Теория и реализация языков программирования

В.А. Серебряков, М.П. Галочкин, Д.Р. Гончар, М.Г. Фуругян

Оглавление

0. Лекция: Предисловие

Предисловие

Предлагаемая вниманию читателя книга основана на курсе лекций, прочитанных на факультете вычислительной математики и кибернетики МГУ им. М.В. Ломоносова и на факультете управления и прикладной математики Московского физико-технического института в 1991-2002 гг. Авторы надеются, что издание книги восполнит существенный пробел в литературе на русском языке по разработке компиляторов.

В книге представлены "классические" разделы теории разработки компиляторов: лексический и синтаксический анализ, организация памяти компилятора (таблицы символов) и периода исполнения (магазина), генерация кода. Рассматриваются такие средства автоматизации процесса разработки трансляторов, как LEX, YACC, СУПЕР, методы генерации оптимального кода. Сделана попытка на протяжении всего изложения провести единую "атрибутную" точку зрения на процесс разработки компилятора. В книге не затрагиваются чрезвычайно важные вопросы глобальной оптимизации и разработки компиляторов для машин с параллельной архитектурой. Авторы надеются восполнить эти пробелы в будущем. Книга рассчитана как на студентов и аспирантов программистских специальностей, так и на профессионалов в области программирования.

Авторы благодарят всех, кто способствовал выходу этой книги в Свет и с признательностью примут все конструктивные замечания по её содержанию и оформлению.

1. Лекция: Введение

В данной лекции рассматривается место компилятора в программном обеспечении, который составляет существенную часть программного обеспечения ЭВМ. Приведены основные понятия, рассмотрена структура компилятора согласно фазам его действия.

Место компилятора в программном обеспечении

Компиляторы составляют существенную часть программного обеспечения ЭВМ. Это связано с тем, что языки высокого уровня стали основным средством разработки программ. Сегодня только очень малая часть программного обеспечения, требующая особой эффективности, разрабатывается с помощью ассемблеров. В настоящее время имеет применение довольно много языков программирования. Наряду с традиционными языками, такими, например, как Фортран, широкое распространение получили так называемые "универсальные" языки (Паскаль, Си, Модула-2, Ада и другие), а также некоторые специализированные (например, язык обработки списочных структур Лисп). Кроме того, большое распространение получили языки, связанные с узкими предметными областями, такие, как входные языки пакетов прикладных программ.

Для ряда названных языков имеется довольно много реализаций. Так, на рынке программного обеспечения представлены десятки реализаций языков Паскаля, Модулы-2 или Си для ЭВМ типа IBM PC.

С другой стороны, постоянно растущая потребность в новых компиляторах связана с бурным развитием архитектур ЭВМ. Это развитие идет по различным направлениям. Наряду с возникновением новых архитектур, совершенствуются старые архитектуры как в концептуальном отношении, так и по отдельным, конкретным параметрам. Это можно проиллюстрировать на примере микропроцессора Intel-80X86. Последовательные версии этого микропроцессора 8086, 80186, 80286, 80386, 80486, 80586 отличаются не только техническими характеристиками, но и, что более важно, новыми возможностями и, значит, изменением (расширением) системы команд. Естественно, это требует новых компиляторов (или модификации старых). То же можно сказать о микропроцессорах Motorola 68010, 68020, 68030, 68040.

В рамках традиционных последовательных машин развивается большое число различных направлений архитектур. Примерами могут служить архитектуры CISC, RISC. Такие ведущие фирмы, как Intel, Motorola, Sun, начинают переходить на выпуск машин с RISC- архитектурами. Естественно, для каждой новой системы команд требуется полный набор новых компиляторов с распространенных языков.

Наконец, бурно развиваются различные параллельные архитектуры. Среди них отметим векторные, многопроцессорные, с широким командным словом архитектуры (вариантом которых являются суперскалярные ЭВМ). На рынке уже имеются десятки типов ЭВМ с параллельной архитектурой, начиная от супер-ЭВМ (Cray, CDC и другие), через рабочие станции (например, IBM RS/6000) и кончая персональными компьютерами (например, на основе микропроцессора I-860). Естественно, для каждой из новых машин создаются новые компиляторы для многих языков программирования. Здесь необходимо также отметить, что новые архитектуры требуют разработки совершенно новых подходов к созданию компиляторов, так что наряду с собственно разработкой компиляторов ведется и большая научная работа по созданию новых методов

Структура компилятора

Обобщенная структура компилятора и основные фазы компиляции показаны на рис. 1.1

На начальной фазе лексического анализа входная программа, представляющая собой поток литер, разбивается на лексемы - слова в соответствии с определениями языка. Основными формализмами, лежащим в основе реализации лексических анализаторов, являются конечные автоматы и регулярные выражения. Лексический анализатор может работать в двух основных режимах: либо как подпрограмма, вызываемая синтаксическим анализатором для получения очередной лексемы, либо как полный проход, результатом которого является файл лексем.

В процессе выделения лексем лексический анализатор может как самостоятельно строить таблицы объектов (чисел, строк, идентификаторов и так далее), так и выдавать значения для каждой лексемы при очередном к нему обращении. В этом случае таблицы объектов строятся в последующих фазах (например, в процессе синтаксического анализа).

На этапе лексического анализа обнаруживаются некоторые (простейшие) ошибки (недопустимые символы, неправильная запись чисел, идентификаторов и другие).

Центральная задача синтаксического анализа - разбор структуры программы. Как правило, под структурой понимается дерево, соответствующее разбору в контекстно- свободной грамматике языка. В настоящее время чаще всего используется либо LL(1)-анализ (и его вариант - рекурсивный спуск), либо LR(1)-анализ и его варианты (LR(0), SLR(1), LALR(1) и другие). Рекурсивный спуск чаще используется при ручном программировании синтаксического анализатора, LR(1) - при использовании систем автоматического построения синтаксических анализаторов.

Результатом синтаксического анализа является синтаксическое дерево со ссылками на таблицы объектов. Ошибки, связанные со структурой программы, также обнаруживаются в процессе синтаксического анализа. На этапе контекстного анализа выявляются зависимости между частями программы, которые не могут быть описаны контекстно-свободным синтаксисом. Это, в основном, связи "описание-использование", в частности, анализ типов объектов, анализ областей видимости, соответствие параметров, метки и другие. В процессе контекстного анализа таблицы объектов пополняются информацией об описаниях (свойствах) объектов.

Основным формализмом, используемым при контекстном анализе, является аппарат атрибутных грамматик. Результатом контекстного анализа является атрибутированное дерево программы. Информация об объектах может быть как рассредоточена в самом дереве, так и сосредоточена в отдельных таблицах объектов. В процессе контекстного анализа также могут быть обнаружены ошибки, связанные с неправильным использованием объектов.

Затем программа может быть переведена во внутреннее представление. Это делается для целей оптимизации и/или удобства генерации кода. Еще одной целью преобразования программы во внутреннее представление является желание иметь переносимый компилятор. Тогда только последняя фаза (генерация кода) является машинно-зависимой. В качестве внутреннего представления может использоваться префиксная или постфиксная запись, ориентированный граф, тройки, четверки и другие способы.

Фаз оптимизации может быть несколько. Оптимизации обычно делят на машинно-зависимые и машинно-независи- мые, локальные и глобальные. Определенная часть машин- но-зависимой оптимизации выполняется на фазе генерации кода. Глобальная оптимизация пытается принять во внимание структуру всей программы, локальная - только небольших ее фрагментов. Глобальная оптимизация основывается на глобальном потоковом анализе, который выполняется на графе программы и представляет по существу преобразование этого графа. При этом могут учитываться такие свойства программы, как межпроцедурный анализ, межмодульный анализ, анализ областей жизни переменных и так далее.

Наконец, генерация кода - последняя фаза трансляции. Результатом ее является либо ассемблерный модуль, либо объектный (или загрузочный) модуль. В процессе генерации кода могут выполняться некоторые локальные оптимизации, такие как распределение регистров, выбор длинных или коротких переходов, учет стоимости команд при выборе конкретной последовательности команд. Для генерации кода разработаны различные методы, такие как таблицы решений, сопоставление образцов, включающее динамическое программирование, различные синтаксические методы. Конечно, те или иные фазы транслятора могут либо отсутствовать совсем, либо объединяться. В простейшем случае однопроходного транслятора нет явной фазы генерации промежуточного представления и оптимизации, остальные фазы объединены в одну, причем нет и явно построенного синтаксического дерева.

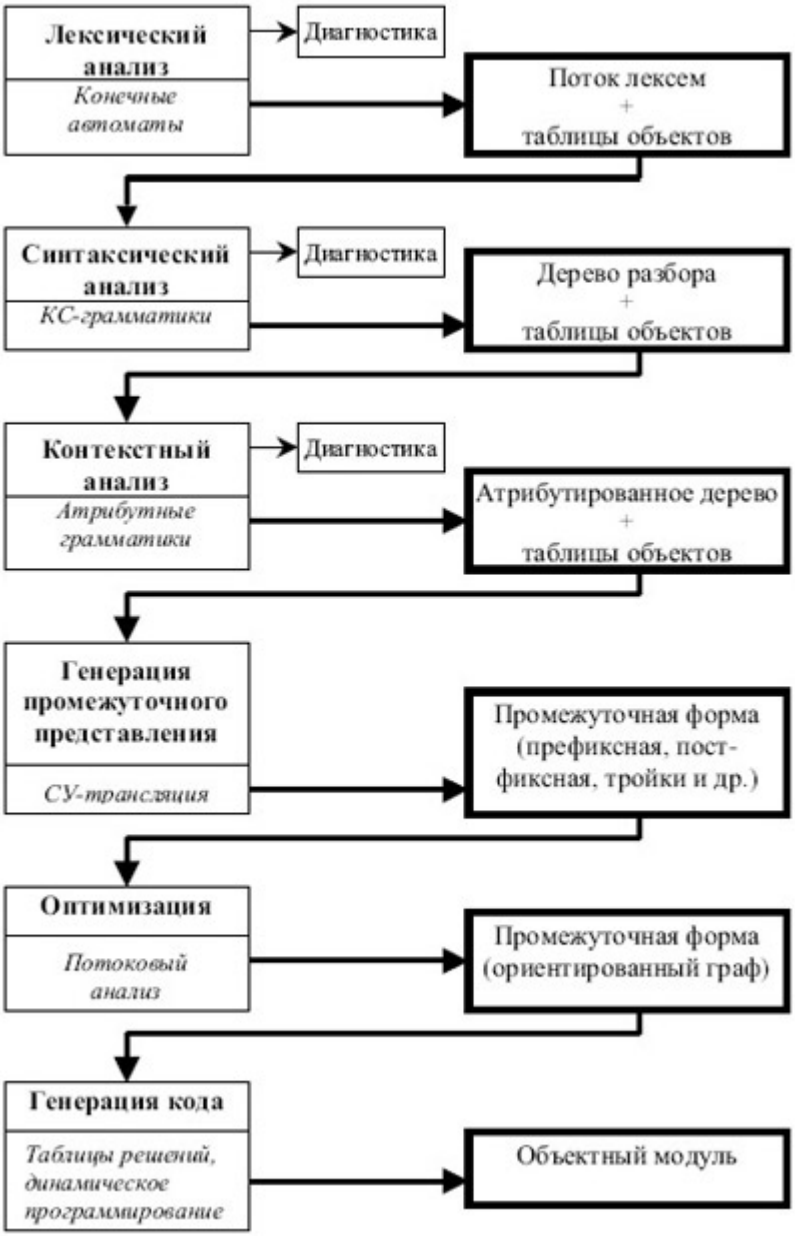


Рис. 1.1.

2. Лекция: Языки и их представление

В данной лекции рассматривается понятие языков и их представление. Приведены такие определения, как алфавит, цепочка, грамматика, машина Тьюринга. Также приведены примеры практической реализации основных понятий в теории программирования.

Алфавиты, цепочки и языки

Алфавит, или словарь - это конечное множество символов. Для обозначения символов мы будем пользоваться цифрами, латинскими буквами и специальными литерами типа $, $

Пусть V - алфавит. Цепочка в алфавите V - это любая строка конечной длины, составленная из символов алфавита V . Синонимом цепочки являются предложение, строка и слово. Пустая цепочка (обозначается e) - это цепочка, в которую не входит ни один символ.

Конкатенацией цепочек x и y называется цепочка xy. Заметим, что xe = ex = x для любой цепочки x.

Пусть x, y, z - произвольные цепочки в некотором алфавите. Цепочка y называется подцепочкой цепочки xyz. Цепочки x и y называются, соответственно, префиксом и суффиксом цепочки xy. Заметим, что любой префикс или суффикс цепочки является подцепочкой этой цепочки. Кроме того, пустая цепочка является префиксом, суффиксом и подцепочкой для любой цепочки.

Пример 2.1. Для цепочки abbba префиксом является любая цепочка из множества $L_1 = \{e, a, ab, abb, abbb, abbba\}$ суффиксом является любая цепочка из множества $\{e, a, ba, bba, bbba, abbba\}$ подцепочкой является любая цепочка из множества $L_1 \bigcup L_2$

Длиной цепочки w (обозначается |w|) называется число символов в ней. Например, |abababa| = 7, а |e| = 0. Язык в алфавите V - это некоторое множество цепочек в алфавите V .

Пример 2.2. Пусть дан алфавит V = {a, b}. Вот некоторые языки в алфавите V:

1. $L_1 = \emptyset$ — пустой язык;
2. $L_2 = \{e\}$ - язык, содержащий только пустую цепочку (заметим, что L_1 и L_2 - различные языки)
3. L_3 - язык, содержащий цепочки из a и b, длина которых не превосходит 2
4. L_4 - язык, включающий всевозможные цепочки из a и b, содержащие четное число a и четное число b
5. $L_5 = \{a^{n^2} | n > 0\}$ - язык цепочек из a, длины которых представляют собой квадраты натуральных чисел.

Два последних языка содержат бесконечное число цепочек.

Введем обозначение V^* для множества всех цепочек в алфавите V^*, включая пустую цепочку. Каждый язык в алфавите V является подмножеством V^*. Для обозначения множества всех цепочек в алфавите V , кроме пустой цепочки, будем использовать V^+.

Пример 2.3. Пусть $V = \{0, 1\}$. Тогда
$V^* = \{e, 0, 1, 00, 01, 10, 11, 000, \ldots\}, V^+ = \{0, 1, 00, 01, 10, 11, 000, \ldots\}$

Введем некоторые операции над языками.

Пусть L_1 и L_2 - языки в алфавите V . Конкатенацией языков L_1 и L_2 называется язык $L_1 L_2 = \{xy | x \in L_1, y \in L_2\}$.

Пусть L - язык в алфавите V . Итерацией языка L называется язык L^*, определяемый следующим образом:

$$L^0 = \{|e\} \qquad (1)$$

$$L^n = LL^{n-1}, n \geq 1 \quad (2)$$

$$L^* = \bigcup_{n=0}^{\infty} L^n \quad (3)$$

Пример 2.4. Пусть $L_1 = \{aa, bb\}$ и $L_2 = |e, a, bb\}$. Тогда

$L_1 L_2 = |aa, bb, aaa, bba, aabb, bbbb\}$, и $L_1^* = \{e, aa, bb, aaaa, aabb, bbaa, bbbb, aaaaaa, \ldots\}$

Большинство языков, представляющих интерес, содержат бесконечное число цепочек. При этом возникают три важных вопроса.

Во-первых, как представить язык (то есть специфицировать входящие в него цепочки)? Если язык содержит только конечное множество цепочек, ответ прост. Можно просто перечислить его цепочки. Если язык бесконечен, необходимо найти для него конечное представление. Это конечное представление, в свою очередь, будет строкой символов над некоторым алфавитом вместе с некоторой интерпретацией, связывающей это представление с языком.

Во-вторых, для любого ли языка существует конечное представление? Можно предположить, что ответ отрицателен. Мы увидим, что множество всех цепочек над алфавитом счетно. Язык - это любое подмножество цепочек. Из теории множеств известно, что множество всех подмножеств счетного множества несчетно. Хотя мы и не дали строгого определения того, что является конечным представлением, интуитивно ясно, что любое разумное определение конечного представления ведет только к счетному множеству конечных представлений, поскольку нужно иметь возможность записать такое конечное представление в виде строки символов конечной длины. Поэтому языков значительно больше, чем конечных представлений.

В-третьих, можно спросить, какова структура тех классов языков, для которых существует конечное представление?

Представление языков

Процедура - это конечная последовательность инструкций, которые могут быть механически выполнены. Примером может служить машинная программа. Процедура, которая всегда заканчивается, называется алгоритмом.

Один из способов представления языка - дать алгоритм, определяющий, принадлежит ли цепочка языку. Более общий способ состоит в том, чтобы дать процедуру, которая останавливается с ответом "да" для цепочек, принадлежащих языку, и либо останавливается с ответом "нет", либо вообще не останавливается для цепочек, не принадлежащих языку. Говорят, что такая процедура или алгоритм распознает язык.

Такой метод представляет язык с точки зрения распознавания. Язык можно также представить методом порождения. А именно, можно дать процедуру, которая систематически порождает в определенном порядке цепочки языка.

Если мы можем распознать цепочки языка над алфавитом V либо с помощью процедуры, либо с помощью алгоритма, то мы можем и генерировать язык, поскольку мы можем систематически генерировать все цепочки из V^*, проверять каждую цепочку на принадлежность языку и выдавать список только цепочек языка. Но если процедура не всегда заканчивается при проверке цепочки, мы не сдвинемся дальше первой цепочки, на которой процедура не заканчивается. Эту проблему можно обойти, организовав проверку таким образом, чтобы процедура никогда не продолжала проверять одну цепочку бесконечно. Для этого введем следующую конструкцию.

Предположим, что V имеет p символов. Мы можем рассматривать цепочки из V^* как числа, представленные в базисе p, плюс пустая цепочка e. Можно занумеровать цепочки в порядке возрастания длины и в "числовом" порядке для цепочек одинаковой длины. Такая нумерация для цепочек языка $\{a; b; c\}^*$ приведена на рис. 2.1, а.

Пусть P - процедура для проверки принадлежности цепочки языку L. Предположим, что P может быть представлена дискретными шагами, так что имеет смысл говорить об i-ом шаге процедуры для любой данной цепочки. Прежде чем дать процедуру перечисления цепочек языка L, дадим процедуру нумерации пар положительных чисел.

Все упорядоченные пары положительных чисел (x, y) можно отобразить на множество положительных чисел следующей формулой:

z = (x + y - 1)(x + y - 2)/2 + y

Пары целых положительных чисел можно упорядочить в соответствии со значением z (рис. 2.1, б).

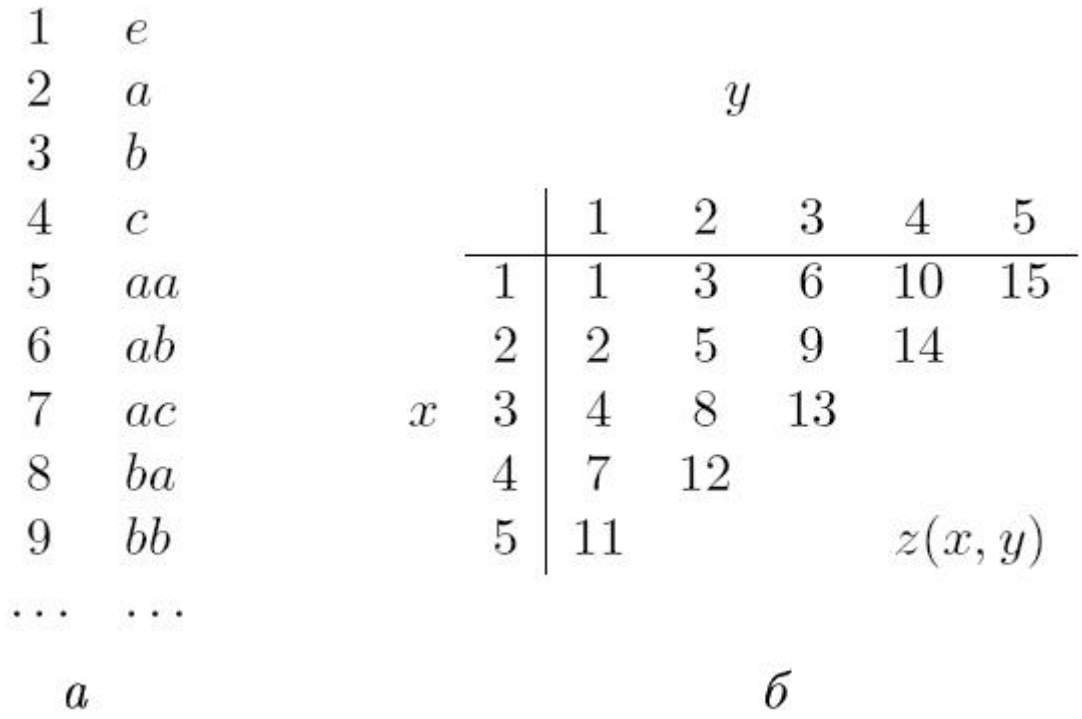


Рис. 2.1.

Теперь можно дать процедуру перечисления цепочек L. Нумеруем упорядоченные пары целых положительных чисел - (1,1), (2,1), (1,2), (3,1), (2,2), ... При нумерации пары (i, j) генерируем i-ю цепочку из V* и применяем к цепочке первые j шагов процедуры P. Как только мы определили, что сгенерированная цепочка принадлежит L, добавляем цепочку к списку элементов L. Если цепочка i принадлежит L, это будет определено P за j шагов для некоторого конечного j. При перечислении (i; j) будет сгенерирована цепочка с номером i. Легко видеть, что эта процедура перечисляет все цепочки L.

Если мы имеем процедуру генерации цепочек языка, то мы всегда можем построить процедуру распознавания предложений языка, но не всегда алгоритм. Для определения того, принадлежит ли x языку L, просто нумеруем предложения L и сравниваем x с каждым предложением. Если сгенерировано x, процедура останавливается, распознав, что x принадлежит L. Конечно, если x не принадлежит L, процедура никогда не закончится.

Язык, предложения которого могут быть сгенерированы процедурой, называется рекурсивно перечислимым. Язык рекурсивно перечислим, если имеется процедура, распознающая предложения языка. Говорят, что язык рекурсивен, если существует алгоритм для распознавания языка. Класс рекурсивных языков является собственным подмножеством класса рекурсивно перечислимых языков. Мало того, существуют языки, не являющиеся даже рекурсивно перечислимыми.

Грамматики

Формальное определение грамматики

Для нас наибольший интерес представляет одна из систем генерации языков - грамматики. Понятие грамматики изначально было формализовано лингвистами при изучении естественных языков. Предполагалось, что это может помочь при их автоматической трансляции. Однако, наилучшие результаты в этом направлении достигнуты при описании не естественных языков, а языков программирования. Примером может служить способ описания синтаксиса языков программирования при помощи БНФ - формы Бэкуса-Наура.

Определение. Грамматика - это четверка G = (N,T,P,S), где

(1) N - алфавит нетерминальных символов;

(2) T - алфавит терминальных символов, $N \cap T = \emptyset$

(3) P - конечное множество правил вида $\alpha \to \beta$**, где** $\alpha \in (N \cup T)^* N (N \cup T)^*, \beta \in (N \cup T)^*$

(4) $S \in N$ - начальный знак (или аксиома) грамматики.

Мы будем использовать большие латинские буквы для обозначения нетерминальных символов, малые латинские буквы из начала алфавита для обозначения терминальных символов, малые латинские буквы из конца алфавита для обозначения цепочек из T^* и, наконец, малые греческие буквы для обозначения цепочек из $(N \cup T)^*$.

Будем использовать также сокращенную запись $A \to \alpha_1 | \alpha_2 | \ldots | \alpha_n$ для обозначения группы правил $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_n$.

Определим на множестве $(N \cup T)^*$ бинарное отношение выводимости $\Rightarrow$ следующим образом: если $\delta \to \gamma \in P$, то $\alpha\delta\beta \Rightarrow \alpha\gamma\beta$ для всех $\alpha, \beta \in (N \cup T)^*$. Если $\alpha_1 \Rightarrow \alpha_2$, то говорят, что цепочка α_2 непосредственно выводима из α_1.

Мы будем использовать также рефлексивно-транзитивное и транзитивное замыкания отношения $\Rightarrow$, а также его степень $k \geq 0$ (обозначаемые соответственно $\Rightarrow^*$, $\Rightarrow^+$ и $\Rightarrow^k$). Если $\alpha_1 \Rightarrow^* \alpha_2 (\alpha_1 \Rightarrow^+ \alpha_2, \alpha_1 \Rightarrow^k \alpha_2)$, то говорят, что цепочка α_2 выводима (нетривиально выводима, выводима за k шагов) из α_1.

Если $\alpha \Rightarrow^k \beta (k \geq 0)$, то существует последовательность шагов

$$\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{k-1} \Rightarrow \gamma_k$$

где $\alpha = \gamma_0$ и $\beta = \gamma_k$. Последовательность цепочек $\gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_k$ в этом случае называют выводом β из α

Сентенциальной формой грамматики G называется цепочка, выводимая из ее начального символа.

Языком, порождаемым грамматикой G (обозначается L(G)), называется множество всех ее терминальных сентенциальных форм, то есть

$$L(G) = \{\omega \mid \omega \in T^*, S \to^+ \omega\}$$

Грамматики G1 и G2 называются эквивалентными, если они порождают один и тот же язык, то есть $L(G_1) = L(G_2)$

Пример 2.5. Грамматика G = ({S, B, C}, {a, b, c}, P, S), где $P = \{S \to aSBC,\ S \to aBC,\ CB \to BC,\ aB \to ab,\ bB \to bb,\ bC \to bc,\ cC \to cc\}$, порождает язык $L(G) = \{a^n b^n c^n \mid n > 0\}$

Действительно, применяем n - 1 раз правило 1 и получаем $a^{n-1} S (BC)^{n-1}$, затем один раз правило 2 и получаем $a^n (BC)^n$, затем n(n - 1)/2 раз правило 3 и получаем $a^n B^n C^n$.

Затем используем правило 4 и получаем $a^nbB^{n-1}C^n$. Затем применяем n - 1 раз правило 5 и получаем $a^nb^nC^n$. Затем применяем правило 6 и n - 1 раз правило 7 и получаем $a^nb^nc^n$. Можно показать, что язык L(G) состоит из цепочек только такого вида.

Пример 2.6. Рассмотрим грамматику $G = (\{S\}, \{0,1\}, \{S \to 0S1,\ S \to 01\}, S)$. Легко видеть, что цепочка $000111 \in L(G)$, так как существует вывод

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000111$$

Нетрудно показать, что грамматика порождает язык $L(G) = \{0^n1^n \mid n > 0\}$.

Пример 2.7. Рассмотрим грамматику
$G = (\{S, A\}, \{0,1\}, \{S \to 0S,\ S \to 0A,\ A \to 1A,\ A \to 1\},\ S)$. Нетрудно показать, что грамматика порождает язык $L(G) = \{0^n1^m \mid n, m > 0\}$

Типы грамматик и их свойства

Рассмотрим классификацию грамматик (предложенную Н.Хомским), основанную на виде их правил.

Определение. Пусть дана грамматика G = (N, T, P, S). Тогда

(1) если правила грамматики не удовлетворяют никаким ограничениям, то ее называют грамматикой типа 0, или грамматикой без ограничений.

(2) если

1. каждое правило грамматики, кроме $S \to \varepsilon$, имеет вид $\alpha \to \beta$, где $|\alpha| \le |\beta|$, и
2. в том случае, когда $S \to \varepsilon \in P$, символ S не встречается в правых частях правил, то грамматику называют грамматикой типа 1, или неукорачивающей или контекстно-зависимой (КЗ- грамматикой) или контекстно - чувствительной (КЧ- грамматикой).

(3) если каждое правило грамматики имеет вид $A \to \beta$, где $A \in N, \beta \in (N \cup T)^*$, то ее называют грамматикой типа 2, или контекстно-свободной (КС-грамматикой).

(4) если каждое правило грамматики имеет вид либо $A \to xB$, либо $A \to x$, где $A, B \in N, x \in T^*$ то ее называют грамматикой типа 3, или праволинейной.

Легко видеть, что грамматика в примере 2.5 - неукорачивающая, в примере 2.6 - контекстно-свободная, в примере 2.7 - праволинейная.

Язык, порождаемый грамматикой типа i, называют языком типа i. Язык типа 0 называют также языком без ограничений, язык типа 1 - контекстно-зависимым (КЗ), язык типа 2 - контекстно-свободным (КС), язык типа 3 - праволинейным.

Теорема 2.1. Каждый контекстно-свободный язык может быть порожден неукорачивающей контекстно-свободной грамматикой.

Доказательство. Пусть L - контекстно-свободный язык. Тогда существует контекстно-свободная грамматика G = (N, T, P, S), порождающая L.

Построим новую грамматику G' = (N',T,P',S') следующим образом:

1. Если в P есть правило вида $A \to \alpha_0 B_1 \alpha_1 \ldots B_k \alpha_k$, где $k \geq 0,\ B_i \Rightarrow^+ e$ для $1 \leq i \leq k$ и ни из одной цепочки $\alpha_j (0 \leq j \leq k)$ не выводится e, то включить в P' все правила (кроме $A \to e$) вида

   $$A \to \alpha_0 X_1 \alpha_1 \ldots X_k \alpha_k$$

   где X_i это либо B_i, либо e.

2. Если $S \to^+ e$, то включить в P' правила $S' \to S,\ S' \to e$ и положить $N' = N \cup \{S'\}$. В противном случае положить $N' = N$ и $S' = S$. Порождает ли грамматика пустую цепочку можно установить следующим простым алгоритмом:

   Шаг 1. Строим множество $N_0 = \{N \mid N \to e\}$

   Шаг 2. Строим множество $N_i = N_{i-1} \cup \{N \mid N \to \alpha,\ \alpha \in \{N_{i-1}\}^*\}$

   Шаг 3. Если $N_i = N_{i-1}$, перейти к шагу 4, иначе шаг 2.

   Шаг 4. Если $S \in N_i$, значит $S \to^* e$.

   Легко видеть, что G' - неукорачивающая грамматика. Можно показать по индукции, что $L(G') = L(G)$.

Пусть K_i - класс всех языков типа i. Доказано, что справедливо следующее (строгое) включение: $K_3 \subset K_2 \subset K_1 \subset K_0$.

Заметим, что если язык порождается некоторой грамматикой, это не означает, что он не может быть порожден грамматикой с более сильными ограничениями на правила. Приводимый ниже пример иллюстрирует этот факт.

Пример 2.8. Рассмотрим грамматику $G = (\{S, A, B\}, \{0, 1\}, \{S \to AB,\ A \to 0A,\ A \to 0,\ B \to 1B,\ B \to 1\},\ S)$. Эта грамматика является контекстно-свободной. Легко показать, что $L(G) = \{0^n 1^m \mid n, m > 0\}$. Однако, в примере 2.7 приведена праволинейная грамматика, порождающая тот же язык.

Ниже приводятся подробные примеры решения двух практически интересных более сложных задач на построение КС- и НС-грамматик.

Пример 2.9. Данный пример относится к несколько парадоксальной для грамматик постановке: построить КС- грамматику, порождающую язык:

$$\{\{a, b\}^* \backslash a^n b^m a^n b^m \mid n, m \geq 1\}$$

т.е. построить все цепочки кроме указанных (обычно-то говорят о том, что надо построить). Но, может быть, в такой постановке заложена и подсказка к решению? Известно, что иные задачи с подобными требованиями так и решаются: нужно сделать все, "что не надо", а потом отклониться от этого "не надо" всеми возможными способами.

Однако воодушевл_нных построением в рамках КС- грамматики цепочек вида $\{a^n b^m a^n b^m\}$ (здесь и далее в этом примере $n, m, k, l, j \geq 1$) ждет некоторое разочарование. Действительно, в отличие от таких случаев, как $\{a^n b^n a^m b^m\}$, $\{a^n b^m a^m b^n\}$, $\{a^n b^m b^n a^m\}$ и т.п., обе зависимости (по n и по m)

придется отслеживать одновременно и из двух разных центров порождения, к чему КС-грамматики по своей природе (виду своих правил) оказываются не предназначены.

Попробуем тогда пересказать условие задачи в конструктивном (созидательном) плане, т.е. обозначая лишь то, что нам нужно построить, а не наоборот. Поначалу такое множество цепочек кажется необозримым. Но попробуем, "Дорогу осилит идущий"! Начнем с очевидных случаев:

$$\{a^n\}, \{b^n\}, \{a^n b^m\}, \{b^n a^m\}, \{a^n b^m a^k\}, \{b^n a^m b^k\} \ldots$$

Однако бесконечно продолжать в духе $\{b^n a^m b^k a^l b^j\}$ уже как- то скучно. Замечаем, что $b\{a,b\}^*$ вполне конечным образом определяет половину из упомянутых бесчисленных описаний, а в следующий момент симметрия нам подсказывает и язык $\{a,b\}^* a$.

Таким образом, все цепочки вышеперечисленных видов укладываются в три случая:

$$\{a^n b^m\},\ b\{a,b\}^*,\ \{a,b\}^* a$$

Далее рассмотрим случай $\{a^n b^m a^k b^l \mid n \neq k \text{ или } m \neq l\}$. Но что такое, к примеру, $n \neq k$? То же самое, что объединение условий $n > k$ и $n < k$! И здесь перешли к конструктиву, который несложно строится в рамках КС-грамматики.

Остается единственный неупомянутый случай:

$$\{a^n b^m a^k b^l\} a \{a,b\}^*$$

Вспоминая, что объединение КС-языков есть КС-язык, получаем искомое решение задачи.

Так, если язык $\{a^n b^m\}$ может быть порожден грамматикой

$$S_1 \to AB$$
$$A \to aA \mid a$$
$$B \to bB \mid b$$

а язык $b\{a,b\}^*$ - грамматикой

$$S_2 \to bC$$
$$C \to CC \mid a \mid b \mid \varepsilon,$$

то для объединения этих языков (в общем случае использующих каждый свой уникальный набор вспомогательных знаков) достаточно добавить правило старта из новой общей аксиомы:

$$S \to S_1 \mid S_2$$

Пример 2.10. Построение НС-грамматики.

Грамматики непосредственных составляющих (или, кратко, НС-грамматики) есть вид представления контекстно-зависи- мых грамматик, т.е. они обладают теми же выразительными возможностями, что и КЗ-грамматики в целом. Каждое правило НС-грамматики должно соответствовать виду:

$$\varphi A \psi \to \varphi \eta \psi,\ (|\ \eta\ | \geq 1)$$

то есть левое и правое окружение (контекст) заменяемого знака А должны сохраниться и вокруг непустой заменяющей цепочки η (греческая буква "эта".

Такое дополнительное ограничение позволяет удобнее переходить от КЗ-грамматики к соответствующему линейно- ограниченному автомату

Рассмотрим построение НС-грамматики для языка

$\{a^{n^2} \mid n \geq 0\}$, порождающего слова вида $\varepsilon,\ a,\ a^4,\ a^9,\ \ldots$

Для большей ясности сперва построим для этого языка грамматику общего вида, а потом перестроим ее в соответствии с НС-ограничениями.

Сам алгоритм порождения a^{n^2} может основываться как на известном свойстве квадратов чисел, разность между соседними из которых образуют арифметическую прогрессию, так и на собственно "квадратности" интересующих чисел, т.е. того, что каждое квадратное число представимо наподобие матрицы из n строк и n столбцов единичных элементов (в связи с чем Пифагор и дал название подобным числам - квадратные, а среди других чисел по тому же принципу отметил треугольные, кубические, пирамидальные и т.п.). Последний подход представляется более общим, поскольку подобным образом мы сможем построить и $\{a^{n^k} \mid k > 2\}$.

Итак, порождаем две группы по n элементов

Правила	Вид получаемой цепочки
$S \to CSD \mid CD$	$C^n D^n$
$CD \to DCA$	$C^{n-1}\, DCAD^{n-1}$ (A задерживает C)
$CA \to AC$	$(DA^n)^n C^n$ (отработали все C)
$A \to a$	$(Da^n)^n C^n$

Получили a^{n^2}, но что делать с C и D? Сделав свое дело, они стали лишними.

В грамматике общего вида такие знаки сокращают ("увольняют"), а в КЗ-грамматиках - "переводят на другую работу" (в основные знаки). Но если мы просто напишем $C \to \varepsilon, D \to \varepsilon$, вывод в случайный момент времени может закончиться досрочно и станет возможным порождение лишних цепочек.

Поэтому в обоих случаях не обойтись без дальнейшего уточнения предназначения (миссий) и состава "действующих лиц". Отметим для этого самый первый из команды знаков C (назовем его B) и самый последний из D (обозначим его E). Когда B и E встретятся, это и будет признаком полного завершения процесса порождения знаков a.

Начнем вывод с начала:

Правила	Вид получаемой цепочки
$S \to BS'E$	$BS'E$
$S' \to CS'D \mid CD$	$BC^n D^n E$
$CD \to DCA$	$BC^{n-1}\,(DA)^n\,CE$ (C прошло первый раз)
$CA \to AC$	$B(DA^n)^n C^n\ E$ (прошли все C)
$BD \to B$	$BA^n\,(DA^n)^{n-1}\,C^n\,E$
$BA \to AB$	$A^{n*n} BC^n E$ (ушли все D)
$CE \to E$	$A^{n*n} BE$ (ушли все C)
$BE \to \varepsilon$	A^{n*n} (ушли B с E)
$A \to a$	a^{n*n}

Результат получили, но какой ценой (для B, C, D и E)? Прямо-таки сталинские методы. Точнее скажем, в военных или иных чрезвычайных условиях иначе, порой, и нет возможности поступить. А в более мирное время? Попробуем "соблюдать КЗОТ" и обойтись без сокращений.

Снова:

Правила	Вид получаемой цепочки
$S' \to BSE$	BSE
$S \to CSD \mid CD$	BC^nD^nE
$CD \to DCA$	$BC^{n-1}(DA)^nCE$ (C прошло первое C)
$CA \to AC$	$B(DA^n)^nC^nE$ (прошли все C)
$BD \to aaB$	$a^{2n}A^n(DA^n)^{n-1}C^nE$
$BA \to AB$	$(a^2A^n)^nBC^nE$ (ушли все D)
$CE \to Eaa$	$(a^2A^n)^nBEa^{2n}$ (ушли все C)
$A \to a$	$a^{n*n+2n}BEa^{2n}$ (ушли B с E)
$BE \to aaaa$	$a^{n^2+4n+4} = a^{(n+2)^2}$
$S \to \varepsilon \mid a \mid a^4$	(восполнили частные решения)

Итак, если сокращать нельзя, достраиваем слово до ближайшего подходящего квадрата. В данном случае удобнее достроить слово до $a^{(n+2)^2}$, т.к. для достройки до $a^{(n+1)^2}$ нам бы потребовалось перевести $BE \to !a$, т.е. опять что-то сократить. Напомним, что в КЗ-грамматиках допускается переход аксиомы в пустую цепочку (ε), если аксиома нигде более не встречается в правых частях правил (т.е. когда из начального ничего получают другое ничего).

Мы получили несокращающую грамматику. Но широко используемые при ее построении правила вида $AB \to !BA$ ($ABC \to !CBA$ и т.п.), очевидно, не подходят под определение НС-грамматики (убедитесь!). Такие "рокировки" , однако, легко раскрыть через цепочку правил вида

$$AB \to A'B \to A'B' \to BB' \to BA$$

где A' и B'- нигде более в грамматике не используемые вспомогательные знаки. Отметим, что замену на промежуточные знаки и обратно на исходные нужно осуществлять в одном и том же порядке (слева-направо или, наоборот, только справа-налево), иначе в общем случае (когда назначение A и B в грамматике различно) возникают лишние цепочки.

Так, применение замены

$$AB \to A'B \to A'B' \to A'A \to BA$$

(нарушен порядок замен) при наличии соответствующего прово- кационного окружения допускает подмену B на A:

$$x - AABB \xrightarrow{*} AA'AB \xrightarrow{*} AA'BA \xrightarrow{*} ABAA - y$$

$$(|x|_A = |x|_B = 2, \quad a\ |y|_A = 3 \text{ и } |y|_B = 1)!$$

Замена AB на BA в рамках НС-грамматики коротко обозначается, как и обычный вывод: $AB \xrightarrow{*} BA$.

Таким образом, один из возможных наборов правил искомой НС-грамматики имеет следующий вид:

Правила	Вид получаемой цепочки
$S' \to \varepsilon \mid a \mid a^4 \mid BSE$	BSE
$S \to CSD \mid CD$	$BC^nD^n E$
$CD \xrightarrow{*} DCA$	$BC^{n-1}DCAD^{n-1}E$
$CA \xrightarrow{*} AC$	$BC^{n-1}(DA)^nCE$
	$B(DA^n)^nC^nE$
$BD \xrightarrow{*} aaB$	$a^2BA^n(DA^n)^{n-1}C^nE$
$BA \xrightarrow{*} AB$	$(a^2A^n)^nBC^nE$
$CE \xrightarrow{*} Eaa$	$(a^2A^n)^nBEa^{2n}$
$A \xrightarrow{*} a$	$a^{n*n+2n}BEa^{2n}$
$BE \xrightarrow{*} a^4$	$a^{n^2+4n+4} = a^{(n+2)^2}$

Машины Тьюринга

Формально машина Тьюринга (Tm) - это $Tm = (Q, \Gamma, \Sigma, D, q_0, F)$, где

Q - конечное множество состояний;

$F \subseteq Q$ - множество заключительных состояний;

Γ - множество допустимых ленточных символов; один из них, обычно обозначаемый B, - пустой символ

Σ - множество входных символов, подмножество \Gamma, не включающее B,

D функция переходов, отображение из $(Q - F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$; для некоторых аргументов функция D может быть не определена.

q_0 - начальное состояние.

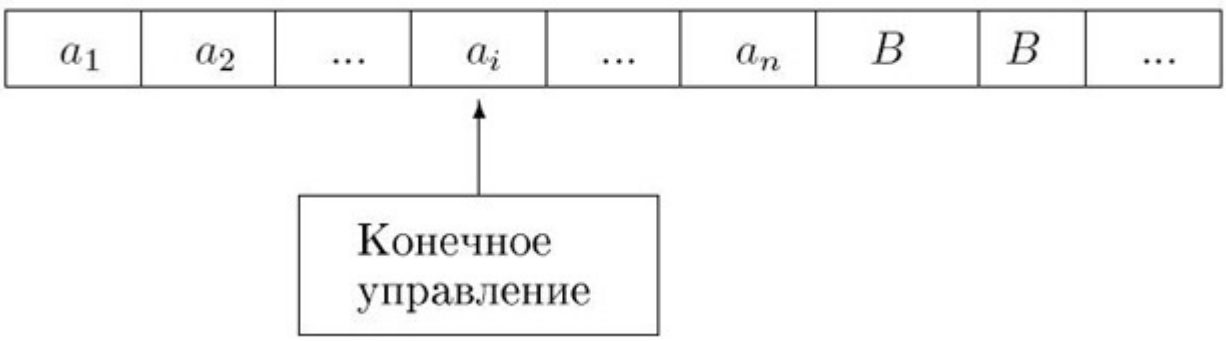


Рис. 2.2. Машина Тьюринга

Так определенная машина Тьюринга называется детерминированной. Недетерминированная машина Тьюринга для каждой пары $(Q - F) \times \Gamma$ может иметь несколько возможных переходов. В начале n ячеек ленты содержат вход $w \in (\Gamma - \{B\})^*$, остальная часть ленты содержит пустые символы. Обозначим конфигурацию машины Тьюринга как (q, w, i), где $q \in Q$ - текущее состояние, i - выделенный элемент строки, "положение головки" , w - текущее содержимое занятого участка ленты. Если головка сдвигается с ячейки, машина должна записать в нее символ, так что лента всегда состоит из участка, состоящего из конечного числа непустых символов и бесконечного количества пустых символов.

Шаг Tm определим следующим образом.

Пусть (q, A_1, A_2, ... A_n, i) - конфигурация Tm,

где $1 \leq i \leq n+1$.

Если $1 \leq i \leq n$ и D(q, A_i) = (p, A, R)

(R от англ. Right), то $A \neq B$ и)

$$(q,\ A_1 A_2 \ldots A_n, i) \vdash_{Tm} (p,\ A_1 A_2 \ldots A_{i-1} A A_{i+1} \ldots A_n,\ i+1)$$

То есть Tm печатает символ A и передвигается вправо.

Если $2 \leq i \leq n$ и $D(q, A_i) = (p, A, L)$

(L от англ. Left), то если i = n, то допустимо A = B и

$$(q,\ A_1 A_2 \ldots A_n, i) \vdash_{Tm} (p,\ A_1 A_2 \ldots A_{i-1} A A_{i+1} \ldots A_n,\ i-1)$$

Tm печатает A и передвигается влево, но не за конец ленты.

Если i = n + 1, головка просматривает пустой символ B.

Если D(q, B) = (p, A, R), то $A \neq B$ и

$$(q,\ A_1 A_2 \ldots A_n, n+1) \vdash_{Tm} (p,\ A_1 A_2 \ldots A_n A, n+2)$$

Если D(q, B) = (p, A, L), то допустимо A=B и

$$(q,\ A_1 A_2 \ldots A_n, n+1) \vdash_{Tm} (p,\ A_1 A_2 \ldots A_n A, n)$$

Если две конфигурации связаны отношением $\vdash_{Tm}$, то мы говорим, что вторая получается из первой за один шаг. Если вторая получается из первой за конечное, включая ноль, число шагов, то такое отношение будем обозначать $\vdash_{Tm*}$.

Язык, допускаемый Tm, это множество таких слов из T^*, которые будучи расположены в левом конце ленты переводят Tm из начального состояния q_0 с начальным положением головки в самом левом конце ленты в конечное состояние. Формально, язык, допускаемй Tm, это

$$L = \{w \mid w \in \Sigma^* \text{ и } (q_0, w, 1) \vdash_{Tm*} (q, u, i) \text{ для некоторых } q \in F, u \in \Gamma^* \text{ и целого } i\}$$

Если Tm распознает L, то Tm останавливается, то есть не имеет переходов после того, как слово допущено. Однако, если слово не допущено, возможно, что Tm не останавливается.

Язык, допускаемый некоторой Tm, называется рекурсивно перечислимым. Если Tm останавливается на всех входах, то говорят, что Tm задает алгоритм и язык называется рекурсивным.

Существует машина Тьюринга, которая по некоторому описанию произвольной Tm и кодированию слова x моделирует поведение Tm со входом x. Такая машина Тьюринга называется универсальной машиной Тьюринга.

Неразрешимость проблемы останова

Проблема останова для машины Тьюринга формулируется следующим образом: можно ли определить по данной машине Тьюринга в произвольной конфигурации со строкой конечной длины непустых символов на ленте остановится ли она? Говорят, что эта проблема рекурсивно неразрешима, что означает, что не существует алгоритма, который для любой Tm в произвольной конфигурации определял бы остановится ли в конце концов Tm.

Перенумеруем все машины Тьюринга и все возможные входы над алфавитом Σ. Рассмотрим язык

$L_1=\{x_i|x_i$ не допускается $T_i\}$

Ясно, что L_1 не допускается никакой Tm. Допустим, что это не так. Пусть L_1 допускается Tj . Тогда $x_j \in L_1$ тогда и только тогда, когда x_j не допускается T_j. Но поскольку T_j допускает $L_1, x_j \in L_1$ тогда и только тогда, когда допускается T_j, - противоречие. Так что L_1 - не является рекурсивно перечислимым множеством.

Предположим, что мы имеем алгоритм (то есть машину Тьюринга, которая всегда останавливается) для определения, остановится ли машина Тьюринга в данной конфигурации. Тогда следующим образом можно построить машину Тьюринга T, допускающую L_1.

1. Если дано слово x, T перечисляет слова x_1, x_2... пока не будет $x_i = x$.
2. T генерирует кодировку машины Тьюринга T_i.
3. Управление передается гипотетической машине, которая определяет останавливается ли T_i на входе x_i.
4. Если выясняется, что T_i не останавливается на входе x_i, то T останавливается и допускает x_i.
5. Если выясняется, что T_i останавливается на входе x_i, то управление передается универсальной машине Тьюринга, которая моделирует T_i на входе x_i. Поскольку T_i в конце концов останавливается, универсальная машина Тьюринга в конце концов останавливается и определяет допускает ли T_i слово x_i.

В любом случае T останавливается, допуская x_i в том случае, когда T_i отвергает x_i, и отвергая x_i, когда T_i допускает x_i.

Таким образом, из нашего предположения, что существует машина Тьюринга, которая определяет, останавливается ли произвольная машина Тьюринга, следует, что L_1 допускается некоторой машиной Тьюринга, а это противоречит доказанному выше. Это, в свою очередь, дает теорему:

Теорема 2.2. Не существует алгоритма для определения того, остановится ли произвольная машина Тьюринга в произвольной конфигурации.

Класс рекурсивных множеств

Теперь можно показать, что класс рекурсивных множеств является собственным подклассом класса рекурсивно перечислимых множеств. То есть существует множество, слова которого могут быть распознаны машиной Тьюринга, которая не останавливается на некоторых словах, не принадлежащих множеству, но не может быть распознано никакой машиной Тьюринга, которая останавливается на всех словах. Примером такого множества является дополнение к L_1.

Лемма 2.1. Если множество рекурсивно, то и его дополнение рекурсивно.

Доказательство. Если L - рекурсивное множество, $L \subseteq T^*$, то существует Tm допускающая L и гарантированно останавливающаяся. Можно считать, что после допуска Tm не делает больше шагов. Построим Tm_1 по Tm, добавив новое состояние q как единственное допускающее состояние Tm_1. Правила Tm_1 включают все правила Tm, так что Tm_1 симулирует Tm. Кроме того, для каждой пары, составленной из недопускающего состояния и ленточного символа Tm, для которых у Tm переход не определен, Tm_1 переходит в состояние q и затем останавливается.

Таким образом Tm_1 симулирует Tm вплоть до остановки. Если Tm останавливается в одном из допускающих состояний, Tm_1 останавливается без допуска. Если Tm останавливается в одном из недопускающих состояний, значит не допускает вход. Тогда Tm_1 делает еще один переход в состояние q и допускает. Ясно, что Tm_1 допускает $T^*\backslash L$.

Лемма 2.2. Пусть $x_1, x_2, \ldots$ - нумерация всех слов некоторого конечного алфавита - и T_1, T_2, ... - нумерация всех машин Тьюринга с ленточными символами, выбранными из некоторого конечного алфавита, включающего -. Пусть $L_2 = \{x_i \mid x_i$ допускается $T_i\}$. L_2 - рекурсивно перечислимое множество, дополнение которого не рекурсивно перечислимо.

Доказательство. Слова L_2 допускаются некоторой Tm, работающей следующим образом. Отметим, что Tm не обязательно останавливается на словах не из L_2.

1. Если дано x,Tm перечисляет предложения x_1, x_2,... пока не найдет $x_i = x$, определяя тем самым, что x - это i-е слово в перечислении.
2. Tm генерирует Tm_i и передает управление универсальной машине Тьюринга, которая симулирует Tm_i со входом x.
3. Если Tm_i останавливается со входом x_i и допускает, Tm останавливается и допускает; если Tm_i останавливается и отвергает x_i, то Tm останавливается и отвергает x_i. Наконец, если Tm_i не останавливается, то Tm не останавливается.
4. Таким образом L_2 рекурсивно перечислимо, поскольку L_2 - это множество допускаемое Tm. Но дополнение к $L_2 (\sim L_2)$ не может быть рекурсивно перечислимо, поскольку если T_j - машина Тьюринга, допускающая $\sim L_2$, то $x_j \in \sim L_2$ тогда и только тогда, когда x_j не допускается Tm_j. Это противоречит утверждению, что $\sim L_2$ - это язык, допускаемый T_j.

Теорема 2.3. Существует рекурсивно перечислимое множество, не являющееся рекурсивным.

Доказательство. Доказательство. По лемме 2.2 L_2 - рекурсивно перечислимое множество, дополнение которого не рекурсивно перечислимо. Если бы L_2 было рекурсивно, по лемме $1 \sim L_2$ было бы рекурсивно, и следовательно рекурсивно перечислимо, что противоречит второй половине леммы 2.2.

Связь машин Тьюринга и грамматик типа 0

Докажем, что язык распознается машиной Тьюринга тогда и только тогда, когда он генерируется грамматикой типа 0. Для доказательства части "если" мы построим недетерминированную машину Тьюринга, которая будет Связь машин Тьюринга и грамматик типа 0 35 недетерминированно выбирать выводы в грамматике и смотреть, является ли вывод входом. Если да, машина допускает вход.

Для доказательства части "только если" мы построим грамматику, которая недетерминированно генерирует представления терминальной строки и затем симулирует машину Тьюринга на этой строке. Если строка допускается некоторой Tm, строка конвертируется в терминальные символы, которые она представляет.

Теорема 2.4. Если L генерируется грамматикой типа 0, то L распознается машиной Тьюринга.

Доказательство. Пусть $G = (N, \Sigma, P, S)$ - грамматика типа 0 и L = L(G). Опишем неформально недетерминированную машину Тьюринга Tm, допускающую L.

$Tm = (Q, \Sigma, \Gamma, D, q_0, F)$

где $\Gamma = \mathrm{T} \cup \Sigma \cup \{B, \#, X\}$

Предполагается, что последние три символа не входят в $N \cup \Sigma$.

Вначале Tm содержит на входной ленте $w \in \Sigma^*$. Tm вставляет # перед w, сдвигая все символы w на одну ячейку вправо, и #S# после w, так что содержимым ленты становится #w#S#.

Теперь Tm недетерминированно симулирует вывод в G, начиная с S. Каждая сентенциальная форма вывода появляется по порядку между последними двумя #. Если некоторый выбор переходов ведет к терминальной строке, она сравнивается с w. Если они совпадают, Tm допускает.

Формально, пусть Tm имеет на ленте #w#$A_1A_2...A_k$#. Tm передвигает недетерминированно головку по $A_1A_2...A_k$, выбирая позицию i и константу r между 1 и максимальной длиной левой части любого правила вывода в P. Затем Tm проверяет подстроки $A_iA_{i+1}...A_{i+r-1}$. Если $A_iA_{i+1}...A_{i+r-1}$ - левая часть некоторого правила вывода из P, она может быть заменена на правую часть. Tm может сдвинуть $A_{i+r}A_{i+r+1}...A_k$# либо влево, либо вправо, освобождая или заполняя место, если правая часть имеет длину, отличную от r.

Из этой простой симуляции выводов в G видно, что Tm печатает на ленте строку вида $\#w\#y\#, y \in V^*$ в точности, если $S \Rightarrow_{G^*} y$. Если y = w, Tm допускает L.

Теорема 2.5. Если L распознается некоторой машиной Тьюринга,то L генерируется грамматикой типа 0.

Доказательство. Пусть $Tm = (Q, \Sigma, \Gamma, D, q_0, F)$ допускает L. Построим грамматику G, которая недерминированно генерирует две копии представления некоторого слова из Σ^* и затем симулирует поведение Tm на одной из копий. Если Tm допускает слово, то G трансформирует вторую копию в терминальную строку. Если Tm не допускает L, то вывод никогда не приводит к терминальной строке.

Формально, пусть

$G = (N, \Sigma, P, A_1)$, где $N = ([\Sigma \cup \{e\}] \times \Gamma) \cup Q \cup \{A_1, A_2, A_3\}$

Продукции таковы:

1. $A_1 \rightarrow q_0 A_2$
2. $A_2 \rightarrow [a, a]A_2$ для каждого $a \in \Sigma$
3. $A_2 \rightarrow A_3$
4. $A_3 \rightarrow [e, B]A_3$
5. $A_3 \rightarrow e$
6. $q[a, C] \rightarrow [a, E]p$ для каждого $a \in \Sigma \cup \{e\}$ и каждого $q \in Q$ и $C \in \Gamma$ такого, что $D(q, C) = (p, E, R)$
7. $[b, I]q[a, C] \rightarrow p[b, I][a, J]$ для каждого C, J, I из Γ, a и b
8. $[a, C]q \rightarrow qaq,\ q[a, C] \rightarrow qaq,\ q \rightarrow e$ для каждого $a \in \Sigma \cup \{e\}, C \in \Gamma, q \in F.$

Используя правила 1 и 2

$A_1 \Rightarrow^* q_0[a_1, a_1][a_2, a_2] \ldots [a_n, a_n]A_2$

где $a_i \in \Sigma$ для некоторого i

Предположим, что Tm допускает строку $a_1a_2 ... a_n$. Тогда для некоторого m Tm использует не более, чем m ячеек справа от входа. Используя правило 3, затем правило 4 m раз, и наконец, правило 5, имеем

$$A_1 \Rightarrow^* q_0[a_1, a_1][a_2, a_2] \dots [a_n, a_n][e, B]^m$$

Начиная с этого момента могут быть использованы только правила 6 и 7, пока не сгенерируется допускающее состояние. Отметим, что первые компоненты ленточных символов в $(\Sigma \cup \{e\}) \times \Gamma$ никогда не меняются. Индукцией по числу шагов Tm можно показать, что если

$(q_0, a_1 a_2 \dots a_n, 1) \vdash_{Tm*} (q, X_1 X_2 \dots X_S, r)$, то

$$q_0[a_1, a_1][a_2, a_2] \dots [a_n, a_n][e, B]^m \Rightarrow_{G*}$$

$$\Rightarrow_{G*} [a_1, X_1][a_2, X_2] \dots [a_{r-1}, X_{r-1}] q [a_r, X_r] \dots [a_{n+m}, X_{n+m}]$$

где $a_1, a_2, \dots a_n$ принадлежат Σ, $a_{n+1} = a_{n+2} = \dots a_{n+m} = e$,

$X_1 X_2, \dots X_{n+m}$ принадлежат Γ и $X_{S+1}=X_{S+2}=\dots X_{n+m}=B$

Предположение индукции тривиально для нуля шагов. Предположим, что оно справедливо для k - 1 шагов. Пусть

$$(q_0, a_1 a_2 \dots a_n, 1) \vdash_{Tm*} \vdash_{Tm*} (q_1, X_1 X_2 \dots X_r, j_1) \vdash_{Tm*} \vdash_{Tm*} (q_2, Y_1 Y_2 \dots Y_S, j_2)$$

за k шагов. По предположению индукции

$$q_0[a_1, a_1][a_2, a_2] \dots [a_n, a_n][e, B]^m \to_{G*} \Rightarrow_{G*} [a_1, X_1][a_2, X_2] \dots [a_{r-1}, X_{r-1}] q_1 [a_{j1}, X_{j1}] \dots \dots [a_{n+m}, X_{n+m}]$$

Пусть E = L, если $j_2 = j_1 - 1$ и E = R, если $j_2 = j_1 + 1$. В этом случае $D(q_1, X_{j1}) = (q_2, Y_{j1}, E)$.

По правилам 6 или 7

$$q_1[a_{j1}, X_{j1}] \to [a_{j1}, Y_{j1}] q_2 \text{ или}$$

$$[a_{j1-1}, X_{j1-1}] q_1 [a_{j1}, X_{j1}] \to q_2 [a_{j1-1}, X_{j1-1}][a_{j1}, Y_{j1}]$$

в зависимости от того, равно ли E значению R или L.

Теперь $X_i = Y_i$ для всех $i \neq j_1$.

Таким образом,

$$q_0[a_1, a_1][a_2, a_2] \dots [a_n, a_n][e, B]^m \Rightarrow^{G*} \Rightarrow^{G*} [a_1, Y_1] q_2 [a_{j2}, Y_{j2}] \dots [a_{n+m}, Y_{n+m}]$$

что доказывает предположение индукции.

По правилу 8, если $q \in F$, легко показать, что

$$[a_1, X_1] \ldots q[a_j, X_j] \ldots [a_{n+m}, X_{n+m}] \Rightarrow^* a_1 a_2 \ldots a_n.$$

Таким образом, G может генерировать $a_1a_2 \ldots a_n$, если $a_1a_2 \ldots a_n$ допускается Tm. Таким образом, L(G) включает все слова, допускаемые Tm. Для завершения доказательства необходимо показать, что все слова из L(G) допускаются Tm. Индукцией доказывается, что $A_1 \Rightarrow_{G*} w$ только если w допускается Tm.

Линейно-ограниченные автоматы и их связь с контекстно-зависимыми грамматиками

Каждый КЗ-язык является рекурсивным, но обратное не верно. Покажем что существует алгоритм, позволяющий для произвольного КЗ-языка L в алфавите T, и произвольной цепочки $w \in T^*$ определить, принадлежит ли w языку L.

Теорема 2.6. Каждый контекстно-зависимый язык является рекурсивным языком.

Доказательство. Пусть L - контекстно-зависимый язык. Тогда существует некоторая неукорачивающая грамматика G = (N, T, P, S), порождающая L.

Пусть $w \in T^*$ и $|w| = n$. Если n = 0, то есть w = e, то принадлежность $w \in L$ проверяется тривиальным образом. Так что будем предполагать, что n > 0.

Определим множество Tm как множество строк $u \in (N \cup T)^+$ длины не более n таких, что вывод $S \Rightarrow^* u$ имеет не более m шагов. Ясно, что $T_0 = \{S\}$.

Легко показать, что T_m можно получить из T_{m-1} просматривая, какие строки с длиной, меньшей или равной n можно вывести из строк из T_{m-1} применением одного правила, то есть

$$T_m = T_{m-1} \cup \{u \mid v \Rightarrow \text{для некоторого } v \in T_{m-1}, \text{ где } |u| \leq n\}$$

Если $S \Rightarrow^* u$ и $|u| \leq n$, то $u \in T_m$ для некоторого m. Если из S не выводится u или |u| > n, то u не принадлежит T_m ни для какого m.

Очевидно, что $T_m \subseteq T_{m-1}$ для всех m > 1. Поскольку T_m зависит только от T_{m-1}, если $Tm = T_{m-1}$, то $T_m = T_{m+1} = T_{m+2} = \ldots$. Процедура будет вычислять $T_1, T_2, T_3, \ldots$ пока для некоторого m не окажется $T_m = T_{m-1}$. Если w не принадлежит T_m, то не принадлежит и L(G), поскольку для j > m выполнено $T_j = T_m$. Если $w \in T_m$, то $S \Rightarrow^* w$.

Покажем, что существует такое m, что $T_m = Tm-1$. Поскольку для каждого i > 1 справедливо $T_i \supseteq T_{i-1}$, то если $T_i \neq T_{i-1}$, то число элементов в T_i по крайней мере на 1 больше, чем в T_{i-1}. Пусть $|N \cup T| = k$. Тогда число строк в $(N \cup T)^+$ длины меньшей или равной n равно $k + k^2 + \ldots + k^n \leq nk^n$. Только эти строки могут быть в любом T_i. Значит, $T_m = T_{m-1}$ для некоторого $m \leq nk^n$. Таким образом, процедура, вычисляющая T_i для всех $i \geq 1$ до тех пор, пока не будут найдены два равных множества, гарантированно заканчивается, значит, это алгоритм.

Линейно-ограниченный автомат (ЛОА) - это недетерминированная машина Тьюринга с одной лентой, которая никогда не выходит за пределы |w| ячеек, где w - вход. Формально, линейно-ограниченный автомат обозначается как $M = (Q, \Sigma, \Gamma, D, q_0, F)$. Обозначения имеют тот же смысл, что и для машин Тьюринга. Q - это множество состояний, $F \subseteq Q$ - множество заключительных состояний, Γ - множество ленточных символов, $\Sigma \subseteq \Gamma$ - множество входных символов, $q_0 \in Q$ - начальное состояние, D- отображение из Q x Γ в подмножество Q x Γ x {L, R}.

Σ содержит два специальных символа, обычно обозначаемых © и $, - левый и правый концевые маркеры, соответственно. Эти символы располагаются сначала по концам входа и их функция - предотвратить переход головки за пределы области, в которой расположен вход.

Конфигурация M и отношение $\vdash M$, связывающее две конфигурации, если вторая может быть получена из первой применением D, определяются так же, как и для машин Тьюринга. Конфигурация M обозначается как

(q,A_1A_2,...,A_n,i) где $q \in Q,\ A_1, A_2, \ldots, A_n \in \Gamma, i$ -целое от 1 до n. Предположим, что $(p, A, L) \in D(q, A_i)$ и i > 1

Будем говорить, что

$$(q, A_1, A_2 \ldots A_n, i) \vdash_M (p, A_1, A_2 \ldots A_{i-1} A A_{i+1} \ldots A_n, i-1)$$

$(p, A, R) \in D(q, A_i)$ и i < n, будем говорить, что

$$(q, A_1, A_2 \ldots A_n, i) \vdash_M (p, A_1, A_2 \ldots A_{i-1} A A_{i+1} \ldots A_n, i+1)$$

То есть M печатает A поверх A_i, меняет состояние на p и передвигает головку влево или вправо, но не за пределы области, в которой символы располагались исходно. Как обычно, определим отношение $\vdash *_M$ как

$(q, \alpha, i) \vdash *_M (q, \alpha, i)$ и

Если $(q_1, \alpha_1, i_1) \vdash *_M (q_2, \alpha_2, i_2)$ и $(q_2, \alpha_2, i_2) \vdash *_M (q_3, \alpha_3, i_3)$,

то $(q_1, \alpha_1, i_1) \vdash *_M (q_3, \alpha_3, i_3)$

Язык, допускаемый M - это $\{w \mid w \in (\Sigma\ \{©, \$\})^*$ и $(q_0, ©w\$, 1) \vdash *(q, \alpha, i)$ для некоторого $q \in F, \alpha \in \Gamma^*$ и целого i}.

Будем называть M детерминированным, если D(q, A) содержит не более одного элемента для любых $q \in Q, A \in \Gamma$. Не известно, совпадает ли класс множеств, допускаемых детерминированными и недетеминированными ЛОА. Ясно, что любое множество, допускаемое недетерминированным ЛОА, допускается некоторой детерминированной МТ. Однако, число ячеек ленты, требуемой этой МТ, может экспоненциально зависеть от длины входа.

Класс множеств, допускаемых ЛОА, в точности совпадает с классом контекстно - зависимых языков.

Теорема 2.7. Если L - контекстно-зависимый язык, то L допускается ЛОА.

Доказательство. Пусть G = (V_N, V_T, P, S) - контекстно- зависимая грамматика. Построим ЛОА M такой, что он допускает язык L(G). Не вдаваясь в детали построения M, поскольку он довольно сложен, рассмотрим схему его работы. В качестве ленточных символов будем рассматривать пары (s_i^1, s_i^2), где $s_i^1 \in \Sigma,\ \Sigma = V_T \cup \{@, \$\},\ s_i^2 \in \Gamma,\ \Gamma = V_T \cup V_N \cup \{B\}$ В начальной конфигурации лента содержит (@, B), (a_1, B), ... (a_n, B), ($, B), где a^1 ... a_n = w - входная цепочка, n=|w|. Цепочку символов $s_1^1 \ldots s_n^1$n будем называть " первым треком ", $s_1^2 \ldots s_n^2$n - " вторым треком ". Первый трек будет содержать входную строку x с концевыми маркерами. Второй трек будет использоваться для вычислений. На первом шаге M помещает символ S в самой левой ячейке второго трека. Затем M выполняет процедуру генерации в соответствии со следующими шагами:

1. Процедура выбирает подстроку символов α из второго трека такую, что $\alpha \to \beta \in P$.

2. Подстрока α заменяется на β, перемещая, если необходимо, вправо символы справа от α. Если эта операция могла бы привести к перемещению символа за правый концевой маркер, ЛОА останавливается.
3. Процедура недетерминированно выбирает перейти на шаг 1 или завершиться.

На выходе из процедуры первый трек все еще содержит строку x, а второй трек содержит строку γ такую, что $S \Rightarrow_G^* \gamma$ ЛОА сравнивает символы первого трека с соответствующими символами второго трека. Если сравнение неуспешно, строки символов первого и второго треков не одинаковы и ЛОА останавливается без допуска. Если строки одинаковы, ЛОА останавливается и допускает.

Если $x \in L(G)$, то существует некоторая последовательность шагов, на которой ЛОА строит x на втором треке и допускает вход. Аналогично, для того, чтобы ЛОА допустил x, должна существовать последовательность шагов такая, что x может быть построен на втором треке. Таким образом, должен быть вывод x из S в G.

Отметим схожесть этих рассуждений и рассуждений в случае произвольной грамматики. Тогда промежуточные сентенциальные формы могли иметь длину, произвольно большую по сравнению с длиной входа. Как следствие, требовалась вся мощь машин Тьюринга. В случае контекстно- зависимых грамматик промежуточные сентенциальные формы не могут быть длиннее входа.

Теорема 2.8. Если L допускается ЛОА, то L - контекстно - зависимый язык.

Доказательство. Конструкция КЗГ по ЛОА аналогична конструкции грамматики типа 0, моделирующей машину Тьюринга. Различие заключатся в том, что нетерминалы КЗГ должны указывать не только текущее и исходное содержимое ячеек ленты ЛОА, но и то, является ли ячейка соседней справа или слева с концевым маркером. Кроме того, состояние ЛОА должно комбинироваться с символом под головкой, поскольку КЗГ не может иметь раздельные символы для концевых маркеров и состояния ЛОА, так как эти символы должны были бы быть заменены на e, когда строка превращается в терминальную.

Теорема 2.9. Существуют рекурсивные множества, не являющиеся контекстно - зависимыми.

Доказательство. Все строки в $\{0,1\}^*$ можно занумеровать. Пусть x_i - i-ое слово. Мы можем занумеровать все грамматики типа 0, терминальными символами которых являются 0 и 1. Поскольку имена переменных не важны и каждая грамматика имеет конечное их число, можно предположить, что существует счетное число переменных.

Представим переменные в двоичной кодировке как 01, 011, 0111, 01111 и т.д. Предположим, что 01 всегда является стартовым символом. Кроме того, в этой кодировке терминал 0 будет представляться как 00, а терминал 1 как 001. Символ « $\rightarrow$» представляется как 0011, а запятая как 00111. Любая грамматика с терминалами 0 и 1 может быть представлена строкой правил, использующей стрелку (0011) для разделения левой и правой частей, и запятой (00111) для разделения правил. Строки, представляющие символы, используемые в правилах, - это 00, 001 и 01^i для i = 1, 2, ... Множество используемых переменных определяется неявно правилами.

Отметим, что не все строки из 0 и 1 представляют грамматики, и не обязательно КЗГ. Однако, по данной строке легко можно сказать, представляет ли она КЗГ. i- ю грамматику можно найти, генерируя двоичные строки в описанном порядке пока не сгенерируется i-я строка, являющаяся КЗГ. Поскольку имеется бесконечное число КЗГ, их можно занумеровать в некотором порядке G_1, G_2, ...

Определим $L = \{x_i | x_i \notin L(G_i)\}$. L рекурсивно. По строке x_i легко можно определить i и затем определить G_i. По теореме 2.6. имеется алгоритм, определяющий для x_i принадлежит ли он $L(G_i)$, поскольку G_i КЗГ. Таким образом имеется алгоритм для определения для любого x принадлежит ли он G.

Покажем теперь, что L не генерируется никакой КЗ-грамматикой. Предположим, что L генерируется КЗ-грамматикой G_i. Во-первых, предположим, что $x_i \in L$. Поскольку $L(G_i) = L, x_i \in L(G_i)$. Но тогда по определению xi $\notin L(G_i)$ - противоречие. Таким образом предположим, что $x_i \notin L$. Поскольку $L(G_i) = L$, x_i

$\notin L(G_i)$. Но тогда по определению $x_i \in L(G_i)$ - снова противоречие. Из чего можно заключить, что L не генерируется G_i. Поскольку приведенный выше аргумент справедлив для каждой КЗ- грамматики G_i в перечислении, и поскольку перечисление содержит все КЗ-грамматики, можно заключить, что L не КЗ-язык. Поэтому L - рекурсивное множество, не являющееся контекстно-зависимым.

3. Лекция: Лексический анализ

В данной лекции приводится понятие лексического анализа. Рассмотрены основные задачи лексического анализа, приведены основные определения, такие как регулярное множество, конечный автомат, конфигурация, лексический анализатор. Также приведены примеры решения задач, связанных с лексическим анализом.

Основная задача лексического анализа - разбить входной текст, состоящий из последовательности одиночных символов, на последовательность слов, или лексем, то есть выделить эти слова из непрерывной последовательности символов. Все символы входной последовательности с этой точки зрения разделяются на символы, принадлежащие каким-либо лексемам, и символы, разделяющие лексемы (разделители). В некоторых случаях между лексемами может и не быть разделителей. С другой стороны, в некоторых языках лексемы могут содержать незначащие символы (например, символ пробела в Фортране). В Си разделительное значение символов-разделителей может блокироваться ("\" в конце строки внутри "...").

Обычно все лексемы делятся на классы. Примерами таких классов являются числа (целые, восьмеричные, шестнадцатиричные, действительные и т.д.), идентификаторы, строки. Отдельно выделяются ключевые слова и символы пунктуации (иногда их называют символы-ограничители). Как правило, ключевые слова - это некоторое конечное подмножество идентификаторов. В некоторых языках (например, ПЛ/1) смысл лексемы может зависеть от ее контекста и невозможно провести лексический анализ в отрыве от синтаксического.

Для осуществления двух дальнейших фаз анализа лексический анализатор выдает информацию двух типов: для синтаксического анализатора, работающего вслед за лексическим, существенна информация о последовательности классов лексем, ограничителей и ключевых слов, а для контекстного анализатора, работающего вслед за синтаксическим, существенна информация о конкретных значениях отдельных лексем (идентификаторов, чисел и т.д.).

Таким образом, общая схема работы лексического анализатора такова. Сначала выделяется отдельная лексема (при этом, возможно, используются символы- разделители). Ключевые слова распознаются явным выделением непосредственно из текста, либо сначала выделяется идентификатор, а затем делается проверка на принадлежность его множеству ключевых слов.

Если выделенная лексема является ограничителем, то этот ограничитель (точнее, некоторый его признак) выдается как результат лексического анализа. Если выделенная лексема является ключевым словом, то выдается признак соответствующего ключевого слова. Если выделенная лексема является идентификатором - выдается признак идентификатора, а сам идентификатор сохраняется отдельно. Наконец, если выделенная лексема принадлежит какому-либо из других классов лексем (например, лексема представляет собой число, строку и т.д.), то выдается признак соответствующего класса, а значение лексемы сохраняется отдельно.

Лексический анализатор может быть как самостоятельной фазой трансляции, так и подпрограммой, работающей по принципу "дай лексему". В первом случае (рис. 3.1, а) выходом анализатора является файл лексем, во втором - (рис. 3.1., б) лексема выдается при каждом обращении к анализатору (при этом, как правило, признак класса лексемы возвращается как результат функции "лексический анализатор", а значение лексемы передается через глобальную переменную). С точки зрения обработки значений лексем, анализатор может либо просто выдавать значение каждой лексемы, при этом построение таблиц объектов (идентификаторов, строк, чисел и т.д.) переносится на более поздние фазы, либо он может самостоятельно строить таблицы объектов. В этом случае в качестве значения лексемы выдается указатель на вход в соответствующую таблицу.

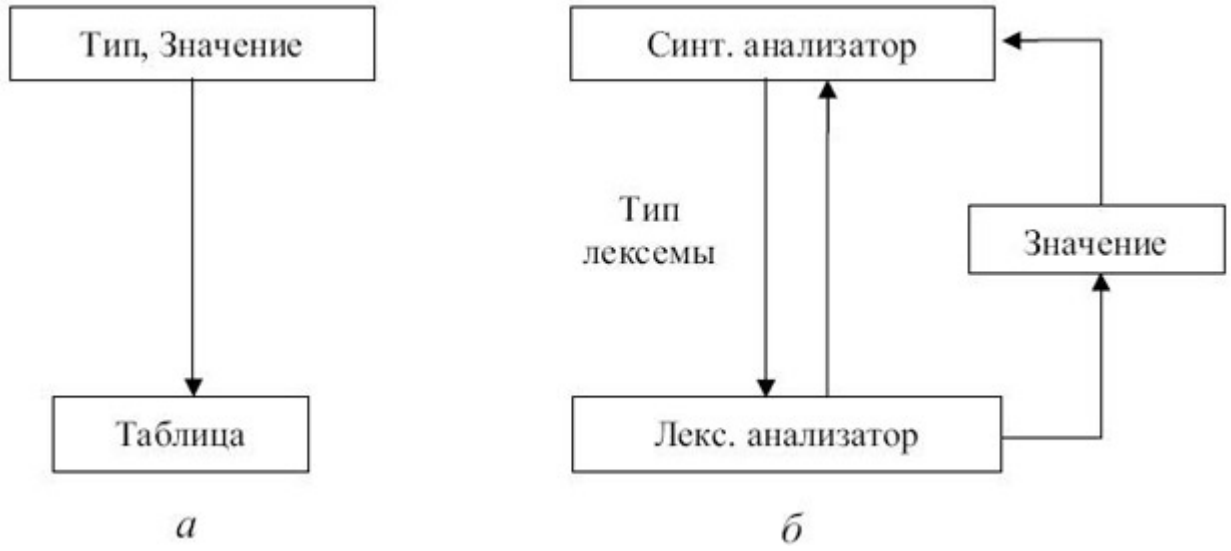


Рис. 3.1.

Работа лексического анализатора задается некоторым конечным автоматом. Однако, непосредственное описание конечного автомата неудобно с практической точки зрения. Поэтому для задания лексического анализатора, как правило, используется либо регулярное выражение, либо праволинейная грамматика. Все три формализма (конечных автоматов, регулярных выражений и праволинейных грамматик) имеют одинаковую выразительную мощность. В частности, по регулярному выражению или праволинейной грамматике можно сконструировать конечный автомат, распознающий тот же язык.

Регулярные множества и выражения

Введем понятие регулярного множества, играющего важную роль в теории формальных языков.

Регулярное множество в алфавите T определяется рекурсивно следующим образом:

1. ∅(пустое множество) - регулярное множество в алфавите T;
2. {e} - регулярное множество в алфавите T (e - пустая цепочка);
3. {a} - регулярное множество в алфавите T для каждого $a \in T$;
4. если P и Q - регулярные множества в алфавите T, то регулярными являются и множества
    1. $P \cup Q$ (объединение),
    2. PQ (конкатенация, то есть множество $\{pq \mid p \in P,\ q \in Q\}$)
    3. P^* (итерация: $P^* = \bigcup_{n=0}^{\infty} P^n$);
5. ничто другое не является регулярным множеством в алфавите T.

Итак, множество в алфавите T регулярно тогда и только тогда, когда оно либо ∅, либо {e}, либо {a} для некоторого $a \in T$, либо его можно получить из этих множеств применением конечного числа операций объединения, конкатенации и итерации.

Приведенное выше определение регулярного множества позволяет ввести следующую удобную форму его записи, называемую регулярным выражением.

Регулярное выражение в алфавите T и обозначаемое им регулярное множество в алфавите T определяются рекурсивно следующим образом:

1. ∅регулярное выражение, обозначающее регулярное множество ∅;
2. {e} - регулярное выражение, обозначающее регулярное множество {e};
3. {a} - регулярное выражение, обозначающее регулярное множество {a};
4. если p и q - регулярные выражения, обозначающие регулярные множества P и Q соответственно, то

   1. (p|q) - регулярное выражение, обозначающее регулярное множество $P \cup Q$,
   2. (pq) - регулярное выражение, обозначающее регулярное множество PQ,
   3. (p^*) - регулярное выражение, обозначающее регулярное множество P^*;
5. ничто другое не является регулярным выражением в алфавите T.

Мы будем опускать лишние скобки в регулярных выражениях, договорившись о том, что операция итерации имеет наивысший приоритет, затем идет операции конкатенации, наконец, операция объединения имеет наименьший приоритет.

Кроме того, мы будем пользоваться записью p^+ для обозначения pp^*. Таким образом, запись (a|((ba)(a^*))) эквивалентна a|ba^+.

Также, мы будем использовать запись L(r) для регулярного множества, обозначаемого регулярным выражением r.

Пример 3.1. Несколько примеров регулярных выражений и обозначаемых ими регулярных множеств:

1. a(e|a)|b - обозначает множество {a; b; aa};
2. a(a|b)* - обозначает множество всевозможных цепочек, состоящих из a и b, начинающихся с a;
3. (a|b)*(a|b)(a|b)* - обозначает множество всех непустых цепочек, состоящих из a и b, то есть множество {a, b}$^+$;
4. ((0|1)(0|1)(0|1))* - обозначает множество всех цепочек, состоящих из нулей и единиц, длины которых делятся на 3.

Ясно, что для каждого регулярного множества можно найти регулярное выражение, обозначающее это множество, и наоборот. Более того, для каждого регулярного множества существует бесконечно много обозначающих его регулярных выражений.

Будем говорить, что регулярные выражения равны или эквивалентны (=), если они обозначают одно и то же регулярное множество.

Существуют алгебраические законы, позволяющие осуществлять эквивалентное преобразование регулярных выражений.

Лемма. Пусть p, q и r - регулярные выражения. Тогда справедливы следующие соотношения:

1. p|q = q|p;
2. $\varnothing^*$ = e;
3. p|(q|r) = (p|q)|r;
4. p(qr) = (pq)r;
5. p(q|r) = pq|pr;
6. (p|q)r = pr|qr;
7. pe = ep = p;
8. $\varnothing$p = p$\varnothing$ = $\varnothing$;
9. p^* = p|p^*;
10. (p^*)* = p^*;
11. p|p = p;
12. p|$\varnothing$ = p;

Следствие. Для любого регулярного выражения существует эквивалентное регулярное выражение, которое либо есть $\varnothing$, либо не содержит в своей записи $\varnothing$.

В дальнейшем будем рассматривать только регулярные выражения, не содержащие в своей записи $\varnothing$. При практическом описании лексических структур бывает полезно сопоставлять регулярным выражениям некоторые имена, и ссылаться на них по этим именам. Для определения таких имен мы будем использовать запись вида

$$d_1 = r_1$$
$$d_2 = r_2$$
$$\dots$$
$$d_n = r_n$$

где d_i - различные имена, а каждое r_i - регулярное выражение над символами $T \cup \{d_1, d_2, \dots, d_{i-1}\}$, то есть символами основного алфавита и ранее определенными символами (именами). Таким образом, для любого r_i можно построить регулярное выражение над T, повторно заменяя имена регулярных выражений на обозначаемые ими регулярные выражения.

Пример 3.2. Несколько примеров использования имен для обозначения регулярных выражений.

1. Регулярное выражение для множества идентификаторов.
$$Letter = a \mid b \mid c \mid \dots \mid x \mid y \mid z$$
$$Digit = 0 \mid 1 \mid \dots \mid 9$$
$$Identifier = Letter(Letter \mid Digit)^*$$
2. Регулярное выражение для множества чисел в десятичной записи.
$$Digit = 0 \mid 1 \mid \dots \mid 9$$
$$Integer = Digit^+$$
$$Fraction = .Integer \mid e$$
$$Exponent = (E(+ \mid - \mid e)Integer) \mid e$$
$$Number = Integer\ Fraction\ Exponent$$

Конечные автоматы

Регулярные выражения, введенные ранее, служат для описания регулярных множеств. Для распознавания регулярных множеств служат конечные автоматы. Недетерминированный конечный автомат (НКА) - по определению есть пятерка M = (Q, T, D, q_0, F), где

1. Q - конечное множество состояний,
2. T - конечное множество допустимых входных символов (входной алфавит),
3. D - функция переходов (отображающая множество $Q \times (T \cup \{e\})$ во множество подмножеств множества Q), определяющая поведение управляющего устройства,
4. $q_0 \in Q$ - начальное состояние управляющего устройства,
5. $F \subseteq Q$ - множество заключительных состояний.

Работа конечного автомата представляет собой некоторую последовательность шагов, или тактов. Такт определяется текущим состоянием управляющего устройства и входным символом, обозреваемым в данный момент входной головкой. Сам шаг состоит из изменения состояния и, возможно, сдвига входной головки на одну ячейку вправо (рис. 3.2.).

Недетерминизм автомата заключается в том, что, во- первых, находясь в некотором состоянии и обозревая текущий символ, автомат может перейти в одно из, вообще говоря, нескольких возможных состояний, и во-вторых, автомат может делать переходы по e.

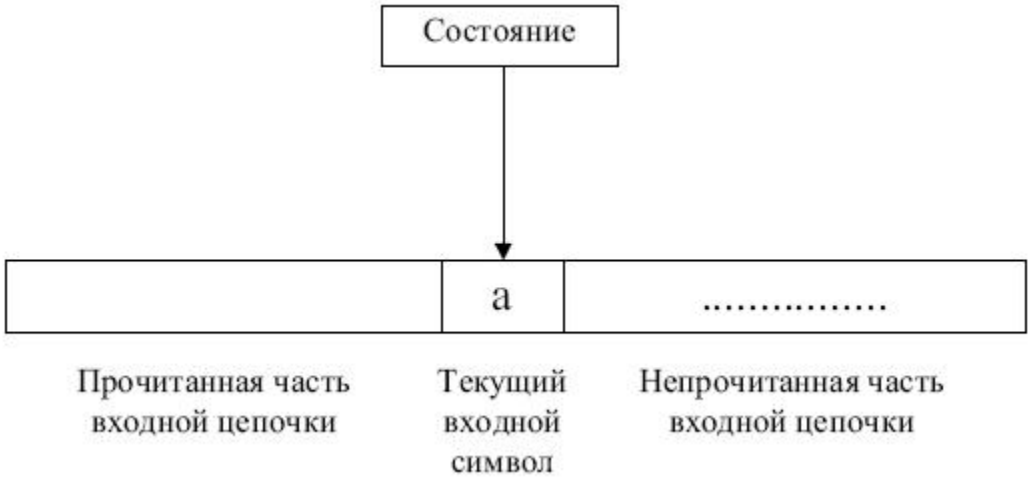


Рис. 3.2.

Пусть M = (Q, T, D, q_0, F) - НКА. Конфигурацией автомата M называется пара $(q, w) \in Q \times T^*$, где q - текущее состояние управляющего устройства, а w - цепочка символов на входной ленте, состоящая из символа под головкой и всех символов справа от него. Конфигурация (q_0, w) называется начальной, а конфигурация (q, e), где $q \in F$ - заключительной (или допускающей). Тактом автомата M называется бинарное отношение $\vdash$, определенное на конфигурациях M следующим образом: если $p \in D(q, a)$, где $a2 \in T \cup \{e\}$, то $(q, aw) \vdash (p, w)$ для всех $w2 \in T^*$.

Будем обозначать символом $^{-+}$ $(\vdash^*)$ транзитивное (реф- лексивно-транзитивное) замыкание отношения $\vdash$. Будем говорить, что автомат M допускает цепочку w, если $(q_0, w) \vdash^* (q, e)$ для некоторого $q \in F$. Языком, допускаемым, (распознаваемым, определяемым) автоматом M, (обозначается L(M)), называется множество входных цепочек, допускаемых автоматом M. То есть,

$$L(M) = \{w \mid w \in T^* \text{ и } (q_0, w) \vdash^* (q, e) \text{ для некоторого } q \in F\}$$

Важным частным случаем недетерминированного конечного автомата является детерминированный конечный автомат, который на каждом такте работы имеет возможность перейти не более чем в одно состояние и не может делать переходы по e.

Пусть M = (Q, T, D, q_0, F) - НКА. Будем называть M детерминированным конечным автоматом (ДКА), если выполнены следующие два условия:

1. D(q, e) = $\varnothing$, для любого $q \in Q$, и
2. D(q, a) содержит не более одного элемента для любых $q \in Q$ и $a \in T$.

Так как функция переходов ДКА содержит не более одного элемента для любой пары аргументов, для ДКА мы будем пользоваться записью D(q, a)=p вместо D(q, a)={p}.

Конечный автомат может быть изображен графически в виде диаграммы, представляющей собой ориентированный граф, в котором каждому состоянию соответствует вершина, а дуга, помеченная символом $a \in T \cup \{e\}$, соединяет две вершины p и q, если $p \in D(q, a)$. На диаграмме выделяются начальное и заключительные состояния (в примерах ниже, соответственно, входящей стрелкой и двойным контуром).

Пример 3.3. Пусть L = L(r), где r = $(a|b)^*a(a|b)(a|b)$.

1.
    1. Недетерминированный конечный автомат M, допускающий язык L:

M = {{1, 2, 3, 4}, {a, b}, D, 1, {4}},

где функция переходов D определяется так:

$$D(1,a) = \{1,2\}, \qquad D(3,a) = \{4\},$$
$$D(1,b) = \{1\}, \qquad D(2,b) = \{3\},$$
$$D(2,a) = \{3\}, \qquad D(3,b) = \{4\}.$$

Диаграмма автомата приведена на рис. 3.3 а.

2. Детерминированный конечный автомат M, допускающий язык L:

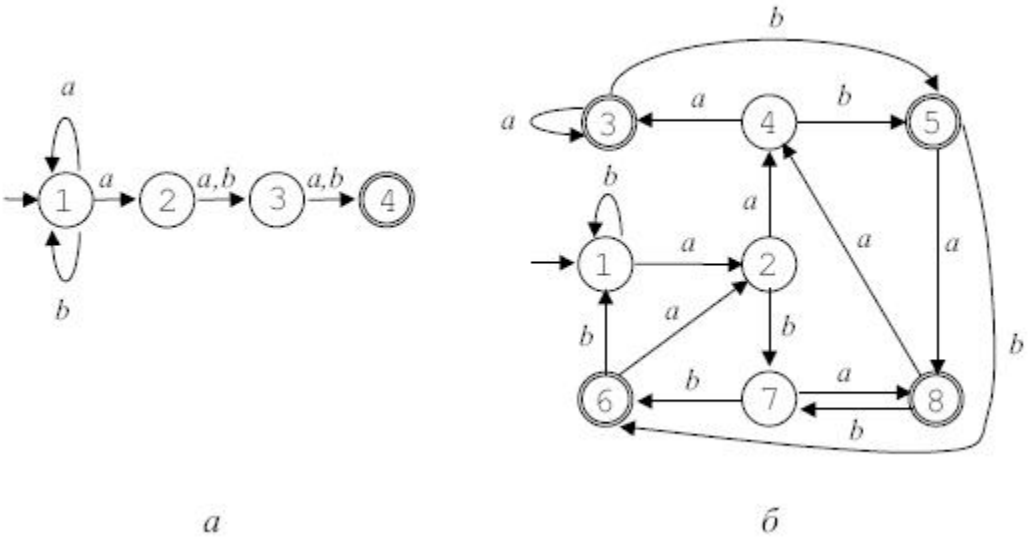
M = {{1, 2, 3, 4, 5, 6, 7, 8}, {a, b}, D, 1, {3, 5, 6, 8}}

где функция переходов D определяется так:

$$D(1,a) - 2, \qquad D(5,a) - 8,$$
$$D(1,b) - 1, \qquad D(5,b) - 6,$$
$$D(2,a) - 4, \qquad D(6,a) - 2,$$
$$D(2,b) = 7, \qquad D(6,b) = 1,$$
$$D(3,a) - 3, \qquad D(7,a) - 8,$$
$$D(3,b) = 5, \qquad D(7,b) = 6,$$
$$D(4,a) - 3, \qquad D(8,a) - 4,$$
$$D(4,b) = 5, \qquad D(8,b) = 7.$$

Диаграмма автомата приведена на рис. 3.3 б.

2.



а *б*

Рис. 3.3.

Пример 3.4. Диаграмма автомата, допускающего множество чисел в десятичной записи, приведена на рис. 3.4.

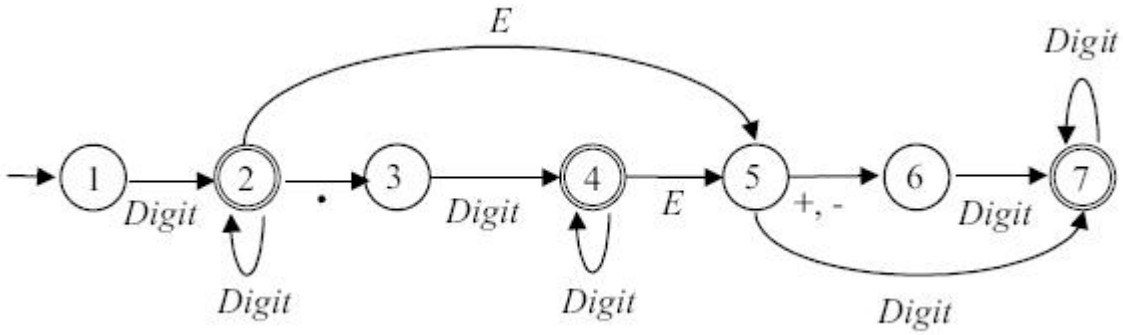


Рис. 3.4.

Пример 3.5. Анализ цепочек.

1. При анализе цепочки w = ababa автомат из примера рис. 3.3, а, может сделать следующую последовательность тактов:

   $$(1, ababa) \vdash (1, baba) \vdash (1, aba) \vdash (2, ba) \vdash (3, a) \vdash (4, e).$$

   Состояние 4 является заключительным, отсюда, цепочка w допускается этим автоматом.

2. При анализе цепочки w = ababab автомат из примера рис. 3.3, б, должен сделать следующую последовательность тактов:

   $$(1, ababab) \vdash (2, babab) \vdash (7, abab) \vdash (8, bab) \vdash (7, ab) \vdash (8, b) \vdash (7, e).$$

   Так как состояние 7 не является заключительным, цепочка w не допускается этим автоматом.

Алгоритмы построения конечных автоматов

Построение недетерминированного конечного автомата по регулярному выражению

Рассмотрим алгоритм построения по регулярному выражению недетерминированного конечного автомата, допускающего тот же язык.

Алгоритм 3.1. Построение недетерминированного конечного автомата по регулярному выражению.

Вход. Регулярное выражение r в алфавите T.

Выход. НКА M, такой что L(M) = L(r).

Метод. Автомат для выражения строится композицией из автоматов, соответствующих подвыражениям. На каждом шаге построения строящийся автомат имеет в точности одно заключительное состояние, в начальное состояние нет переходов из других состояний и нет переходов из заключительного состояния в другие.

1. Для выражения e строится автомат

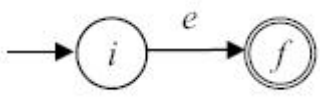


Рис. 3.5.

2. Для выражения $a(a \in T)$ строится автомат
3. Пусть M(s) и M(t) - НКА для регулярных выражений s и t соответственно.

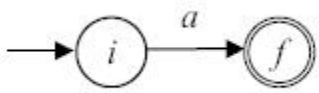


Рис. 3.6.

1. Для выражения s|t автомат M(s|t) строится как показано на рис. 3.7. Здесь i - новое начальное состояние и f - новое заключительное состояние. Заметим, что имеет место переход по e из i в начальные состояния M(s) и M(t) и переход по e из заключительных состояний M(s) и M(t) в f. Начальное и заключительное состояния автоматов M(s) и M(t) не являются таковыми для автомата M(s|t).

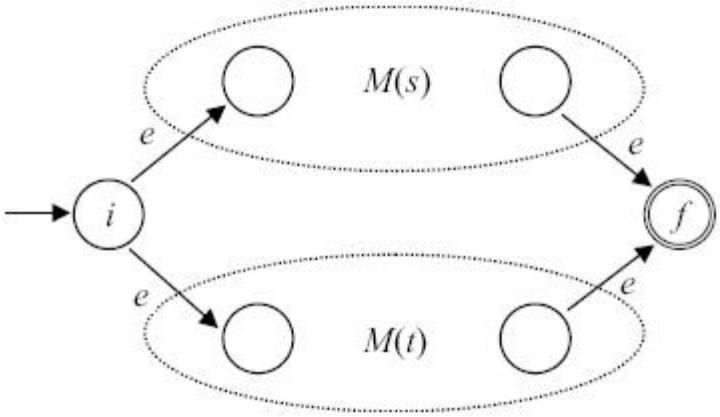


Рис. 3.7.

2. Для выражения st автомат M(st) строится следующим образом:

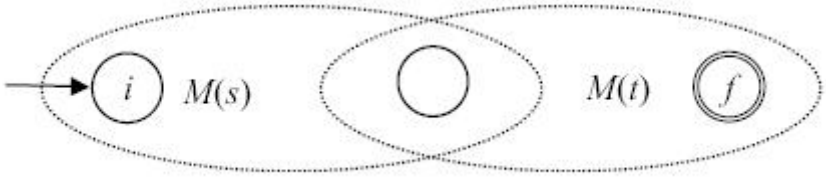


Рис. 3.8.

Начальное состояние автомата M(s) становится начальным для нового автомата, а заключительное состояние M(t) становится заключительным для нового автомата. Начальное состояние M(t) и заключительное состояние M(s) сливаются, то есть все переходы из начального состояния M(t) становятся переходами из заключительного состояния M(s). В новом автомате это объединенное состояние не является ни начальным, ни заключительным.

3. Для выражения s^* автомат M(s^*) строится следующим образом:

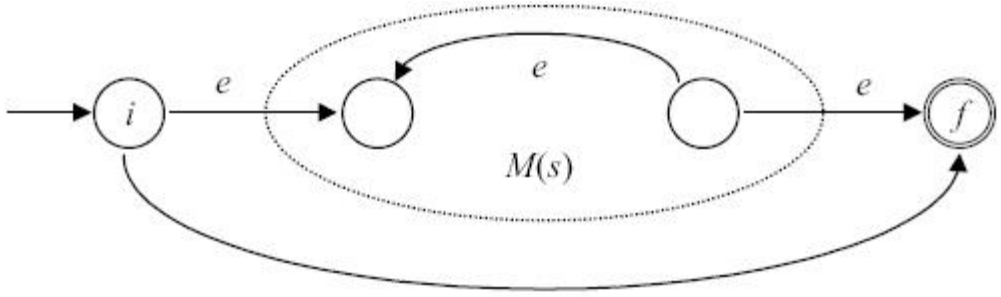


Рис. 3.9.

Здесь i - новое начальное состояние, а f - новое заключительное состояние.

Построение детерминированного конечного автомата по недетерминированному

Рассмотрим алгоритм построения по недетерминированному конечному автомату детерминированного конечного автомата, допускающего тот же язык.

Алгоритм 3.2. Построение детерминированного конечного автомата по недетерминированному.

Вход. НКА M = (Q, T, D, q_0, F).

Выход. ДКА $M' = (Q', T, D', q'_0, F')$ такой что $L(M) = L(M')$.

Метод. Каждое состояние результирующего ДКА - это некоторое множество состояний исходного НКА.

В алгоритме будут использоваться следующие функции: $\text{e-closure}(R)\ (R \subseteq Q)$ - множество состояний НКА, достижимых из состояний, входящих в R, посредством только переходов по e, то есть множество

$$S = \bigcup_{q \in R} \{p \mid (q, e) \vdash^* (p, e)\}$$

$move(R, a)(R \subseteq Q)$ - множество состояний НКА, в которые есть переход на входе a для состояний из R, то есть множество

$$S = \bigcup_{q \in R} \{p \mid p \in D(q, a)\}$$

Вначале Q' и D' пусты. Выполнить шаги 1-4:

(1) Определить $q'_0 - e - closure(\{q_0\})$.

(2) Добавить q'_0 в Q' как непомеченное состояние

(3) Выполнить следующую процедуру:

```
while (в Q' есть непомеченное состояние R){
  пометить R;
  for (каждого входного символа a ∈ T){
    S = e – closure(move(R;a));
    if (S ≠ ∅){
      if (S ∉ Q')
        добавить S в Q' как непомеченное
        состояние;
      определить D'(R,a) = S,
    }
  }
}
```

(4) Определить $F'=\{S|S\in Q', S\cap F\neq\varnothing\}$.

Пример 3.6. Результат применения алгоритма 3.2 приведен на рис. 3.10.

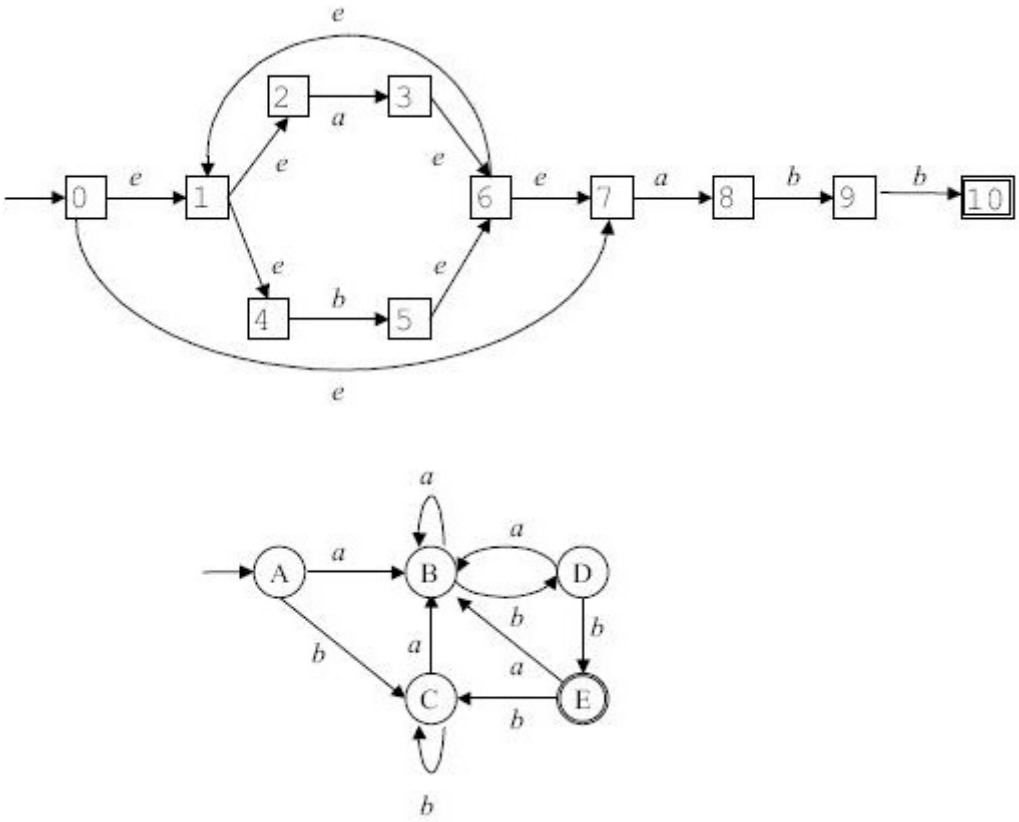


Рис. 3.10.

Построение детерминированного конечного автомата по регулярному выражению

Приведем теперь алгоритм построения по регулярному выражению детерминированного конечного автомата, допускающего тот же язык [?].

Пусть дано регулярное выражение r в алфавите T. К регулярному выражению r добавим маркер конца: (r)#. Такое регулярное выражение будем называть пополненным. В процессе своей работы алгоритм будет использовать пополненное регулярное выражение.

Алгоритм будет оперировать с синтаксическим деревом для пополненного регулярного выражения (r)# , каждый лист которого помечен символом $a \in T \cup \{e,\}$, а каждая внутренняя вершина помечена знаком одной из операций: • (конкатенация), | (объединение), * (итерация).

Каждому листу дерева (кроме e-листьев) присвоим уникальный номер, называемый позицией, и будем использовать его, с одной стороны, для ссылки на лист в дереве, и, с другой стороны, для ссылки на символ, соответствующий этому листу. Заметим, что если некоторый символ используется в регулярном выражении несколько раз, он имеет несколько позиций.

Обойдем дерево T снизу-вверх слева-направо и вычислим четыре функции: nullable,firstpos, lastpos и followpos. Три первые функции - nullable, firstpos и lastpos - определены на узлах дерева, а followpos - на множестве позиций. Значением всех функций, кроме nullable, является множество позиций. Функция followpos вычисляется через три остальные функции.

Функция firstpos(n) для каждого узла n синтаксического дерева регулярного выражения дает множество позиций, которые соответствуют первым символам в подцепочках, генерируемых подвыражением с вершиной в n. Аналогично, lastpos(n) дает множество позиций, которым соответствуют последние символы в подцепочках, генерируемых подвыражениями с вершиной n. Для узла n, поддеревья которого (то есть деревья, у которых узел n является корнем) могут породить пустое слово, определим nullable(n)=true, а для остальных узлов nullable(n)=false.

Таблица для вычисления функций nullable, firstpos и lastpos приведена на рис. 3.11.

Пример 3.7. На рис. 3.12 приведено синтаксическое дерево для пополненного регулярного выражения $(a|b)^*abb\#$ с результатом вычисления функций firstpos и lastpos. Слева от каждого узла расположено значение firstpos, справа от узла - значение lastpos. Заметим, что эти функции могут быть вычислены за один обход дерева.

Если i - позиция, то followpos(i) есть множество позиций j таких, что существует некоторая строка ... cd ..., входящая в язык, описываемый регулярным выражением, такая, что позиция i соответствует этому вхождению c, а позиция j - вхождению d.

узел n	$nullable(n)$	$firstpos(n)$	$lastpos(n)$
лист e	$true$	$\varnothing$	$\varnothing$
лист i (не e)	$false$	$\{i\}$	$\{i\}$
$\begin{array}{c}\vert \\ /\backslash \\ u\ v\end{array}$	$\begin{array}{c}nullable(u) \\ \texttt{or} \\ nullable(v)\end{array}$	$\begin{array}{c}firstpos(u) \\ \cup \\ firstpos(v)\end{array}$	$\begin{array}{c}lastpos(u) \\ \cup \\ lastpos(v)\end{array}$
$\begin{array}{c}\cdot \\ /\backslash \\ u\ v\end{array}$	$\begin{array}{c}nullable(u) \\ \texttt{and} \\ nullable(v)\end{array}$	$\begin{array}{l}\texttt{if } nullable(u) \texttt{ then} \\ firstpos(u) \\ \cup \\ firstpos(v) \\ \texttt{else } firstpos(u)\end{array}$	$\begin{array}{l}\texttt{if } nullable(v) \texttt{ then} \\ lastpos(u) \\ \cup \\ lastpos(v) \\ \texttt{else } lastpos(v)\end{array}$
$\begin{array}{c}* \\ \vert \\ v\end{array}$	$true$	$firstpos(v)$	$lastpos(v)$

Рис. 3.11.

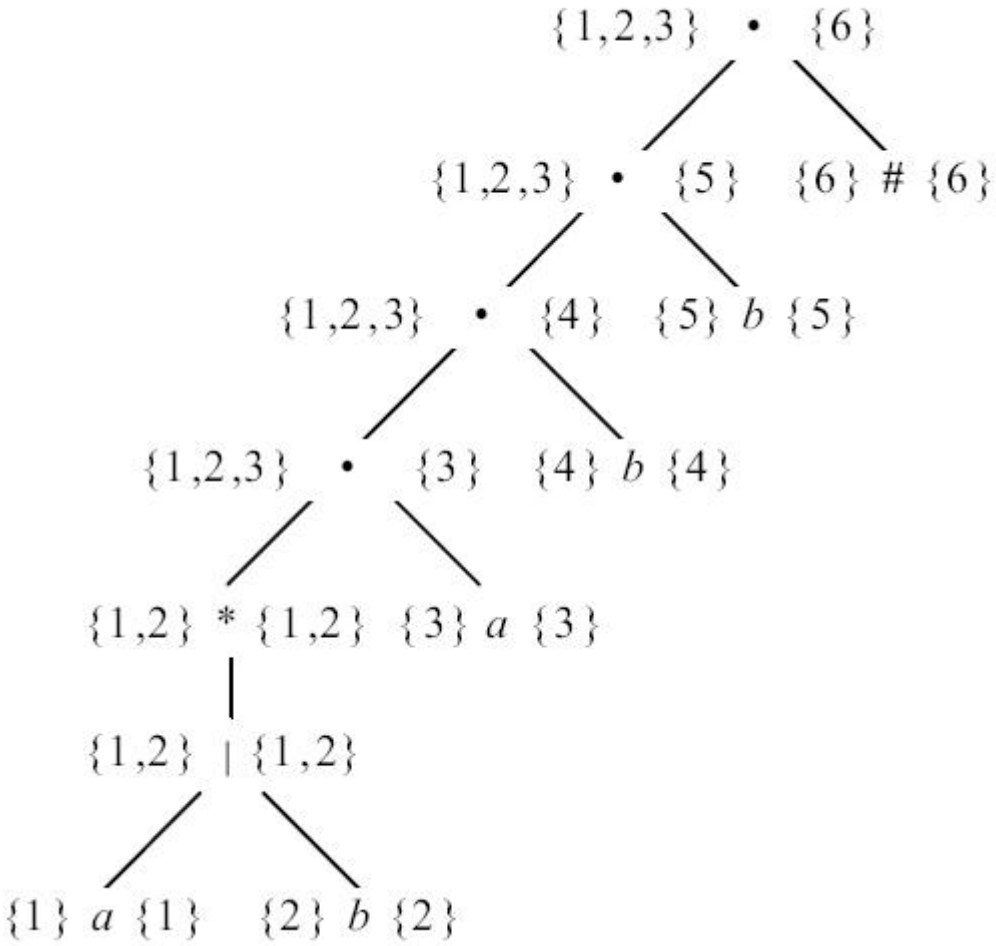


Рис. 3.12.

позиция	*followpos*
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	∅

Рис. 3.13.

Функция followpos может быть вычислена также за один обход дерева снизу-вверх по таким двум правилам.

1. Пусть n - внутренний узел с операцией • (конкатенация), u и v - его потомки. Тогда для каждой позиции i, входящей в lastpos(u), добавляем к множеству значений followpos(i) множество firstpos(v).

2. Пусть n - внутренний узел с операцией * (итерация), u - его потомок. Тогда для каждой позиции i, входящей в lastpos(u), добавляем к множеству значений followpos(i) множество firstpos(u).

Пример 3.8. Результат вычисления функции followpos для регулярного выражения из предыдущего примера приведен на рис. 3.13.

Алгоритм 3.3. Прямое построение ДКА по регулярному выражению.

Вход. Регулярное выражение r в алфавите T.

Выход. ДКА M = (Q, T, D, q_0, F), такой что L(M) = L(r).

Метод. Состояния ДКА соответствуют множествам позиций.

Вначале Q и D пусты. Выполнить шаги 1-6:

(1) Построить синтаксическое дерево для пополненного регулярного выражения (r)#.

(2) Обходя синтаксическое дерево, вычислить значения функций nullable, firstpos, lastpos и followpos.

(3) Определить q_0 = firstpos(root), где root - корень синтаксического дерева.

(4) Добавить q_0 в Q как непомеченное состояние.

(5) Выполнить следующую процедуру:

```
while (в Q есть непомеченное состояние R){
  пометить R;
  for (каждого входного символа a ∈ T),
    такого, что в R имеется позиция,
      которой соответствует a{
  пусть символ a в R соответствует позициям
  p1, ..., pn, и пусть S = ⋃ followpos(pi);
                           1≤i≤n
    if (S ≠ ∅){
      if (S ∉ Q)
        добавить S в Q как непомеченное
          состояние;
      определить D(R, a) = S,
    }
  }
}
```

(6) Определить F как множество всех состояний из Q, содержащих позиции, связанные с символом #.

Пример 3.9. Результат применения алгоритма 3.3 для регулярного выражения $(a|b)^*abb$ приведен на рис. 3.14.

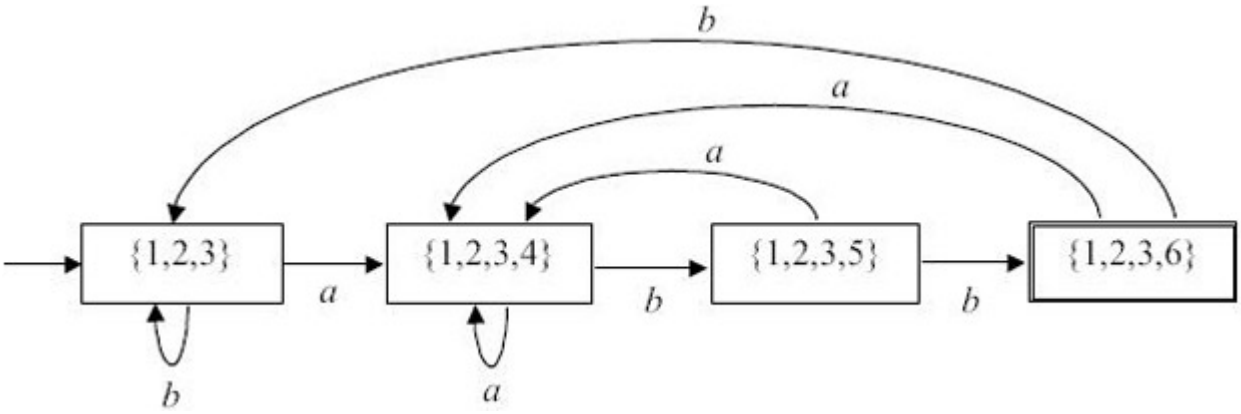


Рис. 3.14.

Построение детерминированного конечного автомата с минимальным числом состояний

Рассмотрим теперь алгоритм построения ДКА с минимальным числом состояний, эквивалентного данному ДКА [?].

Пусть M = (Q, T, D, q_0, F) - ДКА. Будем называть M всюду определенным, если D(q, a) $\neq\varnothing$, для всех $q \in Q$ и $a \in T$.

Лемма. Пусть M = (Q, T, D, q_0, F) - ДКА, не являющийся всюду определенным. Существует всюду определенный ДКА M', такой что L(M) = L(M').

Доказательство. Рассмотрим автомат

$$M' = (Q \cup \{q'\}, T, D', q_0, F),$$

где q' $\notin$ Q - некоторое новое состояние, а функция D' определяется следующим образом:

(1) Для всех $q \in Q$ и $a \in T$, таких что D(q, a) $\neq\varnothing$,, определить D'(q, a) = D(q, a).

(2) Для всех $q \in Q$ и $a \in T$, таких что D(q, a) = $\varnothing$, определить D'(q, a) = q'.

(3) Для всех $a \in T$ определить D'(q', a) = q'.

Легко показать, что автомат M' допускает тот же язык, что и M.

Приведенный ниже алгоритм получает на входе всюду определенный автомат. Если автомат не является всюду определенным, его можно сделать таковым на основании только что приведенной леммы.

Алгоритм 3.4. Построение ДКА с минимальным числом состояний.

Вход. Всюду определенный ДКА M = (Q, T, D, q_0, F).

Выход. ДКА $M' = (Q', T, D', q_0', F')$, такой что L(M) = L(M') и M' содержит наименьшее возможное число состояний.

Метод. Выполнить шаги 1-5:

(1) Построить начальное разбиение $\prod$ множества состояний из двух групп: заключительные состояния Q и остальные Q - F, то есть $\prod$ = {F, Q - F}.

(2) Применить к $\prod$ следующую процедуру и получить новое разбиение $\prod_{new}$:

```
for (каждой группы  G  в ∏){
   разбить G на подгруппы так, чтобы
   состояния s и t из G оказались
   в одной подгруппе тогда и только тогда,
   когда для каждого входного символа a
   состояния s и t имеют переходы по a
   в состояния из одной и той же группы в ∏,
   заменить G в ∏new на множество всех
   полученных подгрупп,
}
```

(3) Если $\prod_{new} = \prod$, полагаем $\prod_{res} = \prod$ и переходим к шагу 4, иначе повторяем шаг 2 с $\prod := \prod_{new}$.

(4) Пусть $\prod_{res} = \{G_1, \ldots, G_n\}$. Определим:

$Q' = \{G_1, \ldots, G_n\}$;

$q'_0 = G$, где группа $G \in Q'$ такова, что $q_0 \in G$;

$F = \{G \mid G \in Q' \text{ и } G \cap F \neq \emptyset\}$;

$D'(p', a) = q'$, если $D(p, a) = q$, где $p \in p'$ и $q \in q'$

Таким образом, каждая группа в $\prod_{res}$ становится состоянием нового автомата M'. Если группа содержит начальное состояние автомата M, то эта группа становится начальным состоянием автомата M'. Если группа содержит заключительное состояние M, она становится заключительным состоянием M'. Отметим, что каждая группа $\prod_{res}$ либо состоит только из состояний из F, либо не имеет состояний из F. Переходы определяются очевидным образом.

(5) Если M' имеет "мертвое"состояние, то есть состояние, которое не является допускающим и из которого нет путей в допускающие, удалить его и связанные с ним переходы из M'. Удалить из M' также все состояния, недостижимые из начального.

Пример 3.10. Результат применения алгоритма 3.4 приведен на рис. 3.15.

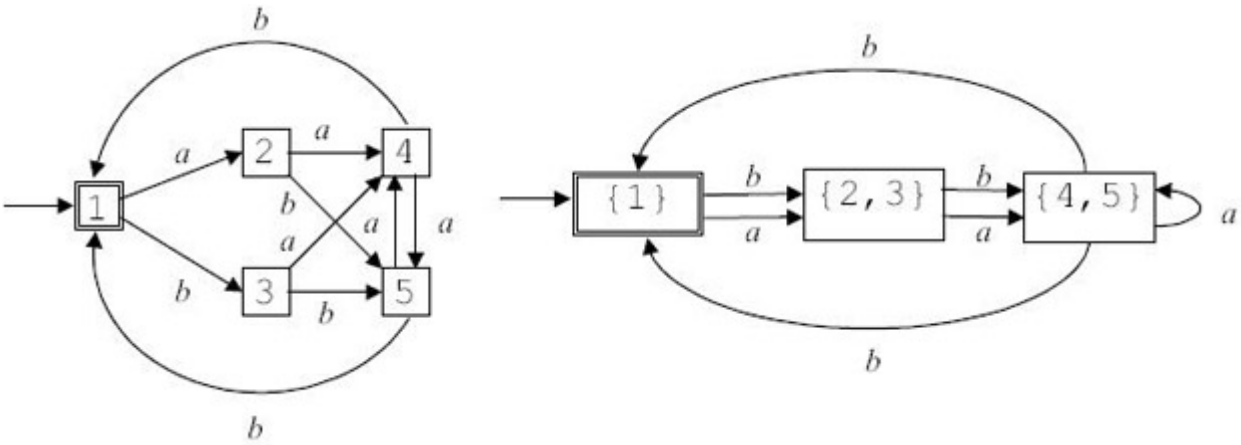


Рис. 3.15.

Связь регулярных множеств, конечных автоматов и регулярных грамматик

В разделе 3.3.3 приведен алгоритм построения детерминированного конечного автомата по регулярному выражению. Рассмотрим теперь как по описанию конечного автомата построить регулярное множество, совпадающее с языком, допускаемым конечным автоматом.

Теорема 3.1. Язык, допускаемый детерминированным конечным автоматом, является регулярным множеством.

Доказательство. Пусть L - язык, допускаемый детерминированным конечным автоматом

$$M = (\{q_1, \ldots, q_n\}, T, D, q_1, F).$$

Введем D^e - расширенную функцию переходов автомата M: $D^e(q, w) = p$, где $w \in T^*$, тогда и только тогда, когда $(q, w) \vdash^* (p, e)$.

Обозначим посредством R_{ij}^k множество всех слов x таких, что $D^e(q_i, x) = q_j$ и если $D^e(q_i, y) = q_s$ для любой цепочки y - префикса x, отличного от x и e, то $s \leq k$.

Иными словами, R_{ij}^k есть множество всех слов, которые переводят конечный автомат из состояния q_i в состояние q_j , не проходя ни через какое состояние q_s для $s > k$. Однако, i и j могут быть больше k.

R_{ij}^k может быть определено рекурсивно следующим образом:

$$R_{ij}^0 = \{a \mid a \in T, D(q_i, a) = q_j\},$$
$$R_{ij}^k = R_{ij}^{k-1} \bigcup R_{ik}^{k-1}(R_{kk}^{k-1}) * R_{kj}^{k-1}, \text{ где } 1 \leq k \leq n$$

Таким образом, определение R_{ij}^k означает, что для входной цепочки w, переводящей M из q_i в q_j без перехода через состояния с номерами, большими k, справедливо ровно одно из следующих двух утверждений:

1. Цепочка w принадлежит R_{ij}^{k-1}, то есть при анализе цепочки w автомат никогда не достигает состояний с номерами, большими или равными k.
2. Цепочка w может быть представлена как $w = w_1w_2w_3$, где $w_1 \in R_{ik}^{k-1}$ (подцепочка w_1 переводит M сначала в q_k), $w_2 \in (R_{kk}^{k-1})^*$ (подцепочка w_2 переводит M из q_k обратно в q_k, не проходя через состояния с номерами, большими или равными k), и $w_3 \in R_{kj}^{k-1}$ (подцепочка w_3 переводит M из состояния q_k в q_j) - рис. 3.16.

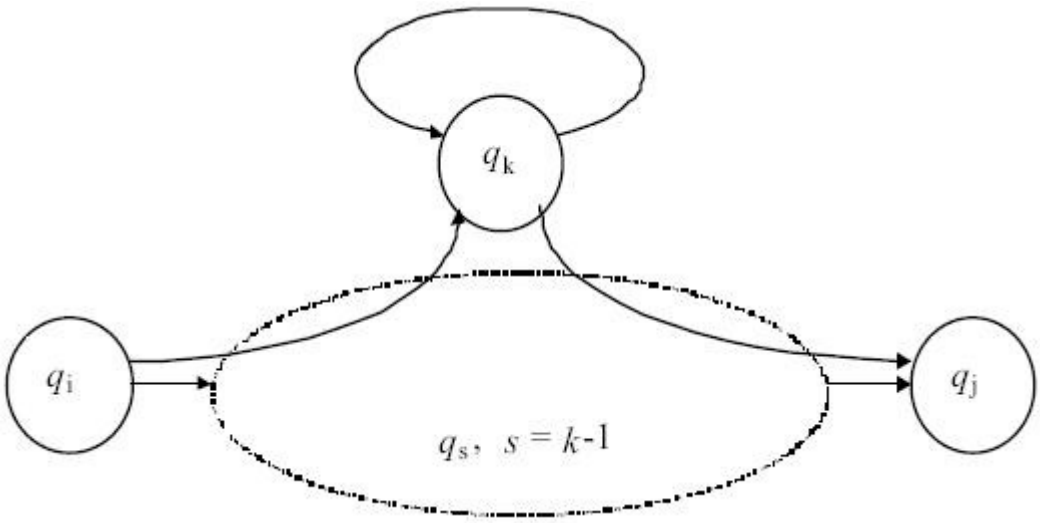


Рис. 3.16.

Тогда $L = \bigcup_{q_j \in F} R_{1j}^n$. Индукцией по k можно показать, что это множество является регулярным.

Таким образом, для всякого регулярного множества имеется конечный автомат, допускающий в точности это регулярное множество, и наоборот - язык, допускаемый конечным автоматом есть регулярное множество.

Рассмотрим теперь соотношение между языками, порождаемыми праволинейными грамматиками и допускаемыми конечными автоматами.

Праволинейная грамматика G = (N, T, P, S) называется регулярной, если

(1) каждое ее правило, кроме S $\to$e, имеет вид либо A $\to$aB, либо A $\to$a, где $A, B \in N, a \in T,$

(2) в том случае, когда $S \to e \in P$, начальный символ S не встречается в правых частях правил.

Лемма. Пусть G - праволинейная грамматика. Существует регулярная грамматика G' такая, что L(G) = L(G').

Доказательство. Предоставляется читателю в качестве упражнения.

Теорема 3.2. Пусть G = (N, T, P, S) - праволинейная грамматика. Тогда существует конечный автомат M = (Q, T, D, q_0, F) для которого L(M) = L(G).

Доказательство. На основании приведенной выше леммы, без ограничения общности можно считать, что G - регулярная грамматика.

Построим НКА M следующим образом:

1. состояниями M будут нетерминалы G плюс новое состояние R, не принадлежащее N. Так что $Q = N \cup \{R\}$,
2. в качестве начального состояния M примем S, то есть q_0 = S,
3. если P содержит правило S $\to$e, то $F = S, R$, иначе F = {R}. Напомним, что S не встречается в правых частях правил, если $S \to e \in P$,
4. состояние $R \in D(A, a)$, если $A \to a \in P$. Кроме того, D(A, a) содержит все B такие, что $A \to aB \in P$. $D(R, a) = \emptyset$, для каждого $a \in T$.

M, читая вход w, моделирует вывод w в грамматике G. Покажем, что L(M) = L(G). Пусть $w = a_1 a_2 \ldots a_n \in L(G), n > 1$. Тогда $S \Rightarrow a_1 A_1 \Rightarrow \ldots \Rightarrow a_1 a_2 \ldots a_{n-1} A_{n-1} \Rightarrow a_1 a_2 \ldots a_{n-1} a_n$ для некоторой последовательности нетерминалов A_1, A_2, ... , A_{n-1}. По определению, D(S, a_1) содержит A_1, D(A_1, a_2) содержит A_2, и т.д., D(A_{n-1}, a_n) содержит R. Так что $w \in L(M)$, поскольку D^e(S, w) содержит R, а $R \in F$. Если $e \in L(G)$, то $S \in F$, так что e \in L(M).

Аналогично, если $w = a_1 a_2 \ldots a_n \in L(M), n \geq 1$, то существует последовательность состояний S, A_1, A_2, ... , A_{n-1}, R такая, что D(S, a_1) содержит A_1, D(A_1, a_2) содержит A_2, и т.д. Поэтому $S \Rightarrow a_1 A_1 \Rightarrow a_1 a_2 A_2 \Rightarrow \ldots \Rightarrow a_1 a_2 \ldots a_{n-1} A_{n-1} \Rightarrow a_1 a_2 \ldots a_{n-1} a_n$ - вывод в G и $x \in L(G)$. Если $e \in L(M)$, то $S \in F$, так что $S \to e \in P$ и $e \in L(G)$.

Теорема 3.3. Для каждого конечного автомата M = (Q, T, D, q_0, F) существует праволинейная грамматика G = (N, T, P, S) такая, что L(G) = L(M).

Доказательство. Без потери общности можно считать, что автомат M - детерминированный. Определим грамматику G следующим образом:

1. нетерминалами грамматики G будут состояния автомата M. Так что N = Q,
2. в качестве начального символа грамматики G примем q_0, то есть S = q_0,
3. $A \to aB \in P$, если D(A, a) = B,
4. $A \to a \in P$, если D(A, a) = B и $B \in F$,
5. $S \to e \in P$, если $q_0 \in F$.

Доказательство того, что $S \Rightarrow^* w$ тогда и только тогда, когда $D^c(q_0, w) \in F$, аналогично доказательству теоремы 3.2.

В некоторых случаях для определения того, является ли язык регулярным, может быть полезным необходимое условие, которое называется леммой Огдена о разрастании.

Теорема 3.4. (Лемма о разрастании для регулярных множеств). Пусть L - регулярное множество. Существует такая константа k, что если $w \in L$ и $|w| \geq k$, то цепочку w можно представить в виде xyz, где $0 < |y| \leq k$ и $xy^iz \in L$ для всех $i \geq 0$.

Доказательство. Пусть M = (Q, Σ, D, q_0, F) - конечный автомат, допускающий L, то есть L(M) = L и k = |Q|. Пусть $w \in L$ и $|w| \geq k$. Рассмотрим последовательность конфигураций, которые проходит автомат M, допуская цепочку w. Так как в ней по крайней мере k + 1 конфигурация, то среди первых k+1 конфигурации найдутся две с одинаковыми состояниями. Таким образом, получаем существование такой последовательности тактов, что

$$(q_0, xyz) \vdash^* (q_1, yz) \vdash^r (q_1, z) \vdash^* (q_2, e)$$

для некоторых $q_1 \in Q, q_2 \in F$ и $0 < r \leq k$. Отсюда $0 < |y| \leq n$. Но тогда для любого i > 0 автомат может проделать последовательность тактов
$(q_0, xyz) \vdash^* (q_1, y^iz) \vdash^+ (q_1, y^{i-1}z) \ldots \vdash^+ (q_1, yz) \vdash^+ (q_1, z) \vdash^* (q_2, e)$Таким образом, $xy^iz \in L$ для всех i ≥1. Случай i = 0 то есть $xy \in L$ также очевиден.

С помощью леммы о разрастании можно показать, что не является регулярным множеством язык $L=\{0^n1^n|n\geq1\}$.

Допустим, что L регулярен. Тогда для достаточно большого $n0^n1^n$ можно представить в виде xyz, причем y ≠e и $xy^iz \in L$ для всех i ≥0. Если $y \in 0^+$ или $y \in 1^+$, то $xz = xy^0z \in L$. Если $y \in 0^+1^+$, то $xyyz \in L$. Получили противоречие. Следовательно, L не может быть регулярным множеством.

Программирование лексического анализа

Как уже отмечалось ранее, лексический анализатор (ЛА) может быть оформлен как подпрограмма. При обращении к ЛА, вырабатываются как минимум два результата: тип выбранной лексемы и ее значение (если оно есть).

Если ЛА сам формирует таблицы объектов, он выдает тип лексемы и указатель на соответствующий вход в таблице объектов. Если же ЛА не работает с таблицами объектов, он выдает тип лексемы, а ее значение передается, например, через некоторую глобальную переменную. Помимо значения лексемы, эта глобальная переменная может содержать некоторую дополнительную информацию: номер текущей строки, номер символа в строке и т.д. Эта информация может использоваться в различных целях, например, для диагностики.

В основе ЛА лежит диаграмма переходов соответствующего конечного автомата. Отдельная проблема здесь - анализ ключевых слов. Как правило, ключевые слова - это выделенные идентификаторы. Поэтому возможны два основных способа распознавания ключевых слов: либо очередная лексема сначала диагностируется на совпадение с каким-либо ключевым словом и в случае неуспеха делается попытка выделить лексему из какого-либо класса, либо, наоборот, после выборки лексемы идентификатора происходит обращение к таблице ключевых слов на предмет сравнения. Подробнее о механизмах поиска в таблицах будет сказано ниже (гл. "Организация таблиц символов "), здесь отметим только, что поиск ключевых слов может вестись либо в основной таблице имен и в этом случае в нее до начала работы ЛА загружаются ключевые слова, либо в отдельной таблице. При первом способе все ключевые слова непосредственно встраиваются в конечный автомат ЛА, во втором конечный автомат содержит только разбор идентификаторов.

В некоторых языках (например, ПЛ/1 или Фортран) ключевые слова могут использоваться в качестве обычных идентификаторов. В этом случае работа ЛА не может идти независимо от работы синтаксического анализатора. В Фортране возможны, например, следующие строки:

```
DO 10 I=1,25
DO 10 I=1.25
```

В первом случае строка - это заголовок цикла DO, во втором - оператор присваивания. Поэтому, прежде чем можно будет выделить лексему, ЛА должен заглянуть довольно далеко. Еще сложнее дело в ПЛ/1. Здесь возможны такие операторы:

```
IF ELSE THEN ELSE = THEN, ELSE THEN = ELSE,
```

или

```
DECLARE (A1, A2, ... , AN)
```

и только в зависимости от того, что стоит после ")", можно определить, является ли DECLARE ключевым словом или идентификатором. Длина такой строки может быть сколь угодно большой и уже невозможно отделить фазу синтаксического анализа от фазы лексического анализа.

Рассмотрим несколько подробнее вопросы программирования ЛА. Основная операция ЛА, на которую уходит большая часть времени его работы - это взятие очередного символа и проверка на принадлежность его некоторому диапазону. Например, основной цикл при выборке числа в простейшем случае может выглядеть следующим образом:

```
while (Insym <='9' && Insym>='0')
{ ... }
```

Программу можно значительно улучшить следующим образом **[5]**. Пусть LETTER, DIGIT, BLANK - элементы перечислимого типа. Введем массив map, входами которого будут символы, значениями - типы символов. Инициализируем массив map следующим образом:

```
map['a']=LETTER,
........
map['z']=LETTER,
map['0']=DIGIT,
........
map['9']=DIGIT,
map[' ']=BLANK,
........
```

Тогда приведенный цикл примет следующую форму:

```
while (map[Insym]==DIGIT)
{ ... }
```

Выделение ключевых слов может осуществляться после выделения идентификаторов. ЛА работает быстрее, если ключевые слова выделяются непосредственно.

Для этого строится конечный автомат, описывающий множество ключевых слов. На рис. 3.17 приведен фрагмент такого автомата.

Рассмотрим пример программирования этого конечного автомата на языке Си, приведенный в **[17]**:

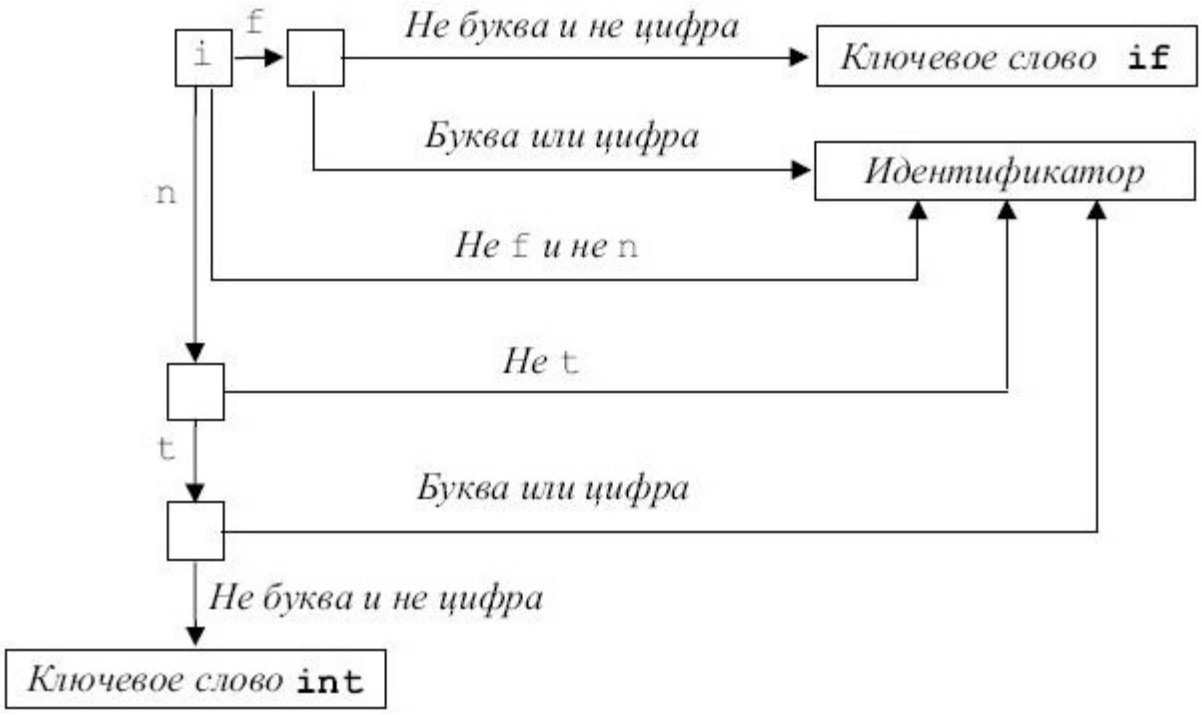
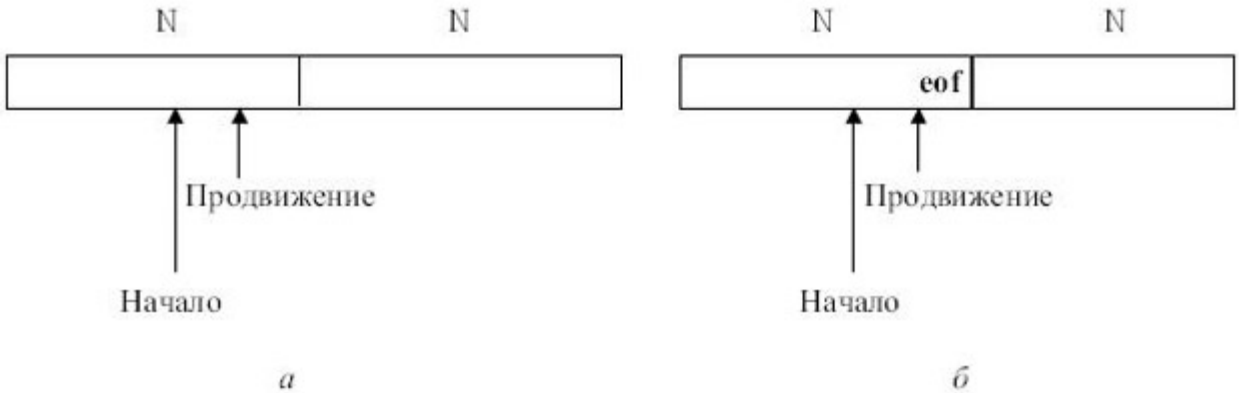


Рис. 3.17.

```
........
case 'i':
if (cp[0]=='f' &&!(map[cp[2]] & (DIGIT | LETTER)))
{cp++, return IF,}
if (cp[0]=='n' && cp[1]=='t'
&&!(map[cp] & (DIGIT | LETTER)))
{cp+=2, return INT,}
{ обработка идентификатора }
........
```

Здесь cp - указатель текущего символа. В массиве map классы символов кодируются битами.

Поскольку ЛА анализирует каждый символ входного потока, его скорость существенно зависит от скорости выборки очередного символа входного потока. В свою очередь, эта скорость во многом определяется схемой буферизации. Рассмотрим возможные эффективные схемы буферизации.



а *б*

Рис. 3.18.

Будем использовать буфер, состоящий из двух одинаковых частей длины N (рис. 3.18, а), где N - размер блока обмена (например, 1024, 2048 и т.д.). Чтобы не читать каждый символ отдельно, в каждую из половин буфера поочередно одной командой чтения считывается N символов. Если на входе осталось меньше N символов, в буфер помещается специальный символ (eof). На буфер указывают два указателя: продвижение и начало. Между указателями размещается текущая лексема. Вначале они оба указывают на первый символ выделяемой лексемы. Один из них, продвижение, продвигается вперед, пока не будет выделена лексема, и устанавливается на ее конец. После обработки лексемы оба указателя устанавливаются на символ, следующий за лексемой. Если указатель продвижение переходит середину буфера, правая половина заполняется новыми N символами. Если указатель продвижение переходит правую границу буфера, левая половина заполняется N символами и указатель продвижение устанавливается на начало буфера.

При каждом продвижении указателя необходимо проверять, не достигли ли мы границы одной из половин буфера. Для всех символов, кроме лежащих в конце половин буфера, требуются две проверки. Число проверок можно свести к одной, если в конце каждой половины поместить дополнительный _сторожевойї символ, в качестве которого логично взять eof (рис. 3.18, б).

В этом случае почти для всех символов делается единственная проверка на совпадение с eof и только в случае совпадения нужно дополнительно проверить, достигли ли мы середины или правого конца.

Конструктор лексических анализаторов LEX

Для автоматизации разработки ЛА было создано довольно много средств. Как правило, входным языком для них служат или праволинейные грамматики, или язык регулярных выражений. Одной из наиболее распространенных систем является LEX, работающий с расширенными регулярными выражениями. LEX-программа состоит из трех частей:

```
Объявления
%%
Правила трансляции
%%
```

Вспомогательные подпрограммы Секция объявлений включает объявления переменных, констант и определения регулярных выражений. При определении регулярных выражений могут использоваться следующие конструкции:

[abc]	- либо a, либо b, либо c,
[a-z]	- диапазон символов,
R*	- 0 или более повторений регулярного выражения R,
R+	- 1 или более повторений регулярного выражения R,
R1/R2	- R_1, если за ним следует R_2,
R1|R2	- либо R_1, либо R_2,
R?	- если есть R, выбрать его,
R$	- выбрать R, если оно последнее в строке,
^R	- выбрать R, если оно первое в строке,

[^R]	- дополнение к R,
R{n,m}	- повторение R от n до m раз,
{имя}	- именованное регулярное выражение,
(R)	- группировка.

Правила трансляции LEX-программ имеют вид

```
p_1 { действие_1 }
p_2 { действие_2 }
.................
p_n { действие_n }
```

где p_i - регулярное выражение, а действие_i - фрагмент программы, описывающий, какое действие должен сделать ЛА, когда образец p_i сопоставляется лексеме. В LEX действия записываются на Си.

Третья секция содержит вспомогательные процедуры, необходимые для действий. Эти процедуры могут транслироваться раздельно и загружаться с ЛА.

ЛА, сгенерированный LEX, взаимодействует с синтаксическим анализатором следующим образом. При вызове его синтаксическим анализатором ЛА посимвольно читает остаток входа, пока не находит самый длинный префикс, который может быть сопоставлен одному из регулярных выражений p_i. Затем он выполняет действие_i. Как правило, действие_i возвращает управление синтаксическому анализатору. Если это не так, то есть в соответствующем действии нет возврата, то ЛА продолжает поиск лексем до тех пор, пока действие не вернет управление синтаксическому анализатору. Повторный поиск лексем вплоть до явной передачи управления позволяет ЛА правильно обрабатывать пробелы и комментарии.

Синтаксическому анализатору ЛА возвращает единственное значение - тип лексемы. Для передачи информации о типе лексемы используется глобальная переменная yylval. Текстовое представление выделенной лексемы хранится в переменной yytext, а ее длина в переменной yylen.

Пример 3.11. LEX-программа для ЛА, обрабатывающего идентификаторы, числа, ключевые слова if, then, else и знаки логических операций.

```
%{ /*определения констант LT,LE,EQ,NE,GT,
GE,IF,THEN,ELSE,ID,NUMBER,RELOP, например,
через DEFINE или скалярный тип*/ %}
/*регулярные определения*/
delim [ \t\n]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*
number {digit}+(\.{digit}+)?,(E[+\-]?;,{digit}+)?;,
%%
{ws} {/* действий и возврата нет */}
if {return(IF),}
then {return(THEN),}
else {return(ELSE),}
{id} {yylval=install_id(), return(ID),}
{number} {yylval=install_num(), return(NUMBER),}
"<" {yylval=LT, return(RELOP),}
"<=" {yylval=LE, return(RELOP),}
"=" {yylval=EQ, return(RELOP),}
"<>" {yylval=NE, return(RELOP),}
">" {yylval=GT, return(RELOP),}
">=" {yylval=GE, return(RELOP),}
%%
install_id()
```

```
{/*подпрограмма, которая помещает лексему,
на первый символ которой указывает yytext,
длина которой равна yylen, в таблицу
символов и возвращает указатель на нее*/
}
install_num()
{/*аналогичная подпрограмма для размещения
лексемы числа*/
}
```

В разделе объявлений, заключенном в скобки %{ и %}, перечислены константы, используемые правилами трансляции. Все, что заключено в эти скобки, непосредственно копируется в программу ЛА lex.yy.c и не рассматривается как часть регулярных определений или правил трансляции. То же касается и вспомогательных подпрограмм третьей секции. В данном примере это подпрограммы install_id и install_num.

В секцию определений входят также некоторые регулярные определения. Каждое такое определение состоит из имени и регулярного выражения, обозначаемого этим именем. Например, первое определенное имя - это delim. Оно обозначает класс символов { \t\n\}, то есть любой из трех символов: пробел, табуляция или новая строка. Второе определение - разделитель, обозначаемый именем ws. Разделитель - это любая последовательность одного или более символов-разделителей. Слово delim должно быть заключено в скобки, чтобы отличить его от образца, состоящего из пяти символов delim.

В определении letter используется класс символов. Сокращение [A-Za-z] означает любую из прописных букв от A до Z или строчных букв от a до z. В пятом определении для id для группировки используются скобки, являющиеся метасимволами LEX. Аналогично, вертикальная черта - метасимвол LEX, обозначающий объединение.

В последнем регулярном определении number символ "+" используется как метасимвол "одно или более вхождений", символ "?" как метасимвол "ноль или одно вхождение". Обратная черта используется для того, чтобы придать обычный смысл символу, использующемуся в LEX как метасимвол. В частности, десятичная точка в определении number обозначается как "\", поскольку точка сама по себе представляет класс, состоящий из всех символов, за исключением символа новой строки. В классе символов [+\] обратная черта перед минусом стоит потому, что знак минус используется как символ диапазона, как в [A-Z].

Если символ имеет смысл метасимвола, то придать ему обычный смысл можно и по-другому, заключив его в кавычки. Так, в секции правил трансляции шесть операций отношения заключены в кавычки.

Рассмотрим правила трансляции, следующие за первым %%. Согласно первому правилу, если обнаружено ws, то есть максимальная последовательность пробелов, табуляций и новых строк, никаких действий не производится. В частности, не осуществляется возврат в синтаксический анализатор.

Согласно второму правилу, если обнаружена последовательность букв "if", нужно вернуть значение IF, которое определено как целая константа, понимаемая синтаксическим анализатором как лексема "if". Аналогично обрабатываются ключевые слова "then" и "else" в двух следущих правилах.

В действии, связанном с правилом для id, два оператора. Переменной yylval присваивается значение, возвращаемое процедурой install_id. Переменная yyl- val определена в программе lex.yy.c, выходе LEX, и она доступна синтаксическому анализатору. yylval хранит возвращаемое лексическое значение, поскольку второй оператор в действии, return(ID), может только возвратить код класса лексем. Функция install_id заносит идентификаторы в таблицу символов.

Аналогично обрабатываются числа в следующем правиле. В последних шести правилах yylval используется для возврата кода операции отношения, возвращаемое же функцией значение - это код лексемы relop.

Если, например, в текущий момент ЛА обрабатывает лексему "if", то этой лексеме соответствуют два образца: "if" и {id} и более длинной строки, соответствующей образцу, нет. Поскольку образец "if" предше-

ствует образцу для идентификатора, конфликт разрешается в пользу ключевого слова. Такая стратегия разрешения конфликтов позволяет легко резервировать ключевые слова.

Если на входе встречается "<=", то первому символу соответствует образец <, но это не самый длинный образец, который соответствует префиксу входа. Стратегия выбора самого длинного префикса легко разрешает такого рода конфликты.

4. Лекция: Синтаксический анализ

В данной лекции рассматривается понятие синтаксического анализа. Приводятся определения понятий упорядоченного графа, дерева вывода, автомата с магазинной памятью и его конфигурации. Приведены примеры задач, алгоритмов и доказательства теорем синтаксического анализа.

Контекстно-свободные грамматики и автоматы с магазинной памятью

Пусть G = (N, T, P, S) - КС-грамматика. Введем несколько важных понятий и определений.

Вывод, в котором в любой сентенциальной форме на каждом шаге делается подстановка самого левого нетерминала, называется левосторонним. Если $S \Rightarrow^* u$ в процессе левостороннего вывода, то u - левая сентенциальная форма. Аналогично определим правосторонний вывод. Обозначим шаги левого (правого) вывода $\Rightarrow_l$ ($\Rightarrow_r$).

Упорядоченным графом называется пара (V,E), где V есть множество вершин, а E - множество линейно упорядоченных списков дуг, каждый элемент которого имеет вид $((v, v_1), (v, v_2), \ldots , (v, v_n))$. Этот элемент указывает, что из вершины v выходят n дуг, причем первой из них считается дуга, входящая в вершину v_1, второй - дуга, входящая в вершинуv_2, и т.д.

Упорядоченным помеченным деревом называется упорядоченный граф (V,E), основой которого является дерево и для которого определена функция $f : V \rightarrow F$ (функция разметки) для некоторого множества F.

Упорядоченное помеченное дерево D называется деревом вывода (или деревом разбора) цепочки w в КС-грамматике G = (N, T, P, S), если выполнены следующие условия:

(1) корень дерева D помечен S;

(2) каждый лист помечен либо $a \in T$, либо e;

(3) каждая внутренняя вершина помечена нетерминалом $A \in N$;

(4) если X - нетерминал, которым помечена внутренняя вершина и $X_1, \ldots , X_n$ - метки ее прямых потомков в указанном порядке, то $X \rightarrow X_1 \ldots X_k$ - правило из множества P;

(5) Цепочка, составленная из выписанных слева направо меток листьев, равна w.

Процесс определения принадлежности данной строки языку, порождаемому данной грамматикой, и, в случае указанной принадлежности, построение дерева разбора для этой строки, называется синтаксическим анализом. Можно говорить о восстановлении дерева вывода (в частности, правостороннего или левостороннего) для строки, принадлежащей языку. По восстановленному выводу можно строить дерево разбора.

Грамматика G называется неоднозначной, если существует цепочка w, для которой имеется два или более различных деревьев вывода в G.

Грамматика G называется леворекурсивной, если в ней имеется нетерминал A такой, что для некоторой цепочки R существует вывод $A \Rightarrow^{+} A\alpha$.

Автомат с магазинной памятью (МП-автомат) - это семерка M = (Q, T, Г, D, q_0, Z_0, F), где

(1) Q - конечное множество состояний, представляющих всевозможные состояния управляющего устройства;

(2) T - конечный входной алфавит;

(3) Г - конечный алфавит магазинных символов;

(4) D - отображение множества Q x (T ∪{e}) x Г в множество конечных подмножеств Q x $Г^*$, называемое функцией переходов;

(5) $q_0 \in Q$ - начальное состояние управляющего устройства;

(6) $Z_0 \in \Gamma$ - символ, находящийся в магазине в начальный момент (начальный символ магазина);

(7) $F \subseteq Q$ - множество заключительных состояний.

Конфигурация МП-автомата - это тройка (q, w, u), где

(1) $q \in Q$ - текущее состояние управляющего устройства;

(2) $w \in T^*$ - непрочитанная часть входной цепочки; первый символ цепочки w находится под входной головкой; если w = e, то считается, что вся входная лента прочитана;

(3) $u \in \Gamma^*$ - содержимое магазина; самый левый символ цепочки u считается верхним символом магазина; если u = e, то магазин считается пустым.

Такт работы МП-автомата M будем представлять в виде бинарного отношения $\vdash$, определенного на конфигурациях.

Будем писать

$$(q,\ qw,\ Zu) \vdash (p,\ w,\ vu)$$

если множество D(q, a, Z) содержит (p, v), где $q, p \in Q,\ a \in T \cup \{e\},\ w \in T^*,\ Z \in \Gamma^*$ и $u, v \in \Gamma^*$ (верхушка магазина слева).

Начальной конфигурацией МП-автомата M называется конфигурация вида (q_0, w, Z_0), где $w \in T^*$, то есть управляющее устройство находится в начальном состоянии, входная лента содержит цепочку, которую нужно проанализировать, а в магазине имеется только начальный символ Z_0.

>Заключительной конфигурацией называется конфигурация вида (q, e, u), где $q \in F, u \in \Gamma^*$, то есть управляющее устройство находится в одном из заключительных состояний, а входная цепочка целиком прочитана.

Введем транзитивное и рефлексивно-транзитивное замыкание отношения $\vdash$, а также его степень k > 0 (обозначаемые $\vdash^{+}$, $\vdash^{*}$ и $\vdash^{k}$ соответственно).

Говорят, что цепочка w допускается МП-автоматом M, если $(q_0, w, Z_0) \vdash^* (q, e, u)$ для некоторых $q \in F$ и $u \in \Gamma^*$.

Множество всех цепочек, допускаемых автоматом M называется языком, допускаемым (распознаваемым, определяемым) автоматом M (обозначается L(M)).

Пример 4.1. Рассмотрим МП-автомат

M = ({q_0, q_1, q_2}, {a, b}, {Z, a, b}, D, q_0, Z, {q_2}),

у которого функция переходов D содержит элементы:

```
D(q0, a, Z) = {(q0, aZ)},
D(q0, b, Z) = {(q0, bZ)},
D(q0, a, a) = {(q0, aa), {q1, e)},
D(q0, a, b) = {(q0, ab)},
D(q0, b, a) = {(q0, ba)},
D(q0, b, b) = {(q0, bb), (q1, e)},
D(q1, a, a) = {(q1, e)},
D(q1, b, b) = {(q1, e)},
D(q1, e, Z) = {(q2, e)}.
```

Нетрудно показать, что $L(M) = \{ww^R | w \in \{a, b\}^+\}$, где w^R обозначает обращение ("переворачивание") цепочки w.

Иногда допустимость определяют несколько иначе: цепочка w допускается МП-автоматом M, если $(q_0, w, Z_0) \vdash^* (q, e, e)$ для некоторого $q \in Q$. В таком случае говорят, что автомат допускает цепочку опустошением магазина. Эти определения эквивалентны, ибо справедлива

Теорема 4.1. Язык допускается МП-автоматом тогда и только тогда, когда он допускается (некоторым другим автоматом) опустошением магазина.

Доказательство. Пусть L = L(M) для некоторого МП- автомата M = (Q, T, Γ, D, q_0, Z_0, F). Построим новый МП- автомат M', допускающий тот же язык опустошением магазина.

Пусть $M' = (Q \cup \{q_0', q_e\}, T, \Gamma \cup \{Z_0'\}, D', q_0', Z_0', \varnothing)$, где функция переходов D' определена следующим образом:

1. Если $(r, u) \in D(q, a, Z)$, то $(r, u) \in D'(q, a, Z)$ для всех $q \in Q, a \in T \cup \{e\}$ и $Z \in \Gamma$ (моделирование M),
2. $D'(q_0', e, Z_0') = \{(q_0, Z_0 Z_0')\}$ (начало работы),
3. Для всех $q \in F$ и $Z \in \Gamma \cup \{Z_0'\}$ множество D'(q, e, Z) содержит (q_e, e) (переход в состояние сокращения магазина без продвижения),
4. D'(q_e, e, Z) = {(q_e, e)} для всех $Z \in \Gamma \cup \{Z_0'\}$, (сокращение магазина).

Автомат сначала переходит в конфигурацию $(q_0, u, Z_0 Z_0')$ соответственно определению D' в п.2, затем в $(q, e, Y_1 \ldots Y_k Z_0')$,

$q \in F$ соответственно п.1, затем в $(q_e, e, Y_1 \ldots Y_k Z_0'), q \in F$ соответственно п.3, затем в (q_e, e, e) соответственно п.4. Нетрудно показать по индукции, что $(q_0, w, Z_0) \vdash^+ (q, e, u)$ (где $q \in F$) выполняется для автомата M тогда и только тогда, когда $(q_0', w, Z_0') \vdash^+ (q_e, e, e)$ выполняется для автомата M'. Поэтому L(M) = L', где L' - язык, допускаемый автоматом M' опустошением магазина.

Обратно, пусть M = (Q, T, Γ, D, q_0, Z_0, ∅) - МП - автомат, допускающий опустошением магазина язык L. Построим автомат M', допускающий тот же язык по заключительному состоянию.

Пусть $M' = (Q \cup \{q_0', q_f\}, T, \Gamma \cup \{Z_0'\}, D', q_0', Z_0', \{q_f\})$, где D' определяется следующим образом:

1. $D'(q_0', e, Z_0') = \{(q_0, Z_0 Z_0')\}$ - переход в "режим M",
2. Для каждого $q \in Q, a \in T \cup \{e\}, и Z \in \Gamma$ определим $D'(q, a, Z) = D(q, a, Z)$ - работа в "режиме M" ,
3. Для всех $q \in Q, (q_f, e) \in D'(q, e, Z_0')$ - переход в заключительное состояние.

Нетрудно показать по индукции, что L = L(M'). Одним из важнейших результатов теории контекстно-свободных языков является **доказательство эквивалентности МП-автоматов и КС-грамматик**.

Теорема 4.2. Язык является контекстно-свободным тогда и только тогда, когда он допускается МП-автоматом.

Доказательство. Пусть G = (N, T, P, S) - КС-граммати- ка. Построим МП-автомат, допускающий язык L(G) опустошением магазина.

Пусть M = ({q}, T, N ∪T, D, q, S, ∅), где D определяется следующим образом:

1. Если $A \to u \in P$, то $(q, u) \in D(q, e, A)$,
2. D(q, a, a) = {(q, e)} для всех $a \in T$.

Фактически, этот МП-автомат в точности моделирует все возможные выводы в грамматике G. Нетрудно показать по индукции, что для любой цепочки $w \in T^*$ вывод S ⇒$^+$w в грамматике G существует тогда и только тогда, когда существует последовательность тактов $(q, w, S) \vdash^+ (q, e, e)$ автомата M.

Наоборот, пусть дан M = (Q, T, Γ , D, q_0, Z_0, ∅) - МП- автомат, допускающий опустошением магазина язык L.

Построим грамматику G, порождающую язык L.

Пусть $G = (\{[qZr] \mid q, r \in Q, Z \in \Gamma\} \cup \{S\}, T, P, S)$, где P состоит из правил следующего вида:

1. $S \to [q_0 Z_0 q] \in P$ для всех $q \in Q$.
2. Если $(r, e) \in D(q, a, Z)$, то $[qZr] \to a \in P, a \in T \cup \{e\}$,
3. Если $(r, X_1 \ldots X_k) \in D(q, a, Z), k \geq 1$, то

$$[qZs_k] \to a[rX_1 s_1][s_1 X_2 s_2] \ldots [s_{k-1} X_k s_k]$$

для любого набора $s_1, s_2, \ldots, s_k$ состояний из Q,

Нетерминалы и правила вывода грамматики определены так, что работе автомата M при обработке цепочки w соответствует левосторонний вывод w в грамматике G.

Индукцией по числу шагов вывода в G или числу тактов M нетрудно показать, что $(q, w, A) \vdash^+ (p, e, e)$ тогда и только тогда, когда [qAp] ⇒$^+$ w.

Тогда, если $w \in L(G)$, то S ⇒[$q_0 Z_0$q] ⇒$^+$ w для некоторого $q \in Q$. Следовательно, $(q_0, w, Z_0) \vdash^+ (q, e, e)$ и поэтому $w \in L$. Аналогично, если $w \in L$, то $(q_0, w, Z_0) \vdash^+ (q, e, e)$. Значит, S ⇒[$q_0 Z_0$q] ⇒$^+$ w, и поэтому $w \in L(G)$.

МП-автомат M = (Q, T, Γ, D, q_0, Z_0, F) называется детерминированным (ДМП-автоматом), если выполнены два следующих условия:

(1) Множество D(q, a, Z) содержит не более одного элемента для любых $q \in Q, a \in T \cup \{e\}, Z \in \Gamma$;

(2) Если D(q, e, Z) $\neq\varnothing$, то D(q, a, Z) = $\varnothing$ для всех $a \in T$.

Допускаемый ДМП-автоматом язык называется детерминированным КС-языком.

Так как функция переходов ДМП-автомата содержит не более одного элемента для любой тройки аргументов, мы будем пользоваться записью D(q, a, Z) = (p, u) для обозначения D(q, a, Z) = {(p, u)}.

Пример 4.2. Рассмотрим ДМП-автомат

M = ({q_0, q_1, q_2}, {a, b, c}, {Z, a, b}, D, q_0, Z, {q_2}),

функция переходов которого определяется следующим образом:

$$D(q_0, X, Y) = (q_0, XY), X \in \{a, b\}, Y \in \{Z, a, b\},$$
$$D(q_0, c, Y) = (q_1, Y), Y \in \{a, b\},$$
$$D(q_1, X, X) = (q_1, e), X \in \{a, b\},$$
$$D(q_1, e, Z) = (q_2, e).$$

Нетрудно показать, что этот детерминированный МП-автомат допускает язык
$L = \{wcw^R \mid w \in \{a, b\}^+\}$.

К сожалению, ДМП-автоматы имеют меньшую распознавательную способность, чем МП-автоматы. Доказано, в частности, что существуют КС-языки, не являющиеся детерминированными КС-языками (таковым, например, является язык из примера 4.1).

Рассмотрим еще один важный вид МП-автомата.

Расширенным автоматом с магазинной памятью назовем семерку M = (Q, T, Γ , D, q_0, Z_0, F), где смысл всех символов тот же, что и для обычного МП-автомата, кроме D, представляющего собой отображение конечного подмножества множества Q x(T ∪{e}) x Γ^* во множество конечных подмножеств множества Q x Γ^*. Все остальные определения (конфигурации, такта, допустимости) для расширенного МП-автомата остаются такими же, как для обычного.

Теорема 4.3. Пусть M = (Q, T, Γ, D, q_0, Z_0, F) - расширенный МП-автомат. Тогда существует МП- автомат M', такой, что L(M') = L(M).

Расширенный МП-автомат M = (Q, T, Γ, D, q_0, Z_0, F) называется детерминированным, если выполнены следующие условия:

(1) Множество D(q, a, u) содержит не более одного элемента для любых $q \in Q, a \in T \cup \{e\}, u \in \Gamma^*$,

(2) Если D(q, a, u) $\neq\varnothing$, D(q, a, v) $\neq\varnothing$ и u $\neq$v, то не существует цепочки x такой, что u = vx или v = ux,

(3) Если D(q, a, u) $\neq\varnothing$, D(q, e, v) $\neq\varnothing$, то не существует цепочки x такой, что u = vx или v = ux.

Теорема 4.4. Пусть M = (Q, T, Γ, D, q_0, Z_0, F) - расширенный ДМП-автомат. Тогда существует ДМП- автомат M', такой, что L(M') = L(M).

ДМП-автомат и расширенный ДМП-автомат лежат в основе рассматриваемых далее в этой главе, соответственно, LL- и LR-анализаторов.

Определение. Говорят, что КС-грамматика находится в нормальной форме Хомского, если каждое правило имеет вид:

(1) либо A $\rightarrow$BC, A, B, C - нетерминалы,

(2) либо A $\rightarrow$a, a - терминал,

(3) либо S $\rightarrow$e и в этом случае S - не встречается в правых частях правил.

Утверждение. Любую КС-грамматику можно преобразовать в эквивалентную ей в нормальной форме Хомского.

Утверждение. Если КС-грамматика находится в нормальной форме Хомского, тогда для любой цепочки α, если $\alpha \in L(G)$ и m - высота дерева вывода с кроной α, $|\alpha| \leq 2^{m-1}$.

Теорема 4.5. (Лемма о разрастании для контекстно- свободных языков). Для любого КС-языка L существуют такие целые l и k, что любая цепочка $R \in L, |R| > l$, представима в виде R = uvwxy, где

(1) $|vwx| \leq k$

(2) $vx \neq e$

(3) $uv^i wx^i y \in L$ для любого $i \geq 0$.

Доказательство. Пусть L = L(G), где G = (N, Σ, P, S) - контекстно- свободная грамматика в нормальной форме Хомского. Обозначим через n число нетерминалов, т.е. n = |N|, и рассмотрим $l = 2^n$ и $k = 2^{n+1}$.

Для доказательства того, что l и k удовлетворяют условию теоремы, рассмотрим произвольную цепочку $\alpha \in L$, для которой $|\alpha| > l = 2n$. В силу Утверждения получаем, что высота дерева с кроной α больше n + 1 и есть путь по дереву (обозначим его через P), который проходит более чем через n + 1 вершин. Отсюда по определению дерева вывода имеем, что P содержит более n вершин, помеченных нетерминалами. Таким образом, существует нетерминал, который метит не менее двух вершин пути P. Среди всех таких нетерминалов пусть A - такой, что его вхождение, ближайшее к листу, не содержит других нетерминалов, обладающих этим свойством (если бы это было не так, то выбрали бы этот другой). Пусть q - вхождение A, ближайшее к листу, p - расположенное выше. Представим крону α в виде uvwxy, где w - крона поддерева D_1 с корнем q и vwx - крона поддерева D_2 с корнем p. Тогда высота поддерева D_2 не более (n - 1) + 2 + 1 = n + 2, так что $|vwz| \leq 2^{n+1}$.

Также очевидно, что $vx \neq e$, поскольку в силу определения нормальной формы Хомского p имеет двух сыновей, помеченных нетерминалами, из которых не выводится пустая цепочка.

Кроме того, $S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvwxy$, а также $A \Rightarrow^* vAx \Rightarrow^* vwx$. Отсюда получаем $A \Rightarrow^* v^i wx^i$ для всех $i \geq 0$ и $S \Rightarrow^* uv^i wx^i y$ для всех $i \geq 0$.

Пример. Покажем, что язык $L = \{a^n b^n c^n | n \geq 1\}$ не является контекстно-свободным языком.

Если бы он был КС-языком, то мы взяли бы константу k, которая определяется в лемме о разрастании. Пусть $z = a^k b^k c^k$. Тогда z = uvwxy. Так как $|vwx| \leq k$, то в цепочке vwx не могут быть вхождения каждого из символов a, b и c. Таким образом, цепочка uwy, которая по лемме о разрастании принадлежит L, содержит либо k символов a, либо k символов c. Но она не может иметь k вхождений каждого из символов

a, b и c, потому, что |uwy| < 3k. Значит, вхождений какого-то из этих символов в uwy больше, чем другого и, следовательно, uwy $\notin$L. Полученное противоречие позволяет заключить, что L - не КС-язык.

Преобразования КС-грамматик

Рассмотрим ряд преобразований, позволяющих "улучшить" вид контекстно-свободной грамматики без изменения порождаемого ею языка.

Назовем символ $X \in (N \cup T)$ недостижимым в КС- грамматике G = (N, T, P, S), если X не появляется ни в одной выводимой цепочке этой грамматики. Иными словами, символ X является недостижимым, если в G не существует вывода $S \Rightarrow^* \alpha X \beta$ ни для каких $\alpha, \beta \in (N \cup T)^*$.

Назовем символ $X \in (N \cup T)$ несводимым (бесплодным) в той же грамматике, если в ней не существует вывода вида $X \Rightarrow^* xwy$, где w, x, y принадлежат T^*.

Очевидно, что каждый недостижимый и/или несводимый символ является бесполезным, как и все правила, его содержащие.

При внимательном изучении вышеприведенных определений становится понятным, что а) целесообразно искать не непосредственно сами недостижимые (или несводимые) символы, а последовательно определять множество достижимых (или сводимых) символов, начиная с тех, которые по определению являются достижимыми (аксиома) и сводимыми (терминалы) - все остальные символы оказываются бесполезными, б) одновременное определение достижимых и сводимых символов невозможно, так как соответствующие процессы идут в противоположных направлениях (от корня к листьям и наоборот).

Алгоритм 4.1. Устранение недостижимых символов.

Вход. КС-грамматика G = (N, T, P, S).

Выход. КС-грамматика G' = (N', T', P', S) без недостижимых символов, такая, что L(G') = L(G).

Метод. Выполнить шаги 1-4:

(1) Положить $V_0 = \{S\}$ и i = 1,

(2) Положить $V_i = \{X \mid$ в P есть $A \to \alpha X \beta$ и $A \in V_{i-1}\} \cup V_{i-1}$,

(3) Если $V_i \neq V_{i-1}$, положить i = i + 1 и перейти к шагу 2, в противном случае перейти к шагу 4,

(4) Положить $N' = V_i \cap N$, $T' = V_i \cap T$. Включить в P' все правила из P, содержащие только символы из V_i.

Алгоритм 4.2. Устранение несводимых символов.

Вход. КС-грамматика G = (N, T, P, S).

Выход. КС-грамматика G' = (N', T', P', S) без несводимых символов, такая, что L(G') = L(G).

Метод. Выполнить шаги 1-4:

(1) Положить N' = T и i = 1,

(2) Положить $N_i = \{A \mid A \to \alpha \in P \text{ и } \alpha \in (N_{i-1})^*\} \cup N_{i-1}$,

(3) Если $N_i \neq N_{i-1}$, положить i = i + 1 и перейти к шагу 2, в противном случае положить $N_e = N_i$ и перейти к шагу 4,

(4) Положить $G_1 = ((N \cap N_e) \cup \{S\}, T, P_1, S)$, где P_1 состоит из правил множества P, содержащих только символы из $N_e \cup T$,

Чтобы устранить все бесполезные символы, необходимо применить к исходной грамматике сначала Алгоритм 4.2, а затем Алгоритм 4.1.

Пример. Все символы следующей грамматики

S $\rightarrow$AS | b

A $\rightarrow$AB

B $\rightarrow$a

являются достижимыми. Поэтому нарушение предложенного порядка применения к ней алгоритмов приведет лишь к частичному решению задачи.

КС-грамматика без бесполезных символов называется приведенной. Легко видеть, что для любой КС-грамматики существует эквивалентная приведенная. В дальнейшем будем предполагать, что все рассматривамые грамматики - приведенные.

Алгоритм Кока-Янгера-Касами

Приведем алгоритм синтаксического анализа, применимый для любой грамматики в нормальной форме Хомского

Алгоритм Кока-Янгера-Касами

Вход. КС-грамматика G = (N, T, P, S) в нормальной форме Хомского и входная цепочка $w = a_1 a_2 \ldots a_n \in T^+$.

Выход. Таблица разбора Tab для w такая, что $A \in t_{ij}$ тогда и только тогда, когда A $\Rightarrow^+$ $a_i a_{i+1} \ldots a_{i+j-1}$.

Метод.

(1) Положить $t_{i1} = \{A \mid A \rightarrow a_i \in P\}$ для каждого i. Так что, если $A \in t_{i1}$, то A $\Rightarrow^+$ a_i.

(2) Пусть t_{ij} вычислено для $1 \leq i \leq n$ и $1 \leq j' < j$. Положим t_{ij} = {A| для некоторого $1 \leq k < j$ правило $A \rightarrow BC \in P, B \in t_{ik}, C \in t_{i+k,j-k}\}$.

Так как $1 \leq k < j$, то $k < j$ и $j - k < j$. Так что t_{ik} и $t_{i+k,j-k}$ вычисляются раньше, чем t_{ij} . Если $A \in t_{ij}$, то A $\Rightarrow$BC $\Rightarrow^+$ a_i a_{i+k-1} C $\Rightarrow^+$ $a_I \ldots a_{i+k-1} a_{i+k} \ldots a_{i+j-1}$.

(3) Повторять шаг 2 до тех пор, пока не станут известны t_{ij} для всех $1 \leq i \leq n$ и $1 \leq j \leq n-i+1$.

Алгоритм нахождения левого разбора по таблице разбора Tab.

Вход. КС-грамматика G = (N, T, P, S) в нормальной форме Хомского с правилами, занумерованными от 1 до p, входная цепочка $w = a_1 a_2 \ldots a_n \in T^+$ и таблица разбора Tab.

Выход. Левый разбор цепочки w или сигнал ошибка.

Метод. Процедура gen(i, j, A):

(1) Если j = 1 и A $\Rightarrow a_i = p_m$, выдать m.

(2) Пусть j > 1 и k - наименьшее из чисел от 1 до j-1, для которых существует $B \in t_{ik}, C \in t_{i+k,j-k}$ и правило pm = A $\rightarrow$BC. Выдать m и выполнить gen(i, k, B), затем gen(i + k, j - k, C).

Выполнить gen(1, n, S), если $S \in t_{1,n}$, иначе ошибка.

Разбор сверху-вниз (предсказывающий разбор)

Алгоритм разбора сверху-вниз

Пусть дана КС-грамматика G = (N; T; P; S). Рассмотрим разбор сверху-вниз (предсказывающий разбор) для грамматики G.

Главная задача предсказывающего разбора - определение правила вывода, которое нужно применить к нетерминалу. Процесс предсказывающего разбора с точки зрения построения дерева разбора проиллюстрирован на рис. 4.1

Фрагменты недостроенного дерева соответствуют сентенциальным формам. Вначале дерево состоит только из одной вершины, соответствующей аксиоме S. В этот момент по первому символу входной цепочки предсказывающий анализатор должен определить правило S $\rightarrow X_1X_2$ ... ; которое должно быть применено к S. Затем необходимо определить правило, которое должно быть применено к X_1, и т.д., до тех пор, пока в процессе такого построения сентенциальной формы, соответствующей левому выводу, не будет применено правило Y $\rightarrow$a ... : Этот процесс затем применяется для следующего самого левого нетерминального символа сентенциальной формы.

На рис. 4.2 условно показана структура предсказывающего анализатора, который определяет

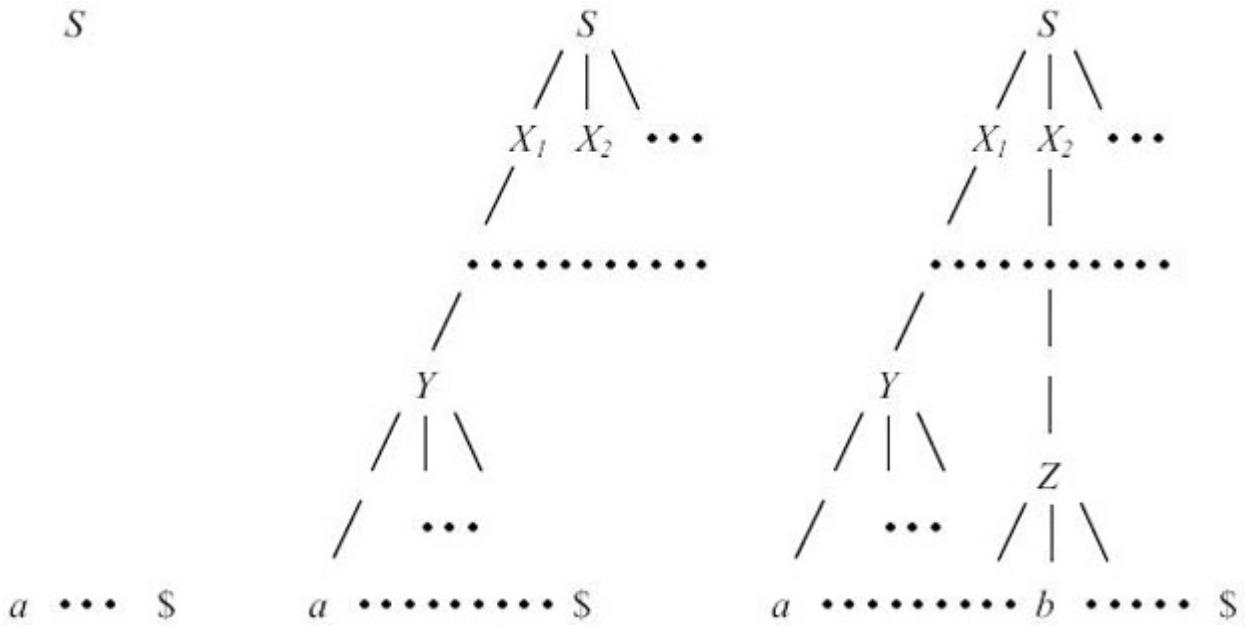


Рис. 4.1.

очередное правило с помощью таблицы. Такую таблицу можно построить и непосредственно по грамматике. Таблично-управляемый предсказывающий анализатор имеет входную ленту, управляющее устройство (программу), таблицу анализа, магазин (стек) и выходную ленту. Входная лента содержит анализи-

руемую строку, заканчивающуюся символом $ - маркером конца строки. Выходная лента содержит последовательность примененных правил вывода.

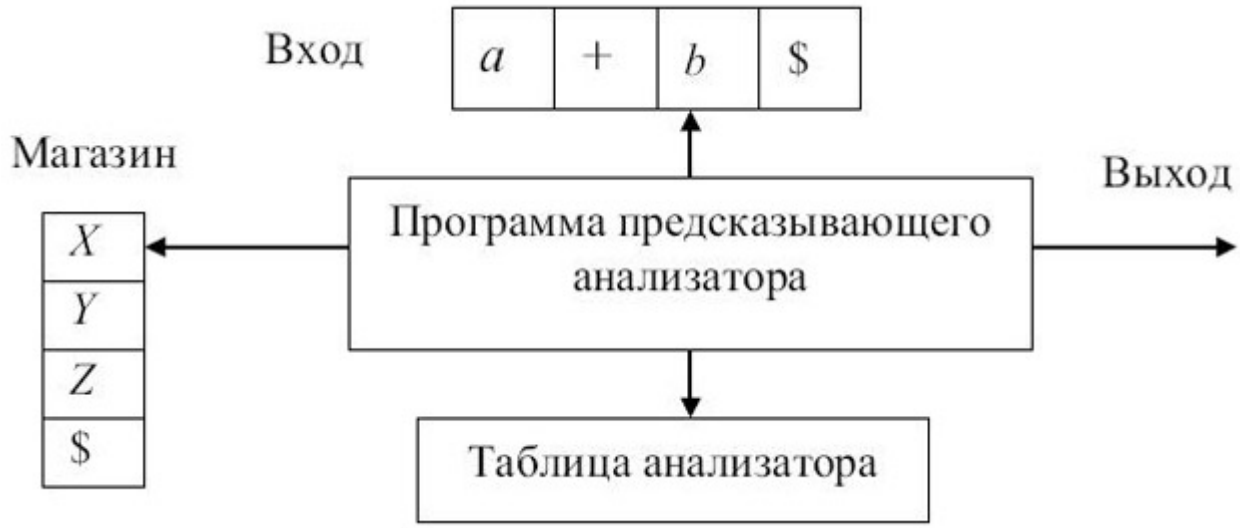


Рис. 4.2.

Таблица анализа - это двумерный массив M[A; a], где A - нетерминал, и a - терминал или символ $. Значением M[A; a] может быть некоторое правило грамматики или элемент "ошибка".

Магазин может содержать последовательность символов грамматики с $ на дне. В начальный момент магазин содержит только начальный символ грамматики на верхушке и $ на дне.

Анализатор работает следующим образом. Вначале анализатор находится в конфигурации, в которой магазин содержит S$, на входной ленте w$ (w - анализируемая цепочка), выходная лента пуста. На каждом такте анализатор рассматривает X - символ на верхушке магазина и a - текущий входной символ. Эти два символа определяют действия анализатора. Имеются следующие возможности.

1. Если X=a=$, анализатор останавливается, сообщает об успешном окончании разбора и выдает содержимое выходной ленты.
2. Если X= a ≠$, анализатор удаляет X из магазина и продвигает указатель входа на следующий входной символ.
3. Если X - терминал, и X ≠a, то анализатор останавливается и сообщает о том, что входная цепочка не принадлежит языку.
4. Если X - нетерминал, анализатор заглядывает в таблицу M[X; a]. Возможны два случая:
    1. Значением M[X; a] является правило для X. В этом случае анализатор заменяет X на верхушке магазина на правую часть данного правила, а само правило помещает на выходную ленту. Указатель входа не передвигается.
    2. Значением M[X; a] является "ошибка". В этом случае анализатор останавливается и сообщает о том, что входная цепочка не принадлежит языку. Работа анализатора может быть задана следующей программой:

```
Поместить '$', затем S в магазин;
do
   {X=верхний символ магазина;
if (X - терминал)
   if (X==InSym)
      {удалить X из магазина;
      InSym=очередной символ;
      }
   else {error(); break;}
else if (X - нетерминал)
   if (M[X,InSym]=="X->Y1Y2...Yk")
    {удалить X из магазина;
     поместить Yk,Yk-1,...Y1 в магазин
```

```
      (Y1 на верхушку);
     вывести правило X->Y1Y2...Yk;
    }
   else {error(); break;} /*вход таблицы M пуст*/
 }
while (X!='$'); /*магазин пуст*/
if (InSym != '$') error(); /*Не вся строка прочитана*/
```

Пример 4.3. Рассмотрим грамматику арифметических выражений G=({E; E', T, T', F}, {id, +, *, (,)}, P, E) с правилами:

$E \to TE'$ $\quad T' \to *FT'$

$E' \to +TE'$ $\quad T' \to \epsilon$

$E' \to \epsilon$ $\quad F \to (E)$

$T \to FT'$ $\quad F \to id.$

В таблица 4.3 приведена предсказывающего анализатора для этой грамматики. Пустые клетки таблицы соответствуют элементу "ошибка".

Таблица 4.3.

Нетерминал	Входной символ					
	id	+	*	(	)	$
E	E⟶TE'			E⟶TE'		
E'		E'⟶+TE'			E'⟶e	E'⟶e
T	T⟶FT'			T⟶FT'		
T'		T'⟶e	T'⟶*FT'		T'⟶e	T'⟶e
F	F⟶id			F⟶(E)		

При разборе входной цепочки id + id * id$ анализатор совершает последовательность шагов, изображенную в таблица 4.4. Заметим, что анализатор осуществляет в точности левый вывод. Если за уже просмотренными входными символами поместить символы грамматики в магазине, то можно получить в точности левые сентенциальные формы вывода. Дерево разбора для этой цепочки приведено на рис. рис. 4.3.

Таблица 4.4.

Магазин	Вход	Выход
E$	id + id * id$	
TE'$	id + id * id$	E ⟶TE'
FT'E'$	id + id * id$	T ⟶FT'
id T'E'$	id + id * id$	F ⟶id

T'E'$	+id * id$	
E'$	+id * id$	T' ⟶e
+TE'$	+id * id$	E' ⟶+TE
TE'$	id * id$	
FT'E'$	id * id$	T ⟶FT'
id T'E'$	id * id$	F ⟶id
T'E'$	*id$	
*F'T'E'$	*id$	T' ⟶*FT'
FT'E'$ id T'E'$	id$ id$	 F ⟶id
T'E'$	$	
E'$	$	T' ⟶e
$	$	E' ⟶e

Функции FIRST и FOLLOW

При построении таблицы предсказывающего анализатора нам потребуются две функции - FIRST и FOLLOW.

Пусть G = (N, T, P, S) - КС-грамматика. Для α- произвольной цепочки, состоящей из символов грамматики, определим FIRST(α) как множество терминалов, с которых

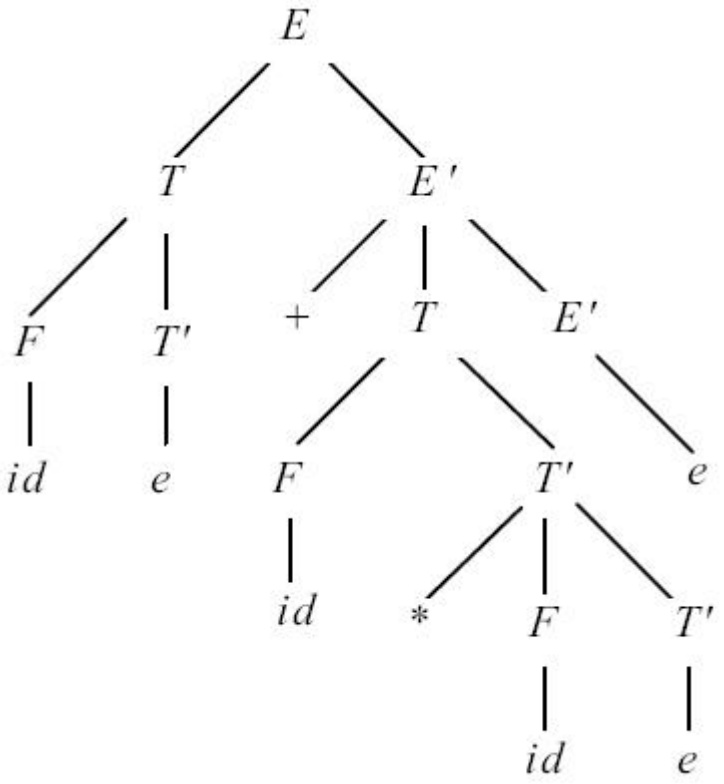


Рис. 4.3.

начинаются строки, выводимые из α. Если $\alpha \Rightarrow^* e$, то e также принадлежит FIRST(α).

Определим FOLLOW(A) для нетерминала A как множество терминалов a, которые могут появиться непосредственно справа от A в некоторой сентенциальной форме грамматики, то есть множество терминалов a таких, что существует вывод вида $S \Rightarrow^* \alpha A a \beta$ для некоторых $\alpha, \beta \in (N \cup T)^*$. Заметим, что между A и a в процессе вывода могут находиться нетерминальные символы, из которых выводится e. Если A может быть самым правым символом некоторой сентенциальной формы, то $ также принадлежит FOLLOW(A).

Рассмотрим алгоритмы вычисления функции FIRST.

Алгоритм 4.3. Вычисление FIRST для символов КС- грамматики.

Вход. КС-грамматика G = (N, T, P, S).

Выход. Множество FIRST(X) для каждого символа $X \in (N \cup T)$.

Метод. Выполнить шаги 1-3:

(1) Если X - терминал, то положить FIRST(X) = {X}; если X - нетерминал, положить FIRST(X) = ∅.

(2) Если в P имеется правило $X \rightarrow e$, то добавить e к FIRST(X).

(3) Пока ни к какому множеству FIRST(X) нельзя уже будет добавить новые элементы, выполнять:

```
do { continue = false;
  Для каждого нетерминала X
   Для каждого правила X → Y1Y2...Yk
   {i=1; nonstop = true;
```

```
   while (i ≤ k && nonstop)
    {добавить FIRST(Yi) n {e} к FIRST(X);
    if (Были добавлены новые элементы)
      continue = true;
    if (e ≠ FIRST (Yi)) nonstop = false;
    else i+ = 1;
    }
    if (nonstop) {добавить e к FIRST(X);
      continue = true;
  } } }
while (continue);
```

Алгоритм 4.4. Вычисление FIRST для цепочки.

Вход. КС-грамматика G = (N, T, P, S).

Выход. Множество $FIRST(X_1X_2 \dots X_n), X_i \in (N \cup T)$.

Метод. Выполнить шаги 1-3:

(1) При помощи **алгоритма 4.3**. вычислить FIRST(X) для каждого $X \in (N \cup T)$.

(2) Положить FIRST(X_1X_2 ... X_n) = ⌀.

(3)

```
 {i = 1; nonstop = true;
   while (i ≤ && nonstop)
    {добавить FIRST(Xi) n {e} к FIRST(u);
      if (e ∉ FIRST(Xi)nonstop = false;
        else i+ = 1;
    }
if (nonstop) {добавить e к FIRST(u);
    } }
```

Рассмотрим алгоритм вычисления функции FOLLOW.

Алгоритм 4.5. Вычисление FOLLOW для нетерминалов грамматики.

Вход. КС-грамматика G = (N, T, P, S).

Выход. Множество FOLLOW(X) для каждого символа $X \in N$.

Метод. Выполнить шаги 1-4:

(1) Положить FOLLOW(X) = ⌀для каждого символа $X \in N$.

(2) Добавить $ к FOLLOW(S).

(3) Если в P есть правило вывода A →αBβ, где $\alpha, \beta \in (N \cup T)^*$, то все элементы из FIRST(β), за исключением e, добавить к FOLLOW(B).

(4) Пока ничего нельзя будет добавить ни к какому множеству FOLLOW(X), выполнять:

если в P есть правило A →αB или A →αBβ, $\alpha, \beta \in (N \cup T)^*$, где FIRST(β) содержит e (β⇒*e), то все элементы из FOLLOW(A) добавить к FOLLOW(B).

Пример 4.4. Рассмотрим грамматику из **примера 4.3**. Для нее

```
FIRST(E) = FIRST(T) = FIRST(F) = {(, id}
FIRST(E') = {+, e}
FIRST(T') = {*, e}
FOLLOW(E) = FOLLOW(E') = { ), $}
FOLLOW(T) = FOLLOW(T') = {+, ), $}
FOLLOW(F) = {+, *, ), $}
```

Например, id и левая скобка добавляются к FIRST(F) на шаге 3 при i = 1, поскольку FIRST(id) = {id} и FIRST("(") = {"("} в соответствии с шагом 1. На шаге 3 при i = 1, в соответствии с правилом вывода T →FT', к FIRST(T) добавляются также id и левая скобка. На шаге 2 в FIRST(E') включается e.

Также при вычислении множеств FOLLOW на шаге 2 в FOLLOW(E) включается $. На шаге 3, на основании правила F →(E), к FOLLOW(E) добавляется также правая скобка. На шаге 4, примененном к правилу E →TE', в FOLLOW(E') включаются $ и правая скобка. Поскольку $E' \Rightarrow^* e$, они также попадают и во множество FOLLOW(T). В соответствии с правилом вывода E →TE', на шаге 3 в FOLLOW(T) включаются и все элементы из FIRST(E'), отличные от e.

Определим теперь функцию FIRSTk(R), где k - натуральное число и $\alpha \in (N \cup \Sigma)^*$.

$FIRST_k(\alpha) = \{w \in \Sigma^* \mid$ либо |w| < k и $\alpha \Rightarrow_G w$, либо $|w| = k$ и $R \Rightarrow_G wx$ для некоторого $x \in \Sigma^*\}$.

Если $\alpha \in \Sigma^*$, то $FIRST_k(\alpha)$ = {w}, где w - это первые k символов цепочки α при $|\alpha| \geq k$ и w = α при $|\alpha| < k$.

Приведем алгоритм вычисления функции $FIRST_k(\beta)$, где $\beta = X_1X_2 \ldots X_n \in (N \cup \Sigma)^*$.

Определение. Пусть Σ - некоторый алфавит. Если L_1 и L_2 - подмножества Σ^*, то положим

$$L_1 \oplus_k L_2 = \{w \mid \text{для некоторых } x \in L_1 \text{ и } y \in L2$$
$$\text{либо } w = xy, \text{ если } |xy| \leq k,$$
$$\text{либо } w \text{ состоит из первых } k \text{ символов}$$
$$\text{цепочки } xy\}$$

Лемма 4.1. Для любой КС-грамматики G = (N, Σ, P, S) и любых $\alpha, \beta \in (N \cup \Sigma)^*$

$$FIRST_k(\alpha\beta) = FIRST_k(\alpha) \oplus_k FIRST_k(\beta)$$

Доказательство оставляем читателю в качестве упражнения.

Алгоритм 4.6. Вычисление функции $FIRST_k(\alpha)$.

Вход. КС-грамматика G = (N, Σ, P, S) и цепочка $\alpha = X_1X_2 \ldots X_n \in (N \cup \Sigma)^*$.

Выход. $FIRST_k(\beta)$.

Метод. Так как по последней лемме

$$FIRST_k(\beta) = FIRST_k(X_1) \oplus_k FIRST_k(X_2) \oplus_k \ldots$$
$$\ldots \oplus_k FIRST_k(X_n);$$

то достаточно показать, как найти $FIRST_k(X)$ для $X \in N$.

Если $X \in \Sigma \cup \{e\}$, то очевидно, что $FIRST_k(X) = \{X\}$.

Определим множества $F_i(X)$ для каждого $X \in N \cup \Sigma$ и возрастающих значений i ≥0:

(1) $F_i(a) = \{a\}$ для всех $a \in \Sigma$ и i ≥0:

(2) $F_0(A) = \{x \mid x \in \Sigma^{*k}$ и существует правило A →x$^{\alpha}$ из P, для которого либо |x| = k, либо |x| < k и α= e}.

(3) Допустим, что $F_0, F_1..., F_{i-1}$ уже определены для всех $A \in N$. Тогда

$F_i(A) = F_{i-1}(A) \cup$ {x|A →$Y_1...Y_n$ принадлежит P и $x \in F_{i-1}(Y_1) \oplus_k F_{i-1}(Y_2) \oplus_k \ldots \oplus_k F_{i-1}(Y_n)\}$

(4) Так как $F_{i-1}(A) \subseteq F_i(A) \subseteq \Sigma^{*k}$ для всех A и i, то в конце концов мы дойдем до такого i, что $F_{i-1}(A) = F_i(A)$ для всех $A \in N$. Тогда положим $FIRST_k(A) = F_i(A)$ для этого значения i.

Конструирование таблицы предсказывающего анализатора

Для конструирования таблицы предсказывающего анализатора по грамматике G может быть использован алгоритм, основанный на следующей идее. Предположим, что A →α- правило вывода грамматики и $a \in FIRST(R)$. Тогда анализатор делает развертку A по α, если входным символом является a. Трудность возникает, когда α= e или $\alpha \Rightarrow^* e$. В этом случае нужно развернуть A в α, если текущий входной символ принадлежит FOLLOW(A) или если достигнут $ и $\$ \in FOLLOW(A)$.

Алгоритм 4.7. Построение таблицы предсказывающего анализатора.

Вход. КС-грамматика G = (N, T, P, S).

Выход. Таблица M[A; a] предсказывающего анализатора, $A \in N, a \in T \cup \{\$\}$.

Метод. Для каждого правила вывода A→α грамматики выполнить шаги 1 и 2. После этого выполнить шаг 3.

(1) Для каждого терминала a из FIRST(R) добавить A→R к M[A; a].

(2) Если $e \in FIRST(R)$, добавить A →R к M[A; b] для каждого терминала b из FOLLOW(A). Кроме того, если $e \in FIRST(R)$ и $\$ \in FOLLOW(A)$, добавить A →$\alpha$к M[A; $].

(3) Положить все неопределенные входы равными "ошибка".

Пример 4.5. Применим **алгоритм 4.7** к грамматике из **примера 4.3.** Поскольку FIRST(TE') = FIRST(T) = {(, id }, в соответствии с правилом вывода E →TE' входы M[E, ( ] и M[E, id] становятся равными E →TE'.

В соответствии с правилом вывода E' →+TE' значение M[E', +] равно E' →+TE'. В соответствии с правилом вывода E' →e значения M[E',)] и M[E', $] равны E' →e, поскольку FOLLOW(E') = {), $}.

Таблица анализа, построенная по **алгоритму 4.7.** для этой грамматики, приведена в таблица 4.3.

LL(1)-грамматики

Алгоритм 4.7 построения таблицы предсказывающего анализатора может быть применен к любой КС-грамматике. Однако для некоторых грамматик построенная таблица может иметь неоднозначно определенные входы. Например, нетрудно доказать, что если грамматика леворекурсивна или неоднозначна, таблица будет иметь по крайней мере один неоднозначно определенный вход.

Грамматики, для которых таблица предсказывающего анализатора не имеет неоднозначно определенных входов, называются LL(1)-грамматиками. Предсказывающий анализатор, построенный для LL(1)-грамматики, называется LL(1)-анализатором. Первая буква L в названии связано с тем, что входная цепочка читается слева направо, вторая L означает, что строится левый вывод входной цепочки, 1 - что на каждом шаге для принятия решения используется один символ непрочитанной части входной цепочки.

Доказано, что **алгоритм 4.7** для каждой из LL(1)-грам- матик G строит таблицу предсказывающего анализатора, распознающего все цепочки из L(G) и только эти цепочки. Нетрудно доказать также, что если G - LL(1)-грамматика, то L(G) - детерминированный КС-язык.

Справедлив также следующий критерий LL(1)-граммати- ки. Грамматика G = (N, T, P, S) является LL(1)-грамматикой тогда и только тогда, когда для каждой пары правил $A \to \alpha$, $A \to \beta$ из P

(то есть правил с одинаковой левой частью) выполняются следующие 2 условия:

(1) $FIRST(\alpha) \cap FIRST(\beta) = \varnothing$;

(2) Если $\epsilon \in FIRST(\alpha)$, то FIRST(β) ∩FOLLOW(A)= ∅.

Язык, для которого существует порождающая LL(1)- грамматика, называют LL(1)-языком. Доказано, что проблема определения, порождает ли грамматика LL-язык, является алгоритмически неразрешимой.

Пример 4.6. Неоднозначная грамматика не является LL(1). Примером может служить следующая грамматика

```
G = ({S, E}, {if, then, else, a, b}, P, S) с правилами:
   S → if E then S | if E then S else S | a
   E → b
```

Эта грамматика является неоднозначной, что иллюстрируется на рис. 4.4.

LL(k)-грамматики

Определение. КС-грамматика G = (N, Σ, P, S) называется LL(k)-грамматикой для некоторого фиксированного k, если из

(1) $S \to_l^* \omega A \alpha \to_l \omega \beta \alpha \to^* \omega x$

(2) $S \Rightarrow_l^* \omega A \alpha \Rightarrow_l \omega \gamma \alpha \Rightarrow^* \omega x$ для которых

и

$FIRST_k(x) = FIRST_k(y)$, вытекает, что β = γ.

Говоря менее формально, G будет LL(k)- грамматикой, если для данной цепочки $\omega A \alpha \in (N \cup \Sigma)^*$ и первых k символов (если они есть), выводящихся из A^{α}, существует не более одного правила, которое можно применить к A, чтобы получить вывод какой-нибудь терминальной цепочки,

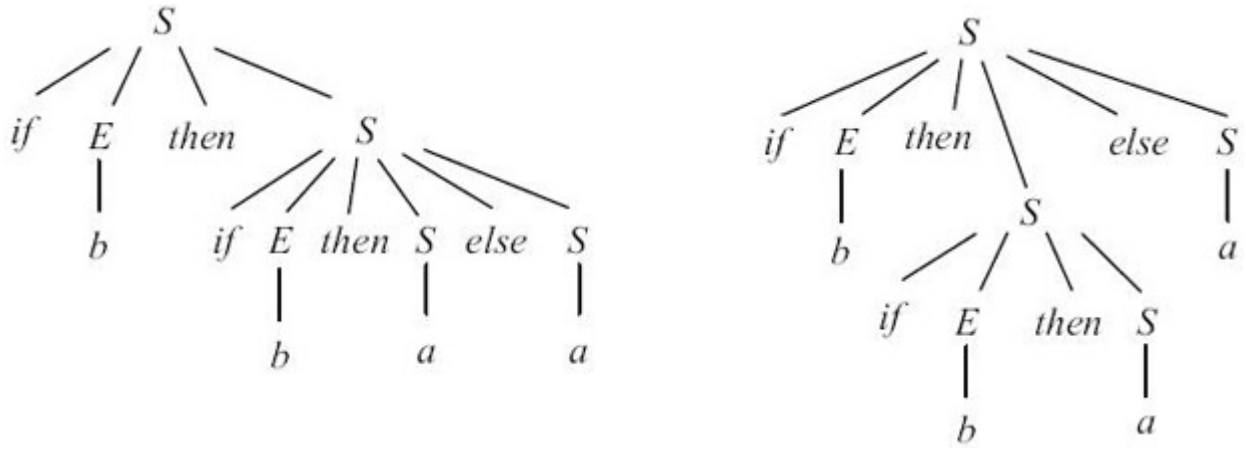


Рис. 4.4.

начинающейся с ω и продолжающейся упомянутыми k терминалами.

Грамматика называется LL(k)-грамматикой, если она LL(k)-грамматика для некоторого k.

Пример 4.7. Рассмотрим грамматику G = ({S, A, B}, {0, 1, a, b}, P, S), где P состоит из правил

```
S ⟶ A | B,
A ⟶ aAb | 0,
B ⟶ aBbb | 1.
```

Здесь $L(G) = a^n0b^n \mid n \geq 0 \cup a^n1b^{2n} \mid n \geq 0$. G не является LL(k)-грамматикой ни для какого k. Интуитивно, если мы начинаем с чтения достаточно длинной цепочки, состоящей из символов a, то не знаем, какое из правил S ⟶A и S ⟶B было применено первым, пока не встретим 0 или 1.

Обращаясь к точному определению LL(k)-грамматики, положим ω = α= e; β = A; γ = B; $x = a^k0b^k$ и $y = a^k1b^{2k}$. Тогда выводы

$$S \Rightarrow_l^0 S \Rightarrow_l A \Rightarrow_l^* a^k0b^k$$
$$S \Rightarrow_l^0 S \Rightarrow_l B \Rightarrow_l^* a^k1b^{2k}$$

соответствуют выводам (1) и (2) определения. Первые k символов цепочек x и y совпадают. Однако заключение β = γ ложно. Так как k здесь выбрано произвольно, то G не является LL-грамматикой.

Следствия определения LL(k)- грамматики

Теорема 4.6. КС-грамматика G = (N, Σ, P, S) является LL(k)-грамматикой тогда и только тогда, когда для двух различных правил A ⟶β и A ⟶γ из P пересечение $FIRST_k(\beta\alpha) \cap FIRST_k(\gamma\alpha)$ пусто при всех таких ωAα, что $S \Rightarrow_l^* \omega A\alpha$.

Доказательство. Необходимость. Допустим, что ω, A, α, β и γ удовлетворяют условиям теоремы, а $FIRST_k(\beta\alpha) \cap FIRST_k(\gamma\alpha)$ содержит x. Тогда по определению FIRST для некоторых y и z найдутся выводы

$$S \Rightarrow_l^* \omega A\alpha \Rightarrow_l \omega\beta\alpha \Rightarrow_l^* \omega xy$$

и

$$S \Rightarrow_l^* \omega A\alpha \Rightarrow_l \omega\gamma\alpha \Rightarrow_l^* \omega xy$$

(Заметим, что здесь мы использовали тот факт, что N не содержит бесполезных нетерминалов, как это предполагается для всех рассматриваемых грамматик.) Если |x| < k; то y = z = e. Так как β ≠γ, то G не LL(k)-грамматика.

Достаточность. Допустим, что G не LL(k)-грамматика.

Тогда найдутся такие два вывода

$$S \Rightarrow_l^* \omega A\alpha \Rightarrow_l \omega\beta\alpha \Rightarrow_l^* \omega x$$

и

$$S \Rightarrow_l^* \omega A\alpha \Rightarrow_l \omega\gamma\alpha \Rightarrow_l^* \omega y$$

что цепочки x и y совпадают в первых k позициях, но β ≠γ. Поэтому A →β и A →γ - различные правила из P и каждое из множеств FIRSTk(βα) и FIRSTk(γα) содержит цепочку $FIRST_k(x)$, совпадающую с цепочкой $FIRST_k(y)$.

Пример 4.8. Грамматика G, состоящая из двух правил S →aS | a, не будет LL(1)-грамматикой, так как

$FIRST_1(aS) = FIRST_1(a) = a$.

Интуитивно это можно объяснить так: видя при разборе цепочки, начинающейся символом a, только этот первый символ, мы не знаем, какое из правил S →aS или S →a надо применить к S. С другой стороны, G - это LL(2)-грамматика. В самом деле, в обозначениях только что представленной теоремы, если $S \Rightarrow_l^* \omega A\alpha$, то A = S и α= e. Так как для S даны только два указанных правила, то β = aS и γ = a. Поскольку FIRST2(aS) = aa и FIRST2(a) = a, то по последней теореме G будет LL(2)-грамматикой.

Удаление левой рекурсии

Основная трудность при использовании предсказывающего анализа - это нахождение такой грамматики для входного языка, по которой можно построить таблицу анализа с однозначно определенными входами. Иногда с помощью некоторых простых преобразований грамматику, не являющуюся LL(1), можно привести к эквивалентной LL(1)-грамматике. Среди этих преобразований наиболее эффективными являются левая факторизация и удаление левой рекурсии. Здесь необходимо сделать два замечания. Во-первых, не всякая грамматика после этих преобразований становится LL(1), и, во-вторых, после таких преобразований получающаяся грамматика может стать менее понимаемой.

Непосредственную левую рекурсию, то есть рекурсию вида A→Aα, можно удалить следующим способом. Сначала группируем A-правила:

A →$A\alpha_1$ |$A\alpha_2$| ... |$A\alpha_m$|β_1|β_2| ... |β_n;

где никакая из строк β_i не начинается с A. Затем заменяем этот набор правил на

$$A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$
$$A \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid e$$

где A' - новый нетерминал. Из нетерминала A можно вывести те же цепочки, что и раньше, но теперь нет левой рекурсии. С помощью этой процедуры удаляются все непосредственные левые рекурсии, но не удаляется левая рекурсия, включающая два или более шага. Приведенный ниже **алгоритм 4.8** позволяет удалить все левые рекурсии из грамматики.

Алгоритм 4.8. Удаление левой рекурсии.

Вход. КС-грамматика G без e-правил (вида A $\rightarrow$e).

Выход. КС-грамматика G' без левой рекурсии, эквивалентная G.

Метод. Выполнить шаги 1 и 2.

(1) Упорядочить нетерминалы грамматики G в произвольном порядке.

(2) Выполнить следующую процедуру:

```
for(i = 1; i <= n; i++){
  for(j = 1; j <= i - 1; j++){
    пусть A_j → β_1 | β_2 | ... | β_k - все текущие правила
    для A_j;
    заменить все правила вида A_i → A_jα
    на правила A_i → β_1α | β_2α | ... | β_kα;
  }
  удалить правила вида A_i → A_i;
  удалить непосредственную левую рекурсию в
  правилах для A_i;
}
```

После (i-1)-й итерации внешнего цикла на шаге 2 для любого правила вида $A_k \rightarrow A_s\alpha$, где k < i, выполняется s > k. В результате на следующей итерации (по i) внутренний цикл (по j) последовательно увеличивает нижнюю границу по m в любом правиле $A_i \rightarrow A_m\alpha$, пока не будет m $\geq$i. Затем, после удаления непосредственной левой рекурсии для A_i-правил, m становится больше i.

Алгоритм 4.8 применим, если грамматика не имеет e- правил (правил вида A $\rightarrow$e). Имеющиеся в грамматике e- правила могут быть удалены предварительно. Получающаяся грамматика без левой рекурсии может иметь e-правила.

Левая факторизация

Основная идея левой факторизации в том, что в том случае, когда неясно, какую из двух альтернатив надо использовать для развертки нетерминала A, нужно изменить A-правила так, чтобы отложить решение до тех пор, пока не будет достаточно информации для принятия правильного решения.

Если $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ - два A-правила и входная цепочка начинается с непустой строки, выводимой из α, мы не знаем, разворачивать ли по первому правилу или по второму. Можно отложить решение, развернув $A \rightarrow \alpha A'$. Тогда после анализа того, что выводимо из α, можно развернуть по $A' \rightarrow \beta_1$ или по $A' \rightarrow \beta_2$. Левофакторизованные правила принимают вид

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 | \beta_2$

Алгоритм 4.9. Левая факторизация грамматики.

Вход. КС-грамматика G.

Выход. Левофакторизованная КС-грамматика G', эквивалентная G.

Метод. Для каждого нетерминала A найти самый длинный префикс α, общий для двух или более его альтернатив. Если $\alpha \neq e$, то есть существует нетривиальный общий префикс, заменить все A-правила

$A \to \alpha\beta_1 | \alpha\beta_2 | \ldots | \alpha\beta_n | z,$

где z обозначает все альтернативы, не начинающиеся с α, на

$A \to \alpha A' | z$

$A' \to \beta_1 | \beta_2 | \ldots | \beta_n$

где A' - новый нетерминал. Применять это преобразование, пока никакие две альтернативы не будут иметь общего префикса.

Пример 4.9. Рассмотрим вновь грамматику условных операторов из **примера 4.6**:

```
S → if E then S | if E then S else S | a
E → b
```

После левой факторизации грамматика принимает вид

```
S → if E then SS' | a
S' → else S | e
E → b
```

К сожалению, грамматика остается неоднозначной, а значит, и не LL(1)-грамматикой.

Рекурсивный спуск

Выше был рассмотрен один из вариантов таблично-управ- ляемого предсказывающего анализа, когда магазин явно использовался в процессе работы анализатора. Возможен иной вариант предсказывающего анализа, в котором каждому нетерминалу сопоставляется процедура (вообще говоря, рекурсивная), и магазин образуется неявно при вызовах таких процедур. Процедуры рекурсивного спуска могут быть записаны, как показано ниже.

В процедуре A для случая, когда имеется правило $A \to u_i$, такое, что $u_i \Rightarrow^* e$ (напомним, что не может быть больше одного правила, из которого выводится e), приведены два варианта - 1.1 и 1.2. В варианте 1.1 делается проверка, принадлежит ли следующий входной символ FOLLOW(A): Если нет - выдается ошибка. В варианте 1.2 этого не делается, так что анализ ошибки откладывается на процедуру, вызвавшую A.

```
void A(){ // A → u1 | u2 | ... | uk
  if (InSym ∈ FIRST(ui)) // только одному!
    if (parse(ui))
      result("A → ui");
    else error();
  else
  //Вариант 1:
  if (имеется правило A → ui такое, что ui ⇒* e)
    //Вариант 1.1
    if (InSym ∈ FOLLOW(A))
      result("A → ui");
    else error();
    //Конец варианта 1.1
    //Вариант 1.2:
    result("A → ui");
    //Конец варианта 1.2
  //Конец варианта 1
  //Вариант 2:
  if (нет правила A → ui такого, что ui ⇒* e)
    error();
  //Конец варианта 2
}
boolean parse(u){ // из u не выводится e
  v = u;
  while (v ≠ e){ // v = Xz
    if (X - терминал a)
      if (InSym ≠ a)
        return(false);
      else InSym = getInsym();
    else // X - нетерминал B
      B();
      v = z;
  }
  return(true);
}
```

Восстановление процесса анализа после синтаксических ошибок

В приведенных программах рекурсивного спуска была использована процедура реакции на синтаксические ошибки error(). В простейшем случае эта процедура выдает диагностику и завершает работу анали-

затора. Но можно попытаться некоторым разумным образом продолжить работу. Для разбора сверху вниз можно предложить следующий простой алгоритм.

Если в момент обнаружения ошибки на верхушке магазина оказался нетерминальный символ A и для него нет правила, соответствующего входному символу, то сканируем вход до тех пор, пока не встретим символ либо из FIRST(A), либо из FOLLOW(A). В первом случае разворачиваем A по соответствующему правилу, во втором - удаляем A из магазина.

Если на верхушке магазина терминальный символ, то можно удалить все терминальные символы с верхушки магазина вплоть до первого (сверху) нетерминального символа и продолжать так, как это было описано выше.

Разбор снизу-вверх типа сдвиг- свертка

Основа

В процессе разбора снизу-вверх типа сдвиг-свертка строится дерево разбора входной цепочки, начиная с листьев (снизу) к корню (вверх). Этот процесс можно рассматривать как "свертку" цепочки w к начальному символу грамматики. На каждом шаге свертки подцепочка, которую можно сопоставить правой части некоторого правила вывода, заменяется символом левой части этого правила вывода, и если на каждом шаге выбирается правильная подцепочка, то в обратном порядке прослеживается правосторонний вывод (рис. 4.5) Здесь ко входной цепочке, так же как и при анализе LL(1)-грамматик, приписан концевой маркер $.

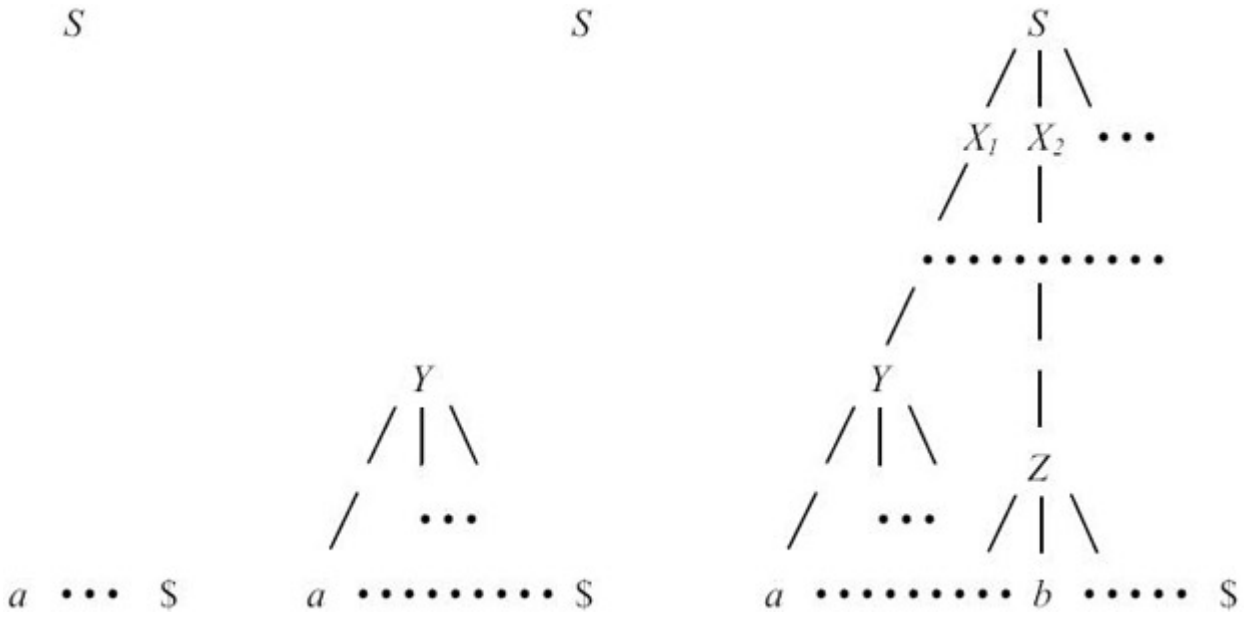


Рис. 4.5.

Основой цепочки называется подцепочка сентенциальной формы, которая может быть сопоставлена правой части некоторого правила вывода, свертка по которому к левой части правила соответствует одному шагу в обращении правостороннего вывода. Самая левая подцепочка, которая сопоставляется правой части некоторого правила вывода A $\rightarrow\gamma$, не обязательно является основой, поскольку свертка по правилу A $\rightarrow\gamma$ может дать цепочку, которая не может быть сведена к аксиоме.

Формально, основа правой сентенциальной формы z - это правило вывода A $\rightarrow\gamma$ и позиция в z, в которой может быть найдена цепочка γ такие, что в результате замены γ на A получается предыдущая сентенциальная форма в правостороннем выводе z. Так, если $S \Rightarrow_r^* \alpha A\beta \Rightarrow_r^* \alpha\gamma\beta$, то A $\rightarrow\gamma$ в позиции, следующей за α, это основа цепочки $\alpha\gamma\beta$. Подцепочка β справа от основы содержит только терминальные символы.

Вообще говоря, грамматика может быть неоднозначной, поэтому не единственным может быть правосторонний вывод $\alpha\gamma\beta$ и не единственной может быть основа. Если грамматика однозначна, то каждая правая сентенциальная форма грамматики имеет в точности одну основу. Замена основы в сентенциальной форме на нетерминал левой части называется отсечением основы. Обращение правостороннего вывода может быть получено с помощью повторного применения отсечения основы, начиная с исходной цепочки w. Если w - слово в рассматриваемой грамматике, то $w = \alpha_n$, где α_n - n-я правая сентенциальная форма еще неизвестного правого вывода $S = \alpha_0 \Rightarrow_r \alpha_1 \Rightarrow_r \ldots \Rightarrow_r \alpha_{n-1} \Rightarrow_r \alpha_n = w$.

Чтобы восстановить этот вывод в обратном порядке, выделяем основу γ_n в α_n и заменяем γ_n на левую часть некоторого правила вывода $A_n \rightarrow \gamma_n$, получая (n - 1)-ю правую сентенциальную форму α_{n-1}. Затем повторяем этот процесс, то есть выделяем основу γ_{n-1} в α_{n-1} и сворачиваем эту основу, получая правую сентенциальную форму α_{n-2}. Если, повторяя этот процесс, мы получаем правую сентенциальную форму, состоящую только из начального символа S, то останавливаемся и сообщаем об успешном завершении разбора. Обращение последовательности правил, использованных в свертках, есть правый вывод входной строки.

Таким образом, главная задача анализатора типа сдвиг- свертка - это выделение и отсечение основы.

LR(1)-анализаторы

В названии LR(1) символ L указывает на то, что входная цепочка читается слева-направо, R - на то, что строится правый вывод, наконец, 1 указывает на то, что анализатор видит один символ непрочитанной части входной цепочки.

LR(1)-анализ привлекателен по нескольким причинам:

- LR(1)-анализ - наиболее мощный метод анализа без возвратов типа сдвиг-свертка;
- LR(1)-анализ может быть реализован довольно эффективно;
- LR(1)-анализаторы могут быть построены для практически всех конструкций языков программирования;
- класс грамматик, которые могут быть проанализированы LR(1)-методом, строго включает класс грамматик, которые могут быть проанализированы предсказывающими анализаторами (сверху-вниз типа LL(1)).

Схематически структура LR(1)-анализатора изображена на рис. 4.4. Анализатор состоит из входной ленты, выходной ленты, магазина, управляющей программы и таблицы анализа (LR(1)-таблицы), которая имеет две части - функцию действий (Action) и функцию переходов (Goto). Управляющая программа одна и та же для всех LR(1)-анализаторов, разные анализаторы отличаются только таблицами анализа.

Анализатор читает символы на входной ленте по одному за шаг. В процессе анализа используется магазин, в котором хранятся строки вида $S_0X_1S_1X_2S_2 \ldots X_mS_m$ (S_m - верхушка магазина). Каждый X_i - символ грамматики (терминальный или нетерминальный), а S_i - символ состояния.

Заметим, что символы грамматики (либо символы состояний) не обязательно должны размещаться в магазине. Однако, их использование облегчает понимание поведения LR-анализатора.

Элемент функции действий Action[S_m; a_i] для символа состояния S_m и входа $a_i \in T \cup \{\}$, может иметь одно из четырех значений:

1. shift S (сдвиг), где S - символ состояния,
2. reduce $A \rightarrow \gamma$ (свертка по правилу грамматики $A \rightarrow \gamma$),
3. accept (допуск),
4. error (ошибка).

Элемент функции переходов Goto[S_m; A] для символа состояния S_mи входа $A \in N$, может иметь одно из двух значений:

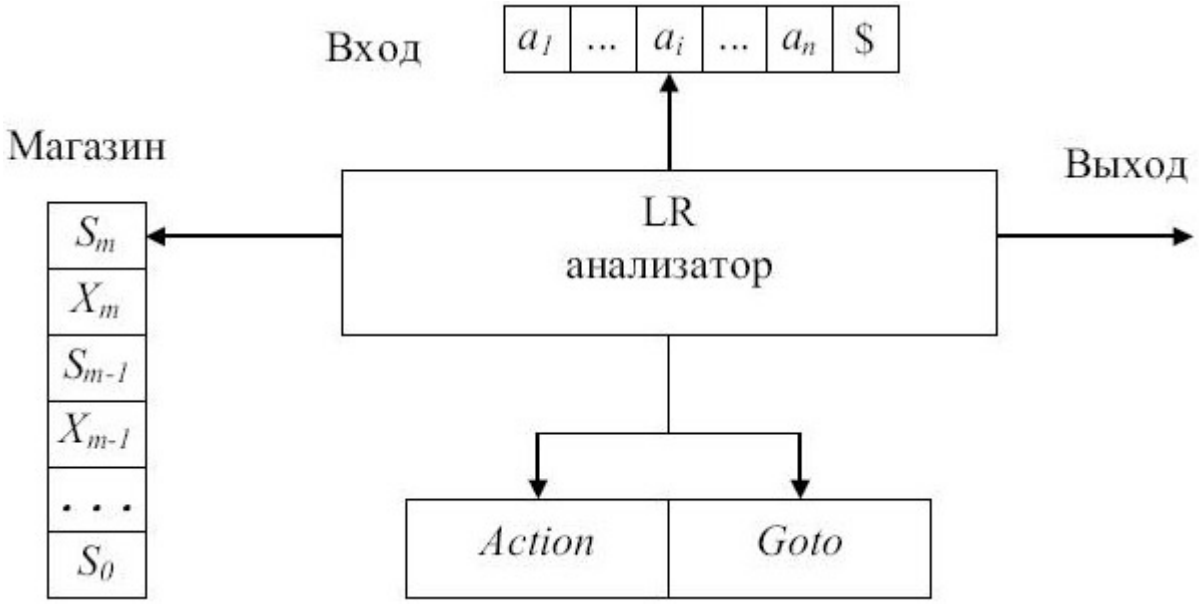


Рис. 4.6.

1. S, где S - символ состояния,
2. error (ошибка).

Конфигурацией LR(1)-анализатора называется пара, первая компонента которой - содержимое магазина, а вторая - непросмотренный вход:

$(S_0X_1S_1X_2S_2 \dots X_mS_m, a_ia_{i+1} \dots a_n\$)$

Эта конфигурация соответствует правой сентенциальной форме

$X_1X_2 \dots X_ma_ia_{i+1} \dots a_n$

Префиксы правых сентенциальных форм, которые могут появиться в магазине анализатора, называются активными префиксами. Основа сентенциальной формы всегда располагается на верхушке магазина. Таким образом, активный префикс - это такой префикс правой сентенциальной формы, который не переходит правую границу основы этой формы.

Когда анализатор начинает работу, в магазине находится только символ начального состояния S_0, на входной ленте - анализируемая цепочка с маркером конца.

Каждый очередной шаг анализатора определяется текущим входным символом a_i и символом состояния на верхушке магазина S_m следующим ниже образом.

Пусть LR(1)-анализатор находится в конфигурации

$(S_0X_1S_1X_2S_2 \dots X_mS_m, a_ia_{i+1} \dots a_n\$)$

Анализатор может проделать один из следующих шагов:

1. Если Action[S_m, a_i] = shift S, то анализатор выполняет сдвиг, переходя в конфигурацию

   $$(S_0X_1S_1X_2S_2 \dots X_mS_ma_iS, a_{i+1} \dots a_n\$)$$

   То есть, в магазин помещаются входной символ a_i и символ состояния S, определяемый Action[S_m, a_i]. Текущим входным символом становится a_{i+1}.

2. Если Action[S_m, a_i] = reduce A ⟶γ, то анализатор выполняет свертку, переходя в конфигурацию

$$(S_0X_1S_1X_2S_2 \ldots X_{m-r}S_{m-r}AS, a_ia_{i+1} \ldots a_n\$)$$

где S = Goto[S_{m-r}; A] и r - длина γ, правой части правила вывода. Анализатор сначала удаляет из магазина 2r символов (r символов состояния и r символов грамматики), так что на верхушке оказывается состояние S_{m-r}. Затем анализатор помещает в магазин A - левую часть правила вывода, и S - символ состояния, определяемый Goto[S_{m-r}, A]. На шаге свертки текущий входной символ не меняется. Для LR(1)- анализаторов последовательность символов грамматики X_{m-r+1} ... Xm, удаляемых из магазина , всегда соответствует γ - правой части правила вывода, по которому делается свертка. После осуществления шага свертки генерируется выход LR(1)-анализатора, то есть исполняются семантические действия, связанные с правилом, по которому делается свертка, например, печатаются номера правил, по которым делается свертка. Заметим, что функция Goto таблицы анализа, построенная по грамматике G, фактически представляет собой функцию переходов детерминированного конечного автомата, распознающего активные префиксы G.

3. Если Action[S_m, a_i] = accept, то разбор успешно завершен.
4. Если Action[S_m, a_i] = error, то анализатор обнаружил ошибку, и выполняются действия по диагностике и восстановлению.

Пример 4.10. Рассмотрим грамматику арифметических выражений G = ({E, T, F}, {id, +, *}, P, E) с правилами:

1. E ⟶E + T
2. E ⟶T
3. T ⟶T * F
4. T ⟶F
5. F ⟶id

На рис. 4.7 изображены функции Action и Goto, образующие LR(1)-таблицу для этой грамматики. Элемент Si функции Action означает сдвиг и помещение в магазин состояния с номером i, Rj - свертку по правилу номер j, acc - допуск, пустая клетка - ошибку. Для функции Goto символ i означает помещение в магазин состояния с номером i, пустая клетка - ошибку.

На входе id + id * id последовательность состояний магазина и входной ленты показаны на рис. 4.8. Например, в первой строке LR-анализатор находится в нулевом состоянии и "видит" первый входной символ id. Действие S6 в нулевой строке и столбце id в поле Action (рис. 4.7) означает сдвиг и помещение символа состояния 6 на верхушку магазина. Это и изображено во второй строке: первый символ id и символ состояния 6 помещаются в магазин, а id удаляется со входной ленты.

Текущим входным символом становится +, и действием в состоянии 6 на вход + является свертка по F ⟶id. Из магазина удаляются два символа (один символ состояния и один символ грамматики). Затем анализируется нулевое состояние. Поскольку Goto в нулевом состоянии по символу F - это 3, F и 3 помещаются в магазин. Теперь имеем конфигурацию, соответствующую третьей строке. Остальные шаги определяются аналогично.

Конструирование LR(1)-таблицы

Рассмотрим алгоритм конструирования таблицы, управляющей LR(1) - анализатором.

Пусть G = (N, T, P, S) - КС-грамматика. Пополненной грамматикой для данной грамматики G называется КС- Состояния Action Goto

Состояния	*Action*				*Goto*		
	id	+	*	$	E	T	F
0	S6				1	2	3
1		S4		acc			
2		R2	S7	R2			
3		R4	R4	R4			
4	S6					5	3
5		R1	S7	R1			
6		R5	R5	R5			
7	S6						8
8		R3	R3	R3			

Рис. 4.7.

Активный префикс	Магазин	Вход	Действие
	0	$id + id * id\$$	сдвиг
id	0 id 6	$+id * id\$$	$F \to id$
F	0 F 3	$+id * id\$$	$T \to F$
T	0 T 2	$+id * id\$$	$E \to T$
E	0 E 1	$+id * id\$$	сдвиг
$E+$	0 E 1 + 4	$id * id\$$	сдвиг
$E + id$	0 E 1 + 4 id 6	$*id\$$	$F \to id$
$E + F$	0 E 1 + 4 F 3	$*id\$$	$T \to F$
$E + T$	0 E 1 + 4 T 5	$id\$$	сдвиг
$E + T*$	0 E 1 + 4 T 5 * 7	$id\$$	сдвиг
$E + T * id$	0 E 1 + 4 T 5 * 7 id 6	$\$$	$F \to id$
$E + T * F$	0 E 1 + 4 T 5 * 7 F 8	$\$$	$T \to T * F$
$E + T$	0 E 1 + 4 T 5	$\$$	$E \to E + T$
E	0 E 1		допуск

Рис. 4.8.

грамматика

$$G' = (N \cup \{S'\}, T, P \cup \{S' \to S\}, S');$$

то есть эквивалентная грамматика, в которой введен новый начальный символ S' и новое правило вывода S' →S.

Это дополнительное правило вводится для того, чтобы определить, когда анализатор должен остановить разбор и зафиксировать допуск входа. Таким образом, допуск имеет место тогда и только тогда, когда анализатор готов осуществить свертку по правилу S' →S.

LR(1)-ситуацией называется пара [A →α.β, a], где A →αβ - правило грамматики, a - терминал или правый концевой маркер $. Вторая компонента ситуации называется аванцепочкой.

Будем говорить, что LR(1)-ситуация [A →α.β, a] допустима для активного префикса +, если существует вывод $S \Rightarrow_r^* \gamma Aw \Rightarrow_r \gamma\alpha\beta w$, где + = γα и либо a - первый символ w, либо w = e и a = $.

Будем говорить, что ситуация допустима, если она допустима для какого-либо активного префикса.

Пример 4.11. Дана грамматика G = ({S, B}, {a, b}, P, S) с правилами

S →BB

B →aB | b

Существует правосторонний вывод $S \Rightarrow_r^* aaBab \Rightarrow_r aaaBab$. Легко видеть, что ситуация [B →a.B, a] допустима для активного префикса + = aaa, если в определении выше положить γ = aa, A = B, w = ab, α= a, β = B. Существует также правосторонний вывод $S \Rightarrow_r^* BaB \Rightarrow_r BaaB$. Поэтому для активного префикса Baa допустима ситуация [B →a.B, $].

Центральная идея метода заключается в том, что по грамматике строится детерминированный конечный автомат, распознающий активные префиксы. Для этого ситуации группируются во множества, которые и образуют состояния автомата. Ситуации можно рассматривать как состояния недетерминированного конечного автомата, распознающего активные префиксы, а их группировка на самом деле есть процесс построения детерминированного конечного автомата из недетерминированного.

Анализатор, работающий слева-направо по типу сдвиг- свертка, должен уметь распознавать основы на верхушке магазина. Состояние автомата после прочтения содержимого магазина и текущий входной символ определяют очередное действие автомата. Функцией переходов этого конечного автомата является функция переходов LR-анализатора.

Чтобы не просматривать магазин на каждом шаге анализа, на верхушке магазина всегда хранится то состояние, в котором должен оказаться этот конечный автомат после того, как он прочитал символы грамматики в магазине от дна к верхушке. Рассмотрим ситуацию вида [A →α.Bβ, a] из множества ситуаций, допустимых для некоторого активного префикса z. Тогда существует правосторонний вывод $S \Rightarrow_r^* yAax \Rightarrow_r y\alpha B\beta ax$, где z = yα. Предположим, что из βax выводится терминальная строка bw. Тогда для некоторого правила вывода вида B →q имеется вывод $S \Rightarrow_r^* zBbw \Rightarrow_r zqbw$. Таким образом [B →.q, b] также допустима для z и ситуация [A →αB.β, a] допустима для активного префикса zB. Здесь либо b может быть первым терминалом, выводимым из β, либо из β выводится e в выводе $\beta ax \Rightarrow_r^* bw$ и тогда b равно a. То есть b принадлежит FIRST(β ax). Построение всех таких ситуаций для данного множества ситуаций, то есть его замыкание, делает приведенная ниже функция closure.

Система множеств допустимых LR(1)-ситуаций для всевозможных активных префиксов пополненной грамматики называется канонической системой множеств допустимых LR(1)-ситуаций. Алгоритм построения канонической системы множеств приведен ниже.

Алгоритм 4.10. Конструирование канонической системы множеств допустимых LR(1)-ситуаций.

Вход. КС-грамматика G = (N, T, P, S).

Выход. Каноническая система C множеств допустимых LR(1)-ситуаций для грамматики G.

Метод. Выполнить для пополненной грамматики G' процедуру items, которая использует функции closure и goto.

```
function closure(I){ /* I - множество ситуаций */
 do{
  for (каждой ситуации [A ⟶ α.Bβ, a] из I,
  каждого правила вывода B ⟶ γ из G',
  каждого терминала b из FIRST(βa),
  такого, что [B ⟶.γ, b] нет в I)
  добавить [B ⟶.γ, b] к I;
 }
 while (к I можно добавить новую ситуацию);
 return I;
}

function goto(I,X){ /* I - множество ситуаций;
                    X - символ грамматики */
  Пусть J = {[A ⟶ αX.β; a] | [A ⟶ α.Xβ, a] \in I};
  return closure(J);
 }

procedure items(G'){ /* G' - пополненная
                грамматика */
  I' = closure({[S' ⟶ .S, $]});
  C = {I0};
  do{
    for (каждого множества ситуаций I из
     системы C, каждого символа грамматики X
      такого, что goto(I, X) не пусто
     и не принадлежит C)
     добавить goto(I, X) к системе C;
   }
   while (к C можно добавить новое множество
         ситуаций);
```

Если I - множество ситуаций, допустимых для некоторого активного префикса +, то goto(I, X) - множество ситуаций, допустимых для активного префикса +X.

Работа алгоритма построения системы C множеств допустимых LR(1)-ситуаций начинается с того, что в C помещается начальное множество ситуаций I_0 = closure({[S' ⟶.S, $]}). Затем с помощью функции goto вычисляются новые множества ситуаций и включаются в C.

По-существу, goto(I, X) - переход конечного автомата из состояния I по символу X.

Рассмотрим теперь, как по системе множеств LR(1)-ситу- аций строится LR(1)-таблица, то есть функции действий и переходов LR(1)-анализатора.

Алгоритм 4.11. Построение LR(1)-таблицы.

Вход. Каноническая система C = {I_0, I_1, ... , I_n} множеств допустимых LR(1)-ситуаций для грамматики G.

Выход. Функции Action и Goto, составляющие LR(1)- таблицу для грамматики G.

Метод. Для каждого состояния i функции Action[i, a] и Goto[i, X] строятся по множеству ситуаций I_i:

1. Значения функции действия (Action) для состояния i определяются следующим образом:
   1. если $[A \to \alpha.a\beta, b] \in I_i$ (a - терминал) и goto(I_i, a)= I_j , то полагаем Action[i, a] = shift j;
   2. если $[A \to \alpha;., a] \in I_i$, причем A≠S', то полагаем Action[i, a] = reduce A $\to\alpha$;
   3. если $[S' \to S., \$] \in I_i$, то полагаем Action[i, $] = accept.
2. Значения функции переходов для состояния i определяются следующим образом: если goto(I_i, A) = I_j , то Goto[i, A] = j (здесь A - нетерминал).
3. Все входы в Action и Goto, не определенные шагами 2 и 3, полагаем равными error.
4. Начальное состояние анализатора строится из множества, содержащего ситуацию [S' ⟶.S, $].

Таблица на основе функций Action и Goto, полученных в результате работы алгоритма 4.11., называется канонической LR(1)-таблицей. Работающий с ней LR(1)-анализатор, называется каноническим LR(1)-анализатором.

Пример 4.12. Рассмотрим следующую грамматику, являющуюся пополненной для грамматики из **примера 4.8.**:

(0) E' ⟶E

(1) E ⟶E + T

(2) E ⟶T

(3) T ⟶T * F

(4) T ⟶F

(5) F ⟶id.

Множества ситуаций и переходы по goto для этой грамматики приведены на рис. 4.9. LR(1)-таблица для этой грамматики приведена на рис. 4.7.

Проследим последовательность создания этих множеств более подробно.

1. Вычисляем I_0 = closure({[E' ⟶.E, $]}).
   1. Ситуация [E' ⟶.E, $] попадает в него по умолчанию как исходная.
   2. Если обратиться к обозначениям функции closure, то для нее

      $$\alpha - \beta - e, \quad B - E, \quad a - \$,$$
      $$first(\beta a) = first(\$) = \{\$\}.$$

      Значит, для терминала $ добавляем ситуации на основе правил со знаком E в левой части правила. Это правила

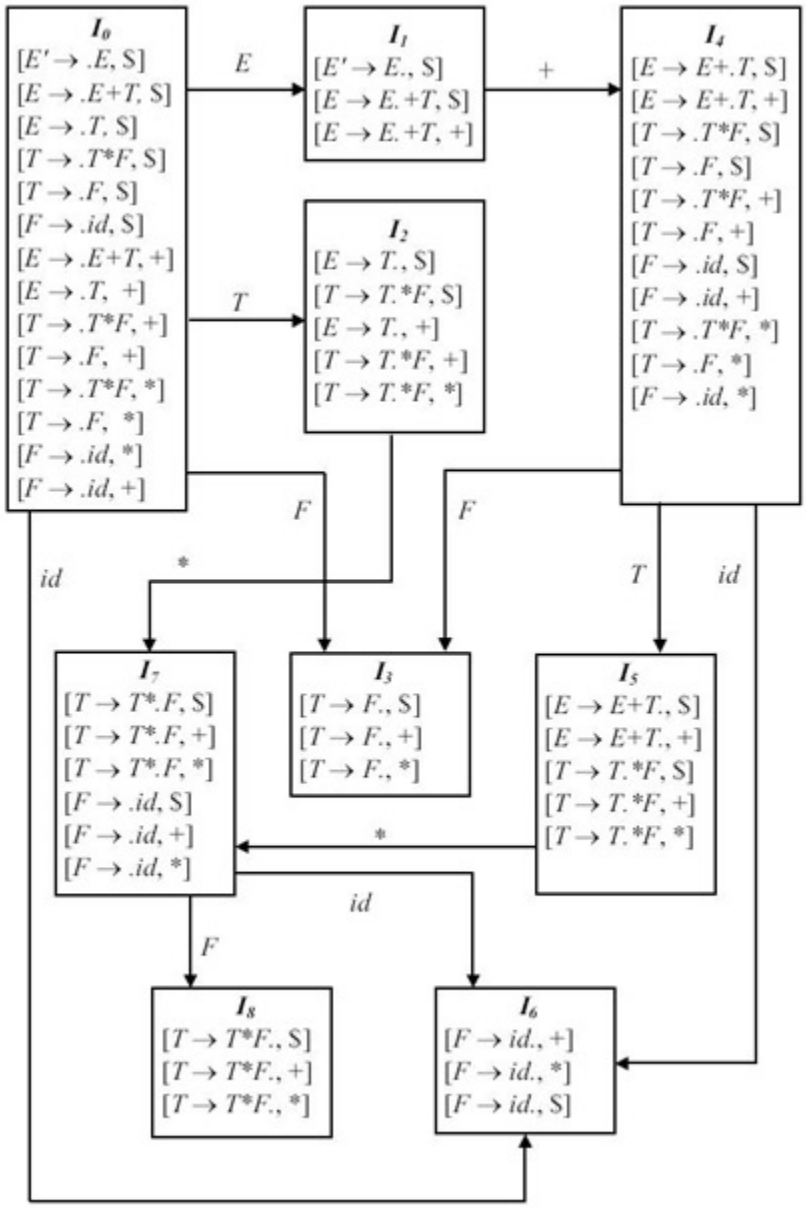
      $$E \to E + T \text{ и } E \to T$$

      и соответствующие им ситуации

      $$[E \to .E + T, \$] \text{ и } [E \to .T, \$]$$

   3. Просматриваем получившиеся ситуации. Для ситуации [E ⟶.E + T, $] β = +, поэтому first(βa) = first(+$) = {+}. На основе этого добавляем к I_0 [E ⟶E + .T, +] и [E ⟶.T, +].
   4. Для ситуации [E ⟶.T, $] β = e, first(βa) = {$}. Поэтому добавляем к I_0 [T ⟶. T * F, $] и [T ⟶.F, $].

5. Подобно этому для ситуации [E ⟶.T, +] β = e, first(βa) = {+}. Поэтому добавляем к I_0 [T ⟶.T * F, +] и [T ⟶.F, +].
6. Из ситуации [T ⟶. T * F, +] β = *, first(βa) = {*}: Поэтому добавляем к I_0



увеличить изображение

Рис. 4.9.

[T ⟶.T * F, *] и [T ⟶.F, *]

7. Далее из ситуации [T ⟶.F, *] получаем ситуацию [F ⟶. id, *]. из ситуации [T ⟶. F, $] - ситуацию [F ⟶. id, $], а из ситуации [T ⟶. F, +] - [F ⟶. id, *].

Таким образом, все 14 искомых ситуаций I_0 получены.

Возвращаемся в головную функцию items, включаем I_0 в множество C и исследуем непустые итоги применения функции goto(I_0; X), где $X \in \{E', E, T, F, +, *, \$, id\}$.

Если посмотреть на вид правил в функции goto(I_0; X), то видно, что X должен встретиться в правой части хотя бы одного правила. Для E0 таких правил у нас нет, поэтому значение функции goto(I_0, E') пусто.

Возьмем goto(I0; E). E встречается после точки в правых частях двух ситуаций из I0, значит берем эти два правила и переносим в них точки на один символ вправо (пока есть куда - не уперлись в запятую), получаем:

[E' →E., $]

и

[E →E. + T, $|+]

Вычислим от каждой из этих ситуаций функцию closure. Но, поскольку справа от точки здесь либо пустая цепочка, либо терминал, то никаких новых ситуаций не возникает. Дальше отслеживаем, может ли куда-то сдвинуться точка дальше на право и по какому символу. Если может, строим соответствующее множество (рис. 4.9). И т.д.

LR(1)-грамматики

Если для КС-грамматики G функция Action, полученная в результате работы алгоритма 4.11., не содержит неоднозначно определенных входов, то грамматика называется LR(1)- грамматикой.

Язык L называется LR(1)-языком, если он может быть порожден некоторой LR(1)-грамматикой.

Иногда используется другое определение LR(1)-грамматики. Грамматика называется LR(1), если из условий

1. $S' \Rightarrow_r^* uAw \Rightarrow_r uvw$
2. $S' \Rightarrow_r^* zBx \Rightarrow_r uvy$
3. FIRST(w)=FIRST(y)

следует, что uAy = zBx (то есть u = z, A = B и x = y).

Согласно этому определению, если uvw и uvy - правовыводимые цепочки пополненной грамматики, у которых FIRST(w) = FIRST(y) и A →β - последнее правило, использованное в правом выводе цепочки uvw, то правило A →β должно применяться и в правом разборе при свертке uvy к uAy. Так как A дает β независимо от w, то LR(1)-условие означает, что в FIRST(w) содержится информация, достаточная для определения того, что uv за один шаг выводится из uA. Поэтому никогда не может возникнуть сомнений относительно того, как свернуть очередную правовыводимую цепочку пополненной грамматики.

Можно доказать, что эти два определения эквивалентны. Дадим теперь определение LR(k)-грамматики.

Определение. Пусть G = (N, Σ, P, S) - КС-грамматика и G' = (N', Σ, P', S') - полученная из нее пополненная грамматика. Будем называть G LR(k)-грамматикой для k ≥0, если из условий

(1) $S' \Rightarrow_{G',r}^* \alpha A\omega' \Rightarrow_{G',r} \alpha\beta\omega'$

(2) $S' \Rightarrow_{G',r}^* \gamma Bx \Rightarrow_{G',r} \alpha\beta y$

(3)$FIRST_k(w) = FIRST_k(y)$

следует, что αAy = γBx (т.е. α= γ, A = B и x = y):

Грамматика G называется LR-грамматикой, если она LR(k)-грамматика для некоторого k ≥0:

Интуитивно это определение говорит о том, что если αβw и αβy - правовыводимые цепочки пополненной грамматики, у которых $FIRST_k(w) = FIRST_k(y)$, и A →β - последнее правило, использованное в правом выводе цепочки αβw, то правило A →β должно использоваться также в правом разборе при свертке αβy к αAy. Так как A дает β независимо от w, то LR(k)-условие говорит о том, что в FIRSTk(w) содержится информация, достаточная для определения того, что αβ за один шаг выводится из αA. Поэтому никогда не может возникнуть сомнений относительно того, как свернуть очередную правовыводимую цепочку пополненной грамматики. Кроме того, работая с LR(k)-грамматикой, мы всегда знаем, допустить ли данную входную цепочку или продолжать разбор.

Пример 4.13. Рассмотрим грамматику G с правилами

S →Sa|a

Согласно определению, G не LR(0)-грамматика, так как из трех условий

(1) $S' \Rightarrow^0_{G'r} S' \Rightarrow_{G'r} S;$

$S' \Rightarrow_{G'r} S \Rightarrow_{G'r} Sa;$

(3)$FIRST_0(e) = FIRST_0(a) = e$

не следует, что S'a = S: Применяя определение к этой ситуации, имеем α= e; β = S; ω = e; γ = e; A = S', B = S; x = e и y = a. Проблема здесь заключается в том, что нельзя установить, является ли S основой правовыводимой цепочки Sa, не видя символа, стоящего после S (т.е. наблюдая "нулевое" количество символов). В соответствии с интуицией G не должна быть LR(0)- грамматикой и она не будет ею, если пользоваться первым определением. Это определение мы и будем использовать далее.

Пример 4.14. Пусть G - леволинейная грамматика с правилами

$$S \to Ab \mid Bc$$
$$A \to Aa \mid e$$
$$B \to Ba \mid e$$

Заметим, что G не является LR(k)-грамматикой ни для какого k.

Допустим, что G - LR(k)-грамматика. Рассмотрим два правых вывода в пополненной грамматике G':

$$S' \Rightarrow_r S \Rightarrow^*_r Aa^k b \Rightarrow_r a^k b$$

и

$$S' \Rightarrow_r S \Rightarrow^*_r Ba^k c \Rightarrow_r a^k c$$

Эти два вывода удовлетворяют условию из определения LR(k)- грамматики при α= e, β = e, ω = a^kb, γ = e и y = a^kc: Но так как заключение неверно, т.е. A ≠B, то G - не LR(k)-грамматика. Более того, так как LR(k)-условие нарушается для всех k, то G - не LR-грамматика.

Если грамматика не является LR(1), то анализатор типа сдвиг-свертка при анализе некоторой цепочки может достигнуть конфигурации, в которой он, зная содержимое магазина и следующий входной символ, не может решить, делать ли сдвиг или свертку (конфликт сдвиг/свертка), или не может решить, какую из нескольких сверток применить (конфликт свертка/свертка).

В частности, неоднозначная грамматика не может быть LR(1). Для доказательства рассмотрим два различных правых вывода

(1) $S \Rightarrow_r u_1 \Rightarrow_r \ldots \Rightarrow_r u_n \Rightarrow_r w$, и

(2) $S \Rightarrow_r v_1 \Rightarrow_r \ldots \Rightarrow_r v_m \Rightarrow_r w$,

Нетрудно заметить, что LR(1) - условие (согласно второму определению LR(1)-грамматики) нарушается для наименьшего из чисел i, для которых $u_{n-i} \neq v_{m-i}$.

Пример 4.15. Рассмотрим еще раз грамматику условных операторов:

$$S \rightarrow if\ E\ then\ S \mid if\ E\ then\ S\ else\ S \mid a$$
$$E \rightarrow b$$

Если анализатор типа сдвиг-свертка находится в конфигурации, такой, что необработанная часть входной цепочки имеет вид else ... $, а в магазине находится ... if E then S, то нельзя определить, является ли if E then S основой, вне зависимости от того, что лежит в магазине ниже. Это конфликт сдвиг/свертка. В зависимости от того, что следует на входе за else, правильной может быть свертка по S →if E then S или сдвиг else, а затем разбор другого S и завершение основы if E then S else S. Таким образом нельзя сказать, нужно ли в этом случае делать сдвиг или свертку, так что грамматика не является LR(1).

Эта грамматика может быть преобразована к LR(1)-виду следующим образом:

$$S \rightarrow M \mid U$$
$$\mathrm{M} \rightarrow \mathrm{if\ E\ then\ M\ else\ M} \mid \mathrm{a}$$
$$\mathrm{U} \rightarrow \mathrm{if\ E\ then\ S} \mid \mathrm{if\ E\ then\ M\ else\ U}$$
$$E \rightarrow b$$

Основная разница между LL(1)- и LR(1)- грамматиками заключается в следующем. Чтобы грамматика была LR(1)- грамматикой, необходимо распознавать вхождение правой части правила вывода, просмотрев все, что выведено из этой правой части и текущий символ входной цепочки. Это требование существенно менее строгое, чем требование для LL(1)-грамматики, когда необходимо определить применимое правило, видя только первый символ, выводимый из его правой части. Таким образом, класс LL(1)-грамматик есть собственный подкласс класса LR(1)-грамматик.

Справедливы также следующие утверждения [2].

Теорема 4.7. Каждый LR(1)-язык является детерминированным КС-языком.

Теорема 4.8. Если L - детерминированный КС-язык, то существует LR(1)-грамматика, порождающая L.

Теорема 4.9. Для любой LR(k)-грамматики для k > 1 существует эквивалентная ей LR(k \Gamma 1)-грамматика.

Доказано, что проблема определения, порождает ли грамматика LR-язык, является алгоритмически неразрешимой.

Восстановление процесса анализа после синтаксических ошибок

Одним из простейших методов восстановления после ошибки при LR(1)-анализе является следующий. При синтаксической ошибке просматриваем магазин от верхушки, пока не найдем состояние s с переходом на выделенный нетерминал A. Затем сканируются входные символы, пока не будет найден такой, который допустим после A. В этом случае на верхушку магазина помещается состояние Goto[s, A] и разбор продолжается. Для нетерминала A может иметься несколько таких вариантов. Обычно A - это нетерминал, представляющий одну из основных конструкций языка, например оператор.

При более детальной проработке реакции на ошибки можно в каждой пустой клетке анализатора поставить обращение к своей подпрограмме. Такая подпрограмма может вставлять или удалять входные символы или символы магазина, менять порядок входных символов.

Варианты LR-анализаторов

Часто построенные таблицы для LR(1)-анализатора оказываются довольно большими. Поэтому при практической реализации используются различные методы их сжатия. С другой стороны, часто оказывается, что при построении для языка синтаксического анализатора типа "сдвиг-свертка" достаточно более простых методов. Некоторые из этих методов базируются на основе LR(1)-анализаторов.

Одним из способов такого упрощения является LR(0)- анализ - частный случая LR-анализа, когда ни при построении таблиц, ни при анализе не учитывается аванцепочка.

Еще одним вариантом LR-анализа являются так называемые SLR(1)-анализаторы (Simple LR(1)). Они строятся следующим образом. Пусть C = {I_0, I_1, ... , I_n} - набор множеств допустимых LR(0)-ситуаций. Состояния анализатора соответствуют I_i. Функции действий и переходов анализатора определяются следующим образом.

1. Если $[A \to u.av] \in I_i$ и $goto(I_i, a) = I_j$, то определим Action[i, a] = shift j.
2. Если $[A \to u.] \in I_i$, то, для всех $a \in FOLLOW(A)$, A≠S', определим Action[i, a] = reduce A ⟶u
3. Если $[S' \to S.] \in I_i$, то определим Action[i, $] = accept.
4. Если goto(I_i, A) = I_j, где $A \in N$, то определим Goto[i, A] = j.
5. Остальные входы для функций Action и Goto определим как error.
6. Начальное состояние соответствует множеству ситуаций, содержащему ситуацию [S' ⟶.S]

Распространенным вариантом LR(1)-анализа является также LALR(1)-анализ. Он основан на объединении (слиянии) некоторых таблиц. Назовем ядром множества LR(1)-ситуаций множество их первых компонент (то есть во множестве ситуаций не учитываются аванцепочки). Объединим все множества ситуаций с одинаковыми ядрами, а в качестве аванцепочек возьмем объединение аванцепочек. Функции Action и Goto строятся очевидным образом. Если функция Action таким образом построенного анализатора не имеет конфликтов, то он называется LALR(1)-анализатором (LookAhead LR(1)).Если грамматика является LR(1), то в таблицах LALR(1) анализатора могут появиться конфликты типы свертка-свертка (если одно из объединяемых состояний имело ситуации [A ⟶α, a] и [B ⟶β, b], а другое [A ⟶α, b] и [B ⟶β a], то в LALR(1) появятся ситуации [A ⟶α, {a, b}] и [B ⟶β, {b, a}]). Конфликты типа сдвиг-свертка появиться не могут, поскольку аванцепочка для сдвига во внимание не принимается.

5. Лекция: Элементы теории перевода

В данной лекции рассматривается теория перевода. Рассматриваются несколько формализмов для определения переводов: преобразователи с магазинной памятью, схемы синтаксически управляемого перевода и атрибутные грамматики. Приведены основные понятия, примеры решения задач и доказательства теорем.

До сих пор мы рассматривали процесс синтаксического анализа только как процесс анализа допустимости входной цепочки. Однако, в компиляторе синтаксический анализ служит основой еще одного важного шага - построения дерева синтаксического анализа. В примерах 4.3 и 4.8 предыдущей главы в процессе синтаксического анализа в качестве выхода выдавалась последовательность примененных правил, на основе которой и может быть построено дерево. Построение дерева синтаксического анализа является простейшим частным случаем перевода - процесса преобразования некоторой входной цепочки в некоторую выходную.

Определение. Пусть T - входной алфавит, а Π- выходной алфавит. Переводом (или трансляцией) с языка $L_1 \subseteq T^*$ на язык $L_2 \subseteq \Pi^*$ называется отображение $\tau : L_1 \rightarrow L_2$. Если $y = \tau(x)$, то цепочка y называется выходом для цепочки x.

Мы рассмотрим несколько формализмов для определения переводов: преобразователи с магазинной памятью, схемы синтаксически управляемого перевода и атрибутные грамматики

Преобразователи с магазинной памятью

Рассмотрим важный класс абстрактных устройств, называемых преобразователями с магазинной памятью. Эти преобразователи получаются из автоматов с магазинной памятью, если к ним добавить выход и позволить на каждом шаге выдавать выходную цепочку.

Преобразователем с магазинной памятью (МП-преоб- разователем) называется восьмерка $P = (Q, T, \Gamma, \Pi, D, q_0, Z_0, F)$, где все символы имеют тот же смысл, что и в определении МП-автомата, за исключением того, что Π- конечный выходной алфавит, а D - отображение множества Q x (T $\cup${e}) x Γ в множество конечных подмножеств множества $Q \times \Gamma^* \times \Pi^*$.

Определим конфигурацию преобразователя P как четверку (q, x, u, y), где $q \in Q$ - состояние, $x \in T^*$ - цепочка на входной ленте, $u \in \Gamma^*$ - содержимое магазина, $y \in \Pi^*$ - цепочка на выходной ленте, выданная вплоть до настоящего момента.

Если множество D(q, a, Z) содержит элемент (r, u, z), то будем писать $(q, ax, Zw, y) \vdash^* (r, x, uw, yz)$ для любых $x \in T^*$, $u \in \Gamma^*$ и $y \in \Pi^*$: Рефлексивно - транзитивное замыкание отношения $\vdash$ будем обозначать $\vdash^*$.

Цепочку y назовем выходом для x, если $(q_0, x, Z_0, e) \vdash^* (q, e, u, y)$ для некоторых $q in F$ и $u \in \Gamma^*$. Переводом (или трансляцией), определяемым МП-преобразователем P (обозначается $\tau(P)$), назовем множество

$$\{(x, y) \mid (q_0, x, Z_0, e) \vdash^* (q, e, u, y)\} \text{ для некоторых } q \in F \text{ и } u \in \Gamma^*$$

Будем говорить, что МП-преобразователь P является детерминированным (ДМП-преобразователем), если выполняются следующие условия:

1. для всех $q \in Q, a \in T \cup \{e\}$ и $Z \in \Gamma$ множество D(q, a, Z) содержит не более одного элемента,
2. если D(q, e, Z) $\neq \varnothing$, то D(q, a, Z) = $\varnothing$ для всех $a \in T$.

Пример 5.1. Рассмотрим перевод τ , отображающий каждую цепочку $x \in \{a, b\}^*\$$, в которой число вхождений символа a равно числу вхождений символа b, в цепочку $y = (ab)^n$, где n - число вхождений a или b в цепочку x. Например, τ (abbaab$) = ababab.

Этот перевод может быть реализован ДМП-преобразователем P = ({q_0, q_f}, {a, b, $}, {Z, a, b}, {a, b}, D, q_0, Z, {q_f}) с функцией переходов:

$$D(q_0, X, Z) = \{(q_0, XZ, e)\}, X \in \{a, b\},$$
$$D(q_0, \$, Z) = \{(q_f, Z, e)\},$$
$$D(q_0, X, X) = \{(q_0, XX, e)\}, X \in \{a, b\},$$
$$D(q_0, X, Y) = \{(q_0, e, ab)\}, X \in \{a, b\}, Y \in \{a, b\}, X \neq Y.$$

Синтаксически управляемый перевод

Другим формализмом, используемым для определения переводов, является схема синтаксически управляемого перевода. Фактически, такая схема представляет собой КС- грамматику, в которой к каждому правилу добавлен элемент перевода. Всякий раз, когда правило участвует в выводе входной цепочки, с помощью элемента перевода вычисляется часть выходной цепочки, соответствующая части входной цепочки, порожденной этим правилом.

Схемы синтаксически управляемого перевода

Определение. Схемой синтаксически управляемого перевода (или трансляции, сокращенно: СУ-схемой) называется пятерка Tr = (N, T, П, R, S), где

(1) N - конечное множество нетерминальных символов;

(2) T - конечный входной алфавит;

П - конечный выходной алфавит;

R - конечное множество правил перевода вида

$$A \to u, v$$

где $u \in (N \cup T)^*, v \in (N \cup \Pi)^*$ и вхождения нетерминалов в цепочку v образуют перестановку вхождений нетерминалов в цепочку u, так что каждому вхождению нетерминала B в цепочку u соответствует некоторое вхождение этого же нетерминала в цепочку v; если нетерминал B встречается более одного раза, для указания соответствия используются верхние целочисленные индексы;

(5) S - начальный символ, выделенный нетерминал из N.

Определим выводимую пару в схеме Tr следующим образом:

(1) (S, S) - выводимая пара, в которой символы S соответствуют друг другу;

(2) если (xAy; x'Ay') - выводимая пара, в цепочках которой вхождения A соответствуют друг другу, и A →u, v - правило из R, то (xuy; x'vy') - выводимая пара. Для обозначения такого вывода одной пары из другой будем пользоваться обозначением ⇒: (xAy, x'Ay') ⇒(xuy, x'vy'). Рефлексивно-транзитивное замыкание отношение ⇒обозначим $\Rightarrow^*$.

Переводом τ (Tr), определяемым СУ-схемой Tr, назовем множество пар

$$\{(x, y) \mid (S, S) \Rightarrow^* (x, y),\ x \in T^*, y \in \Pi^*$$

Если через P обозначить множество входных правил вывода всех правил перевода, то G = (N, T, P, S) будет входной грамматикой для Tr.

СУ-схема Tr = (N, T, П, R, S) называется простой, если для каждого правила A →u, v из R соответствующие друг другу вхождения нетерминалов встречаются в u и v в одном и том же порядке.

Перевод, определяемый простой СУ-схемой, называется простым синтаксически управляемым переводом (простым СУ-переводом).

Пример 5.2. Перевод арифметических выражений в ПОЛИЗ (польскую инверсную запись) можно осуществить простой СУ-схемой с правилами

E → E + T	ET+
E → T	T
T → T * F	TF+
T → F	F
F → id	id
F → (E)	E.

Найдем выход схемы для входа id * (id + id). Нетрудно видеть, что существует последовательность шагов вывода

$$(E,E) \Rightarrow (T,T) \Rightarrow (T*F,TF*) \Rightarrow$$
$$\Rightarrow (F*F,FF*) \Rightarrow (id*F,id\,F*) \Rightarrow (id*(E),id\,E*) \Rightarrow$$
$$\Rightarrow (id*(E+T),\ id\,ET+*) \Rightarrow (id*(T+T),id\,TT+*) \Rightarrow$$
$$\Rightarrow (id*(F+T),\ id\,FT+*) \Rightarrow (id*(id+T),id\,idT+*) \Rightarrow$$
$$\Rightarrow (id*(id+F),id\,idF+*) \Rightarrow$$
$$\Rightarrow (id*(id+id),id\,id\,id+*\Rightarrow;$$

переводящая эту цепочку в цепочку id id id + *.

Рассмотрим связь между переводами, определяемыми СУ- схемами и осуществляемыми МП-преобразователями [2].

Теорема 5.1. Пусть P - МП-преобразователь. Существует такая простая СУ-схема Tr, что τ (Tr) = τ (P).

Теорема 5.2. Пусть Tr - простая СУ-схема. Существует такой МП-преобразователь P, что τ (P) = τ (Tr).

Таким образом, класс переводов, определяемых магазинными преобразователями, совпадает с классом простых СУ-переводов.

Рассмотрим теперь связь между СУ-переводами и детерминированными МП-преобразователями, выполняющими нисходящий или восходящий разбор [2].

Теорема 5.3. Пусть Tr = (N, T, П, R, S) - простая СУ- схема, входной грамматикой которой служит LL(1)-грамматика. Тогда перевод $\{x\$, y) \mid (x, y) \in \tau(Tr)\}$ можно осуществить детерминированным МП-преобразователем.

Существуют простые СУ-схемы, имеющие в качестве входных грамматик LR(1)-грамматики и не реализуемые ни на каком ДМП-преобразователе.

Пример 5.3. Рассмотрим простую СУ-схему с правилами

S → Sa,	aSa
S → Sb, bSb	bSb

S →e, e

Входная грамматика является LR(1)-грамматикой, но не существует ДМП-преобразователя, определяющего перевод $\{(x\$, y) \mid (x, y) \in \tau(Tr)\}$.

Назовем СУ-схему Tr = (N, T, П, R, S) постфиксной, если каждое правило из R имеет вид A →u, v, где $v \in N^*\Pi^*$. Иными словами, каждый элемент перевода представляет собой цепочку из нетерминалов, за которыми следует цепочка выходных символов.

Теорема 5.4. Пусть Tr - простая постфиксная СУ-схема, входная грамматика для которой является LR(1). Тогда перевод

$$\{(x\$, y) \mid (x, y) \in \tau(Tr)\}$$

можно осуществить детерминированным МП-преобразователем.

Обобщенные схемы синтаксически управляемого перевода

Расширим определение СУ-схемы, с тем чтобы выполнять более широкий класс переводов. Во-первых, позволим иметь в каждой вершине дерева разбора несколько переводов. Как и в обычной СУ-схеме, каждый перевод зависит от прямых потомков соответствующей вершины дерева. Во- вторых, позволим элементам перевода быть произвольными цепочками выходных символов и символов, представляющих переводы в потомках. Таким образом, символы перевода могут повторяться или вообще отсутствовать.

Определение. Обобщенной схемой синтаксически управляемого перевода (или трансляции, сокращенно: ОСУ-схемой) называется шестерка Tr = (N,T,П,Г,R,S), где все символы имеют тот же смысл, что и для СУ-схемы, за исключением того, что

1. Г- конечное множество символов перевода вида A_i, где $A \in N$ и i - целое число;
2. R - конечное множество правил перевода вида A →u, $A_1 = v_1, \ldots, A_m = v_m$, удовлетворяющих следующим условиям:
    1. $A_j \in \Gamma$ для $1 \le j \le m$,
    2. каждый символ, входящий в $v_1, \ldots, v_m$, либо принадлежит П, либо является $B_k \in \Gamma$, где B входит в u,
    3. если u имеет более одного вхождения символа B, то каждый символ B_k во всех v соотнесен (верхним индексом) с конкретным вхождением B.

A →u называют входным правилом вывода, A_i - переводом нетерминала A, $A_i = v_i$ - элементом перевода, связанным с этим правилом перевода. Если в ОСУ-схеме нет двух правил перевода с одинаковым входным правилом вывода, то ее называют семантически однозначной.

Выход ОСУ-схемы определим снизу вверх. С каждой внутренней вершиной n дерева разбора (во входной грамматике), помеченной A, свяжем одну цепочку для каждого A_i. Эта цепочка называется значением (или переводом) символа A_i в вершине n. Каждое значение вычисляется подстановкой значений символов перевода данного элемента перевода $A_i = v_i$, определенных в прямых потомках вершины n.

Переводом τ(Tr), определяемым ОСУ-схемой Tr, назовем множество {(x, y) | x имеет дерево разбора во входной грамматике для Tr и y - значение выделенного символа перевода S_k в корне этого дерева}.

Пример 5.4. Рассмотрим формальное дифференцирование выражений, включающих константы 0 и 1, переменную x, функции sin и cos , а также операции * и +. Такие выражения порождает грамматика

E →E + T | T

T ⟶T * F | F

F ⟶(E) | sin (E) | cos (E)|x|0|1

Свяжем с каждым из E, T и F два перевода, обозначенных индексом 1 и 2. Индекс 1 указывает на то, что выражение не дифференцировано, 2 - что выражение продифференцировано. Формальная производная - это E_2. Законы дифференцирования таковы:

```
d(f(x) + g(x)) = df(x) + dg(x)
d(f(x) * g(x)) = f(x) * dg(x) + g(x) * df(x)
d sin (f(x)) = cos (f(x)) * df(x)
d cos (f(x)) = -sin (f(x))df(x)
dx = 1
d0 = 0
d1 = 0
```

Эти законы можно реализовать следующей ОСУ-схемой:

```
E ⟶ E + T      E1 = E1 + T1
      E2 = E2 + T2
E ⟶ T          E1 = T1
      E2 = T2
T ⟶ T * F      T1 = T1 * F1
      T2 = T1 * F2 + T2 * F1
T ⟶ F          T1 = F1
      T2 = F2
F ⟶ (E)                F1 = (E1)
      F2 = (E2)
F ⟶ sin (E)    F1 = sin (E1)
      F2 = cos (E1) * (E2)
F ⟶ cos (E)    F1 = cos (E1)
      F2 = - sin (E1) * (E2)
```

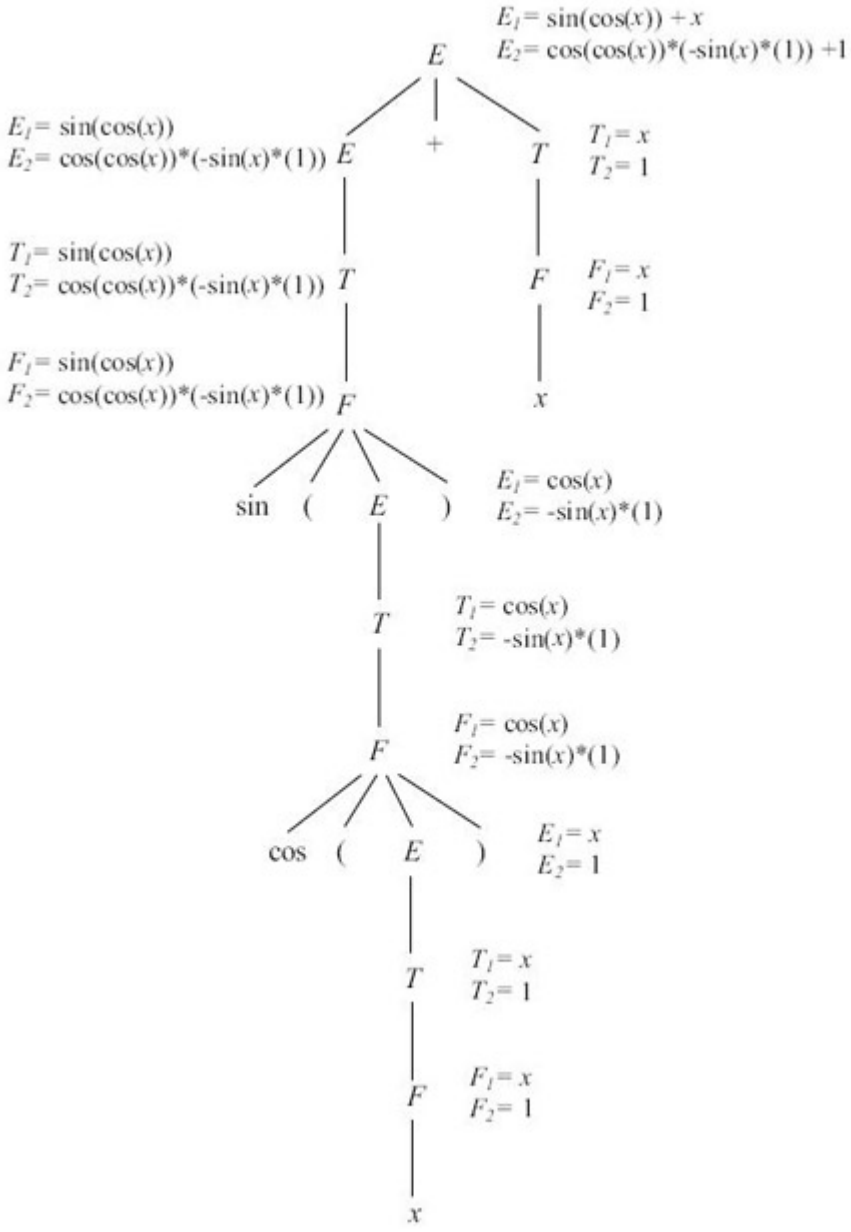


Рис. 5.1.

```
F ⟶ x        F1 = x
    F2 = 1
F ⟶ 0        F1 = 0
    F2 = 0
F ⟶ 1        F1 = 1
    F2 = 0
```

Дерево вывода для sin (cos (x)) + x приведено на рис. 5.1.

Атрибутные грамматики

Среди всех формальных методов описания языков программирования атрибутные грамматики (введенные Кнутом [7]) получили, по-видимому, наибольшую известность и распространение. Причиной этого является то, что формализм атрибутных грамматик основывается на дереве разбора программы в КС-грамматике, что сближает его с хорошо разработанной теорией и практикой построения трансляторов.

Определение атрибутных грамматик

Атрибутной грамматикой называется четверка AG = (G, A_S, A_I, R), где

1. G = (N, T, P, S) - приведенная КС-грамматика;
2. A_S - конечное множество синтезируемых атрибутов;
3. A_I - конечное множество наследуемых атрибутов, $A_S \cap A_I = \varnothing$;
4. R - конечное множество семантических правил.

Атрибутная грамматика AG сопоставляет каждому символу X из $N \cup T$ множество $A_S(X)$ синтезируемых атрибутов и множество $A_I(X)$ наследуемых атрибутов. Множество всех синтезируемых атрибутов всех символов из $N \cup T$ обозначается A_S, наследуемых - A_I . Атрибуты разных символов являются различными атрибутами. Будем обозначать атрибут a символа X как a(X). Значения атрибутов могут быть произвольных типов, например, представлять собой числа, строки, адреса памяти и т.д.

Пусть правило p из P имеет вид $X_0 \rightarrow X_1X_2 \ldots X_n$. Атрибутная грамматика AG сопоставляет каждому правилу p из P конечное множество R(p) семантических правил вида

$a(X_i) = f(b(X_j), c(X_k), \ldots, d(X_m))$

где $0 \le j, k, \ldots, m \le n$, причем $1 \le i \le n$, если $a(X_i) \in A_I(X_i)$(то есть $a(X_i)$ - наследуемый атрибут), и i = 0, если $a(X_i) \in A_S(X_i)$(то есть $a(X_i)$ - синтезируемый атрибут).

Таким образом, семантическое правило определяет значение атрибута a символа X_i на основе значений атрибутов b, c, . . . , d символов X_j , X_k, . . . , X_m соответственно.

В частном случае длина n правой части правила может быть равна нулю, тогда будем говорить, что атрибут a символа X_i "получает в качестве значения константу".

В дальнейшем будем считать, что атрибутная грамматика не содержит семантических правил для вычисления атрибутов терминальных символов. Предполагается, что атрибуты терминальных символов - либо предопределенные константы, либо доступны как результат работы лексического анализатора.

Пример 5.5. Рассмотрим атрибутную грамматику, позволяющую вычислить значение вещественного числа, представленного в десятичной записи. Здесь N = {Num, Int, Frac}, T = {digit, .}, S = Num, а правила вывода и семантические правила определяются следующим образом (верхние индексы используются для ссылки на разные вхождения одного и того же нетерминала):

$$
\begin{array}{ll}
Num \rightarrow Int.Frac & v(Num) = v(Int) + v(Frac) \\
 & p(Frac) = 1 \\
Int \rightarrow e & v(Int) = 0 \\
 & p(Int) = 0 \\
Int^1 \rightarrow digit\ Int^2 & v(Int^1) = v(digit) * 10^{p(Int^2)} + v(Int^2) \\
 & p(Int^1) = p(Int^2) + 1 \\
Frac \rightarrow e & v(Frac) = 0 \\
Frac^1 \rightarrow digit Frac^2 & v(Frac^1) = v(digit) * 10^{-p(Frac^1)} + v(Frac^2) \\
 & p(Frac^2) = p(Frac^1) + 1
\end{array}
$$

Для этой грамматики

$$
\begin{array}{ll}
A_S(Num) = \{v\}, & A_I(Num) = \varnothing, \\
A_S(Int) = \{v, p\}, & A_I(Int) = \varnothing, \\
A_S(Frac) = \{v\}, & A_I(Frac) = \{p\}.
\end{array}
$$

Пусть дана атрибутная грамматика AG и цепочка, принадлежащая языку, определяемому соответствующей G = (N, T, P, S). Сопоставим этой цепочке "значение "следующим образом. Построим дерево разбора T этой цепочки в грамматике G. Каждый внутренний узел этого дерева помечается нетерминалом X_0, соответствующим применению p-го правила грамматики; таким образом, у этого узла будет n непосредственных потомков (рис. 5.2).

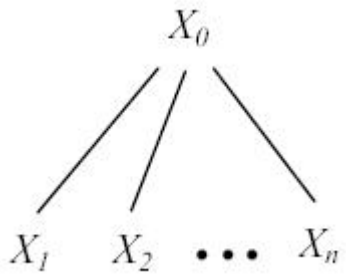


Рис. 5.2.

Пусть теперь X - метка некоторого узла дерева и пусть a - атрибут символа X. Если a - синтезируемый атрибут, то $X = X_0$ для некоторого $p \in P$; если же a - наследуемый атрибут, то $X = X_j$ для некоторых $p \in P$ и $1 \le j \le n$. В обоих случаях дерево "в районе" этого узла имеет вид, приведенный на рис. 5.2. По определению, атрибут a имеет в этом узле значение v, если в соответствующем семантическом правиле

$a(X_i) = f(b(X_j), c(X_k), \ldots, d(X_m))$

все атрибуты b, c, . . . , d уже определены и имеют в узлах с метками X_j, X_k, . . . , X_m значения v_j, v_k, . . . , v_m соответственно, а $v = f(v_1, v_2, \ldots, v_m)$. Процесс вычисления атрибутов на дереве продолжается до тех пор, пока нельзя будет вычислить больше ни одного атрибута. Вычисленные в результате атрибуты корня дерева представляют собой "значение", соответствующее данному дереву вывода.

Заметим, что значение синтезируемого атрибута символа в узле синтаксического дерева вычисляется по атрибутам символов в потомках этого узла; значение наследуемого атрибута вычисляется по атрибутам "родителя" и "соседей".

Атрибуты, сопоставленные вхождениям символов в дерево разбора, будем называть вхождениями атрибутов в дерево разбора, а дерево с сопоставленными каждой вершине атрибутами - атрибутированным деревом разбора.

Пример 5.6. Атрибутированное дерево для грамматики из предыдущего примера и цепочки w = 12:34 показано на рис. 5.3.

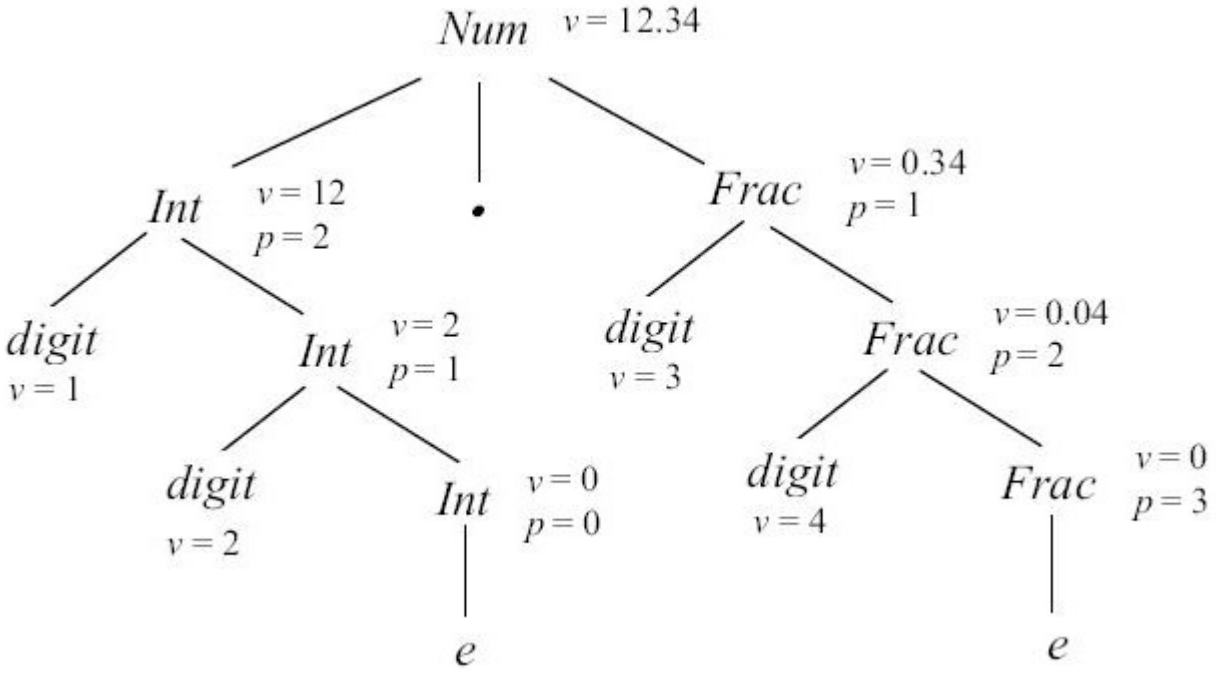


Рис. 5.3.

Будем говорить, что семантические правила заданы корректно, если они позволяют вычислить все атрибуты произвольного узла в любом дереве вывода.

Между вхождениями атрибутов в дерево разбора существуют зависимости, определяемые семантическими правилами, соответствующими примененным синтаксическим правилам. Эти зависимости могут быть представлены в виде ориентированного графа следующим образом.

Пусть T - дерево разбора. Сопоставим этому дереву ориентированный граф D(T), узлами которого являются пары (n; a), где n - узел дерева T, a - атрибут символа, служащего меткой узла n. Граф содержит дугу из (n_1, a_1) в (n_2, a_2) тогда и только тогда, когда семантическое правило, вычисляющее атрибут a_2, непосредственно использует значение атрибута a_1. Таким образом, узлами графа D(T) являются атрибуты, которые нужно вычислить, а дуги определяют зависимости, подразумевающие, какие атрибуты вычисляются раньше, а какие позже.

Пример 5.7. Граф зависимостей атрибутов для дерева разбора из предыдущего примера показан на рис. 5.4.

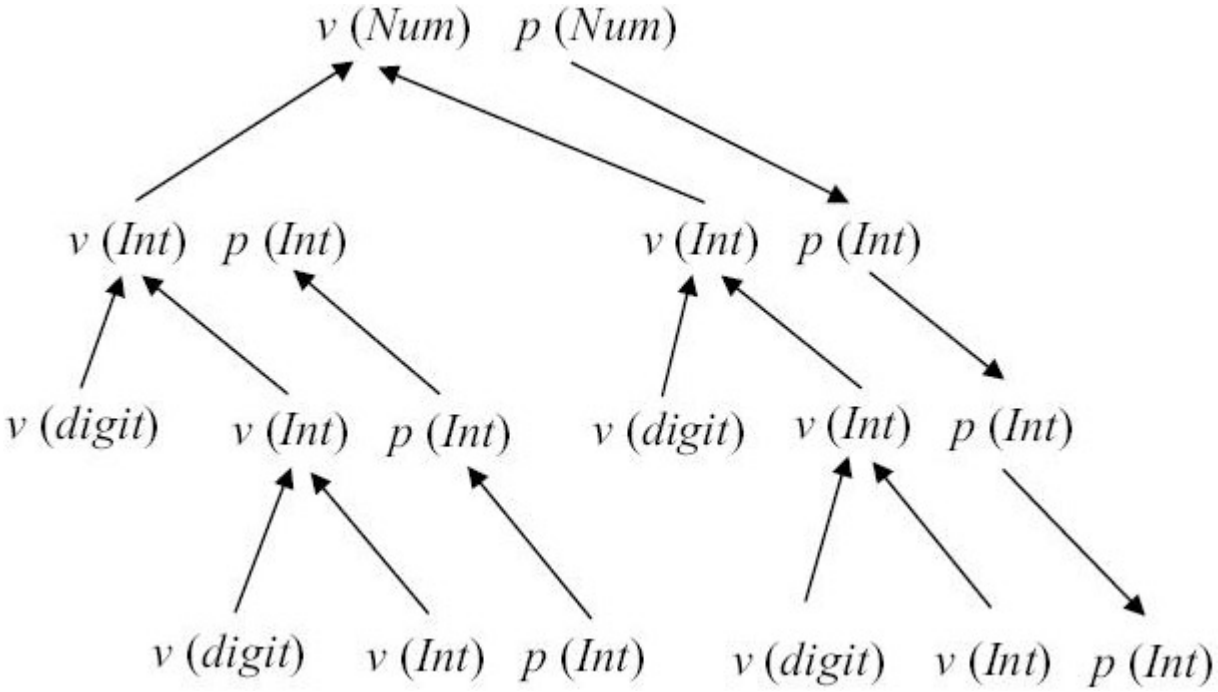


Рис. 5.4.

Можно показать, что семантические правила являются корректными тогда и только тогда, когда для любого дерева вывода T соответствующий граф D(T) не содержит циклов (то есть является ориентированным ациклическим графом).

Классы атрибутных грамматик и их реализация

В общем виде реализация вычислителей для атрибутных грамматик вызывает значительные трудности. Это связано с тем, что множество значений атрибутов, связанных с данным деревом, приходится вычислять в соответствии с зависимостями атрибутов, которые образуют ориентированный ациклический граф. На практике стараются осуществлять процесс вычисления атрибутов, привязывая его к тому или иному способу обхода дерева. Рассматривают многовизитные, многопроходные и другие атрибутные вычислители. Это, как правило, ведет к ограничению допустимых зависимостей между атрибутами, поддерживаемых вычислителем.

Простейшими подклассами атрибутных грамматик, вычисления всех атрибутов для которых может быть осуществлено одновременно с синтаксическим анализом, являются S-атрибутные и L-атрибутные грамматики. Определение. Атрибутная грамматика называется S- атрибутной, если она содержит только синтезируемые атрибуты.

Нетрудно видеть, что для S-атрибутной грамматики на любом дереве разбора все атрибуты могут быть вычислены за один обход дерева снизу вверх. Таким образом, вычисление атрибутов можно делать параллельно с восходящим синтаксическим анализом, например, LR(1)- анализом.

Пример 5.8. Рассмотрим S-атрибутную грамматику для перевода арифметических выражений в **ПОЛИЗ**. Здесь атрибут v имеет строковый тип, k - обозначает операцию конкатенации. Правила вывода и семантические правила определяются следующим образом

$$E^1 \to E^2 + T \qquad v(E^1) = v(E^2) \parallel v(T) \parallel '+'$$
$$E \to T \qquad v(E) = v(T)$$
$$T \to T * F \qquad v(T^1) = v(T^2) \parallel v(F) \parallel '*'$$
$$T \to F \qquad v(T) = v(F)$$
$$F \to id \qquad v(F) = v(id)$$
$$F \to (E) \qquad v(F) = v(E)$$

Определение. Атрибутная грамматика называется L- атрибутной, если любой наследуемый атрибут любого символа X_j из правой части каждого правила $X_0 \to X_1 X_2 \ldots X_n$ грамматики зависит только от

1. атрибутов символов $X_1, X_2, \ldots, X_{j-1}$, находящихся в правиле слева от X_j , и
2. наследуемых атрибутов символа X_0.

Заметим, что каждая S-атрибутная грамматика является L-атрибутной. Все атрибуты на любом дереве для L- атрибутной грамматики могут быть вычислены за один обход дерева сверху-вниз слева-направо. Таким образом, вычисление атрибутов можно осуществлять параллельно с нисходящим синтаксическим анализом, например, LL(1)- анализом или рекурсивным спуском.

В случае рекурсивного спуска в каждой функции, соответствующей нетерминалу, надо определить формальные параметры, передаваемые по значению, для наследуемых атрибутов, и формальные параметры, передаваемые по ссылке, для синтезируемых атрибутов. В качестве примера рассмотрим реализацию атрибутной грамматики из примера 5.5 (нетрудно видеть, что грамматика является L-атрибутной).

```
void int_part(float * V0, int * P0)
{if (Map[InSym]==Digit)
  { int I=InSym;
    float V2;
    int P2;
    InSym=getInSym();
    int_part(&V2,&P2);
    *V0=I*exp(P2*ln(10))+V2;
    *P0=P2+1;
  }
  else {*V0=0;
        *P0=0;
    }
 }
void fract_part(float * V0, int P0)
{if (Map[InSym]==Digit)
  { int I=InSym;
    float V2;
    int P2=P0+1;
    InSym=getInSym();
    fract_part(&V2,P2);
    *V0=I*exp(-P0*ln(10))+V2;
  }
  else {*V0=0;
    }
}
void number()
{ float V1,V3,V0;
  int P;
    int_part(&V1,&P);
    if (InSym!='.') error();
    fract_part(&V3,1);
    V0=V1+V3;
}
```

Язык описания атрибутных грамматик

Формализм атрибутных грамматик оказался очень удобным средством для описания семантики языков программирования. Вместе с тем выяснилось, что реализация вычислителей для атрибутных грамматик общего вида сталкивается с большими трудностями. В связи с этим было сделано множество попыток рассматривать те или иные классы атрибутных грамматик, обладающих "хорошими" свойствами. К числу таких свойств относятся прежде всего простота алгоритма проверки атрибутной грамматики на зацикленность и простота алгоритма вычисления атрибутов для атрибутных грамматик данного класса.

Атрибутные грамматики использовались для описания семантики языков программирования и было создано несколько систем автоматизации разработки трансляторов, основанных на формализме атрибутных грамматик. Опыт их использования показал, что "чистый" атрибутный формализм может быть успешно применен для описания семантики языка, но его использование вызывает трудности при создании транслятора. Эти трудности связаны как с самим формализмом, так и с некоторыми технологическими проблемами. К трудностям первого рода можно отнести несоответствие чисто функциональной природы атрибутного вычислителя и связанной с ней неупорядоченностью процесса вычисления атрибутов (что в значительной степени является преимуществом этого формализма) и упорядоченностью элементов программы. Это несоответствие ведет к тому, что приходится идти на искусственные приемы для их сочетания. Технологические трудности связаны с эффективностью трансляторов, полученных с помощью атрибутных систем. Как правило, качество таких трансляторов довольно низко из-за больших расходов памяти, неэффективности искусственных приемов, о которых было сказано выше.

Учитывая это, мы будем вести дальнейшее изложение на языке, сочетающем особенности атрибутного формализма и обычного языка программирования, в котором предполагается наличие операторов, а значит, и возможность управления порядком исполнения операторов. Этот порядок может быть привязан к обходу атрибутированного дерева разбора сверху вниз слева направо. Что касается грамматики входного языка, то мы не будем предполагать принадлежность ее определенному классу (например, LL(1) или LR(1)). Будем считать, что дерево разбора входной программы уже построено как результат синтаксического анализа и атрибутные вычисления осуществляются в результате обхода этого дерева. Таким образом, входная грамматика атрибутного вычислителя может быть даже неоднозначной, что не влияет на процесс атрибутных вычислений.

При записи синтаксиса мы будем использовать расширенную БНФ. Элемент правой части синтаксического правила, заключенный в скобки [ ], может отсутствовать. Элемент правой части синтаксического правила, заключенный в скобки ( ), означает возможность повторения один или более раз. Элемент правой части синтаксического правила, заключенный в скобки [()], означает возможность повторения ноль или более раз. В скобках [ ] или [()] может указываться разделитель конструкций.

Ниже дан синтаксис языка описания атрибутных грамматик. Приведен только синтаксис конструкций, собственно описывающих атрибутные вычисления. Синтаксис обычных выражений и операторов не приводится - он основывается на Си.

```
Атрибутная грамматика::='ALPHABET'
        ( ОписаниеНетерминала ) ( Правило )
ОписаниеНетерминала::=ИмяНетерминала
            '::' [( ОписаниеАтрибутов / ';')]'.'
ОписаниеАтрибутов::=Тип ( ИмяАтрибута / ',')
Правило::='RULE' Синтаксис 'SEMANTICS' Семантика'.'
Синтаксис::=ИмяНетерминала '::=' ПраваяЧасть
ПраваяЧасть::=[( ЭлементПравойЧасти )]
ЭлементПравойЧасти::=ИмяНетерминала
        | Терминал
        | '(' Нетерминал [ '/' Терминал ] ')'
        | '[' Нетерминал ']'
        | '[(' Нетерминал [ '/' Терминал ] ')]'
Семантика::=[(ЛокальноеОбъявление / ';')]
        [( СемантическоеДействие / ';')]
СемантическоеДействие::=Присваивание
        | [ Метка ] Оператор
Присваивание::=Переменная ':=' Выражение
Переменная::=ЛокальнаяПеременная
```

```
      | Атрибут
Атрибут::=ЛокальныйАтрибут
    | ГлобальныйАтрибут
ЛокальныйАтрибут::=ИмяАтрибута '<' Номер '>'
ГлобальныйАтрибут::=ИмяАтрибута '<' Нетерминал '>'
Метка::=Целое ':'
  | Целое 'E' ':'
  | Целое 'A' ':'
Оператор::=Условный
    | ОператорПроцедуры
    | ЦиклПоМножеству
    | ПростойЦикл
    | ЦиклСУсловиемОкончания
```

Описание атрибутной грамматики состоит из раздела описания атрибутов и раздела правил. Раздел описания атрибутов определяет состав атрибутов для каждого символа грамматики и тип каждого атрибута. Правила состоят из синтаксической и семантической части. В синтаксической части используется расширенная БНФ. Семантическая часть правила состоит из локальных объявлений и семантических действий. В качестве семантических действий допускаются как атрибутные присваивания, так и составные операторы.

Метка в семантической части правила привязывает выполнение оператора к обходу дерева разбора сверху- вниз слева направо. Конструкция i : оператор означает, что оператор должен быть выполнен сразу после обхода i- й компоненты правой части. Конструкция i E : оператор означает, что оператор должен быть выполнен, только если порождение i-й компоненты правой части пусто. Конструкция i A : оператор означает, что оператор должен быть выполнен после разбора каждого повторения i-й компоненты правой части (имеется в виду конструкция повторения).

Каждое правило может иметь локальные определения (типов и переменных). В формулах используются как атрибуты символов данного правила (локальные атрибуты) и в этом случае соответствующие символы указываются номерами в правиле (0 - для символа левой части, 1 - для первого символа правой части, 2 - для второго символа правой части и т.д.), так и атрибуты символов предков левой части правила (глобальные атрибуты). В этом случае соответствующий символ указывается именем нетерминала. Таким образом, на дереве образуются области видимости атрибутов: атрибут символа имеет область видимости, состоящую из правила, в которое символ входит в правую часть, плюс все поддерево, корнем которого является символ, за исключением поддеревьев - потомков того же символа в этом поддереве.

Значение терминального символа доступно через атрибут VAL соответствующего типа.

Пример 5.9. Атрибутная грамматика из **примера 5.5** записывается следующим образом:

```
ALPHABET
Num :: float V.
Int :: float V;
        int P.
Frac :: float V;
        int P.
digit :: int VAL.

RULE
Num ::= Int '.' Frac
SEMANTICS
V<0>=V<1>+V<3>; P<3>=1.

RULE
Int ::= e
SEMANTICS
V<0>=0; P<0>=0.

RULE
Int ::= digit Int
SEMANTICS
V<0>=VAL<1>*10**P<2>+V<2>; P<0>=P<2>+1.
```

```
RULE
Frac ::= e
SEMANTICS
V<0>=0.

RULE
Frac ::= digit Frac
SEMANTICS
V<0>=VAL<1>*10**(-P<0>)+V<2>; P<2>=P<0>+1.
```

6. Лекция: Проверка контекстных условий

В данной лекции рассматривается проверка контекстных условий. Приведены определения понятий компоненты программы, области действия и области видимости, основных действий со средой. Также приведены примеры программного кода и решения задач.

Описание областей видимости и блочной структуры

Задачей контекстного анализа является установление свойств объектов и их использования. Наиболее часто решаемой задачей является определение существования объекта и соответствия его использования контексту, что осуществляется с помощью анализа типа объекта. Под контекстом здесь понимается вся совокупность свойств текущей точки программы, например множество доступных объектов, тип выражения и т.д.

Таким образом, необходимо хранить объекты и их типы, уметь находить эти объекты и определять их типы, определять характеристики контекста. Совокупность доступных в данной точке объектов будем называть средой. Обычно среда программы состоит из частично упорядоченного набора компонент

$E = \{DS_1, DS_2, \dots, DS_n\}$

Каждая компонента - это множество объявлений,

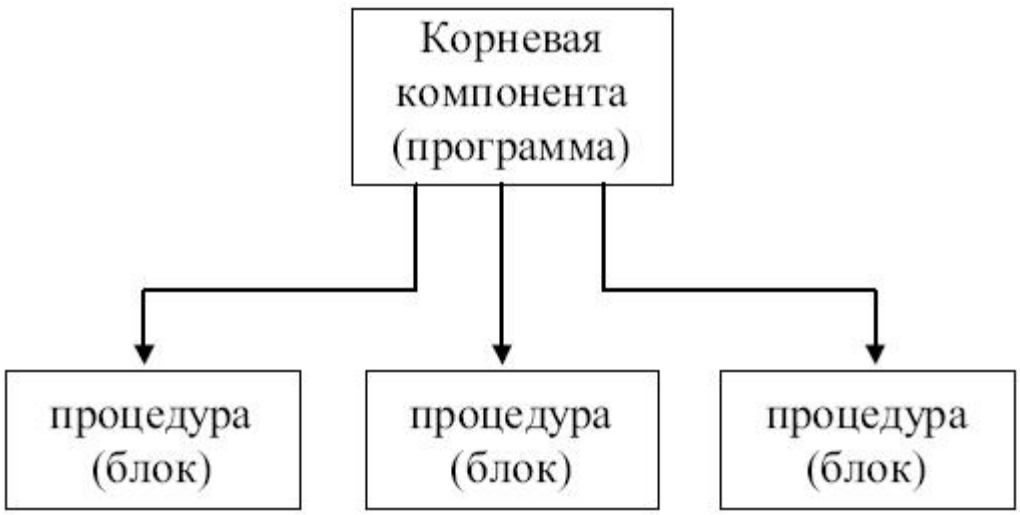


Рис. 6.1.

представляющих собой пары (имя, тип):

$DS_i = \{(имя_j, тип_j) \mid 1 \leq j \leq k_i\}$

где под типом будем подразумевать полное описание свойств объекта (объектом, в частности, может быть само описание типа).

Компоненты образуют дерево, соответствующее этому частичному порядку. Частичный порядок между компонентами обычно определяется статической вложенностью компонент в программе. Эта вложенность может соответствовать блокам, процедурам или классам программы (рис. 6.1). Компоненты среды могут быть именованы. Поиск в среде обычно ведется с учетом упорядоченности компонент. Среда может включать в себя как компоненты, полученные при трансляции "текущего" текста программы, так и "внешние" (например, раздельно компилированные) компоненты.

Для обозначения участков программы, в которых доступны те или иные описания, используются понятия области действия и области видимости. Областью действия описания является процедура (блок), содержащая описание, со всеми входящими в нее (подчиненными по дереву) процедурами (блоками). Областью видимости описания называется часть области действия, из которой исключены те подобласти, в которых по тем или иным причинам описание недоступно, например, оно перекрыто другим описанием. В разных языках понятия области действия и области видимости уточняются по-разному.

Обычными операциями при работе со средой являются:

- включить объект в компоненту среды;
- найти объект в среде и получить доступ к его описанию;
- образовать в среде новую компоненту, определенным образом связанную с остальными;
- удалить компоненту из среды.

Среда состоит из отдельных объектов, реализуемых как записи (в дальнейшем описании мы будем использовать имя TElement для имени типа этой записи). Состав полей записи, вообще говоря, зависит от описываемого объекта (тип, переменная и т.д.), но есть поля, входящие в запись для любого объекта:

TObject Object - категория объекта (тип, переменная, процедура и т.д.);

TMode Mode - вид объекта: целый, массив, запись и т.д.;

TName Name - имя объекта;

TType Type - указатель на описание типа.

Занесение в среду и поиск объектов

Рассмотрим схему реализации простой блочной структуры, аналогичной процедурам в Паскале или блокам в Си. Каждый блок может иметь свой набор описаний. Программа состоит из основного именованного блока, в котором имеются описания и операторы. Описания состоят из описаний типов и объявлений переменных. В качестве типа может использоваться целочисленный тип и тип массива. Два типа T1 и T2 считаются эквивалентными, если имеется описание T1=T2 (или T2=T1). Операторами служат операторы присваивания вида Переменная1=Переменная2 и блоки. Переменная - это либо просто идентификатор, либо выборка из массива. Оператор присваивания считается правильным, если типы переменных левой и правой части эквивалентны.

Примером правильной программы может служить

```
program Example
begin
  type  T1=array 100 of array 200 of integer;
      T2=T1;
  var   V1:T1;
      V2:T2;
  begin
   V1=V2;
   V2[1]=V1[2];
   begin
```

```
      type T3=array 300 of T1;
      var V3:T3;
      V3[50]=V1;
    end
  end
end.
```

Рассматриваемое подмножество языка может быть порождено следующей грамматикой (запись в расширенной БНФ):

```
Prog::='program' Ident Block '.'
Block::='begin' [(Declaration)] [(Statement)] 'end'
Declaration::='type' (Type_Decl)
Type_Decl::=Ident '=' Type_Defin
Type_Defin::='ARRAY' Index 'OF' Type_Defin
Type_Defin::=Type_Use
Type_Use::=Ident
Declaration::='var' (Var_Decl)
Var_Decl::=Ident_List ':' Type_Use ';'
Ident_List::=(Ident / ',')
Statement::=Block ';'
Statement::=Variable '=' Variable ';'
Variable::=Ident Access
Access::='[' Expression ']' Access
Access::=
```

Для реализации некоторых атрибутов (в частности среды, списка идентификаторов и т.д.) в качестве типов данных мы будем использовать различные множества. Множество может быть упорядоченным или неупорядоченным, ключевым или простым. Элементом ключевого множества может быть запись, одним из полей которой является ключ:

- SETOF T - простое неупорядоченное множество объектов типа T;
- KEY K SETOF T - ключевое неупорядоченное множество объектов типа T с ключом типа K;
- LISTOF T - простое упорядоченное множество объектов типа T;
- KEY K LISTOF T - ключевое упорядоченное множество объектов типа T с ключом типа K;

Над объектами типа множества определены следующие операции:

- Init(S) - создать и проинициализировать переменную S;
- Include(V,S) - включить объект V в множество S; если множество упорядоченное, то включение осуществляется в качестве последнего элемента;
- Find(K,S) - выдать указатель на объект с ключом K во множестве S и NIL, если объект с таким ключом не найден.

Имеется специальный оператор цикла, пробегающий элементы множества:

for (V in S) Оператор;

Переменная V пробегает все значения множества. Если множество упорядочено, то элементы пробегаются в этом порядке, если нет - в произвольном порядке.

Среда представляет собой ключевое множество с ключом - именем объекта. Идентификаторы имеют тип TName. Обозначение <Нетерминал> в позиции типа - это указатель на вершину типа Нетерминал. Обозначение <Нетерминал> в выражении - это взятие значения указателя

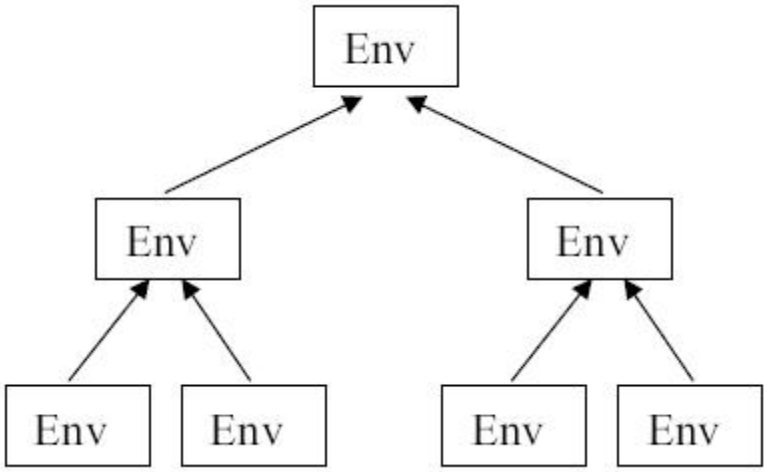


Рис. 6.2.

на ближайшую вершину вверх по дереву разбора, помеченную соответствующим нетерминалом.

Для реализации среды каждый нетерминал Block имеет атрибут Env. Для обеспечения возможности просматривать компоненты среды в соответствии с вложенностью блоков каждый нетерминал Block имеет атрибут Pred - указатель на охватывающий блок. Кроме того, среда блока корня дерева (нетерминал Prog) содержит все предопределенные описания (рис. 6.2). Это заполнение реализуется процедурой PreDefine. Атрибут Pred блока корневой компоненты имеет значение NULL.

Атрибутная реализация выглядит следующим образом.

Листинг 6.1. (html, txt)

Переменная BlockPointer - указатель на ближайший охватывающий блок. Переходя от блока к блоку, ищем объект в его среде. Если не нашли, то переходим к охватывающему блоку. Если дошли до корневой компоненты, пытаемся найти объект среди предопределенных объектов. Если объект нашли, надо убедиться, что он имеет нужную категорию.

Функция ArrayElementType(TType EntryType, TType ExprType) осуществляет проверку допустимости применения операции взятия индекса к переменной и возвращает тип элемента массива.

Функция ArrayType(TType EntryType, int Val) возвращает описание типа - массива с типом элемента EntryType и диапазоном индекса Val.

7. Лекция: Организация таблиц символов

В данной лекции рассматривается организация таблиц символов. Рассматриваются некоторые основные способы организации таблиц символов в компиляторе: таблицы идентификаторов, таблицы расстановки, двоичные деревья и реализация блочной структуры. Приведены также примеры программного кода и графическая интерпретация таблиц символов и идентификаторов.

В процессе работы компилятор хранит информацию об объектах программы в специальных таблицах символов. Как правило, информация о каждом объекте состоит из двух основных элементов: имени объекта и описания объекта. Информация об объектах программы должна быть организована таким образом, чтобы поиск ее был по возможности быстрее, а требуемая память по возможности меньше.

Кроме того, со стороны языка программирования могут быть дополнительные требования к организации информации. Имена могут иметь определенную область видимости. Например, поле записи должно быть

уникально в пределах структуры (или уровня структуры), но может совпадать с именем объекта вне записи (или другого уровня записи). В то же время имя поля может открываться оператором присоединения, и тогда может возникнуть конфликт имен (или неоднозначность в трактовке имени). Если язык имеет блочную структуру, то необходимо обеспечить такой способ хранения информации, чтобы, во-первых, поддерживать блочный механизм видимости, а во-вторых - эффективно освобождать память при выходе из блока. В некоторых языках (например, Аде) одновременно (в одном блоке) могут быть видимы несколько объектов с одним именем, в других такая ситуация недопустима.

Мы рассмотрим некоторые основные способы организации таблиц символов в компиляторе: таблицы идентификаторов, таблицы расстановки, двоичные деревья и реализацию блочной структуры.

Таблицы идентификаторов

Как уже было сказано, информацию об объекте обычно можно разделить на две части: имя (идентификатор) и описание. Если длина идентификатора ограничена (или имя идентифицируется по ограниченному числу первых символов идентификатора), то таблица символов может быть организована в виде простого массива строк фиксированной длины, как это изображено на рис. 7.1. Некоторые входы могут быть заняты, некоторые - свободны.

Ясно, что, во-первых, размер массива должен быть не меньше числа идентификаторов, которые могут реально появиться в программе (в противном случае возникает переполнение таблицы); во-вторых, как правило, потенциальное число различных идентификаторов существенно больше размера таблицы.

Заметим, что в большинстве языков программирования символьное представление идентификатора может иметь произвольную длину. Кроме того, различные объекты в одной или в разных областях видимости могут иметь одинаковые имена, и нет большого смысла занимать память для повторного хранения идентификатора. Таким образом, удобно имя объекта и его описание хранить по отдельности. В этом случае идентификаторы хранятся в отдельной таблице - таблице идентификаторов. В таблице символов же хранится указатель на соответствующий вход в таблицу идентификаторов. Таблицу идентификаторов можно организовать, например, в виде сплошного массива. Идентификатор в массиве заканчивается каким-либо специальным символом EOS (рис. 7.2). Второй возможный

Рис. 7.1.

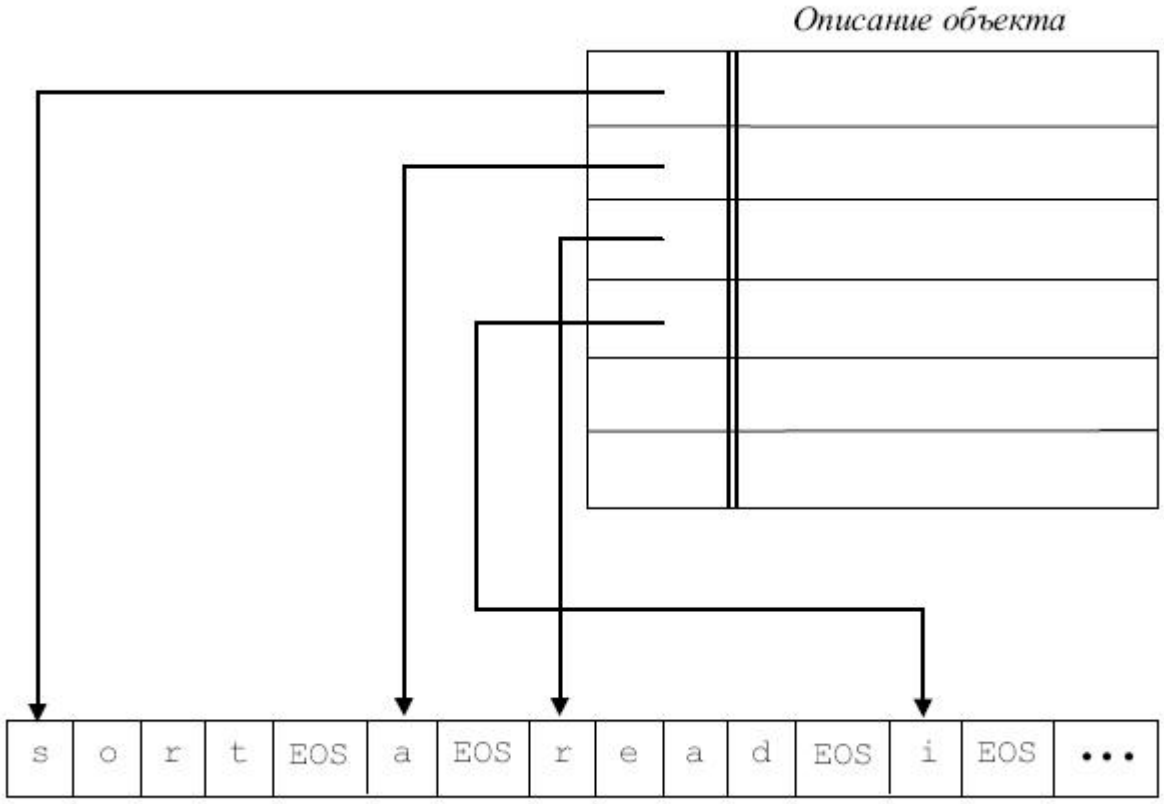


Рис. 7.2.

вариант - в качестве первого символа идентификатора в массив заносится его длина.

Таблицы расстановки

Одним из эффективных способов организации таблицы символов является таблица расстановки (или хеш-таблица). Поиск в такой таблице может быть организован методом повторной расстановки. Суть его заключается в следующем.

Таблица символов представляет собой массив фиксированного размера N. Идентификаторы могут храниться как в самой таблице символов, так и в отдельной таблице идентификаторов.

Определим некоторую функцию h_1 (первичную функцию расстановки), определенную на множестве идентификаторов и принимающую значения от 0 до N - 1 (то есть $0 \le h_1(id) \le N - 1$, где id - символьное представление идентификатора). Таким образом, функция расстановки сопоставляет идентификатору некоторый адрес в таблице символов.

Пусть мы хотим найти в таблице идентификатор id. Если элемент таблицы с номером h1(id) не заполнен, то это означает, что идентификатора в таблице нет. Если же занят, то это еще не означает, что идентификатор id в таблицу занесен, поскольку (вообще говоря) много идентификаторов могут иметь одно и то же значение функции расстановки. Для того чтобы определить, нашли ли мы нужный идентификатор, сравниваем id с элементом таблицы $h_1(id)$. Если они равны - идентификатор найден, если нет - надо продолжать поиск дальше.

Для этого вычисляется вторичная функция расстановки $h_2(h)$ (значением которой опять таки является некоторый адрес в таблице символов). Возможны четыре варианта:

- элемент таблицы не заполнен (то есть идентификатора в таблице нет),
- идентификатор элемента таблицы совпадает с искомым (то есть идентификатор найден),
- адрес элемента совпадает с уже просмотренным (то есть таблица вся просмотрена и идентификатора нет)
- предыдущие варианты не выполняются, так что необходимо продолжать поиск.

Для продолжения поиска применяется следующая функция расстановки $h_3(h_2)$, $h_4(h_3)$ и т.д. Как правило, $h_i = h_2$ для $i \geq 2$. Аргументом функции h_2 является целое в диапазоне [0, N - 1] и она может быть быть устроена по-разному. Приведем три варианта.

1. $h_2(i) = (i + 1) \bmod N$. Берется следующий (циклически) элемент массива. Этот вариант плох тем, что занятые элементы "группируются" , образуют последовательные занятые участки и в пределах этого участка поиск становится по-существу линейным.
2. $h_2(i) = (i + k) \bmod N$, где k и N взаимно просты. По-существу это предыдущий вариант, но элементы накапливаются не в последовательных элементах, а "разносятся".
3. $h_2(i) = (a * i + c) \bmod N$ - "псевдослучайная последовательность". Здесь c и N должны быть взаимно просты, b = a-1 кратно p для любого простого p, являщегося делителем N, b кратно 4, если N кратно 4 [6].

Поиск в таблице расстановки можно описать следующей функцией:

```
void Search(String Id,boolean * Yes,int * Point)
{int H0=h1(Id), H=H0;
  while (1)
  {if (Empty(H)==NULL)
    {*Yes=false;
    *Point=H;
    return;
    }
  else if (IdComp(H,Id)==0)
    {*Yes=true;
    *Point=H;
    return;
    }
  else H=h2(H);
  if (H==H0)
    {*Yes=false;
    *Point=NULL;
    return;
    }
  }
}
```

Функция IdComp(H,Id) сравнивает элемент таблицы на входе H с идентификатором и вырабатывает 0, если они равны. Функция Empty(H) вырабатывает NULL, если вход H пуст. Функция Search присваивает параметрам Yes и Pointer соответственно следующие значения :

true, P - если нашли требуемый идентификатор, где P - указатель на соответствующий этому идентификатору вход в таблице,

false, NULL - если искомый идентификатор не найден, причем в таблице нет свободного места, и

false, P - если искомый идентификатор не найден, но в таблице есть свободный вход P.

Занесение элемента в таблицу можно осуществить следующей функцией:

```
int Insert(String Id)
{boolean Yes;
  int Point=-1;
  Search(Id,&Yes,&Point);
  if (!Yes && (Point!=NULL)) InsertId(Point,Id);
  return(Point);
}
```

Здесь функция InsertId(Point,Id) заносит идентификатор Id для входа Point таблицы.

Таблицы расстановки со списками

Только что описанная схема страдает одним недостатком - возможностью переполнения таблицы. Рассмотрим ее модификацию, когда все элементы, имеющие одинаковое значения (первичной) функции расстановки, связываются в список (при этом отпадает необходимость использования функций h_i для i ≥2). Таблица расстановки со списками - это массив указателей на списки элементов (рис. 7.3)

Вначале таблица расстановки пуста (все элементы имеют значение NULL). При поиске идентификатора Id вычисляется функция расстановки h(Id) и просматривается соответствующий линейный список. Поиск в таблице может быть описан следующей функцией:

```
struct Element
    {String IdentP;
    struct Element * Next;
    };
struct Element * T[N];

struct Element * Search(String Id)
{struct Element * P;
 P=T[h(Id)];
 while (1)
 {if (P==NULL) return(NULL);
  else if (IdComp(P->IdentP,Id)==0) return(P);
  else P=P->Next;
 }
}
```

Занесение элемента в таблицу можно осуществить следующей функцией:

```
struct Element * Insert(String Id)
{struct Element * P,H;
 P=Search(Id);
 if (P!=NULL) return(P);
 else {H=H(Id);
           P=alloc(sizeof(struct Element));
           P->Next=T[H];
       T[H]=P;
       P->IdentP=Include(Id);
      }
 return(P);
}
```

Процедура Include заносит идентификатор в таблицу идентификаторов. Алгоритм иллюстрируется рис. 7.4.

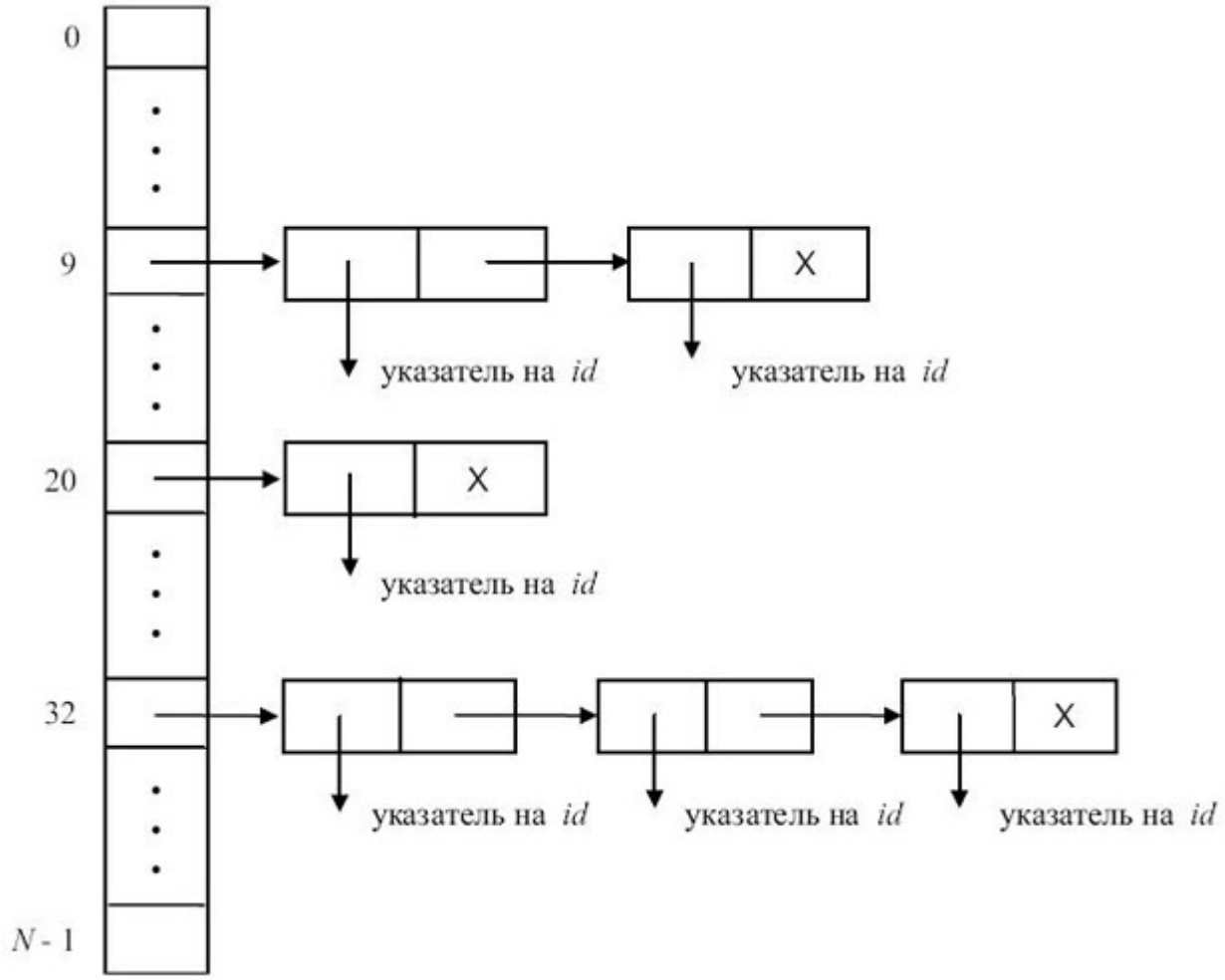


Рис. 7.3.

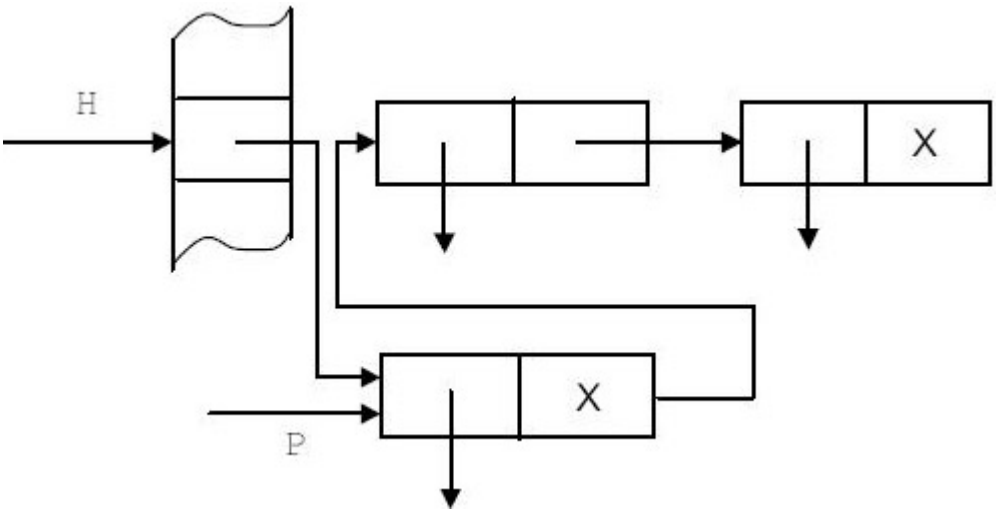


Рис. 7.4.

Функции расстановки

Много внимания исследователями было уделено тому, какой должна быть (первичная) функция расстановки. Основные требования к ней очевидны: она должна легко вычисляться и распределять равномерно. Один из возможных подходов здесь заключается в следующем.

1. По символам строки s определяем положительное целое H. Преобразование одиночных символов в целые обычно можно сделать средствами языка реализации. В Паскале для этого служит функция ord, в Си при выполнении арифметических операций символьные значения трактуются как целые.

2. Преобразуем H, вычисленное выше, в номер элемента, то есть целое между 0 и N - 1, где N - размер таблицы расстановки, например, взятием остатка при делении H на N. Функции расстановки, учитывающие все символы строки, распределяют лучше, чем функции, учитывающие только несколько символов, например, в конце или середине строки. Но такие функции требуют больше вычислений. Простейший способ вычисления H - сложение кодов символов. Перед сложением с очередным символом можно умножить старое значение H на константу q. То есть полагаем $H_0 = 0$, $H_i = q*H_{i-1} + c_i$ для $1 \le i \le k$, k - длина строки. При q = 1 получаем простое сложение символов. Вместо сложения можно выполнять сложение c_i и $q *H_{j-1}$ по модулю
3. Переполнение при выполнении арифметических операций можно игнорировать. Функция Hashpjw, приведенная ниже [?], вычисляется, начиная с H = 0 (предполагается, что используются 32- битовые целые). Для каждого символа с сдвигаем биты H на 4 позиции влево и добавляем очередной символ. Если какой- нибудь из четырех старших бит H равен 1, сдвигаем эти 4 бита на 24 разряда вправо, затем складываем по модулю 2 с H и устанавливаем в 0 каждый из четырех старших бит, равных 1.

```
#define PRIME 211
#define EOS '\0'
int Hashpjw(char *s)
{char *p;
 unsigned H=0, g;
 for (p=s; *p!=EOS; p=p+1)
   {H=(H<<4)+(*p);
    if (g=H&0xf0000000)
   {H=H^(g>>24);
   H=H^g;
 } }
 return H%PRIME;
}
```

Таблицы на деревьях

Рассмотрим еще один способ организации таблиц символов с использованием двоичных деревьев.

Ориентированное дерево называется двоичным, если у него в каждую вершину, кроме одной (корня), входит одна дуга, и из каждой вершины выходит не более двух дуг. Ветвью дерева называется поддерево, состоящее из некоторой дуги данного дерева, ее начальной и конечной вершин, а также всех вершин и дуг, лежащих на всех путях, выходящих из конечной вершины этой дуги. Высотой дерева называется максимальная длина пути в этом дереве от корня до листа.

Пусть на множестве идентификаторов задан некоторый линейный (например, лексикографический) порядок $\prec$, то есть некоторое транзитивное, антисимметричное и антирефлексивное отношение. Таким образом, для произвольной пары идентификаторов id_1 и id_2 либо $id_1 \prec id_2$, либо $id_2 \prec id_1$, либо id_1 совпадает с id_2.

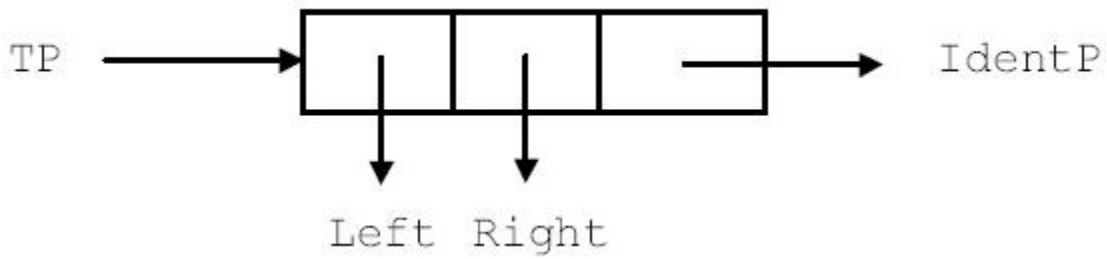


Рис. 7.5.

Каждой вершине двоичного дерева, представляющего таблицу символов, сопоставим идентификатор. При этом, если вершина (которой сопоставлен id) имеет левого потомка (которому сопоставлен id_L), то $id_L \prec id$; если имеет правого потомка (id_R), то $id \prec id_R$. Элемент таблицы изображен на рис. 7.5.

Поиск в такой таблице может быть описан следующей функцией:

```
struct TreeElement * SearchTree(String Id,
         struct TreeElement * TP)
{int comp;
 if (TP==NULL) return NULL;
 comp=IdComp(Id,TP->IdentP);
 if (comp<0) return(SearchTree(Id,TP->Left));
 if (comp>0) return(SearchTree(Id,TP->Right));
 return TP;
}
```

где структура для для элемента дерева имеет вид

```
struct TreeElement
   {String IdentP;
   struct TreeElement * Left, * Right;
   };
```

Занесение в таблицу осуществляется функцией

```
struct TreeElement * InsertTree(String Id,
         struct TreeElement * TP)
{int comp=IdComp(Id,TP->IdentP);
 if (comp<0) return(Fill(Id,TP->Left,
         &(TP->Left)));
 if (comp>0) return(Fill(Id,TP->Right,
         &(TP->Right)));
 return(TP);
}

struct TreeElement * Fill(String Id,
         struct TreeElement * P,
         struct TreeElement ** FP)
{ if (P==NULL)
  {P=alloc(sizeof(struct TreeElement));
  P->IdentP=Include(Id);
  P->Left=NULL;
  P->Right=NULL;
  *FP=P;
  return(P);
  }
 else return(InsertTree(Id,P));
}
```

Как показано в работе [10], среднее время поиска в таблице размера n, организованной в виде двоичного дерева, при равной вероятности появления каждого объекта равно $(2 \ln 2) \log_2 n + O(1)$. Однако, на практике случай равной вероятности появления объектов встречается довольно редко. Поэтому в дереве появляются более длинные и более короткие ветви, и среднее время поиска увеличивается.

Чтобы уменьшить среднее время поиска в двоичном дереве, можно в процессе построения дерева следить за тем, чтобы оно все время оставалось сбалансированным. А именно, назовем дерево сбалансированным, если ни для какой вершины высота выходящей из нее правой ветви не отличается от высоты левой более чем на 1. Для того, чтобы достичь сбалансированности, в процессе добавления новых вершин дерево можно слегка перестраивать следующим образом [1].

Определим для каждой вершины дерева характеристику, равную разности высот выходящих из нее правой и левой ветвей. В сбалансированном дереве характеристика вершины может быть равной -1, 0 и 1, для листьев она равна 0.

Пусть мы определили место новой вершины в дереве. Ее характеристика равна 0. Назовем путь, ведущий от корня к новой вершине, выделенным. При добавлении новой вершины могут измениться характеристики только тех вершин, которые лежат на выделенном пути. Рассмотрим заключительный отрезок выделенного пути, такой, что до добавления вершины характеристики всех вершин на нем были равны 0.

Если верхним концом этого отрезка является сам корень, то дерево перестраивать не надо, достаточно лишь изменить характеристики вершин на этом пути на 1 или -1, в зависимости от того, влево или вправо пристроена новая вершина.

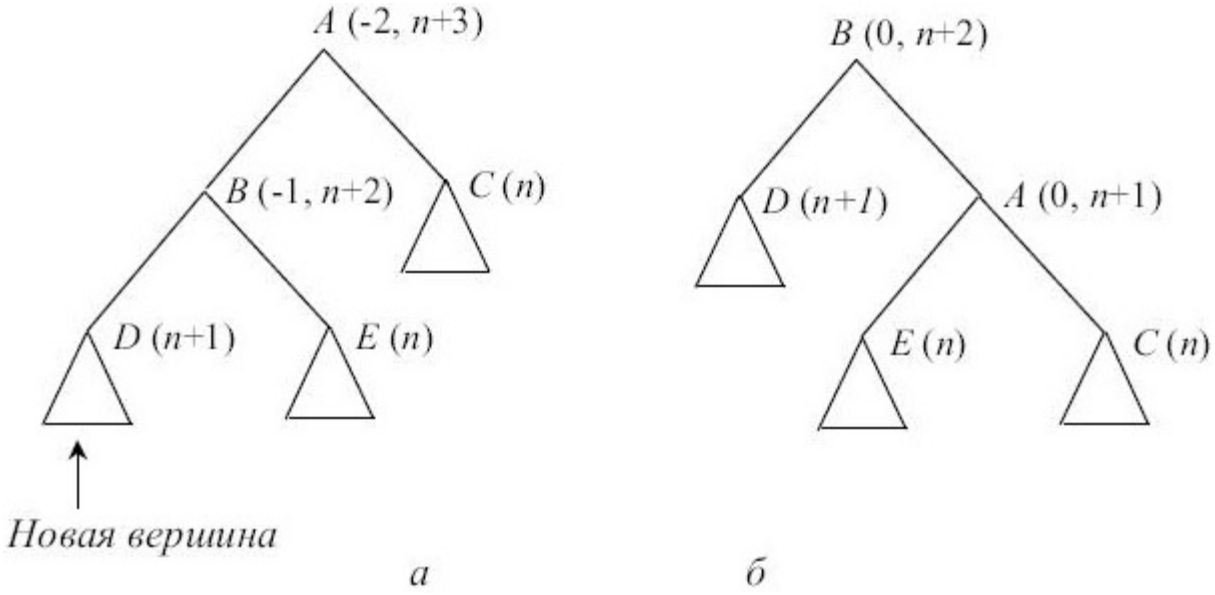


Рис. 7.6.

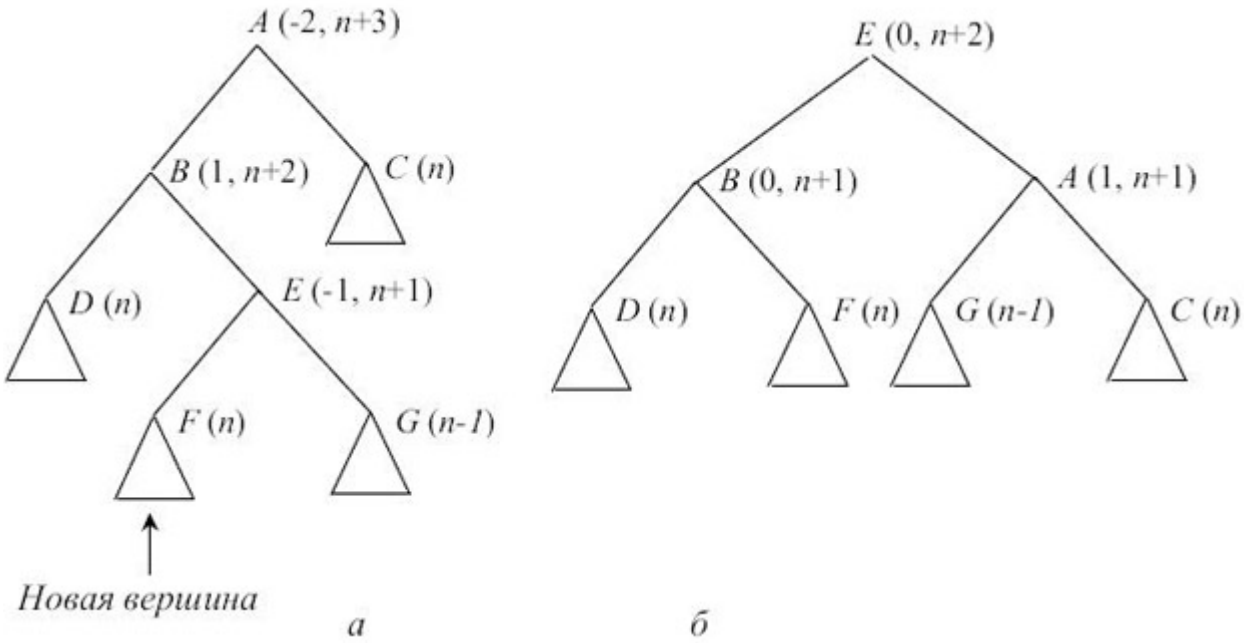


Рис. 7.7.

Пусть верхний конец заключительного отрезка - не корень. Рассмотрим вершину A - "родителя" верхнего конца заключительного отрезка. Перед добавлением новой вершины характеристика A была равна ± 1. Если A имела характеристику 1 (-1) и новая вершина добавляется в левую (правую) ветвь, то характеристика вершины A становится равной 0, а высота поддерева с корнем в A не меняется. Так что и в этом случае дерево перестраивать не надо. Пусть теперь характеристика A до перестраивания была равна -1 и новая вершина добавлена к левой ветви A (аналогично - для случая 1 и добавления к правой ветви). Рассмотрим вершину B - левого потомка A. Возможны следующие варианты.

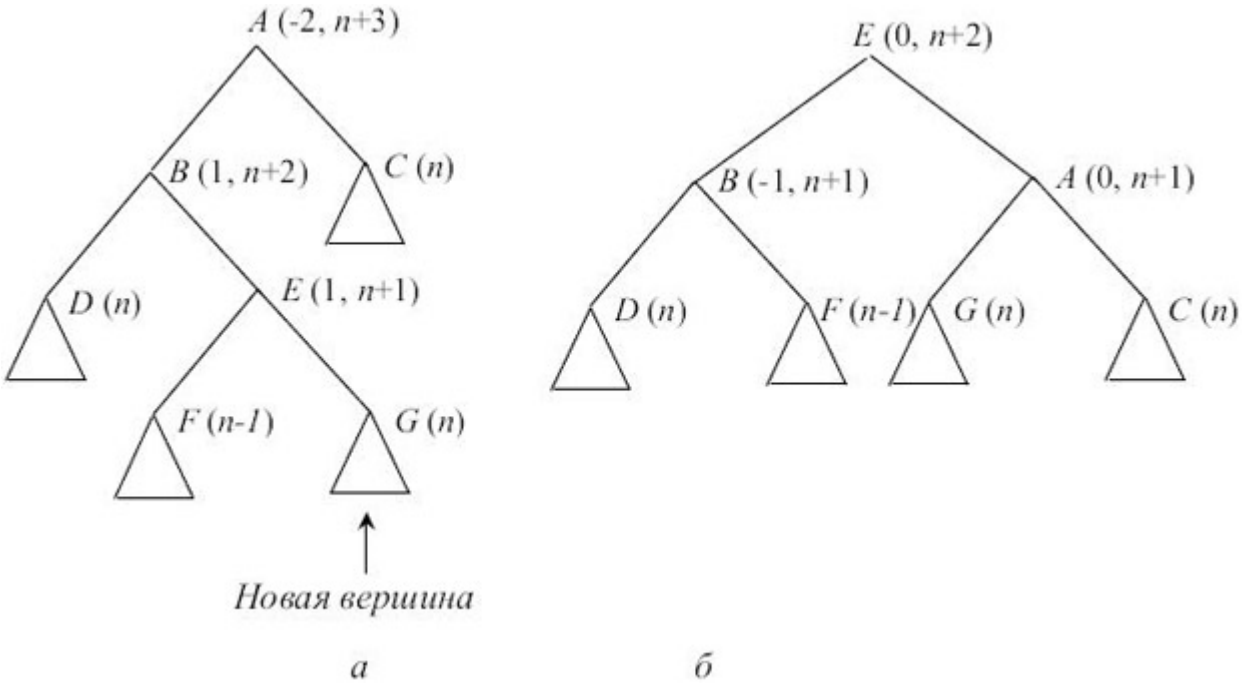


Рис. 7.8.

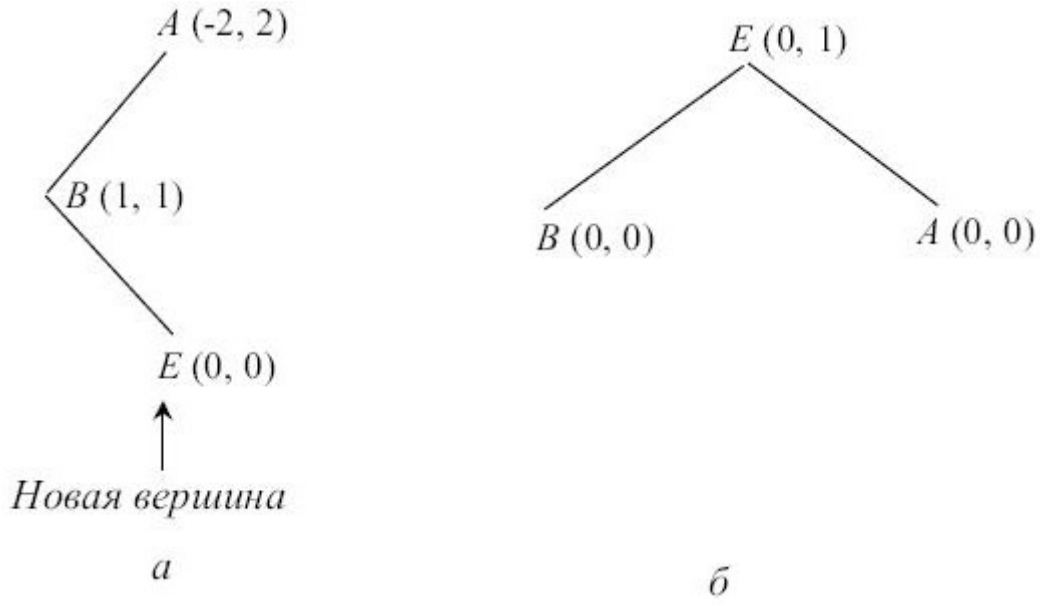


Рис. 7.9.

Если характеристика B после добавления новой вершины в D стала равна -1, то дерево имеет структуру, изображенную на рис. 7.6, а. Перестроив дерево так, как это изображено на рис. 7.6, б, мы добьемся сбалансированности (в скобках указаны характеристики вершин, где это существенно, и соотношения высот после добавления).

Если характеристика вершины B после добавления новой вершины в E стала равна 1, то надо отдельно рассмотреть случаи, когда характеристика вершины E, следующей за B на выделенном пути, стала равна -1, 1 и 0 (в последнем случае вершина E - новая). Вид дерева до и после перестройки для этих случаев показан соответственно на рис. 7.7, рис. 7.8 и рис. 7.9.

Реализация блочной структуры

С точки зрения структуры программы блоки (и/или процедуры) образуют дерево. Каждой вершине дерева этого представления, соответствующей блоку, можно сопоставить свою таблицу символов (и, возможно, одну общую таблицу идентификаторов). Работу с таблицами блоков можно организовать в магазин-

ном режиме: при входе в блок создавать таблицу символов, при выходе - уничтожать. При этом сами таблицы должны быть связаны в упорядоченный список, чтобы можно было просматривать их в порядке вложенности. Если таблицы организованы с помощью функций расстановки, это означает, что для каждой таблицы должна быть создана своя таблица расстановки.

Сравнение методов реализации таблиц

Рассмотрим преимущества и недостатки рассмотренных методов реализации таблиц с точки зрения техники использования памяти.

Использование динамической памяти, как правило, довольно дорогая операция, поскольку механизмы поддержания работы с динамической памятью могут быть достаточно сложны. Необходимо поддерживать списки свободной и занятой памяти, выбирать наиболее подходящий кусок памяти при запросе, включать освободившийся кусок в список свободной памяти и, возможно, склеивать куски свободной памяти в списке.

С другой стороны, использование массива требует отведения заранее довольно большой памяти, а это означает, что значительная память вообще не будет использоваться. Кроме того, часто приходится заполнять не все элементы массива (например, в таблице идентификаторов или в тех случаях, когда в массиве фактически хранятся записи переменной длины, например, если в таблице символов записи для различных объектов имеют различный состав полей). Обращение к элементам массива может означать использование операции умножения при вычислении индексов, что может замедлить исполнение.

Наилучшим, по-видимому, является механизм доступа по указателям и использование факта магазинной организации памяти в компиляторе. Для этого процедура выделения памяти выдает необходимый кусок из подряд идущей памяти, а при выходе из процедуры вся память, связанная с этой процедурой, освобождается простой перестановкой указателя свободной памяти в состояние перед началом обработки процедуры. В чистом виде это не всегда, однако, возможно. Например, локальный модуль в Модуле-2 может экспортировать некоторые объекты наружу. При этом схему реализации приходится "подгонять" под механизм распределения памяти. В данном случае, например, необходимо экспортированные объекты вынести в среду охватывающего блока и свернуть блок локального модуля.

8. Лекция: Промежуточное представление программы

В данной лекции рассматривается промежуточное представление программы, которое предназначено прежде всего для удобства генерации кода и/или проведения различных оптимизаций. Рассматриваются часто используемые формы промежуточного представления такие, как ориентированный граф (в частности, абстрактное синтаксическое дерево, в том числе атрибутированное), трехадресный код (в виде троек или четверок), префиксная и постфиксная запись. Также рассмотрена виртуальная Java-машина и ее команды. Приведены основные понятия, графическая интерпретация промежуточного представления программ и части программного кода.

В процессе трансляции компилятор часто используют промежуточное представление (ПП) исходной программы, предназначенное прежде всего для удобства генерации кода и/или проведения различных оптимизаций. Сама форма ПП зависит от целей его использования.

Наиболее часто используемыми формами ПП является ориентированный граф (в частности, абстрактное синтаксическое дерево, в том числе атрибутированное), трехадресный код (в виде троек или четверок), префиксная и постфиксная запись.

Представление в виде ориентированного графа

Простейшей формой промежуточного представления является синтаксическое дерево программы. Ту же самую информацию о входной программе, но в более компактной форме дает ориентированный ацикли-

ческий граф (ОАГ), в котором в одну вершину объединены вершины синтаксического дерева, представляющие общие подвыражения. Синтаксическое дерево и ОАГ для оператора присваивания

a := b *-c + b * -c

приведены на рис. 8.1

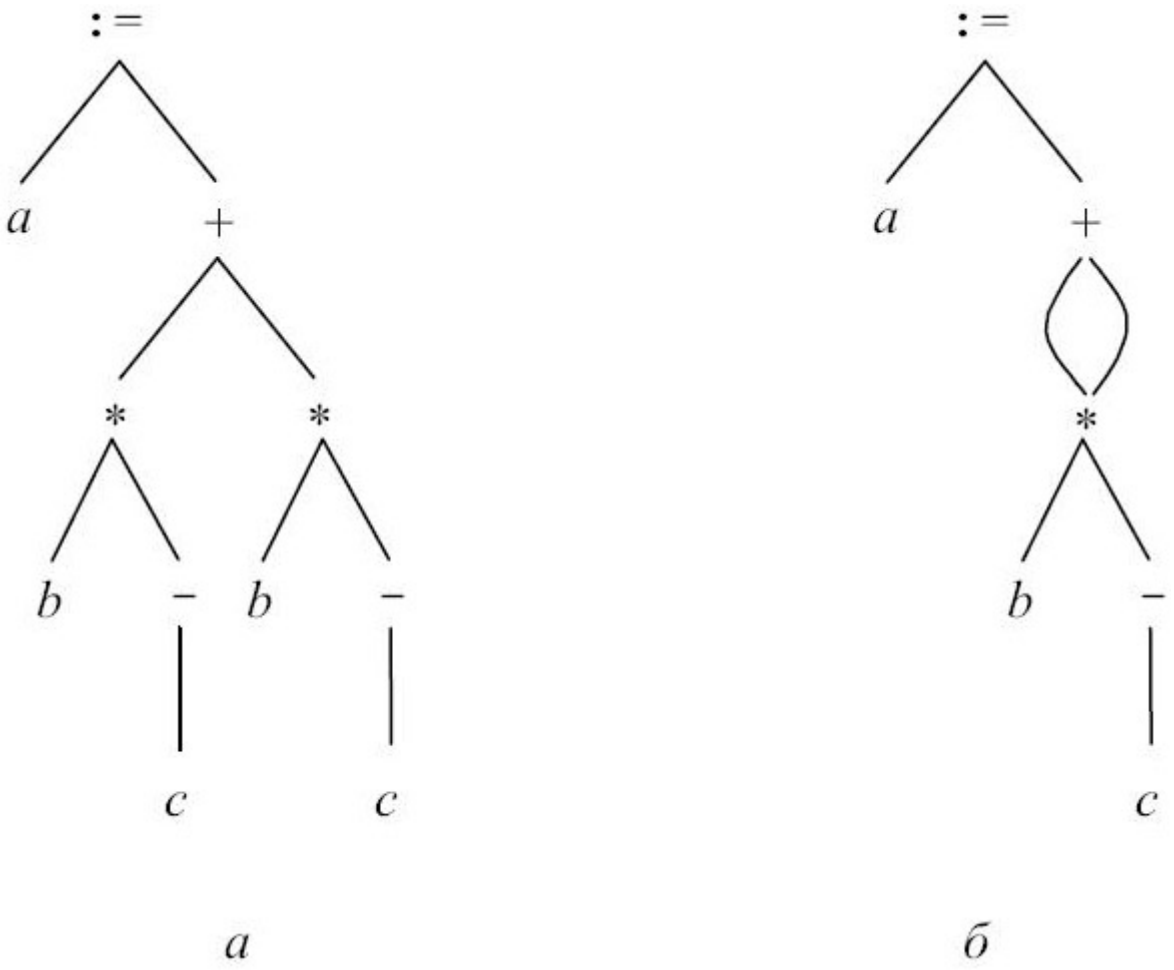


Рис. 8.1.

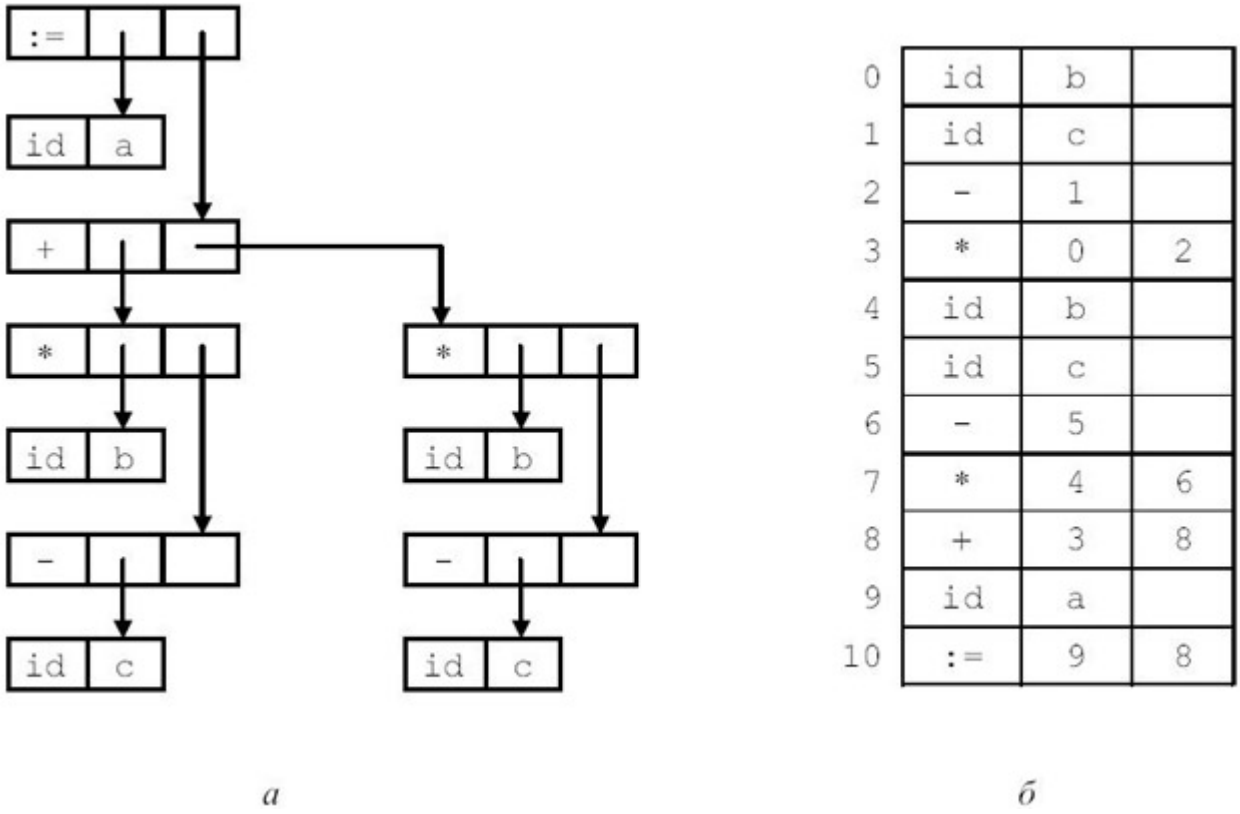


Рис. 8.2.

На рис. 8.2 приведены два представления в памяти синтаксического дерева на рис. 8.1, а. Каждая вершина кодируется записью с полем для операции и полями для указателей на потомков. На рис. 8.2, б, вершины размещены в массиве записей и индекс (или вход) вершины служит указателем на нее.

Трехадресный код

Трехадресный код - это последовательность операторов вида x := y op z, где x, y и z - имена, константы или сгенерированные компилятором временные объекты. Здесь op - двуместная операция, например операция плавающей или фиксированной арифметики, логическая или побитовая. В правую часть может входить только один знак операции.

Составные выражения должны быть разбиты на подвыражения, при этом могут появиться временные имена (переменные). Смысл термина "трехадресный код" в том, что каждый оператор обычно имеет три адреса: два для операндов и один для результата. Трехадресный код - это линеаризованное представление синтаксического дерева или ОАГ, в котором временные имена соответствуют внутренним вершинам дерева или графа. Например, выражение x + y * z может быть протранслировано в последовательность операторов

```
t1 := y * z
t2 := x + t1
```

где t1 и t2 - имена, сгенерированные компилятором. В виде трехадресного кода представляются не только двуместные операции, входящие в выражения. В таком же виде представляются операторы управления программы и одноместные операции. В этом случае некоторые из компонент трехадресного кода могут не использоваться. Например, условный оператор

```
if A > B then S1 else S2
```

может быть представлен следующим кодом:

```
t := A - B
JGT t, S2
```

. . .

Здесь JGT - двуместная операция условного перехода, не вырабатывающая результата.

Разбиение арифметических выражений и операторов управления делает трехадресный код удобным при генерации машинного кода и оптимизации. Использование имен промежуточных значений, вычисляемых в программе, позволяет легко переупорядочивать трехадресный код.

Таблица 8.1.

а	б
t1 := -c	t1 := -c
t2 := b * t1	t2 := b * t1
t3 := -c	t5 := t2 + t2
t4 := b * t3	a := t5
t5 := t2 + t4	
a := t5	

Представления синтаксического дерева и графа рис. 8.1 в виде трехадресного кода дано в таблица 8.1, а, и таблица 8.1, б, соответственно.

Трехадресный код - это абстрактная форма промежуточного кода. В реализации трехадресный код может быть представлен записями с полями для операции и операндов. Рассмотрим три способа реализации трехадресного кода: четверки, тройки и косвенные тройки.

Четверка - это запись с четырьмя полями, которые будем называть op, arg1, arg2 и result. Поле op содержит код операции. В операторах с унарными операциями типа x := -y или x := y поле arg2 не используется. В некоторых операциях (типа "передать параметр") могут не использоваться ни arg2, ни result. Условные и безусловные переходы помещают в result метку перехода. На рис. 8.2, а, приведены четверки для оператора присваивания a := b *-c+b *-c. Они получены из трехадресного кода в таблица 8.1, а.

Таблица 8.2а. четверки

	op	arg1	arg2	result
(0)	-	c		t1
(1)	*	b	t1	t2
(2)	-	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Таблица 8.2б. тройки

	op	arg1	arg2
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Обычно содержимое полей arg1, arg2 и result - это указатели на входы таблицы символов для имен, представляемых этими полями. Временные имена вносятся в таблицу символов по мере их генерации.

Чтобы избежать внесения новых имен в таблицу символов, на временное значение можно ссылаться, используя позицию вычисляющего его оператора. В этом случае трехадресные операторы могут быть представлены записями только с тремя полями: op, arg1 и arg2, как это показано в таблица 8.2б Поля arg1 и arg2 - это либо указатели на таблицу символов (для имен, определенных программистом, или констант), либо указатели на тройки (для временных значений). Такой способ представления трехадресного кода называют тройками. Тройки соответствуют представлению синтаксического дерева или ОАГ с помощью массива вершин.

Числа в скобках - это указатели на тройки, а имена - это указатели на таблицу символов. На практике информация, необходимая для интерпретации различного типа входов в поля arg1 и arg2, кодируется в поле op или дополнительных полях. Тройки таблица 8.2б, соответствуют четверкам таблица 8.2а

Для представления тройками трехместной операции типа x[i] := y требуется два входа, как это показано в таблица 8.3а, представление x := y[i] двумя операциями показано в таблица 8.3б

Таблица 8.3а. x[i]:=y

	op	arg1	arg2
(0)	[]=	x	i
(1)	:=	(0)	y

Таблица 8.3б. x:=y[i]

	op	arg1	arg2
(0)	=[]	y	i
(1)	:=	x	(0)

Трехадресный код может быть представлен не списком троек, а списком указателей на них. Такая реализация обычно называется косвенными тройками. Например, тройки рис. таблица 8.2б, могут быть реализованы так, как это изображено на рис. рис. 8.3.

	оператор
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg1*	*arg2*
(14)	-	c	
(15)	*	b	(14)
(16)	-	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	:=	a	(18)

Рис. 8.3.

При генерации объектного кода каждой переменной, как временной, так и определенной в исходной программе, назначается память периода исполнения, адрес которой обычно хранится в таблице генератора кода. При использовании четверок этот адрес легко получить через эту таблицу.

Более существенно преимущество четверок проявляется в оптимизирующих компиляторах, когда может возникнуть необходимость перемещать операторы. Если перемещается оператор, вычисляющий x, не требуется изменений в операторе, использующем x. В записи же тройками перемещение оператора, определяющего временное значение, требует изменения всех ссылок на этот оператор в массивах arg1 и arg2. Из-за этого тройки трудно использовать в оптимизирующих компиляторах.

В случае применения косвенных троек оператор может быть перемещен переупорядочиванием списка операторов. При этом не надо менять указатели на op, arg1 и arg2. Этим косвенные тройки похожи на четверки. Кроме того, эти два способа требуют примерно одинаковой памяти. Как и в случае простых троек, при использовании косвенных троек выделение памяти для временных значений может быть отложено на этап генерации кода. По сравнению с четверками при использование косвенных троек можно сэкономить память, если одно и то же временное значение используется более одного раза. Например, на рис. 8.3 можно объединить строки (14) и (16), после чего можно объединить строки (15) и (17).

Линеаризованные представления

В качестве промежуточных представлений весьма распространены линеаризованные представления деревьев. Линеаризованное представление позволяет относительно легко хранить промежуточное представление во внешней памяти и обрабатывать его. Наиболее распространенной формой линеаризованного представления является польская запись - префиксная (прямая) или постфиксная (обратная).

Постфиксная запись - это список вершин дерева, в котором каждая вершина следует (при обходе снизу-вверх слева-направо) непосредственно за своими потомками. Дерево на рис. 8.1, а, в постфиксной записи может быть представлено следующим образом:

a b c - * b c - * + :=

В постфиксной записи вершины синтаксического дерева явно не присутствуют. Они могут быть восстановлены из порядка, в котором следуют вершины и из числа операндов соответствующих операций. Восстановление вершин аналогично вычислению выражения в постфиксной записи с использованием стека.

В префиксной записи сначала указывается операция, а затем ее операнды. Например, для приведенного выше выражения имеем

:= a + * b - c * b - c

Рассмотрим детальнее одну из реализаций префиксного представления - Лидер [12]. Лидер - это аббревиатура от "ЛИнеаризованное ДЕРево". Это машинно-независимая префиксная запись. В Лидере сохраняются все объявления и каждому из них присваивается свой уникальный номер, который используется для ссылки на объявление. Рассмотрим пример.

```
module M;
var X,Y,Z: integer;
procedure DIF(A,B:integer):integer;
  var R:integer;
  begin R:=A-B;
    return(R);
  end DIF;
begin Z:=DIF(X,Y);
end M.
```

Этот фрагмент имеет следующий образ в Лидере.

```
program 'M'
var int
var int
var int
procbody proc int int end int
  var int
  begin assign var 1 7 end
     int int mi par 1 5 end par 1 6 end
    result 0 int var 1 7 end
    return
  end
  begin assign var 0 3 end int
  icall 0 4 int var 0 1 end
  int var 0 2 end end
end
```

Рассмотрим его более детально:

program 'M'	Имя модуля нужно для редактора связей.
var int	Это образ переменных X, Y, Z;
var int	переменным X, Y, Z присваиваются
var int	номера 1, 2, 3 на уровне 0.
procbody proc	Объявление процедуры с двумя
int int end	целыми параметрами, возвращающей целое.
int	Процедура получает номер 4 на уровне 0 и параметры имеют номера 5, 6 на уровне 1.
var int	Переменная R имеет номер 7 на уровне 1.
begin	Начало тела процедуры.
assign	Оператор присваивания.
var 1 7 end	Левая часть присваивания (R).
int	Тип присваиваемого значения.

int mi	Целое вычитание.
par 1 5 end	Уменьшаемое (A)
par 1 6 end	Вычитаемое (B)
result 0	Результат процедуры уровня 0
int	Результат имеет целый тип
var 1 7 end	Результат - переменная R
return	Оператор возврата
end	Конец тела процедуры
begin	Начало тела модуля.
assign	Оператор присваивания.
var 0 3 end	Левая часть - переменная Z.
int	Тип присваиваемого значения.
icall 0 4	Вызов локальной процедуры DIF
int var 0 1 end	Фактические параметры X
int var 0 2 end	и Y
end	Конец вызова.
end	Конец тела модуля

Виртуальная машина Java

Программы на языке Java транслируются в специальное промежуточное представление, которое затем интерпретируется так называемой "виртуальной машиной Java ". Виртуальная машина Java представляет собой стековую машину: она не имеет памяти прямого доступа, все операции выполняются над операндами, расположенными на верхушке стека. Чтобы, например, выполнить операцию с участием константы или переменной, их предварительно необходимо загрузить на верхушку стека. Код операции - всегда один байт. Если операция имеет операнды, они располагаются в следующих байтах.

К элементарным типам данных, с которыми работает машина, относятся short, integer, long, float, double (все знаковые).

Организация памяти

Машина имеет следующие регистры: pc - счетчик команд; optop - указатель вершины стека операций; frame - указатель на стек-фрейм исполняемого метода; vars - указатель на 0-ю переменную исполняемого метода. Все регистры 32-разрядные. Стек-фрейм имеет три компоненты: локальные переменные, среду исполнения, стек операндов. Локальные переменные отсчитываются от адреса в регистре vars. Среда исполнения служит для поддержания самого стека. Она включает указатель на предыдущий фрейм, указатель на собственные локальные переменные, на базу стека операций и на верхушку стека. Кроме того, здесь же хранится некоторая дополнительная информация, например, для отладчика.

Куча сборки мусора содержит экземпляры объектов, которые создаются и уничтожаются автоматически. Область методов содержит коды, таблицы символов и т.д. С каждым классом связана область констант. Она содержит имена полей, методов и другую подобную информацию, которая используется методами.

Набор команд виртуальной машины

Виртуальная Java-машина имеет следующие команды:

помещение констант на стек,
помещение локальных переменных на стек,
запоминание значений из стека в локальных переменных,
обработка массивов,
управление стеком,
арифметические команды,
логические команды,
преобразования типов,
передача управления,
возврат из функции,
табличный переход,
обработка полей объектов,
вызов метода,
обработка исключительных ситуаций,
прочие операции над объектами,
мониторы,
отладка.

Рассмотрим некоторые команды подробнее.

Помещение локальных переменных на стек

Команда iload - загрузить целое из локальной переменной. Операндом является смещение переменной в области локальных переменных. Указываемое значение копируется на верхушку стека операций. Имеются аналогичные команды для помещения плавающих, двойных целых, двойных плавающих и т.д.

Команда istore - сохранить целое в локальной переменной. Операндом операции является смещение переменной в области локальных переменных. Значение с верхушки стека операций копируется в указываемую область локальных переменных. Имеются аналогичные команды для помещения плавающих, двойных целых, двойных плавающих и т.д.

Вызов метода

Команда invokevirtual. При трансляции объектно-ориентированных языков программирования из-за возможности перекрытия виртуальных методов, вообще говоря, нельзя статически протранслировать вызов метода объекта. Это связано с тем, что если метод перекрыт в производном классе, и вызывается метод объекта-переменной, то статически неизвестно, объект какого класса (базового или производного) хранится в переменной. Поэтому с каждым объектом связывается таблица всех его виртуальных методов: для каждого метода там помещается указатель на его реализацию в соответствии с принадлежностью самого объекта классу в иерархии классов.

В языке Java различные классы могут реализовывать один и тот же интерфейс. Если объявлена переменная или параметр типа интерфейс, то динамически нельзя определить объект какого класса присвоен переменной:

```
interface I;
class C1 implements I;
class C2 implements I;
I O;
C1 O1;
C2 O2;
...
O=O1;
...
O=O2;
...
```

В этой точке программы, вообще говоря, нельзя сказать, какого типа значение хранится в переменной O. Кроме того, при работе программы на языке Java имеется возможность использования методов из других пакетов. Для реализации этого механизма в Java-машине используется динамическое связывание.

Предполагается, что стек операндов содержит handle объекта или массива и некоторое количество аргументов. Операнд операции используется для конструирования индекса в области констант текущего класса. Элемент по этому индексу в области констант содержит полную сигнатуру метода. Сигнатура метода описывает типы параметров и возвращаемого значения. Из handle объекта извлекается указатель на таблицу методов объекта. Просматривается сигнатура метода в таблице методов. Результатом этого просмотра является индекс в таблицу методов именованного класса, для которого найден указатель на блок метода. Блок метода указывает тип метода (native, synchronized и т.д.) и число аргументов, ожидаемых на стеке операндов.

Если метод помечен как synchronized, запускается монитор, связанный с handle. Базис массива локальных переменных для нового стек-фрейма устанавливается так, что он указывает на handle на стеке. Определяется общее число локальных переменных, используемых методом, и после того, как отведено необходимое место для локальных переменных, окружение исполнения нового фрейма помещается на стек. База стека операндов для этого вызова метода устанавливается на первое слово после окружения исполнения. Затем исполнение продолжается с первой инструкции вызванного метода.

Обработка исключительных ситуаций

Команда athrow - возбудить исключительную ситуацию.

С каждым методом связан список операторов catch. Каждый оператор catch описывает диапазон инструкций, для которых он активен, тип исключения, который он обрабатывает. Кроме того, с оператором связан набор инструкций, которые его реализуют. При возникновении исключительной ситуации просматривается список операторов catch, чтобы установить соответствие. Исключительная ситуация соответствует оператору catch, если инструкция, вызвавшая исключительную ситуацию, находится в соответствующем диапазоне и исключительная ситуация принадлежит подтипу типа ситуации, которые обрабатывает оператор catch. Если соответствующий оператор catch найден, управление передается обработчику. Если нет, текущий стек-фрейм удаляется, и исключительная ситуация возбуждается вновь.

Порядок операторов catch в списке важен. Интерпретатор передает управление первому подходящему оператору catch.

Организация информации в генераторе кода

Синтаксическое дерево в чистом виде несет только информацию о структуре программы. На самом деле в процессе генерации кода требуется также информация о переменных (например, их адреса), процедурах (также адреса, уровни), метках и т.д. Для представления этой информации возможны различные решения. Наиболее распространены два:

- информация хранится в таблицах генератора кода;
- информация хранится в соответствующих вершинах дерева.

Рассмотрим, например, структуру таблиц, которые могут быть использованы в сочетании с Лидер-представлением. Поскольку Лидер-представление не содержит информации об адресах переменных, значит, эту информацию нужно формировать в процессе обработки объявлений и хранить в таблицах. Это касается и описаний массивов, записей и т.д. Кроме того, в таблицах также должна содержаться информация о процедурах (адреса, уровни, модули, в которых процедуры описаны, и т.д.).

При входе в процедуру в таблице уровней процедур заводится новый вход - указатель на таблицу описаний. При выходе указатель восстанавливается на старое значение. Если промежуточное представление - дерево, то информация может храниться в вершинах самого дерева.

Уровень промежуточного представления

Как видно из приведенных примеров, промежуточное представление программы может в различной степени быть близким либо к исходной программе, либо к машине. Например, промежуточное представление может содержать адреса переменных, и тогда оно уже не может быть перенесено на другую машину. С другой стороны, промежуточное представление может содержать раздел описаний программы, и тогда информацию об адресах можно извлечь из обработки описаний. В то же время ясно, что первое более эффективно, чем второе. Операторы управления в промежуточном представлении могут быть представлены в исходном виде (в виде операторов языка if, for, while и т.д.), а могут содержаться в виде переходов. В первом случае некоторая информация может быть извлечена из самой структуры (например, для оператора for - информация о переменной цикла, которую, может быть, разумно хранить на регистре, для оператора case - информация о таблице меток и т.д.).

Во втором случае представление проще и унифицированней. Некоторые формы промежуточного представления удобны для различного рода оптимизаций, некоторые - нет (например, косвенные тройки, в отличие от префиксной записи, позволяют эффективное перемещение кода).

9. Лекция: Генерация кода

В данной лекции рассматривается генерация кода, задачей которой является построение для программы на входном языке эквивалентной машинной программы. Рассматривается действие модели машины, осуществляющей генерацию кода. Приведены основные понятия и части программного кода.

Задача генератора кода - построение для программы на входном языке эквивалентной машинной программы. Обычно в качестве входа для генератора кода служит некоторое промежуточное представление программы.

Генерация кода включает ряд специфических, относительно независимых подзадач: распределение памяти (в частности, распределение регистров), выбор команд, генерацию объектного (или загрузочного) модуля. Конечно, независимость этих подзадач относительна: например, при выборе команд нельзя не учитывать схему распределения памяти, и, наоборот, схема распределения памяти (регистров, в частности)

ведет к генерации той или иной последовательности команд. Однако удобно и практично эти задачи все же разделять, обращая при этом внимание на их взаимодействие.

В какой-то мере схема генератора кода зависит от формы промежуточного представления. Ясно, что генерация кода из дерева отличается от генерации кода из троек, а генерация кода из префиксной записи отличается от генерации кода из ориентированного графа. В то же время все генераторы кода имеют много общего, и основные применяемые алгоритмы отличаются, как правило, только в деталях, связанных с используемым промежуточным представлением.

В дальнейшем в качестве промежуточного представления мы будем использовать префиксную нотацию. А именно, алгоритмы генерации кода будем излагать в виде атрибутных схем со входным языком Лидер.

Модель машины

При изложении алгоритмов генерации кода мы будем следовать некоторой модели машины, в основу которой положена система команд микропроцессора Motorola MC68020. В микропроцессоре имеется регистр - счетчик команд PC, 8 регистров данных и 8 адресных регистров.

В системе команд используются следующие способы адресации:

ABS - абсолютная: исполнительным адресом является значение адресного выражения.

IMM - непосредственный операнд: операндом команды является константа, заданная в адресном выражении.

D - прямая адресация через регистр данных, записывается как Xn, операнд находится в регистре Xn.

A - прямая адресация через адресный регистр, записывается как An, операнд находится в регистре An.

INDIRECT - записывается как (An), адрес операнда находится в адресном регистре An.

POST - пост-инкрементная адресация, записывается как (An)+, исполнительный адрес есть значение адресного регистра An и после исполнения команды значение этого регистра увеличивается на длину операнда.

PRE - преинкрементная адресация, записывается как -(An): перед исполнением операции содержимое адресного регистра An уменьшается на длину операнда, исполнительный адрес равен новому содержимому адресного регистра.

INDISP - косвенная адресация со смещением, записывается как (bd,An), исполнительный адрес вычисляется как (An)+d - содержимое An плюс d.

INDEX - косвенная адресация с индексом, записывается как (bd,An, Xn*sc), исполнительный адрес вычисляется как (An)+bd+(Xn)*sc - содержимое адресного регистра + адресное смещение + содержимое индексного регистра, умноженное на sc.

INDIRPC - косвенная через PC (счетчик команд), записывается как (bd, PC), исполнительный адрес определяется выражением (PC)+bd.

INDEXPC - косвенная через PC со смещением, записывается как (bd,PC, Xn*sc), исполнительный адрес определяется выражением (PC)+bd+(Xn)*sc.

INDPRE - пре-косвенная через память, записывается как ([bd,An,sc*Xn], od) (схема вычисления адресов для этого и трех последующих способов адресации приведена ниже).

INDPOST - пост-косвенная через память: ([bd,An],sc*Xn,od).

INDPREPC - прекосвенная через PC: ([bd,PC,sc*Xn],od).

INDPOSTPC - пост-косвенная через PC: ([bd,PC],Xn,od). Здесь bd - это 16- или 32-битная константа, называемая смещением, od - 16- или 32-битная литеральная константа, называемая внешним смещением. Эти способы адресации могут использоваться в упрощенных формах без смещений bd и/или od и без регистров An или Xn. Следующие примеры иллюстрируют косвенную постиндексную адресацию:

```
MOVE D0, ([A0])
MOVE D0, ([4,A0])
MOVE D0, ([A0],6)
MOVE D0, ([A0],D3)
MOVE D0, ([A0],D4,12)
MOVE D0, ([$12345678,A0],D4,$FF000000)
```

Индексный регистр Xn может масштабироваться (умножаться) на 2,4,8, что записывается как sc*Xn. Например, в исполнительном адресе ([24,A0, 4*D0]) содержимое квадратных скобок вычисляется как [A0] + 4 * [D0] + 24.

Эти способы адресации работают следующим образом. Каждый исполнительный адрес содержит пару квадратных скобок [...] внутри пары круглых скобок, то есть ([...], ...). Сначала вычисляется содержимое квадратных скобок, в результате чего получается 32-битный указатель. Например, если используется постиндексная форма [20,A2], то исполнительный адрес - это 20 + [A2]

Аналогично, для преиндексной формы [12,A4,D5] исполнительный адрес - это 12 + [A4] + [D5]. Указатель, сформированный содержимым квадратных скобок, используется для доступа в память, чтобы получить новый указатель (отсюда термин косвенная адресация через память). К этому новому указателю добавляется содержимое внешних круглых скобок и таким образом формируется исполнительный адрес операнда.

В дальнейшем изложении будут использованы следующие команды (в частности, рассматриваются только арифметические команды с целыми операндами, но не с плавающими):

MOVEA ИА, A - загрузить содержимое по исполнительному адресу ИА на адресный регистр A.

MOVE ИА1, ИА2 - содержимое по исполнительному адресу ИА1 переписать по исполнительному адресу ИА2.

MOVEM список_регистров, ИА - сохранить указанные регистры в памяти, начиная с адреса ИА (регистры указываются маской в самой команде).

MOVEM ИА, список_регистров - восстановить указанные регистры из памяти, начиная с адреса ИА (регистры указываются маской в самой команде).

LEA ИА, A - загрузить исполнительный адрес ИА на адресный регистр A.

MUL ИА, D - умножить содержимое по исполнительному адресу ИА на содержимое регистра данных D и результат разместить в D (на самом деле в системе команд имеются две различные команды MULS и MULU для чисел со знаком и чисел без знака соответственно; для упрощения мы не будем принимать во внимание это различие).

DIV ИА, D - разделить содержимое регистра данных D на содержимое по исполнительному адресу ИА и результат разместить в D.

ADD ИА, D - сложить содержимое по исполнительному адресу ИА с содержимым регистра данных D и результат разместить в D.

SUB ИА, D - вычесть содержимое по исполнительному адресу ИА из содержимого регистра данных D и результат разместить в D.

Команды CMP и TST формируют разряды регистра состояний. Всего имеется 4 разряда: Z - признак нулевого результата, N - признак отрицательного результата, V - признак переполнения, C - признак переноса.

CMP ИА, D - из содержимого регистра данных D вычитается содержимое по исполнительному адресу ИА, при этом формируется все разряды регистра состояний, но содержимое регистра D не меняется.

TST ИА - выработать разряд Z регистра состояний по значению, находящемуся по исполнительному адресу ИА.

BNE ИА - условный переход по признаку Z = 1 (не равно) по исполнительному адресу ИА.

BEQ ИА - условный переход по признаку Z = 0 (равно) по исполнительному адресу ИА.

BLE ИА - условный переход по признаку N or Z (меньше или равно) по исполнительному адресу ИА.

BGT ИА - условный переход по признаку not N (больше) по исполнительному адресу ИА.

BLT ИА - условный переход по признаку N (меньше) по исполнительному адресу ИА.

BRA ИА - безусловный переход по адресу ИА.

JMP ИА - безусловный переход по исполнительному адресу.

RTD размер_локальных - возврат из подпрограммы с указанием размера локальных.

LINK A, размер_локальных - в стеке сохраняется значение регистра A, в регистр A заносится указатель на это место в стеке и указатель стека продвигается на размер локальных.

UNLK A - стек сокращается на размер локальных и регистр A восстанавливается из стека.

Динамическая организация памяти

Динамическая организация памяти - это организация памяти периода исполнения программы. Оперативная память программы обычно состоит из нескольких основных разделов: стек (магазин), куча, область статических данных (инициализированных и неинициализированных). Наиболее сложной является работа со стеком. Вообще говоря, стек периода исполнения необходим для программ не на всех языках программирования. Например, в ранних версиях Фортрана нет рекурсии, так что программа может исполняться без стека. С другой стороны, исполнение программы с рекурсией может быть реализовано и без стека (того же эффекта можно достичь, например, и с помощью списковых структур). Однако, для эффективной реализации пользуются стеком, который, как правило, поддерживается на уровне машинных команд.

Рассмотрим схему организации магазина периода выполнения для простейшего случая (как, например, в языке Паскаль), когда все переменные в магазине (фактические параметры и локальные переменные) имеют известные при трансляции смещения. Магазин служит для хранения локальных переменных (и параметров) и обращения к ним в языках, допускающих рекурсивные вызовы процедур. Еще одной задачей, которую необходимо решать при трансляции языков с блочной структурой - обеспечение реализации механизмов статической вложенности. Пусть имеется следующий фрагмент программы на Паскале:

```
procedure P1;
  var V1;
  procedure P2;
    var V2;
  begin
    ...
    P2;
    V1:=...
    V2:=...
```

```
    ...
  end;
begin
  ...
  P2;
  ...
end;
```

В процессе выполнения этой программы, находясь в процедуре P2, мы должны иметь доступ к последнему экземпляру значений переменных процедуры P2 и к экземпляру значений переменных процедуры P1, из которой была вызвана P2. Кроме того, необходимо обеспечить восстановление состояния программы при завершении выполнения процедуры.

Мы рассмотрим две возможные схемы динамической организации памяти: схему со статической цепочкой и с дисплеем в памяти. В первом случае все статические контексты связаны в список, который называется статической цепочкой; в каждой записи для процедуры в магазине хранится указатель на запись статически охватывающей процедуры (помимо, конечно, указателя динамической цепочки - указателя на "базу" динамически предыдущей процедуры). Во втором случае для хранения ссылок на статические контексты используется массив, называемый дисплеем. Использование той или иной схемы определяется, помимо прочих условий, прежде всего числом адресных регистров.

Организация магазина со статической цепочкой

Итак, в случае статической цепочки магазин организован, как это изображено на рис. 9.1.

Таким образом, на запись текущей процедуры в магазине указывает регистр BP (Base Pointer), с которого начинается динамическая цепочка. На статическую цепочку указывает регистр LP (Link Pointer). В качестве регистров BP и LP в различных системах команд могут использоваться

Рис. 9.1.

универсальные, адресные или специальные регистры. Локальные переменные отсчитываются от регистра BP вверх, фактические параметры - вниз с учетом памяти, занятой точкой возврата и самим сохраненным регистром BP. Вызов подпрограмм различного статического уровня производится несколько поразному. При вызове подпрограммы того же статического уровня, что и вызывающая подпрограмма (например, рекурсивный вызов той же самой подпрограммы), выполняются следующие команды:

Занесение фактических параметров в магазин JSR A

Команда JSR A продвигает указатель SP, заносит PC на верхушку магазина и осуществляет переход по адресу A. После выполнения этих команд состояние магазина становится таким, как это изображено на рис. 9.2. Занесение BP, отведение локальных, сохранение регистров делает вызываемая подпрограмма (см. ниже).

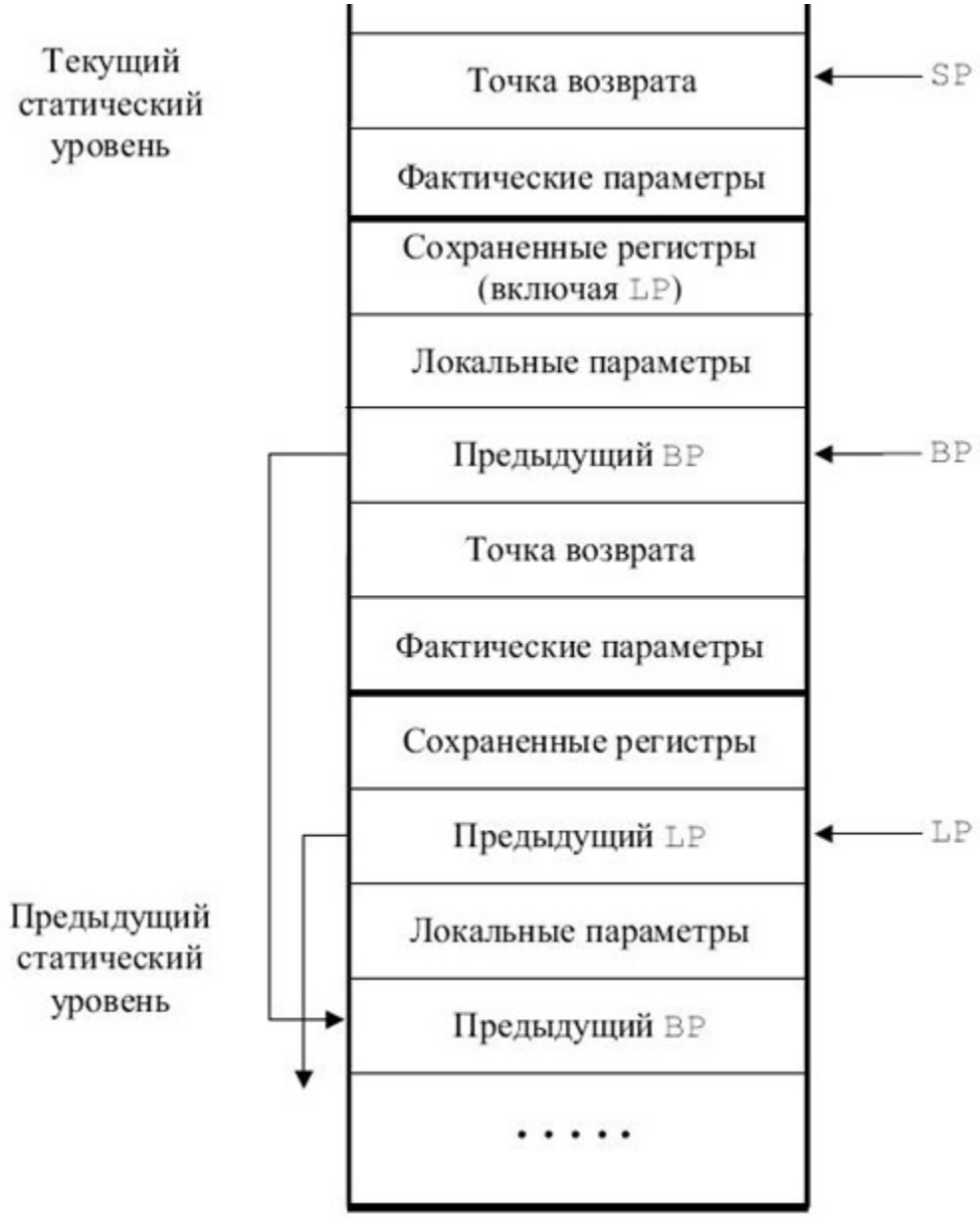


Рис. 9.2.

При вызове локальной подпрограммы необходимо установить указатель статического уровня на текущую подпрограмму, а при выходе - восстановить его на старое значение (охватывающей текущую). Для этого исполняются следующие команды:

Занесение фактических параметров в магазин MOVE BP, LP

```
SUB Delta, LP
JSR A
```

Здесь Delta - размер локальных вызывающей подпрограммы плюс двойная длина слова. Магазин после этого принимает состояние, изображенное на рис. 9.3. Предполагается, что регистр LP уже сохранен среди сохраняемых регистров, причем самым первым (сразу после локальных переменных).

После выхода из подпрограммы в вызывающей подпрограмме выполняется команда

```
MOVE (LP), LP
```

которая восстанавливает старое значение статической цепочки. Если выход осуществлялся из подпрограммы 1-го уровня, эту команду выполнять не надо, поскольку для 1-го уровня нет статической цепочки.

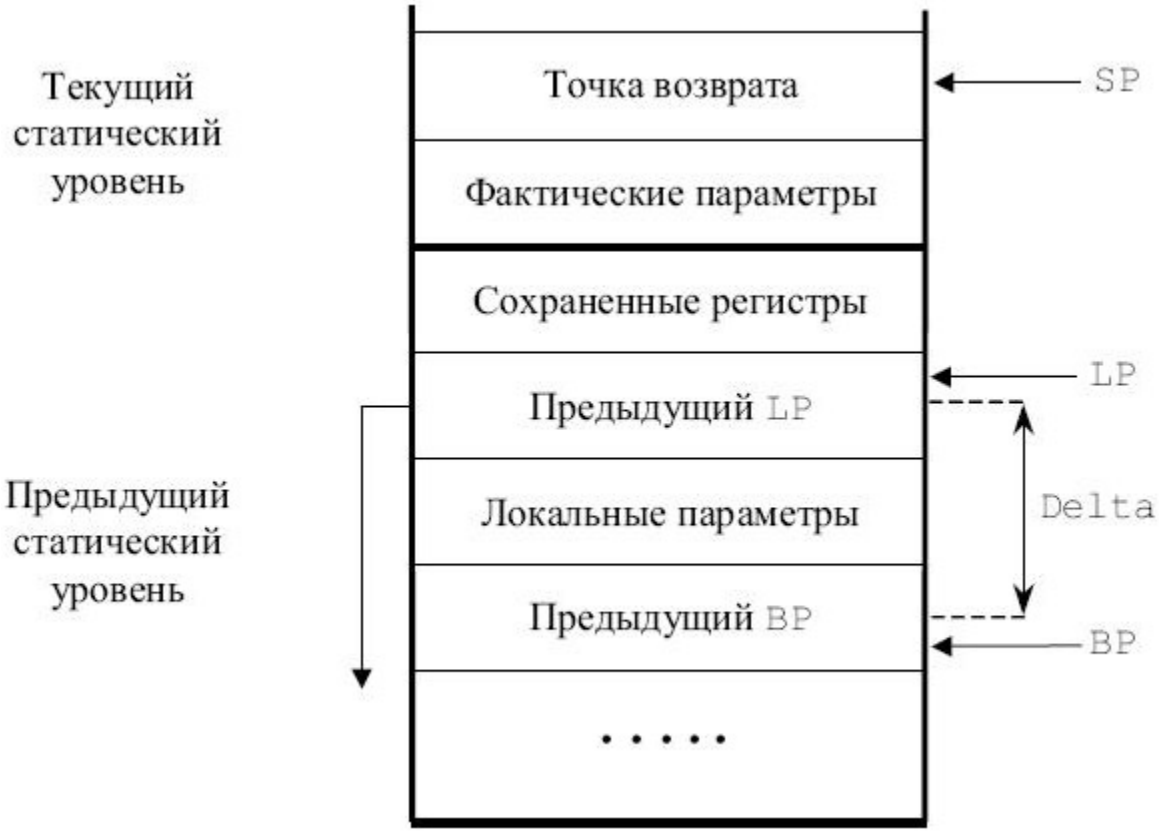


Рис. 9.3.

При вызове подпрограммы меньшего, чем вызывающая, уровня выполняются следующие команды:

```
Занесение фактических параметров в магазин
MOVE (LP), LP /* столько раз, какова разность
   уровней вызывающей и вызываемой ПП */
JSR A
```

Тем самым устанавливается статический уровень вызываемой подпрограммы. После выхода из подпрограммы выполняется команда

```
MOVE -Delta(BP), LP
```

восстанавливающая статический уровень вызывающей подпрограммы.

Тело подпрограммы начинается со следующих команд:

```
LINK BP , -размер"локальных
MOVEM -(SP)
```

Команда LINK BP, размер_локальных эквивалентна трем командам:

```
MOVE BP, -(SP)
MOVE SP, BP
ADD -размер_локальных, SP
```

Команда MOVEM сохраняет в магазине регистры. В результате выполнения этих команд магазин приобретает вид, изображенный на рис. 9.1.

Выход из подпрограммы осуществляется следующей последовательностью команд:

```
MOVEM (SP)+
UNLK BP
RTD размер_фактических
```

Команда MOVEM восстанавливает регистры из магазина.

Команда UNLK BP эквивалентна такой последовательности команд:

```
MOVE BP,SP
MOVE (SP),BP
ADD #4, SP /* 4 - размер слова */
```

Команда RTD размер_фактических, в свою очередь, эквивалентна последовательности

```
ADD размер"фактических+4, SP
JMP -размер"фактических-4(SP)
```

После ее выполнения магазин восстанавливается до состояния, которое было до вызова.

В зависимости от наличия локальных переменных, фактических параметров и необходимости сохранения регистров каждая из этих команд может отсутствовать.

Организация магазина с дисплеем

Рассмотрим теперь организацию магазина с дисплеем. Дисплей - это массив (DISPLAY) , i-й элемент которого представляет собой указатель на область активации последней вызванной подпрограммы i-го статического уровня. Доступ к переменным самой внутренней подпрограммы осуществляется через регистр BP. Дисплей может быть реализован либо через регистры (если их достаточно), либо через массив в памяти.

При вызове процедуры следующего (по отношению к вызывающей) уровня в дисплее отводится очередной элемент. Если вызывающая процедура имеет статический уровень i, то при вызове процедуры уровня j $\leq$i элементы дисплея j,...,i должны быть скопированы (обычно в стек вызывающей процедуры), текущим уровнем становится j и в DISPLAY[j] заносится указатель на область активации вызываемой процедуры. По окончании работы вызываемой процедуры содержимое дисплея восстанавливается из стека.

Иногда используется комбинированная схема - дисплей в магазине. Дисплей хранится в области активации каждой процедуры. Формирование дисплея для процедуры осуществляется в соответствии с правилами, описанными выше.

Отдельного рассмотрения требует вопрос о технике передачи фактических параметров. Конечно, в случае простых параметров (например, чисел) проблем не возникает. Однако передача массивов по значению - операция довольно дорогая, поэтому с точки зрения экономии памяти целесообразнее сначала в подпрограмму передать адрес массива, а затем уже из подпрограммы по адресу передать в магазин сам массив. В связи с передачей параметров следует упомянуть еще одно обстоятельство.

Рассмотренная схема организации магазина допустима только для языков со статически известными размерами фактических параметров. Однако, например, в языке Модула-2 по значению может быть передан гибкий массив, и в этом случае нельзя статически распределить память для параметров. Обычно в таких случаях заводят так называемый "паспорт" массива, в котором хранится вся необходимая информация, а сам массив размещается в магазине в рабочей области выше сохраненных регистров.

Назначение адресов

Назначение адресов переменным, параметрам и полям записей происходит при обработке соответствующих объявлений. В однопроходном трансляторе это может производиться вместе с построением основной таблицы символов и соответствующие адреса (или смещения) могут храниться в этой же таблице. В промежуточном представлении Лидер объявления сохранены, что делает это промежуточное представление машинно-независимым. Напомним, что в Лидер-представлении каждому описанию соответствует некоторый номер. В процессе работы генератора кодов поддерживается таблица Table, в которой по этому номеру (входу) содержится следующая информация:

- для типа: его размер;
- для переменной: смещение в области процедуры (или глобальной области);
- для поля записи: смещение внутри записи;
- для процедуры: размер локальных параметров;
- для массива: размер массива, размер элемента, значение левой и правой границы.

Для вычисления адресов определим для каждого объявления два синтезируемых атрибута: DISP будет обозначать смещение внутри области процедуры (или единицы компиляции), а SIZE - размер. Тогда семантика правила для списка объявлений принимает вид

```
RULE
DeclPart ::= ( Decl )
SEMANTICS
   Disp<1>=0;
1A: Disp<1>=Disp<1>+Size<1>;
   Size<0>=Disp<1>.
```

Все объявления, кроме объявлений переменных, имеют нулевой размер. Размер объявления переменной определяется следующим правилом:

```
RULE
Decl ::= 'VAR' TypeDes
SEMANTICS
Tablentry Entry;
0: Entry=IncTab;
  Size<0>=((Table[VAL<2>]+1) / 2)*2;
  // Выравнивание на границу слова
  Table[Entry]=Disp<0>+Size<0>.
```

В качестве примера трансляции определения типа рассмотрим обработку описания записи:

```
RULE
TypeDes ::= 'REC' ( TypeDes ) 'END'
SEMANTICS
int Disp;
Tablentry Temp;
0: Entry<0>=IncTab;
  Disp=0;
2A: {Temp=IncTab;
  Table[Temp]=Disp;
  Disp=Disp+Table[Entry<2>]+1) / 2)*2;
  // Выравнивание на границу слова
  }
Table[Entry<0>]=Disp.
```

Трансляция переменных

Переменные отражают все многообразие механизмов доступа в языке. Переменная имеет синтезированный атрибут ADDRESS - это запись, описывающая адрес в команде MC68020. Этот атрибут сопоставляется всем нетерминалам, представляющим значения. В системе команд MC68020 много способов адресации, и они отражены в структуре значения атрибута ADDRESS, имеющего следующий тип:

```
enum Register
{D0,D1,D2,D3,D4,D5,D6,D7,
   A0,A1,A2,A3,A4,A5,A6,SP,NO};

enum AddrMode
{D,A,Post,Pre,Indirect,IndPre,IndPost,IndirPC,
IndPrePC,IndPostPC,InDisp,Index,IndexPC,Abs,Imm};

struct AddrType{
  Register AddrReg,IndexReg;
  int IndexDisp,AddrDisp;
  short Scale;
};
```

Значение регистра NO означает, что соответствующий регистр в адресации не используется.

Доступ к переменным осуществляется в зависимости от их уровня: глобальные переменные адресуются с помощью абсолютной адресации; переменные в процедуре текущего уровня адресуются через регистр базы A6.

Если стек организован с помощью статической цепочки, то переменные предыдущего статического уровня адресуются через регистр статической цепочки A5; переменные остальных уровней адресуются "пробеганием" по статической цепочке с использованием вспомогательного регистра. Адрес переменной формируется при обработке структуры переменной слева направо и передается сначала сверху вниз как наследуемый атрибут нетерминала VarTail, а затем передается снизу-вверх как глобальный атрибут нетерминала Variable. Таким образом, правило для обращения к переменной имеет вид (первое вхождение Number в правую часть - это уровень переменной, второе - ее Лидер-номер):

```
RULE
Variable ::= VarMode Number Number VarTail
SEMANTICS
int Temp;
struct AddrType AddrTmp1, AddrTmp2;
3: if (Val<2>==0) // Глобальная переменная
   {Address<4>.AddrMode=Abs;
   Address<4>.AddrDisp=0;
   }
  else // Локальная переменная
   {Address<4>.AddrMode=Index;
    if (Val<2>==Level<Block>) // Переменная
      // текущего уровня
     Address<4>.AddrReg=A6;
    else if (Val<2>==Level<Block>-1)
      // Переменная предыдущего уровня
      Address<4>.AddrReg=A5;
    else
      {Address<4>.Addreg=
        GetFree(RegSet<Block>);
      AddrTmp1.AddrMode=Indirect;
      AddrTmp1.AddrReg=A5;
      Emit2(MOVEA,AddrTmp1,
      Address<4>.AddrReg);
      AddrTmp1.AddrReg=Address<4>.AddrReg;
      AddrTmp2.AddrMode=A;
      AddrTmp2.AddrReg=Address<4>.AddrReg;
      for (Temp=Level<Block>-Val<2>;
         Temp>=2;Temp--)
      Emit2(MOVEA,AddrTmp1,AddrTmp2);
     }
   if (Val<2>==Level<Block>)
    Address<4>.AddrDisp=Table[Val<3>];
   else
    Address<4>.AddrDisp=
      Table[Val<3>]+Table[LevelTab[Val<2>]];
   }.
```

Функция GetFree выбирает очередной свободный регистр (либо регистр данных, либо адресный регистр) и отмечает его как использованный в атрибуте RegSet нетерминала Block. Процедура Emit2 генерирует двухадресную команду. Первый параметр этой процедуры - код команды, второй и третий параметры имеют тип AddrType и служат операндами команды. Смещение переменной текущего уровня отсчитывается от базы (A6), а других уровней - от указателя статической цепочки, поэтому оно определяется как алгебраическая сумма размера локальных параметров и величины смещения переменной. Таблица LevelTab - это таблица уровней процедур, содержащая указатели на последовательно вложенные процедуры.

Если стек организован с помощью дисплея, то трансляция для доступа к переменным может быть осуществлена следующим образом:

```
RULE
Variable ::= VarMode Number Number VarTail
SEMANTICS
int Temp;
3: if (Val<2>==0) // Глобальная переменная
   {Address<4>.AddrMode=Abs;
   Address<4>.AddrDisp=0;
   }
  else // Локальная переменная
   {Address<4>.AddrMode=Index;
    if (Val<2>=Level<Block>) // Переменная
      // текущего уровня
    {Address<4>.AddrReg=A6;
    Address<4>.AddrDisp=Table[Val<3>];
    }
   else
    {Address<4>.AddrMode=IndPost;
    Address<4>.AddrReg=NO;
    Address<4>.IndexReg=NO;
    Address<4>.AddrDisp=Display[Val<2>];
    Address<4>.IndexDisp=Table[Val<3>];
   }
  }.
```

Рассмотрим трансляцию доступа к полям записи. Она описывается следующим правилом (Number - это Лидер- номер описания поля):

```
RULE
VarTail ::= 'FIL' Number VarTail
SEMANTICS
if (Address<0>.AddrMode==Abs)
  {Address<3>.AddrMode=Abs;
  Address<3>.AddrDisp=
  Address<0>.AddrDisp+Table[Val<2>];
  }
 else
  {Address<3>=Address<0>;
  if (Address<0>.AddrMode==Index)
   Address<3>.AddrDisp=
   Address<0>.AddrDisp+Table[Val<2>];
  else
   Address<3>.IndexDisp=
   Address<0>.IndexDisp+Table[Val<2>];
  }.
```

Трансляция целых выражений

Трансляция выражений различных типов управляется синтаксически благодаря наличию указателя типа перед каждой операцией. Мы рассмотрим некоторые наиболее характерные проблемы генерации кода для выражений.

Система команд MC68020 обладает двумя особенностями, сказывающимися на генерации кода для арифметических выражений (то же можно сказать и о генерации кода для выражений типа "множества"):

1. один из операндов выражения (правый) должен при выполнении операции находиться на регистре, поэтому если оба операнда не на регистрах, то перед выполнением операции один из них надо загрузить на регистр;
2. система команд довольно "симметрична", то есть нет специальных требований к регистрам при выполнении операций (таких, например, как пары регистров или требования четности и т.д.).

Поэтому выбор команд при генерации арифметических выражений определяется довольно простыми таблицами решений. Например, для целочисленного сложения такая таблица приведена на рис. 9.4.

Левый операнд A1 \ Правый операнд A2	R	V
R	ADD A1,A2	ADD A2, A1
V	ADD A1, A2	MOVE A1, R ADD A2, R

Рис. 9.4.

Здесь имеется в виду, что R - операнд на регистре, V - переменная или константа. Такая таблица решений должна также учитывать коммутативность операций.

```
RULE
IntExpr ::= 'PLUS' IntExpr IntExpr
SEMANTICS
if (Address<2>.AddrMode!=D) &&
      (Address<3>.AddrMode!=D)
   {Address<0>.AddrMode=D;
   Address<0>.Addreg=GetFree(RegSet<Block>);
   Emit2(MOVE,Address<2>,Address<0>);
   Emit2(ADD,Address<2>,Address<0>);
   }
else
   if (Address<2>.AddrMode==D)
      {Emit2(ADD,Address<3>,Address<2>);
      Address<0>:=Address<2>);
      }
   else {Emit2(ADD,Address<2>,Address<3>);
      Address<0>:=Address<3>);
      }.
```

Трансляция арифметических выражений

Одной из важнейших задач при генерации кода является распределение регистров. Рассмотрим хорошо известную технику распределения регистров при трансляции арифметических выражений, называемую алгоритмом Сети-Ульмана. (Замечание: в целях большей наглядности, в данном параграфе мы немного отступаем от семантики арифметических команд MC68020 и предполагаем, что команда

Op Arg1, Arg2

выполняет действие Arg2:=Arg1 Op Arg2.)

Пусть система команд машины имеет неограниченное число универсальных регистров, в которых выполняются арифметические команды. Рассмотрим, как можно сгенерировать код, используя для данного арифметического выражения минимальное число регистров.

Пусть имеется синтаксическое дерево выражения. Предположим сначала, что распределение регистров осуществляется по простейшей схеме сверху-вниз слева- направо, как изображено на рис. 9.5. Тогда к моменту генерации кода для поддерева LR занято n регистров. Пусть поддерево L требует n_l регистров, а поддерево R - n_r регистров. Если $n_l = n_r$, то при вычислении L будет использовано n_l регистров и под результат будет занят (n+1)-

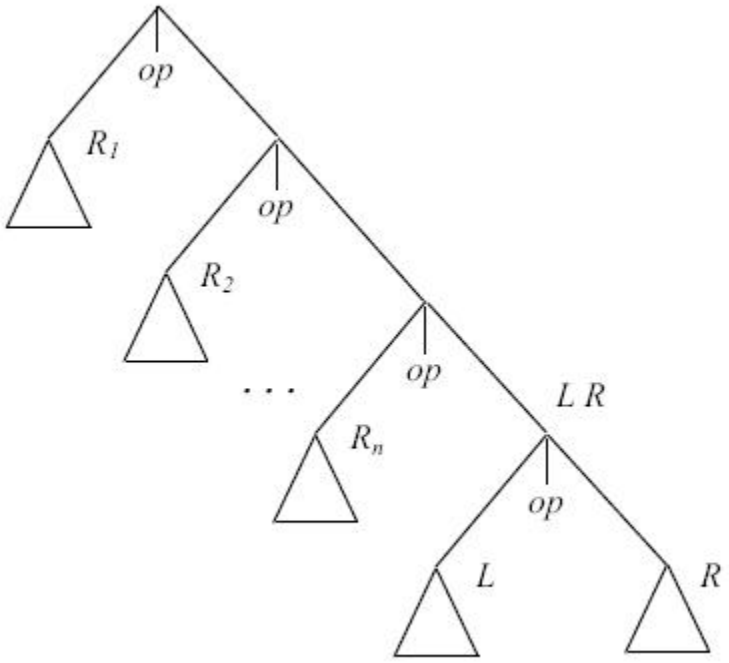


Рис. 9.5.

й регистр. Еще $n_r(= n_l)$ регистров будет использовано при вычислении R. Таким образом, общее число использованных регистров будет равно n + nl + 1.

Если $n_l > n_r$, то при вычислении L будет использовано n_l регистров. При вычислении R будет использовано $n_r < n_l$ регистров, и всего будет использовано не более чем $n + n_l$ регистров. Если $n_l < n_r$, то после вычисления L под результат будет занят один регистр (предположим, (n + 1)-й) и nr регистров будет использовано для вычисления R. Всего будет использовано $n + n_r + 1$ регистров.

Видно, что для деревьев, совпадающих с точностью до порядка потомков каждой вершины, минимальное число регистров при распределении их слева-направо достигается на дереве, у которого в каждой вершине слева расположено более "сложное" поддерево, требующее большего числа регистров. Таким образом, если дерево таково, что в каждой внутренней вершине правое поддерево требует меньшего числа регистров, чем левое, то, обходя дерево слева направо, можно оптимально распределить регистры. Без перестройки дерева это означает, что если в некоторой вершине дерева справа расположено более сложное поддерево, то сначала сгенерируем код для него, а затем уже для левого поддерева.

Алгоритм работает следующим образом. Сначала осуществляется разметка синтаксического дерева по следующим правилам.

Правила разметки:

1. если вершина - правый лист или дерево состоит из единственной вершины, помечаем эту вершину числом 1, если вершина - левый лист, помечаем ее 0 (рис. 9.6).

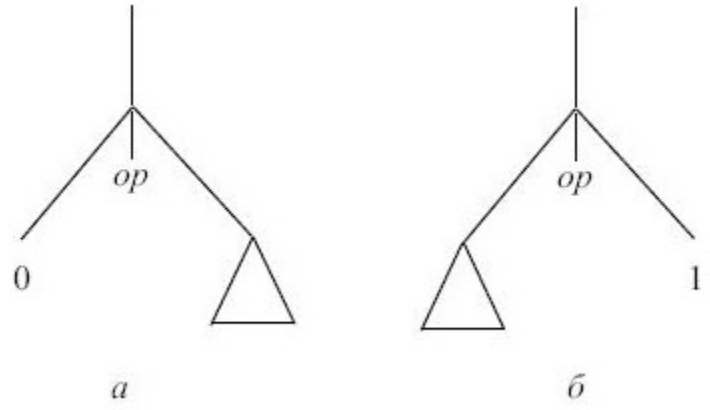


Рис. 9.6.

2. если вершина имеет прямых потомков с метками l_1 и l_2, то в качестве метки этой вершины выбираем наибольшее из чисел l_1 или l_2 либо число $l_1 + 1$, если $l_1 = l_2$ (рис. 9.6.).

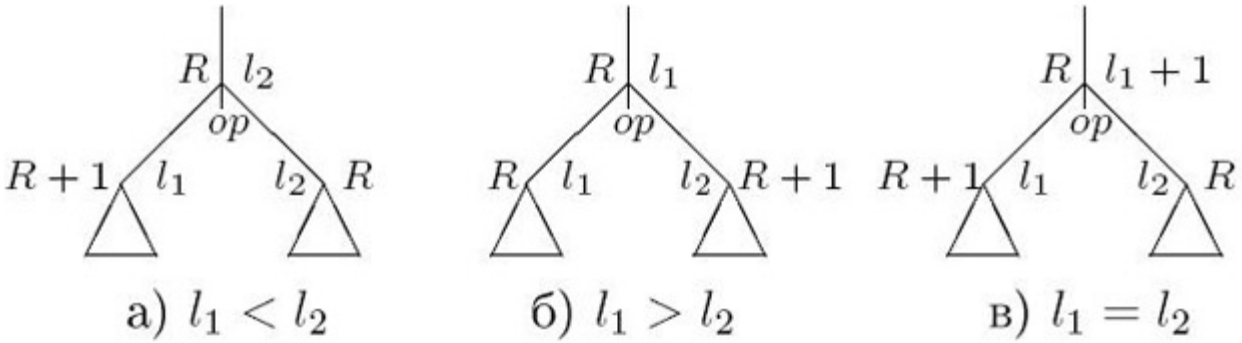


Рис. 9.7.

Эта разметка позволяет определить, какое из поддеревьев требует большего количества регистров для своего вычисления. Далее осуществляется распределение регистров для результатов операций по следующим правилам:

1. Корню назначается первый регистр.
2. Если метка левого потомка меньше метки правого, то левому потомку назначается регистр на единицу больший, чем предку, а правому - с тем же номером (сначала вычисляется правое поддерево и его результат помещается в регистр R), так что регистры занимаются последовательно. Если же метка левого потомка больше или равна метке правого потомка, то наоборот, правому потомку назначается регистр на единицу больший, чем предку, а левому - с тем же номером (сначала вычисляется левое поддерево и его результат помещается в регистр R - рис. 9.7).

После этого формируется код по следующим правилам:

1. если вершина - правый лист с меткой 1, то ей соответствует код
2. `MOVE X, R`

   где R - регистр, назначенный этой вершине, а X - адрес переменной, связанной с вершиной (рис. 9.8, б);

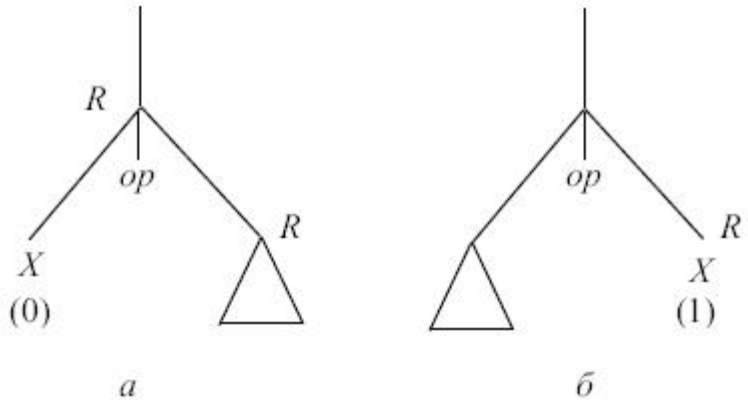


Рис. 9.8.

3. если вершина внутренняя и ее левый потомок - лист с меткой 0, то ей соответствует код

```
4.   Код правого поддерева
5.   Op X, R
```

где R - регистр, назначенный этой вершине, X - адрес переменной, связанной с вершиной, а Op - операция, примененная в вершине (рис. 9.8, а);

6. если непосредственные потомки вершины не листья и метка правой вершины больше или равна метки левой, то вершине соответствует код

```
7.   Код правого поддерева
8.   Код левого поддерева
9.   Op R+1, R
```

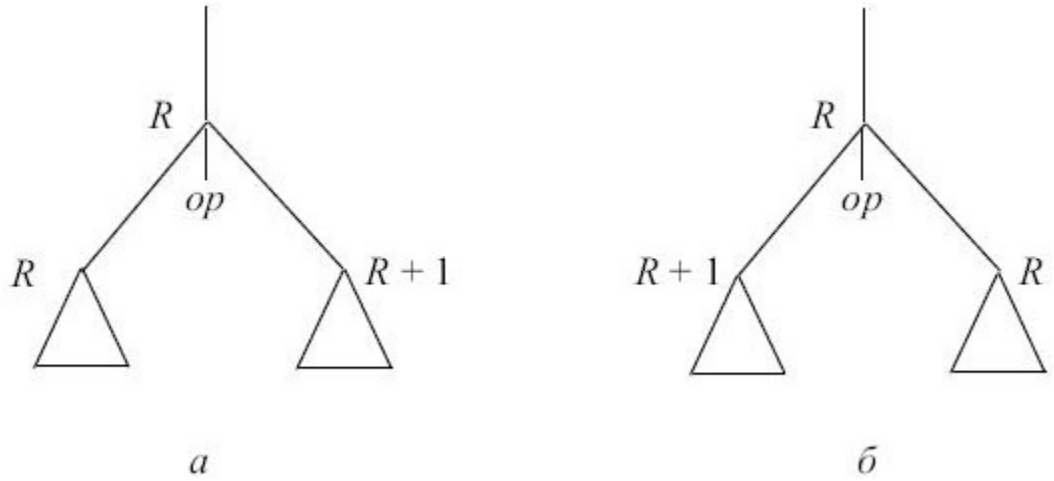


Рис. 9.9.

где R - регистр, назначенный внутренней вершине, и операция Op, вообще говоря, не коммутативная (рис. 9.9, б);

10. если непосредственные потомки вершины не листья и метка правой вершины меньше метки левой вершины, то вершине соответствует код

```
11.  Код левого поддерева
12.  Код правого поддерева
13.  Op R, R+1
14.  MOVE R+1, R
```

Последняя команда генерируется для того, чтобы получить результат в нужном регистре (в случае коммутативной операции ее операнды можно поменять местами и избежать дополнительной пересылки - рис. 9.9, а).

Рассмотрим атрибутную схему, реализующую эти правила генерации кода (для большей наглядности входная грамматика соответствует обычной инфиксной записи, а не Лидер-представлению). В этой схеме генерация кода происходит не непосредственно в процессе обхода дерева, как раньше, а из-за необходимости переставлять поддеревья код строится в виде текста с помощью операции конкатенации. Практически, конечно, это нецелесообразно: разумнее управлять обходом дерева непосредственно, однако для простоты мы будем пользоваться конкатенацией.

Листинг 9.1. (html, txt)

Атрибутированное дерево для выражения A*B+C*(D+E) приведено на рис. 9.10. При этом будет сгенерирован следующий код:

```
MOVE B, R1
MUL A, R1
MOVE E, R2
ADD D, R2
MUL C, R2
ADD R1, R2
MOVE R2, R1
```

Приведенная атрибутная схема требует двух проходов по дереву выражения. Рассмотрим теперь другую атрибутную схему, в которой достаточно одного обхода для генерация

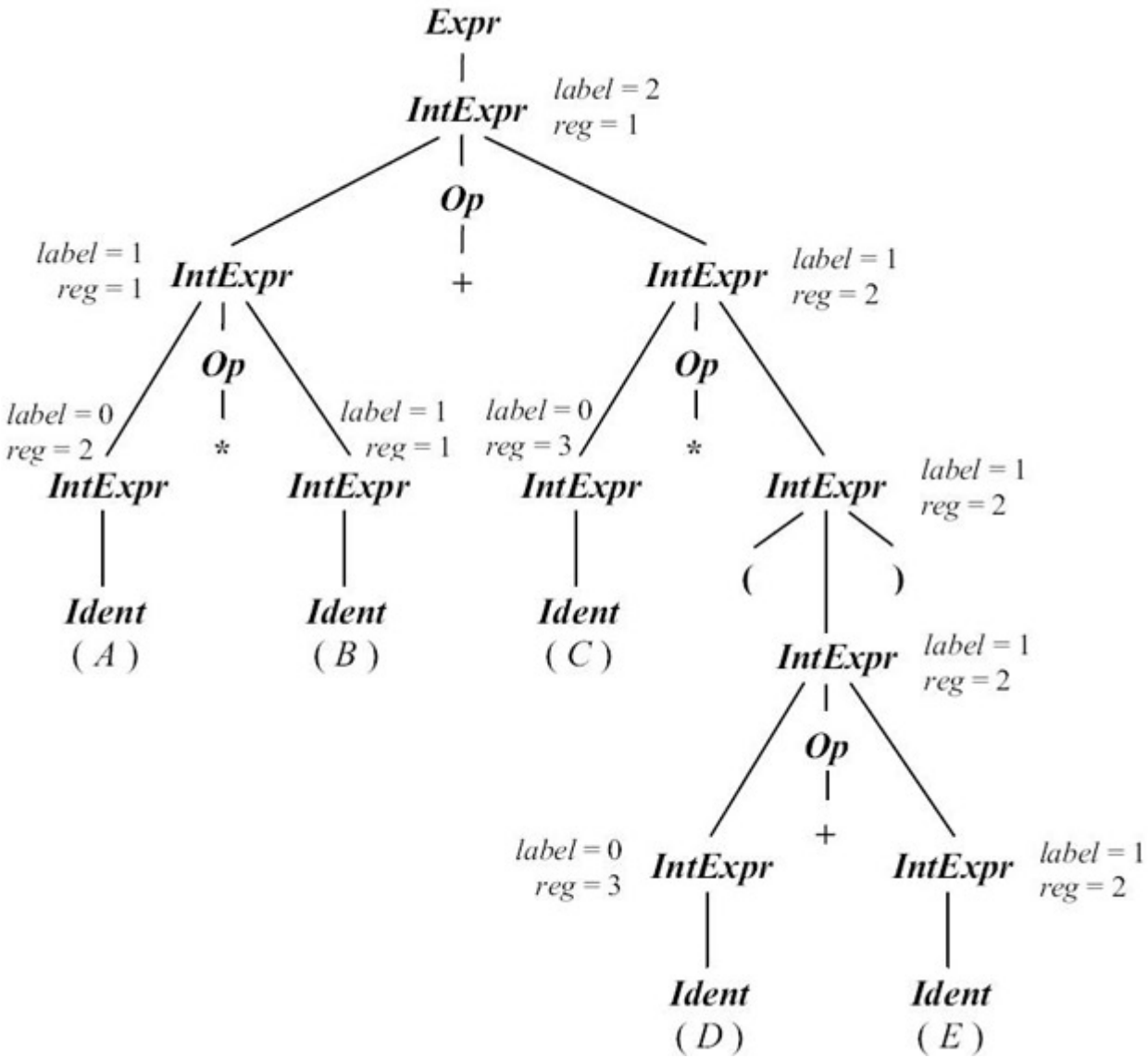


Рис. 9.10.

программы для выражений с оптимальным распределением регистров [9].

Пусть мы произвели разметку дерева разбора так же, как и в предыдущем алгоритме. Назначение регистров будем производить следующим образом.

Левому потомку всегда назначается регистр, равный его метке, а правому - его метке, если она не равна метке его левого брата, и метке + 1, если метки равны. Поскольку более сложное поддерево всегда вычисляется раньше более простого, его регистр результата имеет больший номер, чем любой регистр, используемый при вычислении более простого поддерева, что гарантирует правильность использования регистров.

Приведенные соображения реализуются следующей атрибутной схемой:

Листинг 9.2. (html, txt)

Команды пересылки требуются для согласования номеров регистров, в которых осуществляется выполнение операции, с регистрами, в которых должен быть выдан результат. Это имеет смысл, когда эти регистры разные. Получиться это может из-за того, что по приведенной схеме результат выполнения операции всегда находится в регистре с номером метки, а метки левого и правого поддеревьев могут совпадать.

Для выражения A*B+C*(D+E) будет сгенерирован следующий код:

```
MOVE E, R1

ADD D, R1
MUL C, R1
MOVE R1, R2
MOVE B, R1
MUL A, R1
ADD R1, R2
```

В приведенных атрибутных схемах предполагалось, что регистров достаточно для трансляции любого выражения. Если это не так, приходится усложнять схему трансляции и при необходимости сбрасывать содержимое регистров в память (или магазин).

Трансляция логических выражений

Логические выражения, включающие логическое умножение, логическое сложение и отрицание, можно вычислять как непосредственно, используя таблицы истинности, так и с помощью условных выражений, основанных на следующих простых правилах:

A AND B эквивалентно if A then B else False,

A OR B эквивалентно if A then True else B.

Если в качестве компонент выражений могут входить функции с побочным эффектом, то, вообще говоря, результат вычисления может зависеть от способа вычисления. В некоторых языках программирования не оговаривается, каким способом должны вычисляться логические выражения (например, в Паскале), в некоторых требуется, чтобы вычисления производились тем или иным способом (например, в Модуле-2 требуется, чтобы выражения вычислялись по приведенным формулам), в некоторых языках есть возможность явно задать способ вычисления (Си, Ада). Вычисление логических выражений непосредственно по таблицам истинности аналогично вычислению арифметических выражений, поэтому мы не будем их рассматривать отдельно. Рассмотрим подробнее способ вычисления с помощью приведенных выше фор-

мул (будем называть его "вычислением с условными переходами"). Иногда такой способ рассматривают как оптимизацию вычисления логических выражений.

Рассмотрим следующую атрибутную грамматику со входным языком логических выражений:

```
RULE
Expr ::= BoolExpr
SEMANTICS
FalseLab<1>=False; TrueLab<1>=True;
Code<0>=Code<1>.

RULE
BoolExpr ::= BoolExpr 'AND' BoolExpr
SEMANTICS
FalseLab<1>=FalseLab<0>; TrueLab<1>=NodeLab<3>;
FalseLab<3>=FalseLab<0>; TrueLab<3>=TrueLab<0>;
Code<0>=NodeLab<0> + ":" + Code<1> + Code<3>.

RULE
BoolExpr ::= BoolExpr 'OR' BoolExpr
SEMANTICS
FalseLab<1>=NodeLab<3>; TrueLab<1>=TrueLab<0>;
FalseLab<3>=FalseLab<0>; TrueLab<3>=TrueLab<0>;
Code<0>=NodeLab<0> + ":" + Code<1> + Code<3>.

RULE
BoolExpr ::= F
SEMANTICS
Code<0>=NodeLab<0> + ":" + "GOTO" + FalseLab<0>.

RULE
BoolExpr ::= T
SEMANTICS
Code<0>=NodeLab<0> + ":" + "GOTO" + TrueLab<0>.
```

Здесь предполагается, что все вершины дерева занумерованы и номер вершины дает атрибут NodeLab. Метки вершин передаются, как это изображено на рис. 9.11.

Таким образом, каждому атрибутированному дереву в этой атрибутной грамматике сопоставляется код, полученный в результате обхода дерева сверху-вниз слева-направо

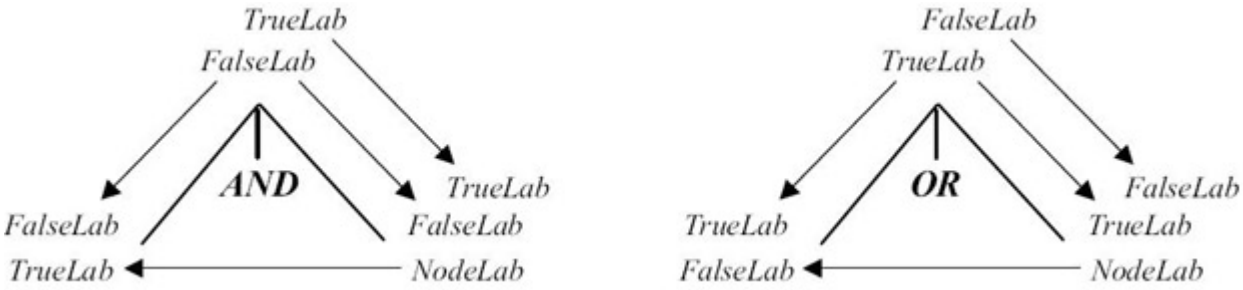


Рис. 9.11.

следующим образом. При входе в вершину BoolExpr генерируется ее номер, в вершине F генерируется текст GOTO значение атрибута FalseLab<0>, в вершине T - GOTO значение атрибута TrueLab<0>. Например, для выражения

```
F OR ( F AND T AND T ) OR T
```

получим атрибутированное дерево, изображенное на рис. 9.12, и код

```
1:7:            GOTO 2
2:8:4:9:        GOTO 3
```

```
5:10:          GOTO 6
6:             GOTO True
3:             GOTO True
True: ...
False: ...
```

Эту линеаризованную запись можно трактовать как программу вычисления логического значения: каждая строка может быть помечена номером вершины и содержать либо переход на другую строку, либо переход на True или False, что соответствует значению выражения true или false. Будем говорить, что полученная программа вычисляет (или интерпретирует) значение выражения, если в результате ее выполнения (от первой строки) мы придем к строке, содержащей GOTO True или GOTO False.

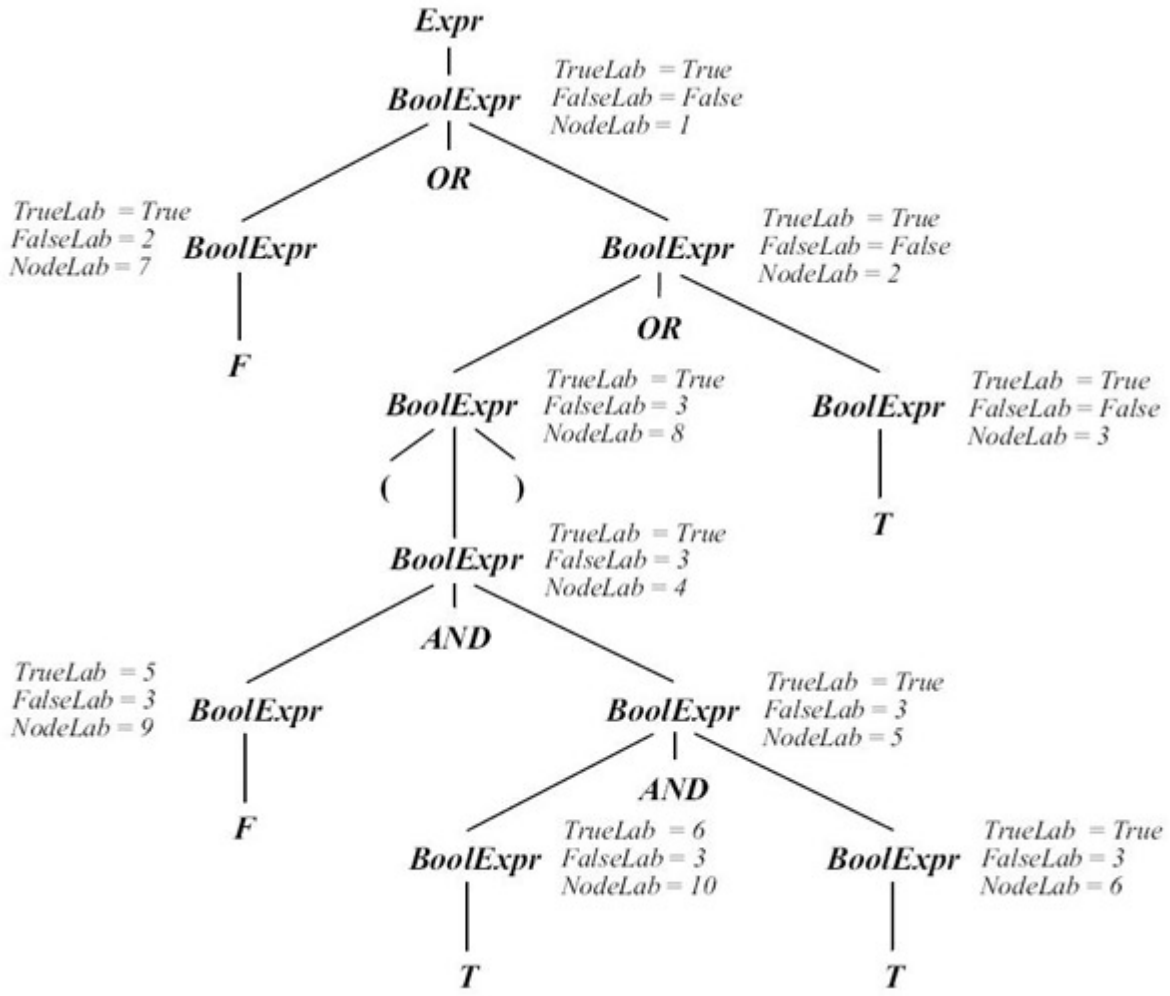


Рис. 9.12.

Утверждение 9.1. В результате интерпретации поддерева с некоторыми значениями атрибутов FalseLab и TrueLab в его корне выполняется команда GOTO TrueLab, если значение выражения истинно, и команда GOTO FalseLab, если значение выражения ложно.

Доказательство. Применим индукцию по высоте дерева. Для деревьев высоты 1, соответствующих правилам

```
BoolExpr ::= F и BoolExpr ::= T,
```

справедливость утверждения следует из соответствующих атрибутных правил. Пусть дерево имеет высоту n > 1. Зависимость атрибутов для дизъюнкции и конъюнкции приведена на рис. 9.13.

Если для конъюнкции значение левого поддерева ложно и по индукции вычисление левого поддерева

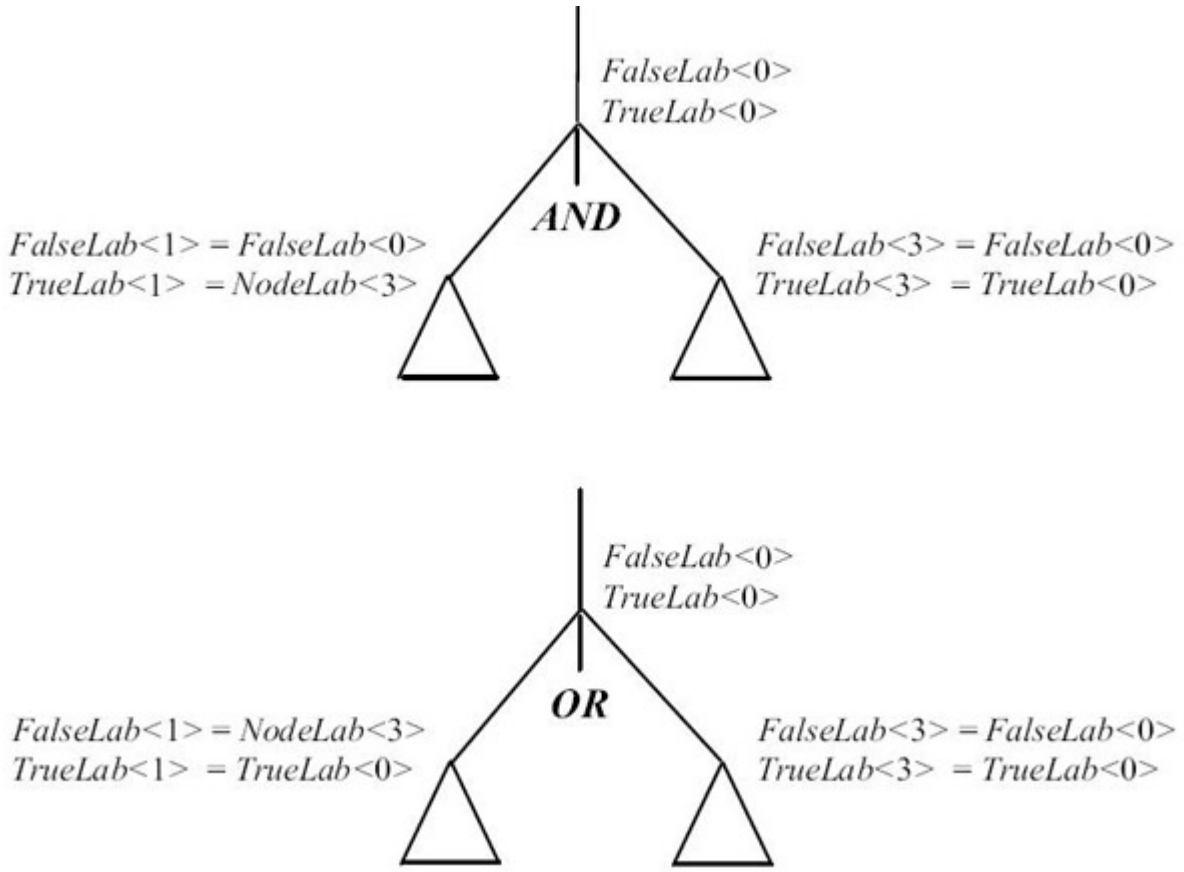


Рис. 9.13.

завершается командой GOTO FalseLab<1>, то получаем, что вычисление всего дерева завершается командой перехода GOTO FalseLab<0> (= FalseLab<1>). Если же значение левого поддерева истинно, то его вычисление завершается командой перехода GOTO TrueLab<1> (= NodeLab<3>). Если значение правого поддерева ложно, то вычисление всего дерева завершается командой GOTO FalseLab<0> (= FalseLab<3>). Если же оно истинно, вычисление всего дерева завершается командой перехода GOTO TrueLab<0> (= TrueLab<3>). Аналогично - для дизъюнкции.

Утверждение 9.2. Для любого логического выражения, состоящего из констант, программа, полученная в результате обхода дерева этого выражения, завершается со значением логического выражения в обычной интерпретации, то есть осуществляется переход на True для значения, равного true, и переход на метку False для значения false.

Доказательство. Это утверждение является частным случаем предыдущего. Его справедливость следует из того, что метки корня дерева равны соответственно TrueLab = True и FalseLab = False.

Добавим теперь новое правило в предыдущую грамматику:

```
RULE
BoolExpr ::= Ident
SEMANTICS
Code<0>=NodeLab<0> + ":" + "if (" + Val<1> +
      "==T) GOTO" + TrueLab<0> + "else GOTO" +
      FalseLab<0>.
```

Тогда, например, для выражения A OR (B AND C AND D) OR E получим следующую программу:

```
1:7:           if (A==T) GOTO True else GOTO 2
2:8:4:9:       if (B==T) GOTO 5 else GOTO 3
5:10:          if (C==T) GOTO 6 else GOTO 3
6:             if (D==T) GOTO True else GOTO 3
3:             if (E==T) GOTO True else GOTO False
True: ...
```

```
False: ...
```

При каждом конкретном наборе данных эта программа превращается в программу вычисления логического значения.

Утверждение 9.3. В каждой строке программы, сформированной предыдущей атрибутной схемой, одна из меток внутри условного оператора совпадает с меткой следующей строки.

Доказательство. Действительно, по правилам наследования атрибутов TrueLab и FalseLab, в правилах для дизъюнкции и конъюнкции либо атрибут FalseLab, либо атрибут TrueLab принимает значение метки следующего поддерева. Кроме того, как значение FalseLab, так и значение TrueLab, передаются в правое поддерево от предка. Таким образом, самый правый потомок всегда имеет одну из меток TrueLab или FalseLab, равную метке правого брата соответствующего поддерева. Учитывая порядок генерации команд, получаем справедливость утверждения.

Дополним теперь атрибутную грамматику следующим образом:

Листинг 9.3. (html, txt)

Правила наследования атрибута Sign приведены на рис. 9.14.

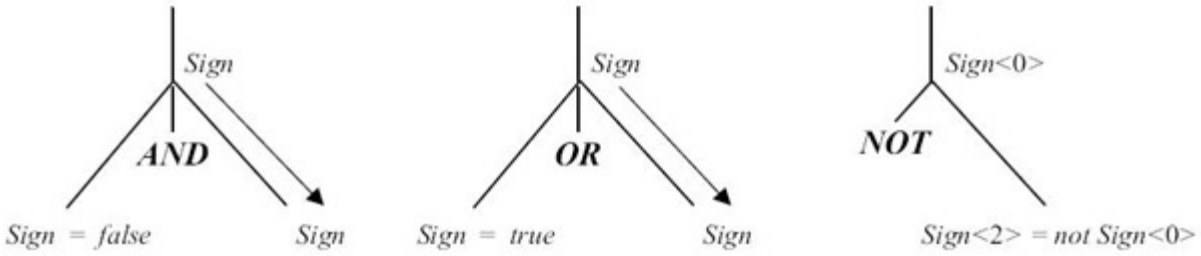


Рис. 9.14.

Программу желательно сформировать таким образом, чтобы else-метка была как раз меткой следующей вершины. Как это можно сделать, следует из следующего утверждения.

Утверждение 9.4. В каждой терминальной вершине, метка ближайшего правого для нее поддерева равна значению атрибута FalseLab этой вершины, тогда и только тогда, когда значение атрибута Sign этой вершины равно true, и наоборот, метка ближайшего правого для нее поддерева равна значению атрибута TrueLab этой вершины, тогда и только тогда, когда значение атрибута Sign равно false.

Доказательство. Действительно, если ближайшей общей вершиной является AND, то в левого потомка передается Node- Lab правого потомка в качестве TrueLab и одновременно Sign правого потомка равен true. Если же ближайшей общей вершиной является OR, то в левого потомка передается NodeLab правого потомка в качестве FalseLab и одновременно Sign правого потомка равен false. Во все же правые потомки значения TrueLab, FalseLab и Sign передаются из предка (за исключением правила для NOT, в котором TrueLab и FalseLab меняются местами, но одновременно на противоположный меняется и Sign).

Эти два утверждения (3 и 4) позволяют заменить последнее правило атрибутной грамматики следующим образом:

```
RULE
BoolExpr ::= Ident
SEMANTICS
Code<0>=NodeLab<0> + ":" +
   (Sign<0>
   ? "if (" + Val<1> + "==T) GOTO" + TrueLab<0>
```

```
   : "if (" + Val<1> + "==F) GOTO" + FalseLab<0>).
```

В свою очередь, при генерации машинных команд это правило можно заменить на следующее:

```
RULE
BoolExpr ::= Ident
SEMANTICS
Code<0>=NodeLab<0> + ":" + "TST" + Val<1> +
   (Sign<0>
   ? "BNE" + TrueLab<0>
   : "BEQ" + FalseLab<0>).
```

Таким образом, для выражения A OR (B AND C AND D) OR E получим следующий код на командах перехода:

```
1:7:            TST A
      BNE True
2:8:4:9:        TST B
      BEQ 3
5:10:           TST C
      BEQ 3
6:              TST D
      BNE True
3:              TST E
      BEQ False
True: ...
False: ...
```

Если элементом логического выражения является сравнение, то генерируется команда, соответствующая знаку сравнения (BEQ для =, BNE для <>, BGE для >= и т.д.), если атрибут Sign соответствующей вершины имеет значение true, и отрицание (BNE для =, BEQ для <>, BLT для >= и т.д.), если атрибут Sign имеет значение false.

Выделение общих подвыражений

Выделение общих подвыражений относится к области оптимизации программ. В общем случае трудно (или даже невозможно) провести границу между оптимизацией и "качественной трансляцией". Оптимизация - это и есть качественная трансляция. Обычно термин "оптимизация" употребляют, когда для повышения качества программы используют ее глубокие преобразования такие, например, как перевод в графовую форму для изучения нетривиальных свойств программы.

В этом смысле выделение общих подвыражений - одна из простейших оптимизаций. Для ее осуществления требуется некоторое преобразование программы, а именно построение ориентированного ациклического графа, о котором говорилось в главе, посвященной промежуточным представлениям.

Линейный участок - это последовательность операторов, в которую управление входит в начале и выходит в конце без остановки и перехода изнутри.

Рассмотрим дерево линейного участка, в котором вершинами служат операции, а потомками - операнды. Будем говорить, что две вершины образуют общее подвыражение, если поддеревья для них совпадают, то есть имеют одинаковую структуру и, соответственно, одинаковые операции во внутренних вершинах и одинаковые операнды в листьях. Выделение общих подвыражений позволяет генерировать для них код один раз, хотя может привести и к некоторым трудностям, о чем вкратце будет сказано ниже.

Выделение общих подвыражений проводится на линейном участке и основывается на двух положениях.Выделение общих подвыражений проводится на линейном участке и основывается на двух положениях.

1. Поскольку на линейном участке переменной может быть несколько присваиваний, то при выделении общих подвыражений необходимо различать вхождения переменных до и после присваивания. Для этого

каждая переменная снабжается счетчиком. Вначале счетчики всех переменных устанавливаются равными 0. При каждом присваивании переменной ее счетчик увеличивается на 1.

2. Выделение общих подвыражений осуществляется при обходе дерева выражения снизу вверх слева направо. При достижении очередной вершины (пусть операция, примененная в этой вершине, есть бинарная op; в случае унарной операции рассуждения те же) просматриваем общие подвыражения, связанные с op. Если имеется выражение, связанное с op и такое, что его левый операнд есть общее подвыражение с левым операндом нового выражения, а правый операнд - общее подвыражение с правым операндом нового выражения, то объявляем новое выражение общим с найденным и в новом выражении запоминаем указатель на найденное общее выражение. Базисом построения служит переменная: если операндами обоих выражений являются одинаковые переменные с одинаковыми счетчиками, то они являются общими подвыражениями. Если выражение не выделено как общее, оно заносится в список операций, связанных с op.

Рассмотрим теперь реализацию алгоритма выделения общих подвыражений. Поддерживаются следующие глобальные переменные:

Table - таблица переменных; для каждой переменной хранится ее счетчик (Count) и указатель на вершину дерева выражений, в которой переменная встретилась в последний раз в правой части (Last);

OpTable - таблица списков (типа LisType) общих подвыражений, связанных с каждой операцией. Каждый элемент списка хранит указатель на вершину дерева (поле Addr) и продолжение списка (поле List).

С каждой вершиной дерева выражения связана запись типа NodeType, со следующими полями:

Left - левый потомок вершины,

Right - правый потомок вершины,

Comm - указатель на предыдущее общее подвыражение,

Flag - признак, является ли поддерево общим подвыражением,

Varbl - признак, является ли вершина переменной,

VarCount - счетчик переменной. Выделение общих подвыражений и построение дерева осуществляются приведенными ниже правилами. Атрибут Entry нетерминала Variable дает указатель на переменную в таблице Table. Атрибут Val символа Op дает код операции. Атрибут Node символов IntExpr и Assignment дает указатель на запись типа NodeType соответствующего нетерминала.

Листинг 9.4. (html, txt)

Рассмотрим теперь некоторые простые правила распределения регистров при наличии общих подвыражений. Если число регистров ограничено, можно выбрать один из следующих двух вариантов.

1. При обнаружении общего подвыражения с подвыражением в уже просмотренной части дерева (и, значит, с уже распределенными регистрами) проверяем, расположено ли его значение на регистре. Если да, и если регистр после этого не менялся, заменяем вычисление поддерева на значение в регистре. Если регистр менялся, то вычисляем подвыражение заново.
2. Вводим еще один проход. На первом проходе распределяем регистры. Если в некоторой вершине обнаруживается, что ее поддерево общее с уже вычисленным ранее, но значение регистра потеряно, то в такой вершине на втором проходе необходимо сгенерировать команду сброса регистра в рабочую память. Выигрыш в коде будет, если стоимость команды сброса регистра + доступ к памяти в повторном использовании этой памяти не превосходит стоимости заменяемого поддерева. Поскольку стоимость команды MOVE известна, можно сравнить стоимости и принять оптимальное решение: пометить предыдущую вершину для сброса либо вычислять поддерево полностью.

Трансляция объектно-ориен- тированных свойств языков программирования

В этом разделе будут рассмотрены механизмы трансляции базовых конструкций объектно-ориентированных языков программирования, а именно наследования и виртуальных функций на примере языка C++.

Виртуальные базовые классы

К описателю базового класса можно добавить ключевое слово virtual. В этом случае единственный подобъект виртуального базового класса разделяется каждым базовым классом, в котором тот, исходный, базовый класс определен как виртуальный.

Пусть мы имеем следующую иерархию наследования:

```
class L {. . . }
class A : public virtual L {. . . }
class B : public virtual L {. . . }
class C : public A, public B {. . . }
```

Это можно изобразить следующей диаграммой классов:

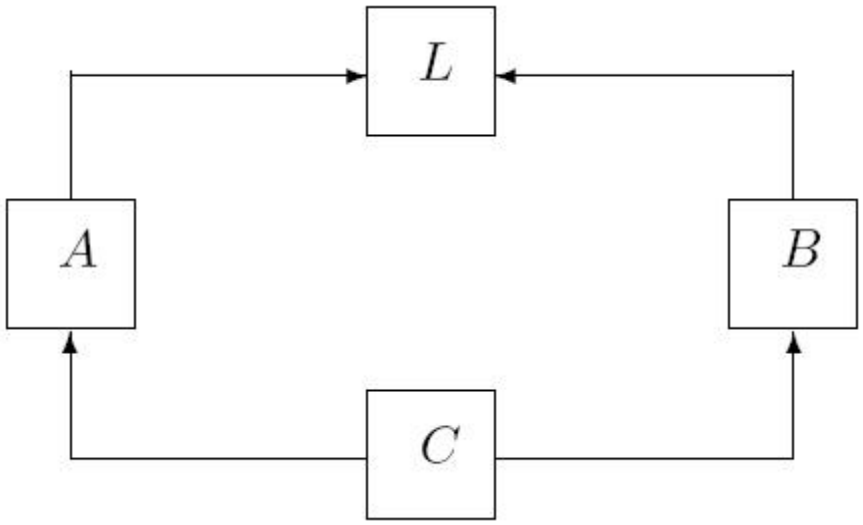


Рис. 9.15. Диаграмма классов

Каждый объект A или объект B будет содержать L, но в объекте C будет существовать лишь один объект класса L. Ясно, что представление объекта виртуального базового класса L не может быть в одной и той же позиции относительно и A, и B для всех объектов. Следовательно, во всех объектах классов, которые включают класс L как виртуальный базовый класс, должен храниться указатель на L. Реализация A, B и C объектов могла бы выглядеть следующим образом:

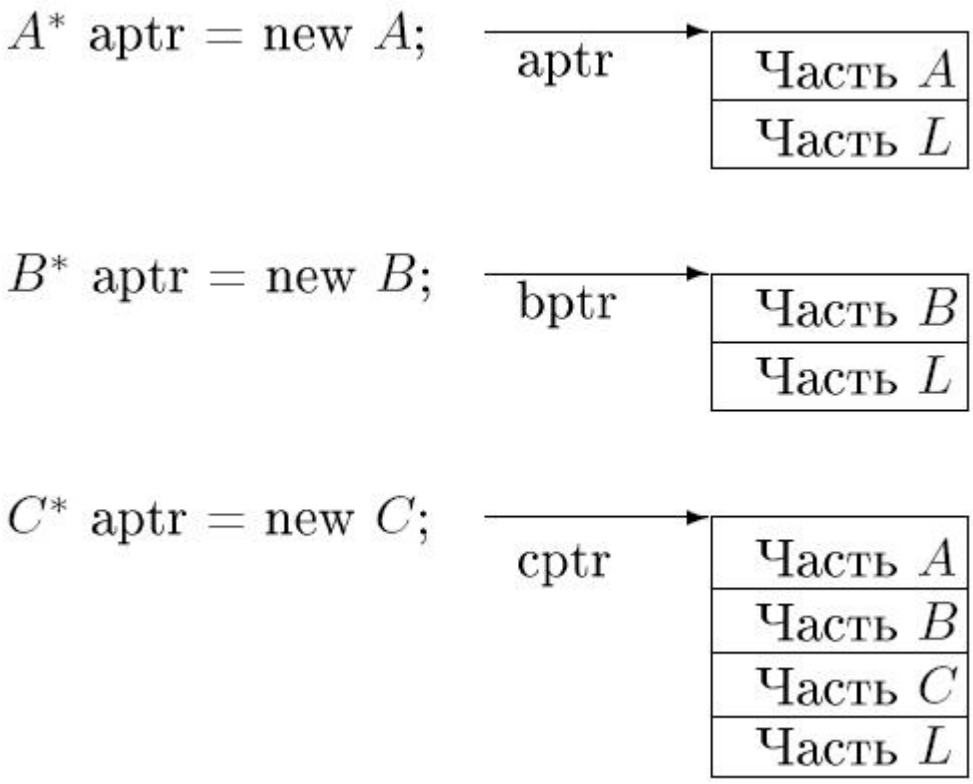


Рис. 9.16. Реализация A, B и C объектов

Множественное наследование

Имея два класса

```
class A {. . . af (int);}
class B {. . . bf (int); }
```

можно объявить третий класс с этими двумя в качестве базовых:

```
class C : public A, public B {. . . }
```

Объект класса C может быть размещен как непрерывный объект вида:

Рис. 9.17.

Как и в случае с единичным наследованием, здесь не гарантируется порядок выделения памяти для базовых классов, поэтому объект класса C может выглядеть и так:

Рис. 9.18.

Доступ к члену класса A, B или C реализуется в точности так же, как и для единичного наследования: компилятор знает положение в объекте каждого члена и порождает соответствующий код.

Если объект размещен в памяти в соответствии с первой диаграммой: сначала часть A объекта, а затем части B и C, то вызов функции - члена класса A или C будет таким же, как вызов функции-члена при единичном наследовании. Вызов функции-члена класса B для объекта, заданного указателем на C, реализуется несколько сложнее. Рассмотрим

```
C* pc = new C;
pc → bf(2);
```

Функция B :: bf() естественно предполагает, что ее параметр this является указателем на B. Чтобы получить указатель на часть B объекта C, следует добавить к указателю pc смещение B относительно C - константу времени компиляции, которую мы будем называть delta(B). Соотношение указателя pc и указателя this, передаваемого в B::bf, показано ниже.

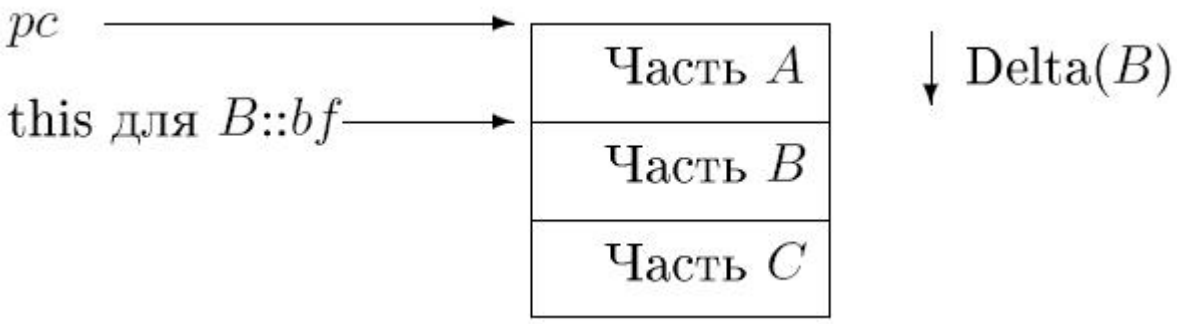


Рис. 9.19.

Единичное наследование и виртуальные функции

Если класс base содержит виртуальную функцию vf, а класс derived, порожденный по классу base, также содержит функцию vf того же типа, то обращение к vf для объекта класса derived вызывает derived :: vf даже при доступе через указатель или ссылку на base. В таком случае говорят, что функция производного класса подменяет (override) функцию базового класса. Если, однако, типы этих функций различны, то функции считаются различными и механизм виртуальности не включается.

Виртуальные функции можно реализовать при помощи таблицы указателей на виртуальные функции vtbl. В случае единичного наследования таблица виртуальных функций класса будет содержать ссылки на соответствующие функции, а каждый объект данного класса будет содержать указатель на таблицу vtbl.

```
class A {
  public:
    int a;
    virtual void f(int);
    virtual void g(int);
    virtual void h(int);
};
class B : public A {
  public:
    int b;
    void g(int);
};
class C : public B {
```

```
  public:
    int c;
    void h(int);
};
```

Объект класса C будет выглядеть примерно так:

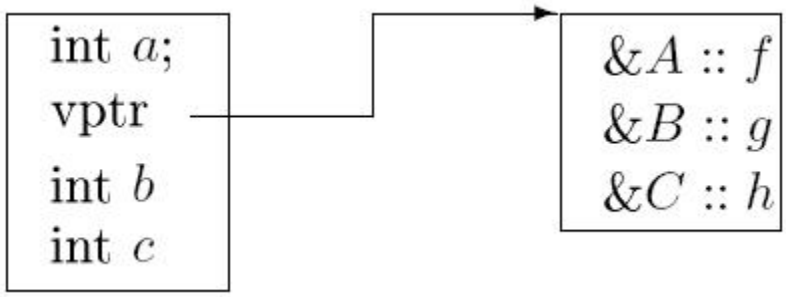


Рис. 9.20.

Множественное наследование и виртуальные функции

При множественном наследовании виртуальные функции реализуются несколько сложнее. Рассмотрим следующие объявления:

```
class A {
  public:
    virtual void f(int);
};
class B : {
  public:
    virtual void f(int);
    virtual void g(int);
};
class C : public A, public B {
  public:
    void f();
};
```

Поскольку класс A порожден по классам A и B, каждый из следующих вызовов будет обращаться к C :: f() (считая, что каждый из трех указателей смотрит на объект класса C):

```
pa → f()
pb → f()
pc → f()
```

Рассмотрим, для примера, вызов pb →f(). При входе в C :: f указатель this должен указывать на начало объекта C, а не на часть B в нем. Во время компиляции вообще говоря не известно, указывает ли pb на часть B в C. Например, из-за того, что pb может быть присвоен просто указатель на объект B. Так что величина delta(B), упомянутая выше, может быть различной для разных объектов в зависимости от структуры классов, порождаемых из B и должна где-то хранится во время выполнения.

Следовательно, delta(B) должно где-то храниться и быть доступно во время исполнения. Поскольку это смещение нужно только для виртуального вызова функции, логично хранить его в таблице виртуальных функций.

Указатель this, передаваемый виртуальной функции, может быть вычислен путем вычитания смещения объекта, для которого была определена виртуальная функция, из смещения объекта, для которого она вызвана, а затем вычитания этой разности из указателя, используемого при вызове. Здесь значение delta(B) будет необходимо для поиска начала объекта (в нашем случае C), содержащего B, по указателю

на B. Сгенерированный код вычтет значение delta(B) из значения указателя, так что хранится смещение со знаком минус, -delta(B). Объект класса C будет выглядеть следующим образом:

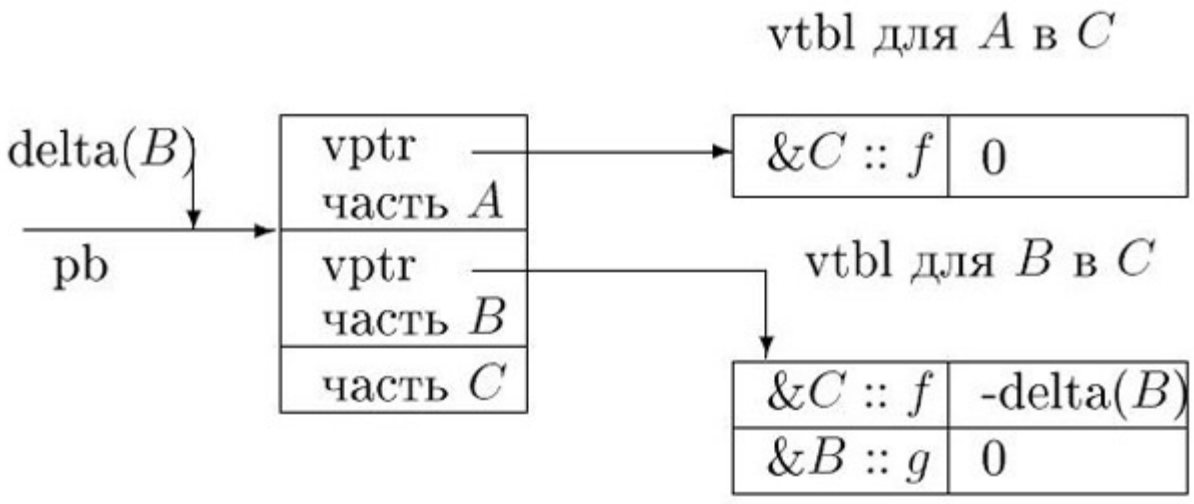


Рис. 9.21.

Таблица виртуальных функций vtbl для B в C отличается от vtbl для отдельно размещенного B. Каждая комбинация базового и производного классов имеет свою таблицу vtbl. В общем случае объект производного класса требует таблицу vtbl для каждого базового класса плюс таблицу для производного класса, не считая того, что производный класс может разделять таблицу vtbl со своим первым базовым классом. Таким образом, для объекта типа C в этом примере требуется две таблицы vtbl (таблица для A в C объединена с таблицей для C объекта и еще одна таблица нужна для B объекта в C).

Виртуальные базовые классы с виртуальными функциями

При наличии виртуальных базовых классов построение таблиц для вызовов виртуальных функций становится более сложным. Рассмотрим следующие объявления:

```
class W {
  public:
    virtual void f();
    virtual void g();
    virtual void h();
    virtual void k();
};
class MW : public virtual W {
  public:
    void g();
};
class BW : public virtual W {
  public:
    void f();
};
class BMW : public BW, public MW, public virtual W {
  public:
    void h();
};
```

Отношение наследования для этого примера может быть изображено в виде ациклического графа таким образом:

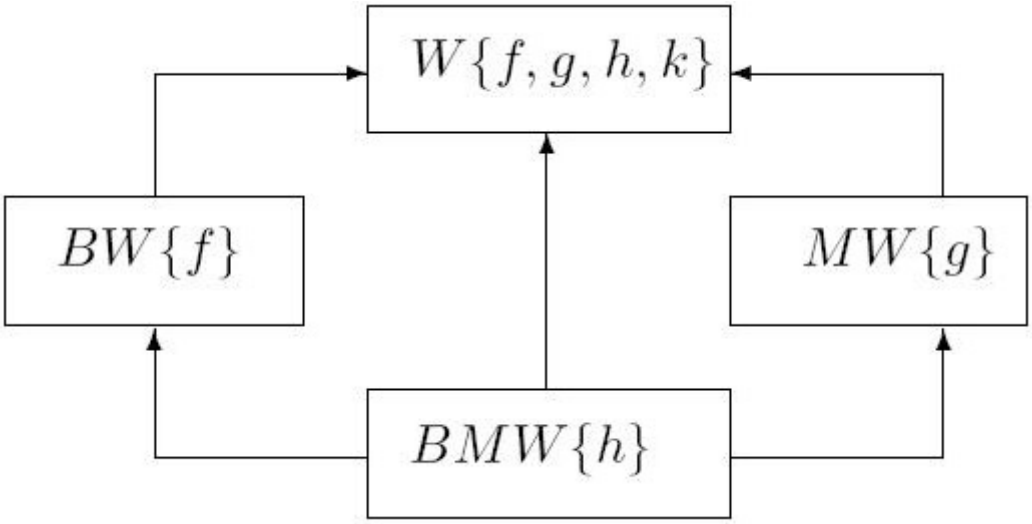


Рис. 9.22.

Функции-члены класса BMW могут использоваться, например, так:

```
void g(BMW¤pbmw)
{pbmw ! f(); == вызывает BW :: f()
pbmw ! g(); == вызывает MW :: g()
pbmw ! h(); == вызывает BMW :: h()
}
```

Рассмотрим теперь следующий вызов виртуальной функции f():

```
  void h(BMW*pbmw)
  {MW*pmw = pbmw;
  pmw ! f(); == вызывает BW :: f(); потому, что
  // pbmw указывает на BMW, для которого f бер"тся из
BW!
  }
```

Виртуальный вызов функции по одному пути в структуре наследования может привести к обращению к функции, переопределенной на другом пути.

Структура объектов класса BMW и его таблиц виртуальных функций vtbl могут выглядеть следующим образом:

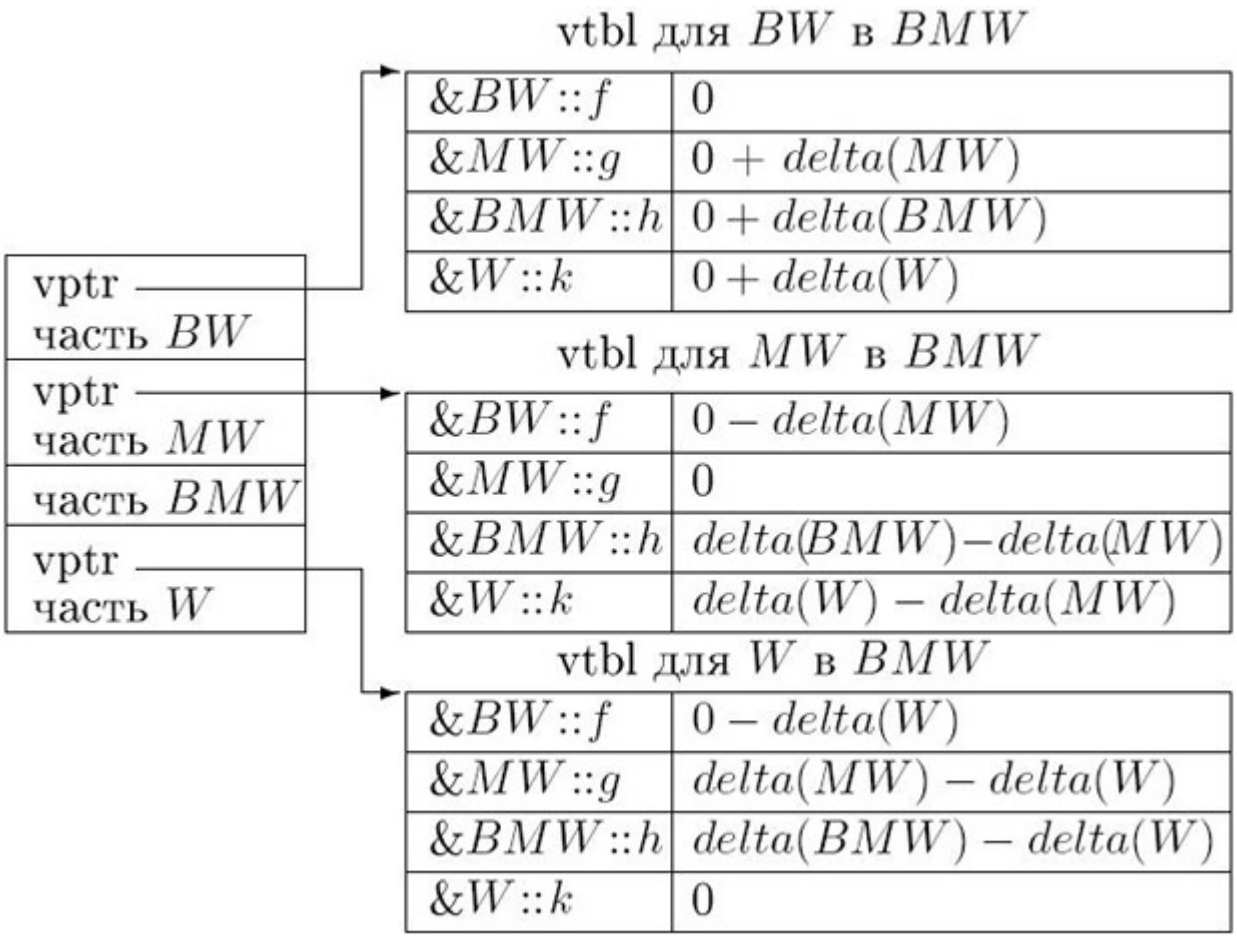


Рис. 9.23.

Виртуальной функции должен быть передан указатель this на объект класса, в котором эта функция описана. Поэтому следует хранить смещение для каждого указателя функции из vtbl. Когда объект размещен в памяти так, как это изображено выше, смещение, хранимое с указателем виртуальной функции, исчисляется вычитанием смещения класса, для которого эта таблица vtbl создана, из смещения класса, поставляющего эту функцию. Рассмотрим пример:

```
void callvirt(w*pw)
{ pw ! f();
}
main ()
{ callvirt(new BMW);
}
```

В функции main вызов callvirt с указателем на BMW требует приведения к указателю на W, поскольку функция callvirt ожидает параметр типа W*. Так как функция callvirt вызывает f() (через указатель на BMW, преобразованный к указателю на W), будет использована таблица vtbl класса W (в BMW), где указано, что экземпляром виртуальной функции f(), которую нужно вызвать, является BW :: f(). Чтобы передать функции BW :: f() указатель this на BW, указатель pw должен быть вновь приведен к указателю на BMW (вычитанием смещения для W), а затем к указателю на BW (добавлением смещения BW в объекте BMW). Значение смещения BW в объекте BMW минус смещение W в объекте BMW и есть смещение, хранимое в строке таблицы vtbl для w в BMW для функции BW :: f().

Генерация оптимального кода методами синтаксического анализа

Сопоставление образцов

Техника генерации кода, рассмотренная выше, основывалась на однозначном соответствии структуры промежуточного представления и описывающей это представление грамматики. Недостатком такого "жесткого" подхода является то, что как правило одну и ту же программу на промежуточном языке мож-

но реализовать многими различными способами в системе команд машины. Эти разные реализации могут иметь различную длину, время выполнения и другие характеристики. Для генерации более качественного кода может быть применен подход, изложенный в настоящей главе.

Этот подход основан на понятии "сопоставления образцов": командам машины сопоставляются некоторые "образцы", вхождения которых ищутся в промежуточном представлении программы, и делается попытка "покрыть" промежуточную программу такими образцами. Если это удается, то по образцам восстанавливается программа уже в кодах. Каждое такое покрытие соответствует некоторой программе, реализующей одно и то же промежуточное представление.

На рис. 9.24 показано промежуточное дерево для

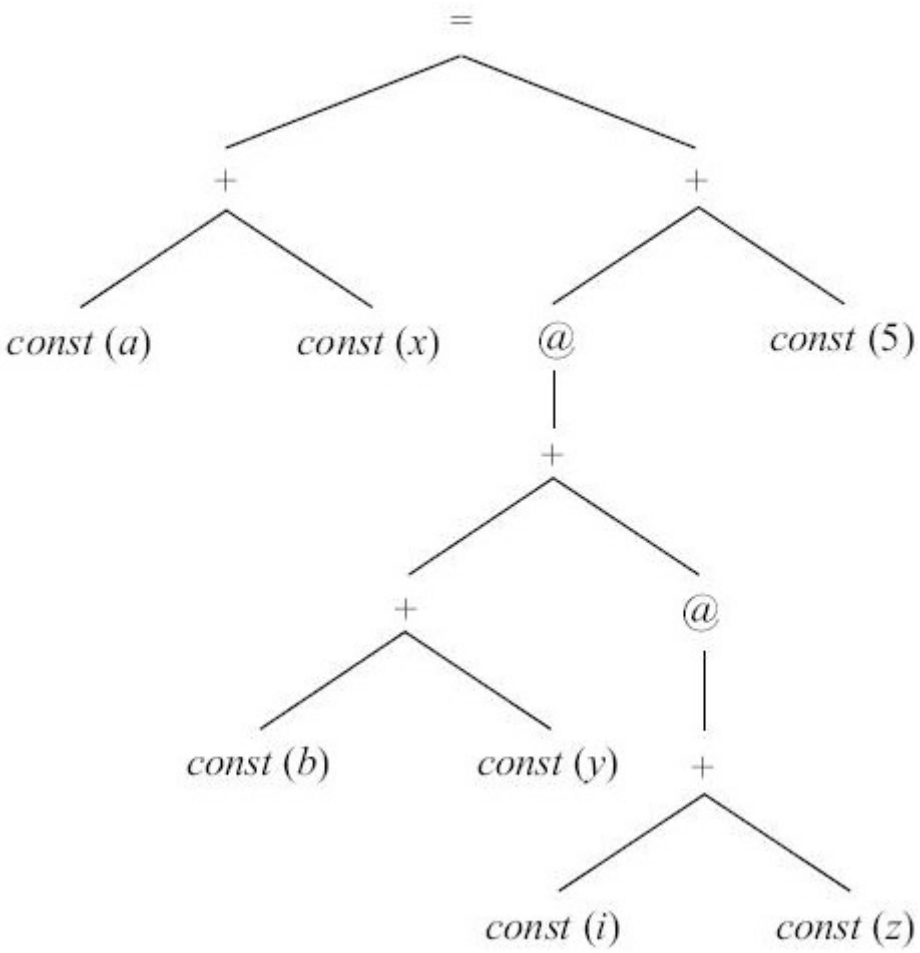


Рис. 9.24.

оператора a = b[i] + 5, где a, b, i - локальные переменные, хранимые со смещениями x, y, z соответственно в областях данных с одноименными адресами.

Элемент массива b занимает память в одну машинную единицу. 0-местная операция const возвращает значение атрибута соответствующей вершины промежуточного дерева, указанного на рисунке в скобках после оператора. Одноместная операция @ означает косвенную адресацию и возвращает содержимое регистра или ячейки памяти, имеющей адрес, задаваемый аргументом операции.

На рис. 9.25 показан пример сопоставления образцов машинным командам. Приведены два варианта задания образца: в виде дерева и в виде правила контекстно-свободной грамматики. Для каждого образца указана машинная команда, реализующая этот образец, и стоимость этой команды.

В каждом дереве-образце корень или лист может быть

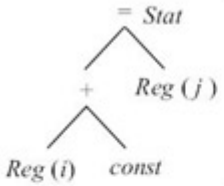
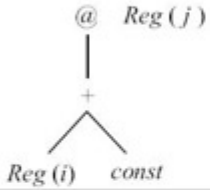
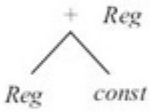
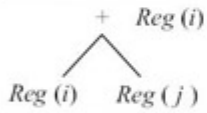
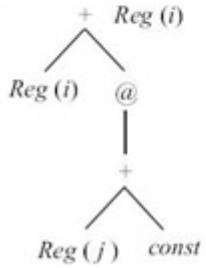
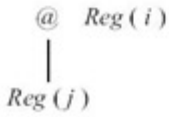
№	Образец	Правило грамматики	Команда / стоимость	
1		*Reg* → *const*	`MOVE #const, R`	2
2		*Stat* → '=' '+' *Reg const Reg*	`MOVE Rj, const(Ri)`	4
3		*Reg* → '@' '+' *Reg const*	`MOVE const(Ri), Rj`	4
4		*Reg* → '+' *Reg const*	`ADD #const,R`	3
5		*Reg* → '+' *Reg Reg*	`ADD Rj,Ri`	2
6		*Reg* → '+' *Reg* '@' '+' *Reg const*	`ADD const(Rj),Ri`	4
7		*Reg* → '@' *Reg*	`MOVE (Rj),Ri`	2

Рис. 9.25.

помечен терминальным и/или нетерминальным символом. Внутренние вершины помечены терминальными символами - знаками операций. При наложении образца на дерево выражения, во-первых, терминальный символ образца должен соответствовать терминальному символу дерева, и, во-вторых, образцы должны "склеиваться" по типу нетерминального символа, то есть тип корня образца должен совпадать с типом вершины, в которую образец подставляется корнем. Допускается использование "цепных" образцов, то есть образцов, корню которых не соответствует терминальный символ, и имеющих единственный элемент в правой части. Цепные правила служат для приведения вершин к одному типу. Например, в рассматриваемой системе команд одни и те же регистры используются как для целей адресации, так и для вычислений. Если бы в системе команд для этих целей использовались разные группы регистров, то в грамматике команд могли бы использоваться разные нетерминалы, а для пересылки из адресного регистра в регистр данных могла бы использоваться соответствующая команда и образец.

Нетерминалы Reg на образцах могут быть помечены индексом (i или j), что (неформально) соответствует номеру регистра и служит лишь для пояснения смысла использования регистров. Отметим, что при генерации кода рассматриваемым методом не осуществляется распределение регистров. Это является отдельной задачей. Стоимость может определяться различными способами, например числом обращений к памяти при выборке и исполнении команды. Здесь мы не рассматриваем этого вопроса. На рис. 9.26 приведен пример покрытия промежуточного дерева рис. 9.24 образцами рис. 9.25. В рамки заключены фрагменты дерева, сопоставленные образцу правила, номер которого указывается в левом верхнем углу рамки. В квадратных скобках указаны результирующие вершины.

Приведенное покрытие дает такую последовательность команд:

```
1 MOVE #b,Rb
4 ADD #y,Rb
1 MOVE #i,Ri
6 ADD #z(Ri),Rb
7 MOVE (Rb),Rb
4 ADD #5,Rb

1 MOVE #a,Ra
```

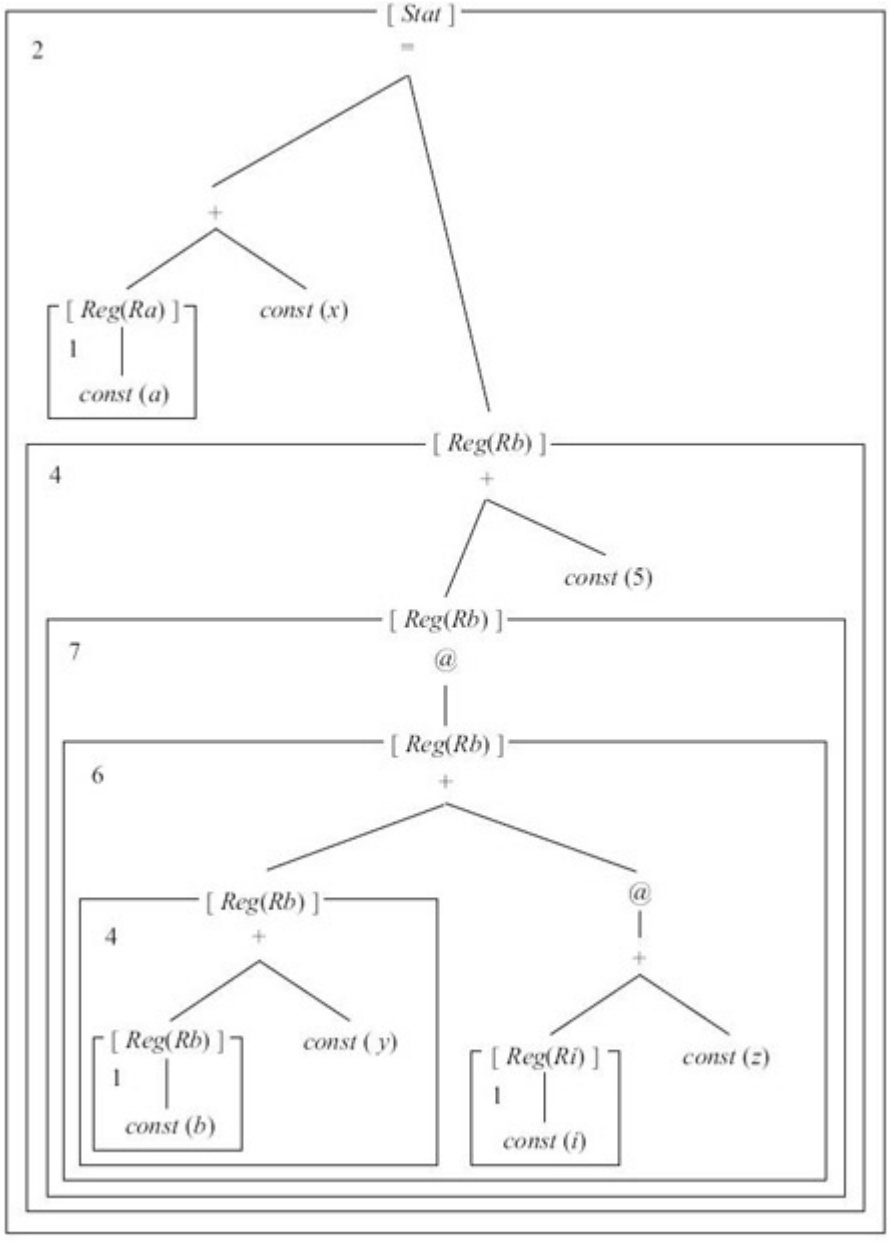


Рис. 9.26.

```
2 MOVE Rb,#x(Ra)
```

Основная идея подхода заключается в том, что каждая команда машины описывается в виде такого образца. Различные покрытия дерева промежуточного представления соответствуют различным последовательностям машинных команд. Задача выбора команд состоит в том, чтобы выбрать наилучший способ реализации того или иного действия или последовательности действий, то есть выбрать в некотором смысле оптимальное покрытие.

Для выбора оптимального покрытия было предложено несколько интересных алгоритмов, в частности использующих динамическое программирование [14, 16]. Мы здесь рассмотрим алгоритм [15], комбинирующий возможности синтаксического анализа и динамического программирования. В основу этого ал-

горитма положен синтаксический анализ неоднозначных грамматик (модифицированный алгоритм Кока, Янгера и Касами [18, 19]), эффективный в реальных приложениях. Этот же метод может быть применен и тогда, когда в качестве промежуточного представления используется дерево.

Синтаксический анализ для Т-грам- матик

Обычно код генерируется из некоторого промежуточного языка с довольно жесткой структурой. В частности, для каждой операции известна ее размерность, то есть число операндов, большее или равное 0. Операции задаются терминальными символами, и наоборот - будем считать все терминальные символы знаками операций. Назовем грамматики, удовлетворяющие этим ограничениям, Т-грам- матиками. Правая часть каждой продукции в Т-грамматике есть правильное префиксное выражение, которое может быть задано следующим определением:

1. Операция размерности 0 является правильным префиксным выражением;
2. Нетерминал является правильным префиксным выражением;
3. Префиксное выражение, начинающееся со знака операции размерности n > 0, является правильным, если после знака операции следует n правильных префиксных выражений;
4. Ничто другое не является правильным префиксным выражением

Образцы, соответствующие машинным командам, задаются правилами грамматики (вообще говоря, неоднозначной). Генератор кода анализирует входное префиксное выражение и строит одновременно все возможные деревья разбора. После окончания разбора выбирается дерево с наименьшей стоимостью. Затем по этому единственному оптимальному дереву генерируется код.

Для Т-грамматик все цепочки, выводимые из любого нетерминала A, являются префиксными выражениями с фиксированной арностью операций. Длины всех выражений из входной цепочки $a_1 \ldots a_n$ можно предварительно вычислить (под длиной выражения имеется ввиду длина подстроки, начинающейся с символа кода операции и заканчивающейся последним символом, входящим в выражение для этой операции). Поэтому можно проверить, сопоставимо ли некоторое правило с подцепочкой ai : : : ak входной цепочки $a_1 \ldots a_n$, проходя слева-направо по $a_i \ldots a_k$. В процессе прохода по цепочке предварительно вычисленные длины префиксных выражений используются для того, чтобы перейти от одного терминала к следующему терминалу, пропуская подцепочки, соответствующие нетерминалам правой части правила.

Цепные правила не зависят от операций, следовательно, их необходимо проверять отдельно. Применение одного цепного правила может зависеть от применения другого цепного правила. Следовательно, применение цепных правил необходимо проверять до тех пор, пока нельзя применить ни одно из цепных правил. Мы предполагаем, что в грамматике нет циклов в применении цепных правил. Построение всех вариантов анализа для Т-грамматики дано ниже в алгоритме 9.5. Тип Titem в алгоритме 9.5 ниже служит для описания ситуаций (то есть правил вывода и позиции внутри правила). Тип Tterminal - это тип терминального символа грамматики, тип Tproduction - тип для правила вывода.

Листинг 9.5. (html, txt)

Проверить, принадлежит ли (S->w) множеству r[0]; Множества r[i] имеют размер O(|P|). Можно показать, что алгоритм имеет временную и емкостную сложность O(n). Рассмотрим вновь пример рис. 9.24. В префиксной записи приведенный фрагмент программы записывается следующим образом:

= + a x + @ + + b y @ + i z 5

На рис. 9.27 приведен результат работы алгоритма. Правила вычисления стоимости приведены в следующем разделе. Все возможные выводы входной цепочки (включая оптимальный) можно построить, используя таблицу l длин префиксных выражений и таблицу r применимых правил. Операция Длина Правила (стоимость)

Операция	Длина	Правила (стоимость)	
=	15	2(22)	
+	3	4(5)	5(6)
a	1	1(2)	
x	1	1(2)	
+	11	4(16)	5(17)
@	9	7(11)	
+	8	5(13)	6(11)
+	3	4(5)	5(6)
b	1	1(2)	
y	1	1(2)	
@	4	7(7)	3(6)
+	3	4(5)	5(6)
i	1	1(2)	
z	1	1(2)	
5	1	1(2)	

Рис. 9.27.

Пусть G - это T-грамматика. Для каждой цепочки z из L(G) можно построить абстрактное синтаксическое дерево соответствующего выражения (рис. 9.24). Мы можем переписать алгоритм так, чтобы он принимал на входе абстрактное синтаксическое дерево выражения, а не цепочку. Этот вариант алгоритма приведен ниже. В этом алгоритме дерево выражения обходится сверху вниз и в нем ищутся поддеревья, сопоставимые с правыми частями правил из G. Обход дерева осуществляется процедурой PARSE. После обхода поддерева данной вершины в ней применяется процедура MATCHED, которая пытается найти все образцы, сопоставимые поддереву данной вершины. Для этого каждое правило-образец разбивается на компоненты в соответствии с встречающимися в нем операциями. Дерево обходится справа налево только для того, чтобы иметь соответствие с порядком вычисления в **алгоритме 9.5**. Очевидно, что можно обходить дерево вывода и слева направо.

Структура данных, представляющая вершину дерева, имеет следующую форму:

```
struct Tnode {
  Tterminal op;
  Tnode * son[MaxArity];
  setofTproduction RULEs;
};
```

В комментариях указаны соответствующие фрагменты алгоритма 9.5.

Листинг 9.6. (html, txt)

Выходом алгоритма является дерево выражения для z, вершинам которого сопоставлены применимые правила. С помощью такого дерева можно построить все выводы для исходного префиксного выражения.

Выбор дерева вывода наименьшей стоимости

Т-грамматики, описывающие системы команд, обычно являются неоднозначными. Чтобы сгенерировать код для некоторой входной цепочки, необходимо выбрать одно из возможных деревьев вывода. Это дерево должно представлять желаемое качество кода, например размер кода и/или время выполнения.

Для выбора дерева из множества всех построенных деревьев вывода можно использовать атрибуты стоимости, атрибутные формулы, вычисляющие их значения, и критерии стоимости, которые оставляют для каждого нетерминала единственное применимое правило. Атрибуты стоимости сопоставляются всем нетерминалам, атрибутные формулы - всем правилам Т-грамматики.

Предположим, что для вершины n обнаружено применимое правило

$p : A \rightarrow z_0X_1z_1 \ldots X_kz_k;$

где $z_i \in T^*$ для $0 \leq i \leq k$ и $X_j \in N$ для $0 \leq j \leq k$. Вершина n имеет потомков n1; : : : ; nk, которые соответствуют нетерминалам $X_1,\ldots, X_k$. Значения атрибутов стоимости вычисляются обходя дерево снизу вверх. Вначале атрибуты стоимости инициализируются неопределенным значением UndefinedValue. Предположим, что значения атрибутов стоимости для всех потомков $n_1,\ldots, n_k$ вершины n вычислены. Если правилу p сопоставлена формула

$a(A) = f(b(X_i), c(X_j),\ldots)$ для $1 \leq i, j \leq k;$

то производится вычисление значения атрибута a нетерминала A в вершине n. Для всех примененных правил ищется такое, которое дает минимальное значение стоимости. Отсутствие примененных правил обозначается через Undefined, значение которого полагается большим любого определенного значения.

Добавим в алгоритм 9.6 реализацию атрибутов стоимости, формул их вычисления и критериев отбора. Из алгоритма можно исключить поиск подвыводов, соответствующих правилам, для которых значение атрибута стоимости не определено. Структура данных, представляющая вершину дерева, принимает следующий вид:

Листинг 9.7. (html, txt)

Процедура ВычислитьАтрибутыСтоимостиДля (A, n, (A →bu)) вычисляет стоимость применения правила в данной вершине для данного нетерминала.

Процедура ПроверитьКритерийДля(C, n →nonterm[C]: CostAttr) определяет наилучшее правило.

Процедура Модифицировать(n →nonterm[C]:CostAttr) позволяет хранить это наилучшее значение в варианте. Дерево наименьшей стоимости определяется как дерево, соответствующее минимальной стоимости корня. Когда выбрано дерево вывода наименьшей стоимости, вычисляются значения атрибутов, сопоставленных вершинам дерева вывода, и генерируются соответствующие машинные команды. Вычисление значений атрибутов, генерация кода осуществляются в процессе обхода выбранного дерева вывода сверху вниз, слева направо. Обход выбранного дерева вывода выполняется процедурой вычислителя атрибутов, на вход которой поступают корень дерева выражения и аксиома грамматики. Процедура использует правило $A \rightarrow z_0X_1z_1 \ldots X_kz_k$, связанное с указанной вершиной n, и заданный нетерминал A, чтобы определить соответствующие им вершины $n_1,\ldots, n_k$ и нетерминалы $X_1,\ldots, X_k$. Затем вычислитель рекурсивно обходит каждую вершину n_i, имея на входе нетерминал X_i.

Атрибутная схема для алгоритма сопоставления образцов

Алгоритмы 9.5 и 9.6 являются "универсальными" в том смысле, что конкретные грамматики выражений и образцов являются, по-существу, параметрами этих алгоритмов. В то же время, для каждой конкретной грамматики можно написать свой алгоритм поиска образцов. Например, в случае нашей грамматики выражений и приведенных на рис. 9.25 образцов алгоритм 9.6 может быть представлен атрибутной грамматикой, приведенной ниже.

Наследуемый атрибут Match содержит упорядоченный список (вектор) образцов для сопоставления в поддереве данной вершины. Каждый из образцов имеет вид либо <op op-list> (op - операция в данной вершине, а op- list - список ее операндов), либо представляет собой нетерминал N. В первом случае op-list "распределяется" по потомкам вершины для дальнейшего сопоставления. Во втором случае сопоставление считается успешным, если есть правило N ⟶op fPatig, где w состоит из образцов, успешно сопоставленных потомкам данной вершины. В этом случае по потомкам в качестве образцов распределяются элементы правой части правила. Эти два множества образцов могут пересекаться. Синтезируемый атрибут Pattern - вектор логических значений, дает результат сопоставления по вектору-образцу Match.

Таким образом, при сопоставлении образцов могут встретиться два случая:

1. Вектор образцов содержит образец <op fPatig>, где op - операция, примененная в данной вершине. Тогда распределяем образцы Pati по потомкам и сопоставление по данному образцу считаем успешным (истинным), если успешны сопоставления элементов этого образца по всем потомкам.
2. Образцом является нетерминал N. Тогда рассматриваем все правила вида N ⟶op fPatig. Вновь распределяем образцы Pati по потомкам и сопоставление считаем успешным (истинным), если успешны сопоставления по всем потомкам. В общем случае успешным может быть сопоставление по нескольким образцам.

Отметим, что в общем случае в потомки одновременно передается неско-лько образцов для сопоставления. В приведенной ниже атрибутной схеме не рассматриваются правила выбора покрытия наименьшей стоимости (см. предыдущий раздел). Выбор оптимального покрытия может быть сделан еще одним проходом по дереву, аналогично тому, как это было сделано выше. Например, в правиле с '+' имеется несколько образцов для Reg, но реального выбора одного из них не осуществляется. Кроме того, не уточнены некоторые детали реализации. В частности, конкретный способ формирования векторов Match и Pattern. В тексте употребляется термин "добавить", что означает добавление к вектору образцов очередного элемента. Векторы образцов записаны в угловых скобках.

```
RULE
Stat ::= '=' Reg Reg
SEMANTICS
Match<2>=<'+' Reg Const>;
Match<3>=<Reg>;
Pattern<0>[1]=Pattern<2>[1]&Pattern<3>[1].
```

Этому правилу соответствует один образец 2. Поэтому в качестве образцов потомков через их атрибуты Match передаются, соответственно, <'+' Reg Const> и <Reg>.

```
RULE
Reg ::= '+' Reg Reg
SEMANTICS
if (Match<0> содержит Reg в позиции i)
 {Match<2>=<Reg,Reg,Reg>;
  Match<3>=<Const,Reg,<'@' '+' Reg       Const>>;
  }
if (Match<0> содержит образец <'+' Reg Const>
в позиции j)
  {добавить Reg к Match<2> в некоторой позиции k;
  добавить Const к Match<3> в некоторой позиции k;
  }
if (Match<0> содержит образец <'+' Reg Const>
      в позиции j)
Pattern<0>[j]=Pattern<2>[k]&Pattern<3>[k];
```

```
if (Match[0] содержит Reg в i-й позиции)
Pattern<0>[i]=(Pattern<2>[1]&Pattern<3>[1])
      |(Pattern<2>[2]&Pattern<3>[2])
      |(Pattern<2>[3]&Pattern<3>[3]).
```

Образцы, соответствующие этому правилу, следующие:

```
(4) Reg ⟶ '+' Reg Const,
(5) Reg ⟶ '+' Reg Reg,
(6) Reg ⟶ '+' Reg '@' '+' Reg Const.
```

Атрибутам Match второго и третьего символов в качестве образцов при сопоставлении могут быть переданы векторы <Reg, Reg, Reg> и <Const, Reg, <'@' '+' Reg Const>>, соответственно. Из анализа других правил можно заключить, что при сопоставлении образцов предков левой части данного правила атрибуту Match символа левой части может быть передан образец <'+' Reg Const> (из образцов 2, 3, 6) или образец Reg.

```
RULE
Reg ::= '@' Reg
SEMANTICS
 if (Match<0> содержит Reg в i-й позиции)
  Match<2>=<<'+' Reg Const>,Reg>;
   if (Match<0> содержит <'@' '+' Reg   Const>
    в j-й позиции)
   добавить к Match<2> <'+' Reg Const> в k позиции;
  if (Match<0> содержит Reg в i-й позиции)

  Pattern<0>[i]=Pattern<2>[1]|Pattern<2>[2];
if (Match<0> содержит <'@' '+' Reg Const>
в j-й позиции)
  Pattern<0>[j]=Pattern<2>[k].
```

Образцы, соответствующие этому правилу, следующие:

```
(3) Reg ⟶ '@' '+' Reg Const,
(7) Reg ⟶ '@' Reg.
```

Соответственно, атрибуту Match второго символа в качестве образцов при сопоставлении могут быть переданы <'+' Reg Const> (образец 3) или <Reg> (образец 7). Из анализа других правил можно заключить, что при сопоставлении образцов предков левой части данного правила атрибуту Match могут быть переданы образцы <'@' '+' Reg Const> (из образца 6) и Reg.

```
RULE
Reg ::= Const
SEMANTICS
if (Pattern<0> содержит Const в j-й позиции)
Pattern<0>[j]=true;
if (Pattern<0> содержит Reg в i-й позиции)
Pattern<0>[i]=true.
```

Для дерева рис. 9.24 получим значения атрибутов, приведенные на рис. 9.28. Здесь M обозначает Match, P - Pattern, C - Const, R - Reg.

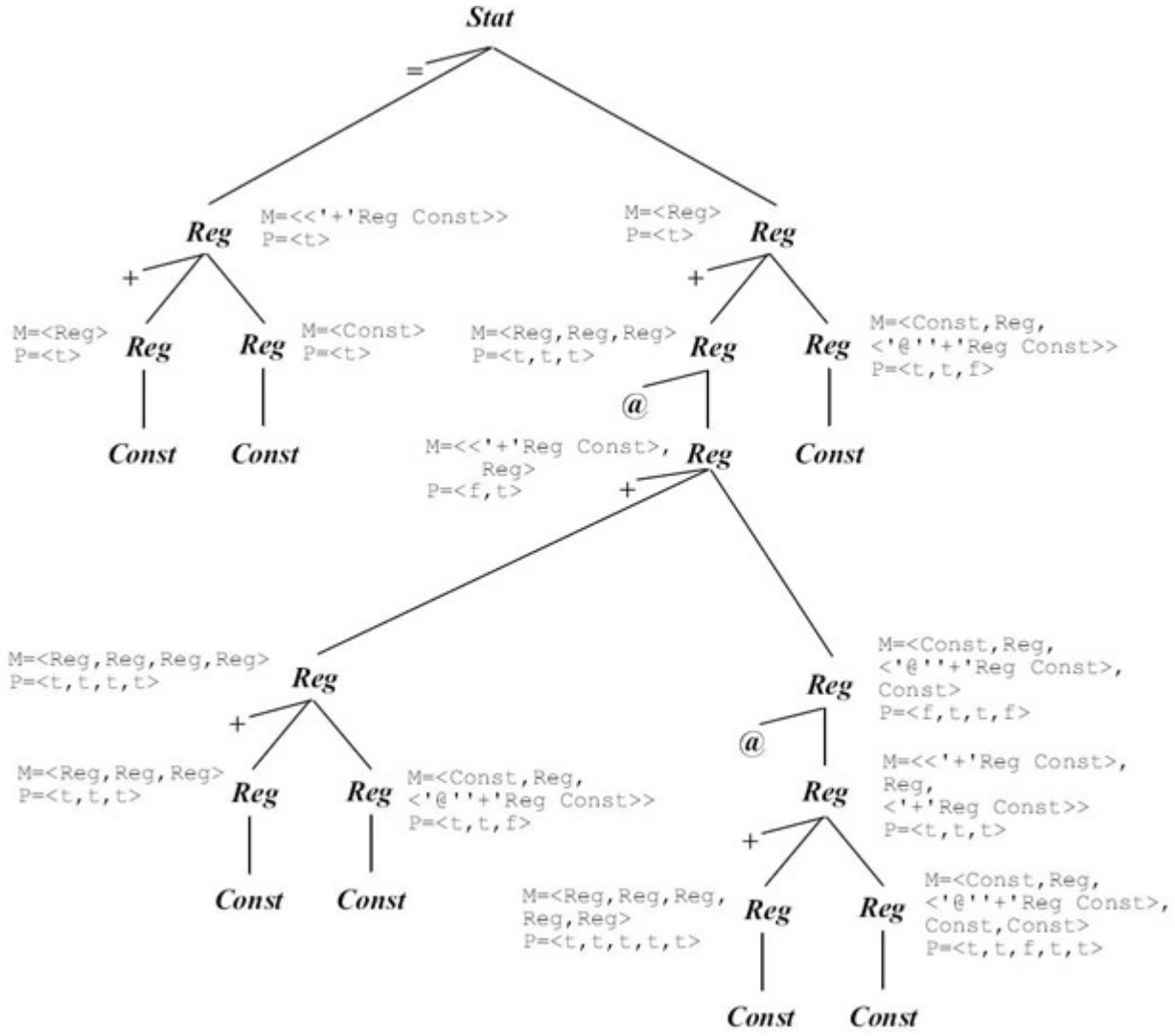


Рис. 9.28.

10. Лекция: Системы автоматизации построения трансляторов

В данной лекции рассматриваются системы автоматизации построения трансляторов на примере систем автоматизации построения трансляторов СУПЕР и YACC. Приведены структуры этих систем, основные термины и определения и части программного кода реализации систем автоматизации построения трансляторов.

Системы автоматизации построения трансляторов (САПТ) предназначены для автоматизации процесса разработки трансляторов. Очевидно, что для того, чтобы описать транслятор, необходимо иметь формализм для описания. Этот формализм затем реализуется в виде входного языка САПТ. Как правило, формализмы основаны на атрибутных грамматиках. Ниже описаны две САПТ, получившие распространение: СУПЕР [4] и Yacc. В основу первой системы положены LL(1)-грамматики и L-атрибутные вычислители, в основу второй - LALR(1)-грамматики и S-атрибутные вычислители.

Система СУПЕР

Программа на входном языке СУПЕР ("метапрограмма") состоит из следующих разделов:

- Заголовок;
- Раздел констант;
- Раздел типов;
- Алфавит;
- Раздел файлов;

- Раздел библиотеки;
- Атрибутная схема.

Заголовок определяет имя атрибутной грамматики, первые три буквы имени задают расширение имени входного файла для реализуемого транслятора. Раздел констант содержит описание констант, раздел типов - описание типов.

Алфавит содержит перечисление нетерминальных символов и классов лексем, а также атрибутов (и их типов), сопоставленных этим символам. Классы лексем являются терминальными символами с точки зрения синтаксического анализа, но могут иметь атрибуты, вычисляемые в процессе лексического анализа. Определение класса лексем состоит в задании имени класса, имен атрибутов для этого класса и типов этих атрибутов.

В разделе определения нетерминальных символов содержится перечисление этих символов с указанием приписанных им атрибутов и их типов. Аксиома грамматики указывается первым символом в списке нетерминалов. Раздел библиотеки содержит заголовки процедур и функций, используемых в формулах атрибутной грамматики. Раздел файлов содержит описание файловых переменных, используемых в формулах атрибутной грамматики. Файловые переменные можно рассматривать как атрибуты аксиомы.

Атрибутная схема состоит из списка синтаксических правил и сопоставленных им семантических правил. Для описания синтаксиса языка используется расширенная форма Бэкуса-Наура. Терминальные символы в правой части заключаются в кавычки, классы лексем и нетерминалы задаются их именами. Для задания в правой части необязательных символов используются скобки [ ], для задания повторяющихся конструкций используются скобки ( ). В этом случае может быть указан разделитель символов (после /). Например,

```
A ::= B [ C ] ( D ) ( E / ',' )
```

Первым правилом в атрибутной схеме должно быть правило для аксиомы.

Каждому синтаксическому правилу могут быть сопоставлены семантические действия. Каждое такое действие - это оператор, который может использовать атрибуты как символов данного правила (локальные атрибуты), так и символов, могущих быть предками (динамически) символа левой части данного правила в дереве разбора (глобальные атрибуты). Для ссылки на локальные атрибуты символы данного правила (как терминальные, так и нетерминальные) нумеруются от 0 (для символа левой части). При ссылке на глобальные атрибуты надо иметь в виду, что атрибуты имеют области видимости на дереве разбора. Областью видимости атрибута вершины, помеченной N, является все поддерево N, за исключением его поддеревьев, также помеченных N.

Исполнение операторов семантической части правила привязывается к обходу дерева разбора сверху вниз слева направо. Для этого каждый оператор может быть помечен меткой, состоящей из номера ветви правила, к выполнению которой должен быть привязан оператор, и, возможно, одного из суффиксов A, B, E, M.

Суффикс A задает выполнение оператора перед каждым вхождением синтаксической конструкции, заключенной в скобки повторений ( ). Суффикс B задает выполнение оператора после каждого вхождения синтаксической конструкции, заключенной в скобки повторений ( ). Суффикс M задает выполнение оператора между вхождениями синтаксической конструкции, заключенной в скобки повторений ( ). Суффикс E задает выполнение оператора в том случае, когда конструкция, заключенная в скобки [ ], отсутствует.

Пример использование меток атрибутных формул:

```
D ::= 'd' =>
   $0.y:=$0.x+1.
A ::= B (C) [D] =>
   $2.x:=1;
2M: $2.x:=$2.x+1;
```

```
   $3.x:=$2.x;
3E: $3.y:=$3.x;
3: writeln($3.y).
```

Процедура writeln напечатает число вхождений символа C в C-список, если D опущено. В противном случае напечатанное число будет на единицу больше.

Система YACC

В основу системы YACC положен синтаксический анализатор типа LALR(1), генерируемый по входной (мета) программе. Эта программа состоит из трех частей:

```
%{
Си-текст
%}
%token Список имен лексем
%%
Список правил трансляции
%%
Служебные Си-подпрограммы
```

Си-текст (который вместе с окружающими скобками %{ и %} может отсутствовать) обычно содержит Си-объявления (включая #include и #define), используемые в тексте ниже. Этот Си-текст может содержать и объявления (или предобъявления) функций.

Список имен лексем содержит имена, которые преобразуются YACC-препроцессором в объявления констант (#define). Как правило, эти имена используются как имена классов лексем и служат для определения интерфейса с лексическим анализатором.

Каждое правило трансляции имеет вид

```
Левая"часть : альтернатива"1
   {семантические"действия"1}
 | альтернатива"2 {семантические"действия"2}
 |...
 | альтернатива"n {семантические"действия"n}
 ;
```

Каждое семантическое действие - это последовательность операторов Си. При этом каждому нетерминалу может быть сопоставлен один синтезируемый атрибут. На атрибут нетерминала левой части ссылка осуществляется посредством значка $$, на атрибуты символов правой части - посредством значков $1, $2 , . . . , $n, причем номер соответствует порядку элементов правой части, включая семантические действия. Каждое семантическое действие может вырабатывать значение в результате выполнения присваивания $$=Выражение. Исполнение текого оператора в последнем семантическом действии определяет значение атрибута символа левой части.

В некоторых случаях допускается использование грамматик, имеющих конфликты. При этом синтаксический анализатор разрешает конфликты следующим образом:

- конфликты типа свертка/свертка разрешаются выбором правила, предшествующего во входной метапрограмме;
- конфликты типа сдвиг/свертка разрешаются предпочтением сдвига. Поскольку этих правил не всегда достаточно для правильного определения анализатора, допускается определение старшинства и ассоциативности терминалов.

Например, объявление

```
%left '+' '-'
```

определяет + и -, имеющими одинаковый приоритет и имеющими левую ассоциативность. Операцию можно определить как правоассоциативную в результате объявления:

```
%right '^'
```

Бинарную операцию можно определить как неассоциативную (то есть не допускающую появления объединения двух подряд идущих знаков операции):

```
%nonassoc '<'
```

Символы, перечисленные в одном объявлении, имеют одинаковое старшинство. Старшинство выше для каждого последующего объявления. Конфликты разрешаются путем присваивания старшинства и ассоциативности каждому правилу грамматики и каждому терминалу, участвующим в конфликте. Если необходимо выбрать между сдвигом входного символа s и сверткой по правилу A →w, свертка делается, если старшинство правила больше старшинства s или если старшинство одинаково, а правило левоассоциативно. В противном случае делается сдвиг.

Обычно за старшинство правила принимается старшинство самого правого терминала правила. В тех случаях, когда самый правый терминал не дает нужного приоритета, этот приоритет можно назначить следующим объявлением:

```
%prec терминал
```

Старшинство и ассоциативность правила в этом случае будут такими же, как у указанного терминала.

YACC не сообщает о конфликтах, разрешаемых с помощью ассоциативности и приоритетов. Восстановление после ошибок управляется пользователем с помощью введения в грамматику "правил ошибки" вида

```
A → error w:
```

Здесь error - ключевое слово YACC. Когда встречается синтаксическая ошибка, анализатор трактует состояние, набор ситуаций для которого содержит правило для error, некоторым специальным образом: символы из стека выталкиваются до тех пор, пока на верхушке стека не будет обнаружено состояние, для которого набор ситуаций содержит ситуацию вида [A →:error w]. После чего в стек фиктивно помещается символ error, как если бы он встретился во входной строке.

Если w пусто, делается свертка. После этого анализатор пропускает входные символы, пока не найдет такой, с которым можно продолжить нормальный разбор.

Если w не пусто, просматривается входная строка и делается попытка свернуть w. Если w состоит только из терминалов, эта строка ищется во входном потоке.

Литература

Список литературы

1. Адельсон-Вельский Г.М., Ландис Е.М
   Один алгоритм организации информации ДАН СССР. 1962. Т. 146. N 2. С. 263-266
2. Ахо А., Ульман Д
   Теория синтаксического анализа, перевода и компиляции, в 2-х т М.: Мир, 1978
3. Ахо А., Сети Р., Ульман Дж. Компиляторы
   Принципы, технологии, инструменты М. - СПб. - Киев: Вильямс, 2001
4. Бездушный А.Н., Лютый В.Г., Серебряков В.А
   Разработка компиляторов в системе СУПЕР М.: ВЦ АН СССР, 1991
5. Грис Д
   Конструирование компиляторов для цифровых вычислительных машин М.: Мир, 1975
6. Кнут Д
   Искусство программирования для ЭВМ. Т. 1. Основные алгоритмы М.: Мир, 1976

7. Кнут Д
   Семантика контекстно-свободных языков Семантика языков программирования. М.: Мир, 1980
8. Курочкин В.М., Столяров Л.Н., Сушков Б.Г., Флеров Ю.А
   Теория и реализация языков программирования Курс лекций. МФТИ, 1973 (1-е изд.) и 1978 г. (2-е изд.)
9. Курочкин В.М
   Алгоритм распределения регистров для выражений за один обход дерева вывода 2 Всес. конф. Литература 349 Автоматизация производства ППП и трансляторов. 1983. С. 104-105
10. Лавров С.С., Гончарова Л.И
    Автоматическая обработка данных. Хранение информации в памяти ЭВМ М.: Наука, 1971
11. Мартыненко Б.К
    Языки и трансляции
    СПб.: СПб. ГУ, 2004
12. Надежин Д.Ю., Серебряков В.А., Ходукин В.М
    Промежуточный язык Лидер (предварительное сообщение) Обработка символьной информации. М.: ВЦ АН СССР, 1987. С. 50-63
13. Хопкрофт Д., Мотвани Р., Ульман Д
    Введение в теорию автоматов, языков и вычислений, изд. 2-е М.: Вильямс, 2002. С. 527
14. Aho A.U., Ganapathi M., Tjiang S.W
    Code generation using tree matching and dynamic programing ACM Trans. Progr. Languages and Systems. 1989. V.11. N 4
15. Bezdushny A., Serebriakov V
    The use of the parsing method for optimal code generation and common subexpression elimination Techn. et Sci. Inform. 1993. V. 12. N. 1. P. 69-92
16. Emmelman H., Schroer F.W., Landweher R
    BEG -a generator for efficient back-ends ACM SIGPLAN. 1989. V. 11. N 4. P. 227-237
17. Fraser C.W., Hanson D.R
    A Retargetable compiler for ANS C SIGPLAN Notices. 1991. V. 26
18. Graham S.L., Harrison M.A., Ruzzo W.L
    An improved context-free recognizer ACM Trans. Program. Languages and Systems. 1980. N. 2
19. Harrison M.A
    Introduction to formal language theory Reading, Mass.: Addison-Wesley, 1978

Дополнительные материалы: Семантика контекстно-свободных языков

Введение

Допустим, что нам нужно дать точное определение двоичной системы записи чисел. Это можно сделать многими способами. В данном разделе мы рассмотрим метод, который может быть использован и для других систем счисления. В случае двоичной системы этот метод сводится к определению, основанному на следующей констекстно-свободной (КС) грамматике.

```
B ⟶ 0 B ⟶ 1
L ⟶ B L ⟶ LB
N ⟶ L N ⟶ L.L
```

Пример 1.1. (html, txt)

Здесь терминальными символами являются ".", "0" и "1", нетерминальными - B, L и N, обозначающие соответственно бит, список битов и число. Двоичным числом будем считать любую цепочку терминальных символов, выводимую из N при помощи правил (пример 1.1). Эта грамматика в действительности выражает тот факт, что двоичное число представляет собой последовательность из одного или более нулей и единиц, за которой может следовать точка и еще одна последовательность нулей и единиц. Кроме того, грамматика приписывает каждому двоичному числу определенную древовидную структуру. Например, цепочка 1101.01 получает следующую структуру:

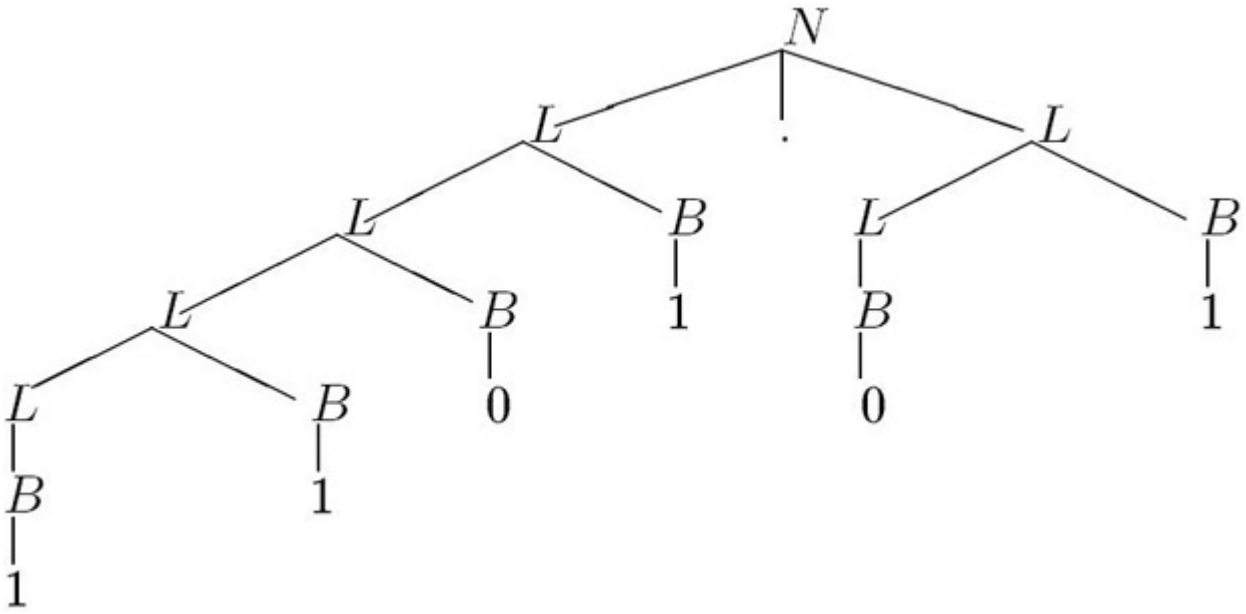


Рис. 1.2.

Естественно определять значение двоичной записи (пример 1.1) с помощью некоторого пошагового процесса, сопоставленного ее структуре (рис. 1.2); значение всей двоичной записи строится из значений отдельных частей. Это можно сделать, приписав каждому нетерминалу атрибуты следующим образом:

∀ бит B имеет целочисленный атрибут "значение", обозначаемый v(B).

∀ список битов L имеет целочисленный атрибут "длина", обозначаемый l(L). ∀ список битов L имеет целочисленный атрибут "значение", обозначаемый v(L).

∀ число N имеет атрибут "значение", являющийся рациональным числом и обозначаемый v(N).

(Заметим, что у всех нетерминалов L по два атрибута; вообще говоря, каждому нетерминалу можно приписывать любое желаемое число атрибутов).

Грамматику (пример 1.1) можно теперь расширить так, чтобы каждому синтаксическому правилу отвечали семантические правила.

```
B ⟶ 0          v(B) = 0
B ⟶ 1          v(B) = 1
L ⟶ B          v(L) = v(B); l(L) = 1
L1 ⟶ L2B       v(L1) = 2v(L2) + v(B); l(L1) = l(L2) + 1
N ⟶ L          v(N) = v(L)
N ⟶ L1:L2      v(N) = v(L1) + v(L2)=2l(L2)
```

Пример 1.3. (html, txt)

(Индексы в четвертом и шестом правилах применяются для того, чтобы различать вхождения одноименных нетерминалов.). В этих семантических правилах значения атрибутов всех нетерминалов определяются через значения атрибутов их непосредственных потомков, так что окончательные значения определены для всех атрибутов. Предполагается, что смысл обозначений, использованных для записи семантических правил, понятен. Отметим, например, что символ "0" в семантическом правиле v(B) = 0 понимается не так, как символ "0" в синтаксическом правиле B ⟶0. В первом случае "0" обозначает математическое понятие, а именно число нуль, во втором - некоторый символ, имеющий эллиптическую форму. В каком-то смысле, то, что эти два символа выглядят одинаково, - не более чем простое совпадение.

Структуру (рис. 1.2) можно расширить, выписав явно атрибуты при всех узлах.

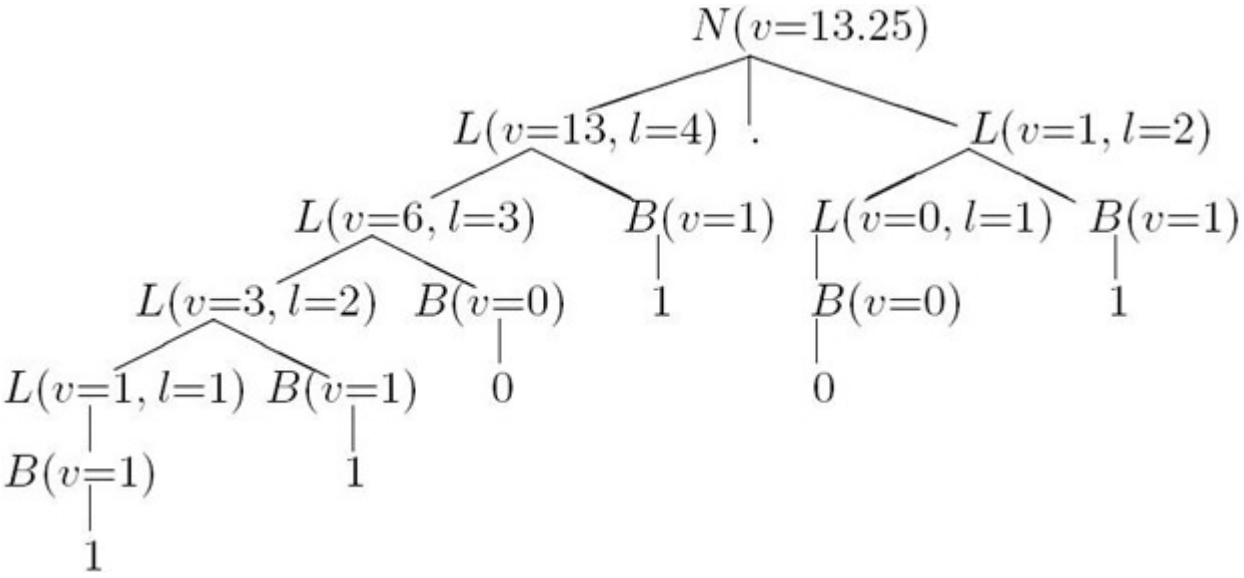


Рис. 1.4.

Таким образом, "1101.01" обозначает 13.25 (в десятичной записи).

Такой способ определения семантики КС-языков в сущности хорошо известен, так как он уже использовался несколькими авторами. Однако существует важное расширение этого метода. Именно это расширение и представляет для нас интерес.

Предположим, например, что мы хотим определить семантику двоичной записи другим способом, более близким к нашему обычному ее пониманию. Первая единица в записи 1101.01 на самом деле означает 8, хотя в соответствии с (рис. 1.4) ей приписывается значение 1. Возможно, поэтому будет лучше определять семантику таким образом, чтобы местоположение символа тоже играло определ_нную роль. Можно ввести следующие атрибуты:

∀ символ B имеет атрибут "значение", являющийся рациональным числом и обозначаемый v(B).

∀ символ B имеет целочисленный атрибут "масштаб ", обозначаемый s(B).

∀ символ L имеет атрибут "значение", являющийся рациональным числом и обозначаемый v(L).

∀ символ L имеет целочисленный атрибут "длина", обозначаемый l(L).

∀ символ L имеет целочисленный атрибут "масштаб", обозначаемый s(L).

∀ символ N имеет атрибут "значение", принимающий в качестве значений рациональные числа и обозначаемый v(N).

Эти атрибуты можно определить следующим образом:

Таблица 1.1.

Синтаксические правила	Семантические правила
B ⟶ 0	v(B) = 0
B ⟶ 1	$v(B) = 2^{s(B)}$

$L \longrightarrow B$	$v(L) = v(B),\ s(B) = s(L),\ l(L) = 1$
$L_1 \longrightarrow L_2 B$	$v(L_1) = v(L_2) + v(B),\ s(B) = s(L_1),\ s(L_2) = s(L_1) + 1,\ l(L_1) = l(L_2) + 1$
$N \longrightarrow L$	$v(N) = v(L);\ s(L) = 0$
$N \longrightarrow L_1.L_2$	$v(N) = v(L_1) + v(L_2),\ s(L_1) = 0,\ s(L_2) = -l(L_2)$

Здесь при записи семантических правил принято следующее соглашение. Правая часть каждого правила представляет собой определение левой части, таким образом, s(B) = s(L) означает, что сначала должно быть вычислено s(L), а затем полученное значение следует присвоить s(B).

Важным свойством грамматики (таблица 1.1) является то, что некоторые из атрибутов, которым присваиваются значения, приписаны нетерминалам, стоящим в правой части соответствующего синтаксического правила, в то время как в 1.3 атрибуты левых частей семантических правил относились только к нетерминалам, стоящим в левой части синтаксического правила. Здесь мы используем как синтезированные атрибуты (вычисляемые через атрибуты потомков данного нетерминала), так и унаследованные атрибуты (вычисляемые через атрибуты предков). Синтезированные атрибуты вычисляются в древовидной структуре снизу вверх, а унаследованные - сверху вниз. Грамматика (таблица 1.1) включает синтезированные атрибуты v(B), v(L), l(L), v(N) и унаследованные атрибуты s(B) и s(L), так что при их вычислении необходимо проходить по дереву в обоих направлениях. Вычисление на структуре, соответствующей цепочке 1101.01, имеет вид:

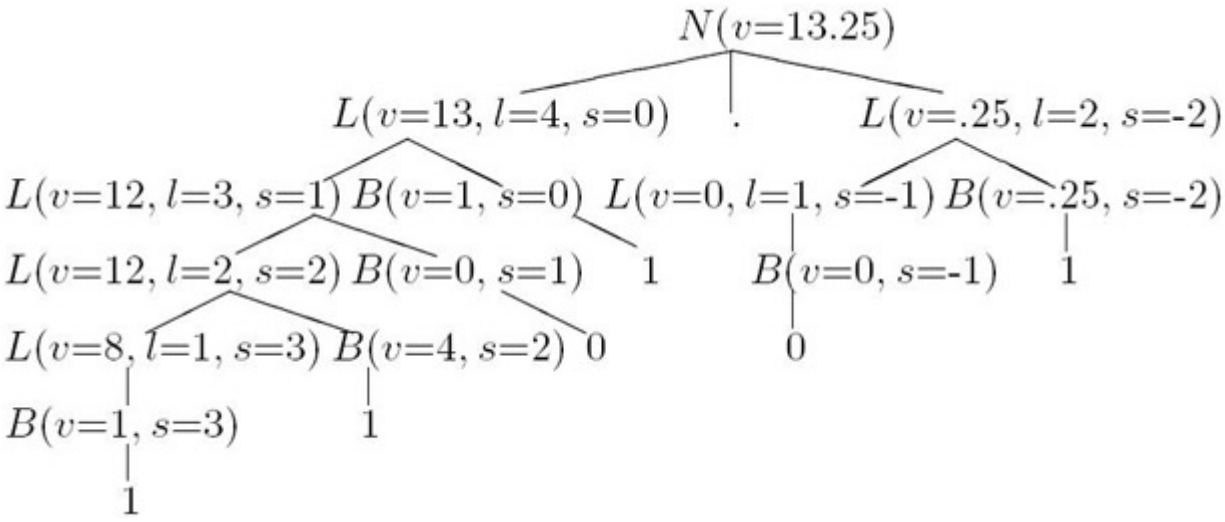


Рис. 1.6.

Можно заметить, что атрибуты "длина" символов L, стоящих справа от точки, должны быть вычислены снизу вверх до того, как будут вычислены (сверху вниз) атрибуты "масштаб" и атрибуты "значение" (снизу вверх).

Грамматика (таблица 1.1), вероятно, не является "наилучшей возможной" грамматикой для системы двоичной записи, но похоже, что она лучше согласуется с нашей интуицией, чем грамматика (пример 1.3). (Грамматика, которая более точно соответствует нашему традиционному толкованию двоичной нотации, содержит другое множество правил вывода. Эти правила сопоставляют цепочке битов справа от точки иную структуру, вследствие чего атрибут "длина", не играющий принципиальной роли, становится ненужным.)

Наш интерес к грамматике (таблица 1.1) вызван не тем, что она представляет собой идеальное определение двоичной системы записи, а тем, что она демонстрирует взаимодействие унаследованных и синтезированных атрибутов. Тот факт, что семантические правила, подобные правилам в (таблица 1.1), не при-

водят к зацикленности определения атрибутов, не является очевидным, поскольку здесь атрибуты вычисляются не при однократном обходе дерева в одном направлении. Алгоритм, проверяющий семантические правила на зацикленность, будет описан ниже.

Важность унаследованных атрибутов состоит в том, что они естественно возникают в практике и в очевидном смысле "двойственны" синтезированным атрибутам. Хотя для определения смысла двоичной записи достаточно только синтезированных атрибутов, существует ряд языков, для которых такое ограничение приводит к неуклюжему и неестественному определению семантики. Ситуации, когда встречаются и унаследованные, и синтезированные атрибуты, представляют собой те самые случаи, которые в предшествующих определениях семантики вызывали серьезные трудности.

Формальные свойства

Придадим теперь идее использования синтезированных и унаследованных атрибутов более точную и более общую форму.

Пусть имеется КС-грамматика G = (V, N, S, P), где V - (конечный) алфавит терминальных и нетерминальных символов; $N \subseteq V$ - множество нетерминальных символов; S $\in$N - "начальный "символ, не входящий в правые части правил, и P - множество правил.

Семантические правила дополняют G следующим образом. С каждым символом X $\in$V связывается конечное множество атрибутов A(X). A(X) разбивается на два непересекающихся множества: множество синтезированных атрибутов $A_0(X)$ и множество унаследованных атрибутов $A_1(X)$. Множество $A_1(S)$ должно быть пустым (то есть начальный символ S не должен иметь унаследованных атрибутов); аналогично, множество $A_0(X)$ пусто, если X - терминальный символ. Каждый атрибут R из множества A(X) имеет (возможно, бесконечное) множество значений VR. Для каждого вхождения X в дерево вывода семантические правила позволяют определить одно значение из множества VR для соответствующего атрибута.

Пусть P состоит из m правил, и пусть p-е правило имеет вид

$X_{p0} \longrightarrow X_{p1}X_{p2}\ldots X_{pnp}$;
Пример 2.1. (html, txt)

где np > 0, X_{p0} $\in$N и X_{pj} $\in$V для 1 $\leq$j $\leq n_p$. Семантическими правилами называются функции f_{pjR}, определ_нные для всех 1 $\leq$p $\leq$m, 0 $\leq$j $\leq n_p$ и некоторых $\alpha \in A_0(X_{pj})$, если j = 0, или $\alpha \in A_1(X_{pj})$, если j > 0. Каждая такая функция представляет собой отображение из $V\alpha_1$ x $V\alpha_2$ x ... x $V\alpha_t$ в VR для некоторого t = t(p, j, α) > 0, где все $\alpha_i = \alpha_i(p, j, \alpha)$ являются атрибутами некоторых X_{pki} , при 0 $\leq k_i = k_i(p, j, \alpha) \leq n_p$, 1 $\leq$i $\leq$t. Другими словами, каждое семантическое правило отображает значения некоторых атрибутов символов $X_{p0}, X_{p1}, \ldots X_{pn_p}$ и значение некоторого атрибута символа X_{pj}.

Грамматика (таблица 1.1), например, представляется в виде G = ({0, 1, ".", B, L, N}, {B, L, N}, N, {B $\longrightarrow$0, B $\longrightarrow$1, L $\longrightarrow$B, L $\longrightarrow$LB, N $\longrightarrow$L, N $\longrightarrow$L.L}).

Атрибутами здесь являются

$A_0(B) = \{v\}$, $A_1(B) = \{s\}$;
$A_0(L) = \{v, l\}$, $A_1(L) = \{s\}$;
$A_0(N) = \{v\}$, $A_1(N) = \varnothing$
и $A_0(x) = A_1(x) = \varnothing$

для x $\in${0, 1, .}. Множествами значений атрибутов будут V_v ={рациональные числа}, $V_s = V_l$ = {целые числа}. Типичным примером правил вывода служит четвертое правило $X_{40} \longrightarrow X_{41}X_{42}$, где $n_4 = 2$, $X_{40} = X_{41}$ = L, X_{42} = B. Так же типично и семантическое правило f_{40v}, соответствующее этому правилу вывода. Оно

определяет $v(X_{40})$ через другие атрибуты; в данном случае f_{40v} отображает Vv x Vv в Vv согласно формуле $f_{40v}(x, y) = x + y$. (Это есть не что иное, как правило $v(L_1) = v(L_2) + v(B)$ из (таблица 1.1); используя довольно громоздкую запись, введенную в предыдущем абзаце, получим:

$t(4, 0, v) = 2$, $\alpha_1(4, 0, v) = \alpha_2(4, 0, v) = v$, $k_1(4, 0, v) = 1$, $k_2(4, 0, v) = 2$).

Семантические правила используются для сопоставления цепочкам КС языка"значения" следующим образом[1] . Для любого вывода терминальной цепочки t из S при помощи синтаксических правил построим обычное дерево вывода. А именно, корнем дерева будет S, а каждый узел помечается либо терминальным символом, либо нетерминалом X_{p0}, соответствующим применению p-го правила для некоторого p; в последнем случае у этого узла будет np непосредственных потомков.

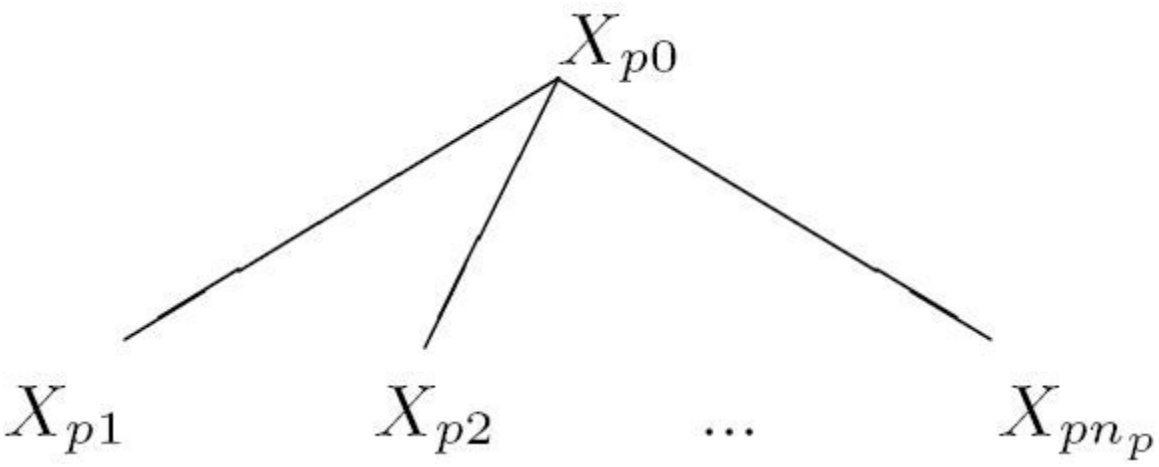


Рис. 2.2.

Пусть теперь X - метка некоторого узла дерева и пусть $R \in A(X)$ - атрибут символа X. Если $\alpha \in A_0(X)$, то $X = X_{p0}$ для некоторого p, если же $\alpha \in A_1(X)$, то $X = X_{pj}$ для некоторых j и p. В обоих случаях дерево "в районе" этого узла имеет вид (рис. 2.2). По определению атрибут α имеет в этом узле значение v, если в соответствующем семантическом правиле

$$f_{pj}\alpha : V\alpha_1 x \ \ldots \ x \ V\alpha_t \longrightarrow V\alpha$$

все атрибуты $\alpha_1, \ldots, \alpha_t$ уже определены и имеют в узлах с метками $X_{pk1}, \ldots, X_{pkt}$ значения $v_1, \ldots, v_t$ соответственно, а $v = f_{pj}\alpha(v_1, \ldots, v_t)$. Процесс вычисления атрибутов на дереве продолжается до тех пор, пока нельзя будет вычислить больше ни одного атрибута. Вычисленные в результате атрибуты корня дерева представляют собой "значение", соответствующее данному дереву вывода (рис. 1.6).

Естественно потребовать, чтобы семантические правила давали возможность вычислить все атрибуты произвольного узла в любом дереве вывода. Если это условие выполняется, будем говорить, что семантические правила заданы корректно[2] . Поскольку деревьев вывода, вообще говоря, бесконечно много, важно уметь определять по самой грамматике, являются ли корректными ее семантические правила.

Отметим, что этот метод определения семантики обладает такой же мощностью, как и всякий другой возможный метод, в том смысле, что значение любого атрибута в любом узле может произвольным образом зависеть от структуры всего дерева. Предположим, например, что в КС грамматике всем символам, кроме S, приписано по два унаследованных атрибута: l ("положение") и t ("дерево"), а всем нетерминалам, кроме того, по одному синтезированному атрибуту s ("поддерево"). Значениями l будут конечные последовательности положительных целых чисел $\{a_1 \cdot a_2 \cdot \ldots \cdot a_k\}$, определяющие местонахождение узла в дереве в соответствии с системой обозначения Дьюи. Атрибуты t и s представляют собой множество упорядоченных пар (l, X), где l - положение узла, а X - символ грамматики, обозначающий метку узла с положением l. Семантическими правилами для каждого синтаксического правила (пример 2.1) служат

$$l(X_{pj}) = \begin{cases} l(X_{p0}) \cdot j, & \text{если } X_{p0} \neq S; \\ j, & \text{если } X_{p0} = S; \end{cases}$$
$$t(X_{pj}) = \begin{cases} t(X_{p0}), & \text{если } X_{pj} \neq S; \\ s(X_{p0}), & \text{если } X_{pj} = S; \end{cases} \quad (2.4)$$

$$s(X_{p0}) = \{(l(X_{p0}), X_{p0}) \mid X_{p0} \neq S\} \cup \bigcup_{j=1}^{n_p} \{S(X_{pj}) \mid X_{pj} \in N\}$$

Следовательно, для дерева (рис. 1.2), например, мы имеем

```
s(N) = {(1, L), (2, ·), (3, L),
   (1.1, L), (1.2, B), (3.1, L), (3.2, B),
   (1.1.1, L), (1.1.2, B), (1.2.1, 1), (3.1.1, B), (3.2.1, 1),
   (1.1.1.1, L), (1.1.1.2, B), (1.1.2.1, 0), (3.1.1.1, 0),
   (1.1.1.1.1, B), (1.1.1.2.1, 1), (1.1.1.1.2.1, 1)}.
```

Ясно, что эта запись содержит всю информацию о дереве вывода. Согласно семантическим правилам (2.4), атрибут t во всех узлах (кроме корня) представляет собой множество, характеризующее все дерево вывода; атрибут l определяет местонахождение этих узлов. Отсюда сразу следует, что любая мыслимая функция, определ_нная на дереве вывода, может быть представлена как атрибут произвольного узла, поскольку эта функция имеет вид f(t, l), для некоторого f. Аналогично, можно показать, что для определения значения, связанного с произвольным деревом вывода, достаточно только синтезированных атрибутов, поскольку синтезированный атрибут w, вычисляемый по формуле

$$w(X_{p0}) = \{(0, X_{p0}) \cup \bigcup_{j=1}^{n_p} \{(j \cdot \alpha, X) \mid (\alpha, X) \in w(X_{pj}), X_{pj} \in N\} \quad (2.5)$$

в корне дерева полностью определяет все дерево[3)] . Каждое семантическое правило, определяемое методами этого раздела, можно рассматривать как функцию этого атрибута w. Следовательно, описанный общий метод по существу не более мощен, чем метод, вовсе не использующий наследованных атрибутов. Правда, это утверждение не следует понимать как практическую рекомендацию, поскольку семантические правила, не использующие унаследованных атрибутов, будут зачастую гораздо более сложными (а также менее понимаемыми и практичными), чем правила, включающие атрибуты обоих типов. Если допустить, чтобы атрибуты в каждом узле дерева могли зависеть от всего дерева, то семантические правила часто могут стать проще и будут лучше соответствовать нашему пониманию процесса вычисления.

Проверка на зацикленность

Рассмотрим теперь алгоритм, проверяющий, является ли корректной система семантических правил, определенная в предыдущем разделе. Другими словами, мы хотим знать, когда семантические правила позволяют вычислить значение любого атрибута любого узла произвольного дерева вывода. Можно считать, что грамматика не содержит "бесполезных" правил вывода, то есть что каждое правило из P участвует в выводе хотя бы одной терминальной цепочки.

Пусть T - произвольное дерево вывода, соответствующее данной грамматике; метками концевых узлов могут быть только терминальные символы, корню же разрешим иметь меткой не только аксиому, но и любой символ из V . Тогда можно определить ориентированный граф D(T), соответствующий T, взяв в качестве его узлов упорядоченные пары (X, α), где X - узел дерева T, а α- атрибут символа, служащего меткой узла X. Дуга из (X_1; α1) в (X_2, α_2) проводится в том и только в том случае, когда семантическое правило, вычисляющее атрибут α_2, непосредственно использует значение атрибута α_1. Например, если T -

дерево (рис. 1.2), а в качестве семантических правил взяты правила (таблица 1.1), то орграф D(T) будет иметь такой вид:

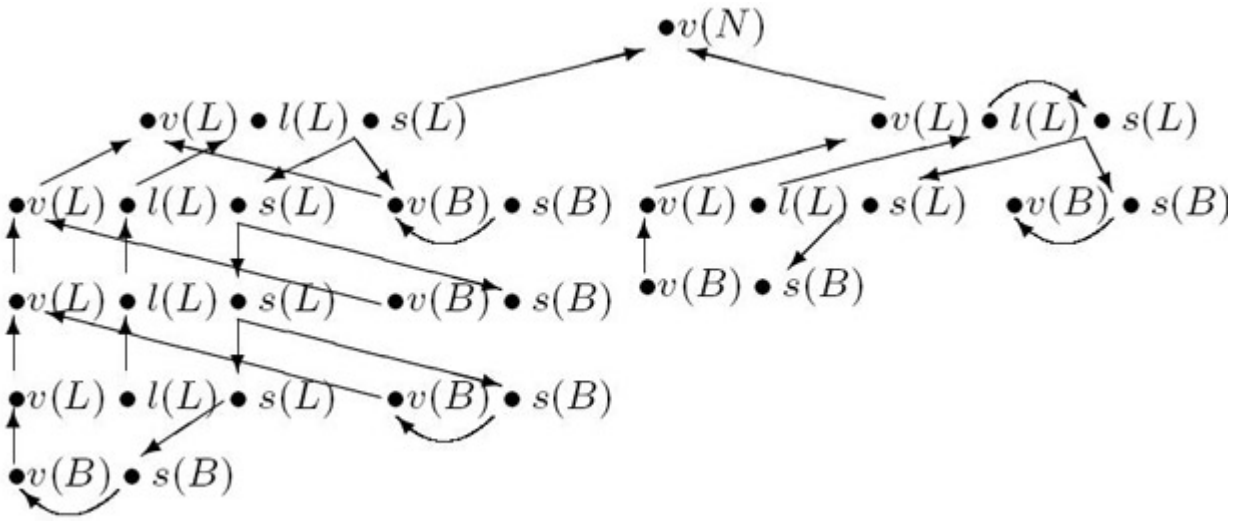
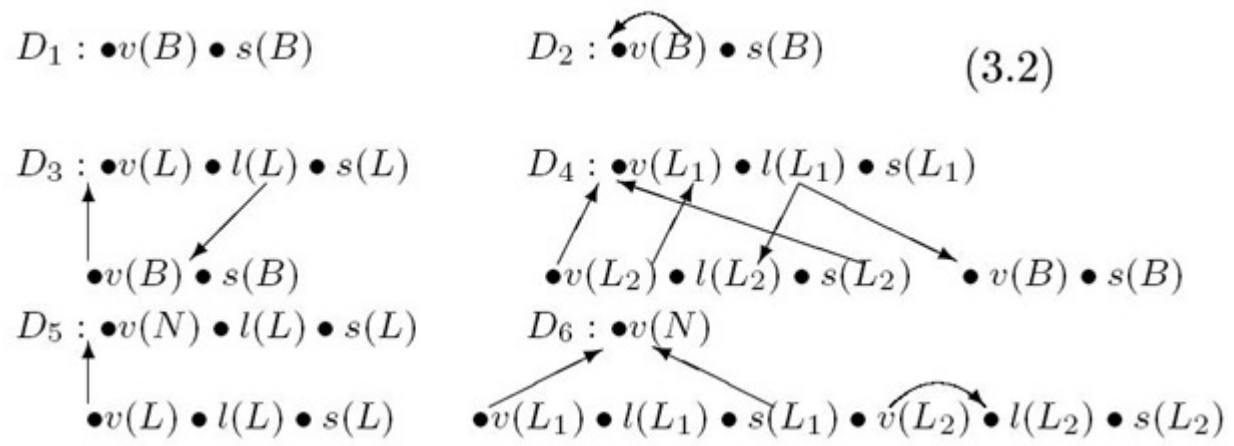


Рис. 3.1.

Другими словами, узлами графа D(T) служат атрибуты, значения которых нужно вычислить, а дуги определяют зависимости, подразумевающие, какие атрибуты вычисляются раньше, а какие позже (рис. 1.6).

Ясно, что семантические правила являются корректными тогда и только тогда, когда ни один из орграфов D(T) не содержит ориентированного цикла. Дело в том, что если в графе нет ориентированных циклов, то можно применить хорошо известную процедуру, позволяющую присвоить значения всем атрибутам. Если же в некотором графе D(T) есть ориентированный цикл, то ввиду того что грамматика не содержит бесполезных правил, можно утверждать, что существует ориентированный цикл в некотором графе D(T), у которого меткой корня дерева T служит S. Для такого дерева T все атрибуты вычислить не уда_тся. Таким образом, задача "Являются ли семантические правила корректным?" сводится к задаче "Содержат ли орграфы D(T) ориентированные циклы?"

Каждый орграф D(T) можно рассматривать как суперпозицию меньших орграфов Dp, соответствующих правилам $X_{p0} \to X_{p1} \dots X_{pn}$ грамматики, $1 \le p \le m$. В обозначениях разд. 2 орграф D_p имеет узлы (X_{pj}, α) для $0 \le j \le n_p$, $\alpha \in A(X_{pj})$ и дуги из (X_{pki}, α) в (X_{pj}, α) для $0 \le j \le n_p$, $\alpha \in A_0(X_{pj})$, если $j = 0$, $\alpha \in A_1(X_{pj})$, если $j > 0$, $k_i = k_i(p, j, \alpha)$, $\alpha i = \alpha i(p, j, \alpha)$, $1 \le i \le t(p, j, \alpha)$. Другими словами, Dp отражает связи, которые порождают все семантические правила, соответствующие p-му синтаксическому правилу. Например, шести правилам грамматики (таблица 1.1) соответствуют шесть следующих орграфов:



(3.2)

Рис. 3.2.

Орграф (рис. 3.1) получается в результате "объединения" таких подграфов. Вообще, если T имеет в качестве метки корня терминал, D(T) не содержит дуг. Если корень дерева T помечен нетерминальным символом, T имеет вид

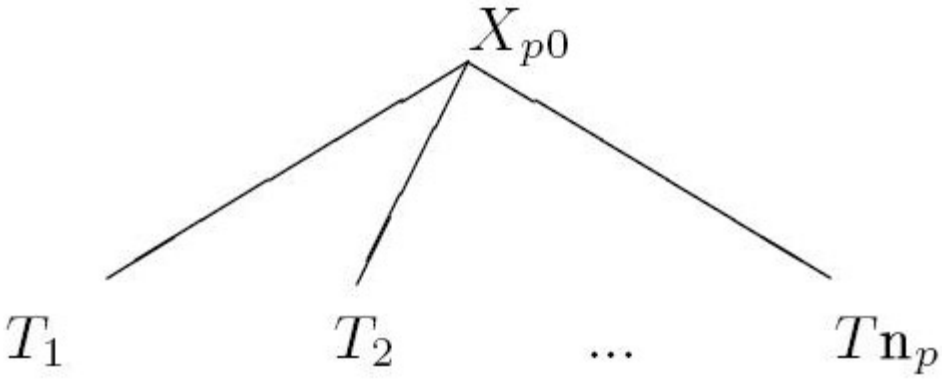


Рис. 3.3.

для некоторого p, где T_j - дерево вывода, у которого корень помечен символом X_{pj}, где $1 \leq j \leq n_p$. В первом случае говорят, что T - дерево вывода типа 0, во втором случае T называется деревом вывода типа p. В соответствии с этим определением для того, чтобы по D_p, $D(T_1)$, ... , $D(T_{np})$ построить D(T), нужно для всех j, $1 \leq j \leq n_p$ совместить узлы, соответствующие атрибутам символа X_{pj} графа D_p с соответствующими узлами (отвечающими тем же атрибутам корня дерева T_j) в графе $D(T_j)$.

Для проверки того, содержит ли граф D(T) ориентированный цикл, нам понадобится еще одно понятие. Пусть p - номер правила вывода. Обозначим через G_j произвольный орграф ($1 \leq j \leq np$), множество узлов которого является подмножеством множества $A(X_{pj})$ атрибутов символа X_{pj}. Пусть

$D_p[G_1, \ldots, G_{np}]$

Пример 3.4. (html, txt)

орграф, полученный из D_p добавлением дуг, идущих из (X_{pj}, α) в (X_{pj}, α_0), если в графе G_j есть дуга из α в α_0

Например, если

и если D_4 - ориентированный граф из (3.2), то $D_4(G_1, G_2)$ имеет вид

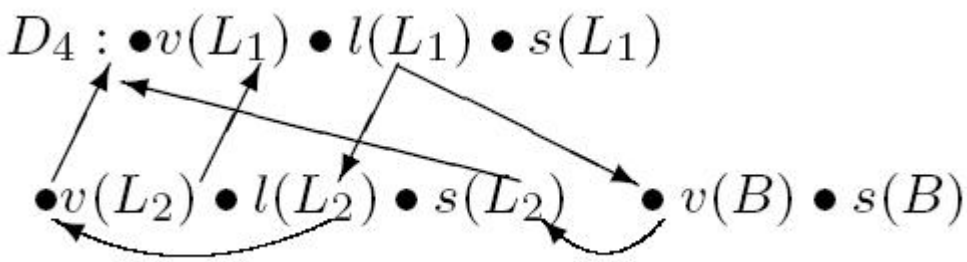


Рис. 3.4.

Теперь можно воспользоваться следующим алгоритмом. Для любого X $\in$V S(X) будет некоторым множеством ориентированных графов с узлами из A(X). Сначала для всех X $\in$N S(X) пусто, а для X = $\in$N S(X) состоит из единственного графа с множеством узлов A(X) и не содержащего дуг. Будем добавлять к множествам S(X) новые орграфы при помощи следующей процедуры до тех пор, пока в S(X) не перестанут появляться новые элементы. Выберем целое p; 1 $\leq$p $\leq$m и для каждого j, 1 $\leq$j $\leq$np, выберем орграф D_{0j} $\in$S(Xpj). Затем добавим в $S(X_{p0})$ орграф с множеством узлов $A(X_{po})$, обладающий тем свойством, что в нем дуга от αк α0 идет тогда и только тогда, когда в орграфе

$$D_p[D'_1, \ldots, D'_{n_p}] \quad (3.5)$$

существует ориентированный путь из (X_{p0}; α) в (X_{p0}; α'): Ясно, что этот процесс рано или поздно закончится и новые орграфы перестанут порождаться, поскольку вообще существует лишь конечное число ориентированных графов. В случае грамматики (таблица 1.1) алгоритм построит следующие множества:

$$S(N) = \left\{ \begin{array}{c} v \\ . \end{array} \right\}; \qquad S(L) = \left\{ \begin{array}{cc} v\ l\ s & v\ l\ s \\ \ldots, & \ldots \end{array} \right\};$$

$$S(B) = \left\{ \begin{array}{cc} v\ s & v\ \ s \\ \ldots, & \ldots \end{array} \right\}; S(0) = S(1) = \{\}.$$

Пусть T - дерево вывода с корнем X, и пусть D'(T) - ориентированный граф с множеством узлов A(X), у которого есть дуга из αв α' тогда и только тогда, когда в D(T) существует ориентированный путь из (X, α) в (X, α'). Покажем, что после окончания работы описанного выше алгоритма для всех X $\in$V S(X) - это множество всех D'(T), где T - дерево вывода с корнем X [1)] . Действительно, построение не добавляет к S(X) новых ориентированных графов, не являющихся D'(T). Алгоритм можно даже легко обобщить так, чтобы для каждого графа из S(X) он печатал на выходе соответствующее дерево вывода T. Обратно, если T - дерево вывода, мы можем показать индукцией по числу узлов дерева T, что D'(T) принадлежит некоторому множеству S(X). В противном случае T должно иметь вид (3.3) и D(T) "составлен" из $D_p D(T_1), \ldots, D(T_{n_p})$. По индукции и вследствие того, что при j $\neq$j' из $D(T_j)$ в $D(T_{j'})$ не проходит дуг вне D_p, дуги в $D_p D(T_1), \ldots, D(T_{n_p})$, составляющей рассматриваемый путь графа D(T), можно заменить соответствующими дугами в $D_p[D'1, \ldots D'_{n_p}]$, где $D'_j \in S(X_{pj}), 1 \leq j \leq n_p$. Поэтому ориентированный граф, включаемый в $S(X_{p0})$ на базе $D_p[D'1, \ldots, D'_{n_p}]$, просто совпадает с D'(T).

Вышеприведенный алгоритм решает задачу, поставленную в этом разделе.

Теорема. Семантические правила, добавленные к грамматике так, как это сделано в разд. 2, являются корректными тогда и только тогда, когда ни один из ориентированных графов (3.5) ни при каком выборе p и $D'_1 \subset S(X_{p1}), \ldots, D'_{n_p} \subset S(X_{n_p})$ не содержит ориентированных циклов.

Доказательство. Если (3.5) содержит ориентированный цикл, то, как было показано выше, некоторый D(T) содержит ориентированный цикл. Наоборот, если T - дерево с наименьшим возможным числом

улов, такое, что D(T) содержит ориентированный цикл, то T должно иметь вид (рис. 3.3), а D(T) "составляется" из $D_p; D(T_1), \ldots, D(T_{n_p})$. Из минимальности T следует, что ориентированный цикл включает по меньшей мере одну дугу графа D_p, и, следовательно, можно, рассуждая, как выше, все дуги, образующие этот цикл и лежащие в одном из графов $D(T_1), \ldots, D(T_{n_p})$, заменить дугами графа (3.5).

Простой язык программирования

Сейчас мы продемонстрируем, как описанный выше метод семантического определения можно применять к языкам программирования. Для простоты изучим формальное определение небольшого языка, описывающего программы для машин Тьюринга.

Машина Тьюринга (в классическом смысле) работает с бесконечной лентой, которую можно представлять себе раздел_нной на клетки. Машина может считывать или записывать символы некоторого конечного алфавита в обозреваемую в некоторый момент клетку, а также сдвигать читающее устройство на одну клетку вправо или влево. Следующая программа, например, прибавляет единицу к целому числу, представленному в двоичном виде, и печатает точку справа от этого числа. Предполагается, что в начале и в конце работы программы читающее устройство находится на первой пустой клетке справа от числа.

```
Алфавит пробел, единица, нуль, точка;
печатать "точка"; перейти на выполнить;
тест: если символ на ленте _единицаї, то
{печатать "нуль";                                                    (4.1)
  выполнить: сдвинуться влево на одну клетку;
  перейти на тест};
печатать "единица";
возврат: сдвинуться вправо на одну клетку;
если символ на ленте "нуль", то перейти на возврат.
```

Читатель, по-видимому, найдет этот язык программирования достаточно прозрачным для того, чтобы понять его, прежде чем будет дано какое-либо формальное определение, хотя это и не обязательно. Приведенная выше программа не является примером искусного программирования. Она лишь иллюстрирует некоторые черты простого языка, рассматриваемого в настоящем разделе.

Поскольку каждый язык программирования нужно как- то называть, назовем наш язык Тьюринголом. Всякая правильная программа на Тьюринголе определяет программу для машины Тьюринга; будем говорить, что программа для машины Тьюринга состоит из:

- множества "состояний" Q,
- множества "символов" Σ
- начального "состояния" $q_0 \in Q$
- конечного "состояния" $q_1 \in Q$
- и "функции переходов" δ, отображающей множество

$(Q-\{q_\infty\})x\Sigma$ в $\Sigma\{-1, 0, +1\}xQ$. Если $\delta(q, s) = (s', k', q')$, то это означает, что если машина находится в состоянии q и обозревает символ s, то она печатает символ s', сдвигает читающее устройство на k клеток вправо (сдвигу на одну клетку влево соответствует случай k = 1) и переходит в состояние q'. Формально программа машины Тьюринга определяет вычисление для ленты с "любым начальным содержимым", то есть для любой бесконечной в обе стороны последовательности

$\ldots, a_{-3}, a_{-2}, a_{-1}, a_0, a_1, a_2, a_3, \ldots$

Пример 4.2. (html, txt)

элементов алфавита Σ следующим образом. В произвольный момент вычисления существует "текущее состояние" $q \in Q$ и целочисленная величина "положение читающего устройства" p. Вначале $q = q_0$ и p = 0. Если $q \neq q_\infty$ и если $\delta(q, a_p) = (s', k', q')$, то следующим шагом вычисления будет замена значения p на p + k,

q на q' и a_p на s'. Если $q = q_\infty$, вычисление заканчивается. (Вычисление может не закончиться; для программы (4.1) это произойдет тогда и только тогда, когда a_j="единица" для всех $_j < 0$.)

Теперь, когда у нас имеется точное определение программ машин Тьюринга, мы хотим определить программу машины Тьюринга, соответствующую произвольной программе на Тьюринголе (и одновременно определить синтаксис Тьюрингола). Для этого удобно принять некоторое соглашение о форме записи.

(1) Семантическое правило " включить x в B ", связанное с синтаксическим правилом, означает, что x должен стать элементом множества B, где B - атрибут аксиомы S грамматики. Значением B будет множество всех x, для которых существует такое семантическое правило, связанное с каждым применением соответствующего синтаксического правила в дереве вывода. (Это правило можно рассматривать как сокращенную запись семантического правила

$$B(X_{p0}) = \bigcup_{j=1}^{n_p} B(X_{pj}) \cup \qquad (4.3)$$
$$\cup \{x \mid \text{"включить } x \text{ в } B \text{ "связано с } p\text{-м правилом}\},$$

связанного с каждым синтаксическим правилом. Здесь B - синтезированный атрибут, имеющийся у всех нетерминальных символов. Для терминальных символов B(x) пусто. Ясно, что эти правила позволяют получить нужное B(S).)

(2) Семантическое правило "определить f(x) = y", связанное с синтаксическим правилом, означает, что значение функции f в точке x будет равно y; здесь f - атрибут аксиомы S грамматики. Если встречается два правила, задающих значение f(x) для одного и того же значения x, то возникает ситуация ошибки, а дерево вывода, в котором она возникла, называется неправильным. Далее, к функции f можно обращаться в других семантических правилах при условии, что f(x) будет использоваться только тогда, когда значение функции для аргумента x определено. Любое дерево вывода, для которого встретилось обращение к неопределенной величине f(x), называется неправильным. (Правила такого типа важны, например, тогда, когда нужно обеспечить соответствие между описанием и использованием идентификаторов. В приведенном ниже примере в соответствии с этим соглашением неправильными будут программы, в которых один и тот же идентификатор дважды встречается в качестве метки, или в операторе перехода используется идентификатор, не являющийся меткой оператора. В сущности, это правило можно представлять себе, аналогично (1), как " **включить** (x, y) **в** f", если рассматривать f как множество упорядоченных пар; необходимо соответственно ввести дополнительные проверки на зацикленность. Признак "правильно или неправильно" можно считать атрибутом аксиомы S. Построение соответствующих семантических правил, аналогичных (4.3), которые аккуратно определяют запись "определить f(x) = y", несложно и предоставляется читателю.)

(3) Функция "новсимвол", в каком бы правиле она ни встретилась, будет вырабатывать некий абстрактный объект, причем при каждом обращении к "новсимвол" этот объект будет отличен от всех полученных при предшествовавших обращениях к новсимвол. (Эту функцию легко выразить при помощи других семантических правил, например используя атрибуты l из (2.3), принимающие в разных вершинах дерева разные значения. Функция новсимвол служит подходящим источником "сырья" для построения множеств.)

Мы видели, что соглашения (1), (2) и (3) можно заменить другими семантическими конструкциями, не использующими таких соглашений, следовательно, они не являются "базисными" для семантики. Но они чрезвычайно полезны, так как соответствуют понятиям, которыми часто пользуются, поэтому их можно считать принципиальными для метода описания семантики, представленного в настоящей статье. Эффект от введения таких соглашений состоит в том, что уменьшается общее количество атрибутов, явно присутствующих в правилах, и в том, что удается обойтись без неоправданно длинных правил. Теперь уже несложно дать формальное определение синтаксиса и семантики Тьюрингола.

Нетерминальные символы:

P (программа), S (оператор), L (список операторов), I (идентификатор), O (направление), A (символ алфавита), D (описание).

Терминальные символы:

a b c d e f g h i j k l m n o p q r s t u v w x y z . , : ; ' ' { }

алфавит перейти на печатать если символ на ленте то сдвинуться на одну клетку влево вправо

Начальный символ: p

Атрибуты:

Имя атрибута	Тип значения	Цель введения
Q	Множество	Состояния программы
Σ	Множество	Символы программы
q_0	Элемент множества Q	Начальное состояние
q_∞	Элемент множества Q	Конечное состояние
δ	Функция, отображающая (Q - {q_∞}) x Σ в Σ x {-1, 0, +1} x Q	Функция переходов
метка	Функция, отображающая цепочки букв в элементы мн-ва Q	Таблица состояний для операторных меток
символ	Функция, отображающая цепочки букв в элементы множества Σ	Таблица символов для символов ленты
следующее	Элемент множества Q	Состояние,непосредственно следующее за оператором или списком операторов
d	±1	Направление
текст	Цепочка букв	Идентификатор
начало	Элемент множества Q	Состояние в начале выполнения оператора или списка операторов (унаследованный атрибут)

Синтаксические и семантические правила см. в таблица 1.

Таблица 1.

Комментарий	№	Синтаксическое правило	Пример	Семантическое правило
Буквы	1.1 ... 1.26	A ⟶a ... A ⟶z	a ...(так же z	текст (A) = a для всех букв) текст (A) = z
Идентификаторы	2.1 2.2	I ⟶A I ⟶IA	m marilyn	текст (I) = текст (A) текст (I) = текст(I) текст

Описания	3.1	D⟶**алфавит** I	**алфавит** marilyn	**определить** символ (текст (I)) = **новсимвол**; включить символ (текст (I)) в Σ
Описания	3.2	D⟶ D, I	**алфавит** marilyn, jayne, brigitta	**определить** символ (текст (I)) = **новсимвол**; включить символ (текст (I)) в Σ
Оператор печати	4.1	S⟶печа- тать I	печатать "jayne"	определить δ (начало (S), s)= символ(текст (I)), 0, следующий (S)) для всех s ∈Σ; следующий (S) = **новсимвол; включить** следующий (S) в Q.
Оператор сдвига	4.2	S⟶ **сдви- нуться О на одну клетку**	**сдвинуться влево на одну клетку**	определить δ (начало (S), s) = (s, d(0), следующий (S) для всех s ∈Σ; следующий (S) = **новсимвол**; **включить** следующий (S) в Q.
	4.2.1	O⟶**влево**	влево	d(O) = -1.
	4.2.2	O⟶ **вправо**	**вправо**	d(O) = +1.
Оператор перехода	4.3	S ⟶перейти на I	**перейти на** boston	**определить** δ (начало (S), s)= (s, 0, метка(текст (I)) для всех s ∈Σ; следующий (S) = **новсимвол**; **включить** следующий (S) в Q.
Пустой оператор	4.4	S ⟶		следующий (S) = начало (S)
Условный оператор	5.1	S1 ⟶если символ на ленте I, то S_2	еслисимвол на ленте marilyn , то печатать jayne	**определить** δ (начало (S1), s)= (s, 0, метка (следующий (S2)) для всех s ∈Σ - символ (текст (I)); **определить** δ (начало (S1), s)= (s, 0, метка (следующий (S2)) для s = символ (текст (I)); начало (S2) = **новсимвол**; следующий (S1) = следующий (S2); **включить** начало (S2) в Q.
Помеченный оператор	5.2	S_1⟶ "I": S_2	boston: **сдвинуться влево на одну клетку**	**определить** метка (текст (I))= начало (S_1): начало (S_2)=начало (S_1); следующий (S_1)= следующий (S_2)
Составной оператор	5.3	S ⟶{L}	(печатать "jayne" перейти на boston	начало (L)=начало (S); следующий (S) = следующий (L).
Список операторов	6.1	L ⟶S	печатать "jayne"	начало (L)=начало (S); следующий (L) = следующий (S).
	6.2	L1⟶L2; S	печатать "jayne" перейти на boston	начало (L2)=начало (L1); начало (S) = следующий (L2). следующий (L1) = следующий (S).
Программа	7	P ⟶D; L	алфавит marilyn, jayne, birgitta печатать "jayne"	q=новсимвол; включить q_0 в Q; начало (L)=q_0; q_∞= следующий (L).

Отметим, что каждому оператору S соответствует два состояния: начало (S) - состояние, соответствующее первой команде, входящей в оператор (если таковая имеется), являющееся унаследованным атрибутом символа S, и следующее (S) - состояние, "следующее" за оператором, или состояние, в которое попадает машина после нормального выполнения оператора. В случае оператора перехода, однако, программа не попадет в состояние следующее (S), поскольку действие оператора состоит в передаче управления в другое место; о состоянии следующее (S) можно сказать, что оно следует за оператором "статически" или "текстуально" , а не "динамически" в ходе выполнения программы.

В таблица 1 следующее (S) является синтезированным атрибутом; можно составить аналогичные семантические правила, в которых атрибут следующее (S) будет унаследованным. При этом, правда, программы, включающие пустые операторы, будут несколько менее эффективны (см. правило 4.4). Аналогично, оба атрибута начало (S) и следующее (S) могут быть сделаны синтезированными, но это будет стоить дополнительных инструкций в программе машины Тьюринга при реализации списка операторов.

Наш пример мог бы быть проще, если бы мы использовали менее традиционную форму инструкций машины Тьюринга. Принятое нами определение требует, чтобы каждая инструкция включала действия чтения, печати и сдвига читающего устройства. Машина Тьюринга представляется при этом в виде некоей одно-плюс-одно- адресной вычислительной машины, в которой каждая инструкция определяет местоположение (состояние) следующей инструкции. Метод определения семантических правил, использованный в этом примере, где атрибут следующее (S) является синтезированным, а начало (S) - унаследованным, годится и для вычислительной машины или автомата, в котором n + 1-я инструкция выполняется после n-й. В этом случае (следующее (S) - начало (S)) есть число инструкций, "скомпилированных" для оператора S.

Создается впечатление, что такое определение Тьюрингола приближает нас к желанной цели: придать точный смысл тем понятиям, которые встречаются в неформальном руководстве по языку для программиста, причем сделать это нужно совершенно формально и однозначно. Другими словами, это определение, возможно, отвечает нашему образу мышления при изучении языка. Определение 4.1 оператора печати, например, можно легко перевести на естественный язык, написав

Оператор может иметь вид: **печатать** "I"

где I - идентификатор. Это означает, что всякий раз при выполнении этого оператора символ на обозреваемой клетке ленты будет заменен символом, обозначенным I, безотносительно к тому, какой символ находится в обозреваемой клетке. После этого выполнение программы продолжится с новой инструкции, которая определяется (другими правилами) как следующая за данным оператором

Обсуждение

Идея определения семантики с помощью синтезированных атрибутов, связанных с каждым нетерминальным символом и семантических правил, сопоставленных каждому правилу вывода, принадлежит Айронсу [6, 7]. Первоначально каждый нетерминальный символ имел ровно один атрибут, называвшийся его "трансляцией" . Эта идея использовалась Айронсом и позже другими авторами, особенно Маклюром [14] при построении "синтаксически управляемых компиляторов" , переводивших языки программирования в машинный код.

Как мы видели в разд. 2, синтезированных атрибутов достаточно (в принципе) для определения любой функции на дереве вывода. Но на практике применение наряду с синтезированными и унаследованных атрибутов, как описано в данной статье, приводит к значительным упрощениям. Определение Тьюрингола, например, показывает, что легко учитывается согласованность описаний и использований символов, а также между метками и операторами. Другой общей особенностью языков программирования, определение которой значительно упрощается в результате применения унаследованных атрибутов, является "блочная структура" . Вообще говоря, унаследованные атрибуты полезны всякий раз, когда часть значений некоторой конструкции определяется контекстом, в котором находится эта конструкция. Метод, приведенный а разд. 2, показывает, как можно формально описывать унаследованные и синтезированные атрибуты, а в разд. 3 показано, что можно не принимать во внимание проблему зацикленности (являющуюся потенциальным источником трудностей при использовании атрибутов разных типов).

Автору к настоящему времени известно несколько работ, внесших принципиальный вклад в решение задачи формального описания семантики языков программирования. Это определение Алгола 60 средствами расширенного алгоритма Маркова, данное Дебаккером [1], определение Алгола 60 с помощью λ-исчисления, принадлежащее Ландину [9, 10, 11] (см. также Бем [2, 3], определение Микро-Алгола с помощью рекурсивных функций, применяемых к программе и к _векторам состоянийї, принадлежащее Маккарти [12] (см. также Маккарти и Пэинтер [13]; определение языка Эйлер средствами семантических правил, применяемых во время синтаксического анализа программы, предложенное Виртом и Вебером [16], и определение языка PL=I, данное Венской лабораторией фирмы IBM и основанное на работе Маккарти и Ландина, а также на понятии абстрактной машины, введ_нном Элготом [4, 5]. Наиболее существенная разница между предшествующими методами и описанием языка Тьюрингол, привед_нным в таблица 1, состоит в том, что остальные определения представляют собой довольно сложные процессы, применяемые ко всей программе; можно сказать, что человек, прежде чем он поймет описание языка, должен будет понять, как устроен его компилятор. Эта трудность особенно ощутима в работе Дебаккера, определяющего машину, подобную Марковским алгоритмам, но значительно более сложную. Эта машина имеет около 800 команд. На каждом шаге вычисления машины нужно выполнять последнюю применимую команду, так что мы не можем проверить, нужно ли выполнить команду номер 100, до тех пор, пока не убедимся, что остальные 700 команд неприменимы. Кроме того, в процессе работы машины список пополняется новыми командами. Ясно, что читателю чрезвычайно трудно понять работу такой машины или формально доказать ее основные свойства. Описание Тьюрингола, напротив, определяет каждую конструкцию языка только через ее "непосредственное окружение" , сводя тем самым к минимуму взаимосвязи между определениями разных частей языка. Определение составных операторов, операторов перехода и т.д. не влияет существенно на определение оператора печати; например, любое из правил 4.1, 4.2, 4.3, 4.4, 5.1, 5.3 можно выбросить, и получится строгое определение другого языка. Такая локализация и разделение семантических

правил помогает сделать определение более понятным и кратким. Хотя определения остальных авторов, упомянутые выше, не так сложны, как определение Дебаккера, в их работах все-таки присутствуют относительно сложные зависимости между отдельными частями определения. Рассмотрим, например, формальное определение языка Эйлер, данное Виртом и Вебером [16]. Это краткое описание весьма сложного языка и потому, безусловно, оно является одним из наиболее удачных формальных определений. И все же, несмотря на то, что Вирт и Вебер проверили свое определение с помощью моделирования на вычислительной машине, весьма вероятно, что некоторые черты Эйлера удивят его создателей. Следующая программа на Эйлере синтаксически и семантически правильна, хотя после метки L нигде не встречаются двоеточия:

$$\perp\ begin\ label\ L;\ new\ A;\ A \leftarrow 0;$$
$$if\ false\ then\ goto\ L\ else\ L;$$
$$out\ 1;\ L;\ A \leftarrow A+1;\ out\ 2;$$
$$if\ false\ then\ go\ to\ L\ else$$
$$if\ A<2\ then\ go\ to\ L\ else\ out\ 3;\ L\ end\ \perp$$

Результатом работы этой программы будет 1, 2, 2, 3! Промахи такого рода не являются неожиданностью при алгоритмическом определении языка. При использовании методов разд. 4 подобные ошибки менее вероятны.

Есть основания утверждать, что ни одна из предыдущих схем (формального определения семантики) не в состоянии дать такого же краткого и простого для понимания определения Тьюрингола, как то, которое представлено выше. Кроме того (хотя детали окончательно не проработаны), оказывается, что Алгол 60, Эйлер, Микро- Алгол и PL=1 также можно определить методами разд. 4, причем все преимущества по сравнению с остальными методами сохраняются. Правда, здесь автор не может быть беспристрастным судьей, поэтому для подтверждения такой точки зрения требуется некоторый дополнительный опыт.

Отметим, что семантические правила в том виде, в котором они даны в настоящей статье, не зависят от конкретно выбранного метода синтаксического анализа. На самом деле они привязаны даже к конкретным формам синтаксиса. Единственное от чего зависят семантические правила - это имя нетерминала в

левой части синтаксического правила и имени нетерминалов в правой его части. Конкретные знаки пунктуации и порядок, в котором нетерминалы располагаются в правых частях правил, несущественны с точки зрения семантических правил. Таким образом, рассматриваемое здесь определение семантики хорошо сочетается с идеей Маккарти об "абстрактном синтаксисе" [12, 13].

Когда синтаксис неоднозначен в том смысле, что некоторые цепочки языка имеют более одного дерева вывода, семантические правила дают для каждого дерева вывода свое "значение" . Предположим, например, что к грамматике (1.3) добавлены правила

$$L_1 \to BL_2,$$
$$v(L_1) = 2^{l(L_2)}v(B) + V(L_2),$$
$$l(L_1) = l(L_2) + 1.$$

Грамматика в результате становится синтаксически неоднозначной, но остается по-прежнему семантически однозначной, поскольку атрибут v(N) имеет одно и то же значение для всех деревьев вывода. С другой стороны, если изменить правило 5.2 определения Тьюрингола с S →I : S на S →S : I, грамматика станет неоднозначной как синтаксически, так и семантически.

Дополнительные материалы: Атрибутные грамматики

Введение

Среди всех формальных методов описания языков программирования атрибутные грамматики получили, по- видимому, наибольшую известность и распространение. Причиной этого является то, что формализм атрибутных грамматик основывается на дереве разбора программы в КС-грамматике, что сближает его с хорошо разработанной теорией и практикой построения трансляторов. Вместе с тем выяснилось, что реализация вычислителей для атрибутных грамматик общего вида сталкивается с большими трудностями. В связи с этим было сделано множество попыток рассматривать те или иные классы атрибутных грамматик, обладающими "хорошими" свойствами. К числу таких свойств относятся, прежде всего, простота алгоритма проверки атрибутной грамматики на зацикленность и простота алгоритма вычисления атрибутов для атрибутных грамматик данного класса. В предлагаемой статье дается обзор работ, посвященных этим вопросам.

Определение атрибутных грамматик

Пусть G - КС-грамматика: G = (T, N, P, Z), где T, N, P, Z, соответственно, множество терминальных символов, нетерминальных символов, множество правил вывода и аксиома грамматики. Правила вывода КС-грамматики будем записывать в виде

p : $X_0 \to X_1 \dots X_n$ (p)

и будем предполагать, что G - редуцированная КС- грамматика, то есть в ней нет нетерминальных символов, для которых не существует полного дерева вывода, в которое входит этот нетерминал. С каждым символом $X \in N \cup T$ свяжем множество A(X) атрибутов символа X. Некоторые из множеств A(x) могут быть пусты. Запись a(X) означает, что $a \in A(X)$.

С каждым правилом вывода $p \in P$ свяжем множество F семантических правил, имеющих следующую форму:

$a_0(i_0) = fpa_0(i_0)(a_1(i_1), \dots , a_j(i_j))$;

где $i_k \in [0, n_p]$ - номер символа правила p, а $a_k(i_k)$ - атрибут символа Xi_k, то есть $a_k(i_k) \in A(Xi_k)$. В таком случае будем говорить, что $a_0<i_0>$ "зависит" от $a_1(i_1), \dots , aj(i_j)$ или что $a_0(i_0)$ "вычисляется по " $a_1(i_1), \dots , a_j(i_j)$.

В частном случае j может быть равно нулю, тогда будем говорить, что атрибут $a_0(i_0)$ "получает в качестве значения константу"

КС-грамматику, каждому символу которой сопоставлено множество атрибутов, а каждому правилу вывода - множество семантических правил, будем называть атрибутной грамматикой (AG).

Назовем атрибут $a(X_0)$ синтезируемым, если одному из правил вывода $p : X_0 \to X_1 \dots Xn_p$ сопоставлено семантическое правило $a(0) = fa(0)(...)$. Назовем атрибут $a(X_i)$ наследуемым, если одному из правил вывода $p : X_0 \to X_1 \dots Xn_p$ сопоставлено семантическое правило $a(i) = fa(i)(...)$, $I \in [1, n_p]$. Множество синтезируемых атрибутов символа X обозначим через S(X), наследуемых атрибутов - через I(X).

Пусть правилу вывода $p : X_0 \to X_1 \dots Xn_p$ приписано семантическое правило $a_0(i_0) = fpa_0(i_0)(a_1(i_1), \dots , a_j(i_j))$. Без снижения общности будем считать, что $a_k(i_k) \in I(X_0) \cup np_n = 1S(X_n)$, $k \in [1, j]$ то есть атрибут может зависеть только от наследуемых атрибутов символа левой части и синтезируемых атрибутов символов правой части (условие Бошмана). Кроме того, будем считать, что значение атрибутов терминальных символов - константы, то есть их значения определены, но для них нет семантических правил, определеяющих их значения.

Атрибутированное дерево разбора

Если дана атрибутная грамматика AG и цепочка, принадлежащая языку, определяемому G, то можно построить дерево разбора этой цепочки в грамматике G. В этом дереве каждая вершина помечена символом грамматики G. Припишем теперь каждой вершине множество атрибутов, сопоставленных символу, которым помечена эта вершина. Атрибуты, сопоставленные вхождениям символов в дерево разбора, будем называть вхождениями атрибутов в дерево разбора, а дерево с сопоставленными каждой вершине атрибутами - атрибутированным деревом разбора.

Между вхождениями атрибутов в дерево разбора существуют зависимости, определяемые семантическими правилами, соответствующими примененным синтаксическим правилам.

Для каждого синтаксического правила $p \in P$ определим D(p) - граф зависимостей атрибутов символов, входящих в правило p, как ориентированный граф, вершинами которого служат атрибуты символов, входящих в правило p, и в котором идет дуга из вершины b(i) в вершину a(j) тогда и только тогда, когда синтаксическому правилу p сопоставлено семантическое правило

$a(j) = fpa(j)(... , b(i), ...)$, $i, j \in [0, n]$:

Граф зависимостей D(t) дерева разбора t цепочки, принадлежащей языку грамматики G, определим как ориентированный граф, полученный объединением графов зависимостей всех примененных в t синтаксических правил.

Незацикленные атрибутные грамматики

Атрибутная грамматика называется незацикленной, если графы зависимостей деревьев всех цепочек, принадлежащих языку, определяемому грамматикой G, не содержат циклов, и зацикленной, если существует хотя бы одна цепочка, принадлежащая языку, для дерева разбора которой граф D(t) содержит ориентированный цикл.

Теорема B.1. Задача определения того, является ли данная атрибутная грамматика зацикленной, имеет экспоненциальную временную сложность, то есть существует константа $c > 0$ такая, что любой алгоритм, проверяющий на зацикленность произвольную атрибутную грамматику размера n, должен работать более, чем $2cn/\log n$ шагов на бесконечно большом числе грамматик [1, 2].

Кнутом [3] был предложен алгоритм проверки атрибутных грамматик на зацикленность.

Пусть D(p) - граф зависимостей атрибутов правила вывода p, а Gi - произвольный ориентированный граф, вершинами которого служат атрибуты символа Xi правой части правила вывода p. Обозначим $D_p[G_1, \ldots, G_{n_p}]$ ориентированный граф, полученный из D(p) добавлением дуг, идущих из b(i) в a(i), если в графе G_i есть дуга из b в a. Через Гобозначим множество ориентированных графов с вершинами - атрибутами символа X, через $D_p[G_1, \ldots, G_{n_p}]$ - ориентированный граф, вершинами которого служат атрибуты символа X в правиле вывода p : $X_0 \rightarrow X_1 \ldots X_{np}$ и в котором идет дуга из вершины b в вершину a тогда и только тогда, когда в $D_p[G_1, \ldots, G_{n_p}]$ есть путь из b(0) в a(0).

Алгоритм B.1. (Алгоритм Кнута). Проверка атрибутной грамматики на зацикленность.

begin
for $X \in N$ do $\Gamma_x := 0$ end;
for $T \in N$ do $\Gamma_x := \{A(X)\}$ end;
{$A(X)$ - граф со множеством вершин-множеством
атрибутов символа X и пустым множеством дуг}
finish := false; cycle := false;
while (not finish) and (not cycle) do
if $(\exists\, p : X_0 \rightarrow X_1 \ldots X_{np})$ & $(\exists\, G_i \in \Gamma_{x_i}, i \in [0, n_p])$
такие, что $D_p[G_1 \ldots Gn_p]$ содержит цикл
then cycle := true
else if $(\exists\, p : X_0 \rightarrow X_1 \ldots X_{np})$ & $(\exists\, G_i(\Gamma_{x_i}, i \in [0, n_p])$
такие, что $D_p[G_1 \ldots Gn_p] \in \Gamma_{x0}$
then $\Gamma_{x0} := \Gamma_{x0}\ \{D_p[G_1 \ldots Gn_p]\}$
else finish := true
end end
end end.

Теорема B.2. Атрибутная грамматика AG незациклена тогда и только тогда, когда ни один из графов $D_p[G_1 \ldots Gn_p]$ не содержит ориентированных циклов, то есть когда алгоритм B.1. заканчивается со значением cycle = false.

Теорема B.3. Алгоритм Кнута проверки на зацикленность атрибутной грамматики размера n требует в общем случае $\exp(cn^2)$ шагов.

Вычислительные последовательности и корректность. Определение визита

Назовем вычислительной последовательностью [4] для дерева вывода t в AG последовательность вида:

$c_s = (n_1, A_1)(n_2, A_2)(n_2, A_2) \ldots (n_r, A_r)$;

где

1. n_j - внутренняя вершина t (в частности, корень);
2. если $n_j \# n_j + 1$, то $n_j + 1$ - отец, сын или брат n_j ;
3. A_j - либо подмножество синтезируемых атрибутов n_j , либо подмножество наследуемых атрибутов (то есть либо либо $A_j \in S(Xn_j)$), $A_j \in S(Xn_j)$);
4. $n_1 = n_r$ - корень дерева;
5. атрибуты A_j не зависят от A_j для $i \geq j$;

6. рассмотрим какую-либо внутреннюю вершину nдерева t. Тогда вычислительную последовательность cs можно записать в следующем виде: $c_s = u_1(n, B_1)u_2(n, B_2) \dots (n, B_h)u_{h+1}$, где подпоследовательности $u_1 \dots u_{h+1}$ не содержат элементов вида (n, A). Тогда
    1. $B_j \subseteq I(X_n)$, если j нечетно;
    2. $B_j \subseteq S(X_n)$, если j четно,
    3. $U_j \in [1, n]$, $B = A(X_n)$ - вычисляются все атрибуты каждого символа X;
    4. $B_i \cup B_j = 0$, если i#j - все атрибуты вычисляются по одному разу.
    5. пусть $c_s = c_{s1}<n, B_j><n_1, A_1><n_2, A_2>\dots<n, B_{j+1}> c_{s2}$, если j нечетно (четно), то n_j - вершины поддерева с корнем n (вершины t вне поддерева с корнем n).

Таким образом при входе "вниз" в поддерево вычисляются некоторые наследуемые атрибуты корня поддерева, при возврате из поддерева вычисляются некоторые синтезируемые атрибуты корня поддерева.

Назовем незацикленную атрибутную грамматику корректной, если для всякого ее атрибутированного дерева существует вычислительная последовательность.

Теорема B.4. Незацикленная атрибутная грамматика корректна тогда и только тогда, когда для каждого правила $p : X_0 \longrightarrow X_1 \dots X_{np}$ если $a \in I(X_i)$, $i \in [1, n_p]$, то имеется в точности одно семантическое правило, сопоставленное p и определяющее значение $a(X_i)$, и если $a \in S(X_0)$, то имеется в точности одно семантическое правило, сопоставленное p и определяющее значение $a(X_0)$.

Теорема B.5. Сложность проверки незацикленной атрибутной грамматики на корректность линейна по размеру атрибутной грамматики.

Пусть t - дерево вывода и n - его внутренняя вершина. Рассмотрим вычислительную последовательность для t вида $c_s = c_{s1}<n, B_1>c_{s2}<n, B_2>c_{s3}$, где n входит в c_{s1} четное число раз, и не входит в c_{s2}. Последовательность c_{s2} обходит поддерево с корнем n. Будем говорить, что $<n, B_1>c_{s2}<n, B_2>$ определяет визит в поддерево с корнем $_n$ и что вершина n в результате этого визита посещается один раз. Таким образом, если n входит в c_s 2h раз, то n посещается h раз.

Чистые многовизитные грамматики

Будем говорить, что атрибутированное дерево k-визитно, если существует вычислительная последовательность c_s для t такая, что никакая вершина n из t не посещается более k раз.

Атрибутная грамматика называется чистой k-визитной (PMV), если каждое атрибутированное дерево вывода t в AG k-визитно [5, 7].

Теорема B.6. Для всякой корректной атрибутной грамматики существует k такое, что грамматика является чистой k-визитной.

На самом деле это k не превосходит максимального по всем символам грамматики числа синтезируемых или наследуемых атрибутов.

Следствием этого являются две следующие теоремы.

Теорема B.7. Сложность задачи определения того, является ли произвольная атрибутная грамматика чистой k-визитной для какого-нибудь $k > 0$, экспоненциальна.

Эта задача просто совпадает с задачей определения корректности атрибутной грамматики.

Теорема B.8. Сложность задачи определения того, является ли произвольная атрибутная грамматика чистой k-визитной для фиксированного k также экспоненциальна.

Атрибуты всякого дерева t чистой k-визитной атрибутной грамматики можно вычислить с помощью следующего алгоритма:

Алгоритм В.2. Вычисление атрибутов чистой k- визитной атрибутной грамматики

```
procedure  визит_в_поддерево (n, i);
{n - корень поддерева;
i - номер визита в это поддерево}
{Предполагается, что в вершине n
применено правило вывода p}
begin вычислить некоторые наследуемые атрибуты Xn;
{эти атрибуты определяются <nj1, A> для начала i-го
визита в соответствующей вычислительной
последовательности}
визит_в_поддерево (n, i);
{Xnij - символ правой части правила pg
.
.
.
визит_в_поддерево (njl, ijl);
вычислить некоторые синтезируемые атрибуты Xn
end;
begin for j := 1 to k do визит_в_поддерево(r, j)
  {r - корень дерева}
end end.
```

В зависимости от того, какие ограничения будут накладываться на порядок посещения вершин и выбор атрибутов, вычисляемых на том или ином визите, будут получаться те или иные классы атрибутных грамматик.

Абсолютно незацикленные атрибутные грамматики

Обозначим IOx ориентированный граф, вершинами которого являются атрибуты символа X и из вершины b идет дуга в вершину a тогда и только тогда, когда в атрибутной грамматике AG существует такое поддерево с корнем X, что в графе зависимостей этого поддерева существует путь из b в a. Через D_p^* обозначим граф $D_p[IOX_1, ..., IOX\ n_p]$.

Атрибутная грамматика называется абсолютно незацикленной (ANC), если ни один из графов D не содержит ориентированных циклов [6].

Абсолютно незацикленные атрибутные грамматики образуют собственный подкласс незацикленных атрибутных грамматик.

Пример В.1. Незацикленная атрибутная грамматика, не являющаяся абсолютно незацикленной (рис. В.1.).

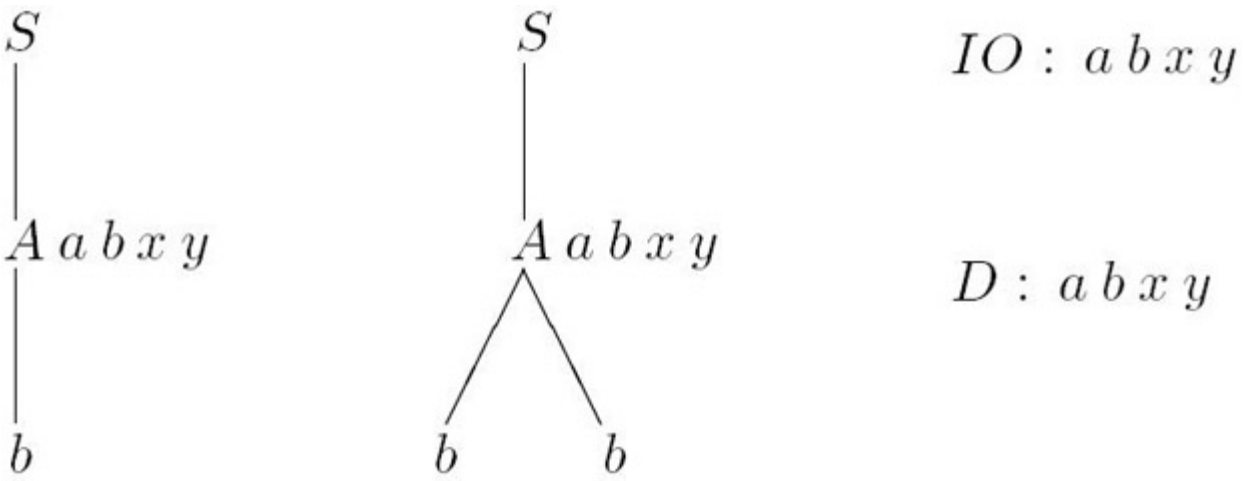


Рис. В.1.

Эта грамматика порождает всего два слова b и bb. Каждое из двух деревьев порождает незацикленные графы зависимостей, однако грамматика не является абсолютно незацикленной. Происходит это от того, что зависимости, реализуемые в разных деревьях, "накладываются" на один граф IO.

Для построения графов IO имеется простой полиномиальный алгоритм:

Алгоритм В.3. Построение графов IO атрибутной грамматики AG.

begin Положить $IOx := \{A(X)\}$
для каждого $X \in N$, {граф без дуг}
while имеется правило p с левой частью X такое, что в
D_p^* есть путь из i в s, $i \in I(X)$,
$s \in S(X)$, нов IOxнетдугиизibs
do добавить эту дугу в IOx
end end:

Поскольку этот алгоритм полиномиален и задача определения наличия ориентированных циклов в графе также полиномиальна, справедлива следующая теорема:

Теорема В.9. Задача определения, является ли данная атрибутная грамматика абсолютно незацикленной, полиномиальна по длине атрибутной грамматики. Абсолютно незацикленные атрибутные грамматики интересны тем, что для них имеется полиномиальный алгоритм планирования визитов.

Обозначим через A(p) множество атрибутов символов синтаксического правила p. Рассмотрим атрибутированное дерево t в AG и некоторую его внутреннюю вершину n, в которой применено правило вывода p. В каждый момент времени в процессе вычисления атрибутов дерева t каким- либо алгоритмом вычисления какие-то атрибуты из A(p) вычислены, а какие-то нет. Назовем состоянием правила, примененного в дереве вывода, множество вычисленных атрибутов символов, входящих в это правило. Начальным состоянием для каждого правила является множество {a<k> j Xk∈T}.

План - это последовательность инструкций вида fpa<k> или V ISIT (k, I), где $I \subset I(X_k)$, если k - номер символа правой части правила p. I называется входным множеством. План всегда завершается инструкцией $S(A), A \subset A(p)$ - перевести правило p в состояние A. Инструкция fpa<k> вычисляет атрибут a<k>, V ISIT(k, I) осуществляет визит в поддерево k-го символа правой части со значениями наследуемых атрибутов I этого символа, инструкция S изменяет состояние правила.

Обозначим Dpa<k> - множество аргументов семантического правила fpa<k>. Будем говорить, что семантическое правило f готово к вычислению в состоянии A правила p, если a<k> ∉A, но $Dpa < k > \subset A$.

Если p : $X_0 \rightarrow X_1 \ldots X_{np}$ и правило p находится в состоянии A, то результатом k-го поддерева, k∈[1, n_p], будем называть множество {a<k> j a<k> ∉A, a∈S(Xk) и для каждого i<j>, для которого есть дуга из i<j> в a<k> в IOXk, i<j>∈A} (предполагается, что у каждого нетерминала есть хотя бы один синтезируемый атрибут).

Планирование осуществляется нижеследующим алгоритмом. Результат работы алгоритма заносится в двумерный массив EVAL, одним входом в который служит состояние правила, другим - входное множество. Строка - это строка инструкций Stv - вектор состояний правил; он передается как аргумент процедуре PLAN, затем дублируется внутри процедуры PLAN и обращение к PLAN меняет значение своего аргумента в точке вызова (что обозначено знаком var перед параметром Stv процедуры PLAN). Если для некоторого элемента таблицы EVAL в процедуре PLAN начато построение плана, то этот элемент метится значком @, чтобы избежать бесконечной рекурсии. Будем говорить, что функция f готова к вычислению, если все ее аргументы определены, но атрибут, который она вычисляет не определен.

Алгоритм В.4. Построение планов для каждого возможного состояния каждого правила.

```
var EVAL : array[состояние, входное множество] of строка;
   St : array [1 .. P] of состояние;
   fP - число синтаксических правил}
   procedure PLAN( p, I, var Stv);
      {p - номер синтаксического правила, I - входное
      множество, Stv - вектор состояния правил}
      var S : строка {строящийся план};
      LStv : array [1 .. p] of состояние;
      {локальный вектор состояния правил}
      A : set of атрибут;{состояние правила p}
      stop: boolean;
   begin
      if (EVAL [Stv[p], I] пуст)
      then
      A := I [ Stv[p], s := пусто , LStv := Stv;
      stop := false, EVAL [Stv[p], I] :='@'
      repeat
         if (∃ fpa<k> готовая к вычислению)
         then
            s := s || fpa<k>, A := A + a<k>
         else
      if (∃ поддерево k, результат Y которого не пуст)
      then
         s := s || V ISIT(k, I(Xk) ∩ A), A := AUY ;
         for pi : Xk⟶u do
         PLAN(pi, I(Xk) ∩ A, LStv)
         {в этой точке меняется значение LStv[pi]}
         end
      else stop := true
      end end
      until stop;
      EVAL [Stv[p], I] := s || st(A), Stv := A
      {Stv[p] меняется в точке вызова}
      end end;
   {тело программы} begin for I := 1 to p do
      St[i] := множество атрибутов терминалов правила i;
      PLAN({}, {}, St)
   end end.
```

Вычисление атрибутов на дереве t заключается в выполнении построенных планов в соответствии с изменениями состояний правил и осуществляется следующей программой:

```
begin каждое правило дерева t перевести
в начальное состояние,
определяемое множеством атрибутов терминалов;
V ISIT(корень, {})
end.
```

Простые многовизитные атрибутные грамматики

Атрибутная грамматика называется простой k-визитной, если для каждого нетерминала X∈V существует разбиение A1(X), ... , Am(X) множества атрибутов A(X), где m∈[1, k] и m может зависеть от X, то есть m = m(X), такое, что для любого дерева вывода t слова из G существует вычислительная последовательность, при которой для любого вхождения X в t все атрибуты Aj(X) вычисляются при выполнении j-го визита в поддерево с корнем X для всех j∈[1;m(X)] [7].

Атрибутная грамматика называется простой многовизитной (SMV), если она является простой k-визитной для какого-нибудь k.

Существуют абсолютно незацикленные атрибутные грамматики, не являющиеся простыми многовизитными.

Пример В.2. Здесь атрибуты a и b символа A левого поддерева вычисляются на первом визите, а x и y - на втором. Для символа A правого поддерева наоборот - атрибуты x и y вычисляются на первом визите, а и b - на втором (рис. В.2).

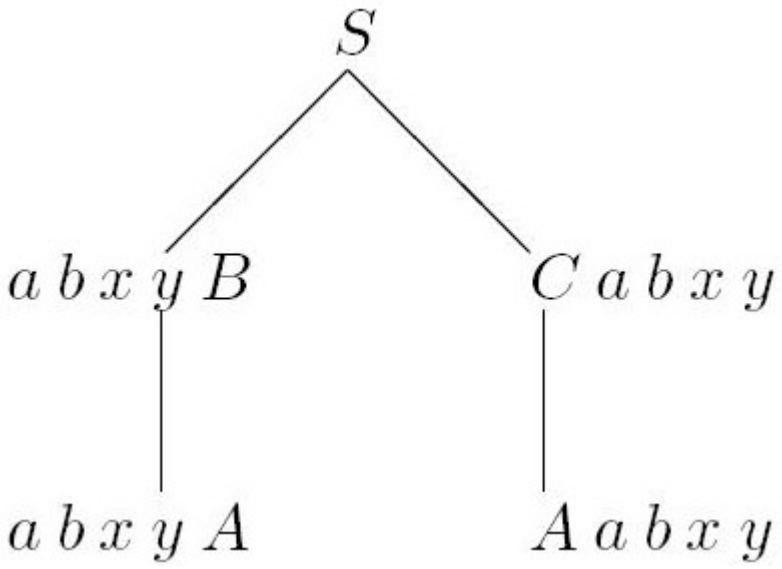


Рис. В.2.

Теорема В.10. Всякая простая k-визитная грамматика является абсолютно незацикленной [7].

Теорема В.11. Задача определения того, является ли произвольная атрибутная грамматика простой многовизитной, NP-полна [7]. Мало того, NP-полна даже задача определения простой 2-визитности [7] Если для каждого символа дано разбиение его атрибутов по визитам, то алгоритм вычисления атрибутов дерева принимает следующий вид:.

Алгоритм В.5. Вычисление атрибутов в простой многовизитной грамматике.

```
procedure визит_в_поддерево(n, i);
begin вычислить наследуемые атрибуты I(Xn);
 визит_в_поддерево (nj1, ij1);
 ...
 визит_в_поддерево (njm, ijm);
 вычислить синтезируемые атрибуты S(Xn)
end;
beginforj := 1tok(Xr)do
 визит_в_поддерево (r, j){r - корень}
end:
```

Одновизитные атрибутные грамматики

Интересный частный случай простых многовизитных грамматик представляют одновизитные грамматики (IV)[8].

Графом BG братьев правила p будем называть граф, вершинами которого являются символы правой части правила p : $X_0 \rightarrow X_1 \ldots X_{np}$ и из Xi в Xj идет дуга тогда и только тогда, когда какие-либо элементы I(Xj) зависят от каких-либо элементов S(Xi), i, j$\in$[1, n]

Теорема В.12. Атрибутная грамматика является одновизитной тогда и только тогда, когда ни один из графов братьев BGp не содержит ориентированных циклов [9].

Из этой теоремы непосредственно следует

Теорема В.13. Задача определения того, является ли произвольная атрибутная грамматика одновизитной, полиномиальна.

Задача планирования визитов для одновизитных грамматик сводится к нахождению какого-нибудь линейного порядка братьев каждого правила, удовлетворяющего частичному порядку, определяемому графом братьев BGp: Алгоритм вычисления атрибутов для одновизитных грамматик выглядит следующим образом:

Алгоритм В.6. Вычисление атрибутов в одновизитной грамматике.

```
procedure визит_в_поддерево (n);
begin вычислить наследуемые атрибуты I(X);
 в соответствии с линейным порядком символов
 правой части правила
 do визит_в_поддерево (n);
 вычислить синтезируемые атрибуты S(X)
end;
begin визит_в_поддерево(r) {r - корень}
end.
```

Многопроходные грамматики

Пусть на последовательность визитов наложено такое ограничение, чтобы они образовали последовательные обходы дерева разбора либо сверху-вниз слева-направо, либо сверху- вниз справа-налево.

Атрибутная грамматика называется чистой k- проходной в обоих направлениях, если существует такая последовательность из k обходов <d1 ... dl> (каждое di - либо справа-налево, либо слева-направо), что атрибуты любого дерева вывода могут быть вычислены в результате выполнения этой последовательности обходов [5].

Атрибутная грамматика называется чистой многопроходной в обоих направлениях (PBD), если она является чистой k-проходной в обоих направлениях для какого-нибудь k.

Пример В.3. Существуют атрибутные грамматики, не являющимися чистыми многопроходными ни для какого k (рис. В.3).

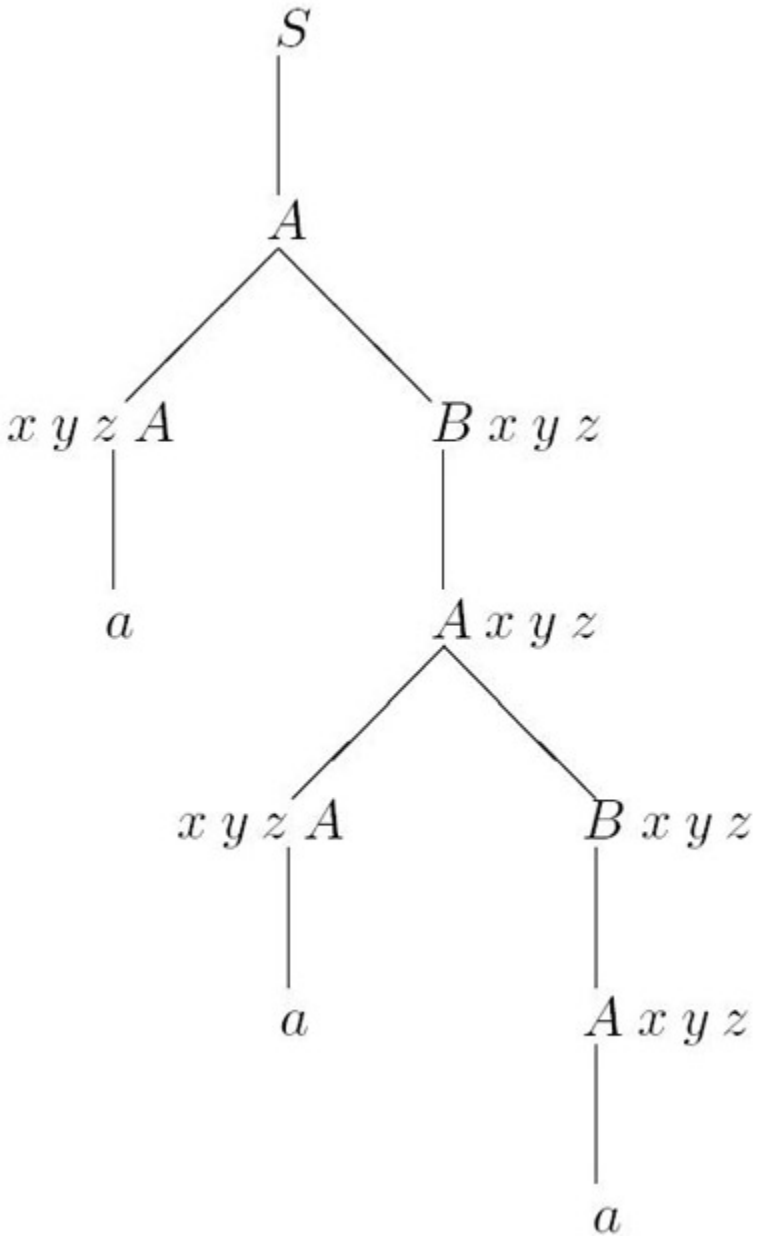


Рис. В.3.

Число необходимых проходов в этом примере зависит от глубины дерева и может быть сколь угодно большим.

Очевидно, что грамматика примера рис. В.1 не является чистой многопроходной и нетрудно видеть, что грамматика примера рис. В.1 является абсолютно незацикленной.

Теорема В.14. Задача определения того, является ли произвольная атрибутная грамматика чистой многопроходной в обоих направлениях, зависит экспоненцально от размера атрибутной грамматики.

Атрибутная грамматика называется чистой k-проходной слева-направо, если атрибуты любого дерева вывода в ней могут быть вычислены за k обходов дерева вывода слева- направо [2, 5].

Атрибутная грамматика называется чистой многопроходной слева-направо (PLR), если она является чистой k-проходной слева-направо для какого-нибудь k.

Теорема В.15. Задача определения того, является ли произвольная атрибутная грамматика чистой многопроходной слева-направо, зависит экспоненциально от размера атрибутной грамматики.

Пример В.4. Существуют атрибутные грамматики, вычисляющиеся в обоих направлениях, но не вычисляющиеся в одном (рис. В.4).

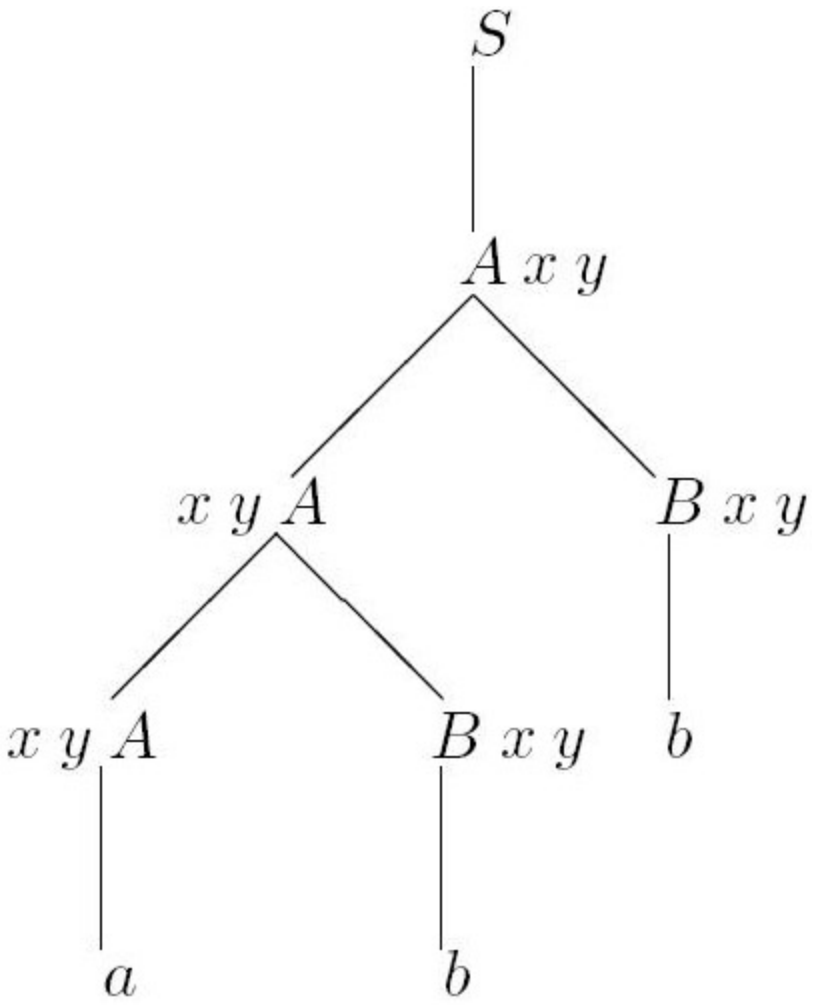


Рис. В.4.

Атрибутная грамматика называется простой k- проходной в обоих направлениях, если существует такая последовательность обходов и такое разбиение атрибутов A1(X), ... , Am(X), m = m(X), m∈[1, k], каждого символа, что все атрибуты из множества Ai(X) вычисляются на i-ом проходе дерева [5].

Атрибутная грамматика называется простой многопроходной в обоих направлениях (SBD), если она является простой k-проходной в обоих направлениях для какого-нибудь k.

Грамматика примера 9.2 является простой многопроходной в обоих направлениях, но не является чистой многопроходной слева-направо. Грамматика примера 9.3 является чистой многопроходной слева-направо, но не является простой многопроходной в обоих направлениях. Так что между классами PLR и SBD нет отношения включения.

Теорема В.16. Задача проверки того, является ли произвольная атрибутная грамматика простой k- проходной в обоих направлениях, полиномиально сложна [5].

Пример В.5. Существуют грамматики, являющиеся чистыми многопроходными, но не являющиеся простыми многопроходными (рис. В.5).

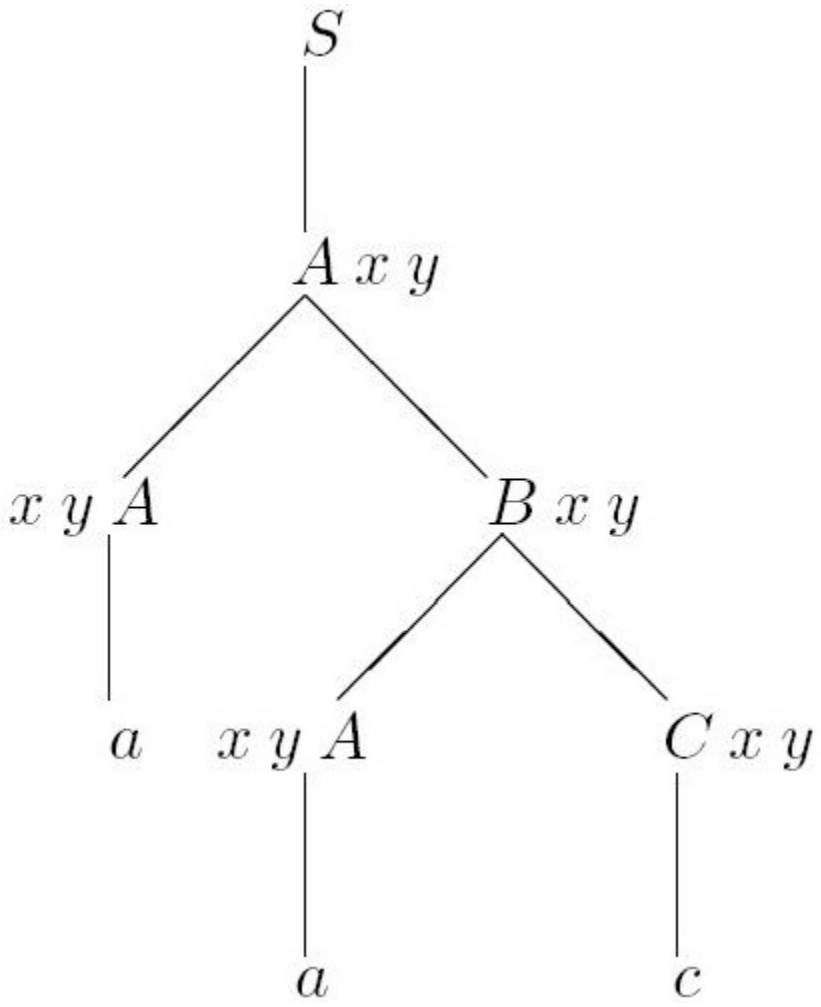


Рис. B.5.

Атрибутная грамматика называется простой k-проходной слева-направо, если существует такое разбиение атрибутов каждого символа A1(X), ..., Am(X), m = m(X), m∈[1, k]; что все атрибуты из множества Ai(X) вычисляются на i-ом обходе дерева слева-направо сверху-вниз.

Атрибутная грамматика называется простой многопроходной слева-направо (SLR), если она является простой k-проходной слева-направо для какого-нибудь k.

Пример B.6. Эта грамматика является простой однопроходной справа-налево, но не является простой однопроходной слева-направо (рис. B.6).

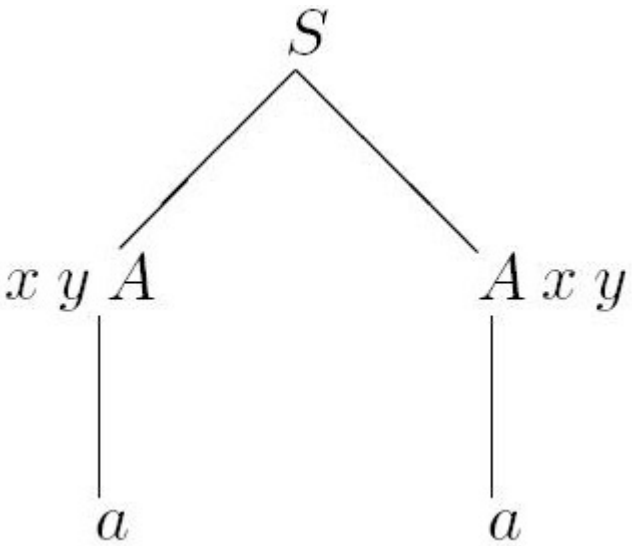


Рис. B.6.

Будем говорить, что между атрибутами a и b имеет место отношение prec, если существует правило вывода p : $X_0 \to X_1 \ldots X_{np}$ с вхождениями атрибутов a<j> и b<k> такое, что a<j> используется в качестве аргумента при вычислении вхождения b<k>.

Между атрибутами a и b имеет место отношение L, если aprecb и для каждого правила вывода p : $X_0 \to X_1 \ldots X_{np}$ с вхождениями атрибутов a<j> и b<k> такими, что b<k> зависит от a<j>, имеет место j<k.

Графом LR-предшествования для AG назовем граф, вершинами которого являются атрибуты всех символов AG и из вершины a в вершину b идет дуга, тогда и только тогда, когда имеет место отношение aprecb. Если имеет место отношение aLb, то дуга (a, b) помечена меткой L, иначе она помечена меткой L. Приведем алгоритм проверки принадлежности классу SLR, который одновременно производит разбиение (если это возможно) атрибутов каждого символа по обходам.

Алгоритм B.7.

```
* Построение функции проходов pass(a) атрибутов
символов,
* дающей либо минимальный номер прохода, на котором
атрибут
* может быть вычислен, либо неопределено,
* если атрибут не может быть занесен ни в
* один из элементов разбиения A(X), aA(X).
begin строим граф LR предшествования для AG;
Полагаем COST(a) неопредел_нной для всех вершин a;
m := -1, repeat m := m + 1;
Положить COST(a) равной m для всех вершин,
  для которых COST(a) неопределено;
repeat для вершины a такой, что COST(a) = m
  положить COST(a) неопредел_нным;
Если существует вершина b и дуга (b, a) такие, что
  (COST(b) = m) and (B, a) имеет метку L)
  or (COST(b) неопределено)
  until нельзя найти такой вершины a,
    что COST(a) можно сделать неопредел_нным;
  until либо для всех a COST(a) вычислена,
  либо не существует вершины b такой,
    что COST(b) = m, для всех a∈A
  if (COST(a) определено)
    then pass(a) := COST(a) + 1
    else pass(a) := неопределено
  end
end.
```

Этот алгоритм легко обобщается на простые многопроходные в обоих направлениях атрибутные грамматики [5].

Совсем простым частным случаем LR многопроходных атрибутных грамматик являются однопроходные атрибутные грамматики.

Теорема B.17. Атрибутная грамматика является LR однопроходной тогда и только тогда, когда ни один из графов братьев для правил вывода не содержит дуг из X в X для i ≥ j.

Таким образом между рассмотренными классами атрибутных грамматик имеет место включение, показанное на рис. B.7:

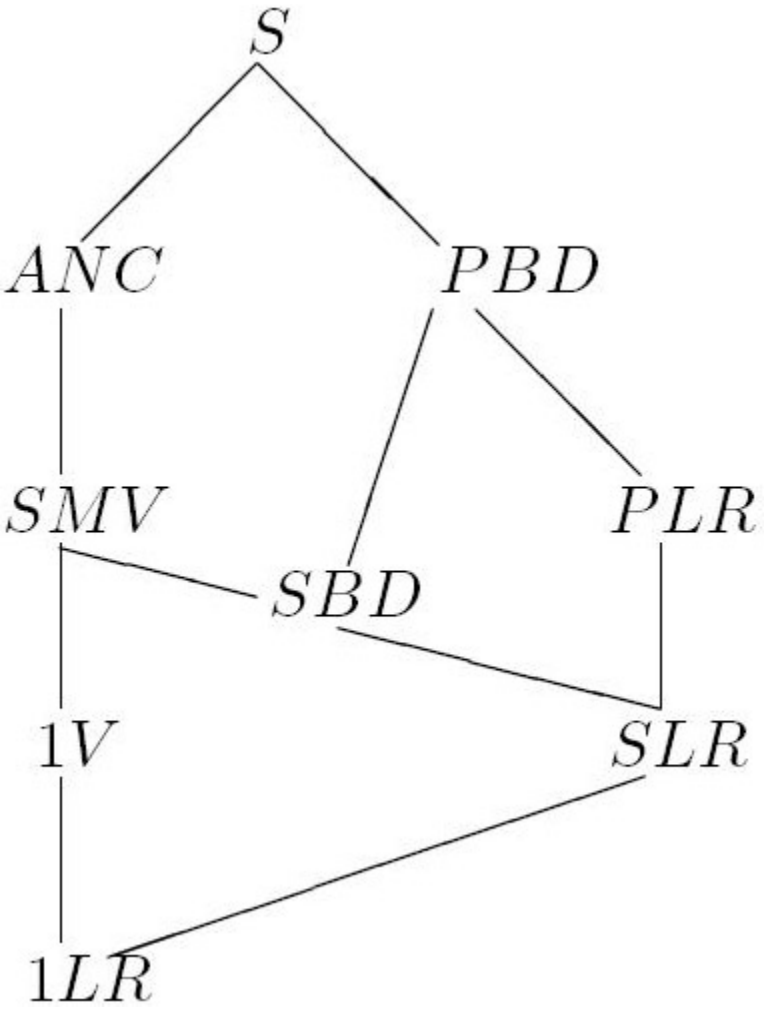


Рис. В.7.

Дополнительные материалы: Задачи по разделам курса

Языки и их представление

Алфавиты, цепочки и языки

1. Пусть A = {ab, c} и B = {c, ca} - два формальных языка над алфавитом {a, b, c}. Найти следующие формальные языки:
    1. $A \cup B$;
    2. $A \setminus B$;
    3. A^2;
    4. $A^2 \setminus B^2$;
    5. AB.

Представление языков

- Для языка $L = \{x \in \{a, b\}^* \mid |x|_a$ - четное, $|x|_b$ - нечетное$\}$ постройте
    1. Детерминированный конечный автомат;
    2. По нему - регулярное выражение;
    3. По этому выражению - грамматику;
    4. По полученной грамматике перейдите по GN-теореме к N- автомату.

Грамматики

2.3.1. Принадлежит ли цепочка x = abaababb языку, порождаемому грамматикой с правилами:

S ⟶ SaSb|ε

2.3.2. Принадлежит ли цепочка x = (()())() языку, порождаемому грамматикой с правилами:

S ⟶ SA|A
A ⟶ (S)|()

2.3.3. Принадлежит ли цепочка x = 00011011 языку, порождаемому грамматикой с правилами:

S ⟶ SS|A
A ⟶ 0A1|S|01

2.3.4. Принадлежит ли цепочка x = 0111000 языку, порождаемому грамматикой с правилами:

S ⟶ A0B|B1A
A ⟶ BB|0
B ⟶ AA|1

2.3.5. Верно ли соотношение a*cb* 2 L(G) для следующей грамматики G?

S ⟶ Bab|aDa; A ⟶ Dc|cA; B ⟶ Sb|b;
D ⟶ AB|aD.

2.3.6. Верно ли соотношение ab*c* 2 L(G) для следующей грамматики G?

S ⟶ SAS|A; A ⟶ Ac|Da|b; B ⟶ DaD;
D ⟶ ABD|AB.

2.3.7. Верно ли соотношение ca*b* 2 L(G) для следующей грамматики G?

S ⟶ bcD|aB; A ⟶ Db|cA; B ⟶ bS|ε;
D ⟶ BA|cD.

2.3.8. Верно ли соотношение c*ab* 2 L(G) для следующей грамматики G?

S ⟶ ASS|A; A ⟶ c|Ab|aD; B ⟶ aDD;
D ⟶ AB|BaB.

2.3.9. Пусть грамматика G определяется правилами

S ⟶ AB; AB ⟶ CBb; CB ⟶ ABB;
A ⟶ a; aB ⟶ a:

Какому классу (по Хомскому) она принадлежит? Порождается ли L(G) грамматикой более узкого класса?

2.3.10. Пусть грамматика G определяется правилами

S ⟶ aAbB; AbB ⟶ aAbB; bBb ⟶ bb; A ⟶ ε.

Какому классу (по Хомскому) она принадлежит? Порождается ли L(G) грамматикой более узкого класса?

2.3.11. Пусть грамматика G определяется правилами

S → AaB; AaB → aAaBb; aBb → abb; A → ε.

Какому классу (по Хомскому) она принадлежит? Порождается ли L(G) грамматикой более узкого класса?

2.3.12. Пусть грамматика G определяется правилами

S → AB; AB → aDB; DB → ABB; B → b; Ab → b.

Какому классу (по Хомскому) она принадлежит? Порождается ли L(G) грамматикой более узкого класса?

2.3.13. Какому классу по Хомскому принадлежит:

а) Грамматика с правилами:

S → AS|ε; A → a|b:

б) Язык, порождаемый этой грамматикой?

2.3.14. Какому классу по Хомскому принадлежит:

а) Грамматика с правилами:

S → AB; AB → aABB; B → b; A → a;

б) Язык, порожденный этой грамматикой?

2.3.15. Какому классу по Хомскому принадлежит:

а) Грамматика с правилами:

S → ASB|BSA; A → a; B → b|ε; SB → ε;

б) Язык, порожденный этой грамматикой?

2.3.16. Какому классу по Хомскому принадлежит:

а) Грамматика с правилами:

S → AcBs; A → AcA|B; B → a|b;

б) Язык, порожденный этой грамматикой?

2.3.17. Сколько существует различных выводов цепочки baaaab, принадлежащей языку, порождаемому грамматикой с правилами:

S → bAb; A → AA|a

2.3.18. Построить праволинейные грамматики для языков, состоящих из:

а) идентификаторов произвольной длины, начинающихся с буквы;

б) идентификаторов, содержащих от 1 до 6 символов и начинающихся с букв I, J, K, L, M, N;

в) вещественных констант;

г) всех цепочек из нулей и единиц, имеющих:

- четное число нулей и четное число единиц;

- либо нечетное число нулей и нечетное число единиц.

2.3.19. Построить КС-грамматики для следующих языков:

а) $\{0^n1^n : n \geq 1$

б) $\{ww^R : w \in \{a, b\}^*\}$

в) Все цепочки из нулей и единиц с одинаковым числом тех и других

г) $\{\{a, b\}^* \setminus \{a^mb^na^mb^n\} : m, n \geq 1\};$

д) $\{\{a, b\}^* \setminus \{a^{2m}b^{3n}a^{2m}b^n\} : m, n \geq 1\};$

е) $\{\{a, b\}^* \setminus \{a^mb^na^m\} : m, n \geq 1\};$

ж) $\{\{a, b\}^* \setminus \{ww\} : w \in \{a, b^*\};$

з) $\{\{a, b\}^* \setminus \{a^nb^na^n\} : n \geq 1\};$

2.3.20. Определить КС-грамматики, которые порождали бы следующие языки:

1) все строки - элементы множества {0, 1}* такие, что в каждой из них непосредственно справа от каждого символа 0 стоит символ 1.

2) все строки - элементы множества {0, 1}* такие, что результаты чтения этих строк слева направо и справа налево совпадают;

3) все строки - элементы множества {0, 1}*, которые содержат символов 0 вдвое больше, чем символов 1;

4) все строки - элементы множества {0, 1}*, которые имеют одинаковое число символов 0 и 1;

5) все строки - элементы множества {0, 1}*, которые имеют четное число символов 0 и нечетное число символов 1;

6) все строки - элементы множества {0, 1}*, в которых скобки расставлены правильно.

2.3.21. Построить КС-грамматики, порождающие языки:

а) $\{a^mb^nc^p | m + n + p \equiv 0 (\mathrm{mod}\ 2);\ m, n, p \geq 0\};$

б) $\{a^pb^qc^r | p + q > r;\ p, q, r \geq 0\};$

в) $\{x | x \in \{a, b\}^*,\ |x|_a = |x|_b\};$

г) $\{x | x \in \{a, b\}^*,\ |x|_a > |x|_b\};$

д) построить однозначную КС-грамматику (однозначность должна быть доказана) для языка $\{x | x \in \{a, b\}^*,\ |x|_a = |x|_b,$ и для $\forall\ u, v : x = uv;\ |u| \neq 0,\ |v| \neq 0$ выполнено $|u|_a > |u|_b\}$.

2.3.22. Построить КС-грамматику, порождающую язык

а) $\{a^ncb^n\} \cup \{b^nac^n\};\ n \geq 0$

б) $\{x | x \in \{a, b\}^* \setminus \varepsilon;\ x \neq yy^R\}$

2.3.23. Построить НС-грамматики для следующих языков:

а) $\{w \in \{a, b, c\}^*, |w|_a = |w|_b = |w|_c\}$ (Винегрет)

б) $\{w \in \{a, b, c\}^*, 3|w|_a = 5|w|_b = 7|w|_c\}$ (Винегрет 2)

в) $\{a^n p^n r^n\} : n \geq; 1\}$ (Три мушкетера)

г) $\{a^m b^n a^m b^n : m, n \geq 1\}$ (Две калоши)

д) $\{a^{2m} b^n a^m b^{5n} : m, n \geq 1\}$ (Калоши 2)

е) $\{a^m b^n c^k : m \geq n \geq k \geq 1\}$ (Горка)

ж) $\{a^m b^n c^k : 2m \geq 3n \geq k \geq 1\}$ (Горка 2)

з) $\{a^{3^n} \mid n \geq 1\}$ (Бог любит троицу)

и) $\{a^{5^n} b^n \mid n \geq 1\}$

к) $\{a^{n^2} : n \geq 1\}$ (Квадратные числа)

л) $\{a^{n^2-5n+1} : n \geq 5\}$

м) $\{a^n b^{n^2} : n \geq 1\}$ (Дама с собачкой)

н) $\{a^{n^2\ 3n\,|\,2} b^n : n \geq 1$

о) $\{a^n : n = 1, 2, 3, 5, 8, 13, ...\}$ (Числа Фиббоначи)

п) {an : n = 1, 3, 6, 10, 15, ...} (Треугольные числа, $a_n = n(n + 1)/2$)

р) $\{a^n : n = 1, 5, 12, 22, ...\}$ (Пятиугольные числа, $a_n = n + 3n(n - 1)/2$. Пятиугольное число может быть разбито на три треугольных + n точек)

с) $\{ww : w \in \{a, b\}^*\}$ (Два лебедя)

т) $\{a^{n^3} : n \geq 1\}$ (Кубические числа)

у) $\{f^{n^3-n^2+2n-1} t^{3n} : n \geq 1$

ф) $\{a^n : n = 1, 2, 6, 24, ... , k!\}$ (Факториал)

х) $\{01^2...0^{n-1}1^n0^{n-1}...1^20 | n \geq 1\}$ (Пирамида Хеопса)

ч) $\{01^2...0^{n-1}1^n1^n0^{n-1}...1^20 | n \geq 1\}$ (Пирамиды майя)

ш) $\{a^{3^n} b^{n^2} a^n \mid n \geq 1\}$

щ) $\{\{a\}^+ \backslash a^{n^2} : n \geq 1\}$ (Для студентов с исследовательской жилкой).

2.3.24. Построить КС-грамматики, порождающие языки

а) $\{xcy | x \neq y;\ \ x, y \in \{a, b\}^*\}$;

б) $\{a^i b^j c^k | i, j, k \geq 1\} \setminus \{a^n b^n c^n | n \geq 1\}$;

в) $\{a, b, c\}^* \setminus \{a^n b^n c^n | n \geq 0\}$.

2.3.25. Пусть G - грамматика с правилами:

S → CD C → aCA|bCB|ε AD → aD

BD → bD Aa → aA Ab → bA

Ba → aB Bb → bB D → ε

Показать, что L(G) = {xx|x ∈ {a, b}*}.

2.3.26. Построить грамматику, порождающую данный язык:

$\{a^n c b^n a^n c b^n \mid n > 0\}$:

2.3.27. Построить регулярную грамматику, порождающую цепочки в алфавите (a, b), в котором символ a не встречается два раза подряд.

2.3.28. Построить грамматику, порождающую сбалансированные относительно круглых скобок цепочки в алфавите $\{a, (,), \perp\}$. Сбалансированную цепочку α определим реккурентно: цепочка α сбалансирована, если:

а) α не содержит скобок,

б) $\alpha = (\alpha_1)$ или $\alpha = \alpha_1\alpha_2$, где α_1 и α_2 сбалансированы.

2.3.29 Показать, что наличие в КС-грамматике правил вида

а) A →AA|α б) A →Aα A|β в) A →αA|Aβ|γ, где α, β, γ ∈ $(V_N \cup V_T)^*$; A ∈ V_N, делает ее неоднозначной. Можно ли преобразовать эти правила таким образом, чтобы полученная эквивалентная грамматика была однозначной?

2.3.30. Показать, что грамматика G неоднозначна.

G : S → abC|aB B → bc; bC → bc

2.3.31. Дана КС-грамматика G = (VT, VN, P, S). Предложить алгоритм построения множества

X = {A ∈ V_N|A ε}.

2.3.32. Для произвольной КС-грамматики G предложить алгоритм, определяющий, пуст ли язык L(G).

2.3.33. Одинаковые ли языки порождают грамматики из а), б), в)?

а) S → aAb A → BB B → ab|A|ε;

б) S → aAb A → AaAb|ε;

в) S → aB B → aBB|b.

2.3.34. Эквивалентны ли грамматики с правилами

S → AB; B → Bb|A; A → Aa|B; C → c.

и

S → ε.

2.3.35. Эквивалентны ли грамматики с правилами

A → AB; B → bC; A → aAc|Sa; C → c|Ca.
и
S → As|Bc; B → Ac|cS; A → Bd; C → c.

Лексический анализ

Регулярные множества и выражения

3.1.1. Показать, что множества, соответствующие двум данным регулярным выражениям, совпадают,

1) (a*b)c и a*(bc); 2) a*b и b + aa*b;
3) b(b + ab)*a и b(b*ab)*b*a; 4) b(ab + b)* и bb*a(bb*a)*:

3.1.2. Заменить каждое из следующих выражений эквивалентным, в котором не используются знак "+":

1) (a + b)*;
2) (a + bb + ba)*;
3) (a + (bb + ab)*)*:

3.1.3. Найти регулярные выражения, обозначающие языки, все слова которых - элементы множества $\{0, 1\}^*$:

1) оканчивающиеся на 011, 101, 110;
2) начинающиеся с 110, 101 или 011;
3) у которых каждый третий символ есть 0 или каждым второй - 1;
4) не содержащие ни одной из подстрок 011 и 101;
5) содержащие каждую из подстрок 011 и 101;
6) начинающиеся с 011 и оканчивающиеся на 110 или 101;
7) начинающиеся с 011 или 110 и оканчивающиеся на 101;
8) начинающиеся с 011 и содержащие вхождения подстроки 110;
9) $\{01^n|n > 1\}$;
10) $\{01^n0|n > 0\}$;
11) $\{0^m1^n|n, m > 1\}$;
12) $\{\alpha \in \{0, 1\}^* : |\alpha|=3$ - целое неотрицательное число$\}$;
13) $\{\alpha a|\alpha \in \{0, 1\}^+$, a $\in \{0, 1\}$, a входит в $\alpha\}$;
14) $\{(010)^n|n > 0\}$;
15) $\{0^m|m > 2\}$ или $\{1^n|n > 0\}$;
16) $\{(01)^m(10)^n|m \geq 0, n \geq 0\}$;
17) содержащее четное число символов 0 и нечетное число символов 1;
18) содержащее четное число символов 0 или четное число символов 1.

3.1.4. Является ли язык, состоящий из всех цепочек из 0 и 1, не содержащих подцепочки 010, регулярным?

3.1.5. Является ли язык, состоящий из всех цепочек из 0 и 1, содержащих четное число 0 и нечетное - 1, регулярным?

3.1.6. Является ли язык, состоящий из всех цепочек четной длины в алфавите{fa, b, c}, регулярным?

3.1.7. Регулярен ли

а) язык формул вида A*(B), где A, B $\in \{a, b\}^+$?

б) язык формул вида $(A_1.A_2)$, где для i = 1, $\in A_i$ есть либо слово в алфавите {a, b}, либо, в свою очередь, формула?

в) язык формул вида (A + B), где A, B $\in$ {a, b}$^+$?

г) язык формул вида $(A_1)A_2$, где для i = 1, $\in$ A_i есть либо слово в алфавите {a, b}, либо, в свою очередь, формула?

3.1.8. Определить язык, состоящий из всех идентификаторов, с помощью:

а) регулярного выражения;
б) леволинейной грамматики;
в) конечного автомата;
г) праволинейной грамматики.

3.1.9. Будет ли регулярным язык L = {x $\in${a, b}* : $|x|_a$ четно и $|x|_b$ нечетно}?

3.1.10. Построить праволинейную грамматику, порождающую язык L всех слов в алфавите {0, 1}, содержащих четное число единиц и нечетное число нулей. Будет ли она однозначной?

3.1.11. Построить регулярное выражение для языка L^R, где L - язык всех слов в алфавите {0, 1}, содержащих четное число единиц и нечетное число нулей.

Конечные автоматы

3.2.1. Какой язык допускается конечным автоматом M = ({q_0}, {a, b}, $\varnothing$, q_0, {q_0})?

3.2.2. Построить недетерминированный конечный автомат, допускающий цепочки в алфавите {1, 2}, у которых последний символ цепочки уже появлялся в ней раньше. Построить эквивалентный детерминированный конечный автомат. Построить аналогичные конечные автоматы в алфавите {1, 2, 3}.

3.2.3. Построить конечный автомат, допускающий язык {xy} $\cup${yx}, где x $\in${a}* \ ε y $\in${b}* \ ε.

3.2.4. Построить детерминированный конечный автомат, допускающий язык L всех слов в алфавите {0, 1}, содержащих четное число единиц и нечетное число нулей;

Алгоритмы построения конечных автоматов

3.3.1. Для регулярного выражения над алфавитом T = {a, b} построить эквивалентный детерминированный конечный автомат:

а) b(ba|b)*|b	б) (ab|b)*ba|ab
в) (a|b)*ba(a|b)	г) (a|b)*ab(a|b)*
д) a(ab|b)*|ba	е) (ba|b)*ab|ba
ж) (a*b)*ab*a	з) (a|b)*(a|b)(a|b(a|b)

Лексический анализ

Регулярные множества и их представления

3.4.1. Будет ли регулярным язык L = {x $\in${a, b}|x не содержит подцепочки aba}?

3.4.2. Возможно ли построить регулярную грамматику, порождающую язык, включающий в себя все непустые цепочки из 0 и 1, не содержащие трех 1 подряд?

Алгебраические свойства регулярных множеств. Лемма о разрастании.

3.5.1. Будут ли регулярными следующие языки в алфавите {a}:

а) L_1 = {{a^{2n+5}} $\cup${a^{7n+4}}, n = 0, 1, ...};

б) $L_2 = \{\{a^{2n+5}\} \cap \{a^{7n+4}\}, n = 0, 1, ...\}$;

в) $L_3 = \{\{a^{4n+5}\}, n = 0, 1, ..., n \neq 5(\text{mod}11)\}$;

д) $L_4 = \{a^{n^2}, n = 0, 1, ...\}$.

3.5.2. Будут ли регулярными следующие языки в алфавите Σ = {a, b}:

а) язык L_1 из всех слов Σ*, содержащих подслова a?b;
б) язык L_2 из всех слов Σ*, не содержащих двух b подряд;
в) язык L_3 из всех слов Σ*, не принадлежащих L_1 или L_2;
г) $L_4 = \{\{a^{2n+5}b^{7n+4}\}, n = 0, 1, ...\}$?

3.5.3. Задается ли язык $\{a^n b^m | n \geq m \geq 1\}$ регулярным выражением?

3.5.4. Является ли грамматика с правилами:

S ⟶ aA|bB|C; B ⟶ bB|b|ε;
A ⟶ aA|a|ε; C ⟶ cSC:

праволинейной грамматикой?

Синтаксический анализ

КС-грамматики и МП-автоматы

4.1.1. Пусть G - грамматика с правилами:

S ⟶ SbS|ScS|a

Найти 2 различных дерева вывода для цепочки abaca.

4.1.2. Дана однозначная КС-грамматика G = (N, T, P, S) и цепочка $w \in L(G)$. Количество элементов во множествах N, T, P равно n_1, n_2, n_3 соответственно, а |w| = l. Найти нижнюю и верхнюю границу для числа деревьев разбора w в G.

4.1.3. Являются ли однозначными следующие грамматики?

а) S ⟶ a|C; C ⟶ AB; A ⟶ aA|Ba|a; B ⟶ aB;
б) S ⟶ BA; A ⟶ Aa|bA|ε; B ⟶ Bb|aB|b;
в) S ⟶ b|C; C ⟶ aC|AC; A ⟶ aA|Aa|a;
г) S ⟶ AB; A ⟶ aA|bA|a; B ⟶ Ba|Bb|ε;
д) S ⟶ A|B; A ⟶ AA|a; B ⟶ aB|b|C; C ⟶ cC;
е) S ⟶ aA|bB; A ⟶ aA|a|b; B ⟶ bB|b|ε;
ж) S ⟶ aAc|bS; A ⟶ aA|Aa|ε;
з) S ⟶ aA|b; A ⟶ abA|abAcb; B ⟶ c;
и) S ⟶ aB|cA; A ⟶ BaA|a; B ⟶ A|a;
к) S ⟶ ABS|ε; A ⟶ abA|a; B ⟶ Ba|Bab|ε.

4.1.4. Является ли однозначной грамматика с правилами:

а) S ⟶ A|B; B ⟶ aB|b|C; A ⟶ AA|a; C ⟶ cC;
б) S ⟶ aAc|bS; A ⟶ aA|Aa|c;

в) $S \to aA|b;\ A \to abA|abAcb;\ B \to c;$

г) $S \to aB|cA;\ A \to BaA|a;\ B \to A|b;$

д) $S \to a|C;\ C \to AB;\ A \to aA|Ba|a;\ B \to aB;$

е) $S \to BA;\ A \to Aa|bA|\varepsilon;\ B \to Bb|aB|b;$

ж) $S \to b|C;\ C \to aC|AC;\ A \to aA|Aa|a;$

з) $S \to AB;\ A \to aA|bA|a;\ B \to Ba|Bb|\varepsilon.$

4.1.5. Пусть G_1 - грамматика, имеющая продукции:

$S \to bA|ab;\ A \to a|aS|bAA;\ B \to b|bS|aBB;$

а G_2 - грамматика, определяемая продукциями:

$S \to aB|aBS|bAS|bA;\ A \to bAA|a;\ B \to bBB|b.$

Показать, что

1) G_1 - неоднозначная грамматика;
2) G_2 - однозначная грамматика;
3) $L(G_1) = L(G_2)$:

4.1.6. Какой язык допускается автоматом с магазинной памятью

$P = (Xq_0\}, \{a, b\}, \{z_0\}, \varnothing, q_0, z_0, \{q_0\})$?

4.1.7. Построить МП-автоматы, определяющие языки

а) $\{ww^R : w \in \{a, b\}^*\};$

б) язык всех цепочек из нулей и единиц с одинаковым числом тех и других

в) $\{\{a, b\}^* \setminus \{a^m b^n a^m b^n\} : m, n \geq 1\};$

г) $\{\{a, b\}^* \setminus \{a^m b^n a^m\} : m, n \geq 1\};$

д) $\{\{a, b\}^* \setminus \{ww\} : w \in \{a, b]^*\}:$

4.1.8. Построить автомат с магазинной памятью, допускающий язык:

а) $(\{a^n b^n c^m | n, m \geq 1\}) \cup (\{a^m b^n c^n | n, m \geq 1\});$

б) $\{a^n c^k b^n | k, n \geq 1\};$

в) $\{a^m b^n c^p | m + n + p \equiv 0 (mod 2), m, n, p \geq 0\};$

г) $\{a^p b^q c^r | p + q > r;\ \ p, q, r \geq 0\};$

д) $\{x | x \in \{a, b\}^*, |x|_a = |x|_b\};$

е) $\{x | x \in \{a, b\}^*, |x|_a \geq |x|_b\};$

ж) $\{x | x \in \{a, b\}^*; |x|_a = |x|_b$, и для $\forall u, v : x = uv; |u| \neq 0, |v| \neq 0$ выполнено $|u|_a > |u|_b\}.$

4.1.9. Пусть A - магазинный автомат. Построить магазинный автомат B, допускающий все префиксы языка

$L(A)$, то есть язык

$L(B) = \{x | xy \in L(A)\}:$

4.1.10. Построить детерминированные МП-автоматы, определяющие языки:

а) $\{wcw^R : w \in \{a, b\}^*\}$;

б) $\{0^n1^n : n \geq 1\}$

в) $\{xcx^Rycy^R | x, y \in \{a, b\}^*\}$.

4.1.11. Является ли язык $L = \{xcx^R | x \in (a^*b^*)^*\}$ детерминированным? Обосновать ответ с помощью магазинного автомата, допускающего язык L.

4.1.12. Является ли детерминированным следующий язык:

а) $L = \{x^Rcx | x \in (a^*b^*)^*\}$;

б) $L = \{xcx^R | x \in (b^*a^*)^*\}$;

в) $L = \{xcx^R | x \in b^*(a^*)^*\}$.

4.1.13. Доказать, что для любой КС-грамматики G' существует эквивалентная ей КС грамматика G, имеющая лишь правила вида

$A \to BC$; $A \to a$, где $A, B, C \in V_N$; $a \in V_T$.

4.1.14. Доказать, что если L_1 - КС-язык, то язык L, состоящий из всех слов L_1 четной длины - КС-язык, то есть

$L = \{X | X \in L_1; |X| = 2K; K = 0, 1, \ldots, \}$ - КС-язык.

4.1.15. Доказать, что для КС-грамматики G существует неукорачивающая КС-грамматика G', порождающая язык

$L(G') = L(G) \setminus \{\varepsilon\}$.

4.1.16. Привести алгоритм, позволяющий узнать, принадлежит ли данное слово данному КС-языку и доказать его правильность.

4.1.17. КС-грамматика называется левооднозначной, если каждое слово порождаемого ею языка имеет единственный левый вывод. Аналогично определяется правооднозначная грамматика. Построить пример левооднозначной, но не правооднозначной КС-грамматики.

С.4.2. Алгебраические свойства КС-языков. Лемма о разрастании.

4.2.1. Пусть L_1, L_2 - КС-языки. Докажите:

1) $L_1 \cup L_2$ - КС-язык;
2) L_1L_2 - КС-язык.

4.2.2. Пусть L - КС-язык. Докажите:

1) L^* - КС-язык;
2) L^R - КС-язык.

4.2.3. Доказать, что не существует КС-грамматик, порождающих языки

а) $\{a^nb^nc^n\} : n \geq 1\}$; б) $\{ww : w \in \{a, b\}^*\}$;

в) $\{a^{n^2} : n \geq 1\}$; г) $\{a^{n^3} : n \geq 1\}$.

4.2.4. Выяснить, какие из приведенных ниже языков не являются КС-языками:

1) $\{a^i b^j c^k \mid 0 \le i < j < k\}$;

2) $\{a^i b^j c^k \mid 0 \le i = j = k\}$;

3) $\{a^i b^j c^k \mid 0 \le i = j,\ k \ge 0,\ i \ne k\}$;

4) $\{a^i b^j c^k \mid 0 \le i = j,\ k \ge 0\}$:

4.2.5. Показать, что язык $\{a^n b^n c^n \mid n \ge 1\}$ не является КС-языком.

4.2.6. Является ли язык $\{a^n b^m a^n b^m \mid n \ge 1,\ m \ge 1\}$ КС-языком?

4.2.7. Является ли язык $\{a^n b^m b^n a^m \mid n \ge 1,\ m \ge 1\}$ КС-языком?

4.2.8. Является ли язык $\{a^p \mid p$ - простое число$\}$ КС- языком?

4.2.9. Является ли язык $\{a^n b^{n^2} \mid n \in N\}$ КС-языком?

4.2.10. Определить, замкнуто ли множество КС-языков относительно дополнения?

4.2.11. Замкнуто ли множество КС-языков относительно обращения? (Иначе говоря, верно ли, что если L - КС-язык, то L^R - тоже КС-язык).

Преобразования КС-грамматик

4.3.1. Указать множество бесполезных символов для грамматики:

$S \to A \mid B;\ B \to aB \mid b \mid C;\ A \to AA \mid a;\ C \to cC$:

4.3.2. Указать множество бесполезных символов в грамматике G = ({S, A, B, C}, {a, b, c}, P, S), где P состоит из

$S \to aSb \mid Abb \mid \varepsilon \quad B \to AB$

$A \to aBCb \mid bAb \quad C \to a \mid c$.

4.3.3. Указать множество бесполезных символов в грамматике G = ({S, A, B, C}, {a, b, c}, P, S), где P состоит из

$S \to A \mid B \qquad A \to aB \mid bS \mid b$

$B \to AB \mid Ba \qquad C \to AS \mid b$.

4.3.4. Указать множество бесполезных символов в грамматике G=({S, A, B, C, D}, {a, b, c}, P, S}, где P состоит из

$S \to aBb \mid aCb \quad A \to Dc \mid cA$

$B \to aS \mid b \qquad C \to AB \mid aD$

$D \to AB \mid cDa$.

4.3.5. Указать множество бесполезных символов в грамматике G = ({S, A, B, C}, {0, 1, 2}, P, S), где P состоит из

$S \to SS \mid A \quad A \to 0A1 \mid C \mid 0$

$B \to 0C \mid 1 \quad C \to BC \mid CS$.

4.3.6. Являются ли следующие грамматики приведенными? Указать для каждой грамматики множества недостижимых, бесплодных и бесполезных символов:

а) S → a|C
C → AB
A → aA|Ba|a
B → aB;

б) S → BA
A → Aa|bA|ε
B → Bb|aB|b;

в) S → b|C
C → aC|AC
A → aA|Aa|a;

г) S → AB
A → aA|bA|a
B → Ba|Bb|ε;

д) S → A|B
A → AA|a
B → aB|b|C
C → cC;

е) S → aA|bB
A → aA|a|b
B → bB|b|ε;

ж) S → aAc|bS
A → aA|Aa|ε;
B → c;

з) S → aA|b
A → abA|abAcb

и) S → aB|cA
A → BaA|a
B → A|a;

к) S → ABS|ε
A → abA|a
B → Ba|Bab|ε.

4.3.7. Построить приведенные грамматики, эквивалентные следующим грамматикам:

а) S → A|B A → C|D B → D|E
C → S|a|ε D → S|b E → S|c|ε;
б) S → AB A → Aa|bB B → a|Sb.

4.3.8. Построить ε-свободные КС-грамматики, эквивалентные следующим грамматикам:

1) S → AB
A → C|ab
C → c|ε
B → aAa;

2) S → ABC
A → BB|ε
B → CC|ε
C → AA|b;

3) S → aSbS
S → bSaS|ε;

4) S → AB
A → SA|BB|bB
B → b|aA|ε.

4.3.9. Доказать, что для каждой КС-грамматики существует эквивалентная ей приведенная КС-грамматика.

4.3.10. Привести алгоритм построения множества достижимых символов и доказать его правильность

4.3.11. Доказать, что для каждой КС-грамматики существует эквивалентная ей КС-грамматика, не являющаяся леворекурсивной

Предсказывающий разбор сверху- вниз

4.4.1. Построить множества FIRST и FOLLOW для каждого нетерминала грамматики

а) S → aAB|B
б) S → aAB|BA

A → aA|a
B → BS|A|b;

A → BBB|a
B → AS|b;

в) S → S + T
S → T
T → a
T → S[S];

г) S → ABC
A → BB|ε
B → CC|a
C → AA|b;

д) S → aB|bA
A → aS|bAA|a
B → bS|aBB|b;

е) S → Ba|Ab
A → Sa|AAb|a
B → Sb|BBa|b;

ж) S → (SbS)
S → (T)
S → a
T → TS
T → S;
S → B;

з) B → begin D; S end
B → s
D → D; d
D → d
S → S; B

и) A → aACd|b
C → c|ε.

4.4.2. Является ли следующая грамматика LL(1)? Использовать критерий LL(1).

S → aAb; A → 0; A → aaA.

4.4.3. Для грамматики написать эквивалентную LL(1)- грамматику

а) S → aS|a;
б) S → ba|A
в) S → aaS|abA
г) S → baaA|babA
д) S → abaA|abbA
е) S → ab|baA

A → a|Aab|Ab;
A → ε|Aa|Ab;
A → ε|Aa|Ab;
A → ε|Aab|Ab.

4.4.4. Для следующих грамматик определить, являются ли они LL(k) грамматиками и найти точное значение k. Для LL(1)-грамматик построить детерминированный левый анализатор:

а) S → aAS|b
б) S → A|B
в) S → ε|abA
г) S → aS|a;
д) S → aAaa|bAba
е) Sa|b;
ж) S → TE';
T' → *FT'|ε

A → a|bSA;
A → aAb|0
A → Saa|b;
A → b|ε;
E' → +TE'|ε
F → (S)|a.

B → aBbb|1;
T → FT'

4.4.5. Определить, являются ли следующие грамматики LL(k)-грамматиками, и указать точное значение k:

а) S → Ab A → Aa|a;

б) S → Ab A → aA|a;

в) S → aAb A → BB B → ab|A|ε;

г) S → aAb A → AaAb|ε;

д) S → aB B → aBB|b.

4.4.6. Преобразовать грамматику к LL(1)-виду и построить для нее LL(1)-таблицу

а) S → Ab A → aA|a;

б) S → aB B → aBB|b:

4.4.7. Сколько тактов сделает LL(1)-анализатор для грамматики G с правилами:

S → aAB A → bC B → SS|ε C → A|ε

при разборе цепочки x = ab; ab, b ?

4.4.8. Является ли грамматика S →Sa|b LL(2)- грамматикой?

4.4.9. Является ли язык, состоящий из всех целых чисел без знака и без незначащих нулей, LL(1)-языком?

4.4.10. Является ли язык, состоящий из всех цепочек из 0 и 1, не содержащих подцепочки 010, LL(1)-языком?

4.4.11. Является ли язык, состоящий из всех непустых цепочек из 0 и 1, не содержащих трех 1 подряд, LL(1)-языком?

4.4.12. Существует ли контекстно-свободная грамматика, LL(1)-таблица для которой не содержит элементов "ошибка" ?

4.4.13. Сформулируйте необходимые и достаточные условия того, что КС-грамматика есть LL(1)-грамматика. Докажите необходимость и достаточность.

Разбор снизу-вверх типа сдвиг- свертка

4.5.1. Построить все состояния для LR(0)-анализа грамматики G:

S → aAb; A → ε; A → aaA

Будет ли G LR(0)-грамматикой? A LR(1)?

4.5.2. Является ли грамматика с правилами:

S → A|B; B → aB|b|C; A → AA|a; C → cC

LR(0)-грамматикой?

4.5.3. Сколько множеств LR(0)-ситуаций в канонической системе LR(0)-ситуаций грамматики G с правилами

а) S → aA|aB A → bA|c B → bB|d;

б) S → A0|F1 A → S0|B1 B → A1|F0 F → B0|S1;

в) E → (L)|a L → EL|E.

4.5.4. Сколько LR(0)-таблиц имеет грамматика с правилами:

S → Aa|Bb; B → b; A → ab.

4.5.5. Построить все состояния LR(1)-анализа для грамматики:

S → aAb; A → ε|aaA.

4.5.6. Сколько множеств LR(1)-ситуаций в канонической системе LR(1)-ситуаций грамматики G с правилами

а) S → aSb|ab;
б) S → aAc|b A → aSc|b.

4.5.7. Определить, является ли грамматика с приведенным набором правил LR(1)-грамматикой:

а) A → aAB|b B → b|ε;
б) S → SaS S → a;
в) S → Abb|Bba A → a B → a;
г) S → aL|a L → Lb|b.

4.5.8. Построить все состояния анализа (K = 1) для грамматики

$S \to S_1$; $S_1 \to S_1S_1$; $S_1 \to a$.

Будет ли эта грамматика LR(1)?

4.5.9. Построить все состояния LR(1) анализа для грамматики:

S → aBc; B → b; B → bBb:

Применив критерий LR(K), определить, будет ли это LR(1)- грамматика.

4.5.10. Выяснить, являются ли следующие грамматики LR(k)-грамматиками. Найти точное значение k и построить детерминированный правый анализатор:

а) S → SaSb|ε;
б) S → Sa|a;
в) S → C|d C → Ac|b D → aD|c;
г) S → Ab|Bc A → Aa|ε B → Ba|ε;
д) S → AB A → a B → CD|aE C → ab D → bb E → bba;
е) S → AB A → 0A1|ε B → 1B|1.

4.5.11. Является ли нижеприведенная грамматика LR(k), и если да, то определить минимальное k.

а) S → aAc A → aSc S → b A → b;
б) S → S1 S1 → S1S1 S1 → a;
в) S → aBc B → b B → bBb;
г) S → aAc S → b A → aSc A → b;
д) S → aAb A → 0 A → aaA;

е) S → aAb A → ε A → aaA.

4.5.12. Являются ли следующие грамматики LR(k)- грамматиками? Указать точное значение k и построить соответствующий детерминированный правый анализатор.

а) S → Ab A → Aa|a;
б) S → Ab A → aA|a;
в) S → aAb A → BB B → ab|A|ε;
г) S → aAb A → AaAb|ε;
д) S → aB B → aBB|b:

4.5.13. Для грамматики

S →Ab|Bc A →Aa|ε B →Ba|ε

написать эквивалентную LR(0)-грамматику.

4.5.14. Сколько сверток и переносов сделает LR(1)- анализатор для грамматики G = ({S, A}, {a}, P, S) с правилами S →A A →Aa|a при анализе цепочки a100?

4.5.15. Сколько SLR(1)-таблиц имеет грамматика с правилами:

S → Aaa|Bb|C B → aa A → aa C → cAc|cBd.

4.5.16. Сколько тактов сделает LALR(1)-анализатор для грамматики с правилами:

S → A|BC B → a A → a; C → AAAS
при разборе цепочки aaaaa?

4.5.17. Выписать цепочку минимальной длины, на которой видны отличия LARL(1) и LR(1)-анализаторов для грамматики с правилами:

S → Aa|Bb|C B → aa A → aa C → cAc|cBd.

4.5.18. Пусть G = (N, T, P, S) - LR(1)-грамматика, w $\in T^*$. В каких случаях (в зависимости от G и w) LR(1)-анализатор при анализе цепочки w не сделает ни одного сдвига?

4.5.19. Пусть G = (N, T, P, S) - LR(1)-грамматика; w $\notin$L(G); |w| = n: Пусть k - число сдвигов, делаемых LR(1)- анализатором при анализе цепочки w. Привести нижнюю и верхнюю оценку для числа k.

4.5.20. Пусть G = (N, T, P, S) - LR(1)-грамматика, |P| = m $\geq$1; w $\in$L(G), |w| = n. Пусть k - число сверток, делаемых LR(1)-анализатором при анализе цепочки w. Привести нижнюю оценку для числа k.

4.5.21. Пусть G = (N, T, P, S) - LR(1)-грамматика, |P| = m $\geq$1; w $\notin$L(G), |w| = n. Пусть k - число сверток, делаемых LR(1)-анализатором при анализе цепочки w. Привести нижнюю оценку для числа k.

4.5.22. Существует ли LR(1)-грамматика, для которой функция действий LR(1)-таблицы не содержит элементов "ошибка" ?

4.5.23. Дана КС-грамматика G = (N, T, P, S). Найти верхнюю оценку числа LR(1)-ситуаций для G.

4.5.24. Дана LR(1)-грамматика без ε-правил G и цепочка w $\in$L(G). В дереве разбора w - n_1 листьев и n_2 внутренних вершин. Сколько сдвигов и сверток сделает LR(1)-анализатор для G при анализе цепочки w?

Элементы теории перевода

Атрибутные грамматики

5.3.1. Дополнить грамматику S →0S11; S →1S00; S →ε до атрибутной так, чтобы вычислялась максимальная длина непрерывной последовательности единиц в порожденном слове.

5.3.2. Дополнить грамматику S →AA; A →0A; A →1A; A →ε до атрибутной так, чтобы вычислялась максимальная длина непрерывной последовательности из 1 в порожденном слове.

5.3.3. Дополнить грамматику S →AA; A →A0; A →A1; A →ε до атрибутной так, чтобы вычислялось число сочетаний 01 в порожденном слове.

5.3.4. В грамматике [целое] →dC; C →dC|ε терминал d имеет атрибут 0 или 1. Определить атрибуты так, чтобы нетерминал [целое] имел атрибут, равный восьмеричному значению выводимого числа.

5.3.5. Построить атрибутные грамматики для следующих переводов:

а) $\{(x, x) \mid x \in \{a, b\}^*\}$; б) $\{(x, x^R) \mid x \in \{a, b\}^*\}$;

в) $\{(x, xx) \mid x \in \{a, b\}^*\}$; г) $\{(a^n b^n; a^n b^n c^n) \mid n \geq 1\}$.

5.3.6. Привести пример атрибутной грамматики с некорректно заданными семантическими правилами

5.3.7. Привести пример атрибутной грамматики, вычисление атрибутов для которой нельзя выполнить параллельно с LL(1)-анализом.

5.3.8. Привести пример атрибутной грамматики, вычисление атрибутов для которой нельзя выполнить параллельно с LR(1)-анализом.

Генерация кода

Трансляция арифметических выражений

9.1.1. Для следующих арифметических выражений с помощью алгоритма Сети-Ульмана сгенерировать программу и изобразить атрибутированное дерево:

```
а) A*B + C*(D + E)*F;          б) A*(B + C)*(D + E)*F;
в) A + B + C*D + E*F;          г) A + B*C*D*E + F;
д) A + B*(C*D + E*F).
```

Трансляция логических выражений

9.2.1. Для следующих логических выражений сгенерировать код на командах перехода и изобразить атрибутированное дерево

```
а) A and not (B or C) or (D and E);
б) A and B and C or not (D or E);
в) A and (B or not (C and D) and E);
г) not (A and B or C or D) and E;
д) A and B or C or D and not E.
```

Генерация оптимального кода методами синтаксического анализа

9.3.1. Для следующих операторов присваивания сгенерировать оптимальный код методом сопоставления образцов:

```
а) a = b[i] + j;          б) a = b[i+5];          в) a = b[i] + c[2];
г) a = b[i+2+j];          д) a = b[2+c[1]];       е) a = b[i+j];
ж) a = b[i+2] + 3;        з) a =j+ b[i+3];        и) a = b[i+j+1];
```

к) `a = b[i+j] + 1.`