МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное

учреждение высшего образования

«Чувашский государственный университет имени И.Н. Ульянова»

Л.А. ПАВЛОВ

ЛЕКСИЧЕСКИЙ И СИНТАКСИЧЕСКИЙ АНАЛИЗ

Учебное пособие

Чебоксары

2022

УДК 004.4’422(075.8)

ББК З973.2р30(2) – 25

П12

*Рецензенты:*

кафедра информационных технологий, электроэнергетики и систем управления Чебоксарского института (филиала) ФГАОУ ВО «Московский политехнический университет» (зав. кафедрой канд. пед. наук *П.В. Матижев*);

главный инженер отдела сопровождения почтовых систем Департамента информационных технологий АО «Россельхозбанк», канд. техн. наук *М.Ю. Харитонов*

**Павлов Л.А.**

**П12**  Лексический и синтаксический анализ: учеб. пособие / Л.А. Павлов. –Чебоксары: Изд-во Чуваш. ун-та, 2021. 152 с.

ISBN ???-?-????-????-?

Даны основные определения и понятия теории формальных языков и грамматик, классификация грамматик, методы их эквивалентных преобразований, рассмотрены методы лексического анализа, нисходящего и восходящего синтаксического анализа.

Для студентов факультета информатики и вычислительной техники по направлению подготовки бакалавров 09.03.01 «Информатика и вычислительная техника», а также других направлений и профилей, связанных с разработкой программного обеспечения.

Утверждено Учебно-методическим советом университета

Ответственный редактор канд. техн. наук, доцент А.А. Андреева

ISBN ???-?-????-????-? УДК 004.4’.422(075.8)

ББК З973.2р30(2) – 25

© Издательство

Чувашского университета, 2022

© Павлов Л.А., 2022

**ОГЛАВЛЕНИЕ**

[ВВЕДЕНИЕ 5](#_Toc85196464)

[Глава 1. ЭЛЕМЕНТЫ ТЕОРИИ ФОРМАЛЬНЫХ ЯЗЫКОВ И ГРАММАТИК 7](#_Toc85196465)

[1.1. Основные понятия и определения 7](#_Toc85196466)

[1.2. Метаязыки 8](#_Toc85196467)

[1.3. Формальные грамматики 12](#_Toc85196468)

[1.4. Классификация грамматик и языков 14](#_Toc85196469)

[1.5. Контекстно-свободные грамматики 17](#_Toc85196470)

[1.6. Эквивалентные преобразования грамматик 22](#_Toc85196471)

[1.6.1. Удаление бесполезных символов 22](#_Toc85196472)

[1.6.2. Замена вхождений 25](#_Toc85196473)

[1.6.3. Устранение леворекурсивных продукций 26](#_Toc85196474)

[1.6.4. Устранение леворекурсивного цикла 27](#_Toc85196475)

[1.6.5. Факторизация 29](#_Toc85196476)

[1.6.6. Удаление ε-продукций 29](#_Toc85196477)

[1.6.7. Удаление цепных продукций 32](#_Toc85196478)

[1.7. Канонические формы КС-грамматик 34](#_Toc85196479)

[1.7.1. Нормальная форма Хомского 34](#_Toc85196480)

[1.7.2. Нормальная форма Грейбах 36](#_Toc85196481)

[1.8. Свойство самовложения КС-грамматик 39](#_Toc85196482)

[1.9. Лемма Огдена и лемма о разрастании для КС-языков 40](#_Toc85196483)

[Упражнения 41](#_Toc85196484)

[Глава 2. ЛЕКСИЧЕСКИЙ АНАЛИЗ 43](#_Toc85196485)

[2.1. Задачи лексического анализа 43](#_Toc85196486)

[2.2. Лексические классы 44](#_Toc85196487)

[2.3. Таблица символов 46](#_Toc85196488)

[2.4. Распознавание токенов 47](#_Toc85196489)

[2.5. Конечный автомат 50](#_Toc85196490)

[2.6. Регулярные грамматики 54](#_Toc85196491)

[2.7. Регулярные выражения 60](#_Toc85196492)

[2.8. Конечные автоматы с ε-переходами 65](#_Toc85196493)

[2.9. Минимизация конечного автомата 68](#_Toc85196494)

[Упражнения 72](#_Toc85196495)

[Глава 3. НИСХОДЯЩИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ 73](#_Toc85196496)

[3.1. Автомат с магазинной памятью 73](#_Toc85196497)

[3.2. МП-автоматы и КС-грамматики 76](#_Toc85196498)

[3.3. *LL*(*k*)-грамматики 78](#_Toc85196499)

[3.4. *LL*(1)-грамматики 80](#_Toc85196500)

[3.4.1. Разделенные грамматики 80](#_Toc85196501)

[3.4.2. Слаборазделенные грамматики 81](#_Toc85196502)

[3.4.3. *LL*(1)-грамматики 82](#_Toc85196503)

[3.4.4. Основные приемы преобразования КС-грамматик в *LL*(1)-форму 86](#_Toc85196504)

[3.5. Метод рекурсивного спуска 90](#_Toc85196505)

[3.6. Табличные методы нисходящего разбора 94](#_Toc85196506)

[3.6.1. Таблица переходов МП-автомата 95](#_Toc85196507)

[3.6.2. Специальные *LL*(1)-таблицы разбора 99](#_Toc85196508)

[Упражнения 105](#_Toc85196509)

[Глава 4. ВОСХОДЯЩИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ 107](#_Toc85196510)

[4.1. Построение дерева разбора 107](#_Toc85196511)

[4.2. Грамматики простого предшествования 110](#_Toc85196512)

[4.2.1. Отношения предшествования 111](#_Toc85196513)

[4.2.2. Вычисление отношений предшествования 112](#_Toc85196514)

[4.2.3. Синтаксический анализ 115](#_Toc85196515)

[4.2.4. Функции предшествования 118](#_Toc85196516)

[4.3. Грамматики слабого предшествования 121](#_Toc85196517)

[4.4. Грамматики операторного предшествования 123](#_Toc85196518)

[4.5. *LR*(*k*)-грамматики 126](#_Toc85196519)

[4.6. *LR*-таблицы разбора 129](#_Toc85196520)

[4.6.1. *LR*(0)-грамматики 130](#_Toc85196521)

[4.6.2. *SLR*(1)-грамматики 135](#_Toc85196522)

[4.6.3. *LALR*(1)-грамматики 138](#_Toc85196523)

[4.6.4. *LR*(1)-грамматики 142](#_Toc85196524)

[4.7. Синтаксический анализ 145](#_Toc85196525)

[4.8. Сравнение с *LL*-методом разбора 148](#_Toc85196526)

[Упражнения 149](#_Toc85196527)

[ЗАКЛЮЧЕНИЕ 150](#_Toc85196528)

[Список рекомендуемой литературы 151](#_Toc85196529)

ВВЕДЕНИЕ

Методы лексического и синтаксического анализа рассматриваются в рамках дисциплины «Теория языков программирования и методы трансляции», изучаемой обучающимися по направлению подготовки бакалавров 09.03.01 «Информатика и вычислительная техника» (профиль «Программное обеспечение средств вычислительной техники и автоматизированных систем»). Целью изучения этих методов является получение теоретических знаний и практических навыков реализации таких фаз компиляции, как лексический и синтаксический анализ.

Данный раздел изучается в 6 семестре и базируется на знаниях, полученных в процессе изучения таких дисциплин, как «Информатика», «Программирование», «Математическая логика и теория алгоритмов», «Дискретная математика», «Структуры и алгоритмы обработки данных» и др.

Знания, умения и навыки, полученные студентом в ходе изучения разделов дисциплины «Теория языков программирования и методы трансляции», вполне достаточны для реализации трансляции простых языков программирования. Полученные знания будут полезны также для понимания особенностей разработки языков программирования, получения навыков работы со сложными структурами данных и алгоритмами.

В первой главе рассматриваются основные элементы теории формальных языков и грамматик как теоретической базы формальных методов проектирования компиляторов: методы формального описания синтаксиса языков программирования, понятие формальной грамматики, классификация языков и грамматик (иерархия Хомского), контекстно-свободные грамматики (КС-грамматики) и языки (КС-языки), эквивалентные преобразования КС-грамматик, канонические формы КС-грамматик (нормальные формы Хомского и Грейбах).

Вторая глава посвящена начальной фазе компиляции – лексическому анализу. В этой главе формулируются основные задачи лексического анализа, определяется понятие лексического класса, дается определение конечного автомата как формального инструмента для распознавания лексических классов. Рассматриваются регулярные грамматики и регулярные выражения как способы формального описания лексических классов, методы синтеза распознающих конечных автоматов по регулярным грамматикам и регулярным выражениям, их детерминизации и минимизации.

В третьей главе изложены основные методы нисходящего синтаксического анализа. Дается определение автомата с магазинной памятью (МП-автомат), являющегося распознавателем КС-языков, рассматривается связь между КС-грамматиками и МП-автоматами, дается определение *LL*(*k*)-грамматики. Рассматриваются *LL*(1)-грамматики как основа для реализации нисходящего синтаксического анализа, основные приемы преобразования КС-грамматик в *LL*(1)-форму. Методы нисходящего синтаксического анализа с использованием *LL*(1)-грамматик (методы рекурсивного спуска и табличные методы).

В четвертой главе рассматриваются основные методы восходящего синтаксического анализа: основанные на грамматиках предшествования (простое, слабое и операторное предшествование) и *LR*(*k*)-грамматиках. Подробно изложены вопросы построения таблиц разбора (*LR*-таблицы разбора) для различных подклассов *LR*(*k*)-грамматик (*LR*(0)-, *SLR*(1)-, *LALR*(1)- и *LR*(1)-грамматик) и их использования для синтаксического анализа.

Каждая глава завершается упражнениями, часть из которых предназначена для закрепления изложенного материала, а часть требует изучения дополнительной литературы (или собственных умственных усилий) для их решения.

Для более глубокого освоения методов проектирования компиляторов рекомендуются фундаментальные работы [1; 2].

# ЭЛЕМЕНТЫ ТЕОРИИ ФОРМАЛЬНЫХ ЯЗЫКОВ И ГРАММАТИК

## Основные понятия и определения

*Язык* – это множество *строк* (*предложений*, *цепочек*). Каждая строка языка формируется из словаря в соответствии с заданными правилами.

*Строка* есть конечная последовательность символов, каждый из которых принадлежит некоторому конечному *алфавиту* (*словарю языка*) *V*; при этом символы в строке могут повторяться. Если строка содержит *m* символов, то говорят, что она имеет длину *m*. Строка длины 0, т. е. не содержащая ни одного символа, называется *пустой строкой* и обозначается ε. Длина строки *x* обозначается ⎪*x*⎪.

Пусть *V* – некоторый алфавит. Тогда через *V*\* (рефлексивно-транзитивное замыкание *V*) обозначается множество всех строк (включая пустую строку), составленных из символов, входящих в *V*, т. е. это множество определенных над алфавитом *V* строк. Через *V*+ обозначается множество всех строк, исключая пустую строку, т. е. *V*\* = *V*+ ∪ {ε}.

Пример: *V* = {0, 1}.

*V*\* = {ε, 0, 1, 00, 01, 10, 11, 000, ...}.

*V*+ = {0, 1, 00, 01, 10, 11, 000, ...}.

Если α ∈ *V*\* – строка длины *m*, а β ∈ *V*\* – строка длины *n*, то их объединение, обозначаемое αβ, есть конкатенация (сцепление) строк α и β. В строке αβ подстроку α называют *префиксом* строки αβ, а подстроку β – *суффиксом* строки αβ. В результате объединения получается строка длины *m* + *n*, т. е. ⎪αβ⎪= *m* + *n*. Например, если α = *a*1*a*2...*am*, а β = *b*1*b*2...*bn*, тогда αβ = *a*1*a*2...*amb*1*b*2...*bn*. Объединение является ассоциативной операцией, т. е. (αβ)γ = α(βγ) = αβγ, но не является коммутативной операцией, так как в общем случае αβ ≠ βα. Для пустой строки, очевидно, справедливы следующие утверждения:

εα = αε = α для любого α ∈ *V*\*.

Часто используется обозначение α*n* (*n* – степень строки α), соответствующее конкатенации *n* строк α: . Свойства степени: α0 = ε; α*n* = αα*n*–1 = α*n*–1α.

Обычно совокупность строк, принадлежащих *V*\* и имеющих длину 2, обозначают *V*2, имеющих длину 3 – соответственно *V*3 и т. д.; *V*0 – пустая строка. Тогда

 и .

Формальным языком *L* над алфавитом *V* называется произвольное подмножество множества *V*\*. Если *L*1, *L*2 – два формальных языка, то их объединение *L*1*L*2 = {αβ | α ∈ *L*1, β ∈ *L*2} также является формальным языком. Например, если *L*1 = {11, 1} и *L*2 = {ε, *a*, *bb*}, то *L*1*L*2 = {11, 1, 11*a*, 1*a*, 11*bb*, 1*bb*}.

Объединение формального языка *L* с самим собой записывается как *L*2, или, в общем случае, как

*L*0 = {ε}, *L*1 = *L*, *Li* = *LLi*–1 = *Li*–1*L* для *i* ≥ 2.

Синтаксис простого языка можно формально описать, используя систему представления множеств. Например, язык *L* = {*anbn* | *n* ≥ 0} включает строки, состоящие из *n* символов *a* и последующих *n* символов *b*. Пустая строка (при *n* = 0) также принадлежит языку.

Синтаксис сложного языка (к ним относятся и языки программирования) описывается с помощью *грамматики*. Грамматика состоит из набора правил для получения (порождения) строк языка.

## Метаязыки

Язык, используемый для описания синтаксиса какого-либо языка, называется *метаязыком*. Для формального описания синтаксиса языков программирования (*формальных языков*) большое распространение получили такие метаязыки, как *синтаксические диаграммы* (СД) и *формы Бэкуса-Наура* (БНФ).

Рассмотрим диаграмму на рис. 1.1, иллюстрирующую понятие «условный оператор». Прямоугольные вершины соответствуют *нетерминальным символам* (или *нетерминалам*) языка, которые должны быть представлены отдельными СД, определяющими соответствующее понятие. В кружках (овалах) указаны *терминальные символы* (или *терминалы*) языка, входящие в словарь языка (ключевые слова, символы-разделители и т. п.). Ориентированный путь по дугам от начальной вершины до конечной вершины определяет порядок формирования синтаксического понятия.

условие

послед\_операторов

послед\_операторов

Рис. 1.1. Пример синтаксической диаграммы

Альтернативой синтаксическим диаграммам являются формы Бэкуса-Наура. БНФ состоит из набора *правил* (или *продукций*). Нетерминальные символы языка (металингвистические переменные) заключаются в угловые скобки вида < и >. Правило БНФ состоит из левой и правой частей, разделенных метасимволом «::=». Метасимвол «::=» означает «определяется как», «это», «есть». В левой части правила записывается нетерминал (название определяемого синтаксического понятия). Правая часть определяет соответствующее синтаксическое понятие. При наличии альтернативных определений они разделяются метасимволом «**|**», означающим «или».

Представим одно из возможных определений синтаксического понятия «условный оператор» в форме БНФ (нетерминал <пусто> означает пустую строку):

<условный оператор> ::= **if** <условие>

**then** <послед\_операторов> <else-часть> **end if ;**

<else-часть> ::= <пусто>

**|** <список\_elsif> **else** <послед\_операторов>

<список\_elsif> ::= <пусто> **|** <элемент\_elsif> <список\_elsif>

<элемент\_elsif> ::= **elsif** <условие>

**then** <послед\_операторов>

Для БНФ характерно наличие рекурсивных правил (в нашем примере определение нетерминала <список\_elsif>, где этот нетерминал встречается как в левой, так и в правой части). Такие правила обычно определяют различные списковые конструкции.

Процесс построения синтаксической конструкции заключается в выводе этой конструкции путем последовательных подстановок правых частей правил БНФ вместо нетерминалов из левых частей. Такой процесс продолжается до тех пор, пока не будет получена конструкция, состоящая только из терминальных символов.

На практике для удобства применяют расширенные формы Бэкуса-Наура (РБНФ). Существует множество различных вариантов РБНФ. Рассмотрим один из таких вариантов (он достаточно близок к международному стандарту РБНФ ISO/IEC 14977).

Металингвистическая переменная (нетерминал) обозначается произвольной символьной строкой (без использования угловых скобок как в БНФ).

Левая и правая части правила разделяются метасимволом "=" (вместо "::=" в БНФ), альтернативные варианты разделяются метасимволом "**|**". Каждое правило заканчивается метасимволом «;» (точка с запятой), альтернативой символу «;» является символ «.» (точка).

Конкатенация определяется метасимволом «,» (запятая). Правило вида A = B, C. означает, что нетерминал A состоит из двух символов – B и C. Элементы конкатенации называют ещё *синтаксическими факторами*, или просто *факторами*. В таком случае несколько слов, написанных через пробелы, следует понимать как одно многословное имя нетерминального символа (пробелы не являются разделителями символов). Если же не использовать в качестве конкатенации символ «,», а сохранить пробел в качестве разделителя символов, тогда если нетерминал состоит из нескольких смысловых слов, то они записываются слитно или разделяются символом подчеркивания или дефисом. При слитном написании для удобства восприятия целесообразно каждое ее слово начинать с прописной буквы. Например, нетерминал, означающий список идентификаторов, можно записать как «СписокИдентификаторов», «Список-идентификаторов», или «Список\_идентификаторов».

Терминальные символы изображаются словами, написанными буквами латинского алфавита (ключевые слова) или цепочками символов, заключенными в одиночные (′) или двойные (″) кавычки. Для удобства восприятия ключевые слова дополнительно можно выделить жирным шрифтом.

Условное вхождение. Квадратные скобки «[» и «]» означают, что заключенная в них синтаксическая конструкция может отсутствовать.

Повторение. Фигурные скобки «{» и «}» означают нуль или более повторений заключенной в них синтаксической конструкции.

Наиболее важными расширениями являются условное вхождение и повторение. Остальные расширения связаны в основном с синтаксисом обозначений метасимволов.

Ниже приведен пример определений синтаксического понятия «условный оператор» в форме РБНФ.

Условный оператор = "**if**", Условие, "**then**"

Последовательность операторов

{ "**elsif**", Условие, "**then**"

Последовательность операторов }

[ "**else**", Последовательность операторов ]

"**end**", "**if**", "**;**" .

Если не использовать запятую в качестве операции конкатенации, то из многословных обозначений нетерминалов следует убрать пробелы, чтобы обозначения воспринимались как единое целое, например, следующим образом:

УсловныйОператор = "**if**" Условие "**then**"

Последовательность-операторов

{ "**elsif**" Условие "**then**"

Последовательность-операторов }

[ "**else**" Последовательность-операторов ]

"**end**" "**if**" "**;**".

Как видно из примеров, определения в РБНФ получаются более простыми, чем в БНФ, отсутствуют рекурсивные правила, не требуется введение новых нетерминалов.

Нотация БНФ и РБНФ эквивалентна синтаксическим диаграммам и предназначена для описания синтаксиса контекстно-свободных языков программирования. Всегда можно перейти от одного способа описания к другому.

Недостатком рассмотренных метаязыков является то, что они описывают грамматическую структуру формального языка без учёта контекстных зависимостей. При наличии таких зависимостей описание оказывается неполным, и некоторые правила синтаксиса описываемого языка приходится излагать в обычной текстовой форме. Например, в ряде типизированных языков требуется описание типа всех используемых в программе переменных. Сформулировать это требование с помощью нотации БНФ (РБНФ) или СД невозможно.

## Формальные грамматики

Формальные грамматики предназначены для описания формальных языков и имеют большое сходство с БНФ. Однако их нельзя отождествлять, поскольку формальные грамматики имеют более общий характер.

Грамматика определяется как четверка *G* = (*VT*, *VN*, *P*, *S*), где *VT* – конечное множество *терминальных* символов (*терминалов*), т. е. символов, принадлежащих собственно описываемому формальному языку; *VN* – конечное множество *нетерминальных* символов (*нетерминалов*), т. е. символов, принадлежащих метаязыку (необходимо заметить, что у *VT* и *VN*нет общих символов, т. е. *VT* ∩ *VN* = ∅); *P* – конечное множество *продукций* (*правил вывода*, *порождающих правил*) вида α → β, где α – левая часть продукции – это строка такая, что α ∈ (*VT* ∪ *VN*)+, a β – правая часть – строка, такая, что β ∈ (*VT* ∪ *VN*)\*; *S* ∈ *VN* – *начальный символ* (*аксиома*) грамматики.

Примем следующие соглашения об обозначениях:

1. Терминалы будем представлять строчными буквами и спецсимволами из словаря языка (символы цифр, символы операций, символы пунктуации и т. п.). Если терминалы являются словами-символами языка, – выделять жирным шрифтом. Примеры терминалов: *a*, +, **begin**, **while**.

2. Нетерминалы будем представлять прописными буквами. Если обозначение нетерминала является многосимвольным словом, заключать их в угловые скобки (при этом не обязательно использовать прописные буквы). Примеры нетерминалов: *A*, <Оператор>, <Последовательность операторов>.

Пусть дана грамматика *G* = ({*a*, *b*}, {*S*}, *P*, *S*), где *P* представляет множество следующих продукций: *S* → *aSb*, *S* → ε, или, в более краткой форме записи: *S* → *aSb*⎪ε. Продукции с одинаковыми левыми частями, будем называть *альтернативными*.

Чтобы вывести предложение этого языка, поступают следующим образом. Начинают с начального символа *S* и заменяют его на *aSb* или ε. Если *S* опять появится в полученной строке, его опять можно заменить с помощью одного из этих правил, и т. д. Полученная таким образом любая строка, не содержащая *S*, является предложением этого языка. Последовательность таких шагов называется *выводом* (*схемой вывода*, *порождением*) строки (предложения) и обычно записывается следующим образом:

*S* ⇒ *aSb* ⇒ *aaSbb* ⇒ *aaaSbbb* ⇒ *aaabbb*.

Выводимая строка формально определяется следующим образом. Пусть *G* = (*VT*, *VN*, *P*, *S*) – грамматика и пусть γ1αγ2 ∈ (*VT* ∪ *VN*)+ – строка терминальных и нетерминальных символов длиной ≥ 1. Если α → β – продукция из *P*, то подстрока α в строке может быть заменена строкой β, и в результате получится γ1βγ2. Это записывается следующим образом:

γ1αγ2 ⇒ γ1βγ2,

при этом говорят, что строка γ1αγ2 *генерирует* строку γ1βγ2, или что строка γ1βγ2 *выводится* из строки γ1αγ2.

Если α1, α2, ..., α*n* ∈ (*VT* ∪ *VN*)\*, и α1 ⇒ α2 ⇒…⇒ α*n*–1 ⇒ α*n* (*n*≥ 1), то обычно пишут сокращенно , и при этом говорят, что строка α*n* выводится из строки α1 за один или более шагов. Аналогично  означает, что строку α*n*можно вывести из строки α1 с помощью нуля или более применений правил грамматики.

Если строка α ∈ (*VT* ∪ *VN*)\* такая, что , то строку α называют *сентенциальной* *формой* грамматики *G*. *Сентенцией* грамматики *G* называют произвольную сентенциальную форму из *VT*\*, т. е. произвольную строку терминальных символов, которая может быть выведена из начального символа *S*. Тогда множество всех сентенций грамматики *G* называется языком, порожденным грамматикой *G*, и обозначается *L*(*G*).

Таким образом, *L*(*G*) = {*x*∈ *VT*\*⎪}.

Если две различные грамматики *G* и *G'* порождают один и тот же язык, т. е. *L*(*G*) = *L*(*G'*), то грамматики *G* и *G'* *эквивалентны*. Например, грамматики *G*1 = ({*a*, *b*}, {*S*}, *P*1, *S*) и *G*2 = ({*a*, *b*}, {*S*, *A*, *B*}, *P*2, *S*) с продукциями

*P*1: *S* → *aSb*⎪*ab* *P*2: *S* → *aA* | *aB*

*A* → *Sb*

*B* → *b*

эквивалентны, поскольку порождают один и тот же язык *L*(*G*1) = *L*(*G*2) = {*anbn* | *n* ≥ 1}.

В последующем изложении для краткости вместо полной четверки *G* = (*VT*, *VN*, *P*, *S*) грамматику будем представлять только множеством продукций *Р*, считая, что начальный символ грамматики обязательно стоит в левой части первой продукции.

## Классификация грамматик и языков

Одной из стандартных классификаций грамматик является *иерархия Хомского*, основанная на наложении определенных ограничений на вид продукций грамматики.

**Тип 0 – грамматика общего вида**

Любая грамматика определенного выше вида называется *грамматикой типа 0* или *грамматикой общего вида*. На вид продукций не наложено никаких ограничений. Не имеют практического применения из-за своей сложности.

**Тип 1 – контекстно-зависимые**

К этому типу относят контекстно-зависимые и неукорачивающие грамматики.

Грамматика называется *контекстно-зависимой*, если каждая продукция имеет вид α*A*β → αγβ, где α, β ∈ (*VT* ∪ *VN*)\*, γ ∈ (*VT* ∪ *VN*)+, *A* ∈ *VN*.

Грамматика называется *неукорачивающей*, если для всех продукций вида α → β выполняется ограничение ⎪α⎪≤⎪β⎪, где ⎪α⎪ и ⎪β⎪ – соответствующие длины строк. В таких продукциях правая часть не короче левой части.

Эти классы грамматик эквивалентны. В виде исключения в них допускается продукция вида *S* → ε (правая часть короче левой), где *S* – начальный символ грамматики, который не встречается в правых частях других продукций.

Эти грамматики могут использоваться при анализе текстов на естественных языках, однако при построении компиляторов практически не используются из-за своей сложности.

Пример грамматики типа 1:

*S* → *aSBC* | *aBC*

*CB* → *BC*

*aB* → *ab*

*bB* → *bb*

*bC* → *bc*

*cC* → *cc*

Данная грамматика порождает язык {*anbncn* | *n* ≥ 1}.

Пример вывода строки *aabbcc*:

*S* ⇒ *aSBC* ⇒ *aaBCBC* ⇒ *aabCBC* ⇒ *aabBCC* ⇒ *aabbCC*

⇒ *aabbcC*⇒ *aabbcc*.

**Тип 2 – контекстно-свободные**

Грамматика называется *грамматикой тип 2* или *контекстно-свободной* (КС), если каждая продукция имеет вид *A* → β, где *A* ∈ *VN*, β ∈ (*VT* ∪ *VN*)\*. В КС-грамматике все левые части продукций состоят из одного нетерминального символа.

Продукцию вида *A* → ε (правая часть – пустая строка) будем называть ε-*продукцией*.

КС-грамматики широко используются для описания синтаксиса языков программирования.

**Тип 3 – регулярные**

Выделяют классы праволинейных и леволинейных регулярных грамматик.

Грамматика называется *праволинейной регулярной*, если все ее продукции имеют вид *А*→ *а* или *А*→ *аВ*, где *а*∈*VT*, *А*, *В*∈*VN*.

Грамматика называется *леволинейной регулярной*, если все ее продукции имеют вид *А*→ *а* или *А*→ *Ва*, где *а*∈*VT*, *А*, *В*∈*VN*.

Если пустая строка принадлежит языку, допускается единственная продукция вида *S*→ ε (*S* – начальный символ грамматики), при этом ни одна продукция грамматики не должна содержать нетерминал *S* в своей правой части.

Эти классы грамматик называют также *автоматными*, поскольку существуют простые методы синтеза конечных автоматов по заданным автоматным грамматикам.

Регулярные (автоматные) грамматики являются частным случаем линейных грамматик.

Грамматика называется *линейной*, если все ее продукции (за исключением ε-продукций) имеют вид *А*→ α или *А*→ α*В*β, где α, β∈*VT*\*, *А*, *В*∈*VN*, т. е. в правой части продукции может содержаться не более одного нетерминала. Если во всех продукциях вида *А*→ α*В*β имеет место α = ε, грамматика называется *леволинейной*, если же β = ε – *праволинейной*. Классы леволинейных и праволинейных грамматик эквивалентны и описывают регулярные языки, они эквивалентны регулярным автоматным грамматикам (поэтому в ряде работ их относят к регулярным грамматикам). Любую праволинейную (леволинейную) грамматику можно преобразовать в эквивалентную регулярную автоматную грамматику.

Например, продукцию вида *A* → *abcB* можно заменить на продукции *A* → *aС*, *C* → *bD*, *D* → *cB* (обозначив подстроку *bcB* через нетерминал *C*, подстроку *cB* через нетерминал *D*) и получить эквивалентную автоматную грамматику.

В связи с тем, что классы праволинейных и леволинейных регулярных грамматик эквивалентны, в дальнейшем будем подразумевать под регулярными грамматиками праволинейные регулярные автоматные грамматики.

Следует заметить, что продукции регулярной грамматики должны быть либо только леволинейными, либо только праволинейными. Их совместное использование в общем случае выводит грамматику из класса регулярных грамматик. Например, грамматика

*S* → *aA* | *aB*

*A* → *Sb*

*B* → *b*

описывает контекстно-свободный язык {*anbn* | *n* ≥ 1} и не является регулярной.

Очевидно, что эта иерархия – включающая, т. е. регулярные грамматики являются контекстно-свободными, которые в свою очередь являются контекстно-зависимыми, и  т. д.

Иерархии грамматик соответствует иерархия языков. Однако при этом необходимо учитывать следующий факт. Например, если язык генерируется посредством контекстно-зависимой грамматики, то это не обязательно означает, что язык только контекстно-зависимый и не может быть контекстно-свободным, поскольку, если язык является контекстно-свободным, то всегда можно определить эквивалентную контекстно-свободную грамматику.

Распознавателем языка типа 0 является машина Тьюринга. Распознавателем контекстно-зависимых языков является линейно ограниченный автомат, представляющий собой машину Тьюринга, в которой лента не бесконечна, а ограничена длиной входного слова.

Наибольшее практическое применение находят регулярные (на этапе лексического анализа) и контекстно-свободные (на этапе синтаксического анализа) грамматики, которые позволяют специфицировать большинство конструкций современных языков программирования и использовать в качестве распознавателей строк языка достаточно простые средства. В частности, распознавателем регулярного языка является конечный автомат, а распознавателем контекстно-свободного языка – магазинный автомат (автомат с магазинной памятью).

## Контекстно-свободные грамматики

Формальное определение основных синтаксических понятий большинства языков программирования может быть учтено с помощью БНФ (РБНФ) или синтаксических диаграмм. При этом вид левой части каждой продукции может быть ограничен лишь единственным нетерминалом, т. е. синтаксис большинства языков программирования в своей основной части может быть определен с помощью контекстно-свободных грамматик.

Контекстно-свободной грамматикой (КС-грамматикой) называется грамматика *G* = (*VT*, *VN*, *P*, *S*), каждая продукция которой имеет вид *A* → β, где *А* ∈ *VN*, β ∈ (*VT* ∪*VN*)\*. Для краткости продукции вида *A* → β с нетерминалом *A* в левой части будем называть *A*-*продукциями*.

При соблюдении соглашений об обозначении терминалов и нетерминалов для задания грамматики достаточно определить множество продукций и указать начальный нетерминал. Если не оговорено особо, начальным нетерминалом будем считать нетерминал левой части первой продукции множества.

Язык, порожденный КС-грамматикой, называется контекстно-свободным языком. Термин «контекстно-свободный» обусловлен тем, что любой символ *А* ∈ *VN* в сентенциальной форме грамматики *G* может быть раскрыт согласно продукции *А* → β независимо от того, какими строками он окружен внутри самой сентенциальной формы.

В КС-грамматике любая строка *x* ∈ *L*(*G*) может быть выведена из начального символа *S*. Общепринятым методом представления такого вывода является *дерево разбора* (*дерево грамматического разбора*, *дерево вывода*).

Дерево разбора строится по порождению *S* ⇒ *A*1*A*2...*An* ⇒...⇒ *T*1*T*2...*Tk*, где *Ai* ∈ (*VT* ∪ *VN*), *i* = 1, 2, …, *n*; *Tj* ∈ *VT*, *j* = 1, 2, …, *k*, следующим образом. Из корня дерева, помеченного начальным нетерминалом *S*, отходит *n* ветвей по числу символов в строке, непосредственно порожденной начальным нетерминалом. Каждая из *n* ветвей заканчивается вершиной, помеченной символом *Аi*. Вершины метятся слева направо в порядке возрастания номера индекса. Если в процессе порождения к нетерминалу *Аi* применена продукция *Аi* → *В*1*В*2...*Вt*, то вершина *Аi* становится корнем поддерева, из которого выходит *t* ветвей, каждая из которых заканчивается вершиной, помеченной символом *Вi*, *i* = 1, 2, …, *t*. Такое построение производится для всех вершин, помеченных нетерминальными символами. Построение дерева заканчивается, когда все листья оказываются помеченными терминалами *T*1, *T*2, ..., *Tk*. Совокупность этих меток при просмотре вершин слева направо образует терминальную строку (сентенцию) *T*1*T*2...*Tk* (*крона* *дерева разбора*). Очевидно, что в полностью построенном дереве листья соответствуют терминалам, а внутренние вершины – нетерминалам.

Вывод строки *x* ∈ *L*(*G*) называется *левосторонним* (*левым*), если каждая очередная сентенциальная форма получается из предыдущей заменой самого левого нетерминала.

Вывод строки *x* ∈ *L*(*G*) называется *правосторонним* (*правым*), если каждая очередная сентенциальная форма получается из предыдущей заменой самого правого нетерминала.

Пример построения дерева разбора для грамматики *G* с продукциями

*S*→*AB*

*A*→*aA*|*a*

*B*→*bB*|*b*

при выводе строки *aaabb* (в более краткой форме – *a*3*b*2) представлен на рис. 1.2. Это дерево соответствует левостороннему выводу: *S* ⇒ *AB* ⇒ *aAB* ⇒ *aaAB* ⇒ *aaaB* ⇒ *aaabB* ⇒ *aaabb*.



Рис. 1.2. Дерево разбора строки *aaabb*

Этому же дереву соответствует правосторонний вывод строки *aaabb*: *S* ⇒ *AB* ⇒ *AbB* ⇒ *Abb* ⇒ *aAbb* ⇒ *aaAbb* ⇒ *aaabb*.

Возможны и другие варианты вывода одной и той же строки, не являющиеся ни левосторонними, ни правосторонними. Например, строку *a*3*b*2 можно вывести следующим образом:

*S* ⇒ *AB* ⇒ *AbB* ⇒ *aAbB* ⇒ *aaAbB* ⇒ *aaAbb* ⇒ *aaabb*,

*S* ⇒ *AB* ⇒ *aAB* ⇒ *aAbB* ⇒ *aAbb* ⇒ *aaAbb* ⇒ *aaabb*,

*S* ⇒ *AB* ⇒ *aAB* ⇒ *aaAB* ⇒ *aaAbB* ⇒ *aaabB* ⇒ *aaabb*.

Все эти схемы вывода соответствуют одному и тому же дереву разбора. Заметим, что дереву разбора соответствует единственный левосторонний (правосторонний) вывод.

Если для любой строки *x* ∈ *L*(*G*) все возможные схемы вывода соответствуют одному и тому же дереву разбора, то такая КС-грамматика называется *однозначной*. Если же различным схемам вывода соответствуют несовпадающие деревья разбора, то грамматика называется *неоднозначной*.

Поскольку каждому дереву разбора соответствует единственный левосторонний вывод, то можно переопределить неоднозначную грамматику следующим образом: КС-грамматика *G* называется неоднозначной, если существует такая строка *x* ∈ *L*(*G*), которой можно поставить в соответствие две или более различные левосторонние схемы вывода.

Для примера рассмотрим неоднозначную грамматику с продукциями *S* → *SaS*⎪*b*.

Строку *babab* можно получить двумя различными левосторонними схемами вывода:

*S* ⇒ *SaS* ⇒ *SaSaS* ⇒ *baSaS* ⇒ *babaS* ⇒ *babab*,

*S* ⇒ *SaS* ⇒ *baS* ⇒ *baSaS* ⇒ *babaS* ⇒ *babab*.

Этим схемам вывода соответствуют различные деревья разбора, изображенные на рис. 1.3.



Рис. 1.3. Деревья разбора строки *babab*

Задача установления неоднозначности какой-либо грамматики является алгоритмически неразрешимой, т. е. не существует алгоритма, который принимал бы любую грамматику в качестве входа и определял бы, однозначна она или нет. В некоторые языки уже заложена неоднозначность. Это означает, что их нельзя генерировать с помощью однозначной грамматики. С другой стороны, некоторые неоднозначные грамматики можно преобразовать в однозначные, генерирующие тот же язык. Например, грамматика с продукциями *S* → *Sab*⎪*b* является однозначной и генерирует тот же язык, что и рассмотренная выше неоднозначная грамматика. Строка *babab* получается следующей левосторонней схемой вывода: *S* ⇒ *Sab* ⇒ *Sabab* ⇒ *babab*.

Рассмотрим еще один, ставший уже классическим, пример неоднозначной грамматики, определяющей условный оператор языка Паскаль:

*S* → **if** *E* **then** *S*⎪**if** *E* **then** *S* **else** *S*⎪**other**

Нетерминал *E* означает выражение, нетерминал *S* – оператор, терминал **other** – другие операторы.

Строку

**if** *E* **then** *S* **if** *E* **then** *S* **else** *S*

можно получить двумя левосторонними схемами вывода. Для конструкций подобного вида существует соглашение, что **else** относится к ближайшему незанятому **then**. Это правило устранения неоднозначности можно реализовать, изменив соответствующим образом грамматику, сделав ее однозначной.

Идея заключается в том, что между **then** и **else** допускается либо полный условный оператор **if-then-else**, либо любой оператор, не являющийся условным (такую синтаксическую конструкцию обозначим нетерминалом *A*). Остальные возможные конструкции обозначим нетерминалом *B*. В результате получится эквивалентная однозначная грамматика.

*S* → *A*⎪*B*

*A* → **if** *E* **then** *A* **else** *A*⎪**other**

*B* → **if** *E* **then** *S*⎪**if** *E* **then** *A* **else** *B*

Грамматика получилась сложнее, менее естественной по сравнению с неоднозначной грамматикой.

При описании языка, если используется неоднозначная грамматика, всегда задаются правила разрешения неоднозначности. Эти правила реализуются либо построением эквивалентных однозначных грамматик (если это не слишком усложняет грамматику), либо непосредственно при разработке синтаксического анализатора, включая в него специальные механизмы реализации этих правил.

## Эквивалентные преобразования грамматик

При построении грамматик часто возникает необходимость в их эквивалентных преобразованиях для того, чтобы они удовлетворяли определенным критериям, но при этом не изменялся порождаемый язык. Рассмотрим некоторые простые, но наиболее часто применяемые и важные приемы преобразований контекстно-свободных грамматик.

### Удаление бесполезных символов

Нетерминал *X* называется *производящим* (*продуктивным*), если , *w* ∈ *VT*\*, т. е. из нетерминала *X* можно вывести какую-нибудь терминальную строку. Нетерминал называется *непроизводящим* (*бесплодным*), если он не порождает ни одной терминальной строки. Очевидно, что если все символы правой части продукции производящие, то производящим является и нетерминал в левой части. На этом свойстве основана процедура выявления непроизводящих нетерминалов:

1. Составить список нетерминалов, для которых существует хотя бы одна продукция, правая часть которой не содержит нетерминалов.

2. Если найдена такая продукция, что все нетерминалы, стоящие в ее правой части, уже занесены в список, то добавить в список нетерминал из левой части.

Если на шаге 2 список больше не пополняется, то получен список всех производящих нетерминалов, а все не попавшие в него нетерминалы – непроизводящие.

Формально данное преобразование можно представить алгоритмом 1.1, который исходную КС-грамматику *G* = (*VT*, *VN*, *P*, *S*) преобразует в эквивалентную грамматику *G'* = (*VT*, *V'N*, *P'*, *S*), не содержащую непроизводящих символов. Сначала определяется множество *V'N* производящих нетерминалов путем рекурсивного построения множеств производящих нетерминалов *N*0, *N*1, *N*2,… Затем в множество *P'* продукций включаются только те продукции из *P*, которые содержат символы из *V'N* ∪*VT*. Логическая переменная *b* служит для реализации выхода из цикла после завершения вычисления множества производящих нетерминалов.

**Алгоритм 1.1.** Удаление непроизводящих символов

**Вход:** КС-грамматика *G* = (*VT*, *VN*, *P*, *S*)

**Выход:** *G'* = (*VT*, *V'N*, *P'*, *S*) – эквивалентная грамматика, не содержащая непроизводящих символов



Символ грамматики *X* ∈ *VT* ∪*VN* (терминал или нетерминал) называется *достижимым*, если существует вывод  для некоторых α, β ∈ (*VT* ∪ *VN*)\*, т. е. символ появляется хотя бы в одной сентенциальной форме грамматики. В противном случае символ грамматики называется *недостижимым*. Очевидно, что если нетерминал левой части продукции является достижимым, то и все символы правой части достижимы. На этом свойстве основана процедура выявления недостижимых символов, которую можно представить следующим образом:

1. Образовать одноэлементный список, состоящий из начального символа грамматики.

2. Если найдена продукция, левая часть которой уже имеется в списке, то включить в список все символы, содержащиеся в ее правой части.

Если на шаге 2 список не пополняется новыми символами, то получен список всех достижимых символов, а символы, не попавшие в список, являются недостижимыми.

Формально данное преобразование можно представить алгоритмом 1.2, который исходную КС-грамматику *G* = (*VT*, *VN*, *P*, *S*) преобразует в эквивалентную грамматику *G'* = (*V'T*, *V'N*, *P'*, *S*), не содержащую недостижимых символов. Сначала определяется множество *Vi* достижимых символов путем рекурсивного построения множеств *V*0, *V*1, *V*2,… Затем в множество *P'* продукций включаются только те продукции из *P*, которые содержат символы из *Vi*. Логическая переменная *b* служит для реализации выхода из цикла после завершения вычисления множества достижимых символов.

**Алгоритм 1.2.** Удаление недостижимых символов

**Вход:** КС-грамматика *G* = (*VT*, *VN*, *P*, *S*)

**Выход:** *G'* = (*V'T*, *V'N*, *P'*, *S*) – эквивалентная грамматика, не содержащая недостижимых символов



Символы, которые являются непроизводящими или недостижимыми, называются *бесполезными*. Исключение бесполезных символов из грамматики заключается в исключении их из соответствующих множеств и удалении продукций, содержащих эти символы. Исключение выполняется в следующем порядке:

1. Удалить из грамматики *G* = (*VT*, *VN*, *P*, *S*) непроизводящие символы и получить грамматику *G'* = (*VT*, *V'N*, *P'*, *S*).

2. Удалить из грамматики *G'* = (*VT*, *V'N*, *P'*, *S*) недостижимые символы и получить грамматику *G''* = (*V''T*, *V''N*, *P''*, *S*).

Рассмотрим грамматику *G* = ({*a*, *b*, *c*}, {*S*, *A*, *B*, *C*}, *P*, *S*) с продукциями

*S* → *aSb*⎪*cAc*⎪*a B* → *aa*

*A* → *ABa*⎪*cAb C* →*ac*

Непроизводящим является нетерминал *A*, после его исключения из *VN* и удаления продукций *S* → *cAc* и *A* → *ABa*⎪*cAb* получится грамматика *G'* = ({*a*, *b*, *c*}, {*S*, *B*, *C*}, *P'*, *S*) с продукциями

*S* → *aSb*⎪*a*

*B* → *aa*

*C* →*ac*

Недостижимыми являются символы *B*, *C* и *c*. После их исключения получим грамматику *G''* = ({*a*, *b*}, {*S*}, *P''*, *S*) с множеством продукций *S* → *aSb*⎪*a*.

Грамматику, не содержащую бесполезные символы, будем называть *приведенной*.

Следует обратить внимание на то, что действия по удалению бесполезных символов должны выполняться именно в указанном порядке. В противном случае не всегда результатом будет приведенная грамматика.

Проиллюстрируем это на рассмотренной выше грамматике *G* = ({*a*, *b*, *c*}, {*S*, *A*, *B*, *C*}, *P*, *S*) с продукциями

*S* → *aSb*⎪*cAc*⎪*a B* → *aa*

*A* → *ABa*⎪*cAb C* →*ac*

Удалим недостижимый нетерминал *C*. В результате получим грамматику *G'* = ({*a*, *b*, *c*}, {*S*, *A*, *B*}, *P'*, *S*) с продукциями

*S* → *aSb*⎪*cAc*⎪*a*

*A* → *ABa*⎪*cAb*

*B* → *aa*

Удалим непроизводящий нетерминал *A*. Полученная грамматика *G''* = ({*a*, *b*, *c*}, {*S*, *B*}, *P''*, *S*) с продукциями

*S* → *aSb*⎪*a*

*B* → *aa*

не является приведенной, поскольку нетерминал *B* и терминал *c* в результате удаления нетерминала *A* стали недостижимыми.

В дальнейшем будем рассматривать только приведенные КС-грамматики.

### Замена вхождений

Данное преобразование позволяет сократить число нетерминалов в грамматике и состоит в том, что если левая часть продукции входит в правую часть продукции *R*, то замена данного вхождения приведет просто к замене продукции *R* другой продукцией. Если такую замену выполнить для всех продукций с данным нетерминалом в левой части, то этот нетерминал можно исключить из грамматики. Исключением является случай, когда правая часть продукции содержит нетерминал левой части, например *S* → *aSb*⎪*a*. В этом случае удаление нетерминала из грамматики невозможно.

Формально данное преобразование можно представить следующим образом. Если *A* → α1*B*α2 – продукция грамматики и *B* → β1⎪β2⎪…⎪β*k* – все *B*-продукции этой грамматики, тогда продукции вида *A* → α1*B*α2 можно поставить в соответствие продукции вида *A* → α1β1α2⎪α1β2α2⎪…⎪α1β*k*α2.

Пусть дана грамматика с множеством продукций

*S* → *AB*⎪*Bb*⎪*Ba*

*A* → *a*

*B* → *aSb*⎪*b*

Выполнив замену вхождений нетерминала *B*, получим

*S* → *AaSb*⎪*Ab*⎪*aSbb*⎪*bb*⎪*aSba*⎪*ba*

*A* → *a*

Замена вхождения нетерминала *A* приведет к следующему множеству продукций:

*S* → *aaSb*⎪*ab*⎪*aSbb*⎪*bb*⎪*aSba*⎪*ba*

В общем случае замена вхождений приводит к увеличению числа продукций. Исключение представляет случай, когда заменяемый нетерминал является левой частью единственной продукции (в примере – нетерминал *A*).

Можно использовать и обратное преобразование, когда некоторая подстрока заменяется новым нетерминалом, с целью сокращения числа продукций.

### Устранение леворекурсивных продукций

Продукция вида *A* → *A*α, где *A* ∈ *VN*, α ∈ (*VT* ∪ *VN*)\*, называется *леворекурсивной* (содержит *прямую левую рекурсию*), а продукция вида *A* → α*A* – *праворекурсивной*. Для любой КС-грамматики, содержащей леворекурсивные продукции, можно построить эквивалентную КС-грамматику, не содержащую леворекурсивных продукций.

Преобразование заключается в следующем. Пусть множество продукций грамматики содержит леворекурсивные продукции *A* → *A*α1⎪*A*α2⎪…⎪*A*α*m* и все остальные *A*-продукции *A* → β1⎪β2⎪…⎪β*n*, не являющиеся леворекурсивными. Тогда новая эквивалентная грамматика может быть построена добавлением нового нетерминала *A'* и заменой леворекурсивных продукций продукциями вида

*A* → β1⎪β2⎪…⎪β*n*⎪β1*A'*⎪β2*A'*⎪…⎪β*nA'*

*A'* → α1⎪α2⎪…⎪α*m*⎪α1*A'*⎪α2*A'*⎪…⎪α*mA'*

Таким образом, рассмотренное преобразование заменяет левую рекурсию на правую, и может быть выполнено для любой КС-грамматики.

Для иллюстрации техники преобразования рассмотрим грамматику (*E* – начальный символ):

*E* → *E* + *T*⎪*T*

*T* → *T* × *F*⎪*F*

*F* → (*E*)⎪*i*

В соответствии с рассмотренным выше правилом продукции *E* → *E* + *T*⎪*T* преобразуются в продукции *E* → *T*⎪*TE'* и *E'* → + *T*⎪+ *TE'*, аналогично продукции *T* → *T* × *F*⎪*F* преобразуются в продукции *T* → *F*⎪*FT'* и *T'* → × *F*⎪× *FT'*. В результате получается грамматика без леворекурсивных продукций:

*E* → *T*⎪*TE'*

*E'* → + *T*⎪+ *TE'*

*T* → *F*⎪*FT'*

*T'* → × *F*⎪× *FT'*

*F* → (*E*)⎪*i*

### Устранение леворекурсивного цикла

Говорят, что грамматика имеет *леворекурсивный цикл* (*косвенную левую рекурсию*), если в грамматике имеется нетерминал *A* такой, что , т. е. из нетерминала *A* можно вывести строку, начинающуюся с *A*. Заметим, что понятие леворекурсивной продукции есть частный случай общего понятия леворекурсивного цикла.

Грамматику, содержащую леворекурсивный цикл, можно достаточно просто преобразовать в грамматику, содержащую только леворекурсивные продукции (прямую левую рекурсию), и далее исключить леворекурсивные продукции, преобразовав их в праворекурсивные. Замена леворекурсивного цикла на прямую левую рекурсию представлена алгоритмом 1.3.

**Алгоритм 1.3.** Замена леворекурсивного цикла на прямую левую рекурсию

**Вход:** КС-грамматика *G* = (*VT*, *VN*, *P*, *S*) с леворекурсивным циклом

**Выход:** КС-грамматика *G'* = (*VT*, *VN*, *P'*, *S*) с прямой левой рекурсией

*Шаг 1*. Упорядочить нетерминалы грамматики, начиная с начального, в порядке их появления в продукциях грамматики, т. е. *VN* = {*A*1, *A*2, …, *Am*}, *m* ≥ 1, где *A*1 соответствует начальному нетерминалу. В результате продукции, у которых правая часть начинается с нетерминала, примут вид *Ai* → *Aj*α, где α ∈ (*VT* ∪ *VN*)\*. Если у всех продукций *i* < *j*, левая рекурсия в грамматике отсутствует, если *i* = *j*, данная продукция является леворекурсивной, если *i* > *j*, имеет место леворекурсивный цикл.

*Шаг 2*. Для продукций вида *Ai* → *Aj*α, где *i* > *j*, производится замена вхождений нетерминала *Aj*. Такая последовательность замен повторяется до тех пор, пока не получится продукция, для которой *i* = *j*, т. е. пока не сведется к прямой левой рекурсии.

В качестве примера такого преобразования рассмотрим следующую грамматику:

*S* → *AS*⎪*AB*

*A* → *BS*⎪*a*

*B* → *SA*⎪*b*

Прежде всего, упорядочим нетерминалы грамматики. Для этого переименуем их следующим образом: *S* переименуем в *A*1, *A* – в *A*2 и *B* – в *A*3. В результате продукции будут иметь вид:

*A*1 → *A*2*A*1⎪*A*2*A*3

*A*2 → *A*3*A*1⎪*a*

*A*3 → *A*1*A*2⎪*b*

Продукция *A*3 → *A*1*A*2 показывает, что в грамматике имеется леворекурсивный цикл. Произведем замену вхождений нетерминала *A*1 следующим образом: *A*3 → *A*2*A*1*A*2⎪*A*2*A*3*A*2. Поскольку условие *i* = *j* еще не выполнено, продолжим процесс замены вхождений *A*2: *A*3 → *A*3*A*1*A*1*A*2⎪*aA*1*A*2⎪*A*3*A*1*A*3*A*2⎪*aA*3*A*2. Получили леворекурсивные продукции, процесс замен прекращается. В результате получена грамматика, в которой вместо леворекурсивного цикла имеется прямая левая рекурсия

*S* → *AS*⎪*AB*

*A* → *BS*⎪*a*

*B* → *BSSA*⎪*aSA*⎪*BSBA*⎪*aBA*⎪*b*

### Факторизация

Если в грамматике имеются продукции вида

*A* → αβ1⎪αβ2⎪…⎪αβ*k*, α ∈ (*VT* ∪ *VN*)+, β*i* ∈ (*VT* ∪ *VN*)\*, 1 ≤ *i* ≤ *k*

с нетерминалом *A* в левой части, то эти продукции можно заменить, добавив новый нетерминал *X*, на следующие:

*A* → α*X*

*X* → β1⎪β2⎪…⎪β*k*

Такое преобразование называется *левой* *факторизацией*. Левую факторизацию можно рассматривать как вынос за скобки общего префикса α, а то, что осталось в скобках заменяется новым нетерминалом *X*. Для наглядности в качестве промежуточной можно использовать форму записи *A* → α(β1⎪β2⎪…⎪β*k*).

Например, пусть дана грамматика с продукциями

*S* → *aSb*⎪*aSc*⎪*d*

Факторизации преобразует их в следующие продукции:

*S* → *aSX*⎪*d*

*X* → *b*⎪*c*

Аналогично можно определить правую факторизацию.

### Удаление ε-продукций

Продукция вида *A* → ε, называется ε-*продукцией* (*аннулирующей продукцией*). Наличие в грамматике ε-продукций может усложнить реализацию некоторых методов синтаксического анализа. Поэтому на практике удаление ε-продукций позволяет упростить процесс построения синтаксического анализатора, хотя и может привести к существенному росту числа продукций. В случаях, когда при преобразованиях грамматики к виду, необходимому для реализации ряда методов синтаксического анализа, напротив, могут появиться ε-продукции, они, обычно, проблем не вызывают.

Следует заметить, что если пустая строка принадлежит языку, то избавиться от ε-продукции в ее грамматике, не изменяя порождаемого языка, невозможно. Самое большее, чего можно достигнуть, это преобразовать грамматику *G* таким образом, чтобы полученная грамматика *G'* не содержала ε-продукции и выполнялось условие *L*(*G'*) = *L*(*G*) – {ε}.

Грамматика *G* = (*VT*, *VN*, *P*, *S*) называется ε-*свободной* (или *неукорачивающей*), если

а) либо она не имеет ε-продукций (в случае, когда ε ∉ *L*(*G*)),

б) либо имеется только одна ε-продукция *S* → ε и *S* не встречается в правых частях всех продукций грамматики (в случае, когда ε ∈ *L*(*G*)).

Нетерминал *A* ∈ *VN* называется ε-*порождающим нетерминалом*, если , т. е. из *A* выводима пустая строка. Построение множества *N*ε ⊆ *VN* ε-порождающих нетерминалов для заданной КС-грамматики *G* = (*VT*, *VN*, *P*, *S*) можно представить алгоритмом 1.4. Множество *N*ε определяется путем рекурсивного построения множеств *N*0, *N*1, *N*2,… Логическая переменная *b* служит для реализации выхода из цикла после завершения вычисления множества *N*ε.

**Алгоритм 1.4.** Построение множества ε-порождающих нетерминалов

**Вход:** КС-грамматика *G* = (*VT*, *VN*, *P*, *S*)

**Выход:** *N*ε – множество ε-порождающих нетерминалов



Для любой КС-грамматики *G* = (*VT*, *VN*, *P*, *S*), содержащей ε-продукции, можно построить эквивалентную ε-свободную КС-грамматику *G'* = (*VT*, *V'N*, *P'*, *S'*) в соответствии с алгоритмом 1.5.

**Алгоритм 1.5.** Удаление ε-продукций

**Вход:** КС-грамматика *G* = (*VT*, *VN*, *P*, *S*)

**Выход:** ε-свободная КС-грамматика *G'* = (*VT*, *V'N*, *P'*, *S'*)

*Шаг 1*. Построить множество *N*ε ⊆ *VN* ε-порождающих нетерминалов в соответствии с алгоритмом 1.4.

*Шаг 2*. *P'* := *P* – {*A* → ε ∈ *P* для всех *A* ∈ *VN* }.

*Шаг 3*. Для каждой продукции *p* ∈ *P'* вида

*A* → α0*B*1α1*B*2α1… *Bk*α*k* ∈ *P*,

где *k* ≥ 0, *Bj* ∈ *N*ε (1 ≤ *j* ≤ *k*) и ни один из символов строк α*i* ∈ (*VT* ∪ *VN*)\* (0 ≤ *i* ≤ *k*) не содержит нетерминалов из *N*ε, включить в *P'* все продукции вида *A* → α0*X*1α1*X*2α1… *Xk*α*k*, где *Xj* = *Bj* или *Xj* = ε. Если все строки α*i* = ε, то продукцию *A* → ε не включать в *P'*.

*Шаг 4*. Если *S* ∈ *N*ε, то *V'N* := {*S'*} ∪ *VN* и добавить в *P'* продукции *S'* → ε⎪*S*, где *S'* – новый начальный символ грамматики. В противном случае положить *V'N* = *VN* и *S'* = *S*. Замечание: если *S* ∈ *N*ε и *S* не встречается в правых частях продукций, то достаточно добавить в *P'* продукцию *S* → ε, тогда *V'N* = *VN* и *S'* = *S*.

На первом шаге алгоритма 1.5 строится множество *N*ε ⊆ *VN* ε-порождающих нетерминалов. На втором шаге в множество *P'* объединяются все имеющиеся в *P* продукции, за исключением ε-продукций. На третьем шаге каждой продукции *p* ∈ *P'*, у которой содержится в правой части ε-порождающие нетерминалы, ставится в соответствие такие продукции, что в их правых частях опущены (по сравнению с продукцией *p*) все возможные комбинации ε-порождающих нетерминалов из множества *N*ε, полученные продукции присоединяются к множеству *P'*. Другими словами, необходимо во все продукции грамматики выполнить все возможные подстановки пустой строки вместо ε-порождающего нетерминала. Четвертый шаг выполняется только для случая ε ∈ *L*(*G*), т. е. пустая строка принадлежит языку.

Проиллюстрируем преобразование на примере грамматики

*S* → *ASB*⎪ε

*A* → *aA*⎪ε

*B* → *bB*⎪*b*

Нетерминалы *S* и *A* являются ε-порождающими. Выполнение всех возможных замен этих нетерминалов пустой строкой приведет к следующим продукциям:

*S* → *ASB*⎪*SB*⎪*AB*⎪*B*

*A* → *aA*⎪*a*

*B* → *bB*⎪*b*

Поскольку пустая строка принадлежит языку, а начальный нетерминал *S* входит в правые части некоторых продукций, добавим продукции *S'* → ε и *S'* → *S*. В результате получается следующая грамматика:

*S'* → ε⎪*S*

*S* → *ASB*⎪*SB*⎪*AB*⎪*B*

*A* → *aA*⎪*a*

*B* → *bB*⎪*b*

### Удаление цепных продукций

*Цепной* (*сингулярной*) *продукцией* (*цепным правилом*) называется продукция вида *A* → *B*, где *A*, *B* ∈ *VN*, т. е. правая часть представляет собой один нетерминал. Цепные продукции могут быть удобны для построения грамматик (повышают наглядность грамматики), но приводят к излишним шагам при выводе.

Продукция вида *A* → *A* также относится к цепным продукциям. Если такая продукция принадлежит множеству продукций грамматики или появляется при каких-либо преобразованиях, то она должна быть просто исключена из множества продукций.

Пусть в грамматике наряду с другими продукциями имеются цепные продукции *A* → *B*, *B* → *C*, *C* → *D*. Тогда существует вывод нетерминала *D* из нетерминала *A*, в котором на всех шагах применяются только цепные правила (*цепной вывод*):

*A* ⇒ *B* ⇒ *C* ⇒ *D*.

Поскольку грамматика приведенная (в ней нет бесполезных символов), обязательно должна существовать нецепная продукция *D* → α, т. е имеется схема вывода строки α:

*A* ⇒ *B* ⇒ *C* ⇒ *D* ⇒ α.

Cтроку α можно вывести из нетерминала *A* за один шаг, если добавить в грамматику продукцию *A* → α. Аналогично из нетерминалов *B* и *C* можно вывести строку α за один шаг, добавив в грамматику продукции *B* → α, *C* → α. Другими словами, выполняется замена вхождений нетерминалов *A*, *B* и *C* на α. На этом и основано удаление цепных правил.

Формально данное преобразование можно представить алгоритмом 1.6.

**Алгоритм 1.6.** Удаление цепных продукций

**Вход:** ε-свободная КС-грамматика *G* = (*VT*, *VN*, *P*, *S*)

**Выход:** ε-свободная КС-грамматика *G'* = (*VT*, *VN*, *P'*, *S*) без цепных продукций

*Шаг 1*. Для каждого нетерминала *A* ∈ *VN* построить множество , т. е. множество нетерминалов, для которых существует цепной вывод из нетерминала *A*, следующим образом (рекуррентное вычисление каждого *NA*):



*Шаг 2*. Построить множество продукций *P'*: если продукция *B* → α ∈ *P* и не является цепной продукцией, то включить в *P'* продукцию *A* → α для всех таких нетерминалов *A*, что *B* ∈ *NA*.

Рассмотрим данное преобразование на примере следующей грамматики:

*E* → *E* + *T*⎪*T*

*T* → *T* × *F*⎪*F*

*F* → (*E*)⎪*i*

Для всех нетерминалов *A* ∈ *VN* вычислим соответствующие множества: *NE* = {*E*, *T*, *F*}, *NT* = {*T*, *F*}, *NF* = {*F*}. Построим новое множество продукций *P'*:

*E* → *E* + *T*⎪*T* × *F*⎪(*E*)⎪*i*

*T* → *T* × *F*⎪(*E*)⎪*i*

*F* → (*E*)⎪*i*

Получена эквивалентная грамматика без цепных продукций.

Как видно из примера, увеличилось число продукций. Такое увеличение числа продукций приводит к усложнению синтаксического анализа. Например, при использовании табличных методов синтаксического анализа это приводит к увеличению размеров таблиц разбора.

## Канонические формы КС-грамматик

Разнообразие форматов продукций КС-грамматик усложняет их теоретические исследования и доказательства ряда важных теорем теории формальных языков и грамматик. Поэтому естественно стремление исследователей ограничить форматы продукций без изменения порождаемых языков и ухудшения выразительных свойств грамматик.

Рассмотрим две канонические формы КС-грамматик: нормальная форма Хомского и нормальная форма Грейбах.

### Нормальная форма Хомского

КС-грамматика *G* = (*VT*, *VN*, *P*, *S*) называется грамматикой в *нормальной форме Хомского*, если множество продукций содержит продукции только следующего вида:

*A* → *BC*, где *A*, *B*, *C* ∈ *VN*;

*A* → *a*, где *A* ∈ *VN*, *a* ∈ *VT*;

*S* → ε, если ε ∈ *L*(*G*), причем *S* не должен встречаться в правых частях других продукций.

Для ε-свободных КС-грамматик существует альтернативное определение грамматики в нормальной форме Хомского, которое отличается тем, что в нем отсутствует ε-продукция *S* → ε и в продукции вида *A* → *BC* нетерминалы *B* и *C* могут быть начальными символами грамматики.

Любая КС-грамматика может быть преобразована в эквивалентную КС-грамматику в нормальной форме Хомского. Преобразование представлено алгоритмом 1.7.

**Алгоритм 1.7.** Преобразование КС-грамматики к нормальной форме Хомского

**Вход:** КС-грамматика *G* = (*VT*, *VN*, *P*, *S*)

**Выход:** КС-грамматика *G'* = (*VT*, *V'N*, *P'*, *S'*) в нормальной форме Хомского

*Шаг 1*. Удалить ε-продукции (см. раздел 1.6.6). Если в грамматике нет ε-продукций, а начальный символ *S* входит в правые части продукций, то пополнить грамматику новым начальным символом *S'* и включить в нее продукцию *S'* → *S*.

*Шаг 2*. Удалить цепные продукции (см. раздел 1.6.7).

*Шаг 3*. Удалить бесполезные символы (см. раздел 1.6.1), поскольку они могут появиться при выполнении шагов 1 и 2.

*Шаг 4*. Для продукций вида *A* → α, ⎪α⎪> 1, правые части которых включают в себя подстроки терминалов, каждому терминалу *a* из правой части поставить в соответствие новый нетерминал *Aa* и новую продукцию *Aa* → *a*. В результате правая часть такой продукции будет состоять только из нетерминалов.

*Шаг 5*. Для продукций вида *A* → *B*1*B*2…*Bm*, *m* > 2, поставить в соответствие совокупность продукций вида

*A* → *B*1*B'*1, *B'*1 → *B*2*B'*2, …, *B'm*–1 → *Bm*–1*Bm*,

где *B'*1, *B'*2, …, *B'm*–1 – новые нетерминалы, не содержащиеся более ни в одной продукции. Заметим, что *B*1*B*2…*Bm* после выполнения шага 4 могут быть только нетерминалами.

Рассмотрим в качестве примера грамматику со следующими продукциями:

*S* → *ASB*⎪ε

*A* → *aA*⎪ε

*B* → *bB*⎪*b*

Шаг 1. Удаление ε-продукций

*S'* → ε⎪*S*

*S* → *ASB*⎪*SB*⎪*AB*⎪*B*

*A* → *aA*⎪*a*

*B* → *bB*⎪*b*

Шаг 2. Удаление цепных продукций

*S'* → ε⎪*ASB*⎪*SB*⎪*AB*⎪*bB*⎪*b*

*S* → *ASB*⎪*SB*⎪*AB*⎪*bB*⎪*b*

*A* → *aA*⎪*a*

*B* → *bB*⎪*b*

Шаг 3. Бесполезных символов нет.

Шаг 4. Замена терминалов на нетерминалы в продукциях вида *A* → α, ⎪α⎪> 1

*S'* → ε⎪*ASB*⎪*SB*⎪*AB*⎪*AbB*⎪*b*

*S* → *ASB*⎪*SB*⎪*AB*⎪*AbB*⎪*b*

*A* → *AaA*⎪*a*

*B* → *AbB*⎪*b*

*Aa* → *a*

*Ab* → *b*

Шаг 5. Замена продукций вида *A* → *B*1*B*2…*Bm*, *m* > 2 на совокупность продукций. Имеем две такие продукции: *S'* → *ASB* и *S* → *ASB*. Добавим новый нетерминал *D* для обозначения подстроки *SB* и продукцию *D* → *SB*. В результате получим

*S'* → ε⎪*AD*⎪*SB*⎪*AB*⎪*AbB*⎪*b*

*S* → *AD*⎪*SB*⎪*AB*⎪*AbB*⎪*b*

*A* → *AaA*⎪*a*

*B* → *AbB*⎪*b*

*Aa* → *a*

*Ab* → *b*

*D* → *SB*

### Нормальная форма Грейбах

КС-грамматика *G* = (*VT*, *VN*, *P*, *S*) называется грамматикой в *нормальной форме Грейбах* (Sheila Greibach), если множество продукций содержит продукции только следующего вида:

*A* → *a*β, где *A*∈ *VN* , *a* ∈ *VT*, β ∈ *VN*\*;

*S* → ε, если ε ∈ *L*(*G*), причем *S* не должен встречаться в правых частях других продукций.

В ряде источников иногда на строку β накладывают ограничение ⎪β⎪≤ 2, т. е. β представляет собой строку из не более, чем двух нетерминалов. Это продукции вида *A* → *a*, *A* → *aB*, *A* → *aBC*, где *A*, *B*, *C* ∈ *VN*, *a* ∈ *VT*. Данный вариант мы рассматривать не будем из-за достаточно сложного формального алгоритма преобразования к такой форме.

Существует альтернативное определение нормальной формы Грейбах, в котором β ∈ (*VT* ∪ *VN*)\*, т. е. β – произвольная строка терминалов и нетерминалов (включая и строку длины 0). Данное определение ослабляет ограничение на строку β, поэтому ряд авторов называют его как *ослабленная нормальная форма Грейбах*.

Следует обратить внимание на то, что КС-грамматика в нормальной форме Грейбах не имеет левой рекурсии.

Любая КС-грамматика может быть преобразована в эквивалентную КС-грамматику в нормальной форме Грейбах. Преобразование представлено алгоритмом 1.8.

**Алгоритм 1.8.** Преобразование КС-грамматики к нормальной форме Грейбах

**Вход:** КС-грамматика *G* = (*VT*, *VN*, *P*, *S*)

**Выход:** КС-грамматика *G'* = (*VT*, *V'N*, *P'*, *S'*) в нормальной форме Грейбах

*Шаг 1*. Удалить ε-продукции (см. раздел 1.6.6).

*Шаг 2*. Устранить левую рекурсию (см. разделы 1.6.3 и 1.6.4). В результате все продукции (кроме *S* → ε, если она есть) будут иметь вид *Ai* → *a*β или *Ai* → *Aj*β, *i* < *j*, где *Ai*, *Aj* ∈ *VN*, *a* ∈ *VT*, β ∈ (*VT* ∪ *VN*)\*.

*Шаг 3*. Определить частичный линейный порядок на множестве *VN* в соответствии со следующим правилом: если существует продукция вида *A* → *B*α, где *A*, *B*∈ *VN*, α ∈ (*VT* ∪ *VN*)\*, то *A* < *B* (если этот порядок не определен на шаге 2). Пусть *VN* = {*A*1, *A*2, …, *Am*}. Тогда все *Am*-продукции имеют вид *Am* → α1, α2, …, α*k*, причем каждая строка α1, α2, …, α*k* начинается с терминала.

*Шаг 4*. *i* := *m* – 1.

*Шаг 5*. Если *i* > 0, для продукций вида *Ai* → *Aj*β, *j* > *i* выполнить замену вхождений нетерминала *Aj* в соответствии с продукциями *Aj* → α1, α2, …, α*k* (каждая строка α1, α2, …, α*k* начинается с терминала), получив продукции *Ai* → α1β⎪α2β⎪…⎪α*k*β. В противном случае (если *i* = 0) перейти к шагу 7.

*Шаг 6*. *i* := *i* – 1 и вернуться к шагу 5.

*Шаг 7*. Построена эквивалентная КС-грамматика в ослабленной нормальной форме Грейбах. Осталось выполнить требование β ∈ *VN*\*. В каждой продукции вида *A* → *a*β, в которой строка β включает в себя подстроки терминалов, каждому терминалу *b* из строки β поставить в соответствие новый нетерминал *Ab* и новую продукцию *Ab* → *b*.

Рассмотрим данное преобразование на примере. Воспользуемся грамматикой, не содержащей левую рекурсию (шаги 1 и 2 алгоритма выполнены):

*E* → *T*⎪*TE'*

*E'* → + *T*⎪+ *TE'*

*T* → *F*⎪*FT'*

*T'* → × *F*⎪× *FT'*

*F* → (*E*)⎪*i*

Шаг 3. Упорядочим нетерминалы. В соответствии с продукциями *E* → *T*⎪*TE'*, имеем *E* < *T*. Из продукций *T* → *F*⎪*FT'* следует, что *T* < *F*. Больше продукций вида *A* → *B*α нет. Для нетерминалов *E'* и *T'* порядок не определен, поэтому считаем, что они находятся в отношении < с остальными нетерминалами. В результате получаем порядок *E* < *T* < *F* < *E'* < *T'*. Заметим, что правая часть каждой продукции с нетерминалами *T'*, *E'* и *F* в левой части начинается с терминала.

Шаги 4, 5, 6. Для продукций *T* → *F*⎪*FT'* выполним замену вхождений нетерминала *F*:

*T* → (*E*)⎪*i*⎪(*E*)*T'*⎪*iT'*

Для продукций *E* → *T*⎪*TE'* выполним замену вхождений нетерминала *T*:

*E* → (*E*)⎪*i*⎪(*E*)*T'*⎪*iT'*⎪(*E*)*E'*⎪*iE'*⎪(*E*)*T'E'*⎪*iT'E'*

В результате получим эквивалентную грамматику в ослабленной нормальной форме Грейбах:

*E* → (*E*)⎪*i*⎪(*E*)*T'*⎪*iT'*⎪(*E*)*E'*⎪*iE'*⎪(*E*)*T'E'*⎪*iT'E'*

*E'* → + *T*⎪+ *TE'*

*T* → (*E*)⎪*i*⎪(*E*)*T'*⎪*iT'*

*T'* → × *F*⎪× *FT'*

*F* → (*E*)⎪*i*

Шаг 7. В продукциях

*E* → (*E*)⎪(*E*)*T'*⎪(*E*)*E'*⎪(*E*)*T'E'*

*T* → (*E*)⎪(*E*)*T'*

*F* → (*E*)

терминалу ) сопоставим нетерминал *B* и добавим соответствующую новую продукцию:

*E* → (*EB*⎪*i*⎪(*EBT'*⎪*iT'*⎪(*EBE'*⎪*iE'*⎪(*EBT'E'*⎪*iT'E'*

*E'* → + *T*⎪+ *TE'*

*T* → (*EB*⎪*i*⎪(*EBT*'⎪*iT*'

*T'* → × *F*⎪× *FT'*

*F* → (*EB*⎪*i*

*B* → )

## Свойство самовложения КС-грамматик

Если в КС-грамматике существует нетерминал *A*, для которого , α, β ∈ (*VT* ∪ *VN*)+, то о такой грамматике говорят, что она содержит *самовложение*. Заметим, что строки α и β являются непустыми строками.

Свойство самовложения позволяет эффективно различать контекстно-свободные (нерегулярные) и регулярные языки. Теоретически любая КС-грамматика, не содержащая самовложения, эквивалентна регулярной грамматике и генерирует регулярный язык. Регулярная грамматика не может содержать самовложения.

Пример грамматики, содержащей самовложение

*S* → *aSb*, *S* → ε

Грамматика арифметических выражений

*E* → *E* + *T*⎪*T*

*T* → *T* × *F*⎪*F*

*F* → (*E*)⎪*i*

также содержит самовложение, поскольку, например, существует вывод *E* ⇒ *T* ⇒ *F* ⇒ (*E*). В таких случаях говорят, что нетерминал *E* проявляет свойство самовложения. Нетерминалы *T* и *F* также проявляют свойство самовложения:

*T* ⇒ *F* ⇒ (*E*) ⇒ (*T*)

*F* ⇒ (*E*) ⇒ (*T*) ⇒ (*F*)

## Лемма Огдена и лемма о разрастании для КС-языков

*Лемма Огдена*. Для любой КС-грамматики *G* = (*VT*, *VN*, *P*, *S*) существует такая константа *k* ≥ 1, что если *z* ∈ *L*(*G*), ⎪*z*⎪≥ *k* и для любых выделенных в *z* не менее *k* позиций, строку *z* можно записать в виде *z* = *uvwxy*, причем:

1) *w* содержит хотя бы одну выделенную позицию;

2) либо и *u* и *v* содержат выделенные позиции, либо её содержат и *x* и *y*;

3) *vwx* содержит не более *k* выделенных позиций;

4) существует такой нетерминал *A*∈ *VN* , что

 для всех *i* ≥ 0.

Следствием леммы Огдена является *лемма о разрастании* (*лемма о накачке*, *лемма о подкачке*; англ. *pumping lemma*) для КС-языков.

Для любого КС-языка *L* существует такая константа *k* ≥ 1, что любую строку *z* ∈ *L*, ⎪*z*⎪≥ *k*, можно записать в виде *z* = *uvwxy*, где ⎪*vwx*⎪≤ *k*, *vx* ≠ ε, и для любого *i* ≥ 0 справедливо *uviwxiy* ∈ *L*.

Лемма Огдена и лемма о разрастании для КС-языков полезны для доказательства утверждений о том, что некоторые языки не являются контекстно-свободными.

Для примера рассмотрим язык *L* = {αα | α ∈ {*a*, *b*}\*}. Доказательство проведем методом от противного, используя лемму о разрастании. Предположим, что *L* – КС-язык. Рассмотрим строку *z* = *akbkakbk*, где *k* – константа из леммы о разрастании. Поскольку *z* ∈ *L*, ее можно представить в виде *z* = *uvwxy*, где ⎪*vwx*⎪≤ *k*, тогда строка *uviwxiy* (*i* ≥ 0) также должна принадлежать языку. Проверим для *i* = 0, т. е. будет ли *uwy* принадлежать языку. Возможны следующие варианты:

1. *v* и *x* находятся либо в первой половине *z*, либо во второй. В этом случае строка *uwy* образуется путем исключения символов *a* и *b* из первой или второй половины *z*, но не из обеих. Следовательно, *uwy* ∉ *L*.

2. *v* содержит символы *b* из первой половины *z*, *x* содержит символы *a* из второй половины z. В этом случае *uwy* образуется путем исключения символов *b* из первой половины и символов *a* из второй половины, т. е. полученная строка *uwy* ∉ *L*.

Таким образом, данная строка не может быть представлена в виде *uvwxy*, чтобы *uwy* тоже принадлежала языку. Поэтому язык *L* = {αα | α ∈ {*a*, *b*}\*} не является КС-языком.

Упражнения

1.1. Построить КС-грамматики, порождающие следующие языки:

а) {*ambnck* | *m*, *n*, *k* ≥ 0};

б) {*anbnck* | *n*, *k* ≥ 0};

в) {*anbcn* | *n* ≥ 0};

г) все строки из множества {*a*, *b*}\*, такие, что в каждой из них непосредственно справа от каждого символом *a* следует символ *b*;

д) все строки из множества {*a*, *b*}\*, такие, что результаты чтения этих строк слева направо и справа налево совпадают;

е) все строки из множества {*a*, *b*}\*, которые содержат вдвое больше символов *a*, чем символов *b*;

ж) все строки из множества {*a*, *b*}\*, которые содержат равное число символов *a* и *b*.

1.2. Дана грамматика

*E* → *E* + *T*⎪*T*

*T* → *T* × *F*⎪*F*

*F* → (*E*)⎪*i*

Построить левостороннюю и правостороннюю схемы вывода и соответствующие деревья разбора для следующих строк:

а) *i* + *i* × *i* + *i* × *i* + *i* г) *i* + (*i* × *i* + *i* × *i* )+ *i*

б) *i* + *i* × (*i* + *i*) × (*i* + *i*) д) ((*i* + *i*) × *i* + *i*) × *i* + *i*

в) (*i* + *i* × *i*) + *i* × (*i* + *i*) е) ((*i* + *i*) × *i* + *i*) × (*i* + *i*)

1.3. Показать, что грамматика *E* → *E* + *E*⎪*E* × *E*⎪(*E*)⎪*i* неоднозначна. Для этого построить различные левосторонние схемы вывода и соответствующие деревья разбора для какой-либо строки языка.

1.4. Устранить левую рекурсию из следующих грамматик:

а) *S* → *SaA*⎪*AA*⎪*b* в) *S* → *Ab*

*A* → *ASa*⎪*Ad*⎪*c* *A* → *Sa*⎪*cB*

*B* → *bS*⎪*c*

б) *S* → *AB*⎪*a* г) *S* → *Ba*⎪*Ab*

*A* → *BS*⎪*Sb* *A* → *Sa*⎪*AAb*⎪*a*

*B* → *SA*⎪*BB*⎪*a* *B* → *Sb*⎪*BBa*⎪*b*

1.5. Из следующих грамматик удалить ε-продукции (если ε принадлежит языку, допускается единственная ε-продукция вида *S* → ε, где *S* – начальный символ грамматики):

а) *S* → *ABa*⎪*aAb* в) *S* → *AB*

*A* → *aA*⎪*ab*⎪ε *A* → *SA*⎪*BB*⎪*bB*

*B* → *Ba*⎪*ba*⎪ε *B* → *b*⎪*aA*⎪ε

б) *S* → *TC* г) *S* → *ASB*⎪ε

*T* → *aTb*⎪ε *A* → *aAS*⎪*a*

*C* → *cC*⎪ε *B* → *SbS*⎪*A*⎪*bb*

1.6. Из следующих грамматик удалить цепные продукции:

а) *S* → *AC* в) *S* → *abAd*

*A* → *B*⎪*AaB* *A* → *B*⎪*BeA*

*B* → *i* *B* → *fD*

*C* → *D*⎪*DaC D* → *t*⎪*teD*

*D* → *i*

б) *S* → *A*⎪*B* г) *S* → *bA*⎪*Ba*

*A* → *bAa*⎪*bca* *A* → *bA*⎪*C*

*B* → *bBaa*⎪*bdaa* *B* → *Ba*⎪*C*

*C* → *bCa*⎪ε

1.7. КС-грамматики, построенные в упр. 1.1, преобразовать в нормальную форму Хомского.

1.8. КС-грамматики, построенные в упр. 1.1, преобразовать в нормальную форму Грейбах.

1.9. Используя лемму о разрастании, доказать, что следующие языки не являются КС-языками:

а) {*anbncn* | *n* ≥ 1} в) {*ak* | *k* – простое число}

б) {*ambnambn* | *m*, *n* ≥ 1} г) {*ambmcn* | *m*, *n* ≥ 1, *n* < *m*}.

# ЛЕКСИЧЕСКИЙ АНАЛИЗ

## Задачи лексического анализа

Лексический анализ (сканирование) является первой фазой компиляции. Реализуется частью компилятора, которая называется *лексическим анализатором* (или *сканером*). Его основная задача состоит в предварительной обработке исходного текста программы, которая заключается в группировании символов входного потока в лексические единицы (*лексемы*). Для каждой лексемы сканер формирует выходной *токен* вида <*код\_токена*, *атрибут*> для последующих фаз компиляции. *Код\_токена* идентифицирует класс лексемы (*лексический класс*) и определяет работу синтаксического анализатора (рассматривается как терминал). Для удобства *код\_токена* будем представлять абстрактным именем (или специальным обозначением), выделенным жирным шрифтом, и ссылаться на токен по его имени (обозначению). *Атрибут* токена обеспечивает доступ к дополнительной информации о лексеме, если лексическому классу соответствует множество лексем, и определяет трансляцию токена (семантический анализ и генерация промежуточного кода).

Часто фазы лексического и синтаксического анализа объединяют в один проход. В этом случае лексический анализатор является подпрограммой синтаксического анализатора (рис. 2.1). Когда синтаксическому анализатору требуется очередной токен, он вызывает лексический анализатор, который формирует очередной токен и возвращает управление синтаксическому анализатору. В результате исходная программа полностью преобразуется в последовательность токенов.



Рис. 2.1. Взаимодействие лексического и синтаксического анализаторов

Лексический анализатор выполняет также и другие функции. В частности, он удаляет из текста исходной программы комментарии и не несущие смысловой нагрузки пробелы, символы табуляции и символы перевода строки. Еще одной задачей является согласование сообщений об ошибках компиляции и текста исходной программы (указать каким-либо образом позицию ошибки и ее характер в тексте программы). Кроме того, лексический анализатор должен строить различные таблицы, необходимые как для собственно лексического анализа, так и для последующих фаз компиляции.

## Лексические классы

Множество лексем разбивается на непересекающиеся подмножества лексических классов, неразличимых с точки зрения синтаксического анализа. Каждый лексический класс описывается соответствующими правилами (*шаблон* токена). В большинстве языков программирования токенами являются ключевые слова, идентификаторы, константы, символы операций, символы пунктуации (скобки, запятые и т. д.). Токену может соответствовать единственная лексема (по одному токену для каждого ключевого слова, символа операции и символа пунктуации), конечное или бесконечное множество лексем (идентификаторы, константы, сгруппированные в один токен наборы операций). Для формального описания шаблонов токенов используются регулярные грамматики или регулярные выражения.

Многие языки используют определенные заранее лексемы в качестве указания конкретных конструкций языка или специальных символов пунктуации (**begin**, **end**, **while**, **do** и т. д.). Такие лексемы называются *ключевыми словами* и обычно удовлетворяют правилам образования идентификаторов. Поэтому необходим специальный механизм, позволяющий отличить ключевые слова от других идентификаторов. Для упрощения решения этой проблемы во многих языках ключевые слова *зарезервированы*, т. е. они не могут использоваться в качестве идентификаторов. Тогда лексема является идентификатором только в том случае, если она не является ключевым словом.

Атрибутом токенов, которым может соответствовать бесконечное множество лексем (идентификаторы, константы), является указатель на соответствующую запись в таблице символов, в которой хранится информация о токене. Если шаблону токена соответствует конечное множество лексем (например, символы операций сравнения), то атрибутом может быть не указатель на соответствующую запись в специальной таблице операций сравнения, а соответствующий код операции. В этом случае отпадает необходимость явно хранить эту таблицу операций. Если шаблону токена соответствует единственная лексема, то атрибут имеет пустое значение (будем обозначать символом 0).

Выделение лексических классов в языках программирования обусловлено в первую очередь эффективностью выполнения последующих фаз компиляции. Например, набор всех шести операций сравнения можно сгруппировать в один токен (чаще всего так и делается), а можно каждой операции сопоставить свой токен. В первом случае токен рассматривается как единственный терминал при синтаксическом анализе для любой операции сравнения, а атрибут дает информацию о семантике операции сравнения для трансляции. Во втором случае имеется шесть токенов (шесть терминалов для синтаксического анализа) и сам же токен несет информацию о семантике операции.

Рассмотрим фрагмент исходной программы:

**for** *i* := 1 **to** 20 **do** *MyArr*[*i*] := 0;

Пусть в процессе формирования таблицы символов информация об идентификаторах *i* и *MyArr* оказалась в записях с номерами 3 и 7 соответственно, а о константах 1, 20 и 0 – в записях с номерами 2, 10 и 11 соответственно. Тогда сканер сформирует следующую последовательность токенов:

<**for**, 0>, <**id**, 3>, <**ass**, 0>, <**num**, 2>, <**to**, 0>,

<**num**, 10>, <**do**, 0>, <**id**, 7>, <**[**, 0>, <**id**, 3>, <**]**, 0>,

<**ass**, 0>, <**num**, 11>, <**;**, 0>.

Здесь имена токенов **for**, **to**, **do** обозначают соответствующие ключевые слова, **ass** – операцию присваивания, **id** – идентификатор, **num** – числовую константу, остальные токены обозначены соответствующими абстрактными символами **[**, **]**, **;**. Токены **for**, **to**, **do**, **ass**, **[**, **]**, **;** имеют пустые значения атрибутов, поскольку для каждого из них в качестве шаблона определена единственная лексема. Атрибутами токенов **id** и **num** являются указатели на соответствующие записи в таблице символов, так как им может соответствовать бесконечное множество лексем.

## Таблица символов

Таблица символов представляет собой структуру данных, которая используется компилятором для хранения информации о конструкциях исходной программы. Структура данных должна обеспечивать компилятору возможность быстрого поиска нужной записи, а также возможность быстрого сохранения данных в записи и получения их из нее. Некоторые компиляторы формируют единую хеш-таблицу, что обеспечивает, по сути, константное время доступа к нужной записи.

В ряде случаев таблицу символов удобно реализовать с помощью нескольких отдельных таблиц, например, таблица ключевых слов, таблица идентификаторов, таблица констант. Очевидно, что в этом случае таблица ключевых слов является статической и ее содержимое не изменяется в процессе компиляции (носит константный характер). Таблицы идентификаторов и констант являются динамическими, для них нужна структура данных, обеспечивающая наибольшую эффективность работы (вплоть до организации хеш-таблиц). Числовые константы перед помещением их в таблицу могут переводиться из внешнего символьного представления во внутреннее машинное представление.

В процессе лексического анализа формируются начальные элементы таблицы символов для хранения информации об объектах. По мере выполнения других фаз компиляции таблица дополняется новыми данными. В общем случае информация, хранимая в таблице символов, зависит от семантики входного языка и вида объекта. Например, для имени переменной может храниться ее лексема, тип (вещественный, целый и т.д.), точность, длина, адрес памяти, число измерений и значения граничных пар (для массивов); для имени функции – количество и типы формальных параметров, тип возвращаемого результата, адрес вызова кода функции и т.п.

Если лексема распознается как идентификатор, то осуществляется ее поиск в таблице символов, если поиск безуспешный, лексема добавляется в таблицу. Заметим, что поиск и добавление идентификаторов в таблицу символов осуществляется сканером. Другие фазы компиляции имеют прямой доступ к нужной записи через атрибут соответствующего токена.

Во многих языках программирования имеют место предопределенные идентификаторы (имена стандартных типов, процедур, функций), которые не являются ключевыми словами. Такие идентификаторы должны быть занесены в таблицу символов заранее.

Многие языки программирования имеют структуру вложенных блоков и процедур, когда один и тот же идентификатор может быть объявлен и использован по-разному в различных блоках и процедурах. В этом случае важным становится понятие области видимости объявлений. *Область видимости* объявления представляет собой часть программы, в которой может применяться данное объявление.

Можно реализовать области видимости путем использования отдельной таблицы символов для каждой области видимости, т.е. программный блок с объявлениями будет иметь собственную таблицу символов с данными для каждого объявления в блоке. При выходе из блока соответствующая таблица символов может быть удалена (если она не требуется для последующих фаз компиляции).

Другой подход заключается в применении одной таблицы символов для всех блоков. В этом случае данные об идентификаторе дополняются номером блока, т.е. один и тот же идентификатор с различными номерами блоков будут иметь отдельную запись в таблице символов и рассматриваться как разные идентификаторы.

Блочную структуру программы можно распознать только при выполнении фазы синтаксического анализа. Поэтому синтаксический анализатор при запросе следующего токена должен предоставить лексическому анализатору номер блока.

## Распознавание токенов

Основной задачей сканера является чтение входных символов исходной программы с целью их группирования в лексемы (которые с точки зрения языка должны рассматриваться как единое целое), представляющие соответствующие токены. Если задан шаблон, описывающий лексему токена, распознавание токена выполняется достаточно просто. Однако возникает сложность распознавания, если лексема одного токена может быть префиксом лексемы другого токена. В таких случаях для корректного распознавания потребуется чтение дополнительных символов входного потока, следующих за текущим символом (*опережающее чтение*).

Например, если в языке в качестве разделителя используется символ '**:**', а в качестве операции присваивания используется обозначение '**:=**', то, встретив символ '**:**', необходимо прочитать следующий за ним символ. Если он окажется символом '**=**', то получена лексема '**:=**' токена «операция присваивания». В противном случае символ '**:**' является лексемой токена «двоеточие», при этом прочитан один дополнительный символ входного потока, который следует вернуть во входной поток (этот символ может являться началом другой лексемы).

Приведем еще один пример. Рассмотрим синтаксически правильные входные строки '3.75' и '3..75'. Первая строка образует одну лексему, относящуюся к токену «числовая константа», а вторая – три лексемы: '3', '**..**', '75', определяющие конструкцию «диапазон», где лексемы '3' и '75' являются числовыми константами. Чтобы распознать лексему '3' как токен «числовая константа», требуется прочитать два дополнительных символа (две точки), которые следует вернуть во входной поток.

Для облегчения реализации опережающего чтения и возврата символов во входной поток рекомендуется использовать входной буфер, из которого лексический анализатор может выполнять чтение и в который может возвращать прочитанные символы путем простого перемещения указателя. Использование входного буфера повышает также эффективность анализатора, так как считывание блока символов обычно существенно более эффективно, чем посимвольное считывание.

Можно использовать следующую схему буферизации [1], которая включает два по очереди загружаемых буфера (рис. 2.2).



Рис. 2.2. Схема буферизации входного потока

Оба буфера имеют один и тот же размер *N*, причем *N* обычно равно размеру дискового блока, например 4096 байт. При этом можно считать *N* символов в буфер одной командой чтения, не используя системный вызов для каждого символа по отдельности. Если во входном файле осталось менее *N* символов, конец исходного файла маркируется специальным символом конца файла **eof**, отличным от любого из возможных символов исходной программы. Предусматриваются два указателя: указатель *f*, маркирующий начало текущей обрабатываемой лексемы, и указатель *t*, который сканирует символы до тех пор, пока выполняется соответствие шаблону.

Как только определена очередная лексема, указатель *t* устанавливается таким образом, чтобы указывать на символ, являющийся ее правым концом. Затем, после того как определен токен для лексемы, передаваемый синтаксическому анализатору, указатель *f* устанавливается на символ, непосредственно следующий за только что обнаруженной лексемой. На рис. 2.2 указатель *t* указывает на символ за концом текущей лексемы Count (идентификатор переменной) и должен быть перемещен на одну позицию влево (возврат дополнительно прочитанного символа во входной поток).

При перемещении указателя *t* первоначально необходимо выполнить проверку, не достигнут ли конец одного из буферов, и, если достигнут, заполнить другой буфер символами из входного потока, и перенести *t* в начало этого только что заполненного буфера.

Таким образом, для каждого считанного символа необходимо выполнить два сравнения: одно – не достигнут ли конец буфера, второе – какой именно символ считан. Однако проверку конца буфера можно совместить с проверкой считанного символа, если расширить каждый буфер для хранения в его конце специального ограничителя (сторожа). Ограничитель представляет собой специальный символ, который не может быть частью исходной программы.

Множество лексем, соответствующих некоторому токену, можно рассматривать как формальный язык этого токена (язык идентификаторов, язык констант, язык символов пунктуации, язык символов операций и т. д.). Языки токенов практически всегда относятся к классу регулярных языков, поэтому в качестве формального описания шаблонов токенов достаточно использовать регулярные грамматики или регулярные выражения.

Распознавателем регулярного языка является конечный автомат. Существуют процедуры построения (*синтеза*) конечного автомата для заданной регулярной грамматики или заданного регулярного выражения, которые будут рассмотрены ниже в соответствующих разделах.

При синтезе конечных автоматов необходимо учитывать следующую особенность. Вместо построения отдельных конечных автоматов для каждого ключевого слова можно использовать специальную таблицу ключевых слов. Если лексический анализатор распознает лексему как идентификатор, то он простым просмотром таблицы ключевых слов может определить, является эта лексема ключевым словом или нет.

После синтеза всех конечных автоматов-распознавателей для всех токенов наиболее предпочтительным подходом является объединение всех конечных автоматов в один конечный автомат, который выбирает наибольшую лексему, соответствующую некоторому шаблону. Тогда программная реализация лексического анализатора сводится к программной реализации этого конечного автомата с соответствующим формированием токенов и реализацией возврата символов во входной поток при опережающем чтении.

## Конечный автомат

Формально конечный автомат-распознаватель (КА) определяется как пятерка *M* = (*K*, *T*, δ, *k*0, *F*), где

*K* – конечное множество состояний;

*T* – конечный входной алфавит;

δ: *K* × *T* → 2*K* – функция переходов автомата, здесь 2*K* обозначает булеан (степень) множества *K* (множество всех подмножеств множества *K*, мощность булеана равна 2|*K*|);

*k*0 ∈ *K* – начальное состояние автомата;

*F*⊆ *K* – множество конечных (финальных, заключительных, принимающих) состояний.

Приведено определение *недетерминированного* конечного автомата (НКА), у которого значением функции переходов δ(*k*, *a*), *k* ∈ *K*, *a* ∈ *T*, может являться любое подмножество состояний из *K*. Если областью значений функции переходов является множество состояний *K*, то есть δ: *K* × *T* → *K*, автомат называется *детерминированным* (ДКА). В ДКА множество δ(*k*, *a*) содержит не более одного элемента для всех *k* ∈ *K* и *a* ∈ *T*.

Запись функции переходов δ(*ki*, *a*) = *kj* для ДКА означает, что при чтении входного символа *а* ∈ *T* в текущем состоянии *ki* ∈ *K* автомата осуществляется переход в состояние *kj* ∈ *K*. В НКА при чтении символа *a* в текущем состоянии *ki* возможен переход в любое из нескольких альтернативных состояний из множества δ(*ki*, *a*).

Функцию переходов можно представить в виде графа (диаграммы) переходов или таблицы переходов.

Граф переходов автомата представляет собой ориентированный граф, в котором вершины помечаются состояниями автомата, а дуги – символами входного алфавита. Начальное состояние будем выделять небольшой входной стрелкой, а конечные состояния будем изображать в виде прямоугольника.

Таблица переходов является обычным матричным представлением функции переходов δ. Формально, согласно определению функции δ: *K* × *T* → 2*K*, в ее матричном представлении строки должны соответствовать состояниям, а столбцы – символам входного алфавита. Но часто удобнее представлять эту матрицу в транспонированном виде, а именно: строки соответствуют символам входного алфавита, а столбцы – состояниям автомата, на пересечении строки и столбца записывается соответствующее значение функции. Для выделения конечных состояний будем заключать их в квадратные скобки. Первый столбец всегда соответствует начальному состоянию.

Рассмотрим НКА *M* = ({*k*0, *k*1, *k*2, *kf*}, {*a*, *b*}, δ, *k*0, {*kf*}) со следующей функцией переходов δ:

δ(*k*0, *a*) = {*k*1, *kf*}; δ(*k*2, *a*) = {*k*2, *kf*};

δ(*k*0, *b*) = ∅; δ(*k*2, *b*) = ∅;

δ(*k*1, *a*) = {*k*1, *kf*}; δ(*kf*, *a*) = ∅;

δ(*k*1, *b*) = {*k*2}; δ(*kf*, *b*) = ∅.

На рис. 2.3. приведено представление функции переходов δ в виде графа переходов, а на рис. 2.4 – в виде таблицы переходов.

*a*

*k*0

*k*1

*a*

*k*2

*b*

*kf*

*a*

*a*

*a*

*a*

Рис. 2.3. Граф переходов НКА

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *k*0 | *k*1 | *k*2 | [*kf*] |
| *a* | {*k*1, *kf*} | {*k*1, *kf*} | {*k*2, *kf*} | ∅ |
| *b* | ∅ | {*k*2} | ∅ | ∅ |

Рис. 2.4. Таблица переходов НКА из рис. 2.3

По мере чтения символов входной строки КА переходит из состояния в состояние в соответствии с функцией переходов. Если после чтения последнего символа входной строки КА находится в одном из конечных состояний, то говорят, что строка принимается автоматом. В противном случае строка не принадлежит языку, принимаемому автоматом.

Расширим определение функции переходов на строку, определив ее как функцию δ: *K* × *T*\* → *K* (для НКА δ: *K* × *T*\* → 2*K*). Тогда δ(*k*, *x*) – состояние, которое будет достигнуто из состояния *k* после ввода строки *x* ∈ *T*\*. Если *x* = *ay* (*a* ∈ *T*, *y* ∈ *T*\*), то расширенную функцию δ можно определить рекурсивно: δ(*k*, *x*) = δ(δ(*k*, *a*), *y*).

Таким образом, множество строк *T*(*M*)⊆ *T*\*, принимаемых автоматом *M*, можно определить следующим образом:

*T*(*M*)= {*x* ∈ *T*\* | δ(*k*0, *x*) ∈ *F*}.

Использование НКА в качестве распознавателя приводит к существенным потерям времени при лексическом анализе, поскольку приходится перебирать все возможные варианты. Поэтому следует использовать всегда ДКА. К счастью, каждому НКА можно поставить в соответствие ДКА, принимающий тот же язык.

Поскольку множество всех подмножеств множества *K* конечно и равно 2|*K*|, то каждое подмножество можно рассматривать как отдельное состояние, т. е. получится 2|*K*| возможных состояний детерминированного автомата. На практике, чтобы не рассматривать все 2|*K*| возможных состояний, ограничиваются рассмотрением множества тех состояний, которые могут быть достигнуты из начального состояния {*k*0}, т. е. *K'* = δ({*k*0}, *a*) для всех *a* ∈ *T*. Далее рассматривается множество состояний δ(*K'*, *a*) для всех *a* ∈ *T*, *K'* ⊆ *K*. Процесс продолжается до тех пор, пока полученные таким образом состояния не начнут совпадать с уже существующими состояниями автомата. В качестве множества конечных состояний полученного ДКА выбирается такое множество *K'* ⊆ *K*, что *K'* ∩ *F* ≠ ∅, т. е. множество *K'*, образующее состояние ДКА, содержит хотя бы одно из конечных состояний исходного НКА.

Процесс определения функций переходов ДКА, эквивалентного НКА на рис. 2.3, представлен на рис. 2.5.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | {*k*0} | [{*k*1, *kf*}] | {*k*2} | [{*k*2, *kf*}] |
| *a* | {*k*1, *kf*} | {*k*1, *kf*} | {*k*2, *kf*} | {*k*2, *kf*} |
| *b* | ∅ | {*k*2} | ∅ | ∅ |

Рис. 2.5. Определение функций переходов ДКА

Из начального состояния {*k*0} достижимо состояние {*k*1, *kf*} при чтении входного символа *a*. Поэтому добавляется соответствующий столбец. Определим состояния, в которые возможен переход из {*k*1, *kf*}:

δ({*k*1, *kf*}, *a*) = δ(*k*1, *a*) ∪ δ(*kf*, *a*) = {*k*1, *kf*} ∪ ∅ = {*k*1, *kf*};

δ({*k*1, *kf*}, *b*) = δ(*k*1, *b*) ∪ δ(*kf*, *b*) = {*k*2} ∪ ∅ = {*k*2}.

Появилось новое состояние {*k*2} (добавляем в таблицу соответствующий столбец). Аналогично, из состояния {*k*2} возможен переход в новое состояние {*k*2, *kf*}, из которого достижимо только состояние {*k*2, *kf*}. Новые состояния не появились, процесс завершен. Конечными состояниями будут состояния {*k*1, *kf*} и {*k*2, *kf*}, поскольку они содержат конечное состояние *kf* ∈ *F* исходного НКА.

Состояние, соответствующее пустому множеству играет роль фиктивного состояния. С позиции лексического анализа это означает наличие лексической ошибки и может быть выделено как специальное состояние «ошибка». Это состояние в действительности может быть опущено, в результате чего будет получена неполная функция переходов.

Далее можно пронумеровать полученные состояния неотрицательными целыми числами (начальному состоянию дадим номер 0): {*k*0} – 0, {*k*1, *kf*} – 1, {*k*2} – 2, {*k*2, *kf*} – 3. В результате получится таблица переходов (рис. 2.6) и граф переходов (рис. 2.7) ДКА, эквивалентного исходному НКА.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | [1] | 2 | [3] |
| *a* | 1 | 1 | 3 | 3 |
| *b* |  | 2 |  |  |

Рис. 2.6. Таблица переходов ДКА

*a*

*a*

*b*

0

2

1

*a*

*a*

3

Рис. 2.7. Граф переходов ДКА

## Регулярные грамматики

Регулярные (или автоматные) грамматики – это грамматики типа 3 в иерархии Хомского. Выделяют классы праволинейных и леволинейных регулярных грамматик.

Регулярная грамматика *G* = (*VT*, *VN*, *P*, *S*) называется *праволинейной*, если все ее продукции имеют вид *А*→ *а* или *А*→ *аВ*, где *а*∈*VT*, *А*, *В*∈*VN*.

Регулярная грамматика *G* = (*VT*, *VN*, *P*, *S*) называется *леволинейной*, если все ее продукции имеют вид *А*→ *а* или *А*→ *Ва*, где *а*∈*VT*, *А*, *В*∈*VN*.

Если пустая строка принадлежит языку, допускается единственная ε-продукция вида *S*→ ε (*S* – начальный символ грамматики), при этом ни одна продукция грамматики не должна содержать нетерминал *S* в своей правой части. Следует заметить, что на практике при проектировании лексических анализаторов для формального описания шаблонов токенов обычно используют ε-свободные регулярные грамматики, поскольку лексемы не могут быть пустыми.

В связи с тем, что классы праволинейных и леволинейных регулярных грамматик эквивалентны, в дальнейшем будем рассматривать праволинейные регулярные грамматики.

Поскольку в процессе вывода некоторой сентенциальной формы производится замена нетерминала на правую часть соответствующей продукции, которая для ε-свободных регулярных грамматик содержит не более одного нетерминала, то любая сентенциальная форма регулярной грамматики либо содержит только терминалы, либо содержит один нетерминал, являющийся самым правым символом.

Язык, порождаемый регулярной грамматикой, является регулярным языком. Для любого регулярного языка, существует порождающая его регулярная грамматика.

Если *L*1 и *L*2 – регулярные языки, то *L*1 ∪ *L*2 и *L*1*L*2 тоже являются регулярными языками. Пусть *L*1 = {*a*, *bb*}, *L*2 = {ε, *cc*, *d*}. Тогда их объединение *L*1 ∪ *L*2 = {ε, *a*, *bb*, *cc*, *d*}, конкатенация (сцепление) *L*1*L*2 = {*a*, *bb*, *acc*, *bbcc*, *ad*, *bbd*}. Эти утверждения являются теоретической базой для объединения языков токенов в один регулярный язык символов языка программирования.

Пример регулярной грамматики для идентификатора в классическом его определении как последовательности букв и цифр, начинающейся с буквы.

*S* → *lA*,

*A* → *lA* | *dA* | ⊥,

где терминалы *l* и *d* обозначают соответственно букву и цифру, нетерминал *A* – последовательность букв и/или цифр, терминал ⊥ ∉ {*l*, *d*} – маркер конца ввода лексемы идентификатора, то есть любой символ кроме буквы и цифры.

Процесс вывода строки *lddld*⊥ (например, индентификатор *x*21*a*8) соответствует следующей схеме вывода:

*S* ⇒ *lA* ⇒ *ldA* ⇒ *lddA* ⇒ *lddlA* ⇒ *lddldA* ⇒ *lddld*⊥.

Существует полное соответствие между регулярными грамматиками и конечными автоматами. Построение конечного автомата-распознавателя *M* = (*K*, *T*, δ, *k*0, *F*) по заданной регулярной грамматике *G* = (*VT*, *VN*, *P*, *S*), такое, что *L*(*G*) = *T*(*M*), заключается в следующем:

1. *T* = *VT*, т. е. входной алфавит КА совпадает с множеством терминалов регулярной грамматики.

2. *K* = *VN* ∪ {#}, где # – специально выделенное конечное состояние, т. е. множество состояний КА есть множество терминалов, дополненное выделенным конечным состоянием.

3. *k*0 = *S*, т. е. начальным состоянием КА является начальный нетерминал грамматики.

4. Если *S*→ ε ∈ *P*, то *F* = {*S*, #}, в противном случае *F* = {#}.

5. Продукции вида *А*→ *аВ* соответствует функция переходов δ(*A*, *a*) = *B*; продукции вида *А*→ *а* соответствует функция переходов δ(*A*, *a*) = #.

В общем случае построенный по регулярной грамматике автомат будет недетерминированным. Поэтому на следующем этапе полученный НКА необходимо преобразовать в эквивалентный ДКА (см. раздел 2.5).

Пусть дана регулярная грамматика *G* с продукциями

*S* → *d* | *dA*,

*A* → *d* | *dA* | *pB*,

*B* → *d* | *dB*.

Эта грамматика описывает синтаксис чисел целого типа (например, *ddd*) и вещественного типа с фиксированной точкой (например, *ddpddd*), где терминал *d* обозначает цифру, терминал *p* – точку.

Определим функцию переходов δ автомата:

δ(*S*, *d*) = {*A*, #}; δ(*B*, *d*) = {*B*, #};

δ(*S*, *p*) = ∅; δ(*B*, *p*) = ∅;

δ(*A*, *d*) = {*A*, #}; δ(#, *d*) = ∅;

δ(*A*, *p*) = {*B*}; δ(#, *p*) = ∅.

Граф переходов полученного НКА показан на рис. 2.8. Эквивалентный ему ДКА приведен на рис. 2.9.

При построении автомата не учтен следующий момент. После распознавания лексического класса принятой лексемы сканер передает синтаксическому анализатору сформированный токен, т. е. автомат-распознаватель должен остановиться, сформировать токен и перейти к распознаванию очередной лексемы с начального состояния. Это означает, что из конечного состояния автомата не должно быть переходов в какое-либо другое состояние, а переход в конечное состояние должен быть выполнен после завершения распознавания лексемы.

*d*

*d*

*d*

*d*

*d*

*p*

*d*

#

*S*

*A*

*B*

Рис. 2.8. Граф переходов НКА, построенный по регулярной грамматике

*d*

*d*

*p*

0

2

1

*d*

*d*

3

Рис. 2.9. Граф переходов ДКА, эквивалентного НКА на рис. 2.8

Для распознавания лексемы сканеру может потребоваться чтение дополнительных символов входного потока, следующих за текущим символом (опережающее чтение). Пусть автомат на рис. 2.9 находится в состоянии 1. Если очередным входным символом является символ *d* или *p*, то продолжается распознавание лексемы. Если же очередным входным символом будет любой другой символ ⊥1 ∉ {*d*, *p*}, то это означает, что завершено распознавание лексемы как числа целого типа. Аналогично, в состоянии 3 чтение любого символа ⊥2 ≠ *d* говорит о том, что лексема является числом вещественного типа. Символы ⊥1 и ⊥2 можно трактовать как маркеры конца ввода лексемы. Таким образом, для распознавания лексемы как числа требуется чтение дополнительного символа входного потока, этот символ следует вернуть во входной поток (символ может являться началом другой лексемы). Действия, связанные с определением токена и возврата лишних прочитанных символов во входной поток, легко можно реализовать в конечных состояниях соответствующих автоматов-распознавателей.

Это требует следующей модификации автомата. Если конечное состояние имеет переход в другое состояние, его надо сделать обычным внутренним состоянием и добавить из него переход в новое конечное состояние по маркеру конца ввода. Рекомендуется конечные состояния нумеровать отрицательными целыми числами, чтобы легко отличать конечные состояния от других состояний автомата. Модификация автомата на рис. 2.9 представлена на рис. 2.10.

⊥1

*d*

*d*

*p*

0

2

–1

*d*

*d*

–2

1

3

⊥2

Рис. 2.10. Модифицированный ДКА

Другой путь построения автомата – включить необходимые маркеры конца ввода лексемы в исходный синтаксис лексического класса, построить соответствующую регулярную грамматику и синтезировать автомат. Например, для описания синтаксиса числа можно построить регулярную грамматику

*S* → *dA*,

*A* → *dA* | ⊥1 | *pB*,

*B* → *dC*,

*C* → *dC* | ⊥2.

Результатом построения автомата-распознавателя по этой грамматике (с последующей его детерминизацией) будет тот же автомат, что и на рис. 2.10.

Можно также, наоборот, по детерминированному автомату *M* = (*K*, *T*, δ, *k*0, *F*) построить регулярную грамматику *G* = (*VT*, *VN*, *P*, *S*):

1. Множество терминалов *VT* грамматики совпадает с входным алфавитом *T* автомата, т. е. *VT* = *T*.

2. Множеству нетерминалов *VN* соответствует множество *K* состояний автомата, т. е. *VN* = *K*. Если состояния автомата помечены их номерами, то лучше предварительно переобозначить их прописными латинскими буквами, чтобы получить множество нетерминалов в соответствии с принятым соглашением об их обозначениях.

3. Начальное состояние *k*0 автомата становится начальным нетерминалом *S* грамматики.

4. В множество *P* включаются все продукции вида *ki* → *akj*, если имеется функция переходов δ(*ki*, *a*) = *kj*, а также все продукции вида *ki* → *a*, где δ(*ki*, *a*) = *kj* и *kj* ∈ *F*.

В общем случае в полученной грамматике могут появиться непроизводящие нетерминалы (т. е. нетерминалы, не порождающие ни одной терминальной строки). Такие нетерминалы соответствуют состояниям автомата, из которых нет переходов в другие состояния. Продукции с такими нетерминалами можно безболезненно удалить из грамматики.

Построим регулярную грамматику по автомату на рис. 2.10. Обозначим состояния латинскими буквами:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Номер состояния | 0 | 1 | 2 | 3 | –1 | –2 |
| Обозначение | *S* | *A* | *B* | *C* | *D* | *E* |

По функции переходов получим продукции:

δ(*S*, *d*) = *A* ⇒ *S* → *dA*

δ(*A*, *d*) = *A* ⇒ *A* → *dA*

δ(*A*, *p*) = *B* ⇒ *A* → *pB*

δ(*A*, ⊥1) = *D* ⇒ *A* → ⊥1*D* | ⊥1

δ(*B*, *d*) = *C* ⇒ *B* → *dC*

δ(*C*, *d*) = *C* ⇒ *C* → *dC*

δ(*C*, ⊥2) = *E* ⇒ *C* → ⊥2*E* | ⊥2

Продукции *A* → ⊥1 и *C* → ⊥2 добавлены, поскольку состояния *D* и *E* являются конечными. В полученной грамматике нетерминалы *D* и *E* непроизводящие, поэтому эти нетерминалы и содержащие их продукции *A* → ⊥1*D* и *C* → ⊥2*E* можно удалить из грамматики. В результате получим регулярную грамматику

*S* → *dA*,

*A* → *dA* | ⊥1 | *pB*,

*B* → *dC*,

*C* → *dC* | ⊥2.

## Регулярные выражения

Альтернативой регулярным грамматикам для формального описания регулярных языков являются регулярные выражения. Регулярные выражения эквивалентны регулярным грамматикам и широко используются на практике.

Регулярное выражение *r* над алфавитом *T* описывает язык *L*(*r*), который рекурсивно определяется на основании языков, описываемых подвыражениями *r*.

Базисные регулярные выражения образованы тремя правилами:

1. Символ ∅, представляющий пустое множество, является регулярным выражением, а *L*(∅) представляет собой пустой язык.

2. Символ пустой строки ε является регулярным выражением, a *L*(ε) представляет собой множество {ε}, т. е. язык, единственный член которого – пустая строка.

3. Если *a* ∈ *T*, то *a* представляет собой регулярное выражение, a *L*(*a*) представляет множество {*a*}, т. е. язык с одной строкой единичной длины с символом *a*.

Имеют место правила, посредством которых регулярные выражения строятся из подвыражений. Пусть *r* и *s* являются регулярными выражениями, описывающими соответственно языки *L*(*r*) и *L*(*s*).

1. *r* | *s* (объединение) – регулярное выражение, описывающее язык *L*(*r*) ∪ *L* (*s*). Вместо символа ′|′ можно использовать символ ′+′, т. е. записи *r* | *s* и *r* + *s* эквивалентны.

2. *rs* (конкатенация) – регулярное выражение, описывающее язык *L*(*r*)*L*(*s*).

3. *r*\* (итерация, т. е. нуль или более повторений *r*) – регулярное выражение, описывающее язык (*L*(*r*))\*.

Все операции левоассоциативны. Подразумевается следующая система приоритетов: унарная операция итерации обладает наивысшим приоритетом, за ним следует операция конкатенации, а затем следует операция объединения. Приоритеты можно изменять с помощью использования скобок.

Одним из расширений регулярных выражений является унарная операция *r*+ (один или более повторений *r*). Эта операция имеет тот же приоритет и ассоциативность, что и операция итерации. Имеют место алгебраические законы: *r*\* = *r*+ | ε и *r*+ = *rr*\* = *r*\**r*.

Регулярное выражение генерирует регулярное множество. Например, регулярное выражение (*a* | *b*)*c* генерирует регулярное множество (регулярный язык) {*ac*, *bc*}, а регулярное выражение (*aab* | *ab*)\* – множество {*aab*, *ab*}\*, включающее строки

ε

*aabaabab*

*ababaab*

*abaabababaab* и т. п.

Регулярное выражение, описывающее классическое определение идентификатора, имеет вид *l*(*l* | *d*)\*, где *l* и *d* обозначают соответственно букву и цифру. Если в определение идентификатора добавить маркер конца ввода лексемы ⊥∉{*l*, *d*}, то есть любой символ кроме буквы и цифры, то регулярное выражение для него примет вид *l*(*l* | *d*)\*⊥.

Существует алгебра регулярных выражений, позволяющая выполнять эквивалентные преобразования выражений. Основные алгебраические свойства регулярных выражений (*q*, *r*, *s* – некоторые регулярные выражения) [14]:

1. *q* | *q* = *q*, *q* | ∅ = *q*
2. *q* | *r* = *r* | *q*, свойство коммутативности
3. (*q* | *r*) | *s* = *q* | (*r* | *s*) = *q* | *r* | *s* – ассоциативность
4. (*qr*)*s* = *q*(*rs*) = *qrs* – ассоциативна, но не коммутативна
5. *q*ε = ε*q* = *q*, *q*∅ = ∅*q* = ∅
6. (*r* | *s*)*q* = *rq* | *sq*
7. *q*(*r* | *s*) = *qr* | *qs*
8. *q*\**q*\* = *q*\*
9. (*q*\*)\* = *q*\*
10. *qq*\* = *q*\**q*
11. ε\* = ε, ∅\* = ε
12. (*q*\* | *r*\*)\* = (*q*\**r*\*)\* = (*q* | *r*)\*
13. (*rq*)\**r* = *r*(*qr*)\*
14. (*q*\**r*)\**q*\* = (*q* | *r*)\*
15. (*q*\**r*)\* = (*q* | *r*)\**r* | ε
16. (*qr*\*)\* = *q*(*q* | *r*)\* | ε

Для любого регулярного выражения можно определить недетерминированный конечный автомат, который принимает регулярный язык, соответствующий заданному регулярному выражению. Как и для регулярных грамматик, существует процедура построения конечного автомата-распознавателя по заданному регулярному выражению.

Конечные автоматы для базисных регулярных выражений ∅, ε, *a* представлены на рис. 2.11, где *k*0 и *kf* – начальное и конечное состояния автомата соответственно.

*k*0

*kf*

*k*0

*kf*

ε

*k*0

*kf*

*a*

*а*

*б*

*в*

Рис. 2.11. Конечные автоматы для базисных регулярных выражений:  
*а* – ∅; *б* – ε; *в* – *a*

Обратите внимание, что в графах переходов автоматов могут появиться переходы, помеченные символом ε (пустая строка). Такие переходы называются ε-*переходами*. Таким образом, функция переходов δ определена на множестве *K* × (*T* ∪ {ε}), т. е. δ: *K* × (*T* ∪ {ε}) → 2*K*.

Рассмотрим построение автоматов для операций над регулярными выражениями. Пусть *M*1 и *M*2 – конечные автоматы, распознающие языки, представленные регулярными выражениями *r* и *s* соответственно, причем их множества состояний не пересекаются. Обозначим через *k*10 и *k*20 начальные состояния этих автоматов, через *k*1*f* и *k*2*f* – их конечные состояния. Тогда конечный автомат *M* с начальным состоянием *k*0 и конечным состоянием *kf*, который представляет регулярное выражение *q* как результат регулярной операции над *r* и *s* строится следующим образом (рис. 2.12).

1. *q* = *r* | *s* (или *r* + *s*). Автомат *M* строится параллельным соединением автоматов *M*1 и *M*2 (рис. 2.12, *а*). Добавляются новые состояния *k*0 и *kf*, добавляются ε-переходы из *k*0 в *k*10 и *k*20, а также из *k*1*f* и *k*2*f* в *kf*. Любой путь из *k*0 в *kf* должен пройти либо исключительно через *M*1, либо исключительно через *M*2.

2. *q* = *rs*. Автомат *M* строится последовательным соединением автоматов *M*1 и *M*2 (рис. 2.12, *б*). Начальным состоянием *M* объявляется *k*10, конечным состоянием – *k*2*f*. Состояния *k*1*f* и *k*20 объединяются в одно состояние со всеми входящими и исходящими переходами обоих состояний. Путь из *k*0 в *kf* должен пройти сначала через *M*1, а затем через *M*2.

3. *q* = *r\**. Автомат *M* строится зацикливанием автомата *M*1 (рис. 2.12, *в*). Добавляются новые состояния *k*0 и *kf*, добавляются также ε-переходы из *k*0 в *k*10 и *kf*, из *k*1*f* в *k*10, из *k*1*f* в *kf*. Для достижения *kf* из *k*0 необходимо пройти либо по ε-переходу от *k*0 к *kf*, соответствующей пустой строке, либо перейти к начальному состоянию *k*10 автомата *M*1, пройти его и вернуться в *k*10 нуль или более раз.

ε

ε

ε

ε

ε

ε

ε

ε

*а*

*б*

*в*

*k*10

*k*1*f*

*M*1

*k*20

*k*2*f*

*M*2

*k*0

*kf*

*k*0=*k*10

*M*1

*M*2

*kf* = *k*2*f*

*k*10

*k*1*f*

*M*1

*k*0

*kf*

*k*1*f* =*k*20

Рис. 2.12. Графы переходов автоматов для регулярных выражений:  
*а* – *r* | *s*; *б* – *rs*; *в* – *r*\*

Заметим, что в этом методе на любом шаге построения начальные состояния не имеют входящих дуг, а конечные состояния – исходящих.

Наличие ε-перехода вносит недетерминированность в функционирование конечного автомата, поскольку автомат может переходить из состояния в состояние без чтения входного символа. Подробнее о НКА с ε-переходами изложено в разделе 2.8.

В качестве примера построим конечный автомат для регулярного выражения (*a* | *b*\*)*b*. Автоматы для *a*, *b* и *c* изображены на рис. 2.11, *в*. С помощью конструкции на рис. 2.12, *в* построим автомат для *b*\*, как показано на рис. 2.13, *а*. Затем с помощью конструкции на рис. 2.12, *а* построим автомат для *a* | *b*\*, как показано на рис. 2.13, *б*. Завершаем построение автомата с помощью конструкции на рис. 2.12, *б* для (*a* | *b*\*)*b*, получив автомат, изображенный на рис. 2.13, *в*.

ε

ε

ε

ε

ε

*а*

*b*

ε

ε

ε

ε

*б*

*b*

*a*

ε

ε

ε

ε

ε

ε

ε

ε

*в*

*b*

*a*

ε

ε

ε

*b*

Рис. 2.13. Построение автомата для выражения (*a* | *b*\*)*b*:  
*а* – для *b*\*; *б* – для *a* | *b*\*; *в* – для (*a* | *b*\*)*b*

## Конечные автоматы с ε-переходами

Недетерминированный конечный автомат *M* = (*K*, *T*, δ, *k*0, *F*) имеет ε-переходы, если функция δ определена на множестве *K* × (*T* ∪ {ε}), т. е. δ: *K* × (*T* ∪ {ε}) → 2*K*, где 2*K* – булеан множества *K* (множество всех подмножеств множества *K*, мощность булеана равна 2|*K*|).

Пример такого автомата представлен на рис. 2.14.

*a*

*c*

*b*

*a*

*b*

ε

*c*

ε

*a*

6

1

2

3

4

5

Рис. 2.14. Конечный автомат с ε-переходами

Наличие ε-перехода вносит недетерминированность в функционирование конечного автомата, поскольку автомат может переходить из состояния в состояние без чтения входного символа.

Пусть состояния *k*, *k*′∈ *K* такие, что *k*′∈ δ (*k*, ε), тогда говорят, что состояние *k*′ ε-*достижимо* из состояния *k* и записывают . Таким образом, автомат может в любой момент перейти из состояния *k* в состояние *k*′ без чтения входного символа. Рассмотрим более широкое понятие ε-достижимости, а именно , т. е. *k*′ ε-достижимо из *k*, если существует путь по графу автомата от вершины *k* до *k*′, состоящий из дуг, помеченных символом ε и длиной ≥ 0.

Обозначим через *R*(*k*) множество всех состояний, которые ε-достижимы из состояния *k*, что формально записывается как

*R*(*k*) = { *k*′ | }.

Следует обратить внимание на то, что *k* ∈ *R*(*k*).

Расширим это определение для подмножества *M* ⊆ *K* состояний следующим образом: .

Для автомата на рис. 2.14 имеем

*R*(1) = {1, 3, 4}, *R*(4) = {4},

*R*(2) = {2}, *R*(5) = {5},

*R*(3) = {3, 4}, *R*(6) = {6},

*R*({1, 3}) = *R*(1) ∪ *R*(3) = {1, 3, 4}.

Рассмотрим, как влияет на функционирование автомата наличие ε-переходов. Для состояния 1 имеем δ(1, *a*) = {2}, т. е. при чтении входного символа *a* автомат должен перейти из состояния 1 в состояние 2. Однако, поскольку имеются ε-переходы из состояния 1 в состояние 3, а из состояния 3 в состояние 4, автомат без обработки входного символа мог перейти в любое из этих состояний. Пусть автомат находится в состоянии 4, тогда чтение символа *a* может перевести автомат в состояние 4 или 6. Таким образом, при чтении символа *a* автомат может перейти из состояния 1 в состояние 2, 4 или 6.

Обозначим через *t*(*k*, *a*) множество состояний, которые могут быть достигнуты из состояния *k* после чтения входного символа *a*. Формально оно определяется следующим образом:

*t*(*k*, ε) = *R*(*k*);

 для всех *k* ∈ *K*, *a* ∈ *T*.

Например, для нашего автомата имеем

*t*(1, ε) = *R*(1) = {1, 3, 4};







Тогда правило построения НКА без ε-переходов *M*′ = (*K*, *T*, δ′, *k*0, *F*′), эквивалентного НКА с ε-переходами *M* = (*K*, *T*, δ, *k*0, *F*), заключается в следующем:

δ′(*k*, *a*) = *t*(*k*, *a*) для всех *k* ∈ *K*, *a* ∈ *T*;

*F*′ = *F* ∪ {*k* ∈ *K* | *R*(*k*) ∩ *F* ≠ ∅}, т. е. к множеству конечных состояний добавляются состояния *k*, для которых в их множества ε-достижимых состояний *R*(*k*) входят конечные состояния исходного автомата.

Построим автомат без ε-переходов, эквивалентный автомату на рис. 2.14. Это удобно делать по таблице переходов автомата, представленной на рис. 2.15 (для выделения конечное состояние заключено в квадратные скобки).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Вход  (*T*) | Состояния (*K*) | | | | | |
| 1 | 2 | 3 | 4 | 5 | [6] |
| *a* | {2} | ∅ | ∅ | {4, 6} | ∅ | ∅ |
| *b* | ∅ | {4} | ∅ | {5} | ∅ | ∅ |
| *c* | ∅ | ∅ | {3} | ∅ | {4} | ∅ |
| ε | {3} | ∅ | {4} | ∅ | ∅ | ∅ |
| *R*(*k*) | {1, 3, 4} | {2} | {3, 4} | {4} | {5} | {[6]} |

Рис. 2.15. Таблица переходов НКА, дополненная строкой для *R*(*k*)

Множества *R*(*k*) легко можно вычислить транзитивно по строке ε. Например, при вычислении *R*(1) само состояние 1 сразу включается в это множество. В строке ε для состояния 1 имеем состояние 3, которое добавляется в *R*(1). В строке ε для состояния 3 имеем состояние 4, которое также добавляется в *R*(1). В строке ε для состояния 4 содержится ∅, следовательно, процесс вычисления *R*(1) = {1, 3, 4} завершен.

После вычисления множеств *R*(*k*) для всех состояний *k* ∈ *K* легко можно вычислить множества *t*(*k*, *a*). Рассмотрим состояние *k*. Сначала для каждого состояния из *R*(*k*) по таблице переходов определяется множество состояний, в которые может перейти автомат при чтении символа *a*. Затем выполняется объединение всех множеств *R*(*k*), содержащихся в столбцах, соответствующих найденным состояниям. В результате будет получено множество *t*(*k*, *a*). Рассмотрим вычисление множеств *t*(*k*, *a*) на примере состояния 1 при чтении символа *c*, т. е. вычислим *t*(1, *c*). Поскольку *R*(1) = {1, 3, 4}, согласно определению

*t*(1, *c*) = *R*(δ(1, *c*)) ∪ *R*(δ(3, *c*)) ∪ *R*(δ(4, *c*)).

По таблице переходов определяем δ(1, *c*) = ∅, δ(3, *c*) = {3}, δ(4, *c*) = ∅, следовательно,

*t*(1, *c*) = *R*(∅) ∪ *R*({3}) ∪ *R*(∅) = {3, 4}.

Процесс построения автомата без ε-переходов завершается после вычисления функции переходов *t*(*k*, *a*) для всех *k* ∈ *K*, *a* ∈ *T* и уточнения множества конечных состояний.

Функция переходов конечного автомата без ε-переходов, эквивалентного конечному автомату с ε-переходами из рис. 2.14, представлена на рис. 2.16, а граф автомата – на рис. 2.17.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Вход  (*T*) | Состояния (*K*) | | | | | |
| 1 | 2 | 3 | 4 | 5 | [6] |
| *a* | {2, 4, 6} | ∅ | {4, 6} | {4, 6} | ∅ | ∅ |
| *b* | {5} | {4} | {5} | {5} | ∅ | ∅ |
| *c* | {3, 4} | ∅ | {3, 4} | ∅ | {4} | ∅ |

Рис. 2.16. Таблица переходов автомата без ε-переходов

*a*

*a*

*b*

*a*|*c*

*a*

*c*

*b*

*a*

*b*

*a*|*c*

*c*

*c*

*a*

6

1

2

3

4

5

*b*

Рис. 2.17. Конечный автомат без ε-переходов

После исключения из НКА ε-переходов автомат в общем случае остается недетерминированным. Поскольку использование НКА в качестве распознавателя приводит к существенным потерям времени при лексическом анализе, на следующем этапе необходимо преобразовать его в эквивалентный ДКА.

Для сложных регулярных выражений построенный ДКА может оказаться не минимальным. Поэтому завершающим этапом является минимизация ДКА.

## Минимизация конечного автомата

При построении ДКА выгодно, чтобы автомат имел как можно меньше состояний. Для произвольного ДКА можно построить эквивалентный автомат с наименьшим числом состояний (возможно, им будет исходный автомат). Процедуру построения такого автомата называют *минимизацией конечного автомата*, а сам автомат – *минимальным конечным автоматом*.

Для простоты будем рассматривать только ДКА, имеющие полную функцию переходов. Если исходный ДКА имеет неполную функцию переходов, то, добавив новое фиктивное состояние, можно определить новый эквивалентный ДКА с полной функцией переходов.

Автомату без выхода можно поставить в соответствие автомат Мура, у которого дуги, ведущие в конечные состояния, помечаются 1, а все остальные дуги помечаются 0. Тогда задача сводится к задаче минимизации автоматов Мура.

Другой путь – минимизация непосредственно автомата без выхода. Пусть *M* = (*K*, *T*, δ, *k*0, *F*) – ДКА с полной функцией переходов, принимающий язык *L*. Рассмотрим построение минимального ДКА *ML* исходя только из языка *L*.

Минимизация заключается в последовательном разбиении множества состояний на непересекающиеся подмножества неразличимых состояний до тех пор, пока такое разбиение возможно.

Определим на множестве *K* отношения *D*0, *D*1, *D*2, … следующим образом: *kD*0*k*′ (состояния *k* и *k*′ различимы по строке длины 0) тогда и только тогда, когда *k* ∈ *F* и *k*′ ∉ *F* или *k* ∉ *F* и *k*′ ∈ *F*.

Пусть *i* > 0, *kDik*′ (состояния *k* и *k*′ различимы по строке длины ≤ *i*) тогда и только тогда, когда *kDi*–1*k*′, т. е. существует *a* ∈ *T*, такое, что δ(*k*, *a*)*Di*–1δ(*k*′, *a*). Говорят, что состояние *k* различимо от состояния *k*′, если существует такое *i* ≥ 0, что *kDik*′. Другими словами, *kDik*′ тогда и только тогда, когда существует такая строка *x*, ⎪*x*⎪ ≤ *i*, что либо δ(*k*, *x*) ∈ *F* и δ(*k*′, *x*) ∉ *F*, либо δ(*k*, *x*) ∉ *F* и δ(*k*′, *x*) ∈ *F*.

Чтобы вычислить отношение *D*, необходимо последовательно вычислить отношения *D*0, *D*1, *D*2, …, и если в процессе этих вычислений на некотором шаге *r* окажется, что *Dr*+1 = *Dr*, то это будет означать, что итерационный процесс окончен и *D* = *Dr*.

Если *m* = ⎪*K*⎪, то существует только (*m*2 – *m*) пар (*ki*, *kj*), где *i* ≠ *j*. В худшем случае всякое *Di*+1 отличается от *Di* двумя такими парами, тогда *D* = *Dr*, где *r* < (*m*2 – *m*) / 2, т. е. процесс всегда конечен. В результате будут найдены все неразличимые состояния автомата, которые образуют множество состояний минимального автомата *ML*.

Пусть *K*′ ⊂ *K* – одно из множеств неразличимых состояний. функция переходов автомата *ML* определяется следующим образом: δ*L*(*K*′, *a*) = *K*′′, где *K*′′ – множество неразличимых состояний, содержащее состояние δ(*k*, *a*) для всех *k* ∈ *K*′. Тогда начальным состоянием автомата *ML* является множество неразличимых состояний, содержащее начальное состояние исходного автомата *M*. Конечными состояниями автомата *ML* являются те множества неразличимых состояний, которые содержат конечные состояния автомата *M*.

Рассмотрим пример минимизации ДКА, граф переходов которого приведен на рис. 2.18.

*b*

*b*

*a*

*b*

*b*

*a*

*a*

1

*a*

2

4

5

3

*b*

*a*

Рис. 2.18. Граф переходов минимизируемого ДКА

Отношение *Di* можно представить в виде булевой матрицы размером 5 × 5, в которой значение элемента (*j*, *k*) равно *T* (true), если *jDik*, и равно *F* (false) противном случае. Поскольку отношение *Di* симметрично и никогда не выполняется отношение *kDik* (элементы главной диагонали всегда равны *F*), для построения матрицы достаточно определить только те элементы, которые находятся над главной диагональю.

Последовательность матриц, соответствующих отношениям *D*0, *D*1, будет следующей:



Например, элемент (1, 2) равен *F*, поскольку оба состояния 1 и 2 не являются конечными (неразличимы по строке длины 0), элемент (1, 3) равен *T*, поскольку состояние 1 не является конечным, а состояние 3 – конечное (различимы по строке длины 0), т. е. 1*D*03. Аналогично определяются остальные значения элементов матрицы для отношения *D*0.

Для отношения *D*1 имеем 1*D*12, поскольку δ(1, *b*)*D*0δ(2, *b*), и 2*D*14, поскольку δ(2, *b*)*D*0δ(4, *b*).

Отношение *D*2 будет полностью совпадать с отношением *D*1, т. е. процесс минимизации завершен. В результате имеем, что состояния 1 и 4 неразличимы между собой, а также состояния 3 и 5 также неразличимы. Таким образом, выполнено разбиение множества состояний на непересекающиеся подмножества неразличимых состояний: {1, 4}, {2}, {3, 5}. Начальным состоянием минимального автомата будет состояние {1, 4} (содержит начальное состояние 1 исходного автомата), конечным состоянием будет {3, 5} (содержат конечные состояния 3 и 5 исходного автомата). Переобозначим эти состояния через 1, 2 и 3 соответственно. Тогда граф переходов полученного минимального ДКА примет вид, приведенный на рис. 2.19.

*b*

*a*

1

*a*

2

*b*

3

*b*

*a*

Рис. 2.19. Минимальный ДКА

Упражнения

2.1. Построить праволинейные регулярные грамматики, порождающие следующие регулярные языки:

а) {*a*β*b* | β ∈ (*a*, *b*)*\**};

б) {*ambn* | *m*, *n* ≥ 1};

в) {*ambnck* | *m*, *n*, *k* ≥ 1};

г) {(*abb*)*n* | *n* ≥ 1};

д) все строки из множества {*a*, *b*}\*, такие, что в каждой из них непосредственно справа от каждого символом *a* следует символ *b*;

е) множество операций сравнения {<, <=, >, >=, =, <>};

ж) числовая константа, формат которой задан в виде РБНФ (терминал **d** означает цифру от 0 до 9):

ЧисловаяКонст = Целое [ "**.**" Целое ] ["**e**" [ ("**+**" **|** "**–**" ) ] Целое ].

Целое = "**d**" { "**d**" }.

2.2. По регулярным грамматикам, полученным в упр. 2.1, построить ДКА, принимающие соответствующие языки.

2.3. Построить регулярные выражения, определяющие регулярные языки из упр. 2.1.

2.4. По регулярным выражениям, полученным в упр. 2.3, построить ДКА, принимающие соответствующие языки.

2.5. Построить минимальные ДКА по следующим регулярным выражениям:

а) (*bab*)\*(*aba*)\*;

б) *a*(*a* | *b*)\**a*;

в) *a*(*ba* | *b*)\* | *b*;

г) (*ab* | *b*\*)\**ba* | *b*;

д) ((*b*\**a*)\**ab*\*)\*.

2.6. Построить ДКА, эквивалентный НКА на рис. 2.17.

2.7. Доказать эквивалентность праволинейных и леволинейных регулярных грамматик.

# НИСХОДЯЩИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ

Фаза синтаксического анализа реализуется частью компилятора, которая называется *синтаксическим анализатором* (или *парсером*). Парсер исследует последовательность токенов, формируемую лексическим анализатором, с целью проверки, соответствует ли она синтаксису языка, создавая промежуточное представление (в общем случае древовидное в виде дерева разбора), описывающее грамматическую структуру потока токенов. На практике редко требуется явное построение дерева разбора, оно больше носит концептуальный характер.

Формальной основой синтаксического анализа являются КС-грамматики. Входной поток токенов рассматривается как строка языка, состоящая из токенов. Для КС-грамматики, описывающей синтаксис языка, токены (точнее, коды токенов, идентифицирующие соответствующие лексические классы) рассматриваются как терминалы. Множество терминалов КС-грамматики – это множество лексических классов (токенов). Парсер должен определить, принадлежит ли входная строка данному языку.

На практике в компиляторах в основном применяют нисходящие (предиктивные) и восходящие методы синтаксического анализа. В данной главе рассматриваются нисходящие методы. Восходящие методы будут рассмотрены в главе 4.

Предварительно рассмотрим общие вопросы распознавания КС-языков.

## Автомат с магазинной памятью

Распознавателем КС-языка является *автомат с магазинной памятью* (*МП-автомат*, *магазинный автомат*), эквивалентный конечному автомату, к которому добавлена память магазинного типа (стек).

В функции МП-автомата входит:

а) чтение входного символа, замещение верхнего символа стека строкой символов (возможно пустой) и изменение состояния или

б) все то же самое, но без чтения входного символа.

Формально автомат с магазинной памятью определяется как семерка *M* = (*K*, *T*, Γ, δ, *k*0, *Z*0, *F*), где

*K* – конечное множество состояний;

*Т* – конечный входной алфавит;

Γ – конечный *стековый* (*магазинный*) алфавит;

– функция переходов, которая отображает множество *K*×(*T* ∪ {ε})×Γ на множество конечных подмножеств множества *K*×Γ\*;

*k*0 ∈ *K* – начальное состояние автомата;

*Z*0 ∈ Γ – начальный символ стека, который первоначально находится в вершине стека;

*F*⊆ *K* – множество конечных (заключительных, финальных, принимающих) состояний.

Запись функции δ(*k*, *a*, *A*) = (*k'*, γ), где *k*, *k'* ∈ *K*, *a* ∈ *T* ∪ {ε}, *A* ∈ Γ, γ ∈ Γ\* означает, что в текущем состоянии автомата *k* с элементом *А* в вершине стека при чтении символа *а* осуществляется переход в состояние *k'* и в стек вместо элемента *A* заносится строка γ (элемент *A* замещается строкой γ).

*Конфигурация* МП-автомата в любой момент времени описывается тройкой (*k*, *x*, α) ∈ *K*×*T*\*×Γ\*, где

*k* ∈ *K* – текущее состояние автомата;

*x* ∈ *T*\* – необработанная часть входной строки (первый символ строки *x* является очередным входным символом, если *x* = ε, то считается, что входная строка полностью прочитана);

α ∈ Γ\* – текущее содержимое стека, причем самый левый символ строки α считается верхним символом стека (если α = ε, то стек считается пустым).

*Такт работы* МП-автомата определяется бинарным отношением ˫, определенным на множестве конфигураций. Если (*k'*, γ) ∈ δ(*k*, *a*, *A*), где *k*, *k'* ∈ *K*, *a* ∈ *T*, *A* ∈ Γ, γ ∈ Γ\*, автомат переходит из конфигурации *c* = (*k*, *ax*, *A*α), *x* ∈ *T*\*, α ∈ Γ\* в конфигурацию *c'* = (*k'*, *x*, γα), т. е. (*k*, *ax*, *A*α) ˫ (*k'*, *x*, γα). Если γ = ε, то просто удаляется символ из вершины стека.

Если (*k'*, γ) ∈ δ(*k*, ε, *A*), то (*k*, *ax*, *A*α) ˫ (*k'*, *ax*, γα). т. е. в этом такте, называемом ε-*тактом*, автомат переходит из одной конфигурации в другую без чтения входного символа (ε-*переход*).  
ε-такты возможны и в случае, если входная строка полностью прочитана, но если стек пуст, следующий такт невозможен.

*Начальной конфигурацией* МП-автомата называется конфигурация вида (*k*0, *x*, *Z*0), т. е. автомат находится в начальном состоянии *k*0, на входе автомата распознаваемая строка *x*, стек содержит начальный символ *Z*0.

*Заключительной конфигурацией* МП-автомата называется конфигурация вида (*kf*, ε, α), где *kf* ∈ *F* – одно из конечных состояний, входная строка прочитана до конца, а в стеке содержится заранее определенная строка α ∈ Γ\* (часто α = ε).

Можно определить транзитивное (˫+) и рефлексивно-транзитивное (˫\*) замыкания отношения ˫. Запись *c* ˫+ *c'* означает, что конфигурация *c'* достижима (выводима) из конфигурации *c* за один или более тактов, а запись *c* ˫\* *c'* – за нуль или более тактов.

Говорят, что строка *x* ∈ *T*\* *допускается* (*принимается*) МП-автоматом *M* = (*K*, *T*, Γ, δ, *k*0, *Z*0, *F*), если (*k*0, *x*, *Z*0) ˫\* (*kf*, ε, α) для некоторых *kf* ∈ *F* и α ∈ Γ\*, т. е. если в результате чтения входной строки автомат перейдет из исходной конфигурации в заключительную конфигурацию. Языком *L*(*M*), *определяемым* (*допускаемым*) МП-автоматом *M*, называется множество строк, допускаемых этим автоматом.

МП-автомат называется *детерминированным*, если выполняются следующие условия:

а) функция вида δ(*k*, *a*, *A*) имеет не более одного элемента;

б) функция вида δ(*k*, ε, *A*) имеет не более одного элемента;

в) если δ(*k*, ε, *A*) ≠ ∅, то δ(*k*, *a*, *A*) = ∅ для любого *a* ∈ *T*, т. е. если из некоторой конфигурации можно осуществить хотя бы один ε-переход, то он является единственным переходом, которое можно осуществить из этой конфигурации.

Другими словами, если из любой конфигурации возможен единственный переход, МП-автомат является детерминированным, в противном случае – *недетерминированным*.

## МП-автоматы и КС-грамматики

Справедливо следующее утверждение: для любого КС-языка существует недетерминированный МП-автомат, который принимает его, и наоборот – если некоторый МП-автомат принимает некоторый язык, то этот язык является КС-языком.

Рассмотрим вопросы построения МП-автомата, распознающего КС-язык, заданный КС-грамматикой *G* = (*VT*, *VN*, *P*, *S*). МП-автомат *M* = (*K*, *T*, Γ, δ, *k*0, *Z*0, *F*), принимающий данный язык, определяется следующим образом:

*K* = {*k*}, т. е. МП-автомат имеет единственное состояние *k*;

*T* = *VT*, т. е. входной алфавит МП-автомата совпадает с множеством терминалов КС-грамматики;

Γ = *VT* ∪*VN*, т. е. стековый алфавит образуется объединением множеств терминалов и нетерминалов;

*k*0 = *k*;

*Z*0 = *S*, т. е. начальным символом стека является начальный нетерминал грамматики;

*F* = {*k*};

т. е. *M* = ({*k*}, *VT*, *VT* ∪*VN*, δ, *k*, *S*, {*k*}), где функция переходов δ определяется следующим образом:

δ(*k*, ε, *A*) = {(*k*, α)⎪*A* → α ∈ *Р*} для всех *A* ∈ *VN*;

δ(*k*, *a*, *a*) = {(*k*, ε)} для всех *а* ∈ *VT*.

Такой МП-автомат эмулирует левосторонний вывод. При каждом такте, выполняемом автоматом, из стека извлекается один символ. Если извлеченный символ оказывается нетерминалом, то ему в соответствие ставится продукция, правая часть которой в таком случае заносится в стек. Если же извлеченный из стека символ оказывается терминалом, то он используется в качестве входного символа и, следовательно, определяет следующий такт автомата.

Определим МП-автомат для КС-грамматики, порождающей КС-язык *L* = {*anbncm* | *n*, *m* ≥ 1},

*S* → *TC*

*T* → *aTb*⎪*ab*

*C* → *cC*⎪*c*

МП-автомат *M* = ({*k*}, {*a*, *b*, *c*}, {*a*, *b*, *c*, *S*, *T*, *C*}, δ, *k*, *S*, {*k*}), распознающий данный язык, имеет следующие функции переходов:

δ(*k*, ε, *S*) = {(*k*, *TC*)};

δ(*k*, ε, *T*) = {(*k*, *aTb*), (*k*, *ab*)};

δ(*k*, ε, *C*) = {(*k*, *cC*), (*k*, *c*)};

δ(*k*, *a*, *a*) = {(*k*, ε)} для всех *а* ∈ {*a*, *b*, *c*}.

При анализе входной строки *aabbc* автомат может выполнить следующую последовательность тактов, приводящую к заключительной конфигурации:

(*k*, *aabbc*, *S*) ˫ (*k*, *aabbc*, *TC*)

˫ (*k*, *aabbc*, *aTbC*)

˫ (*k*, *abbc*, *TbC*)

˫ (*k*, *abbc*, *abbC*)

˫ (*k*, *bbc*, *bbC*)

˫ (*k*, *bc*, *bC*)

˫ (*k*, *c*, *C*)

˫ (*k*, *c*, *c*)

˫ (*k*, ε, ε).

Приведенная последовательность тактов соответствует левосторонней схеме вывода *S* ⇒ *TC* ⇒ *aTbC* ⇒ *aabbC* ⇒ *aabbc*.

Из недетерминированности автомата следует, что построенный на его основе синтаксический анализатор КС-языка в процессе функционирования может осуществлять возврат к предыдущей конфигурации. Это означает, что, обнаружив ошибку выбора, необходимо вернуться к моменту осуществления выбора, вновь осуществить выбор и выполнить другой переход. Однако, если на порождающую язык грамматику наложить определенные ограничения, то можно построить эффективный детерминированный синтаксический анализатор для такого языка.

Язык, принимаемый детерминированным МП-автоматом, называется *детерминированным* языком. Не всякий КС-язык является детерминированным, и такие языки не могут анализироваться детерминированным образом. Тем не менее, детерминированные языки составляют очень важный класс языков, поскольку для них значительно упрощается решение задачи анализа. Большинство языков программирования являются детерминированными или почти таковыми. Некоторые языки можно разбирать детерминированно с помощью только одного из методов грамматического разбора.

Важно заметить, что в отличие от конечных автоматов в общем случае нельзя преобразовать недетерминированный МП-автомат в эквивалентный детерминированный МП-автомат. Это объясняется тем, что детерминированные КС-языки составляют только подкласс КС-языков. Для недетерминированного КС-языка невозможно построить детерминированный МП-автомат, принимающий этот язык.

## *LL*(*k*)-грамматики

Нисходящие методы синтаксического анализа основаны на просмотре входной строки и построении дерева разбора, начиная с начального нетерминала грамматики. Дерево строится до тех пор, пока не получится анализируемая строка терминалов. Если такую строку удается получить, то анализируемая строка принадлежит языку, если нет – не принадлежит. Этот процесс равносилен процессу построения соответствующей схемы вывода анализируемой строки.

Во все рассматриваемые далее грамматики введем специальный символ ⊥, принадлежащий множеству терминалов и обозначающий конец вводимой строки (*маркер конца ввода*). Такие грамматики можно назвать *пополненными*. Даже если маркер явно не указан в грамматике, будем подразумевать, что всегда имеется продукция вида *S'* → *S*⊥, где *S'* и *S* – начальные нетерминалы соответственно пополненной и исходной грамматик.

Рассмотрим так называемый *LL*-разбор. Для успешного *LL*-разбора строк некоторого КС-языка на порождающую его грамматику должны быть наложены строгие ограничения. Практически это означает, что методом *LL*-разбора можно воспользоваться лишь на подмножестве класса детерминированных языков.

Рассмотрим грамматику с продукциями

*S* → *A*⊥

*A* → *aFaB*⎪*bFbB*

*F* → *b*⎪*ba*

*B* → *b*⎪*bB*

Построим дерево нисходящего синтаксического разбора строки *abaabb*⊥ (рис. 3.1). Корень дерева помечен символом *S* и, следовательно, единственной продукцией грамматики, которая может быть использована в данном случае, является продукция *S* → *A*⊥. Перейдем теперь к вершине *A*. Можно воспользоваться продукциями *A* → *aFaB* и *A* → *bFbB.* Поскольку рассматриваемая строка *abaabb*⊥ начинается с символа *a*, воспользуемся первой из этих продукций, т. е. выбирается та продукция, правая часть которой начинается с очередного введенного входного символа. Итак, один из листьев дерева уже помечен символом *а*.

Рис. 3.1. Дерево разбора строки *abaabb*⊥

Далее необходимо показать, что строка *FaB*⊥ может породить строку *baabb*⊥. Рассмотрим для этого продукции *F* → *b*⎪*ba*. На этот раз знать следующий введенный символ и даже два следующих введенных символа входной строки недостаточно для выбора одной из этих продукций. Однако если станут известны три очередных символа входной строки и если это символы *bab*, то следует воспользоваться первой продукцией, а если это символы *baa*, то второй продукцией. В данном случае очередные символы входной строки – это символы *baa*, поэтому следует выбрать продукцию *F* → *ba*. Теперь необходимо убедиться, что из нетерминала *B* можно породить строку *bb*. Для этого надо рассмотреть продукции *B* → *b*⎪*bB*. Опять знания следующего символа *b* недостаточно, но если известны два очередных символа и это символы *b*⊥, то следует использовать продукцию *B* → *b*, а если это символы *bb*, – продукцию *B* →*bB*. В данном случае очередные символы – это символы *bb*, поэтому используем продукцию *B* →*bB*. Затем, согласно двум очередным символам входной строки, применим продукцию *B* → *b*. В результате будет получена левосторонняя схема вывода

*S* ⇒ *A*⊥ ⇒ *aFaB*⊥ ⇒ *abaaB*⊥ ⇒ *abaabB*⊥ ⇒ *abaabb*⊥.

Эта грамматика является примером *LL*(3)-грамматики, т. е. для однозначного определения дерева вывода достаточно знать не более трех очередных символов входной строки на каждом этапе построения дерева.

В общем случае *LL*(*k*)-грамматикой называют такую грамматику *G* = (*VT*, *VN*, *P*, *S*), что для любой ее сентенциальной формы *wAy*, *w* ∈ *VT*\*, *A* ∈ *VN*, *y* ∈ (*VT* ∪ *VN*)\*, полученной в результате некоторого левостороннего вывода, для однозначного выбора продукции, имеющей в левой части нетерминал *А*, достаточно знать *k* очередных символов входной строки. Аббревиатура *LL* означает "левосторонний ввод – левосторонний вывод".

Чем больше *k*, тем больший класс языков может быть представлен *LL*(*k*)-грамматикой, но и разбор выполняется значительно сложнее. Поэтому на практике наибольшее применение имеют *LL*(1)-грамматики, для которых детерминированный распознаватель работает, анализируя по одному входному символу, расположенному в текущей позиции.

## *LL*(1)-грамматики

Прежде чем определить *LL*(1)-грамматику, рассмотрим некоторые подклассы *LL*(1)-грамматик и вопросы их разбора.

### Разделенные грамматики

*Разделенная*, или *простая*, грамматика (*s*-*грамматика*) представляет собой грамматику, в которой правая часть каждой продукции начинается с терминального символа, за терминалом могут следовать нетерминалы и/или терминалы; правые части альтернативных продукций начинаются с разных терминалов.

Эти условия позволяют построить детерминированный нисходящий синтаксический анализатор, т. к. при выводе строки языка всегда можно сделать однозначный выбор между альтернативными продукциями для самого левого нетерминала в сентенциальной форме, предварительно исследовав один следующий символ входной строки.

Например, грамматика с продукциями

*S* → *pA*⊥⎪*qB*⊥

*A* → *a*⎪*cAd*

*B* → *b*⎪*cBg*

представляет собой *s*-грамматику.

Рассмотрим разбор строки *pccadd*⊥. Разбор начинается с начального символа грамматики *S*. На первом шаге из двух альтернативных *S*-продукций выбираем продукцию *S* → *pA*⊥, поскольку входным символом является символ *p*. В результате получим сентенциальную форму *pA*⊥. На втором шаге анализируется второй символ входной строки, поскольку это символ *c*, из *A*-продукций выбираем продукцию *A* → *cAd* и получаем сентенциальную форму *pcAd*⊥. Продолжая аналогичные действия, получим левостороннюю схему вывода

*S* ⇒ *pA*⊥ ⇒ *pcAd*⊥ ⇒ *pccAdd*⊥ ⇒ *pccadd*⊥.

### Слаборазделенные грамматики

Пусть дана следующая грамматика:

*S* → *pA*⊥⎪*qBA*⊥

*A* → *a*⎪*bAd*

*B* → ε⎪*cBg*

Эта грамматика не принадлежит классу *s*-грамматик, поскольку содержит ε-продукцию *B* → ε, правая часть которой не начинается с терминала. Выбрать при разборе продукцию по левому терминалу правой части не удается.

Пусть *Follow*(*X*) – множество терминалов, которые могут следовать непосредственно за нетерминалом *X* в какой-либо сентенциальной форме, выводимой из начального нетерминала. Например, для приведенной грамматики *Follow*(*B*) = {*a*, *b*, *g*}.

Критерием выбора между альтернативными продукциями является *множество направляющих символов* (*множество выбора*) *DS*, определяемое следующим образом:

Если продукция имеет вид *A* → *b*α, где *A* ∈ *VN*, *b* ∈ *VT*, α ∈ (*VT* ∪ *VN*)\*, то *DS*(*A* → *b*α) = {*b*}. Если продукция имеет вид *A* → ε, то *DS*(*A* → ε) = *Follow*(*A*).

Контекстно-свободная грамматика называется *слаборазделенной* (*q*-*грамматикой*) при условии:

а) правая часть каждой продукции либо начинается с терминала, либо представляет собой ε;

б) множества направляющих символов альтернативных продукций не пересекаются.

Таким образом, приведенная выше грамматика относится к классу *q*-грамматик, т. к. множества *DS* альтернативных продукций не пересекаются. Это позволяет при разборе строки детерминированно выбирать нужную продукцию из альтернативных.

Процесс вывода строки *qbad*⊥ соответствует следующей левосторонней схеме вывода:

*S* ⇒ *qBA*⊥ ⇒ *qA*⊥ ⇒ *qbAd*⊥ ⇒ *qbad*⊥.

На втором шаге вывода, поскольку *b* ∈ *DS*(*B* → ε) = {*a*, *b*, *g*}, применена продукция *B* → ε.

### *LL*(1)-грамматики

*LL*(1)-грамматика является обобщением *q*-грамматик, принцип обобщения позволяет строить нисходящие детерминированные анализаторы.

Пусть *G* = (*VT*, *VN*, *P*, *S*) произвольная КС-грамматика. Определим функции *First* и *Follow*.

*First*(*X*) = {*a*⎪}, *X*∈(*VT* ∪ *VN*), *a*∈*VT*, β∈(*VT* ∪ *VN*)\*,

т. е. функция *First*(*X*) определяет множество терминалов, с которых может начинаться строка, выводимая из символа *Х*.

Расширим понятие функции *First* на строку символов α = *X*1*X*2...*Xn*, где *Xi* ∈ *VT* ∪ *VN*, 1 ≤ *i* ≤ *n*:

*First*(*X*1*X*2...*Xn*) = ,

т. е. определяет множество терминалов, с которых может начинаться строка, выводимая из строки α = *X*1*X*2...*Xn*, учитывая наличие ε-порождающих нетерминалов. Таким образом, сначала в множество *First*(α) = *First*(*X*1*X*2...*Xn*), добавляется множество *First*(*X*1), затем, если *X*1 не порождает ε, то процесс вычисления завершается, если же  (т. е. *X*1 – ε-порождающий нетерминал), в множество *First*(α) добавляются элементы множества *First*(*X*2) и т. д.

Очевидно, что если строка α = *a*β, где *a*∈ *VT*, β∈ (*VT* ∪ *VN*)\*, т. е. строка α начинается с терминала, то *First*(*a*β) = {*a*}. Очевидно также, что *First*(ε) = ∅. Из определения следует, что для нетерминала *A* с *A*-продукциями *A* → α1⎪α2⎪…⎪α*k*, α ∈ (*VT* ∪ *VN*)+, α*i* ∈ (*VT* ∪ *VN*)\*, 1 ≤ *i* ≤ *k*

*First*(*A*) = *First*(α1) ∪ *First*(α2) ∪ … ∪ *First*(α*k*).

Аргументом функции *Follow* является нетерминал *X*∈ *VN*:

*Follow*(*X*) = {*a*⎪}, *a*∈ *VT*, α∈ *VT*\*, β∈ (*VT* ∪ *VN*)\*.

Данная функция (как и в *q*-грамматике) определяет множество терминалов, которые могут следовать непосредственно за нетерминалом *X* в какой-либо сентенциальной форме, выводимой из начального нетерминала *S*. Из определения следует, что если существует продукция вида *A* → α*X*γ, α,γ∈ (*VT* ∪ *VN*)\*, то

*Follow*(*X*) = *First*(γ) ∪ *Follow*(*A*) | .

Тогда множество *DS* направляющих символов продукций *LL*(1)-грамматики определяется следующим образом:

*DS*(*A* → α) = *First*(α) ∪ *Follow*(*A*), если ,

*DS*(*A* → α) = *First*(α) в противном случае.

Другими словами, если правая часть продукции может генерировать пустую строку, то к элементам множества *First*(α) необходимо добавить элементы множества *Follow*(*A*) для нетерминала из левой части продукции.

Контекстно-свободная грамматика *G* = (*VT*, *VN*, *P*, *S*) называется *LL*(1)-*грамматикой* (*обладает* *LL*(1)-*свойствами*) тогда и только тогда, когда для любой пары несовпадающих альтернативных *A*-продукций вида *А* → α и *А* → β справедливо утверждение, что

*DS*(*A* → α) ∩ *DS*(*A* → β) = ∅,

т. е. множества направляющих символов альтернативных продукций, не пересекаются.

Из определения следует, что леворекурсивная грамматика не может быть *LL*(1)-грамматикой. Другим важным свойством *LL*(1)-грамматики является ее однозначность.

Существуют формальные алгоритмы определения, относится ли заданная КС-грамматика к классу *LL*(1)-грамматик, основанные на вычислении соответствующих функций и определения множеств направляющих символов через построение бинарных отношений, их транзитивных замыканий и вычислении их произведений [7; 10]. Эти алгоритмы могут быть полезны при создании программного инструментария для работы с грамматиками.

Рассмотрим грамматику с продукциями (в предположении, что имеется продукция *S'* → *S*⊥ с маркером конца ввода ⊥):

*S* → *AB*⎪*DEa C* → *bC*⎪ε

*A* → *ac*⎪*f D* → *dD*⎪ε

*B* → *bC E* → *eE*⎪ε

Определим сначала множество всех ε-порождающих нетерминалов *N*ε = {*C*, *D*, *E*} (см. разд. 1.6.6).

Вычислим значение функции *First*(*A*) для всех нетерминалов *A*∈ *VN* (для терминалов значениями функции являются сами терминалы):

*First*(*S*) = {*a*, *d*, *e*, *f*}, *First*(*C*) = {*b*},

*First*(*A*) = {*a*, *f*}, *First*(*D*) = {*d*},

*First*(*B*) = {*b*}, *First*(*E*) = {*e*}.

Рассмотрим подробнее вычисление *First*(*S*), остальные значения вычисляются аналогично. В соответствии с продукциями *S* → *AB*⎪*DEa* имеем

*First*(*S*) = *First*(*AB*) ∪ *First*(*DEa*).

Поскольку *A*∉ *N*ε (нетерминал *A* не порождает пустую строку),

*First*(*AB*) = *First*(*A*).

В соответствии с продукциями *A* → *ac*⎪*f*

*First*(*A*) = *First*(*ac*) ∪ *First*(*f*) = {*a*} ∪ {*f*} = {*a*, *f*}.

Так как *D*, *E*∈ *N*ε (являются ε-порождающими нетерминалами),

*First*(*DEa*) = *First*(*D*) ∪ *First*(*E*) ∪ *First*(*a*).

В соответствии с продукциями *D* → *dD*⎪ε

*First*(*D*) = *First*(*dD*) ∪ *First*(ε) = {*d*} ∪ ∅ = {*d*}.

В соответствии с продукциями *E* → *eE*⎪ε

*First*(*E*) = *First*(*eE*) ∪ *First*(ε) = {*e*} ∪ ∅ = {*e*}.

Таким образом,

*First*(*DEa*) = {*d*} ∪ {*e*} ∪ {*a*} = {*a*, *d*, *e*}.

В итоге получаем

*First*(*S*) = *First*(*AB*) ∪ *First*(*DEa*) =

= {*a*, *f*} ∪ {*a*, *d*, *e*} = {*a*, *d*, *e*, *f*}.

Вычислим значение функции *Follow*. Следует заметить, что вычисление этой функции требуется только для ε-порождающих нетерминалов. Но для учебных целей вычислим для всех нетерминалов. Вычисление выполняется в соответствии со сформулированным выше правилом: если существует продукция вида *A* → α*X*γ, α,γ∈ (*VT* ∪ *VN*)\*, то

*Follow*(*X*) = *First*(γ) ∪ *Follow*(*A*) | .

*Follow*(*S*) = {⊥}, поскольку предполагается наличие продукции *S'* → *S*⊥ и *First*(⊥) = {⊥}.

*Follow*(*A*) = {*b*}, поскольку *A* встречается только в правой части продукции *S* → *AB*, а *First*(*B*) = {*b*}, причем *B*∉ *N*ε.

*Follow*(*B*) = {⊥}, поскольку *B* встречается только в правой части продукции *S* → *AB*, а *Follow*(*S*) = {⊥}.

*Follow*(*C*) = {⊥}, поскольку *C* встречается в правых частях продукций *B* → *bC* и *C* → *bC*, а *Follow*(*B*) = {⊥}, праворекурсивная продукция *C* → *bC* не влияет на результат, поскольку для нее согласно правилу имеет место *Follow*(*C*) = *Follow*(*C*).

*Follow*(*D*) = {*a*, *e*}, поскольку *D* встречается только в правой части продукции *S* → *DEa* (из тех же соображений, что и при вычислении *Follow*(*C*), не учитываем праворекурсивную продукцию *D* → *dD*), а *First*(*Ea*) = *First*(*E*) ∪ *First*(*a*) = {*a*, *e*}, так как *E*∈ *N*ε, а строка *Ea* не порождает пустую строку.

*Follow*(*E*) = {*a*}, поскольку *E* встречается только в правой части продукции *S* → *DEa* (из тех же соображений, что и при вычислении *Follow*(*C*), не учитываем праворекурсивную продукцию *E* → *eE*), а *First*(*a*) = {*a*}.

Вычислим множества направляющих символов *DS* (в учебных целях вычислим для всех продукций, хотя по определению достаточно вычислить только для альтернативных продукций):

*DS*(*S* → *AB*) = *First*(*AB*) = *First*(*A*) = {*a*, *f*},

*DS*(*S* → *DEa*) = *First*(*D*) ∪ *First*(*E*) ∪ *First*(*a*) = {*a*, *d*, *e*},

*DS*(*A* → *ac*) = *First*(*ac*) = {*a*},

*DS*(*A* → *f*) = *First*(*f*) = {*f*},

*DS*(*B* → *bC*) = *First*(*bC*) = {*b*},

*DS*(*C* → *bC*) = *First*(*bC*) = {*b*},

*DS*(*C* → ε) = *Follow*(*C*) = {⊥},

*DS*(*D* → *dD*) = *First*(*dD*) = {*d*},

*DS*(*D* → ε) = *Follow*(*D*) = {*a*, *e*},

*DS*(*E* → *eE*) = *First*(*eE*) = {*e*},

*DS*(*E* → ε) = *Follow*(*E*) = {*a*}.

Таким образом, грамматика не является *LL*(1)-грамматикой, поскольку для *S*-продукций *S* → *AB* и *S* → *DEa* множества направляющих символов пересекаются:

*DS*(*S* → *AB*) ∩ *DS*(*S* → *DEa*) = {*a*, *f*} ∩ {*a*, *d*, *e*} = {*a*}.

### Основные приемы преобразования КС-грамматик в *LL*(1)-форму

Не существует полностью универсального автоматического процесса преобразования грамматик в *LL*(1)-форму. Это связано с тем, что данная проблема алгоритмически неразрешима. Отсутствие общего решения проблемы не означает невозможности ее решения для частных случаев.

Во многих случаях КС-грамматики, которые не обладают *LL*(1)-свойствами, можно преобразовать в *LL*(1)-грамматики с помощью левой факторизации (см. разд. 1.6.5). Рассмотрим это преобразование на примере следующей грамматики:

*S* → *aSb*⎪*aSc*⎪ε

Определим множества направляющих символов (учитывая, что грамматика пополнена маркером конца строки ⊥, т. е. в предположении, что в грамматике имеется продукция *S'* → *S*⊥):

*DS*(*S* → *aSb*) = {*a*},

*DS*(*S* → *aSc*) = {*a*},

*DS*(*S* → ε) = *Follow*(*S*) = {*b*, *c*, ⊥}.

Направляющий символ *a* является общим для двух первых продукций, т. е. это не *LL*(1)-грамматика. Применим для них правило левой факторизации (общим префиксом является *aS*). В результате получится *LL*(1)-грамматика

*S* → *aSX*⎪ε

*X* → *b*⎪*с*

Таким образом, факторизация как бы выносит за скобки направляющие символы.

Многие языки программирования содержат такие часто используемые конструкции, как списки. Поэтому рассмотрим описание различных типов списков с помощью *LL*(1)-грамматик.

Грамматика *L* → *a* **;** *L*⎪*a* для порождения списка символов *a*, разделенных символом-разделителем (для определенности в качестве символа-разделителя используем точку с запятой), не обладает *LL*(1)-свойствами. Применив левую факторизацию, получим *LL*(1)-грамматику

*L* → *aR*

*R* → **;** *aR*⎪ε

Аналогично, применив левую факторизацию к продукциям *L* → *aL*⎪*a*, порождающих список символов *a* без символов-разделителей, получим *LL*(1)-грамматику

*L* → *aR*

*R* → *aR*⎪ε

Рассмотренные грамматики не допускают пустой список. Если это необходимо, то достаточно добавить в грамматики продукцию *L* → ε. Для списка из нуля или более символов *a* возможно применение грамматики *L* → *aL*⎪ε, но иногда она может оказаться непригодной для трансляции.

Следует заметить, что грамматика, содержащая левую рекурсию, не является *LL*(1)-грамматикой. Рассмотрим продукции

*A* → *A*α (леворекурсивная продукция по *А*),

*А* → *а*

*DS*(*A* → *A*α) = {*a*} и *DS*(*A* → *a*) = {*a*}, т. е. множества направляющих символов пересекаются.

По тем же причинам грамматика, содержащая леворекурсивный цикл, не может быть *LL*(1)-грамматикой.

Как уже отмечалось, левую рекурсию всегда можно исключить из грамматики, преобразовав ее в правую рекурсию, которая не вызывает никаких проблем для нисходящего разбора (см. разд. 1.6.3 и 1.6.4).

После устранения левой рекурсии в соответствии с преобразованием из разд. 1.6.3 для получения *LL*(1)-грамматики всегда требуется последующая левая факторизация. Можно сформулировать правило, объединяющее эти преобразования.

Пусть множество продукций содержит леворекурсивные продукции *A* → *A*α1⎪*A*α2⎪…⎪*A*α*m* и все остальные *A*-продукции *A* → β1⎪β2⎪…⎪β*n*, не являющиеся леворекурсивными. Результатом устранения левой рекурсии будут продукции

*A* → β1⎪β2⎪…⎪β*n*⎪β1*A'*⎪β2*A'*⎪…⎪β*nA'*

*A'* → α1⎪α2⎪…⎪α*m*⎪α1*A'*⎪α2*A'*⎪…⎪α*mA'*

Как видим, имеются продукции с общими префиксами, поэтому выполним левую факторизацию:

*A* → β1*X*⎪β2*X*⎪…⎪β*nX*

*X* → ε⎪*A'*

*A'* → α1*X*⎪α2*X*⎪…⎪α*mX*

Выполнив замену вхождений нетерминала *A'*, получим окончательный результат:

*A* → β1*X*⎪β2*X*⎪…⎪β*nX*

*X* → ε⎪α1*X*⎪α2*X*⎪…⎪α*mX*

Таким образом, модифицированное правило устранения левой рекурсии можно сформулировать следующим образом.

Пусть множество продукций содержит леворекурсивные продукции *A* → *A*α1⎪*A*α2⎪…⎪*A*α*m* и остальные *A*-продукции *A* → β1⎪β2⎪…⎪β*n*, не являющиеся леворекурсивными. Тогда новая эквивалентная грамматика будет иметь продукции вида:

*A* → β1*X*⎪β2*X*⎪…⎪β*nX*

*X* → ε⎪α1*X*⎪α2*X*⎪…⎪α*mX*

Для иллюстрации преобразуем в *LL*(1)-форму следующую грамматику (предполагая наличие продукции *E'* → *E*⊥):

*E* → *E* + *T*⎪*T*

*T* → *T* × *F*⎪*F*

*F* → (*E*)⎪*i*

Она не является *LL*(1)-грамматикой, поскольку содержит леворекурсивные продукции. Применим рассмотренное выше модифицированное правило устранения левой рекурсии:

*E* → *TX*

*X* → + *TX*⎪ε

*T* → *FY*

*Y* → × *FY*⎪ε

*F* → (*E*)⎪*i*

Это *LL*(1)-грамматика. Покажем это, определив множества направляющих символов для продукций с одинаковыми левыми частями:

*DS*(*X* → + *TX*) = {+},

DS(*X* → ε) = {), ⊥},

*DS*(*Y* → × *FY*) = {×},

DS(*Y* → ε) = {+, ), ⊥},

*DS*(*F* → (*E*)) = {(},

DS(*F* → *i*) = {*i*}.

Множества направляющих символов всех пар альтернативных продукций не пересекаются, что позволяет при разборе строки детерминированно выбирать нужную продукцию.

Процесс вывода строки *i*×(*i*+*i*)⊥ соответствует следующей левосторонней схеме вывода:

*E'* ⇒ *E*⊥ ⇒ *TX*⊥ ⇒ *FYX*⊥ ⇒ *iYX*⊥ ⇒ *i*×*FYX*⊥ ⇒ *i*×(*E*)*YX*⊥

⇒ *i*×(*TX*)*YX*⊥ ⇒ *i*×(*FYX*)*YX*⊥ ⇒ *i*×(*iYX*)*YX*⊥ ⇒ *i*×(*iX*)*YX*⊥

⇒ *i*×(*i*+*TX*)*YX*⊥ ⇒ *i*×(*i*+*FYX*)*YX*⊥ ⇒ *i*×(*i*+*iYX*)*YX*⊥

⇒ *i*×(*i*+*iX*)*YX*⊥ ⇒ *i*×(*i*+*i*)*YX*⊥ ⇒ *i*×(*i*+*i*)*X*⊥ ⇒ *i*×(*i*+*i*)⊥.

К сожалению, процесс факторизации нельзя распространить на общий случай. Следующий пример показывает, что может произойти. Рассмотрим грамматику

*S* → *Ac*⎪*Bd*

*A* → *eAf*⎪*a*

*B* → *eBg*⎪*b*

Первые две *S*-продукции в своих множествах направляющих символов содержат общий символ *e*. Для проведения факторизации предварительно выполним замену вхождений нетерминалов *A* и *B*, чтобы направляющие символы явно присутствовали в этих продукциях:

*S* → *eAfc*⎪*ac*⎪*eBgd*⎪*bd*

Выполняя факторизацию, эти продукции можно заменить следующими:

*S* → *ac*⎪*bd*⎪*eS*1

*S*1 → *Afc*⎪*Bgd*

Продукции для *S*1 аналогичны первоначальным продукциям для *S* и имеют пересекающиеся множества направляющих символов. Можно повторить преобразование этих продукций, как это было сделано с продукциями для *S*:

*S*1 → *eAffc*⎪*afc*⎪*eBggd*⎪*bgd*

В результате факторизации получим

*S*1 → *afc*⎪*bgd*⎪*eS*2

*S*2 → *Affc*⎪*Bggd*

Продукции для *S*2 аналогичны продукциям для *S*1 и *S*, но длиннее их, и теперь очевидно, что этот процесс бесконечный.

Это означает, что все попытки преобразовать грамматику в *LL*(1)-форму не всегда приводят к результату. Это является прямым следствием того, что *LL*(1)-языки являются только подклассом более широкого класса языков, допускающих детерминированный разбор.

## Метод рекурсивного спуска

Метод рекурсивного спуска – хорошо известный и легко реализуемый детерминированный метод нисходящего разбора для *LL*(*k*)-грамматик. В простейшей форме метод рекурсивного спуска предоставляет удобную возможность построения синтаксического анализатора языка, порожденного *LL*(1)-грамматикой.

Самый простой (но не самый эффективный) способ реализации рекурсивного спуска заключается в следующем. Каждому символу из множества *VT* ∪ *VN* ставится в соответствие единственная процедура, распознающая строку, порождаемую этим символом.

Если *a* ∈ *VT*, то соответствующая процедура (обозначим *рroc\_а*) обеспечивает проверку на совпадение очередного входного символа с терминалом *a*, в случае совпадения реализуется ввод следующего входного символа, в противном случае фиксируется синтаксическая ошибка.

Если *A* ∈ *VN* и имеются *A*-продукции вида *А* → α1⎪α2⎪...⎪α*n*, то соответствующая процедура (обозначим *рroc\_A*), сравнив очередной входной символ с элементами множеств направляющих символов, определяет, какой из *A*-продукций нужно воспользоваться для осуществления следующего шага вывода. Если выбрана продукция *А* → α*i*, и α*i* = *Х*1*Х*2...*Хm*, *Xj* ∈ *VT* ∪ *VN*, 1 ≤ *j* ≤ *m*, то реализуется последовательность вызовов процедур *proc\_Х*1; *рroc\_Х*2; ...; *рroc\_Хm*.

Рассмотрим *LL*(1)-грамматику со следующими продукциями, предполагая наличие продукции *S'* → *S*⊥ (для каждой продукции справа указаны множества направляющих символов):

*S* → *AbB* {*a*, *b*, *c*, *e*}

*S* → *d* {*d*}

*A* → *aAb* {*a*}

*A* → *edAb* {*e*}

*A* → *B* {*b*, *c*}

*B* → *cSd* {*c*}

*B* → ε {*b*, *d*, ⊥}

Пусть процедура *read*(*sym*) считывает из входной сроки очередной символ и присваивает его значение переменной *sym* (соответствует обращению к сканеру за очередным токеном), процедура *error* каким-либо образом обрабатывает синтаксическую ошибку и прекращает разбор, процедура *stop* завершает синтаксический разбор. Тогда синтаксический анализатор можно представить множеством соответствующих процедур.

Для всех терминалов *a* ∈ *VT* = {*a*, *b*, *c*, *d*, *e*} процедуры *рroc\_а* имеют вид:



Процедуры для нетерминалов:







Процесс разбора начинается со следующих действий (переменная *sym* после выполнения *read*(*sym*) будет содержать первый символ входной строки):

*read*(*sym*); *proc\_S*; **if** *sym* = ⊥ **then** *stop* **else** *error*.

На практике с целью сокращения числа вызовов процедур такие процедуры ставят в соответствие только для нетерминалов, включая операции ввода и анализа терминалов в эти процедуры. При составлении процедур *proc\_A* для всех *A* ∈ *VN* необходимо придерживаться следующих правил:

а) реализовать выбор из *A*-продукций вида *А* → α1⎪α2⎪...⎪α*n* соответствующей продукции, сравнив очередной входной символ с элементами множеств направляющих символов; пусть выбрана продукция *А* → α*i*, и α*i* = *Х*1*Х*2...*Хm*, *Xj* ∈ *VT* ∪ *VN*, 1 ≤ *j* ≤ *m*;

б) если *Xj* ∈ *VN* (текущим символом правой части продукции является нетерминал), ему соответствует вызов процедуры, распознающей порождаемую этим нетерминалом строку;

в) если *Xj* ∈ *VT* (текущим символом правой части продукции является терминал), ему соответствует действие, связанное с проверкой на совпадение этого терминала с текущим входным символом. Если символы совпадают, то входной символ принимается и реализуется ввод очередного входного символа, в противном случае фиксируется синтаксическая ошибка;

г) если α*i* = ε (имеет место ε-продукция), реализуется переход на конец процедуры, т. е. никакие действия не производятся.

Учитывая эти правила, процедуры для нетерминалов рассматриваемой грамматики можно написать следующим образом:







Для разбора строк языка может потребоваться много рекурсивных вызовов процедур, соответствующих нетерминалам грамматики. Если представить грамматику несколько иным способом, в ряде случаев рекурсию можно заменить итерацией. Например, пусть дана грамматика со следующими продукциями:

*S* → *cAdBe*

*A* → *afA*⎪*a*

*B* → *bfB*⎪*b*

Ее можно представить следующим образом (используя нотацию регулярных выражений):

*S* → *cAdBe*

*A* → *a*(*fa*)\*

*B* → *b*(*fb*)\*

Тогда процедуры для нетерминалов *A* и *B* можно представить следующим образом:





Такая замена рекурсии итерацией обычно позволяет делать анализатор более эффективным, за счет сокращения вызовов процедур.

Метод рекурсивного спуска является одним из наиболее эффективных способов создания компиляторов. Преимущества написания рекурсивного нисходящего анализатора очевидны. Основные из них – это скорость написания анализатора на основании соответствующей грамматики. Другое преимущество заключается в соответствии между грамматикой и анализатором, благодаря которому увеличивается вероятность того, что анализатор окажется правильным, или, по крайней мере, того, что ошибки будут носить простой характер.

Недостатки этого метода, хотя и менее очевидны, но не менее реальны. Из-за большого числа вызовов процедур во время синтаксического анализа анализатор становится относительно медленным. Кроме того, он может быть довольно большим по сравнению с анализаторами, основанными на табличных методах разбора. Несмотря на то, что данный метод способствует включению в анализатор действий по генерированию кода, это неизбежно приводит к смешиванию различных фаз компиляции. Это снижает надежность компиляторов или усложняет обращение с ними и привносит в анализатор зависимость от машины.

## Табличные методы нисходящего разбора

Табличные методы синтаксического анализа аналогичны рекурсивному спуску. Здесь исключаются многочисленные вызовы процедур благодаря представлению грамматики в табличном виде (*таблицы разбора*, *управляющие таблицы*) и использованию независящего от анализируемого языка модуля компилятора, проводящего синтаксический разбор по таблице.

Основным достоинством табличных методов разбора является то, что модуль синтаксического анализатора можно применять многократно в компиляторах для различных языков, изменив только содержимое таблицы разбора. Процесс формирования таблиц разбора для *LL*(1)-грамматик обычно носит детерминированный характер. Поэтому этот процесс можно легко автоматизировать, написав программу для получения соответствующей таблицы разбора. В результате сроки проектирования компиляторов существенно сокращаются.

Таблицы разбора организуются таким образом, что модуль синтаксического анализа компилятора всегда указывает на то место в синтаксисе, которое соответствует текущему входному символу. Модулю требуется стек, содержащий последовательность символов грамматики или адреса возврата при обработке новой порождающей продукции, соответствующей какому-либо нетерминалу. Представление синтаксиса в таблице должно быть таким, чтобы обеспечить эффективность синтаксического анализатора в отношении скорости работы.

Возможны различные виды таблиц разбора, которые являются, по сути, различными формами представления магазинного автомата, принимающего данный контекстно-свободный язык.

### Таблица переходов МП-автомата

Часто в качестве управляющей таблицы используют таблицу переходов детерминированного МП-автомата (возможно, слегка модифицированную). В общем случае недетерминированный МП-автомат *M* = (*K*, *T*, Γ, δ, *k*0, *Z*0, *F*), распознающий язык, заданный КС-грамматикой *G* = (*VT*, *VN*, *P*, *S*), определяется как *M* = ({*k*}, *VT*, *VT* ∪*VN*, δ, *k*, *S*, {*k*}), где функция переходов δ определяется следующим образом (см. разд. 3.2):

δ(*k*, ε, *A*) = {(*k*, α)⎪*A* → α ∈ *Р*} для всех *A* ∈ *VN*;

δ(*k*, *a*, *a*) = {(*k*, ε)} для всех *а* ∈ *VT*.

Наличие множеств направляющих символов *DS*(*A* → α) для продукций *LL*(1)-грамматики позволяет определить детерминированный МП-автомат, распознающий язык, заданный соответствующей *LL*(1)-грамматикой. Для этого модифицируется определение функции переходов δ:

для всех продукций *A* → α ∈ *Р* δ(*k*, *a*, *A*) = (*k*, α) для всех *a* ∈ *DS*(*A* → α);

δ(*k*, *a*, *a*) = (*k*, ε) для всех *а* ∈ *VT*.

Тогда таблицу переходов МП-автомата, используемую в качестве управляющей таблицы, можно задать отображением множества (*VT* ∪*VN*)×*VT* (маркер конца ввода ⊥ ∈ *VT* выполняет также функцию *маркера дна* стека; если он находится в вершине стека, то стек пуст). Таким образом, строкам таблицы соответствуют элементы стекового алфавита (объединение множеств терминалов и нетерминалов), столбцы соответствуют входному алфавиту (множеству терминалов).

Элементами управляющей таблицы *M* являются:

1. Правая часть продукции *A* → α, соответствует значению функции переходов δ(*k*, *a*, *A*) = (*k*, α). В ряде модификаций в качестве элемента используют пару (α, *i*), где *i* – номер продукции, или просто номер *i* продукции.

2. Элемент «*выброс*», соответствует значению функции переходов δ(*k*, *a*, *a*) = (*k*, ε), *a* ≠ ⊥.

3. Элемент «*допуск*», соответствует значению функции переходов δ(*k*, ⊥, ⊥) = (*k*, ε).

4. Элемент «*ошибка*», это незаполненный элемент таблицы, соответствует синтаксической ошибке.

В начальной конфигурации входной буфер содержит анализируемую строку *w*⊥, *w* ∈ (*VT* – {⊥})\*, текущим входным символом является первый символ этой строки, стек содержит *S*⊥, где *S* – начальный символ грамматики. При нисходящем анализе содержимое стека удобно представлять строкой, в которой самый левый символ строки считается верхним символом стека.

По элементу *M*[*X*, *a*] управляющей таблицы, где *X* – символ в вершине стека, *a* – текущий входной символ, определяются действия, предпринимаемые синтаксическим анализатором в процессе разбора входной строки:

1. Если *M*[*X*, *a*] = α (в этом случае символ *X* в вершине стека может быть только нетерминалом), символ *X* в вершине стека замещается строкой α.

2. Если *M*[*X*, *a*] = «*выброс*» и *a* ≠ ⊥ (в этом случае символ *X* в вершине стека может быть только терминалом *a* ≠ ⊥, т. е. символ в вершине стека совпадает с текущим входным символом), символ *a* исключается из стека и реализуется переход к анализу следующего входного символа. Это означает, что входной символ *a* принимается синтаксическим анализатором.

3. Если *M*[*X*, *a*] = «*допуск*» (случай возможен только для *X* = ⊥ и *a* = ⊥), входная строка принимается, разбор завершается.

4. Если *M*[*X*, *a*] = «*ошибка*», разбор прекращается и формируется соответствующее сообщение об ошибке.

Рассмотрим *LL*(1)-грамматику со следующими продукциями, предполагая наличие продукции *S'* → *S*⊥ (для каждой продукции справа указаны множества направляющих символов):

*S* → *TC* {*a*, *c*, ⊥}

*T* → *aTb* {*a*}

*T* → ε {*b*, *c*, ⊥}

*C* → *cC* {*c*}

*C* → ε {⊥}

Управляющая таблица для этой грамматики представлена на рис. 3.2. Процесс разбора строки *aabbc*⊥, которая выводится в соответствии с левосторонней схемой

*S'* ⇒ *S*⊥ ⇒ *TC*⊥ ⇒ *aTbC*⊥ ⇒ *aaTbbC*⊥ ⇒ *aabbC*⊥ ⇒

⇒ *aabbcC*⊥ ⇒ *aabbc*⊥,

показан на рис. 3.3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *a* | *b* | *c* | ⊥ |
| *S* | *TC* |  | *TC* | *TC* |
| *T* | *aTb* | ε | ε | ε |
| *C* |  |  | *cC* | ε |
| *a* | *выброс* |  |  |  |
| *b* |  | *выброс* |  |  |
| *c* |  |  | *выброс* |  |
| ⊥ |  |  |  | *допуск* |

Рис. 3.2. Управляющая таблица

| Входная строка | Содержимое стека | Комментарии |
| --- | --- | --- |
| *aabbc*⊥ | *S*⊥ | *M*[*S*, *a*] = *TC*, замещение *S* строкой *TC* |
| *aabbc*⊥ | *TC*⊥ | *M*[*T*, *a*] = *aTb*, замещение *T* строкой *aTb* |
| *aabbc*⊥ | *aTbC*⊥ | *M*[*a*, *a*] = «*выброс*», *a* принимается |
| *abbc*⊥ | *TbC*⊥ | *M*[*T*, *a*] = *aTb*, замещение *T* строкой *aTb* |
| *abbc*⊥ | *aTbbC*⊥ | *M*[*a*, *a*] = «*выброс*», *a* принимается |
| *bbc*⊥ | *TbbC*⊥ | *M*[*T*, *b*] = ε, замещение *T* пустой строкой ε |
| *bbc*⊥ | *bbC*⊥ | *M*[*b*, *b*] = «*выброс*», *b* принимается |
| *bc*⊥ | *bC*⊥ | *M*[*b*, *b*] = «*выброс*», *b* принимается |
| *c*⊥ | *C*⊥ | *M*[*C*, *c*] = *cC*, замещение *T* строкой *cC* |
| *c*⊥ | *cC*⊥ | *M*[*c*, *c*] = «*выброс*», *c* принимается |
| ⊥ | *C*⊥ | *M*[*C*, ⊥] = ε, замещение *C* пустой строкой ε |
| ⊥ | ⊥ | *M*[⊥, ⊥] = «*допуск*», разбор успешно завершен |

Рис. 3.3. Процесс разбора строки *aabbc*⊥

Следует заметить, что в любой строке управляющей таблицы, соответствующей терминалу, возможен только один элемент «*выброс*» («*допуск*» для ⊥), а все остальные элементы являются элементами «*ошибка*». Действия, связанные с этими элементами, реализуются только в том случае, если в вершине стека находится терминал (включая и маркер ⊥). Такую ситуацию легко можно распознать и включить соответствующие действия в алгоритм синтаксического анализа. Поэтому на практике для уменьшения размеров в управляющую таблицу обычно включают только строки, соответствующие нетерминалам. Соответствующий алгоритм синтаксического анализа с применением модифицированной управляющей таблицы представлен алгоритмом 3.1, где

*SP* – рабочий стек синтаксического анализатора (в начальном состоянии он должен содержать *S*⊥, где *S* – начальный символ грамматики, ⊥ – маркер конца ввода);

процедура *read*(*a*) считывает из входной сроки очередной символ и присваивает его значение переменной *a* (соответствует обращению к сканеру за очередным токеном);

процедура *syntax*\_*error* каким-либо образом обрабатывает синтаксическую ошибку и прекращает разбор.

**Алгоритм 3.1.** Предиктивный синтаксический анализ с применением модифицированной управляющей таблицы

**Вход:** Управляющая таблица, анализируемая строка

**Выход:** Сообщения о результатах синтаксического анализа



### Специальные *LL*(1)-таблицы разбора

В ряде случаев для повышения эффективности синтаксического анализа разрабатывают специальные таблицы разбора. В качестве примера рассмотрим достаточно простой и понятный вид таблицы разбора для *LL*(1)-грамматик [18].

*LL*(1)-таблица разбора представляет собой набор строк. Каждая строка содержит поля (столбцы):

а) список терминалов – *terminals*,

б) поле переходов – *jump*,

в) поле приема – *ассept*,

г) поле стека – *staсk*,

д) поле возврата – *return*,

е) поле ошибки – *error*.

Область значений поля *jump* – неотрицательные целые числа (номера строк таблицы), а область значений полей *ассept*, *staсk*, *return* и *error* – {**true**, **false**}.

В таблице каждому шагу процесса разбора соответствует один элемент. Поэтому в нее включаются по одному элементу для каждой продукции грамматики (для нетерминала из левой части) и на каждый экземпляр терминала или нетерминала правой части продукции. Кроме того, в таблицу включаются элементы на каждую реализацию пустой строки в правой части продукции (для ε-продукций). Первая строка таблицы соответствует первой продукции с начальным нетерминалом в левой части. Для этого исходная *LL*(1)-грамматика представляется в виде специальной схемы, в которой каждому символу грамматики соответствует номер строки таблицы разбора. Очень важно, чтобы номера нетерминалов в левых частях альтернативных продукций непосредственно следовали друг за другом. Это необходимо для того, чтобы в случае, если анализируемый символ не принадлежит множеству направляющих символов одной продукции, увеличением номера на единицу перейти к проверке следующей альтернативной продукции.

Правила заполнения таблицы заключаются в следующем.

1. Поле *terminals*. Если строка таблицы соответствует продукции (т. е. нетерминалу в левой части), в поле заносится множество направляющих символов данной продукции. Если строка соответствует нетерминалу из правой части продукции, в поле записывается множество терминалов, являющееся объединением множеств направляющих символов всех продукций с данным нетерминалом в левой части. Если строка соответствует терминалу, в поле заносится этот терминал. Если строка соответствует правой части ε-продукции, в поле заносится множество направляющих символов этой ε-продукции.

2. Поле *jump*. В поле *jump* записывается номер строки следующего элемента обработки.

3. Поле *accept*. Устанавливается значение **true**, если строка таблицы соответствует терминалу, в противном случае – **false**. Означает, что если просматриваемый символ входной строки совпадает с данным терминалом, то этот символ можно принять и перейти к анализу следующего входного символа.

4. Поле *stack*. Устанавливается значение **true**, если строка таблицы соответствует нетерминалу из правой части продукции за исключением случая, когда этот нетерминал является крайним правым символом продукции. Поле показывает, что необходимо занести в стек точку возврата (номер строки для символа, непосредственно следующего в продукции за данным нетерминалом), поскольку далее осуществляется переход к обработке первой продукции, относящейся к данному нетерминалу. Очевидно, что если нетерминал является крайним правым символом продукции, нет необходимости помещать в стек адрес возврата. Во всех остальных случаях поле принимает значение **false**.

5. Поле *return*. Устанавливается значение **true**, если строка таблицы соответствует крайнему правому терминалу продукции или правой части ε-продукции, в остальных случаях – **false**. Означает, что завершена обработка продукции и необходимо вернуться к точке возврата, сохраненной в стеке. При этом значение поля *jump* во внимание не принимается и устанавливается нулевое значение.

6. Поле *error*. Если имеется несколько альтернативных продукций, например *k*, то в строках, соответствующих первым *k* – 1 продукциям, поле ошибки принимает значение **false**, для *k*-й продукции – **true**. Это связано с тем, что если входной символ не является направляющим, то это еще не означает ошибку, поскольку он может быть направляющим для какой-либо другой альтернативной продукции, проверяемой на следующем этапе. Если же анализируемый символ не будет направляющим ни для одной из альтернативных продукций (а это выяснится только после проверки последней из них), тогда можно констатировать наличие синтаксической ошибки. Во всех остальных случаях поле ошибки принимает значение **true**.

В процессе синтаксического анализа осуществляется ряд различных шагов:

1. Проверка текущего входного символа с целью выяснения, не является ли данный символ направляющим для какой-либо конкретной правой части продукции. Если этот символ не направляющий и имеется альтернативная продукция, то она проверяется на следующем этапе.

2. Проверка терминала, появляющегося в правой части продукции.

3. Проверка нетерминала. Она заключается в проверке нахождения текущего входного символа в одном из множеств направляющих символов для данного нетерминала, помещении в стек адреса возврата и переходу к первой продукции, относящейся к этому нетерминалу.

Модуль синтаксического анализатора обрабатывает элемент таблицы разбора и определяет следующий элемент для обработки. Поле перехода *jump* дает следующий элемент обработки, если значение поля возврата *return* не окажется равным **true**. В последнем случае адрес следующего элемента берется из стека (что соответствует концу продукции). Если же текущий входной символ отсутствует в списке терминалов и значение поля ошибки окажется **false**, нужно обрабатывать следующий элемент таблицы с тем же текущим входным символом.

Псевдокод алгоритма синтаксического анализа представлен алгоритмом 3.2. Переменная *nextsym* определяет, надо ли считывать новый входной символ до обработки следующего элемента таблицы разбора. Например, *nextsym* = **false**, когда текущий входной символ не является направляющим для какой-либо конкретной продукции и требуется исследовать множество направляющих символов следующей альтернативной продукции. Если символ не содержится в текущем множестве направляющих символов и поле ошибки *error* будет **true**, то выдается сообщение о синтаксической ошибке. Если поле *stack* обрабатываемой *i*-й строки таблицы разбора (*Т*[*i*]) имеет значение **true**, то до перехода к адресу, задаваемому полем перехода *jump*, в стек помещается адрес следующей строки таблицы.

**Алгоритм 3.2.** Синтаксический анализ с применением специальной *LR*-таблицы разбора

**Вход:** *LR*-таблица разбора, анализируемая строка

**Выход:** Сообщения о результатах синтаксического анализа



Такой модуль синтаксического анализа совершенно не зависит от разбираемого языка и может использоваться в целом ряде компиляторов. Он относительно мал, т. к. большая часть объема памяти, занимаемого синтаксическим анализатором, приходится на таблицу разбора, размер которой пропорционален размеру грамматики.

Построим таблицу разбора для *LL*(1)-грамматики со следующими продукциями и их множествами направляющих символов (предполагая наличие продукции *S'* → *S*⊥):

*S* → *TC* {*a*, *c*, ⊥}

*T* → *aTb* {*a*}

*T* → ε {*b*, *c*, ⊥}

*C* → *cC* {*c*}

*C* → ε {⊥}

Сначала представим грамматику в виде схемы, в которой все символы помечены номерами строк таблицы разбора:



На основании этой схемы легко строится таблица разбора в соответствии с рассмотренными выше правилами заполнения ее полей (рис. 3.4).

|  | *terminals* | *jump* | *accept* | *stack* | *return* | *error* |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | {*a*, *c*, ⊥} | 2 | **false** | **false** | **false** | **true** |
| 2 | {*a*, *b*, *c*, ⊥} | 4 | **false** | **true** | **false** | **true** |
| 3 | {*c*, ⊥} | 10 | **false** | **false** | **false** | **true** |
| 4 | {*a*} | 6 | **false** | **false** | **false** | **false** |
| 5 | {*b*, *c*, ⊥} | 9 | **false** | **false** | **false** | **true** |
| 6 | {*a*} | 7 | **true** | **false** | **false** | **true** |
| 7 | {*a*, *b*, *c*, ⊥} | 4 | **false** | **true** | **false** | **true** |
| 8 | {*b*} | 0 | **true** | **false** | **true** | **true** |
| 9 | {*b*, *c*, ⊥} | 0 | **false** | **false** | **true** | **true** |
| 10 | {*c*} | 12 | **fase** | **false** | **false** | **false** |
| 11 | {⊥} | 14 | **false** | **false** | **false** | **true** |
| 12 | {*c*} | 13 | **true** | **false** | **false** | **true** |
| 13 | {*c*, ⊥} | 10 | **false** | **false** | **false** | **true** |
| 14 | {⊥} | 0 | **false** | **false** | **true** | **true** |

Рис. 3.4. *LL*(1)-таблица разбора

Можно сократить число элементов в таблице разбора. Заметим, что имеет место взаимно однозначное соответствие между нулевым значением поля *jump* и значением **true** поля *return*. Следовательно, вместо проверки поля *return* для определения, следует ли брать номер следующей строки из стека, достаточно проверить поле *jump* на равенство нулю. Это позволяет исключить из таблицы разбора поле (столбец) *return*.

Рассмотрим разбор строки *aabbc*⊥, которая выводится в соответствии со следующей левосторонней схемой

*S'* ⇒ *S*⊥ ⇒ *TC*⊥ ⇒ *aTbC*⊥ ⇒ *aaTbbC*⊥ ⇒ *aabbC*⊥ ⇒

⇒ *aabbcC*⊥ ⇒ *aabbc*⊥.

Процесс разбора показан на рис. 3.5. Содержимое стека представляется строкой, в которой самый левый символ находится в вершине стека, символ ⊥ показывает дно стека, столбец *i* есть номер применяемой строки таблицы разбора.

| Вводимая строка | *i* | Содержимое стека | Комментарии |
| --- | --- | --- | --- |
| *aabbc*⊥ | 1 | 0⊥ | проверка *a*, перейти к строке 2 |
| *aabbc*⊥ | 2 | 0⊥ | проверка *a*, в стек поместить  2 + 1 = 3, перейти к строке 4 |
| *aabbc*⊥ | 4 | 3,0⊥ | проверка *a*, перейти к строке 6 |
| *aabbc*⊥ | 6 | 3,0⊥ | *a* принимается, перейти к строке 7 |
| *abbc*⊥ | 7 | 3,0⊥ | проверка *a*, в стек поместить  7 + 1 = 8, перейти к строке 4 |
| *abbc*⊥ | 4 | 8,3,0⊥ | проверка *a*, перейти к строке 6 |
| *abbc*⊥ | 6 | 8,3,0⊥ | *a* принимается, перейти к строке 7 |
| *bbc*⊥ | 7 | 8,3,0⊥ | проверка *a*, в стек поместить  7 + 1 = 8, перейти к строке 4 |
| *bbc*⊥ | 4 | 8,8,3,0⊥ | *b* не совпадает с *a*, *error* = **false**,  перейти к строке 4 + 1 = 5 |
| *bbc*⊥ | 5 | 8,8,3,0⊥ | проверка *b*, перейти к строке 9 |
| *bbc*⊥ | 9 | 8,8,3,0⊥ | проверка *b*, взять из стека 8,  перейти к строке 8 |
| *bbc*⊥ | 8 | 8,3,0⊥ | *b* принимается, взять из стека 8,  перейти к строке 8 |
| *bc*⊥ | 8 | 3,0⊥ | *b* принимается, взять из стека 3,  перейти к строке 3 |
| *c*⊥ | 3 | 0⊥ | проверка *c*, перейти к строке 10 |
| *c*⊥ | 10 | 0⊥ | проверка *c*, перейти к строке 12 |
| *c*⊥ | 12 | 0⊥ | *c* принимается, перейти к строке 13 |
| ⊥ | 13 | 0⊥ | проверка *c*, перейти к строке 10 |
| ⊥ | 10 | 0⊥ | ⊥ не совпадает с *c*, *error* = **false**,  перейти к строке 10 + 1 = 11 |
| ⊥ | 11 | 0⊥ | проверка ⊥, перейти к строке 14 |
| ⊥ | 14 | 0⊥ | проверка '⊥', взять из стека 0,  перейти к строке 0 |
| ⊥ | 0 | ⊥ | стек пуст, разбор успешно завершен |

Рис. 3.5. Процесс разбора строки *aabbc*⊥

В процессе разбора некоторые терминалы проверяются несколько раз. Такой неоднократной проверки можно избежать, если отложить обнаружение некоторых синтаксических ошибок на более поздний этап (поздний в смысле шагов разбора, но не считанного текста). Тогда можно сократить число строк в таблице разбора.

С целью экономии памяти можно упаковывать элементы таблицы разбора в машинные слова, но это, очевидно, замедлит работу синтаксического анализатора.

Можно написать программу для получения соответствующей таблицы разбора. Модуль синтаксического анализа можно применять многократно для различных компиляторов, так что при наличии соответствующих программных средств можно получить *LL*(1)-анализатор из *LL*(1)-грамматики с минимальными затратами времени.

*LL*(1)-метод разбора имеет следующие достоинства:

1) никогда не требуется возврат, поскольку это детерминированный метод;

2) время разбора примерно пропорционально длине программы;

3) имеются хорошие диагностические характеристики и возможность исправления ошибок, т. к. синтаксические ошибки распознаются по первому неприемлемому символу, а в таблице разбора есть список возможных символов продолжения;

4) таблица разбора меньше, чем соответствующие таблицы в других методах разбора.

Упражнения

3.1. Преобразовать в *LL*(1)-форму следующие грамматики:

а) *S* → *aBA*⎪*bBbA* в) *S* → *Aa*⎪*BcA* д) *S* → *abAd*

*A* → *abA*⎪*c* *A* → *abS*⎪ε *A* → *B*⎪*BeA*

*B* → *a*⎪*ab* *B* → *AS*⎪*cA*⎪ε *B* → *fD*

б) *S* → *ABC* г) *S* → *AB* *D* → *t*⎪*teD*

*A* → *a*⎪*Cb*⎪ε *A* → *Ba*⎪ε

*B* → *c*⎪*dA*⎪ε *B* → *Cb*⎪*C*

*C* → *e*⎪*f* *C* → *c*⎪ε

3.2. Построить *LL*(1)-грамматики для следующих языков:

а) {*ancb*2*n* | *n* ≥ 0};

б) {*anbn* | *n* ≥ 0};

в) {(*abb*)*n* | *n* ≥ 0}.

3.3. Для *LL*(1)-грамматик, полученных в упр. 3.1 и 3.2, разработать процедуры для метода рекурсивного спуска.

3.4. Для *LL*(1)-грамматик, полученных в упр. 3.1 и 3.2, построить управляющие таблицы согласно разд. 3.6.1.

3.5. Для *LL*(1)-грамматик, полученных в упр. 3.1 и 3.2, построить *LL*(1)-таблицы разбора согласно разд. 3.6.2.

# ВОСХОДЯЩИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ

## Построение дерева разбора

При восходящем синтаксическом анализе дерево разбора для входной строки строится снизу вверх, начиная с листьев и завершая корнем дерева. Этот процесс можно рассматривать как свертку входной строки к начальному символу грамматики. Входная строка анализируется слева направо в поисках подстроки, которая может быть свернута, т. е. ищется правая часть продукции грамматики, совпадающая с подстрокой. Если такая продукция найдена, то соответствующая подстрока заменяется нетерминалом левой части продукции. В результате такой замены может быть получена сентенциальная форма грамматики. Процесс повторяется до тех пор, пока входная строка не преобразуется в начальный символ грамматики (корень дерева разбора).

Действие замены подстроки нетерминалом левой части продукции называется *сверткой*, а свертываемая подстрока – *основой правосторонней сентенциальной формы* (или просто *основой*). Таким образом, основа – это подстрока, совпадающая с правой частью продукции, свертка которой в левую часть продукции представляет собой один шаг правосторонней схемы вывода в обратном направлении. Формально основой правосторонней сентенциальной формы αβ*w* (α, β ∈ (*VT* ∪ *VN*)\*, *w* ∈ *VT*\*) является подстрока β, если существует продукция *A* → β такая, что β может быть заменена нетерминалом *A* для получения предыдущей сентенциальной формы в правосторонней схеме вывода

.

Следует обратить внимание на то, что подстрока *w* может состоять только из терминалов. Из определения следует, что не любая подстрока β, соответствующая правой части некоторой продукции *A* → β, является основой, поскольку свертка для этой продукции может привести к строке, которая не может быть в последующем свернута к начальному символу грамматики.

Пример построения дерева разбора снизу вверх для грамматики с продукциями

*S* → *AB*

*A* → *aA* | *a*

*B* → *bB* | *b*

при разборе строки *aabb* представлен на рис. 4.1. Это дерево соответствует правосторонней схеме вывода

*S* ⇒ *AB* ⇒ *AbB* ⇒ *Abb* ⇒ *aAbb* ⇒ *aabb* .

*а*

*б*

*в*

*д*

*г*

Рис. 4.1. Процесс построения дерева разбора снизу вверх

В приведенном примере строка *aabb* представляет собой правостороннюю сентенциальную форму, основой которой является второй символ *a* (но не первый!) в соответствии с продукцией *A* → *a*. В результате свертки получается правосторонняя сентенциальная форма *aAbb* (рис. 4.1, *а*), основой которой является подстрока *aA* в соответствии с продукцией *A* → *aA*. В результате свертки получается строка *Abb* (рис. 4.1, *б*). Такая последовательность операций определения основы и свертки продолжается до полного построения дерева разбора: основа сентенциальной формы *Abb* – второй символ *b* (но не первый!) в соответствии с продукцией *B* → *b*, результат свертки – строка *AbB* (рис. 4.1, *в*); основа сентенциальной формы *AbB* – подстрока *bB* в соответствии с продукцией *B* → *bB*, результат свертки – строка *AB* (рис. 4.1, *г*); сентенциальная форма *AB* сама является основой в соответствии с продукцией *S* → *AB*, результат свертки – начальный символ грамматики *S*, который является корнем построенного дерева разбора (рис. 4.1, *д*).

Следует заметить, что если бы в строке *aabb* заменили первый символ *a* на нетерминал *A*, то получили бы строку *Aabb*, которую невозможно свернуть в *S*, т. е. первый символ *a* не является основой, хотя и имеется продукция *A* → *a*. По аналогичной причине и первый символ *b* не является основой для строки *Abb*.

Таким образом, при восходящем синтаксическом анализе основной проблемой является поиск и своевременная свертка основы сентенциальной формы на каждом шаге построения дерева разбора.

В синтаксических анализаторах процесс восходящего анализа реализуется с помощью стека, который соответствует части магазинного автомата. При этом анализатор выполняет две основные операции:

1) *перенос*, во время которого считывается символ входной строки и помещается в вершину стека (поэтому основа будет находиться в верхней части стека);

2) *свертка* заключается в замене множества элементов в верхней части стека (основы) на один элемент – нетерминал из левой части соответствующей продукции.

В связи с этим построенные на таких принципах синтаксические анализаторы часто называют анализаторами типа «перенос-свертка».

Процесс анализа строки *aabb* для приведенной выше грамматики показан на рис. 4.2. Содержимое стека представляется строкой, в которой самый правый символ находится в вершине стека, символ ⊥ показывает дно стека.

Разбор считается успешно завершенным, когда в стеке остается только начальный символ грамматики и полностью считана входная строка.

|  |  |  |
| --- | --- | --- |
| Вводимая строка | Содержимое стека | Выполняемые действия |
| *aabb* | ⊥ | Перенос *a* в стек |
| *abb* | ⊥*a* | Перенос *a* в стек |
| *bb* | ⊥*aa* | Свертка для продукции *A* → *a* |
| *bb* | ⊥*aA* | Свертка для продукции *A* → *aA* |
| *bb* | ⊥*A* | Перенос *b* в стек |
| *b* | ⊥*Ab* | Перенос *b* в стек |
| ε | ⊥*Abb* | Свертка для продукции *B* → *b* |
| ε | ⊥*AbB* | Сертка для продукции *B* → *bB* |
| ε | ⊥*AB* | Свертка для продукции *S* → *AB* |
| ε | ⊥*S* | Разбор успешно завершен |

Рис. 4.2. Процесс разбора строки *aabb*

При такой организации восходящего анализа проблема поиска основы и ее своевременной свертки сводится к проблеме разрешения конфликтов типа «перенос/свертка» (анализатор в конкретной ситуации должен знать, какое из этих действий необходимо выполнить) и «свертка/свертка» (если возможны разные свертки, необходимо знать, какую из них выполнить и выполнять ли их вообще). Очевидно, что необходимым условием для выполнения свертки является то, чтобы правая часть какой-либо продукции появилась в верхней части стека. Но это условие не является достаточным. Такие ситуации встречались и в рассмотренном выше примере. Поэтому для разрешения конфликтов анализатор должен иметь дополнительную информацию.

## Грамматики простого предшествования

Одним из наиболее легких подходов к решению проблемы поиска и своевременной свертки основы является реализация восходящего синтаксического анализа для небольшого класса КС-грамматик, называемых *грамматиками простого предшествования*. Технология синтаксического анализа для таких грамматик предполагает введение специальных бинарных отношений между каждой парой символов грамматики (как терминалов, так и нетерминалов). Эти отношения управляют выбором основ сентенциальных форм для последующей их свертки.

Основным недостатком этого метода является применимость его лишь в узком классе грамматик простого предшествования.

### Отношения предшествования

Пусть *G* = (*VT*, *VN*, *P*, *S*) – контекстно-свободная грамматика, а строка α*XY*β – правосторонняя сентенциальная форма, где α, β ∈ (*VT* ∪ *VN*)\*, *X*, *Y* ∈ *VT* ∪ *VN*. В некоторый момент (на одном из этапов процесса последовательных сверток сентенциальной формы) возникает одна из следующих возможных ситуаций:

1. *Y* – самый левый символ (*заголовок*) основы сентенциальной формы, а *X* не входит в основу. В этом случае говорят, что символ *Y* *предшествует* символу *X* (поскольку символ *Y* должен быть свернут раньше символа *X*), и записывают в виде *X**Y*.

2. *X* и *Y* входят в одну и ту же основу. В этом случае говорят, что *X* и *Y* имеют *равное предшествование* (поскольку сворачиваются одновременно), и записывают в виде *X**Y*.

3. *X* – последний символ (*окончание*) основы, а *Y* не входит в основу. В этом случае говорят, что символ *X* *предшествует* символу *Y* (поскольку символ *X* должен быть свернут раньше символа *Y*), и записывают в виде *Х**Y*.

Отношения , ,  называются *отношениями предшествования*. Следует заметить, что, хотя эти отношения похожи на арифметические отношения <, =, >, они имеют совершенно иные свойства. В частности, они не обладают свойствами коммутативности и ассоциативности. Из отношения *Х**Y*, не следует, что существует отношение *Y**X*. Эти отношения не являются симметричными, из отношения *X**Y* не следует *Y**X*. Для одной и той же грамматики может быть так, что *X**Y* и *Х**Y*, или для некоторых пар символов не выполняется ни одно из отношений предшествования.

Формально эти отношения для символов *X*, *Y* ∈ *VT* ∪ *VN* грамматики определяются следующим образом:

1. *X**Y*, если существует некоторая продукция *A* → α*XY*β, *A* ∈ *VN*, α, β ∈ (*VT* ∪ *VN*)\*. Это значит, что в правосторонней сентенциальной форме *X* и *Y* входят в одну и ту же основу.

2. *X**Y*, если существует некоторая продукция *A* → α*XB*β, *A*, *B* ∈ *VN*, α, β ∈ (*VT* ∪ *VN*)\*, такая, что , δ ∈ (*VT* ∪ *VN*)\*. Это значит, что в правосторонней сентенциальной форме основа начинается с символа *Y* (*Y* является заголовком основы).

3. *Х**Y*, если существует некоторая продукция *A* → α*BZ*β, *A*, *B* ∈ *VN*, *Z* ∈ *VT* ∪ *VN*, α, β ∈ (*VT* ∪ *VN*)\*, такая, что  и , γ, δ ∈ (*VT* ∪ *VN*)\*. Это значит, что в правосторонней сентенциальной форме основа завершается символом *Х* (*Х* является окончанием основы). Следует заметить, что в правосторонней сентенциальной форме справа от основы может быть только терминальная строка. Поэтому в данном случае символ *Y* может быть только терминалом, т. е. *Y* ∈ *VT*, и отношение  определяется на множестве (*VT* ∪ *VN*) × *VT*. Заметим также, что если *Y*δ выводится из *Z* за нуль шагов, то *Z* = *Y*.

Контекстно-свободная грамматика *G* = (*VT*, *VN*, *P*, *S*) называется *грамматикой простого предшествования*, если:

1) не содержит ε-продукций;

2) никакие две продукции грамматики не имеют совпадающих правых частей (грамматики, в которых нет двух продукций с одинаковыми правыми частями, называются *обратимыми*);

3) любые два символа, составляющие элемент множества (*VT* ∪ *VN*) × (*VT* ∪ *VN*), связаны одним и тем же отношением предшествования.

Отношения предшествования обычно записывают в виде *матрицы предшествования*, строки и столбцы которой соответствуют символам грамматики. На пересечении *i*-й строки и *j*-го столбца записывается отношение предшествования между соответствующими символами грамматики. Элементами матрицы являются знаки , ,  или «пусто». Последний случай означает, что соответствующие символы не могут стоять рядом ни в одной сентенциальной форме.

### Вычисление отношений предшествования

Формальный процесс вычисления отношений предшествования для символов *X*, *Y* ∈ *VT* ∪ *VN* заданной КС-грамматики можно представить такой последовательностью действий (в описании действий строки α, β ∈ (*VT* ∪ *VN*)\*, *A* ∈ *VN*):

1. Определить для каждого нетерминала *X* грамматики множество *L*(*X*) = {*Y* |}, т. е. множество символов грамматики (как терминалов, так и нетерминалов), с которых могут начинаться строки, выводимые из нетерминала *X*. Для этого необходимо построить отношение <LEFT>, определяемое следующим образом: *X* <LEFT> *Y*, если в грамматике существует продукция вида *X* → *Y*β. Затем вычислить отношение <LEFT>+ как транзитивное замыкание отношения <LEFT>. Тогда *L*(*X*) есть множество символов *Y*, для которых выполняется отношение *X* <LEFT>+ *Y*.

2. Вычислить отношение <LEFT>\* как рефлексивно-транзитивное замыкание отношения <LEFT>. Очевидно, что отношение <LEFT>\* легко вычисляется по отношению <LEFT>+, поскольку имеет место соотношение <LEFT>\* = <LEFT>+ ∪ *I*, где *I* – отношение тождественности. Отношение <LEFT>\* понадобится для вычисления отношения .

3. Определить для каждого нетерминала *X* грамматики множество *R*(*X*) = {*Y* |}, т. е. множество символов грамматики, являющихся крайними справа в строках, выводимых из нетерминала *X*. Для этого необходимо построить отношение <RIGHT>, определяемое следующим образом: *Y* <RIGHT> *X*, если в грамматике существует продукция вида *X* → β*Y*. Затем вычислить отношение <RIGHT>+ как транзитивное замыкание отношения <RIGHT>. Тогда *R*(*X*) есть множество символов *Y*, для которых выполняется отношение *Y* <RIGHT>+ *X*.

4. Построить для всех символов грамматики отношение  по его определению, т. е. *X**Y*, если в грамматике существует продукция вида *A* → α*XY*β.

5. Вычислить отношение . Из его формального определения следует, что *X**Y*, если в грамматике имеется продукция вида *A* → α*XB*β, где *B* ∈ *VN*, и *Y* ∈ *L*(*B*). Таким образом, отношение  можно вычислить как произведение отношений  и <LEFT>+, т. е. () = () (<LEFT>+).

6. Вычислить отношение . Из его формального определения следует, что *Х**Y* (напомним, что отношение определено только для *Y* ∈ *VT*), если существует продукция

а) вида *A* → α*BY*β, где *B* ∈ *VN*, *Y* ∈ *VT*, и *X* ∈ *R*(*B*);

б) вида *A* → α*BZ*β, где *B*, *Z* ∈ *VN*, и *X* ∈ *R*(*B*), *Y* ∈ *LT*(*Z*), где *LT*(*Z*) ⊆ *L*(*Z*) – подмножество терминалов множества *L*(*Z*).

Таким образом, () = (<RIGHT>+) () (<LEFT>\*). Поскольку *Y* может быть только терминалом, при вычислении произведения отношений следует рассматривать только те отношения *X* <LEFT>\* *Y*, где *Y* ∈ *VT*.

7. Построить матрицу предшествования, объединив матрицы отношений ,  и  в одну и заменив единицы на соответствующие обозначения (, , ) отношений.

Пример процесса вычисления отношений предшествования для грамматики с продукциями

*S* → *AB*

*A* → *aA* | *a*

*B* → *bB* | *b*

представлен на рис. 4.3.



Рис. 4.3. Процесс вычисления отношений предшествования

Из отношений <LEFT>+ и <RIGHT>+ следует, что

*L*(*S*) = {*A*, *a*}, *L*(*A*) = {*a*}, *L*(*B*) = {*b*},

*R*(*S*) = {*B*, *b*}, *R*(*A*) = {*A*, *a*}, *R*(*B*) = {*B*, *b*}.

Грамматика не является грамматикой простого предшествования, поскольку она не удовлетворяет третьему условию, требующему, чтобы любые два символа грамматики были связаны одним и тем же отношением предшествования. В нашем случае одновременно выполняются *A**b* и *A**b*. Действительно *A**b*, так как имеется продукция *S* → *AB* и *b* ∈ *L*(*B*), т. е. ,  
а *A**b*, так как существует продукция *S* → *AB*, такая, что *A* ∈ *R*(*A*) и *b* ∈ *L*(*B*), т. е.  и .

### Синтаксический анализ

Если грамматика является грамматикой простого предшествования, то отношения предшествования позволяют легко определить границы основы правосторонней сентенциальной формы. Для этого достаточно проанализировать сентенциальную форму слева направо и найти самую левую пару символов *Xj* и *Xj*+1, таких, что *Xj**Xj*+1, т. е. *Xj* – окончание основы, а отношение  отмечает ее правую границу. Затем сентенциальная форма просматривается справа налево, начиная с символа *Xj*, до тех пор, пока не будет найдена пара символов *Xi*–1 и *Xi*, таких, что *Xi*–1*Xi*, т. е. *Xi* – заголовок основы, а отношение  отмечает ее левую границу. Ясно, что между всеми соседними символами внутри основы выполняется отношение . В результате будет найдена основа, имеющая вид β = *Xi* … *Xj*, и должна существовать продукция вида *A* → β, т. е. основа может быть свернута к нетерминалу *A*. Например, если сентенциальная форма с отношениями предшествования имеет вид

*X*1*X*2*X*3*X*4*X*5,

то *X*2*X*3*X*4 является ее основой (в грамматике должна быть продукция вида *A* → *X*2*X*3*X*4), а в результате свертки получится сентенциальная форма *X*1*AX*5.

Возникает трудность, если заголовок основы является первым символом или окончание основы является последним символом сентенциальной формы. Для преодоления этой трудности вводится специальный символ ⊥, принадлежащий множеству терминалов, который используется для обозначения начала и конца сентенциальной формы. При этом предполагается, что  
⊥*X* и *X*⊥ для всех *X* ∈ *VT* ∪ *VN*, и результатом последовательности сверток входной строки ⊥*x*⊥, где *x* ∈ *L*(*G*), будет строка вида ⊥*S*⊥. Другой подход основан на том, что отношения предшествования вычисляются в предположении, что в грамматике имеется продукция вида *S*′ → ⊥*S*⊥. Тогда ⊥*X* для всех *X*, для которых , т. е. для всех *X* ∈ *L*(*S*), а *Y*⊥ для всех *Y*, для которых , т. е. для всех *Y* ∈ *R*(*S*). Это позволяет выявлять ошибки на более ранних этапах синтаксического анализа.

Как уже отмечалось, для реализации свертки удобным средством является стек. В начале анализа входной строки в вершину стека записывается символ ⊥. Затем осуществляется поиск окончания самой левой основы анализируемой строки. Для этого по матрице предшествования проверяются отношения между символом *X*, находящимся в вершине стека, и очередным входным символом *Y*. Если условие *Х**Y* не выполняется, то очередной входной символ заносится в стек и процесс продолжается до тех пор, пока не будет выполнено условие *Х**Y*, т. е. не будет найдено окончание самой левой основы. Таким образом, если между символом в вершине стека и очередным входным символом выполняется отношение  или , то синтаксический анализатор выполняет операцию переноса входного символа в стек (окончание основы еще не найдено). После обнаружения окончания основы (выполняется отношение ) в верхней части стека будет содержаться основа сентенциальной формы.

Затем, чтобы выполнить свертку, производится поиск заголовка основы. Для этого из стека исключаются все символы, имеющие равное предшествование  (начиная с вершины стека), до тех пор, пока между символом в вершине стека и последним исключенным из стека символом не выполнится отношение , т. е. последний исключенный из стека символ будет являться заголовком основы. Завершается свертка поиском продукции с соответствующей правой частью и записью в стек нетерминала из левой части этой продукции. Если такая продукция не найдена (т. е. подстрока не является основой), то фиксируется синтаксическая ошибка.

Разбор входной строки успешно завершен, если в результате работы стек будет содержать строку ⊥*S* (содержимое стека представляется строкой, в которой самый правый символ находится в вершине стека), а входная строка будет содержать символ ⊥. Если основа не будет найдена или между парами соседних символов не выполняется ни одно из отношений предшествования, то фиксируется синтаксическая ошибка.

Детали реализации алгоритма синтаксического анализа предлагаются в качестве упражнения.

В качестве примера рассмотрим грамматику простого предшествования с продукциями

*S* → *DB*

*D* → *A*

*A* → *aA* | *a*

*B* → *bB* | *b*

матрица предшествования для которой приведена на рис. 4.4. Отношения предшествования вычислены в предположении, что в грамматике имеется продукция *S*′ → ⊥*S*⊥.



Рис. 4.4. Матрица предшествования

Рассмотрим работу анализатора на примере разбора строки *aabb*⊥, которая выводится в соответствии со следующей правосторонней схемой (символ ⊥ начала строки опущен)

*S*⊥ ⇒ *DB*⊥ ⇒ *DbB*⊥ ⇒ *Dbb*⊥ ⇒ *Abb*⊥ ⇒ *aAbb*⊥ ⇒ *aabb*⊥.

Процесс разбора показан на рис. 4.5. Содержимое стека представлено строкой, в которой для наглядности между каждой парой соседних символов указано отношение предшествования (реально стек содержит только символы грамматики).

|  |  |  |  |
| --- | --- | --- | --- |
| Входной буфер | Содержимое стека | Основа | Выполняемое действие |
| *aabb*⊥ | ⊥ |  | Перенос *a* в стек, т. к. ⊥*a* |
| *abb*⊥ | ⊥*a* |  | Перенос *a* в стек, т. к. *a**a* |
| *bb*⊥ | ⊥*a**a* | *a* | Свертка для *A* → *a*, т. к. *a**b* |
| *bb*⊥ | ⊥*a**A* | *aA* | Свертка для *A* → *aA*, т. к. *A**b* |
| *bb*⊥ | ⊥*A* | *A* | Свертка для *D* → *A*, т. к. *A**b* |
| *bb*⊥ | ⊥*D* |  | Перенос *b* в стек, т. к. *D**b* |
| *b*⊥ | ⊥*D**b* |  | Перенос *b* в стек, т. к. *b**b* |
| ⊥ | ⊥*D**b**b* | *b* | Свертка для *B* → *b*, т. к. *b*⊥ |
| ⊥ | ⊥*D**b**B* | *bB* | Свертка для *B* → *bB*, т. к. *B*⊥ |
| ⊥ | ⊥*D**B* | *DB* | Свертка для *S* → *DB*, т. к. *B*⊥ |
| ⊥ | ⊥*S* |  | Разбор успешно завершен |

Рис. 4.5. Процесс разбора строки *aabb*⊥

### Функции предшествования

Если грамматика состоит из *n* символов, то матрица предшествования имеет размер *n* × *n*. Причем многие элементы будут иметь значение «пусто». Можно применить известные методы компактного хранения разреженных массивов, что обычно приводит к дополнительным временным затратам. Поэтому чаще вместо матриц предшествования синтаксические анализаторы используют так называемые *функции предшествования* *f* и *g*, отображающие символы грамматики в целые числа. При этом должны выполняться следующие соотношения:

*f*(*X*) = *g*(*Y*), если *X**Y*,

*f*(*X*) < *g*(*Y*), если *X**Y*,

*f*(*X*) > *g*(*Y*), если *Х**Y*.

Однако не всякая матрица предшествования может иметь функции предшествования, хотя обычно на практике они существуют. Тогда вместо *n*2 элементов достаточно хранить только 2*n* элементов.

Процедура построения функций предшествования по заданной матрице предшествования заключается в следующем.

1. Создать символы *fX* и *gX* для каждого символа *X* грамматики, включая и символ ⊥.

2. Сгруппировать созданные символы на как можно большее число групп таким образом, что если *X**Y*, то *fX* и *gY* должны входить в одну группу. Следует заметить, что в одну группу могут попасть символы, не связанные отношением . Например, если *a**b* и *c**b*, то *fa* и *fc* должны находиться в одной и той же группе, поскольку оба находятся в той же группе, что и *gb*. Если, кроме того, *c**d*, то *fa* и *gd* находятся в одной группе, даже если не выполняется условие *a**d*.

3. Создать ориентированный граф, вершины которого представляют собой определенные в п. 2 группы. Для всех *X* и *Y*, если *X**Y*, проводится дуга из группы, в которой находится *gY*, в группу с *fX*. Если *Х**Y*, дуга проводится из группы с *fX* в группу с *gY*. Таким образом, дуга (путь) от группы с *fX* к группе с *gY* означает, что *f*(*X*) превосходит *g*(*Y*); путь от группы с *gY* к группе с *fX* означает, что *g*(*Y*) должно превосходить *f*(*X*).

4. Если построенный граф имеет циклы, то для заданной таблицы предшествования не существуют функции предшествования. Если циклов нет, то значением *f*(*X*) является длина самого длинного пути, начинающегося в группе с *fX*; значение *g*(*X*) равно длине самого длинного пути из группы с *gX*.

В качестве примера определим функции предшествования для матрицы предшествования на рис. 4.4. Построенный граф показан на рис. 4.6. Поскольку *a**A*, элементы *fa* и *gA* объединены в одну группу. Элементы *fb*, *fD* и *gB* объединены в одну группу, т. к. *b**B* и *D**B*, несмотря на то, что символы *b* и *D* не связаны никаким отношением предшествования.

Рис. 4.6. Граф отношений предшествования

Этот граф не имеет циклов, следовательно, функции предшествования существуют. Определим самые длинные пути от каждой вершины графа:

группы {*fS*}, {*g*⊥}, {*f*⊥} и {*gS*} не имеют исходящих дуг, т. е. длины путей равны 0,

{*fA*} → {*gb*} → {*fb*, *fD*, *gB*} → {*g*⊥} длины 3,

{*fB*} → {*g*⊥} длины 1,

{*fb*, *fD*, *gB*} → {*g*⊥} длины 1,

{*fa*, *gA* } → {*gb*} → {*fb*, *fD*, *gB*} → {*g*⊥} длины 3,

{*gD*} → {*f*⊥} длины 1,

{*ga*} → {*fa*, *gA* } → {*gb*} → {*fb*, *fD*, *gB*} → {*g*⊥} длины 4,

{*gb*} → {*fb*, *fD*, *gB*} → {*g*⊥} длины 2.

Полученные длины путей и будут являться значениями соответствующих функций предшествования. Например, поскольку группы {*fS*}, {*g*⊥}, {*f*⊥} и {*gS*} не имеют исходящих дуг (длины путей равны 0), *f*(*S*) = *g*(⊥) = *f*(⊥) = *g*(*S*) = 0. Самый длинный путь из {*fa*, *gA* } имеет длину 3, поэтому *f*(*a*) = *g*(*A*) = 3. Аналогично определяются остальные значения функций. Результат построения функций предшествования выглядит следующим образом:



Недостатком использования функций предшествования является то, что теряется информация о несуществующих отношениях, которые могут быть использованы для обнаружения синтаксических ошибок. На практике потеря возможности обнаружения ошибок не считается достаточно серьезной, поскольку в большинстве случаев они могут быть обнаружены при попытке выполнения свертки для необнаруженной основы, но на более поздних этапах разбора.

В качестве примера обнаружения ошибки рассмотрим разбор строки *aba*⊥, не принадлежащей языку, порождаемому грамматикой. При использовании матрицы предшествования (рис. 4.7) ошибка фиксируется на более раннем этапе разбора по несуществующему отношению для символов *b* и *a*. При использовании функций предшествования (рис. 4.8) ошибка обнаруживается на более позднем этапе при попытке выполнения операции свертки подстроки *bD*, поскольку в грамматике нет продукции с данной правой частью, т. е. основа не обнаружена.

|  |  |  |  |
| --- | --- | --- | --- |
| Входной буфер | Содержимое стека | Основа | Выполняемое действие |
| *aba*⊥ | ⊥ |  | Перенос *a* в стек, т. к. ⊥*a* |
| *ba*⊥ | ⊥*a* | *a* | Свертка для *A* → *a*, т. к. *a**b* |
| *ba*⊥ | ⊥*A* | *A* | Свертка для *D* → *A*, т. к. *A**b* |
| *ba*⊥ | ⊥*D* |  | Перенос *b* в стек, т. к. *D**b* |
| *a*⊥ | ⊥*D**b* |  | Синтаксическая ошибка, т. к. для символов *b* и *a* не существует отношение предшествования |

Рис. 4.7. Процесс разбора строки *aba*⊥ по матрице предшествования

|  |  |  |  |
| --- | --- | --- | --- |
| Входной буфер | Содержимое стека | Основа | Выполняемое действие |
| *aba*⊥ | ⊥ |  | Перенос *a* в стек, т. к. *f*(⊥) < *g*(*a*) |
| *ba*⊥ | ⊥ < *a* | *a* | Свертка для *A* → *a*, т. к. *f*(*a*) > *g*(*b*) |
| *ba*⊥ | ⊥ < *A* | *A* | Свертка для *D* → *A*, т. к. *f*(*A*) > *g*(*b*) |
| *ba*⊥ | ⊥ < *D* |  | Перенос *b* в стек, т. к. *f*(*D*) < *g*(*b*) |
| *a*⊥ | ⊥ < *D* < *b* |  | Перенос *a* в стек, т. к. *f*(*b*) < *g*(*a*) |
| ⊥ | ⊥ < *D* < *b* < *a* | *a* | Свертка для *A* → *a*, т. к. *f*(*a*) > *g*(⊥) |
| ⊥ | ⊥ < *D* < *b* < *A* | *A* | Свертка для *D* → *A*, т. к. *f*(*A*) > *g*(⊥) |
| ⊥ | ⊥ < *D* < *b* = *D* | *bD* | Попытка свертки подстроки *bD*, т. к. *f*(*D*) > *g*(⊥). Синтаксическая ошибка, поскольку нет продукции с правой частью *bD* |

Рис. 4.8. Процесс разбора строки *aba*⊥ по функциям предшествования

## Грамматики слабого предшествования

Грамматики слабого предшествования являются небольшим расширением класса грамматик простого предшествования, связанное с тем, что разрешено пересечение отношений  и . Таким образом, в отличие от грамматик простого предшествования, где между парами символов грамматики допускается не более одного отношения предшествования, в грамматиках слабого предшествования между парами символов могут быть одновременно отношения  и .

Для грамматик слабого предшествования отношение  по-прежнему используется для определения окончания основы. Возникают трудности с определением заголовка основы, связанные с тем, что правая часть одной продукции может быть суффиксом правой части другой продукции.

Пусть αβγ*w*, α, β, γ ∈ (*VT* ∪ *VN*)\*, *w* ∈ *VT*\* – правосторонняя сентенциальная форма, в которой окончанием основы является последний символ строки γ. Если в грамматике есть продукции *A* → γ и *A* → βγ, то возникает вопрос, какую из этих продукций необходимо выбрать для свертки (что будет основой, γ или βγ?). В этом случае между всеми символами строк β и γ выполняется отношение , а между последним символом строки β и первым символом строки γ выполняются отношения  и .

В грамматиках слабого предшествования в случае такого конфликта в качестве основы для свертки выбирается наиболее длинная основа, т. е. βγ.

Формально КС-грамматика *G* = (*VT*, *VN*, *P*, *S*) называется *грамматикой слабого предшествования*, если:

1) не содержит ε-продукций;

2) никакие две продукции грамматики не имеют совпадающих правых частей;

3) отношение  не пересекается с объединением отношений  и ;

4) для продукций *A* → α*X*β и *B* → β, где α, β ∈ (*VT* ∪ *VN*)\*, *X* ∈ *VT* ∪ *VN*, не выполняется ни отношение *X**B*, ни отношение *X**B*.

Для поиска окончания основы (как и для грамматик простого предшествования) достаточно проанализировать сентенциальную форму слева направо и найти самую левую пару символов *Xj* и *Xj*+1, таких, что *Xj**Xj*+1, т. е. *Xj* – окончание основы. Затем сентенциальная форма просматривается справа налево, начиная с символа *Xj*, до тех пор, пока не будет найдена пара символов *Xi*–1 и *Xi*, таких, что *Xi*–1*Xi* и не выполняется отношение *Xi*–1*Xi*, т. е. *Xi* – заголовок основы. Ясно, что между всеми соседними символами внутри основы выполняется либо отношение , либо отношения  и . Таким образом, в процессе поиска заголовка основы при просмотре сентенциальной формы справа налево в случае, если между парой соседних символов выполняется как отношение , так и отношение , то отношение  имеет приоритет над отношением .

Детали реализации алгоритма синтаксического анализа предлагаются в качестве упражнения.

## Грамматики операторного предшествования

Грамматики операторного предшествования представляют достаточно широкий класс КС-грамматик.

*Операторной грамматикой* называется ε-свободная (не содержит ε-продукций) КС-грамматика, в которой правые части всех продукций не содержат смежных нетерминалов. В таких грамматиках терминалы можно рассматривать как операции, а нетерминалы – как операнды. Например, в арифметических выражениях можно сказать, что операция умножения *предшествует* операции сложения, поскольку умножение имеет более высокий приоритет, чем сложение. Порядок вычисления значения арифметического выражения определяется только порядком выполнения операций и не зависит от операндов. Поэтому понятие предшествования можно определить только для операций (терминалов).

Пусть *G* = (*VT*, *VN*, *P*, *S*) – операторная КС-грамматика, пополненная продукцией *S*′ → ⊥*S*⊥, где ⊥ ∈ *VT*. *Отношения операторного предшествования* задаются на множестве *VT* × *VT* следующим образом:

1. *a**b*, если существует некоторая продукция *A* → α*aCb*β, *A* ∈ *VN*, α, β ∈ (*VT* ∪ *VN*)\*, *C* ∈ *VN* ∪ {ε}.

2. *a**b*, если существует некоторая продукция *A* → α*aB*β, *A*, *B* ∈ *VN*, α, β ∈ (*VT* ∪ *VN*)\*, такая, что , *C* ∈ *VN* ∪ {ε}, δ ∈ (*VT* ∪ *VN*)\*.

3. ⊥*a*, если , *C* ∈ *VN* ∪ {ε}, α ∈ (*VT* ∪ *VN*)\*.

4. *a**b*, если существует некоторая продукция *A* → α*Bb*β, *A*, *B* ∈ *VN*, α, β ∈ (*VT* ∪ *VN*)\*, такая, что , *C* ∈ *VN* ∪ {ε}, δ ∈ (*VT* ∪ *VN*)\*.

5. *a*⊥, если , *C* ∈ *VN* ∪ {ε}, α ∈ (*VT* ∪ *VN*)\*.

Отношения операторного предшествования можно задавать с помощью *матрицы операторного предшествования*. Строки и столбцы матрицы соответствуют символам из *VT*. Пустой элемент матрицы соответствует синтаксической ошибке, например, если для пары *a* и *b* элемент пустой, то ни в одной правильной входной строке *b* не может следовать непосредственно за *a*.

Операторная КС-грамматика *G* = (*VT*, *VN*, *P*, *S*) называется *грамматикой операторного предшествования*, если:

1) никакие две продукции грамматики не имеют совпадающих правых частей;

2) между любыми двумя терминалами из множества *VT* × *VT* выполняется не более одного отношения операторного предшествования.

Отношения операторного предшествования вычисляются так же, как и для грамматик простого предшествования, определив для каждого нетерминала *X* множества *L*(*X*) и *R*(*X*). Отличие заключается в том, что в эти множества включаются только терминалы, игнорируя нетерминалы (подставляя вместо нетерминалов пустую строку ε).

Построим матрицу операторного предшествования для грамматики (в предположении, что имеется продукция *S* → ⊥*E*⊥)

*E* → *E* + *T*⎪*T*

*T* → *T* × *F*⎪*F*

*F* → (*E*)⎪*i*

Определим множества *L*(*X*) и *R*(*X*) для *X* ∈ {*E*, *T*, *F*}:

*L*(*E*) = {*E*, *T*, *F*, (, *i*} = {+, ×, (, *i*};

*L*(*T*) = {*T*, *F*, (, *i*} = {×, (, *i*};

*L*(*F*) = {(, *i*};

*R*(*E*) = {*T*, *F*, ), *i*} = {+, ×, ), *i*};

*R*(*T*) = {*F*, ), *i*} = {×, ), *i*};

*R*(*F*) = {), *i*};

Соответствующая матрица операторного предшествования приведена на рис. 4.9.



Рис. 4.9. Матрица операторного предшествования

При распознавании основы возникает проблема с нетерминалами, поскольку для них не определены отношения операторного предшествования. Чтобы решить эту проблему исходная грамматика преобразуется в так называемую остовную грамматику путем замены всех нетерминалов одним начальным нетерминалом и устранением всех цепных продукций.

Пусть *G* = (*VT*, *VN*, *P*, *S*) – операторная грамматика. *Остовной грамматикой* для грамматики *G* называется грамматика *GS* = (*VT*, {*S*}, *P'*, *S*), множество продукций *P'* которой строится следующим образом:

а) если множество *P* грамматики *G* содержит продукцию вида *A* → *Y*1*Y*2...*Yn*, где *Yi* ∈ *VT* ∪ *VN*, 1 ≤ *i* ≤ *n*, то в *P'* включается продукция *S* → *X*1*X*2...*Xn*, где *Xi* = *Yi*, если *Yi* ∈ *VT*, или *Xi* = *S*, если *Yi* ∈ *VN*;

б) множество продукций *P'* не должно содержать продукций вида *S* → *S*.

Например, для рассмотренной выше грамматики арифметических выражений остовной будет грамматика с продукциями

*E* → *E* + *E*⎪*E* × *E*⎪(*E*)⎪*i*

Следует заметить, что язык *L*(*G*) ⊆ *L*(*GS*) и грамматика *GS* может порождать строки, не принадлежащие *L*(*G*). Кроме того, грамматика *GS* может быть неоднозначной. Однако отношения операторного предшествования гарантируют единственность синтаксического разбора и его правильность.

Поиск основы реализуется также как и для грамматик простого предшествования. Отличие заключается в том, что если в вершине стека оказывается нетерминал, то для определения отношения предшествования рассматривается ближайший к вершине стека терминал (нетерминал игнорируется). Детали реализации алгоритма синтаксического анализа предлагаются в качестве упражнения.

Рассмотрим работу анализатора на примере разбора строки *i* × (*i* + *i*)⊥, которая выводится в соответствии со следующей правосторонней схемой (символ ⊥ начала строки опущен)

*E*⊥ ⇒ *E* × *E*⊥ ⇒ *E* × (*E*)⊥ ⇒ *E* × (*E* + *E*)⊥ ⇒ *E* × (*E* + *i*)⊥ ⇒

⇒ *E* × (*i* + *i*)⊥ ⇒ *i* × (*i* + *i*)⊥.

Процесс разбора показан на рис. 4.10.

|  |  |  |  |
| --- | --- | --- | --- |
| Входной буфер | Содержимое стека | Основа | Выполняемое действие |
| *i*×(*i*+*i*)⊥ | ⊥ |  | Перенос *i* в стек, т. к. ⊥*i* |
| ×(*i*+*i*)⊥ | ⊥*i* | *i* | Свертка для *E* → *i*, т. к. *i*× |
| ×(*i*+*i*)⊥ | ⊥*E* |  | Перенос × в стек, т. к. ⊥× |
| (*i*+*i*)⊥ | ⊥*E*× |  | Перенос ( в стек, т. к. ×( |
| *i*+*i*)⊥ | ⊥*E*×( |  | Перенос *i* в стек, т. к. (*i* |
| +*i*)⊥ | ⊥*E*×(*i* | *i* | Свертка для *E* → *i*, т. к. *i*+ |
| +*i*)⊥ | ⊥*E*×(*E* |  | Перенос + в стек, т. к. (+ |
| *i*)⊥ | ⊥*E*×(*E*+ |  | Перенос *i* в стек, т. к. +*i* |
| )⊥ | ⊥*E*×(*E*+*i* | *i* | Свертка для *E* → *i*, т. к. *i*) |
| )⊥ | ⊥*E*×(*E*+*E* | *E*+*E* | Свертка для *E* → *E*+*E*, т. к. +) |
| )⊥ | ⊥*E*×(*E* |  | Перенос ) в стек, т. к. () |
| ⊥ | ⊥*E*×(*E*) | (*E*) | Свертка для *E* → (*E*), т. к. )⊥ |
| ⊥ | ⊥*E*×*E* | *E*×*E* | Свертка для *E* → *E*×*E*, т. к. ×⊥ |
| ⊥ | ⊥*E* |  | Разбор успешно завершен |

Рис. 4.10. Процесс разбора строки *i* × (*i* + *i*)⊥

## *LR*(*k*)-грамматики

Наиболее общим случаем восходящего синтаксического анализа является *LR*-разбор, который применим к широкому классу *LR*(*k*)-грамматик.

*LR*(*k*)-*грамматикой* называется грамматика, при использовании которой всякая основа сентенциальной формы однозначно определяется (т. е. все конфликты типа «перенос/свертка» и «свертка/свертка» можно разрешать) на основании уже прочитанного текста (левый контекст) и фиксированного числа предварительно просматриваемых символов (максимум *k*). Аббревиатура *LR* означает «левосторонний ввод – правосторонний вывод», т. е. входная строка анализируется слева направо и используется правосторонняя схема вывода.

При *LR*-разборе всегда будем рассматривать пополненные грамматики, т. е. в КС-грамматике всегда предполагается наличие продукции вида *S'* → *S*⊥, где *S'* и *S* – начальные нетерминалы соответственно пополненной и исходной грамматик, терминал ⊥ – маркер конца ввода. При этом нетерминал *S'* не должен встречаться в правых частях других продукций.

Практический интерес представляют случаи *k* = 0 и *k* = 1. Рассмотрение даже двух предварительно просматриваемых символов делает *LR*-метод разбора довольно громоздким. Кроме того, из теории формальных языков и грамматик известно, что любой *LR*(*k*)-язык является также *LR*(1)-языком, т. е. может генерироваться *LR*(1)-грамматикой. Это означает, что для любой *LR*(*k*)-грамматики можно построить эквивалентную *LR*(1)-грамматику. Если *LR*(*k*)-язык удовлетворяет *условию собственности префиксов*, то он может генерироваться *LR*(0)-грамматикой. Это условие означает, что если *x* – строка языка, то никакой ее собственный префикс не принадлежит этому же языку. Поэтому, если используется маркер конца ввода ⊥, язык удовлетворяет условию собственности префиксов и, следовательно, может генерироваться *LR*(0)-грамматикой.

Таким образом, в отличие от *LL*(*k*)-грамматик, где увеличение значения *k* позволяет представлять больший класс языков, в *LR*(*k*)-грамматиках этого не происходит. На практике обычно используются *LR*(1)-грамматики (или ее более простые подклассы), поскольку многие типичные свойства грамматик языков программирования относятся к *LR*(1)-свойствам, а не к *LR*(0)-свойствам, т. е. использование *LR*(1)-грамматик вместо *LR*(0)-грамматик позволяет избежать затруднений, связанных с преобразованием грамматик.

Основными достоинствами *LR*-разбора являются:

1) метод применим ко всем языкам, которые можно разбирать детерминированно, т. е. имеет универсальный характер и охватывает широкий класс языков и грамматик;

2) относительно редко возникает необходимость в преобразованиях грамматик;

3) имеются хорошие диагностические характеристики и возможность исправления ошибок.

Для разрешения конфликтов типа «перенос/свертка» и «свертка/свертка» *LR*-анализатор должен отслеживать, где именно он находится в процессе синтаксического анализа. Чтобы ссылаться на конкретную позицию в продукции грамматики, вводится понятие «пункт» (в литературе встречаются также термины «ситуация», «конфигурация»).

В общем случае *LR*(*k*)-*пункт* представляет собой пару [*A* → α•β, *w*], где *A* ∈ *VN*, α, β ∈ (*VT* ∪ *VN*)\*, *w* ∈ *VT*\*. Первый элемент пары – это продукция грамматики, в правой части которой перед одним из символов стоит маркерная точка, а второй элемент *w* – терминальная строка, которая может следовать за продукцией (т. е. за нетерминалом *A*). Строка *w* (ее длина не должна превышать *k* символов) называется *предпросмотром* или *контекстом* пункта. Очевидно, что в *LR*(1)-пункте [*A* → α•β, *a*] предпросмотр представляет собой единственный терминальный символ *a* ∈ *VT*, а в *LR*(0) пункте [*A* → α•β] предпросмотр отсутствует.

Поскольку на практике рассматривают случаи *k* = 0 и *k* = 1, остановимся подробнее на *LR*(1)-пункте [*A* → α•β, *a*]. Первый элемент *LR*(1)-пункта указывает, какая часть продукции уже исследована в данной точке синтаксическим анализатором. Например, продукция *A* → *BCD* дает четыре варианта расположения маркерной точки:

*A* → •*BCD*

*A* → *B*•*CD*

*A* → *BC*•*D*

*A* → *BCD*•

Первый вариант определяет, что во входном потоке ожидается строка, порождаемая *BCD*. Второй вариант указывает, что порожденная *B* строка уже исследована, а во входном потоке ожидается строка, порождаемая *CD*. Следует заметить, что продукция вида *A* → ε дает только один варианта расположения маркерной точки: *A* → •.

Что касается предпросмотра пункта, то он не играет никакой роли в *LR*(1)-пункте вида [*A* → α•β, *a*], где β ≠ ε, а важен для *LR*(1)-пункта вида [*A* → α•, *a*] (этот пункт соответствует окончанию продукции *A* → α), когда выполняется свертка для продукции *A* → α только тогда, когда очередным входным символом является терминал *a*. Другими словами, в *LR*(1)-пункте предпросмотр – это символ-следователь, который при разборе строки языка может оказаться предварительно просматриваемым символом в тех случаях, когда выполняется свертка в соответствии с порождающей продукцией.

Основой построения *LR*-анализатора является детерминированный конечный автомат, который используется для принятия решений в процессе синтаксического анализа. Такой автомат будем называть *LR*-*автоматом*. Чтобы найти все состояния *LR*-автомата, все возможные для заданной грамматики пункты по определенным правилам группируются в множества, соответствующие состояниям автомата. Пункты, которые неразличимы для анализатора, объединяются в одно состояние. Правила объединения пунктов зависят от используемого подкласса *LR*(1)-грамматик и будут изложены при рассмотрении соответствующих подклассов.

## *LR*-таблицы разбора

Возможны различные виды таблиц разбора, которые являются, по сути, различными формами представления магазинного автомата, принимающего данный контекстно-свободный язык. В качестве примера рассмотрим достаточно простой и понятный вид *LR*-таблицы разбора для *LR*-грамматик [18].

*LR*-анализатор использует два стека (стек символов и стек состояний) и специальную *LR*-таблицу разбора. При практической реализации стек символов не требуется, но мы будем его рассматривать для облегчения понимания принципов работы синтаксического анализатора.

*LR*-таблица разбора представляет собой прямоугольную матрицу, состоящую из столбцов для каждого символа грамматики (терминалов и нетерминалов), включая и маркер конца ввода, и строк, соответствующих каждому состоянию, в котором может находиться анализатор. Состояние обобщает информацию, содержащуюся в стеке ниже него. Комбинация состояния в вершине стека и текущего входного символа используется в качестве индекса таблицы разбора и по ее элементу определяется дальнейшее действие. Таблица разбора включает элементы четырех типов:

1. *Элементы переноса*. Обычно записываются в виде *Si*, что означает: поместить в стек символов входной символ, поместить в стек состояний состояние *i* и перейти в состояние *i*. Если входной символ является терминалом, принять его (перейти к следующему символу разбираемой входной строки).

2. *Элементы свертки*. Обычно записываются в виде *Rj*, что означает: выполнить свертку для продукции с номером *j*, т. е. допустив, что *n* есть число символов в правой части продукции с номером *j*, удалить *n* элементов из стека символов и *n* элементов из стека состояний и перейти к состоянию, находящемуся в вершине стека состояний. Нетерминал в левой части продукции с номером *j* считать входным символом на следующем шаге разбора.

3. *Элементы ошибок*. Это пустые ячейки в таблице разбора и соответствуют синтаксическим ошибкам.

4. *Элементы остановки* (*stop*), которые указывают на успешное завершение разбора входной строки.

Следует заметить, что все *LR*-анализаторы (при *k* ≤ 1) работают одинаково (независимо от того, построен анализатор для *LR*(1)-грамматики или какого-либо ее подкласса), разница между ними заключается только в построении таблиц разбора.

### *LR*(0)-грамматики

Рассмотрим самый легкий в реализации, но наименее мощный *LR*(0)-метод построения *LR*-таблицы разбора, основанный на применении *LR*(0)-грамматик. Для таких грамматик основа распознается без исследования предварительно просматриваемого символа, а пункт не содержит предпросмотра, т. е. *LR*(0)-пункт [*A* → α•β] представляет собой только продукцию с маркерной точкой.

Процесс построения *LR*(0)-автомата начинается с исходного пункта [*S*′ → •*S*⊥], где *S*′ – начальный символ пополненной грамматики, которая включается в исходное состояние автомата. Затем выполняется *операция замыкания*, которая заключается в следующем. Если пункт вида [*A* → α•*B*β], *B* ∈ *VN*, входит в некоторое состояние, то все пункты вида [*B* → •γ], γ ∈ (*VT* ∪ *VN*)\*, включаются в это же состояние (если их еще там нет). Этот процесс продолжается рекурсивно до тех пор, пока в состояние не будут включены все возможные пункты. Таким образом, если в пункте маркерная точка стоит перед нетерминалом *B*, то в текущее состояние включаются все пункты, в которых маркерная точка стоит перед первым символом правой части продукции с нетерминалом *B* в левой части. Если в каком-то из добавленных пунктов маркерная точка стоит перед нетерминалом, процесс повторяется. Такое выполнение операции замыкания объясняется следующим образом. Наличие [*A* → α•*B*β] в некотором состоянии указывает, что в данный момент в процессе синтаксического анализа предполагается, что во входном потоке может встретиться подстрока, выводимая из *B*β. Но если имеется продукция *B* → γ, то, естественно, в этот момент может встретиться и строка, выводимая из γ. Поэтому [*B* → •γ] должна быть включена в это же состояние (пункты [*A* → α•*B*β] и [*B* → •γ] являются неразличимыми).

Из заданного состояния *I*, содержащего хотя бы один пункт, не соответствующий окончанию продукции, строится переход в другое состояние *I*′ по символу грамматики, перед которым стоит маркерная точка. При этом точка перемещается на одну позицию вправо, т. е. будет стоять после этого символа. Состояние *I*′ называется *преемником* состояния *I*. Новый пункт, который получается при операции образования преемника, называется *базовым* для состояния *I*′. Таким образом, если пункт [*A* → α•*X*β], *X* ∈ *VT* ∪ *VN*, принадлежит состоянию *I*, то базовым пунктом для состояния *I*′ будет пункт [*A* → α*X*•β] (переход по символу *X*). Затем для базового пункта выполняется операция замыкания, чтобы включить в новое состояние все остальные неразличимые пункты. Следует заметить, что если состояние *I* содержит несколько пунктов, в которых точка стоит перед одним и тем же символом *X*, то в состоянии *I*′ будет множество базовых пунктов, в которых точка будет стоять после *X*. Это связано с тем, что *LR*(0)-автомат должен быть детерминированным, а в таких автоматах не допускается, чтобы из одного состояния существовал переход по одному и тому же символу в разные состояния.

Процесс последовательного выполнения операций замыкания и образования преемника продолжается до тех пор, пока все возможные пункты грамматики не окажутся включенными в какие-либо состояния. Следует иметь в виду, что автомат не должен иметь несколько состояний с одинаковыми множествами пунктов.

Рассмотрим процесс построения *LR*(0)-автомата для грамматики со следующими продукциями (перед продукцией указан ее порядковый номер):

1) *S* → *SbA*

2) *S* → *A*

3) *A* → *a*

Поскольку мы рассматриваем пополненные грамматики, то имеется еще продукция *S*′ → *S*⊥. *LR*(0)-автомат для данной грамматики представлен на рис. 4.11. Для удобства записи в пунктах квадратные скобки опущены.

*b*

*a*

*a*

*A*

**⊥**

*R*2

*S*

*I*5

*I*3

*I*2

*I*4

*I*1

*stop*

*I*0

*S*′ → •*S*⊥

*S* → •*SbA*

*S* → •*A*

*A* → •*a*

*S*′ → *S*•⊥

*S* → *S*•*bA*

*A* → *a*•

*S* → *Sb*•*A*

*A* → •*a*

*S* → *SbA*•

*S* → *A*•

*S*′ → *S*⊥•

*R*3

*R*1

*A*

Рис. 4.11. *LR*(0)-автомат

Образуем исходное состояние *I*0. Базовым пунктом является начальный пункт [*S*′ → •*S*⊥]. Выполним операцию замыкания. Поскольку маркерная точка стоит перед нетерминалом *S*, в состояние *I*0 необходимо включить пункты [*S* → •*SbA*] и [*S* → •*A*] для продукций с нетерминалом *S* в левой части. Среди добавленных пунктов появился нетерминал *A*, перед которым стоит точка, следовательно, необходимо включить в *I*0 пункт [*A* → •*a*]. Операция замыкания выполнена и определены все пункты для состояния *I*0.

В состоянии *I*0 перед тремя разными символами стоит маркерная точка, т. е. будет три состояния-преемника: *I*1 по сим­волу *S* (базовые пункты [*S*′ → *S*•⊥] и [*S* → *S*•*bA*]), *I*2 по символу *A* (базовый пункт [*S* → *A*•]) и *I*3 по символу *a* (базовый пункт [*A* → *a*•]). Следует заметить, что состояния *I*1, *I*2 и *I*3 исчерпываются базовыми пунктами, так как в них нет пунктов с маркерными точками перед нетерминалами. Пункты в состояниях *I*2 и *I*3 соответствуют окончаниям продукций, поэтому они не имеют преемников. Это означает, что в этих состояниях должна выполняться свертка для соответствующих продукций. На рис. 4.11 действия свертки обозначены так же, как и элементы свертки в таблице разбора. Например, в состоянии *I*2 выполняется свертка для продукции *S* → *A* с номером два (обозначено *R*2).

У состояния *I*1 имеются два состояния-преемника: *stop* по символу ⊥ (базовый пункт [*S*′ → *S*⊥•]) и *I*4 по символу *b* (базовый пункт [*S* → *Sb*•*A*]). Состояние *stop* означает, что процесс разбора завершен. Операция замыкания базового пункта в состоянии *I*4 добавляет в него пункт [*A* → •*a*]. Преемниками состояния *I*4 являются состояния *I*3 по символу *a* и *I*5 по символу *A* (базовый пункт [*S* → *SbA*•]). В состоянии *I*5 выполняется свертка *R*1 для продукции *S* → *SbA*.

На этом процесс построения *LR*(0)-автомата завершен, поскольку все возможные пункты заданной грамматики включены в соответствующие состояния. Следует заметить, что один пункт может входить в более чем одно состояние. Например, пункт [*S* → •*a*] содержится в состояниях *I*0 и *I*4.

В полученном автомате никаких конфликтов нет – это особенность *LR*(0)-грамматик. Конфликт «свертка/свертка» возникает, если в одном состоянии возможно несколько сверток. Конфликт «перенос/свертка» возникает, если в состоянии есть свертка и возможен переход в другое состояние.

*LR*(0)-автомат удобно представлять в табличной форме. Пример такого представления полученного *LR*(0)-автомата показан на рис. 4.12. Очевидно, что для состояния *stop* нет смысла выделять специальную строку в таблице, поскольку это состояние не предполагает никаких действий, а только сигнализирует об успешном завершении разбора.

По *LR*(0)-автомату легко строится таблица разбора. Сначала заносятся элементы переноса. Переход из одного состояния в другое соответствует действию переноса. Поэтому элементы переноса вносятся в таблицу разбора на основании информации о состояниях и переходах автомата. Если из состояния *Ii* существует переход в состояние *Ij* по символу *X*, то на пересечении строки *i* и столбца *X* записывается элемент переноса *Sj*. Если осуществляется переход в состояние *stop*, то в таблицу разбора вносится элемент останова *stop*. Следует заметить, что внесение элементов переноса не зависит от рассматриваемой *LR*(1)-грамматики или ее подкласса. Затем в таблицу разбора вносятся элементы свертки. Поскольку для *LR*(0)-грамматик не учитывается предварительно просматриваемый символ, а в *LR*(0)-автомате нет конфликтов, элементы свертки могут помещаться в каждый столбец в любом состоянии, соответствующем окончанию продукции (рис. 4.13).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Состояние | Пункты | Символ перехода | Состояние-преемник | Свертка |
| *I*0 | *S*′ → •*S*⊥ | *S* | *I*1 |  |
| *S* → •*SbA* |
| *S* → •*A* | *A* | *I*2 |  |
| *A* → •*a* | *a* | *I*3 |  |
| *I*1 | *S*′ → *S*•⊥ | ⊥ | *stop* |  |
| *S* → *S*•*bA* | *b* | *I*4 |  |
| *I*2 | *S* → *A*• |  |  | *R*2 |
| *I*3 | *A* → *a*• |  |  | *R*3 |
| *I*4 | *S* → *Sb*•*A* | *A* | *I*5 |  |
| *A* → •*a* | *a* | *I*3 |  |
| *I*5 | *S* → *SbA*• |  |  | *R*1 |

Рис. 4.12. Табличное представление *LR*(0)-автомата

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Номер  состояния | *S*′ | *S* | *A* | *a* | *b* | ⊥ |
| 0 |  | *S*1 | *S*2 | *S*3 |  |  |
| 1 |  |  |  |  | *S*4 | stop |
| 2 | *R*2 | *R*2 | *R*2 | *R*2 | *R*2 | *R*2 |
| 3 | *R*3 | *R*3 | *R*3 | *R*3 | *R*3 | *R*3 |
| 4 |  |  | *S*5 | *S*3 |  |  |
| 5 | *R*1 | *R*1 | *R*1 | *R*1 | *R*1 | *R*1 |

Рис. 4.13. *LR*(0)-таблица разбора

Построенная рассмотренным выше методом таблица разбора называется *LR*(0)-таблицей разбора, а соответствующая грамматика – *LR*(0)-грамматикой. Таким образом, чтобы определить, обладает ли грамматика признаком *LR*(0), необходимо попытаться построить *LR*(0)-таблицу разбора. Если это можно осуществить без возникновения каких-либо конфликтов (элементы свертки можно включить в каждый столбец в любом состоянии, соответствующем окончанию продукции), то грамматика является *LR*(0)-грамматикой. Если же конфликты возникают, грамматика не относится к классу *LR*(0) и следует применить более универсальные методы.

### *SLR*(1)-грамматики

Более мощный подкласс *LR*(1)-грамматик – так называемые *SLR*(1)-грамматики – простые (Simple) *LR*(1)-грамматики. Во многом *SLR*(1)-метод построения *SLR*(1)-таблицы разбора совпадает с *LR*(0)-методом. Как и в *LR*(0)-методе, пункты не содержат предпросмотра, т. е. используются *LR*(0)-пункты. Отличие заключается в том, что *SLR*(1)-таблица разбора может содержать конфликты определенного вида, которые могут быть разрешены путем просмотра очередного входного символа.

*SLR*(1)-метод требует вычисления функции

*Follow*(*X*) = {*a*⎪}, *a* ∈ *VT*, α ∈ *VT*\*, β ∈ (*VT* ∪ *VN*)\*,

для каждого нетерминала *X* грамматики. Эта функция определяет множество терминалов, которые могут следовать непосредственно за нетерминалом *X* в какой-либо сентенциальной форме, выводимой из начального нетерминала *S*. Формальный процесс вычисления функции *Follow* подробно изложен в разделе 3.4.3 при рассмотрении *LL*(1)-грамматик. Значения данной функции используются только для реализации свертки. Если имеется пункт [*A* → α•], свертка должна выполняться для всех терминалов *a* ∈ *Follow*(*A*), т. е. в *SLR*(1)-таблицу разбора элементы свертки вносятся только в столбцы, соответствующие возможным символам-следователям.

Рассмотрим процесс построения *SLR*(1)-таблицы разбора для грамматики с продукциями

1) *S* → *AB*

2) *A* → *aA*

3) *A* → *a*

4) *B* → *bB*

5) *B* → ε

Функция *Follow* для нетерминалов грамматики имеет следующие значения: *Follow*(*S*) = {⊥}, *Follow*(*A*) = {*b*, ⊥}, *Follow*(*B*) = {⊥}.

*SLR*(1)-автомат представлен на рис. 4.14. Значения функции *Follow* включены в столбец «символ перехода» для строк, соответствующих свертке. Это удобно для визуальной проверки состояний на наличие конфликтов.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Состояние | Пункты | Символ перехода | Состояние-преемник | Свертка |
| *I*0 | *S*′ → •*S*⊥ | *S* | *stop* |  |
| *S* → •*AB* | *A* | *I*1 |  |
| *A* → •*aA* | *a* | *I*5 |  |
| *A* → •*a* |
| *I*1 | *S* → *A*•*B* | *B* | *I*2 |  |
| *B* → •*bB* | *b* | *I*3 |  |
| *B* → • | ⊥ |  | *R*5 |
| *I*2 | *S* → *AB*• | ⊥ |  | *R*1 |
| *I*3 | *B* → *b*•*B* | *B* | *I*4 |  |
| *B* → •*bB* | *b* | *I*3 |  |
| *B* → • | ⊥ |  | *R*5 |
| *I*4 | *B* → *bB*• | ⊥ |  | *R*4 |
| *I*5 | *A* → *a*•*A* | *A* | *I*6 |  |
| *A* → *a*• | *b*, ⊥ |  | *R*3 |
| *A* → •*aA* | *a* | *I*5 |  |
| *A* → •*a* |
| *I*6 | *A* → *aA*• | *b*, ⊥ |  | *R*2 |

Рис. 4.14. *SLR*(1)-автомат

Следует обратить внимание на то, что для продукции *S*′ → *S*⊥ построены не все пункты, а только [*S*′ → •*S*⊥] в состоянии *I*0 и [*S*′ → *S*•⊥] в состоянии *stop* (в таблице соответствующая строка не показана). Это связано с тем, что в грамматике не существует более ни одной продукции, в правой части которой содержался бы нетерминал *S*. Поэтому пункт [*S*′ → *S*•⊥] однозначно устанавливает успешное завершение разбора, и нет смысла строить дополнительное состояние для пункта [*S*′ → *S*⊥•].

Из *SLR*(1)-автомата видно, что данная грамматика не является *LR*(0)-грамматикой, поскольку состояния *I*1, *I*3 и *I*5 имеют конфликты типа «перенос/свертка». Такие состояния называются *неадекватными*. В состояниях *I*1 и *I*3 необходимо выполнить два перехода по символам *B* и *b* и одну свертку для продукции *B* → ε. Чтобы разрешить этот конфликт, очередной входной символ сравнивается с возможным символом-следователем для нетерминала *B*, т. е. для нетерминала из левой части продукции, для которой выполняется свертка. Поскольку *Follow*(*B*) = {⊥}, то, если входным символом является символ ⊥, должна выполняться свертка. Таким образом, в состояниях *I*1 и *I*3 по символам *B* и *b* выполняется перенос, а по символу ⊥ – свертка. Эти символы не совпадают, конфликта нет, неадекватность снимается. В состоянии *I*5 перенос выполняется по символам *A* и *a*, а свертка – по символам *b* и ⊥, поскольку *Follow*(*A*) = {*b*, ⊥}, т. е. если очередным входным символом является *b* или ⊥, выполняется свертка для продукции *A* → *a*. Неадекватность снимается.

В состояниях *I*2, *I*4 и *I*6 выполняются только свертки (конфликтов нет), поэтому при построении таблицы разбора можно было бы включить соответствующие элементы свертки во все столбцы этих состояний, как это делалось в *LR*(0)-методе. Однако *SLR*(1)-метод, рассматривая только возможные символы-следователи, исключает из таблицы несколько элементов свертки. В состоянии *I*2 выполняется свертка для продукции *S* → *AB*, а *Follow*(*S*) = {⊥}, т. е. за *S* может следовать только символ ⊥ и никакой другой. Поэтому для состояния *I*2 элемент свертки *R*1 включается только в столбец ⊥. Аналогично в состоянии *I*4 элемент *R*4 (свертка для *B* → *bB*) включается в столбец ⊥, поскольку *Follow*(*B*) = {⊥}, а в состоянии *I*6 элемент *R*2 (свертка для *A* → *aA*) записывается в столбцы *b* и ⊥, так как *Follow* (*A*) = {*b*, ⊥}. Если этого не сделать, то некоторые синтаксические ошибки будут обнаружены на более поздних шагах анализа (но не позднее в смысле считанного текста). Соответствующая *SLR*(1)-таблица разбора представлена на рис. 4.15.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Номер  состояния | *S* | *A* | *B* | *a* | *b* | ⊥ |
| 0 | stop | *S*1 |  | *S*5 |  |  |
| 1 |  |  | *S*2 |  | *S*3 | *R*5 |
| 2 |  |  |  |  |  | *R*1 |
| 3 |  |  | *S*4 |  | *S*3 | *R*5 |
| 4 |  |  |  |  |  | *R*4 |
| 5 |  | *S*6 |  | *S*5 | *R*3 | *R*3 |
| 6 |  |  |  |  | *R*2 | *R*2 |

Рис. 4.15. *SLR*(1)-таблица разбора

Таким образом, если все конфликты можно разрешить рассмотренным выше способом, т. е. достаточно проанализировать значения функции *Follow* при реализации свертки, грамматика относится к подклассу *SLR*(1)-грамматик. В противном случае необходимо использовать более сложные и мощные методы построения таблиц разбора.

*SLR*(1)-метод часто используется на практике, так как большинство стандартных синтаксических конструкций языков программирования легко выражаются с помощью *SLR*(1)-грамматик. Однако возможны конструкции, с которыми *SLR*(1)-метод не может справиться.

### *LALR*(1)-грамматики

Рассмотренный *SLR*(1)-метод недостаточно мощный, поскольку он не учитывает левый контекст, необходимый в общем случае для принятия решения о действиях синтаксического анализатора для разрешения конфликтов. В ряде случаев левый контекст играет важную роль при решении вопроса о том, можно ли считать заданный символ действительно символом-следователем, хотя в смысле *SLR*(1)-метода он и является возможным символом-следователем (входит в множество, определяемое функцией *Follow*).

Левый контекст учитывается при построении таблицы разбора так называемым *LR*(1)-методом с предпросмотром (Look Ahead), или *LALR*(1)-методом. Грамматика, для которой все конфликты в таблице разбора разрешаются при использовании данного метода, относится к подклассу *LALR*(1)-грамматик.

В *LALR*(1)-методе, чтобы учитывать левый контекст, состояние *LALR*(1)-автомата должно содержать дополнительную информацию. Это достигается с помощью предпросмотра. Поэтому в этом методе используется *LR*(1)-пункт вида [*A* → α•β, *a*], где терминал *a* является предпросмотром пункта. Множество таких терминалов всегда является подмножеством множества, определяемого функцией *Follow*.

Метод построения *LALR*(1)-таблицы разбора, по сути, тот же, что и *SLR*(1)-метод. Отличия связаны с определением предпросмотров пунктов и их использованием для разрешения конфликтов. Предпросмотр не играет никакой роли в пункте вида [*A* → α•β, *a*], где β ≠ ε, а важен для пункта вида [*A* → α•, *a*], когда выполняется свертка для продукции *A* → α, если очередным входным символом является терминал *a*.

Предпросмотры определяются при выполнении операции замыкания. Пусть имеется *LR*(1)-пункт [*A* → α•*B*β, *a*], *B* ∈ *VN*. Тогда должна существовать правосторонняя схема вывода

, где δ ∈ (*VT* ∪ *VN*)\*, *x* ∈ *VT*\*.

Предположим, что β*ax* генерирует терминальную строку *by*, *b* ∈ *VT*, *y* ∈ *VT*\*. Тогда для любой продукции вида *B* → γ, γ ∈ (*VT* ∪ *VN*)\*, можно получить схему вывода

,

т. е. пункт [*B* → •γ, *b*] должен войти в это же состояние. Необходимо обратить внимание на то, что символ *b* может быть первым терминалом, выводимым из β; возможно также, что β может генерировать пустую строку ε, в этом случае *b* = *a*. Следовательно, символ *b* может быть любым терминалом из множества *First*(β*a*), где *First* – функция, подробно описанная в разделе 3.4.3 при рассмотрении *LL*(1)-грамматик.

Таким образом, операция замыкания заключается в следующем. Если пункт вида [*A* → α•*B*β, *a*] входит в некоторое состояние, то все пункты вида [*B* → •γ, *b*] для всех терминалов *b* из *First*(β*a*) включаются в это же состояние. Процесс продолжается рекурсивно до тех пор, пока в состояние не будут включены все возможные пункты.

Множество первых элементов *LR*(1)-пунктов, входящих в одно состояние, называется *ядром* состояния. *LALR*(1)-метод не допускает разных состояний с одинаковыми ядрами, даже если предпросмотры пунктов отличаются. Поэтому если грамматика дает более двух состояний с одним и тем же ядром, то эти состояния должны быть объединены в одно состояние. В остальных случаях выполнение перехода из состояния в состояние-преемник не влияет на предпросмотр пункта.

Рассмотрим *LALR*(1)-метод на примере грамматики с продукциями

1) *S* → *AaBb*

2) *A* → *C*

3) *A* → *db*

4) *B* → *C*

5) *C* → *Ccd*

6) *C* → *d*

Для сравнения с *SLR*(1)-методом для всех нетерминалов определим значения функции *Follow*:

*Follow*(*S*) = {⊥},

*Follow*(*A*) = {*a*},

*Follow*(*B*) = {*b*},

*Follow*(*C*) = {*a*, *b*, *c*}.

Базовым пунктом для исходного состояния *I*0 *LALR*(1)-автомата является начальный пункт [*S*′ → •*S*⊥, ε], в котором предпросмотром является пустая строка ε, так как за *S*′ не может следовать никакой символ. Поскольку маркерная точка стоит перед нетерминалом *S*, а *First*(⊥) = {⊥}, необходимо добавить пункт [*S* → •*AaBb*, ⊥]. Продолжим вычисление замыкания. Поскольку *First*(*aBb*⊥) = {*a*}, добавляются пункты [*A* → •*C*, *a*] и [*A* → •*db*, *a*]. Маркерная точка стоит перед нетерминалом *C*, *First*(*a*) = {*a*}, поэтому добавляются [*C* → •*Ccd*, *a*] и [*C* → •*d*, *a*]. Опять маркерная точка перед нетерминалом *C*, *First*(*cda*) = {*c*}, добавляются пункты [*C* → •*Ccd*, *c*] и [*C* → •*d*, *c*]. Больше никакие новые пункты добавить нельзя, операция замыкания для состояния *I*0 завершена. Заметим, что в одном состоянии могут быть *LR*(1)-пункты с одинаковыми первыми элементами, но с разными предпросмотрами, например, [*C* → •*d*, *a*] и [*C* → •*d*, *c*]. Для удобства записи предпросмотры пунктов будем представлять как множества, а пункты с одинаковыми первыми элементами – как один пункт (хотя формально это разные пункты), предпросмотром которого является объединение множеств их предпросмотров, например, пункты [*C* → •*d*, *a*] и [*C* → •*d*, *c*] будут записываться как [*C* → •*d*, {*a*, *c*}].

Состояния-преемники определяются так же, как и в ранее описанных методах, т. е. по первым элементам пунктов. *LALR*(1)-автомат для рассматриваемой грамматики представлен на рис. 4.16 (символы свертки указаны в столбце «символ перехода»), а *LALR*(1)-таблица разбора – на рис. 4.17.

Следует обратить внимание на то, что для состояния *I*2 в соответствии с пунктом [*C* → *C*•*cd*, {*a*, *c*}] по символу *c* образуется преемник с базовым пунктом [*C* → *Cc*•*d*, {*a*, *c*}], а для состояния *I*8 в соответствии с [*C* → *C*•*cd*, {*b*, *c*}] – преемник с базовым пунктом [*C* → *Cc*•*d*, {*b*, *c*}]. Ядра этих состояний-преемников совпадают, поэтому, согласно требованию *LALR*(1)-метода, они слиты в одно состояние *I*5, а предпросмотры их пунктов объединены в единое множество, т. е. получилось состояние с пунктом [*C* → *Cc*•*d*, {*a*, *b*, *c*}].

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Состояние | Пункты | Символ перехода | Состояние-преемник | Свертка |
| *I*0 | *S*′ → •*S*⊥, {ε} | *S* | *stop* |  |
| *S* → •*AaBb*, {⊥} | *A* | *I*1 |  |
| *A* → •*C*, {*a*} | *C* | *I*2 |  |
| *C* → •*Ccd*, {*a*, *c*} |
| *A* → •*db*, {*a*} | *d* | *I*3 |  |
| *C* → •*d*, {*a*, *c*} |
| *I*1 | *S* → *A*•*aBb*, {⊥} | *a* | *I*4 |  |
| *I*2 | *A* → *C*•, {*a*} | *a* |  | *R*2 |
| *C* → *C*•*cd*, {*a*, *c*} | *c* | *I*5 |  |
| *I*3 | *A* → *d*•*b*, {*a*} | *b* | *I*6 |  |
| *C* → *d*•, {*a*, *c*} | *a*, *c* |  | *R*6 |
| *I*4 | *S* → *Aa*•*Bb*, {⊥} | *B* | *I*7 |  |
| *B* → •*C*, {*b*} | *C* | *I*8 |  |
| *C* → •*Ccd*, {*b*, *c*} |
| *C* → •*d*, {*b*, *c*} | *d* | *I*9 |  |
| *I*5 | *C* → *Cc*•*d*, {*a*, *b*, *c*} | *d* | *I*10 |  |
| *I*6 | *A* → *db*•, {*a*} | *a* |  | *R*3 |
| *I*7 | *S* → *AaB*•*b*, {⊥} | *b* | *I*11 |  |
| *I*8 | *B* → *C*•, {*b*} | *b* |  | *R*4 |
| *C* → *C*•*cd*, {*b*, *c*} | *c* | *I*5 |  |
| *I*9 | *C* → *d*•, {*b*, *c*} | *b*, *c* |  | *R*6 |
| *I*10 | *C* → *Ccd*•, {*a*, *b*, *c*} | *a*, *b*, *c* |  | *R*5 |
| *I*11 | *S* → *AaBb*•, {⊥} | ⊥ |  | *R*1 |

Рис. 4.16. *LALR*(1)-автомат

Все конфликты для рассматриваемой грамматики успешно разрешены, неадекватных состояний нет, следовательно, она является *LALR*(1)-грамматикой. Эта грамматика не обладает признаком *LR*(0), так как нельзя, например, поместить все элементы свертки *R*2 в каждый столбец для состояния *I*2. Не обладает она также и признаком *SLR*(1), так как в противном случае в состоянии *I*3, где выполняется свертка для продукции *C* → *d*, можно было бы занести элемент свертки *R*6 в столбец, соответствующий терминалу *b*, поскольку *Follow*(*C*) = {*a*, *b*, *c*}. Однако это привело бы к конфликту с уже имеющимся в этой позиции элементом переноса *S*6, т. е. *LALR*(1)-метод устанавливает, что в состоянии *I*3 действительными символами-следователями являются только терминалы *a* и *c*, а символ *b* не может быть символом-следователем. По аналогичной причине в состоянии *I*9 в столбце, соответствующем терминалу *a*, нет элемента свертки *R*6, поскольку в этом состоянии действительными символами-следователями могут быть только терминалы *b* и *c*.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номер  состояния | *S* | *A* | *B* | *C* | *a* | *b* | *c* | *d* | ⊥ |
| 0 | *stop* | *S*1 |  | *S*2 |  |  |  | *S*3 |  |
| 1 |  |  |  |  | *S*4 |  |  |  |  |
| 2 |  |  |  |  | *R*2 |  | *S*5 |  |  |
| 3 |  |  |  |  | *R*6 | *S*6 | *R*6 |  |  |
| 4 |  |  | *S*7 | *S*8 |  |  |  | *S*9 |  |
| 5 |  |  |  |  |  |  |  | *S*10 |  |
| 6 |  |  |  |  | *R*3 |  |  |  |  |
| 7 |  |  |  |  |  | *S*11 |  |  |  |
| 8 |  |  |  |  |  | *R*4 | *S*5 |  |  |
| 9 |  |  |  |  |  | *R*6 | *R*6 |  |  |
| 10 |  |  |  |  | *R*5 | *R*5 | *R*5 |  |  |
| 11 |  |  |  |  |  |  |  |  | *R*1 |

Рис. 4.17. *LALR*(1)-таблица разбора

*LALR*(1)-метод важен с практической точки зрения, поскольку для стандартных синтаксических конструкций языков программирования там, где не приводит к успеху *SLR*(1)-метод, обычно справляется *LALR*(1)-метод, и достаточно редко приходится применять наиболее общий *LR*(1)-метод.

### *LR*(1)-грамматики

Наиболее общей и мощной технологией построения таблиц разбора является *LR*(1)-метод. Он, по сути, тот же, что и рассмотренный выше *LALR*(1)-метод. Отличие заключается в том, что в *LR*(1)-методе состояния с идентичными ядрами, но с несовпадающими предпросмотрами пунктов считаются различными, т. е. их нельзя объединять, как это делалось в *LALR*(1)-методе. Это позволяет разрешать ряд конфликтов, с которыми не справляется *LALR*(1)-метод.

Рассмотрим грамматику с продукциями

1) *S* → *aAd*

2) *S* → *bBd*

3) *S* → *aBe*

4) *S* → *bAe*

5) *A* → *c*

6) *B* → *c*

*LR*(1)-автомат представлен на рис. 4.18, а соответствующая *LR*(1)-таблица разбора – на рис. 4.19.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Состояние | Пункты | Символ перехода | Состояние-преемник | Свертка |
| *I*0 | *S*′ → •*S*⊥, {ε} | *S* | *stop* |  |
| *S* → •*aAd*, {⊥} | *a* | *I*1 |  |
| *S* → •*aBe*, {⊥} |
| *S* → •*bBd*, {⊥} | *b* | *I*2 |  |
| *S* → •*bAe*, {⊥} |
| *I*1 | *S* → *a*•*Ad*, {⊥} | *A* | *I*3 |  |
| *S* → *a*•*Be*, {⊥} | *B* | *I*4 |  |
| *A* → •*c*, {*d*} | *c* | *I*5 |  |
| *B* → •*c*, {*e*} |
| *I*2 | *S* → *b*•*Bd*, {⊥} | *B* | *I*6 |  |
| *S* → *b*•*Ae*, {⊥} | *A* | *I*7 |  |
| *A* → •*c*, {*e*} | *c* | *I*8 |  |
| *B* → •*c*, {*d*} |
| *I*3 | *S* → *aA*•*d*, {⊥} | *d* | *I*9 |  |
| *I*4 | *S* → *aB*•*e*, {⊥} | *e* | *I*10 |  |
| *I*5 | *A* → *c*•, {*d*} | *d* |  | *R*5 |
| *B* → *c*•, {*e*} | *e* |  | *R*6 |
| *I*6 | *S* → *bB*•*d*, {⊥} | *d* | *I*11 |  |
| *I*7 | *S* → *bA*•*e*, {⊥} | *e* | *I*12 |  |
| *I*8 | *A* → *c*•, {*e*} | *e* |  | *R*5 |
| *B* → *c*•, {*d*} | *d* |  | *R*6 |
| *I*9 | *S* → *aAd*•, {⊥} | ⊥ |  | *R*1 |
| *I*10 | *S* → *aBe*•, {⊥} | ⊥ |  | *R*3 |
| *I*11 | *S* → *bBd*•, {⊥} | ⊥ |  | *R*2 |
| *I*12 | *S* → *bAe*•, {⊥} | ⊥ |  | *R*4 |

Рис. 4.18. *LR*(1)-автомат

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номер  состояния | *S* | *A* | *B* | *a* | *b* | *c* | *d* | *e* | ⊥ |
| 0 | stop |  |  | *S*1 | *S*2 |  |  |  |  |
| 1 |  | *S*3 | *S*4 |  |  | *S*5 |  |  |  |
| 2 |  | *S*7 | *S*6 |  |  | *S*8 |  |  |  |
| 3 |  |  |  |  |  |  | *S*9 |  |  |
| 4 |  |  |  |  |  |  |  | *S*10 |  |
| 5 |  |  |  |  |  |  | *R*5 | *R*6 |  |
| 6 |  |  |  |  |  |  | *S*11 |  |  |
| 7 |  |  |  |  |  |  |  | *S*12 |  |
| 8 |  |  |  |  |  |  | *R*6 | *R*5 |  |
| 9 |  |  |  |  |  |  |  |  | *R*1 |
| 10 |  |  |  |  |  |  |  |  | *R*3 |
| 11 |  |  |  |  |  |  |  |  | *R*2 |
| 12 |  |  |  |  |  |  |  |  | *R*4 |

Рис. 4.19. *LR*(1)-таблица разбора

Грамматика является *LR*(1)-грамматикой, поскольку все конфликты успешно разрешаются. Но она не относится к подклассу *LALR*(1), так как в соответствии с *LALR*(1)-методом состояния *I*5 и *I*8, имеющие одно и то же ядро {[*A* → *c*•], [*B* → *c*•]}, должны объединяться в одно состояние с пунктами [*A* → *c*•, {*d*, *e*}] и [*B* → *c*•, {*d*, *e*}], что вызывает конфликт «сверка/сверка».

Рассмотренная выше грамматика специально усложнена для иллюстрации *LR*(1)-метода. Генерируемый ею язык включает в себя четыре строки и может быть представлен эквивалентной *LR*(0)-грамматикой с продукциями *S* → *acd* | *bcd* | *ace* | *bce*.

Если для некоторой грамматики применить все рассмотренные методы построения таблиц разбора, то *LR*(0)-, *SLR*(1)- и *LALR*(1)-таблицы разбора будут иметь одинаковое число состояний, а *LR*(1)-таблица разбора – значительно больше. Таким образом, методы *LR*(0), *SLR*(1) и *LALR*(1) проще и экономичнее общего *LR*(1)-метода. Поскольку большинство конструкций языков программирования легко представляются с помощью *SLR*(1)- или *LALR*(1)-грамматик, лучше сначала опробовать *SLR*(1)-метод. При успешной попытке грамматика будет *SLR*(1)-грамматикой. В противном случае пробуется *LALR*(1)-метод, и если это разрешает все конфликты, то данная грамматика обладает признаком *LALR*(1). Если конфликты остаются, то используется наиболее общий *LR*(1)-метод, и если он не приводит к успеху, то либо необходимо преобразовать грамматику, либо язык не относится к классу *LR*(1)-языков и, следовательно, для него не может существовать *LR*(1)-грамматики.

Очевидно, что классификация *LR*(1)-грамматик включающая, т. е. все грамматики с признаками *LR*(0), *SLR*(1) и *LALR*(1) являются *LR*(1)-грамматиками, все грамматики с признаками *LR*(0) и *SLR*(1) являются *LALR*(1)-грамматиками и т. д.

Рассмотренные выше таблицы разбора обеспечивают быструю выборку и широкие диагностические возможности. Главный их недостаток – для хранения требуется большой объем памяти. Можно использовать известные методы хранения неплотных матриц, но обычно это достигается за счет увеличения времени разбора и более позднего обнаружения синтаксических ошибок.

## Синтаксический анализ

Работа синтаксического анализатора типа перенос-сверка совершенно не зависит от *LR*-метода, использованного для построения таблицы разбора, и от разбираемого языка (грамматика языка определяет содержимое таблицы, но не влияет на сам алгоритм анализа). Главное требование – *LR*-таблица разбора не должна содержать конфликтов. *LR*-анализатор считывает по одному символы входной строки слева направо и в процессе анализа переходит из одного состояния в другое. При этом используется два стека: стек символов (при практической реализации в нем нет необходимости) и стек состояний. Анализатор находится в состоянии, хранящемся в вершине стека состояний. Следующий шаг синтаксического анализатора зависит от элемента таблицы разбора, позиция которого определяется текущим состоянием (находится в вершине стека состояний) и входным символом. В качестве входного символа может быть текущий символ (терминал) входной строки или нетерминал из левой части продукции, для которой на предыдущем шаге выполнялась свертка. Изначально в стеке состояний содержится исходное состояние *I*0 анализатора. Процесс анализа продолжается до тех пор, пока не будет достигнуто успешное завершение (элемент *stop* в таблице разбора) или не обнаружится синтаксическая ошибка (пустая ячейка в таблице разбора, позиция которой может дать информацию о характере ошибки).

Рассмотрим работу анализатора для *LALR*(1)-грамматики, иллюстрирующей работу *LALR*(1)-метода (см. разд. 4.6.3), и соответствующей *LALR*(1)-таблицы разбора (см. табл. 4.17). Процесс разбора строки *dbadcdb*⊥, которая выводится в соответствии с правосторонней схемой

S⊥ ⇒ *AaBb*⊥ ⇒ *AaCb*⊥ ⇒ *AaCcdb*⊥ ⇒ *Aadcdb*⊥ ⇒ *dbadcdb*⊥,

показан на рис. 4.20. Содержимое каждого стека представляется строкой, в которой самый правый символ находится в вершине стека, символ ⊥ показывает дно стека. В стеке состояний указаны номера состояний.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Номер  шага | Вводимая  строка | Стек  символов | Стек  состояний | Выполняемые  действия |
| 1 | *dbadcdb*⊥ | ⊥ | ⊥0 | Перенос *S*3 |
| 2 | *badcdb*⊥ | ⊥*d* | ⊥0,3 | Перенос *S*6 |
| 3 | *adcdb*⊥ | ⊥*db* | ⊥0,3,6 | Сверка *R*3 для *A* → *db* |
| 4 | *adcdb*⊥ | ⊥ | ⊥0 | Входной символ *A*. Перенос *S*1 |
| 5 | *adcdb*⊥ | ⊥*A* | ⊥0,1 | Перенос *S*4 |
| 6 | *dcdb*⊥ | ⊥*Aa* | ⊥0,1,4 | Перенос *S*9 |
| 7 | *cdb*⊥ | ⊥*Aad* | ⊥0,1,4,9 | Сверка *R*6 для *C* → *d* |
| 8 | *cdb*⊥ | ⊥*Aa* | ⊥0,1,4 | Входной символ *C*. Перенос *S*8 |
| 9 | *cdb*⊥ | ⊥*AaC* | ⊥0,1,4,8 | Перенос *S*5 |
| 10 | *db*⊥ | ⊥*AaCc* | ⊥0,1,4,8,5 | Перенос *S*10 |
| 11 | *b*⊥ | ⊥*AaCcd* | ⊥0,1,4,8,5,10 | Сверка *R*5 для *C* → *Ccd* |
| 12 | *b*⊥ | ⊥*Aa* | ⊥0,1,4 | Входной символ *C*. Перенос *S*8 |
| 13 | *b*⊥ | ⊥*AaC* | ⊥0,1,4,8 | Сверка *R*4 для *B* → *C* |
| 14 | *b*⊥ | ⊥*Aa* | ⊥0,1,4 | Входной символ *B*. Перенос *S*7 |
| 15 | *b*⊥ | ⊥*AaB* | ⊥0,1,4,7 | Перенос *S*11 |
| 16 | ⊥ | ⊥*AaBb* | ⊥0,1,4,7,11 | Сверка *R*1 для *S* → *AaBb* |
| 17 | ⊥ | ⊥ | ⊥0 | Входной символ *S*. *stop*.  Разбор успешно завершен |

Рис. 4.20. Процесс разбора строки *dbadcdb*⊥

На шаге 1 входным является символ *d*, анализатор находится в состоянии 0. Поскольку в столбце *d* таблицы разбора для данного состояния содержится элемент *S*3, выполняется перенос: в стек символов помещается символ *d*, в стек состояний – состояние 3, и анализатор переходит в состояние 3. На шаге 2 входной символ – *b*, элемент таблицы – *S*6, следовательно, выполняется перенос: в стек символов заносится *b*, в стек состояний – 6, переход в состояние 6. На шаге 3 для состояния 6 и входного символа *a* в таблице указан элемент свертки *R*3, т. е. выполняется свертка по продукции *A* → *db*. Поскольку в правой части этой продукции два символа, из стеков удаляются по два верхних элемента, т. е. из стека символов исключается основа. В результате в вершине стека состояний будет 0, т. е. анализатор переходит в состояние 0. Входным символом для следующего шага (шаг 4) будет нетерминал *A*. Последовательность выполнения действий свертки и переноса продолжается до тех пор, пока на шаге 17 элементом таблицы разбора не станет элемент *stop*. Это говорит о том, что разбор успешно завершен, входная строка принадлежит языку, т. е. анализатор принимает эту строку.

Пример обнаружения синтаксической ошибки при анализе строки *dbaab*⊥, не принадлежащей языку, показан на рис. 4.21.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Номер  шага | Вводимая  строка | Стек  символов | Стек  состояний | Выполняемые  действия |
| 1 | *dbaab*⊥ | ⊥ | ⊥0 | Перенос *S*3 |
| 2 | *baab*⊥ | ⊥*d* | ⊥0,3 | Перенос *S*6 |
| 3 | *aab*⊥ | ⊥*db* | ⊥0,3,6 | Свертка *R*3 для *A* → *db* |
| 4 | *aab*⊥ | ⊥ | ⊥0 | Входной символ *A*. Перенос *S*1 |
| 5 | *aab*⊥ | ⊥*A* | ⊥0,1 | Перенос *S*4 |
| 6 | *ab*⊥ | ⊥*Aa* | ⊥0,1,4 | Синтаксическая ошибка |

Рис. 4.21. Процесс разбора строки *dbaab*⊥, не принадлежащей языку

На шаге 6 в позиции таблицы разбора, соответствующей состоянию 4 и входному символу *a*, указан элемент ошибки (пустая ячейка), что говорит об обнаружении синтаксической ошибки. К этому моменту принята подстрока *dba*, свернутая в подстроку *Aa*. Характер этой ошибки можно определить, проанализировав строку таблицы разбора, соответствующую состоянию 4. В этой строке в столбцах *B*, *C* и *d* указаны элементы переноса, а во всех остальных – элементы ошибок. Это говорит о том, что после принятой на настоящий момент подстроки *dba* (свернутой в процессе разбора в подстроку *Aa*) может следовать только подстрока, начинающаяся с *d*, либо подстрока, выводимая из нетерминалов *B* или *C*, т. е. с того же терминала *d*. Таким образом, для определения характера ошибки достаточно проанализировать только столбцы таблицы разбора, соответствующие терминалам, для состояния, в котором обнаружилась ошибка. Возможными очередными символами входной строки могут быть только терминалы, в столбцах которых не содержатся элементы ошибок.

## Сравнение с *LL*-методом разбора

Основными достоинствами как *LL*-, так и *LR*-методов разбора являются их детерминированность и возможность обнаружения синтаксических ошибок, как правило, на самых ранних этапах анализа. Кроме того, оба метода позволяют включать в синтаксис действия для выполнения некоторых аспектов процесса трансляции.

Преимуществом *LR*-метода является то, что он применим к более широкому классу грамматик и языков и обычно не требует преобразований грамматик, которые практически всегда необходимы для *LL*-метода. Однако при наличии хорошего преобразователя грамматик сам факт необходимости преобразований грамматик в действительности не вызывает затруднений. В тех случаях, когда преобразователь грамматик не справляется с какой-либо конструкцией языка программирования, то часто эта конструкция не обладает и признаком *LR*(1). Поэтому для обоих методов возникает необходимость в ручном преобразовании грамматик. Это снижает значимость преимущества *LR*-метода перед *LL*-методом в части преобразования грамматик.

В отношении размеров таблиц разбора *LL*-метод существенно превосходит *LR*-метод. Однако использование методов оптимизации *LR*-таблиц разбора делает их размер примерно того же порядка, что и *LL*-таблицы разбора, но обычно это достигается за счет некоторого увеличения времени разбора.

Сравнение максимального, минимального и среднего времени разбора показывает, что *LL*-метод более эффективен, чем *LR*‑метод.

Упражнения

4.1. Вычислить отношения предшествования (построив матрицу предшествования) для следующих грамматик:

а) *S* → *aBA*⎪*bBbA* в) *S* → *TC* д) *S* → *abAd*

*A* → *abA*⎪*c* *T* → *aTb*⎪*ab* *A* → *B*⎪*BeA*

*B* → *a*⎪*ab* *C* → *cC*⎪*c* *B* → *fD*

б) *S* → *ABC* г) *S* → *AB* *D* → *t*⎪*teD*

*A* → *a*⎪*Cb* *A* → *Ba*⎪*b*

*B* → *c*⎪*dA* *B* → *Cb*⎪*C*

*C* → *e*⎪*f* *C* → *c*⎪*a*

Являются ли они грамматиками простого предшествования или слабого предшествования?

4.2. Для грамматик простого предшествования из упр. 4.1 построить функции предшествования.

4.3. Вычислить отношения операторного предшествования для следующих грамматик:

а) *S* → *aAc*⎪*b* в) *S* → *aAb*⎪*d* д) *S* → *aAd*⎪*aBd*⎪*bAe*

*A* → *aSc*⎪*b* *A* → *AgS*⎪*S* *A* → *tv*

б) *S* → *Ac*⎪*Bd* г) *S* → *bFbB*⎪*aFaB* *B* → *tw*

*A* → *a* *F* → *a*⎪*ab*

*B* → *a* *B* → *aB*⎪*a*

Являются ли они грамматиками операторного предшествования?

4.4. Для грамматик из упр. 4.1 и 4.3 построить *LR*-таблицы разбора. Определить, являются ли они *LR*(1)-грамматиками, если да, то к какому подклассу *LR*(1)-грамматик они относятся.

4.5. Разработать алгоритм синтаксического анализа для:

а) грамматик простого предшествования;

б) грамматик слабого предшествования;

в) грамматик операторного предшествования;

г) *LR*(1)-грамматик.

# 

ЗАКЛЮЧЕНИЕ

В данном учебном пособии была сделана попытка обобщить сведения об основных методах лексического и синтаксического анализа и проектирования модулей компилятора, реализующих эти фазы компиляции.

Автор не ставил перед собой цель рассмотреть программную реализацию соответствующих модулей в какой-либо среде программирования (эти задачи должны решаться в процессе выполнения соответствующих упражнений и проведения лабораторных занятий), а решил ограничиться изложением теоретических основ и формальных методов реализации.

Изучение изложенного в пособии теоретического материала, практическая реализация рассмотренных методов в какой-либо среде программирования должно способствовать формированию соответствующих компетенций, связанных с решением следующих основных задач:

а) развитие навыков выбора и применения моделей и методов теории формальных языков и грамматик при конструировании языков программирования и разработке модулей лексического и синтаксического анализа компилятора;

б) формирование умения правильно выбирать алгоритмы и структуры данных при проектировании модулей лексического и синтаксического анализа с целью повышения эффективности их реализации;

в) обеспечение получения практического опыта разработки модулей лексического и синтаксического анализа компилятора как составной части технологии разработки компиляторов.

Список рекомендуемой литературы

1. *Ахо А*. Компиляторы: принципы, технологии и инструментарий / А. Ахо, М. Лам, Р. Сети, Д. Ульман. – 2-е изд.– М.: Вильямс, 2015. – 1184 с.
2. *Ахо А*. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Д. Ульман. – М.: Вильямс, 2003. – 768 с.
3. *Вирт Н*. Построение компиляторов / Н. Вирт. – М.: ДМК-Пресс, 2016. – 192 с.
4. *Гавриков* *М.М*. Теоретические основы разработки и реализации языков программирования: учеб. пособие / М.М. Гавриков, А.Н. Иванченко, Д.В. Гринченков. – М.: КНОРУС, 2010. – 184 с.
5. *Компаниец Р.И*. Системное программирование. Основы построения трансляторов. – 2-е изд. – Р.И. Компаниец, Е.В. Маньков, Н.Е. Филатов. – СПб.: КОРОНА принт, 2004.–256 с.
6. *Лаздин А*.*В*. Формальные языки, грамматики, автоматы: учеб. пособие / А.В. Лаздин. – СПб.: Университет ИТМО, 2019. – 99 с.
7. *Льюис* *Ф*. Теоретические основы проектирования компиляторов / Ф. Льюис, Д. Розенкранц, Р. Стирнз. – М.: Мир, 1979. – 654 с.
8. *Опалева* *Э*.*А*. Языки программирования и методы трансляции / Э.А. Опалева, В.П. Самойленко. – СПб.: БХВ-Петербург, 2005. – 480 с.
9. *Павлов Л.А*. Восходящий синтаксический анализ: конспект лекций / Л.А. Павлов. – Чебоксары: Изд-во Чуваш. ун-та, 2004. – 44 с.
10. *Павлов Л.А*. Нисходящий синтаксический анализ: конспект лекций / Л.А. Павлов. – Чебоксары: Изд-во Чуваш. ун-та, 2003. – 48 с.
11. *Павлов Л.А*. Структуры и алгоритмы обработки данных: учеб. пособие / Л.А. Павлов, Н.В. Первова. – 2-е изд. испр. и доп. – Чебоксары: Изд-во Чуваш. ун-та, 2018. – 254 с.
12. *Павлов Л.А*. Структуры и алгоритмы обработки данных: учебник для вузов / Л.А. Павлов, Н.В. Первова. – 3-е изд. – СПб.: Лань, 2021. – 256 с.
13. *Пратт Т*. Языки программирования: реализация и разработка. – Т. Пратт, М. Зелковиц. – СПб.: Питер, 2001.– 688 с.
14. *Рейуорд-Смит В*. *Дж*. Теория формальных языков. Вводный курс / В. Дж. Рейуорд-Смит. – М.: Радио и связь, 1988. – 128 с.
15. *Свердлов* *С.З*. Конструирование компиляторов: учеб. пособие / С.З. Свердлов. – LAP Lambert Academic Publishing, 2015. – 571 с.
16. *Свердлов* *С.З*. Языки программирования и методы трансляции: учеб. пособие / С.З. Свердлов. – 2-е изд., испр. – СПб.: Лань, 2019. – 564 с.
17. *Свердлов* *С.З*. Языки программирования и методы трансляции: учеб. пособие / С.З. Свердлов. – СПб.: Питер, 2007. – 638 с.
18. *Хантер Р.* Проектирование и конструирование компиляторов / Р. Хантер. – М.: Финансы и статистика, 1984. – 232 с.

*Учебное издание*

**Павлов** Леонид Александрович

**ЛЕКСИЧЕСКИЙ И СИНТАКСИЧЕСКИЙ АНАЛИЗ**

**Учебное пособие**

Редактор *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Компьютерная верстка и правка *Л.А. Павлова*

Согласно Закону № 436-ФЗ от 29 декабря 2010 года

данная продукция не подлежит маркировке

Подписано в печать 10.12.2018. Формат 60×84/16.

Бумага газетная. Печать офсетная. Гарнитура Times New Roman.

Усл. печ. л. 14,77. Уч.-изд. л. 14,25. Тираж 200 экз. Заказ № 1115.

Издательство Чувашского университета

Типография университета

428015 Чебоксары, Московский просп., 15