigma, \beta)\}]$. Теперь воспользуемся определением A.16.2 и получим, что $D[\Delta]$ истинно, тогда и только тогда, когда Δ – непротиворечива.

Лемма A.18.2 *Пусть* $\Delta_1, \Delta_2, \ldots, \Delta_n$ *такие подстановки, что* $\|\Delta_i\| \cap \|\Delta_j\| = \emptyset$ *при* $i \neq j$. *Тогда, если* $D[\Delta_i]$ *истинно для* $i = 1, 2, \ldots, n$, *то* $D[\Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n]$ *истинно.*

Доказательство. Достаточно заметить, что $D[\Delta_i] \Rightarrow D_O[\Delta_i]$. А поскольку $\|\Delta_i\| \cap \|\Delta_j\| = \emptyset$ при $j \neq i$, то истинно и $D[\Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n]$. Теперь остается применить лемму A.18.1

Лемма A.18.3 *Пусть* $\Delta_1, \Delta_2, \ldots, \Delta_n$ *такие подстановки, что* $\|\Delta_i\| \subseteq \Sigma_c$ *для* $i = 1, 2, \ldots, n$. *Тогда, если* $D[\Delta_i]$ *истинно при* $i = 1, 2, \ldots, n$, *то* $D[\Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n]$ *истинно.*

Доказательство. Обозначим $\Delta = \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$. Пусть $(\sigma, \beta) \in \Delta$. Тогда существует такое i, что $(\sigma, \beta) \in \Delta_i$. Теперь из $\|\Delta_i\| \subseteq \Sigma_c$ и $D[\Delta_i]$ получаем $\sigma \in \Sigma_c$ и $\beta = \sigma$. Таким образом, любая пара $(\sigma, \beta) \in \Delta$ имеет вид: (σ, σ). Следовательно, $D_o[\Delta]$ истинно. Применяем лемму A.18.1 и получаем, что $D[\Delta]$ истинно.

Теорема A.18.1 *Пусть $Y, X_1, X_2, \ldots, X_n$ некоторые подмножества Σ, такие, что $X_i \cap X_j \subseteq Y \cup \Sigma_c$ для всех $i \neq j$. Тогда $Y \bot (X_1, X_2, \ldots, X_n)$.*

Доказательство. Пусть $\Omega, \Delta_1, \Delta_2, \ldots, \Delta_n$ такие подстановки, что $\|\Omega\| = Y$, $\|\Delta_i\| \subseteq \|X_i\|$ и $D[\Omega \cup \Delta_i]$ для всех $i = 1, 2, \ldots, n$. Нам нужно доказать, что $D[\Omega \cup \Delta_1 \cup \Delta_2 \cup \ldots \Delta_n]$ истинно.

Представим каждую Δ_i в виде $\Delta_i = \Omega_i \cup \Phi_i \cup \Gamma_i$, где $\|\Omega_i\| \subseteq Y$, $\|\Phi_i\| \subseteq \Sigma_c \backslash Y$, $\|\Gamma_i\| \subseteq \Sigma_v \backslash Y$. Из монотонности $D[\Delta]$ следует, что истинны $D[\Omega]$, $D[\Omega \cup \Omega_i]$, $D[\Phi_i]$, $D[\Gamma_i]$ для $i = 1, 2, \ldots, n$. Заметим, что $D[\Omega \cup \Omega_i] \Rightarrow D_o[\Omega \cup \Omega_i]$ и $\|\Omega_i\| \subseteq Y = \|\Omega\|$. Поэтому $\Omega_i \subseteq \Omega$. Следовательно $\Omega_1 \cup \Omega_2 \cup \ldots \cup \Omega_n \subseteq \Omega$.

Обозначим $\Phi = \Phi_1 \cup \Phi_2 \cup \ldots \cup \Phi_n$. Используя лемму A.18.3, получаем, что $D[\Phi]$ истинно.

Обозначим $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \ldots \cup \Gamma_n$. Из условий $X_i \cap X_j \subseteq Y \cup \Sigma_c$ и $\|\Gamma_i\| \subseteq \Sigma_v \backslash Y$ получаем, что $\|\Gamma_i\| \cap \|\Gamma_j\| = \emptyset$ при $i \neq j$. Применяем лемму A.18.2 и получаем, что $D[\Gamma]$ истинно.

Теперь заметим, что множества $Y, \Sigma_c \backslash Y$ и $\Sigma_v \backslash Y$ попарно не пересекаются. Значит и множества $\|\Omega\|, \|\Phi\|$ и $\|\Gamma\|$ попарно не пересекаются. Кроме того, мы показали, что $D[\Omega]$, $D[\Phi]$ и $D[\Gamma]$ истинны. Следовательно, по лемме A.18.2 $D[\Omega \cup \Phi \cup \Gamma]$ истинно. Однако $\Omega \cup \Phi \cup \Gamma = \Omega \cup \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$, что и завершает доказательство теоремы.

Теорема A.18.2 *Пусть $Y, X_1, X_2, \ldots, X_n$ некоторые подмножества Σ, такие, что $Y \bot (X_1, X_2, \ldots, X_n)$. Тогда, если Σ_a содержит по крайней мере два различных элемента, то $X_i \cap X_j \subseteq Y \cup \Sigma_c$ для всех $j \neq j$.*

Доказательство. Предположим от противного, что имеются такие i и j, что не выполнено: $X_i \cap X_j \subseteq Y \cup \Sigma_c$. Это значит, что существует такая переменная $v \in \Sigma_v \cap X_i \cap X_j$, что $v \notin Y$.

Покажем, что $Y \bot (\{v\}, \{v\})$ неверно. Выберем произвольную подстановку Ω, такую, что $\|\Omega\| = Y$ и $D[\Omega]$. Пусть теперь σ_1 и σ_2 такие символы, что $\sigma_1, \sigma_2 \in \Sigma_a$ и $\sigma_1 \neq \sigma_2$. Обозначим $\Delta_1 = \{(v, \sigma_1)\}$, $\Delta_2 = \{(v, \sigma_2)\}$.

Из определения A.16.1 следует, что $D[\Delta_1]$ и $D[\Delta_2]$, а из $v \notin Y$ и леммы A.18.2 следует, что $D[\Omega \cup \Delta_1]$ и $D[\Omega \cup \Delta_2]$. Теперь рассмотрим подстановку $\Omega \cup \Delta_1 \cup \Delta_2$. Она противоречива, ибо содержит обе пары (v, σ_1) и (v, σ_2). Следовательно, $D[\Omega \cup \Delta_1 \cup \Delta_2]$ ложно. Поэтому $Y \bot (\{v\}, \{v\})$ не выполнено.

В то же время, по теореме A.11.1
$Y \bot (X_1, X_2, \ldots, X_n) \Rightarrow Y \bot (X_i, X_j) \Rightarrow Y \bot (\{v\}, \{v\})$.

Полученное противоречие доказывает теорему.

Определение A.18.1 *Будем говорить, что две строки* $\alpha, \beta \in \Sigma^*$ непосредственно *зависимы при* Δ*, если* $[\|\alpha\| \cap \|\beta\| \cap \Sigma_v] \backslash \|\Delta\| \neq \emptyset$*, т.е. если существует переменная* $v \in \Sigma_v$*, такая, то* $v \in \|\alpha\| \cap \|\beta\|$ *и* $v \notin \|\Delta\|$*. То, что* α *и* β *непосредственно зависимы при* Δ *будем записывать как* $\alpha \underset{\Delta}{\leftrightharpoons} \beta$*.*

Очевидно, что $\alpha \underset{\Delta}{\leftrightharpoons} \beta \Leftrightarrow \beta \underset{\Delta}{\leftrightharpoons} \alpha$.

Определение A.18.2 *Пусть* $\overline{\alpha}$ *– вектор строк* $\overline{\alpha} = (\alpha^1, \alpha^2, \ldots, \alpha^m)$*. Будем говорить, что две строки* α^i *и* α^j *зависимы при* Δ *на* $\overline{\alpha}$*, если существует такая последовательность строк* $\alpha^{k_1}, \alpha^{k_2}, \ldots \alpha^{k_l}$*, что* $k_1 = i$*,* $k_l = j$ *и* $\alpha^{k_{p-1}} \underset{\Delta}{\leftrightharpoons} \alpha^{k_p}$ *при* $2 \leq p \leq l$*. То, что* α^i *и* α^j *зависимы при* Δ *на* $\overline{\alpha}$*, мы будем записывать как* $\alpha^i \underset{\Delta,\overline{\alpha}}{\leftrightarrow} \alpha^j$*.*

Теорема A.18.3 *Пусть* $\overline{\alpha} = (\alpha^1, \alpha^2, \ldots, \alpha^m)$*. Тогда отношение* $\underset{\Delta,\overline{\alpha}}{\leftrightarrow}$ *является отношением эквивалентности на множестве строк* $\{\alpha^1, \alpha^2, \ldots, \alpha^m\}$*.*

Доказательство. Покажем, что отношение $\underset{\Delta,\overline{\alpha}}{\leftrightarrow}$ рефлексивно, симметрично и транзитивно. Во-первых, $\alpha^i \underset{\Delta,\overline{\alpha}}{\leftrightarrow} \alpha^i$ согласно определению A.18.2, ибо последовательность из одного элемента α^i обладает требуемыми свойствами. Во-вторых, симметричность отношения $\underset{\Delta,\overline{\alpha}}{\leftrightarrow}$ следует из симметричности отношения $\underset{\Delta}{\leftrightharpoons}$ и определения A.18.2. Транзитивность очевидна из определения A.18.2.

Следствие A.18.3.1 *Пусть* $\overline{\alpha} = (\alpha^1, \alpha^2, \ldots, \alpha^m)$*. Тогда множество строк* $\{\alpha^1, \alpha^2, \ldots, \alpha^m\}$ *разбивается на* $n(\leq m)$ *классов эквивалентности по отношению* $\underset{\Delta,\overline{\alpha}}{\leftrightarrow}$*.*

Теорема A.18.4 *Пусть* $\overline{x} = \overline{\alpha} * \overline{\beta}$*, где* $\overline{\alpha} = (\alpha^1, \alpha^2, \ldots, \alpha^m), \overline{\beta} = (\beta^1, \beta^2, \ldots, \beta^m)$*. Разобьем множество строк* $\{\alpha^1, \alpha^2, \ldots, \alpha^m\}$ *на* n *классов эквивалентности по отношению* $\underset{\Delta,\overline{\alpha}}{\leftrightarrow}$*. Тогда*

$$L[\Delta, \overline{x}] = \bigcup_{k=1}^{n} L[\Delta, \overline{x}_k]$$

где $\overline{x}_k$ *получается из* $\overline{x}$ *вычеркиванием таких пар* (α^i, β^i)*, в которых* α^i *не принадлежит* k*-му классу эквивалентности.*

Доказательство. Пусть $\overline{x}_k = \overline{\alpha}_k * \overline{\beta}_k$. Теперь заметим, что если α^i и α^j принадлежат к различным классам эквивалентности, то

$$[\ \|\alpha^i\| \cap \|\alpha^j\| \cap \Sigma_v] \backslash \|\Delta\| = \emptyset, \text{ т.е. } \|\alpha^i\| \cap \|\alpha^j\| \subseteq \|\Delta\| \cup \Sigma_c.$$

Следовательно $\|\overline{\alpha}_k\| \cap \|\overline{\alpha}_l\| \subseteq \|\Delta\| \cup \Sigma_c$ при $k \neq l$. Отсюда, по теореме A.18.1 получаем, что $\|\Delta\| \perp (\|\overline{\alpha}_1\|, \|\overline{\alpha}_2\|, \ldots, \|\overline{\alpha}_n\|)$. Поэтому, применяя теорему A.12.5, получаем доказываемое утверждение.

A.19 Удлиняющиеся выражения

Теорема A.19.1 *Рассмотрим подстановку* Δ, *e-переменную* $e_x \in \Sigma_e$ *и выражение* $\mathcal{E}_a \in E$. *Пусть* $\mathcal{E}_a = \mathcal{T}_1\ \mathcal{T}_2 \ldots \mathcal{T}_n$, *где для каждого* $i = 1, 2, \ldots, n$ *либо* $\mathcal{T}_i = e_x$, *либо* $\mathcal{T}_i$ *– терм, жесткий при* Δ. *Тогда, для любой строки* $\beta \in \Sigma_c^*$ *выражение* $e_x\ \mathcal{E}_a$ *удлиняется по* β *при* Δ.

Доказательство. Согласно определению A.14.1 требуется доказать, что для любых подстановок $\Gamma_1, \Gamma_2 \in G[\Delta, (e_x\ \mathcal{E}_a, \beta)]$ из $|\Gamma_1 // e_x\ \mathcal{E}_a| < |\Gamma_2 // e_x\ \mathcal{E}_a|$ следует, что $\Gamma_1 \underset{e_x\ \mathcal{E}_a}{<} \Gamma_2$.

Пусть $\Gamma \in G[\Delta, (e_x\ \mathcal{E}_a, \beta)]$. Тогда по теореме A.16.3 $\Gamma // e_x\ \mathcal{E}_a \in E_o$. Вычислим нуль-длину $\Gamma // e_x\ \mathcal{E}_a$. Очевидно, что

$$< \Gamma // e_x\ \mathcal{E}_a > = < \Gamma // e_x > + \Sigma_{k+1}^n < \Gamma // \mathcal{T}_k > .$$

Пусть среди термов $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n$ ровно l термов совпадают с e_x. Тогда $< \Gamma // e_x\ \mathcal{E}_a > = (l+1) \times < \Gamma // e_x > + (n-l)$. Мы видим, что нуль-длина $\Gamma // e_x\ \mathcal{E}_a$ линейно растет с увеличением нуль-длины $\Gamma // e_x$.

Таким образом, если $|\Gamma_1 // e_x\ \mathcal{E}_a| < |\Gamma_2 // e_x\ \mathcal{E}_a|$, то $< \Gamma_1 // e_x\ \mathcal{E}_a >$ меньше, чем $< \Gamma_2 // e_x\ \mathcal{E}_a >$, поскольку строка $\Gamma_1 // e_x\ \mathcal{E}_a$ является левым концом строки $\Gamma_2 // e_x\ \mathcal{E}_a$. Отсюда $\Gamma_1 \underset{\alpha}{<} \Gamma_2$.

Теорема A.19.2 *Рассмотрим правильную подстановку* Δ *и выражение* $\mathcal{E}$. *Пусть* $\mathcal{E} = t_1\ t_2 \ldots t_m\ e$, *где* $t_1, t_2, \ldots, t_m \in \Sigma_t$, *а* $e \in \Sigma_e$, *причем переменные* $t_1\ t_2 \ldots t_m\ e$ *– попарно различны и не входят в* $\|\Delta\|$. *Тогда для любого* $\mathcal{E}_o \in E_o$, *если* $< \mathcal{E}_o > < m$, *то* $W[\Delta, (\mathcal{E}, \mathcal{E}_o)] = \emptyset$, *а если* $< \mathcal{E}_o > \geq m$, *то* $W[\Delta, (\mathcal{E}, \mathcal{E}_o)] = \{\Gamma\}$.

Доказательство. Выражение $t_1\ t_2 \ldots t_m$ – жесткое при Δ. $< t_1\ t_2 \ldots t_m >_\Delta = m$. Если $< \mathcal{E}_o > < m$, то по теореме A.17.2 $W[\Delta, (\mathcal{E}, \mathcal{E}_o)] = \emptyset$. Пусть теперь $< \mathcal{E}_o > \geq m$. Тогда $\mathcal{E}_o = \mathcal{T}_1\ \mathcal{T}_2 \ldots \mathcal{T}_m\ \mathcal{E}_o'$, где $\mathcal{T}_1\ \mathcal{T}_2 \ldots \mathcal{T}_m \in T_o$, а $\mathcal{E}_o' \in E_o$. Обозначим $\Gamma_i = \{(t_i, \mathcal{T}_i)\}$, $\Gamma' = \{(e, \mathcal{E}_o')\}$, $\Gamma = \Delta \cup \Gamma_1 \cup \ldots \cup \Gamma_m \cup \Gamma'$. Множества $\|\Delta\|, \|\Gamma_1\|, \ldots \|\Gamma_m\|, \|\Gamma'\|$ попарно не пересекаются. Кроме того, истинны $D[\Delta], D[\Gamma_1], \ldots, D[\Gamma_m], D[\Gamma']$. Поэтому, по лемме A.18.2 $D[\Gamma]$ истинно.

Теперь воспользуемся теоремами A.17.4 и A.6.4 и получим

$$W[\Delta, (\mathcal{E}, \mathcal{E}_o)] = W[\Delta, (t_1, \mathcal{T}_1)(t_2, \mathcal{T}_2) \ldots (t_m, \mathcal{T}_m)(e_o, \mathcal{E}_o')] = W[\Gamma, \emptyset].$$

Поскольку справедливо $D[\Gamma]$, то по теореме A.6.2 $W[\Delta, (\mathcal{E}, \mathcal{E}_o)] = W[\Gamma, \emptyset] = \{\Gamma\}$. Теорема доказана

Теорема A.19.3 *Пусть* $t_1, t_2, \ldots, t_m \in \Sigma_t$, $e \in \Sigma_e$, $\gamma \in \Sigma^*$, $\beta \in \Sigma_c^*$, *а* $\overline{y}$ *– вектор пар строк, причем все переменные* $t_1, t_2, \ldots, t_m, e$ *попарно различны и не входят в* $\|\Delta\| \cup \|\gamma\| \cup \|\overline{y}\|$. *Тогда, для любого* $\mathcal{E}_o \in E_o$ *из* $W[\Delta, (t_1 t_2 \ldots t_m\ e\gamma, \beta)\overline{y}] \neq \emptyset$ *следует, что* $W[\Delta, (t_1 t_2 \ldots t_m\ e\gamma, \mathcal{E}_o\beta)\ \overline{y}] \neq \emptyset$.

Доказательство. Пусть $\Gamma \in W[\Delta,(t_1 t_2 \ldots t_m e\gamma,\beta)\ \overline{y}]$. Тогда существуют такие строки β' и β'', что $\Gamma//(t_1 t_2 \ldots t_m e = \beta'$, $\Gamma//\gamma = \beta''$ и $\beta'\ \beta'' = \beta$. При этом, поскольку $t_1\ t_2 \ldots t_m e \in E$, по теореме A.16.3 $\beta' \in E_o$. Следовательно, $\Gamma \in W[\Delta,(t_1 t_2 \ldots t_m\ e,\beta')(\gamma,\beta'')\ \overline{y}]$. По теореме A.6.1 Γ единственным образом представляется в виде $\Gamma = \Gamma' \cup \Gamma''$, где $\Gamma' \in W[\Delta,(t_1 t_2 \ldots t_m e,\beta')]$, $\Gamma'' \in W[\Delta,(\gamma,\beta'')\ \overline{y}]$.

Поскольку $\beta' \in E_o$ и $\mathcal{E}_o \in E_o$, по теореме A.15.2 $\mathcal{E}_o\beta' \in E_o$. Поэтому, по теореме A.19.2 $W[\Delta,(t_1 t_2 \ldots t_m e,\mathcal{E}_o\beta')] = \{\Gamma'''\}$, где Γ''' – некоторая правильная подстановка. Обозначим $\Omega = \Gamma'''\backslash\Delta$. Тогда $\|\Omega\| \cap \|\Gamma''\| = \emptyset$. Следовательно, по лемме A.18.2 $D[\Omega \cup \Gamma'']$ истинно. Таким образом

$$\Omega \cup \Gamma'' \in W[\Delta,(t_1 t_2 \ldots t_m e,\mathcal{E}_o\beta')(\gamma,\beta'')\ \overline{y}] \subseteq W[\Delta,(t_1 t_2 \ldots t_m e\gamma,\mathcal{E}_o\beta)\ \overline{y}].$$

Значит, $W[\Delta,(t_1 t_2 \ldots t_m e\gamma,\mathcal{E}_o\beta)\ \overline{y}] \neq \emptyset$. Теорема доказана.

Теорема A.19.4 *Рассмотрим правильную подстановку Δ, переменную $e_x \in \Sigma_e$, выражения $\mathcal{E}_a, \mathcal{E}_b \in E$, $\beta \in E_o$ и вектор пар строк $\overline{y}$. Пусть выполнены следующие условия:*

1. $e_x \notin \|\Delta\|$
2. $\mathcal{E}_a = \mathcal{T}_1\mathcal{T}_2\ldots\mathcal{T}_n$, *где для каждого $i = 1,2,\ldots,n$ либо $\mathcal{T}_i = e_x$, либо $\mathcal{T}_i$ – терм, жесткий при Δ.*
3. $\|e_x\mathcal{E}_a\| \cap [\ \|\mathcal{E}_b\| \cup \|\overline{y}\|\ ] \subseteq \|\Delta\| \cup \Sigma_c$
4. $\mathcal{E}_b = t_1 t_2 \ldots t_m e_y \mathcal{E}_c$, *где $t_1, t_2, \ldots, t_m \in \Sigma_t$, $e_y \in \Sigma_e$*
5. *Все переменные $t_1, t_2, \ldots, t_m \in \Sigma_t$, e_y – попарно различны и не входят в $\|\Delta\| \cup \|e_a\mathcal{E}_a\| \cup \|\mathcal{E}_c\| \cup \|\overline{y}\|$*
6. $\beta = \mu_1\ \mu_2 \ldots \mu_l \in T_o$

Обозначим $\beta'_i = \mu_1\ \mu_2 \ldots \mu_i$, $\beta''_i = \mu_{i+1}\ \mu_{i+2} \ldots \mu_l$. Тогда

$L[\Delta,(e_x\mathcal{E}_a\mathcal{E}_b,\beta)\ \overline{y}] = L[\Delta,(e_x\mathcal{E}_a,\beta'_k)] \cup L[\Delta.(\mathcal{E}_b,\beta''_k)\ \overline{y}]$,
где $k = \min\{i | W[\Delta,(e_x\mathcal{E}_a,\beta'_i)] \neq \emptyset\}$

Доказательство. Из условия 2 по теореме A.19.1 получаем, что $e_x\mathcal{E}_a$ удлиняется по β при Δ. Из условия 3 по теореме A.18.1 следует, что
$\|\Delta\|\perp(\|e_x\mathcal{E}_a\|, \|\mathcal{E}_b\| \cup \|\overline{y}\|)$.

Теперь предположим, что для некоторого i $W[\Delta,(e_x\mathcal{E}_a,\beta_{1:i})] \neq \emptyset$. Поскольку $e_x\mathcal{E}_a \in E$, по теореме A.16.4 $\beta_{1:i} \in E_o$. Поэтому, из условий 4 и 5, применяя теорему A.19.3, получаем, что для любых i и j, таких, что $W[\Delta,(e_x\mathcal{E}_a,\beta_{1:i})] \neq \emptyset$ и $W[\Delta,(e_x\mathcal{E}_a,\beta_{1:j})] \neq \emptyset$, как только $i < j$ и $W[\Delta,(\mathcal{E}_b,\beta_{j+1:|\beta|})\ \overline{y}] \neq \emptyset$, выполнено $W[\Delta,(\mathcal{E}_b,\beta_{i+1:|\beta|})\ y] \neq \emptyset$.

Таким образом, выполнены все условия теоремы A.14.3. Следовательно,

$L[\Delta,(e_x\mathcal{E}_a\mathcal{E}_b,\beta)\ \overline{y}] = L[\Delta,(e_x\mathcal{E}_a,\beta_{1:p})] \cup L[\Delta,(\mathcal{E}_b,\beta_{p+1:|\beta|})\ \overline{y}]$,
где $p = \min\{i | W[\Delta,(e_x\mathcal{E}_a,\beta_{1:i})] \neq \emptyset\}$.

Теперь осталось вспомнить, что если $W[\Delta, (e_x\mathcal{E}_a, \beta_{1:i})] \neq \emptyset$, то $\beta_{1:i} \in E_o$. Поэтому существует такое k, что $\beta_{1:p} = \beta'_k$, $\beta_{p+1:|\beta|} = \beta''_k$. Кроме того, если для некоторых i_1, i_2, j_1, j_2 выполнено $\beta_{1:i_1} = \beta'_{j_1}$ и $\beta_{1:i_2} = \beta'_{j_2}$, то $i_1 < i_2$ равносильно $j_1 < j_2$. Поэтому $k = \min\{i | W[\Delta, (e_x\mathcal{E}_a, \beta'_i)] \neq \emptyset\}$. Теорема доказана.