ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ БЮДЖЕТНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

О.В. Молдованова

ЯЗЫКИ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Учебное пособие

Новосибирск
2012

УДК 004.4'4
ББК 32.973.26-018.2

Молдованова О.В. Языки программирования и методы трансляции: Учебное пособие. – Новосибирск/СибГУТИ, 2012. – 134с.

Учебное пособие предназначено для студентов, обучающихся по направлению 230100 «Информатика и вычислительная техника» и изучающих дисциплину «Теория языков программирования и методы трансляции». В нём содержится материал, предназначенный для проведения практических или лабораторных занятий по указанному учебному курсу с целью изучения основных принципов теории формальных языков и грамматик, а также методов лексического и синтаксического анализа современных языков программирования.

Кафедра вычислительных систем

Ил. – 20, табл. – 11, список лит. – 9 наим.

Рецензенты: ведущий научный сотрудник КТИ ВТ СО РАН д.т.н., с.н.с. Окольнишников В.В.; доцент кафедры прикладной математики и кибернетики ФГОБУ ВПО «СибГУТИ» к.т.н. Ситняковская Е.И.

Для студентов, обучающихся по направлению 230100 «Информатика и вычислительная техника».

Утверждено редакционно-издательским советом ФГОБУ ВПО «СибГУТИ» в качестве учебного пособия.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Основы теории формальных языков и грамматик были заложены Н. Хомским в 40–50-е годы XX века в связи с его лингвистическими работами, посвящёнными изучению естественных языков. Но уже в следующем десятилетии синтаксически управляемый перевод нашёл широкое практическое применение в области реализации средств разработки программного обеспечения для электронных вычислительных машин.

В настоящее время существует множество разнообразных языков программирования. Их практическое использование невозможно без соответствующей системы программирования, основу которой составляет транслятор или компилятор.

В рамках подготовки специалистов по направлению 230100 «Информатика и вычислительная техника» уделяется внимание изучению принципов построения компиляторов. Согласно Федеральному государственному образовательному стандарту знакомство с этими принципами студенты получают в рамках курса «Теория языков программирования и методы трансляции», отнесённого к вариативной части профессионального блока дисциплин. Важной является организация практических занятий, помогающих студентам закрепить полученные на лекциях теоретические знания.

Данное учебное пособие предназначено для студентов, обучающихся по профилю подготовки 230101 – «Вычислительные машины, комплексы, системы и сети».

В пособии излагаются принципы и методы, лежащие в основе всех современных языков программирования и базирующиеся на теории формальных языков и грамматик. Разработка компилятора даёт студентам возможность совершенствовать свои программистские навыки, глубже понять устройство самих языков программирования.

В учебном пособии представлен материал, предназначенный для организации практических занятий по изучению основных принципов построения компиляторов для современных языков программирования. Все примеры разработаны для выполнения под управлением свободно распространяемой версии операционной системы семейства UNIX, называемой Linux.

ГЛАВА 1
ОСНОВНЫЕ КОНЦЕПЦИИ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

С момента создания первых электронных вычислительных машин (ЭВМ), прилагались усилия по упрощению и автоматизации кодирования программ. Первые ЭВМ программировались исключительно на машинных языках (последовательности нулей и единиц), которые явно указывали компьютеру, какие операции и в каком порядке должны быть выполнены. Надо отметить, что такое программирование было очень медленным, утомительным и подверженным ошибкам.

Прообразом современных языков высокого уровня является Планкалкюль – язык, созданный в 1945 году немецким инженером Конрадом Цузе[1] (нем. Konrad Zuse). Он написал брошюру, где рассказал о своем творении и возможности его использования для решения разнообразных задач, включая сортировку чисел и выполнение арифметических действий в двоичной записи. Цузе написал 49 страниц фрагментов программ на Планкалкюле, которые позволяли компьютеру оценивать шахматные позиции. Но этот язык так и не был реализован.

Первым шагом в создании более дружественных языков программирования была разработка мнемонических ассемблерных языков в начале 1950-х годов. Такие языки называются *языками низкого уровня*, потому что их операторы близки к машинному коду и ориентированы на конкретный тип процессора. Изначально команды ассемблера являлись всего лишь мнемоническими представлениями машинных команд. Позже в языки ассемблера были введены макросы, так что программист мог определять параметризованные сокращения для часто используемых последовательностей машинных команд.

Важной вехой в реализации *высокоуровневых языков программирования* стала разработка во второй половине 1950-х годов языков программирования Фортран (англ. Fortran) и Алгол (англ. Algol) – для научных вычислений, Кобол (англ. Cobol) – для обработки бизнес-данных и Лисп (англ. Lisp) – для символьных вычислений. Философия, стоящая за этими языками, заключается в создании высокоуровневой системы обозначений, облегчающей программисту написание программ для численных вычислений, бизнес-приложений и символьных программ.

В последующие десятилетия было создано множество языков программирования с новыми возможностями, которые сделали написание программ более простым, естественным и надёжным.

[1] Конрад Цузе (1910 – 1995) – немецкий инженер, наиболее известен как создатель первого программируемого компьютера (1941) и первого языка программирования высокого уровня (1945).

1.1 Языки программирования

Языки программирования – это искусственные языки. Они отличаются от естественных ограниченным, достаточно малым числом слов, значение которых понятно транслятору, и очень строгими правилами записи команд (предложений). Совокупность правил, определяющих допустимые конструкции (слова, предложения) языка, т.е. его форму, образуют **синтаксис** языка, а совокупность правил, определяющих смысл синтаксически корректных конструкций языка, т.е. его содержание, – **семантику** языка.

Нарушения формы записи конструкций языка программирования приводят к *синтаксическим* ошибкам, а ошибки, связанные с неправильным содержанием действий и использованием недопустимых значений величин, называются *семантическими*. *Логические ошибки* возникают, когда все конструкции языка записаны правильно, но последовательность их выполнения дает неверный результат.

1.2 Классификация языков программирования

В настоящее время существуют тысячи языков программирования. Их можно классифицировать различными способами.

Один из способов классификации – по поколениям. Языки первого поколения – это машинные языки; языки второго поколения – языки ассемблера, а к языкам третьего поколения относятся высокоуровневые языки программирования, такие как Fortran, Algol, Cobol, Lisp, C, C++, C# и Java.

Языки четвёртого поколения – это языки программирования, разработанные для специализированных областей применения, где хороших результатов можно добиться, используя не универсальные, а проблемно-ориентированные языки, оперирующие конкретными понятиями узкой предметной области. Все языки программирования четвёртого поколения разработаны для снижения временных затрат на разработку программ. К четвёртому поколению обычно относят: языки запросов к базам данных (SQL, Informix-4GL и т.д.), языки обработки данных (Visual DataFlex, FoxPro, XBase++ и т.д.), а также средства генерации программного кода на языках третьего поколения, встраиваемые в системы программирования (см. раздел 4.3).

Термин языки пятого поколения применяется к языкам программирования, основанным на логике или ограничениях, таким как Prolog и OPS5.

Ещё одна классификация языков программирования основана на четырёх парадигмах, отражающих вычислительные модели, с помощью которых описываются существующие методы программирования:

- императивная;
- функциональная;
- декларативная;
- объектно-ориентированная.

1.2.1 Императивные языки программирования

Императивные языки описывают процесс вычислений в виде команд, изменяющих состояние программы. Эти языки разрабатывались для фон-неймановской архитектуры ЭВМ, названной так в честь её автора – Джона фон Неймана[2]. В компьютере фон Неймана данные и программы хранятся в одной и той же памяти, называемой *оперативной*. Центральный процессор получает из оперативной памяти очередную команду, декодирует её, выбирает из памяти указанные данные, выполняет команду и возвращает в память результат.

Основными элементами императивных языков являются переменные, моделирующие ячейки памяти ЭВМ; операторы присваивания, осуществляющие пересылку данных; один или несколько итеративных циклов.

Большинство императивных языков программирования включают в себя конструкции, позволяющие программировать рекурсивные алгоритмы.

К этому виду языков относятся такие распространённые языки программирования, как: Algol, Fortran, Basic, PL/1, Ada, Pascal, C, C++, Java, C#.

1.2.2 Языки функционального программирования

В языках функционального программирования вычисления в основном производятся путём применения функций к заданному набору данных. Разработка программы заключается в создании из простых функций более сложных, которые последовательно применяются к начальным данным до тех пор, пока не получится конечный результат.

На практике наибольшее распространение получил язык Lisp и два его диалекта: Common Lisp и Scheme. Основной структурой данных языка функционального программирования Lisp являются связные списки, элементами которых могут быть либо идентификаторы и числовые константы, либо другие связные списки. Областью применения этого языка являются системы искусственного интеллекта.

Другими известными языками функционального программирования являются ML, Miranda и Haskell.

Программирование, как на императивных, так и на функциональных языках является *процедурным*. Это означает, что программы на этих языках содержат указания, *как* нужно выполнять вычисления.

1.2.3 Декларативные языки

Термин **декларативный** используется по отношению к языкам, которые указывают, *какие* вычисления должны быть выполнены. Типичным примером таких языков являются языки логического программирования (основанные на системе правил). Операторы в программе на языке логического программирования выполняются не в том порядке, в каком они записаны, а в порядке, определяемом системой реализации правил.

---

[2] Джон фон Нейман (1903 – 1957) –американский математик, сделавший важный вклад в квантовую физику, квантовую логику, функциональный анализ, теорию множеств, информатику, экономику и другие отрасли науки. Наиболее известен как создатель архитектуры ЭВМ.

Наиболее распростанённым декларативным языком является язык Prolog. Основными областями его применения являются экспертные системы, системы обработки текста на естественных языках и системы управления реляционными базами данных.

1.2.4 Объектно-ориентированные языки

Объектно-ориентированные языки – это языки программирования, поддерживающие концепцию объектно-ориентированного программирования. Вместо проблемы разбиения задачи на функции, в объектно-ориентированном программировании задача представляется в виде совокупности объектов, обладающих сходными свойствами и набором действий, которые можно с ними производить.

Объекты являются экземплярами *классов*. Класс является формой, определяющей, какие данные и функции будут включены в объект класса. Таким образом, класс – это описание совокупности сходных между собой объектов.

Концепция объектно-ориентированного программирования складывается из трёх ключевых понятий: *абстракция данных*, *наследование* и *полиформизм*.

Абстракция данных позволяет *инкапсулировать* множество объектов данных (члены класса) и набор абстрактных операций над этими объектами данных (методы класса), ограничивая доступ к данным только через определённые абстрактные операции.

В повседневной жизни мы часто сталкиваемся с разбиением классов на подклассы. Например, класс «наземный транспорт» делится на подклассы «автомобили», «автобусы», «мотоциклы» и т.д. Принцип, положенный в основу такого деления, заключается в том, что каждый подкласс обладает свойствами, присущими тому классу, из которого он выделен. В программировании класс также может породить множество подклассов. Класс, порождающий другие классы, называется *базовым*. Порождённые им классы наследуют его свойства, но могут обладать и собственными. Такие классы называются *производными*. А сам процесс выделения производных классов из базового – *наследованием*.

Использование операций и функций различным образом в зависимости от того, с какими типами величин они работают, называется *полиморфизмом*. Когда существующая операция наделяется возможностью совершать действия над операндами нового типа, говорят, что она является *перегруженной*.

Наиболее ранними объектно-ориентированными языками программирования стали Simula 67 и Smalltalk; примерами более поздних объектно-ориентированных языков программирования являются C++, C#, Java и Ruby.

1.3 Критерии оценки языков программирования

Достаточно часто возникает вопрос о том, какой язык программирования предпочтительней. Универсального ответа на вопрос о лучшем языке, конечно, нет. Можно говорить лишь о выборе языка для конкретной задачи, а точнее, для конкретных обстоятельств.

Рассмотрим свойства, которыми должны в той или иной мере обладать языки программирования.

Понятность конструкций языка – это свойство, обеспечивающее лёгкость восприятия программ человеком. Данное свойство языка программирования зависит от выбора ключевых слов и такой нотации, которая позволяла бы при чтении текста программы легко выделять основные понятия каждой конкретной части программы, а также от возможности построения модульных программ. Высокая степень понятности конструкций языка программирования позволяет быстрее находить ошибки.

Под **надёжностью** понимается степень автоматического обнаружения ошибок, которое может быть выполнено транслятором. Надёжный язык позволяет выявлять большинство ошибок во время трансляции программы, а не во время её выполнения. Существует несколько способов проверки правильности выполнения программой своих функций: использование формальных методов верификации программ, проверка путём чтения текста программы, прогон программы с тестовыми наборами данных.

Гибкость языка программирования проявляется в том, сколько возможностей он предоставляет программисту для выражения всех операций, которые требуются в программе.

Простота языка обеспечивает лёгкость понимания семантики языковых конструкций и запоминания их синтаксиса. Простой язык предоставляет ясный, простой и единообразный набор понятий. При этом желательно иметь минимальное количество различных понятий с как можно более простыми и систематизированными правилами их комбинирования, т.е. язык должен обладать свойством *концептуальной целостности*.

Естественность. Язык должен содержать такие структуры данных, управляющие сруктуры и операции, а также иметь такой синтаксис, которые позволяли бы отражать в программе логические структуры, лежащие в основе реализуемого алгоритма.

Мобильность. Язык, независимый от аппаратуры, предоставляет возможность переносить программы с одной платформы на другую. На практике добиться мобильности довольно трудно, особенно в системах реального времени, в которых одна из задач проектирования языка заключается в максимальном использовании преимуществ базового машинного оборудования.

Суммарная **стоимость** использования языка программирования складывается из нескольких составляющих:

- *стоимость обучения языку* определяется степенью сложности языка;
- *стоимость создания программы* зависит от языка и системы программирования, выбранных для реализации конкретного приложения;
- *стоимость трансляции программы* тесно связана со стоимостью её выполнения. Совокупность методов, используемых транслятором для уменьшения объёма и/или сокращения времени выполнения оттранслированной программы, называется *оптимизацией*. Чем выше степень оптимизации, тем качественнее получается результирующая програм-

ма и сильнее уменьшается время её выполнения, но при этом возрастает стоимость трансляции;
- *стоимость выполнения программы* особенно существенна для программного обеспечения систем реального времени;
- *стоимость сопровождения программы* включает в себя затраты на исправление дефектов и модификацию программы в связи с обновлением аппаратуры или расширением функциональных возможностей.

1.4 Влияние языков программирования на трансляторы

Поскольку разработка языков программирования и разработка трансляторов тесно связаны между собой. Новые достижения в области языков программирования приводят к новым требованиям, возникающим перед разработчиками трансляторов, которые должны придумывать алгоритмы и представления для трансляции и поддержки новых возможностей языка. Кроме того, с 1940-х годов произошли существенные изменения и в архитектуре вычислительных систем, так что разработчики трансляторов должны не только учитывать новые свойства языков программирования, но и разрабатывать такие алгоритмы трансляции, которые смогут максимально использовать преимущества новых аппаратурных возможностей.

Трансляторы могут способствовать использованию высокоуровневых языков программирования, минимизируя накладные расходы времени выполнения программ, написанных на этих языках. Они играют важную роль в эффективном использовании высокопроизводительной архитектуры компьютера пользовательскими приложениями. В действительности производительность вычислительной системы настолько зависит от технологии трансляции, что трансляторы используются в качестве инструмента для оценки архитектурных концепций перед созданием компьютера.

Написание транслятора представляет настоящий вызов для программиста. Транслятор сам по себе – большая и сложная программа. Кроме того, многие современные системы разработки программного обеспечения работают с разными языками и целевыми машинами в пределах одного пакета, т.е. представляют собой набор трансляторов, состоящих, возможно, из миллионов строк кода.

Транслятор обязан корректно транслировать потенциально бесконечное множество программ, которые могут быть написаны на соответствующем языке программирования. Задача генерации оптимального целевого кода из исходной программы в общем случае неразрешима; таким образом, разработчики трансляторов должны искать компромиссные решения того, какие эвристики следует использовать для генерации эффективного кода.

Контрольные вопросы

1. Какие языки называются императивными?
2. Какие языки относят к языкам функционального программирования?
3. Какие языки являются декларативными?

4. Назовите три основных свойства объектно-ориентированных языков программирования.
5. Как влияет удобочитаемость языка программирования на лёгкость создания программ на этом языке?
6. Что понимается под естественностью языка программирования?
7. Из каких составляющих складывается суммарная стоимость языка программирования?
8. Какое свойство языка программирования даёт возможность более просто переносить программы с одной аппаратурной платформы на другую?

ГЛАВА 2
ФОРМАЛЬНЫЕ ЯЗЫКИ И ГРАММАТИКИ

2.1 Основные понятия и определения

Тексты на любом языке (естественном или формальном) представляют собой цепочки символов некоторого алфавита.

Цепочкой символов называют произвольную упорядоченную конечную последовательность символов, записанных один за другим. Понятие **символа** является базовым в теории формальных языков. Для цепочки символов имеют значение три фактора: состав входящих в цепочку символов, их количество и порядок символов в цепочке. Далее цепочки символов будем обозначать греческими буквами: α, β, γ и др.

Цепочки символов α и β **равны** (α = β), если они имеют один и тот же состав символов, одно и то же их количество и одинаковый порядок следования символов в цепочке.

Количество символов в цепочке определяет её **длину**. Длина цепочки α обозначается как | α |.

Можно выделить следующие операции над цепочками символов:

- *конкатенация* (объединение, сложение двух цепочек) – это дописывание второй цепочки в конец первой. Конкатенация цепочек α и β обозначается как αβ. Например: α = *аб*, β = *вг*, тогда αβ = *абвг*. При этом αβ ≠ βα, так как в цепочке важен порядок символов. Но конкатенация обладает свойством ассоциативности: (αβ)γ = α(βγ);
- *замена* (подстановка) – замена подцепочки символов на любую произвольную цепочку символов. В результате получается новая цепочка символов. Например: γ = *абвг*, разобьём эту цепочку символов на подцепочки: α = *а*, ω = *б*, β = *вг* и выполним подстановку цепочки υ = *аба* вместо подцепочки ω. Получим новую цепочку γ' = *ааабавг*. Таким образом, подстановка выполняется путём разбиения исходной цепочки на подцепочки и конкатенации;
- *обращение* – запись символов цепочки в обратном порядке. Эта операция обозначается как α^R. Если α = *абвг*, то α^R = *гвба*. Для операции обращения справедливо следующее: $(\alpha\beta)^R = \beta^R\alpha^R$;
- *итерация* – повторение цепочки *n* раз, где *n* > 0 – это конкатенация цепочки с собой *n* раз, обозначается как α^n. Если *n* = 0, то результатом итерации будет пустая цепочка символов.

Пустая цепочка символов – это цепочка, не содержащая ни одного символа. Будем обозначать такую цепочку как ε. Для пустой цепочки справедливы следующие равенства:

$|\varepsilon| = 0$
$\varepsilon\alpha = \alpha\varepsilon = \alpha$
$\varepsilon^R = \varepsilon$
$\varepsilon^n = \varepsilon,\ n \geq 0$

$\alpha^0 = \varepsilon$

Основой любого языка является алфавит, определяющий набор допустимых символов языка.

Алфавит – это счётное множество допустимых символов языка. Будем обозначать это множество как **V.** Цепочка символов α является **цепочкой над алфавитом V**: α(**V**), если в неё входят только символы, принадлежащие множеству символов **V**. Для любого алфавита **V** пустая цепочка может как являться, так и не являться цепочкой над алфавитом.

Если **V** – некоторый алфавит, то:

- $\mathbf{V}^+$ – множество всех цепочек над алфавитом **V** без пустой цепочки;
- **V*** – множество всех цепочек над алфавитом **V**, включая пустую цепочку.

Языком L над алфавитом **V** (**L (V**)) называется некоторое счётное подмножество цепочек конечной длины из множества всех цепочек над алфавитом **V**. Множество цепочек языка не обязано быть конечным; хотя каждая цепочка символов, входящая в язык, обязана иметь конечную длину, эта длина может быть сколь угодно большой и формально ничем не ограничена.

Цепочку символов, принадлежащую заданному языку, часто называют **предложением** языка, а множество цепочек символов некоторого языка **L (V)** – множеством предложений этого языка.

Кроме алфавита язык предусматривает также правила построения допустимых цепочек, поскольку обычно далеко не все цепочки над заданным алфавитом принадлежат языку. Символы могут объединяться в слова или лексемы – элементарные конструкции языка, на их основе строятся предложения – более сложные конструкции. И те и другие в общем виде являются цепочками символов, но предусматривают некоторые правила построения. Таким образом, необходимо указать эти правила, или, строго говоря, задать язык.

В общем случае язык можно определить тремя способами:

1) перечислением всех допустимых цепочек языка;
2) указанием способа порождения цепочек языка (заданием грамматики языка);
3) определением метода распознавания цепочек языка.

Первый из методов является чисто формальным и на практике не применяется, так как большинство языков содержат бесконечное число допустимых цепочек и перечислить их просто невозможно. Иногда для чисто формальных языков можно перечислить множество входящих в них цепочек, прибегнув к математическим определениям множеств. Однако этот подход уже стоит ближе ко второму способу. Например, запись:

$$\mathbf{L}\,(\{0, 1\}) = \{0^n 1^n,\ n > 0\}$$

задаёт язык над алфавитом **V** = {0, 1}, содержащий все последовательности с чередующимися символами 0 и 1, начинающиеся с 0 и заканчивающиеся 1. Видно, что пустая цепочка символов в этот язык не входит.

Второй способ предусматривает некоторое описание правил, с помощью которых строятся цепочки языка. Тогда любая цепочка, построенная с помощью этих правил из символов алфавита языка, будет принадлежать заданному языку.

Третий способ предусматривает построение некоторого логического устройства (распознавателя) – автомата, который на входе получает цепочку символов, а на выходе выдаёт ответ, принадлежит или нет эта цепочка заданному языку.

Грамматика – это описание способа построения предложений некоторого языка. Она относится ко второму способу определения языков – порождению цепочек символов.

Грамматику языка можно описать различными способами. Например, можно использовать формальное описание грамматики, построенное на основе системы правил (или продукций).

Правило (или продукция) – это упорядоченная пара цепочек символов (α, β). В правилах важен порядок цепочек, поэтому их чаще записывают в виде α → β (или α ::= β). Такая запись читается как «α порождает β» или «α по определению есть β».

Грамматика языка программирования содержит правила двух типов: первые (определяющие синтаксические конструкции языка) довольно легко поддаются формальному описанию; вторые (определяющие семантические ограничения языка) обычно излагаются в неформальной форме. Поэтому любое описание (или стандарт) языка программирования обычно состоит из двух частей: вначале формально излагаются правила построения синтаксических конструкций, а потом на естественном языке даётся описание семантических правил.

Язык, заданный грамматикой **G**, обозначается как **L (G)**. Две грамматики, **G** и **G'**, называются **эквивалентными**, если они определяют один и тот же язык: **L (G)** = **L (G')**. Две грамматики, **G** и **G'**, называются **почти эквивалентными**, если заданные ими языки различаются не более чем на пустую цепочку символов: **L (G)** ∪ {ε} = **L (G')** ∪ {ε}.

Формально грамматика **G** определяется как четвёрка **G** (**VT**, **VN**, **P**, *S*), где:

- **VT** – множество терминальных символов, или алфавит терминальных символов;
- **VN** – множество нетерминальных символов, или алфавит нетерминальных символов;
- **P** – множество правил (продукций) грамматики вида $\alpha \to \beta$, где $\alpha \in (\mathbf{VN} \cup \mathbf{VT})^+$, $\beta \in (\mathbf{VN} \cup \mathbf{VT})^*$;
- *S* – целевой (начальный) символ грамматики, $S \in \mathbf{VN}$.

Алфавиты терминальных и нетерминальных символов грамматики не пересекаются: $\mathbf{VN} \cap \mathbf{VT} = \varnothing$. Это значит, что каждый символ в грамматике может быть либо терминальным, либо нетерминальным, но не может быть терминальным и нетерминальным одновременно. Целевой символ грамматики – это все-

гда нетерминальный символ. Множество **V** = **VN** ∪ **VT** называют **полным алфавитом** грамматики **G**.

Множество терминальных символов VT содержит символы, которые входят в алфавит языка, порождаемого грамматикой. Как правило, символы из множества **VT** встречаются только в цепочках правых частей правил.

Множество нетерминальных символов VN содержит символы, которые определяют слова, понятия, конструкции языка. Каждый символ этого множества может встречаться в цепочках как левой, так и правой частей правил грамматики.

Во множестве правил грамматики может быть несколько правил, имеющих одинаковые левые части вида: $\alpha \rightarrow \beta_1$, $\alpha \rightarrow \beta_2$, …, $\alpha \rightarrow \beta_n$. Эти правила можно объединить вместе и записать в виде: $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$. Одной строке в такой записи соответствует сразу *n* правил.

Такую форму записи правил грамматики называют **формой Бэкуса-Наура**[3]. Форма Бэкуса-Наура (англ. Backus-Naur Form (BNF)), как правило, предусматривает также, что нетерминальные символы берутся в угловые скобки: < >.

Пример грамматики, которая определяет язык целых десятичных чисел со знаком в форме Бэкуса-Наура:

G ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –, +}, {<число>, <чс>, <цифра>}, **P**, <число>)
P:

<число> → <чс> | +<чс> | –<чс>
<чс> → <цифра> | <чс><цифра>
<цифра> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Рассмотрим составляющие элементы грамматики **G**:

- множество терминальных символов **VT** содержит двенадцать элементов: десять десятичных цифр и два знака;
- множество нетерминальных символов **VN** содержит три элемента: символы <число>, <чс> и <цифра>;
- множество правил содержит 15 правил, которые записаны в три строки (то есть имеется только три различные левые части правил);
- целевым символом грамматики является символ <число>.

Та же самая грамматика для языка целых десятичных чисел со знаком, в которой нетерминальные символы обозначены большими латинскими буквами (далее это будет часто применяться в примерах):

G' ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –, +}, {*S*, *T*, *F*}, **P**, *S*)
P:
$S \rightarrow T \mid +T \mid -T$

[3] Джон Бэкус (John Backus, 1924 –2007) – американский учёный в области информатики, был руководителем команды, разработавшей высокоуровневый язык программирования ФОРТРАН, разработчиком одной из самых универсальных нотаций, используемых для определения синтаксиса формальных языков.

Питер Науp (Peter Naur; 1928) – датский учёный в области информатики, известен как один из разработчиков первого языка структурного программирования Алгол-60 и, совместно с Бэкусом, как изобретатель формы Бэкуса-Наура.

$T \rightarrow F \mid TF$
$F \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Здесь изменилось только множество нетерминальных символов. Теперь **VN** = {*S, T, F*}. Язык, заданный грамматикой, не изменился – грамматики **G** и **G'** эквивалентны.

Особенность рассмотренных выше формальных грамматик в том, что они позволяют определить бесконечное множество цепочек языка с помощью конечного набора правил. Возможность пользоваться конечным набором правил достигается в такой форме записи за счёт **рекурсивных правил.** Рекурсия в правилах грамматики выражается в том, что один из нетерминальных символов определяется сам через себя. Рекурсия может быть *непосредственной* (явной) – символ определяется сам через себя в одном правиле, либо *косвенной* (неявной) – тогда тоже самое происходит через цепочку правил. В рассмотренной выше грамматике **G** непосредственная рекурсия присутствует в правиле <чс> → <чс> <цифра>, а в эквивалентной ей грамматике **G'** – в правиле $T \rightarrow TF$.

Чтобы рекурсия не была бесконечной, для участвующего в ней нетерминального символа грамматики должны существовать также и другие правила, которые определяют его, минуя самого себя, и позволяют избежать бесконечного рекурсивного определения. Такими правилами являются <чс> → <цифра> – в грамматике **G** и $T \rightarrow F$ – в грамматике **G'**.

Более упрощённой по сравнению с БНФ является **расширенная форма Бэкуса-Наура** (англ. Extended Backus-Naur Form (EBNF)). Она отличается более «ёмкими» конструкциями, позволяющими при той же выразительной способности упростить и сократить описание в объёме. Набор возможных конструкций РБНФ невелик. Это конкатенация, выбор, условное вхождение и повторение.

Конкатенация не имеет специального обозначения, определяется последовательной записью символов в выражении. *Выбор* обозначается вертикальной чертой (|), как и в БНФ. Квадратные скобки ([ ]) выделяют необязательный элемент выражения, который может присутствовать, а может и отсутствовать (*условное вхождение*). Фигурные скобки ({}) обозначают конкатенацию любого числа (включая нуль) записанных в них элементов (*повторение*). Помимо основных операций в РБНФ могут использоваться обычные круглые скобки (( )). Они применяются для группировки элементов при формировании сложных выражений.

Следующая грамматика определяет запись десятичного числа общего вида (с ведущим знаком, возможной дробной частью и порядком) в расширенной форме Бэкуса-Наура:

Число = [+ | –] НатЧисло[. [НатЧисло]][(e | E)[+ | –] НатЧисло]
НатЧисло = Цифра{Цифра}
Цифра = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

В дальнейших примерах для описания грамматик будет использоваться обычная форма Бэкуса-Наура.

2.2 Классификация грамматик и языков

Согласно классификации, предложенной американским лингвистом Ноамом Хомским, профессором Массачусетского технологического института, формальные грамматики классифицируются по структуре их правил. Если все без исключения правила грамматики удовлетворяют некоторой заданной структуре, то такую грамматику относят к определённому типу. Достаточно иметь в грамматике одно правило, не удовлетворяющее требованиям структуры правил, и она уже не попадает в заданный тип. По классификации Хомского выделяют четыре типа грамматик.

Тип 0: грамматики с фразовой структурой

На структуру их правил не накладывается никаких ограничений: для грамматики вида **G** (**VT**, **VN**, **P**, *S*), **V** = **VN** ∪ **VT**, правила имеют вид $\alpha \rightarrow \beta$, где $\alpha \in \mathbf{V}^+$, $\beta \in \mathbf{V}^*$.

Это самый общий тип грамматик. В него попадают все без исключения формальные грамматики, но часть из них может быть также отнесена и к другим классификационным типам. Грамматики, которые относятся только к типу 0 и не могут быть отнесены к другим типам, являются самыми сложными по структуре. Практического применения грамматики, относящиеся только к типу 0, не имеют.

Тип 1: контекстно-зависимые (КЗ) и неукорачивающие грамматики

В этот тип входят два основных класса грамматик:

- *Контекстно-зависимые грамматики*
  G (**VT**, **VN**, **P**, *S*), **V** = **VN** ∪ **VT**, имеют правила вида $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$, где $\alpha_1, \alpha_2 \in \mathbf{V}^*$, $A \in \mathbf{VN}$, $\beta \in \mathbf{V}^+$.
- *Неукорачивающие грамматики*
  G (**VT**, **VN**, **P**, *S*), **V** = **VN** ∪ **VT**, имеют правила вида: $\alpha \rightarrow \beta$, где $\alpha, \beta \in \mathbf{V}^+$, $|\beta| \geq |\alpha|$.

Структура правил КЗ-грамматик такова, что при построении предложений заданного ими языка один и тот же нетерминальный символ может быть заменён на ту или иную цепочку символов в зависимости от того контекста, в котором он встречается. Именно поэтому эти грамматики называют *контекстно-зависимыми*. Цепочки α_1 и α_2 в правилах грамматики обозначают *контекст* (α_1 – *левый* контекст, а α_2 – *правый* контекст), в общем случае любая из них (или даже обе) может быть пустой. Говоря иными словами, значение одного и того же символа может быть различным в зависимости от того, в каком контексте он встречается.

Неукорачивающие грамматики имеют такую структуру правил, что при построении предложений языка, заданного грамматикой, любая цепочка символов может быть заменена на цепочку символов не меньшей длины. Отсюда и название *неукорачивающие*.

Доказано, что эти два класса грамматик эквивалентны. Это значит, что для любого языка, заданного КЗ-грамматикой, можно построить неукорачивающую грамматику, которая будет задавать эквивалентный язык, и, наоборот:

для любого языка, заданного неукорачивающей грамматикой, можно построить КЗ-грамматику, которая будет задавать эквивалентный язык.

Тип 2: контекстно-свободные (КС) грамматики

Неукорачивающие контекстно-свободные (НКС) грамматики **G** (**VT**, **VN**, **P**, *S*), **V** = **VN** ∪ **VT**, имеют правила вида $A \to \beta$, где $A \in \mathbf{VN}$, $\beta \in \mathbf{V}^+$. Такие грамматики называют НКС-грамматиками, поскольку видно, что в правой части правил у них должен всегда стоять как минимум один символ.

Существует также почти эквивалентный им класс грамматик – укорачивающие контекстно-свободные (УКС) грамматики **G** (**VT**, **N**, **P**, *S*), **V** = **VN** ∪ **VT**, правила которых могут иметь вид: $A \to \beta$, где $A \in \mathbf{VN}$, $\beta \in \mathbf{V}^*$.

Эти два класса составляют тип контекстно-свободных грамматик. Разница между ними заключается лишь в том, что в УКС-грамматиках в правой части правил может присутствовать пустая цепочка, а в НКС-грамматиках – нет. Отсюда ясно, что язык, заданный НКС-грамматикой, не может содержать пустой цепочки.

Тип 3: регулярные грамматики

К типу регулярных относятся два эквивалентных класса грамматик: леволинейные и праволинейные.

Леволинейные грамматики **G** (**VT**, **VN**, **P**, *S*), **V** = **VN** ∪ **VT**, могут иметь правила двух видов: $A \to B\gamma$ или $A \to \gamma$, где $A, B \in \mathbf{VN}$, $\gamma \in \mathbf{VT}^*$.

В свою очередь, *праволинейные грамматики* **G** (**VT**, **VN**, **P**, *S*), **V** = **VN** ∪ **VT**, могут иметь правила тоже двух видов: $A \to \gamma B$ или $A \to \gamma$, где A, $B \in \mathbf{VN}$, $\gamma \in \mathbf{VT}^*$.

Языки классифицируются в соответствии с типами грамматик, с помощью которых они заданы. Причём, поскольку один и тот же язык в общем случае может быть задан сколь угодно большим количеством грамматик, которые могут относиться к различным классификационным типам, для классификации самого языка среди всех его грамматик выбирается грамматика с максимально возможным классификационным типом. Например, если язык **L** может быть задан грамматиками $\mathbf{G_1}$ и $\mathbf{G_2}$, относящимися к типу 1 (КЗ), грамматикой $\mathbf{G_3}$, относящейся к типу 2 (КС), и грамматикой $\mathbf{G_4}$, относящейся к типу 3 (регулярные), сам язык должен быть отнесён к типу 3 и является регулярным языком.

Например, грамматика типа 0 $\mathbf{G_1}$ ({0, 1}, {*A*, *S*}, $\mathbf{P_1}$, *S*) и КС-грамматика $\mathbf{G_2}$ ({0, 1}, {*S*}, $\mathbf{P_2}$, *S*), где

$\mathbf{P_1}$: $S \to 0A1$ $\qquad\qquad$ $\mathbf{P_2}$: $S \to 0S1 \mid 01$

$0A \to 00A1$

$A \to \varepsilon$

описывают один и тот же язык $\mathbf{L} = \mathbf{L}(\mathbf{G_1}) = \mathbf{L}(\mathbf{G_2}) = \{0^n1^n \mid n > 0\}$. Язык **L** называют КС-языком, т.к. существует КС-грамматика, его описывающая. Но он не является регулярным языком, т.к. не существует регулярной грамматики, описывающей этот язык.

2.3 Цепочки вывода

Выводом называется процесс порождения предложения языка на основе правил определяющей язык грамматики.

Цепочка $\beta = \delta_1\gamma\delta_2$ называется **непосредственно выводимой** из цепочки $\alpha = \delta_1\omega\delta_2$ в грамматике **G (VT, VN, P, *S*)**, $\mathbf{V} = \mathbf{VN} \cup \mathbf{VT}$, $\delta_1, \gamma, \delta_2 \in \mathbf{V}^*$, $\omega \in \mathbf{V}^+$, если в грамматике существует правило $\omega \to \gamma$. Иными словами, цепочка β выводима из цепочки α в том случае, если можно взять несколько символов в цепочке α, поменять их на другие символы, согласно некоторому правилу грамматики, и получить цепочку β. Непосредственная выводимость цепочки β из цепочки α обозначается как: $\alpha \Rightarrow \beta$.

Цепочка β называется **выводимой** из цепочки α ($\alpha \Rightarrow^* \beta$) в случае, если выполняется одно из двух условий:

- β непосредственно выводима из α ($\alpha \Rightarrow \beta$);
- существует γ такая, что γ выводима из α, и β непосредственно выводима из γ ($\alpha \Rightarrow \gamma$, $\gamma \Rightarrow \beta$).

Пример 1. Грамматика для языка целых десятичных чисел со знаком:

G ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –, +}, {*S*, *T*, *F*}, **P**, *S*)
P:
$S \to T \mid +T \mid -T$
$T \to F \mid TF$
$F \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Построим несколько цепочек вывода в этой грамматике (для понимания каждого шага вывода подцепочка, для которой выполняется подстановка, выделена жирным шрифтом):

1) $\boldsymbol{S} \Rightarrow -\boldsymbol{T} \Rightarrow -\boldsymbol{T}F \Rightarrow -\boldsymbol{T}FF \Rightarrow -\boldsymbol{F}FF \Rightarrow -4\boldsymbol{F}F \Rightarrow -47\boldsymbol{F} \Rightarrow -479$
2) $\boldsymbol{S} \Rightarrow \boldsymbol{T} \Rightarrow T\boldsymbol{F} \Rightarrow \boldsymbol{T}8 \Rightarrow \boldsymbol{F}8 \Rightarrow 18$
3) $\boldsymbol{T} \Rightarrow T\boldsymbol{F} \Rightarrow \boldsymbol{T}0 \Rightarrow T\boldsymbol{F}0 \Rightarrow \boldsymbol{T}50 \Rightarrow \boldsymbol{F}50 \Rightarrow 350$
4) $\boldsymbol{F} \Rightarrow 5$

Получаем, что: $S \Rightarrow^* -479$, $S \Rightarrow^* 18$, $T \Rightarrow^* 350$, $F \Rightarrow^* 5$.

Пример 2. Задана грамматика **G** ({*a, b, c*}, {*B, C, D, S*}, **P**, *S*) с правилами:
P:
$S \to BD$
$B \to aBbC \mid ab$
$Cb \to bC$
$CD \to Dc$
$bDc \to bcc$
$abD \to abc$

Эта грамматика задаёт язык **L** (**G**) = $\{a^n b^n c^n \mid n > 0\}$. Пример вывода предложения *aaaabbbbcccc* для этого языка на основе грамматики **G** выглядит следующим образом (для понимания каждого шага вывода подцепочка, для которой выполняется подстановка, выделена жирным шрифтом):

$\boldsymbol{S} \Rightarrow \boldsymbol{B}D \Rightarrow a\boldsymbol{B}bCD \Rightarrow aa\boldsymbol{B}bCbCD \Rightarrow aaa\boldsymbol{B}bCbCbCD \Rightarrow aaaabb\boldsymbol{Cb}CbCD \Rightarrow aaaabbbC\boldsymbol{Cb}CD \Rightarrow aaaabbb\boldsymbol{Cb}CCD \Rightarrow aaaabbbbCC\boldsymbol{CD} \Rightarrow aaaabbbbC\boldsymbol{CD}c \Rightarrow aaaabbbb\boldsymbol{CD}cc \Rightarrow aaaabbb\boldsymbol{bDc}cc \Rightarrow aaaabbbbcccc$

Таким образом, цепочка *aaaabbbbcccc* выводима из *S*: $S \Rightarrow^* aaaabbbbcccc$.

Вывод называется **законченным** (или **конечным**), если на основе цепочки β, полученной в результате этого вывода, нельзя больше сделать ни одного шага вывода. Иначе говоря, вывод называется законченным, если цепочка β, полученная в результате этого вывода, пустая или содержит только терминальные символы грамматики. Цепочка β, полученная в результате законченного вывода, называется **конечной** цепочкой вывода.

В рассмотренном выше примере 1 все построенные выводы являются законченными, вывод $S \Rightarrow^* -4FF$ (из первой цепочки в примере) будет незаконченным.

2.4 Сентенциальная форма грамматики

Цепочка символов $\alpha \in \mathbf{V}^*$ называется **сентенциальной формой** грамматики **G** (**VT**, **VN**, **P**, *S*), $\mathbf{V} = \mathbf{VT} \cup \mathbf{VN}$, если она выводима из целевого символа грамматики *S*: $S \Rightarrow^* \alpha$. Если цепочка $\alpha \in \mathbf{VT}^*$ получена в результате законченного вывода, то она называется **конечной сентенциальной формой.**

Из рассмотренного выше примера можно заключить, что цепочки символов –479 и 18 являются конечными сентенциальными формами грамматики целых десятичных чисел со знаком, так как существуют выводы $S \Rightarrow^* -479$ и $S \Rightarrow^* 18$ (выводы 1 и 2). Цепочка *F*8 из вывода 2, например, тоже является сентенциальной формой, поскольку справедливо $S \Rightarrow^* F8$, но она не является конечной сентенциальной формой вывода. В то же время в выводах 3 и 4 примера 1 явно не присутствуют сентенциальные формы.

2.5 Левосторонний и правосторонний выводы

Вывод называется **левосторонним**, если в нём на каждом шаге вывода правило грамматики применяется всегда к крайнему левому нетерминальному символу в цепочке. Другими словами, если на каждом шаге вывода происходит подстановка цепочки символов на основании правила грамматики вместо крайнего левого нетерминального символа в исходной цепочке.

Аналогично, вывод называется **правосторонним**, если в нём на каждом шаге вывода правило грамматики применяется всегда к крайнему правому нетерминальному символу в цепочке.

Если рассмотреть цепочки вывода из примера 1, то в нём выводы 1 и 4 являются левосторонними, выводы 2, 3 и 4 – правосторонними (вывод 4 является одновременно и правосторонним, и левосторонним).

Для грамматик типов 2 и 3 (КС-грамматик и регулярных грамматик) для любой сентенциальной формы всегда можно построить левосторонний или правосторонний вывод. Для грамматик других типов это не всегда возможно, так как по структуре их правил не всегда можно выполнить замену крайнего левого или крайнего правого нетерминального символа в цепочке.

Рассмотренный в примере 2 вывод $S \Rightarrow^* aaaabbbbcccc$ для грамматики **G**, задающей язык $\mathbf{L(G)} = \{a^n b^n c^n \mid n > 0\}$, не является ни левосторонним, ни правосторонним. Грамматика относится к типу 1, и в данном случае для неё нельзя построить такой вывод, на каждом шаге которого только один нетерминальный символ заменялся бы на цепочку символов.

В грамматике для одной и той же цепочки может быть несколько выводов, эквивалентных в том смысле, что в них в одних и тех же местах применяются одни и те же правила вывода, но в различном порядке.

Например, для цепочки $a+b+a$ в грамматике **G** ($\{a, b, +\}$, $\{S, T\}$, **P**, S) **P**: $S \to T \mid T+S;\ T \to a \mid b$ можно построить выводы:

1) $\boldsymbol{S} \Rightarrow T+\boldsymbol{S} \Rightarrow T+T+\boldsymbol{S} \Rightarrow \boldsymbol{T}+T+T \Rightarrow a+\boldsymbol{T}+T \Rightarrow a+b+\boldsymbol{T} \Rightarrow a+b+a$
2) $\boldsymbol{S} \Rightarrow \boldsymbol{T}+S \Rightarrow a+\boldsymbol{S} \Rightarrow a+\boldsymbol{T}+S \Rightarrow a+b+\boldsymbol{S} \Rightarrow a+b+\boldsymbol{T} \Rightarrow a+b+a$
3) $\boldsymbol{S} \Rightarrow T+\boldsymbol{S} \Rightarrow T+T+\boldsymbol{S} \Rightarrow T+T+\boldsymbol{T} \Rightarrow T+\boldsymbol{T}+a \Rightarrow \boldsymbol{T}+b+a \Rightarrow a+b+a$

Во втором случае применяется левосторонний вывод, в третьем – правосторонний, а первый вывод не является ни левосторонним, ни правосторонним, но все эти выводы являются эквивалентными в указанном выше смысле.

2.6 Дерево вывода

Можно ввести удобное графическое представление вывода, называемое деревом вывода, причём для всех эквивалентных выводов деревья вывода совпадают.

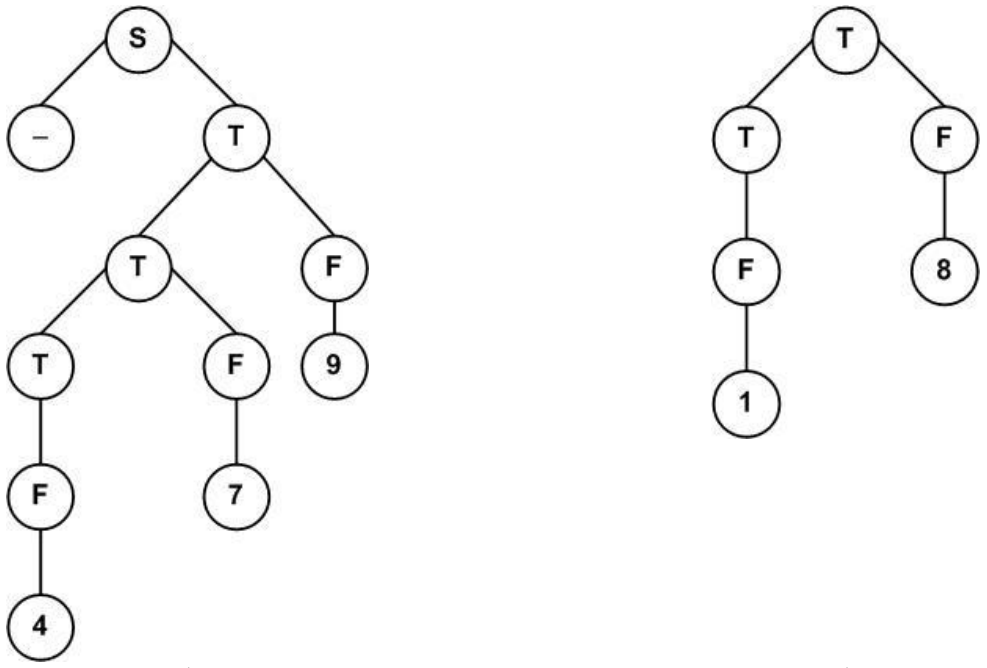
Деревом вывода грамматики **G** (**VT**, **VN**, **P**, S) называется дерево, которое соответствует некоторой цепочке вывода и удовлетворяет следующим условиям:

- каждая вершина дерева обозначается символом грамматики $A \in (\mathbf{VT} \cup \mathbf{VN} \cup \{\varepsilon\})$;
- корнем дерева является вершина, обозначенная целевым символом грамматики – S;
- листьями дерева (концевыми вершинами) являются вершины, обозначенные терминальными символами грамматики или символом пустой цепочки ε;
- если некоторый узел дерева обозначен нетерминальным символом $A \in \mathbf{VN}$, а связанные с ним узлы – символами $b_1, b_2, \ldots, b_n$; $n > 0$,

$\forall i,\ 0 \le i \le n$: $b_i \in$ (**VT** $\cup$ **VN** $\cup$ $\{\varepsilon\}$), то в грамматике **G** (**VT**, **VN**, **P**, S) существует правило $A \to b_1 \mid b_2 \mid \ldots \mid b_n \in$ **P**.

Из определения видно, что по структуре правил дерево вывода в указанном виде всегда можно построить только для грамматик типов 2 и 3 (контекстно-свободных и регулярных). Для грамматик других типов дерево вывода в таком виде можно построить не всегда.

На основе примера 1 из раздела 2.3 построим деревья вывода для цепочек выводов 1 и 2. Эти деревья приведены на рисунке 1 (а – для вывода 1, б – для вывода 2).



а) б)

Рисунок 1. Примеры деревьев вывода для грамматики целых десятичных чисел со знаком

Для того чтобы построить дерево вывода, достаточно иметь только цепочку вывода. Дерево вывода можно построить двумя способами: сверху вниз и снизу вверх. Для строго формализованного построения дерева вывода всегда удобнее пользоваться строго определённым выводом: либо левосторонним, либо правосторонним.

При построении дерева вывода *сверху вниз* построение начинается с целевого символа грамматики, который помещается в корень дерева. Затем в грамматике выбирается необходимое правило, и на первом шаге вывода корневой символ раскрывается на несколько символов первого уровня. На втором шаге среди всех концевых вершин дерева выбирается крайняя (крайняя левая – для левостороннего вывода, крайняя правая – для правостороннего) вершина, обозначенная нетерминальным символом, для этой вершины выбирается нужное правило грамматики, и она раскрывается на несколько вершин следующего уровня. Построение дерева заканчивается, когда все концевые вершины обозначены терминальными символами, в противном случае надо вернуться ко второму шагу и продолжить построение.

Построение дерева вывода *снизу вверх* начинается с листьев дерева. В качестве листьев выбираются терминальные символы конечной цепочки вывода,

которые на первом шаге построения образуют последний уровень (слой) дерева. Построение дерева идёт по слоям. На втором шаге построения в грамматике выбирается правило, правая часть которого соответствует крайним символам в слое дерева (крайним правым символам – при правостороннем выводе и крайним левым – при левостороннем). Выбранные вершины слоя соединяются с новой вершиной, которая выбирается из левой части правила. Новая вершина попадает в слой дерева вместо выбранных вершин. Построение дерева закончено, если достигнута корневая вершина (обозначенная целевым символом), а иначе надо вернуться ко второму шагу и повторить его над полученным слоем дерева.

Если для каждой цепочки символов языка, заданного грамматикой, можно построить единственный левосторонний (и единственный правосторонний) вывод или, что то же самое, построить единственное дерево вывода, то такая грамматика называется **однозначной**. Иначе грамматика называется **неоднозначной.**

Пример 3. Условный оператор, включённый во многие языки программирования, описывается с помощью грамматики с правилами:

$S \rightarrow$ ***if*** *b* ***then*** *S* ***else*** *S* | ***if*** *b* ***then*** *S* | *a*

где *b* – логическое выражение, а *a* – безусловный оператор. Эта грамматика неоднозначна, т.к. в ней возможны два левых вывода цепочки ***if*** *b* ***then if*** *b* ***then*** *a* ***else*** *a*:

1) $\boldsymbol{S} \Rightarrow$ ***if*** *b* ***then S else*** *S* $\Rightarrow$ ***if*** *b* ***then if*** *b* ***then S else*** *S* $\Rightarrow$ ***if*** *b* ***then if*** *b* ***then*** *a* ***else S*** $\Rightarrow$ ***if*** *b* ***then if*** *b* ***then*** *a* ***else*** *a*
2) $\boldsymbol{S} \Rightarrow$ ***if*** *b* ***then S*** $\Rightarrow$ ***if*** *b* ***then if*** *b* ***then S else*** *S* $\Rightarrow$ ***if*** *b* ***then if*** *b* ***then*** *a* ***else S*** $\Rightarrow$ ***if*** *b* ***then if*** *b* ***then*** *a* ***else*** *a*

Наличие двух различных выводов предполагает две различные интерпретации цепочки ***if*** *b* ***then if*** *b* ***then*** *a* ***else*** *a*: первая из них – ***if*** *b* ***then*** ***(if*** *b* ***then*** *a)* ***else*** *a*, а вторая – ***if*** *b* ***then*** ***(if*** *b* ***then*** *a* ***else*** *a)*. При описании языка программирования эту неоднозначность преодолевают, добавляя к семантическим правилам правило вида: «ключевое слово ***else*** ассоциируется с ближайшим слева ключевым словом ***if***».

Для КС-грамматик существуют определённого вида правила, по наличию которых во всём множестве правил грамматики можно утверждать, что она является неоднозначной:

1) $A \rightarrow AA \mid \alpha$
2) $A \rightarrow A\alpha A \mid \beta$
3) $A \rightarrow \alpha A \mid A\beta \mid \gamma$
4) $A \rightarrow \alpha A \mid \alpha A \beta A \mid \gamma$

Здесь $A \in \mathbf{VN}$; $\alpha, \beta, \gamma \in (\mathbf{VN} \cup \mathbf{VT})^*$.

Если в заданной грамматике встречается хотя бы одно правило подобного вида, то такая грамматика точно будет неоднозначной. Однако отсутствие подобных правил во всём множестве правил грамматики не означает, что данная грамматика является однозначной.

Для грамматики из примера 3 можно показать, что она является неоднозначной, поскольку она содержит правила четвёртого вида.

2.7 Преобразование грамматик

В некоторых случаях КС-грамматика может содержать недостижимые и бесплодные символы, которые не участвуют в порождении цепочек языка и поэтому могут быть удалены из грамматики.

Символ $A \in \mathbf{VN}$ называется *бесплодным* в грамматике **G** (**VT**, **VN**, **P**, *S*), если множество $\{\alpha \in \mathbf{VT}^* \mid A \Rightarrow \alpha\}$ пусто.

Алгоритм удаления бесплодных символов:

Вход: КС-грамматика **G** (**VT**, **VN**, **P**, *S*).
Выход: КС-грамматика **G′** (**VT**, **VN′**, **P′**, *S*), не содержащая бесплодных символов, для которой **L (G)** = **L (G′)**.
Метод:
Рекурсивно строим множества $\mathbf{N_0}, \mathbf{N_1}, ..., \mathbf{N_m}$
1. $\mathbf{N_0} = \varnothing$, $i = 1$.
2. $\mathbf{N_i} = \{A \mid (A \to \alpha) \in \mathbf{P} \text{ и } \alpha \in (\mathbf{N_{i-1}} \cup \mathbf{VT})^*\} \cup \mathbf{N_{i-1}}$.
3. Если $\mathbf{N_i} \neq \mathbf{N_{i-1}}$, то $i = i + 1$ и переходим к шагу 2, иначе $\mathbf{VN'} = \mathbf{N_i}$; **P′** состоит из правил множества **P**, содержащих только символы из $\mathbf{VN'} \cup \mathbf{VT}$; **G′** (**VT**, **VN′**, **P′**, *S*).

Пример 4.
Дана грамматика **G** ({*a, b, c*}, {*A, B, C, D, E, F, G, S*}, **P**, *S*) с правилами **P**:
$S \to aAB \mid E$
$A \to aA \mid bB$
$B \to ACb \mid b$
$C \to A \mid bA \mid cC \mid aE$
$E \to cE \mid aE \mid Eb \mid ED \mid FG$
$D \to a \mid c \mid Fb$
$F \to BC \mid EC \mid AC$
$G \to Ga \mid Gb$
Удаление бесплодных символов:
1. $\mathbf{N_0} = \varnothing$, $i = 1$.
2. Выбираем правила, в которых в правой части находятся терминальные символы: $B \to b, D \to a \mid c$.
Нетерминалы *B* и *D* формируют множество $\mathbf{N_1}$: $\mathbf{N_1} = \{B, D\}$, $\mathbf{N_1} \neq \mathbf{N_0}$, $i = 2$.
3. На следующем шаге ищем правила, правая часть которых содержит нетерминалы, найденные на предыдущем шаге: $A \to bB$. Получаем множество $\mathbf{N_2}$: $\mathbf{N_2} = \{B, D, A\}$, $\mathbf{N_2} \neq \mathbf{N_1}$, $i = 3$.
Аналогично формируем множества $\mathbf{N_3}$, $\mathbf{N_4}$ и $\mathbf{N_5}$:
4. $\mathbf{N_3} = \{B, D, A, S, C\}$, $\mathbf{N_3} \neq \mathbf{N_2}$, $i = 4$
5. $\mathbf{N_4} = \{B, D, A, S, C, F\}$, $\mathbf{N_4} \neq \mathbf{N_3}$, $i = 5$
6. $\mathbf{N_5} = \{B, D, A, S, C, F\}$, $\mathbf{N_5} = \mathbf{N_4}$
Нетерминальные символы, не вошедшие во множество $\mathbf{N_5}$, и являются бесплодными.

Результирующая грамматика имеет вид:

G′ ({*a, b, c*}, {*A, B, C, D, F, S*}, **P′**, *S*)

P′: $S \to aAB$

$A \to aA \mid bB$

$B \to ACb \mid b$

$C \to A \mid bA \mid cC$

$D \to a \mid c \mid Fb$

$F \to BC \mid AC$

Символ $x \in (\mathbf{VT} \cup \mathbf{VN})$ называется *недостижимым* в грамматике **G** (**VT**, **VN**, **P**, *S*), если он не появляется ни в одной сентенциальной форме этой грамматики.

Алгоритм удаления недостижимых символов:

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

Выход: КС-грамматика **G′** (**VT′**, **VN′**, **P′**, *S*), не содержащая недостижимых символов, для которой **L (G)** = **L (G′)**.

Метод:

1. $\mathbf{V_0} = \{S\}$; $i = 1$.

2. $\mathbf{V_i} = \{x \mid x \in (\mathbf{VT} \cup \mathbf{VN})$, в **P** есть $A \to \alpha x \beta$ и $A \in \mathbf{V_{i-1}}$, $\alpha, \beta \in (\mathbf{VT} \cup \mathbf{VN})^*\} \cup \mathbf{V_{i-1}}$.

3. Если $\mathbf{V_i} \neq \mathbf{V_{i-1}}$, то $i = i + 1$ и переходим к шагу 2, иначе $\mathbf{VN'} = \mathbf{V_i} \cap \mathbf{VN}$; $\mathbf{VT'} = \mathbf{V_i} \cap \mathbf{VT}$; **P′** состоит из правил множества **P**, содержащих только символы из $\mathbf{V_i}$; **G′** (**VT′**, **VN′**, **P′**, *S*).

Пример 5.

В качестве входной рассмотрим грамматику, из которой были удалены бесплодные символы в предыдущем примере:

G′ ({*a, b, c*}, {*A, B, C, D, F, S*}, **P′**, *S*)

P': $S \to aAB$

$A \to aA \mid bB$

$B \to ACb \mid b$

$C \to A \mid bA \mid cC$

$D \to a \mid c \mid Fb$

$F \to BC \mid AC$

Удаление недостижимых символов:

1. $\mathbf{V_0} = \{S\}$, $i = 1$.

2. Среди правил грамматики находим правила, содержащие *S* в левой части: $S \to aAB$. Нетерминалы *A* и *B* добавляются во множество $\mathbf{V_1}$:

$\mathbf{V_1} = \{S, A, B\}$, $\mathbf{V_1} \neq \mathbf{V_0}$, $i = 2$.

3. Теперь находим правила с нетерминалами *A* и *B* в левой части: $A \to aA \mid bB$, $B \to ACb \mid b$. Нетерминал *C* добавляется во множество $\mathbf{V_3}$.

$\mathbf{V_2} = \{S, A, B, C\}$, $\mathbf{V_2} \neq \mathbf{V_1}$, $i = 3$.

4. Правила для *C* не добавляют новых нетерминалов во множество $\mathbf{V_3}$:

$\mathbf{V_3} = \{S, A, B, C\}$, $\mathbf{V_3} = \mathbf{V_2}$.

В результирующее множество не вошли символы D и F. Следовательно, они являются недостижимыми.

Результирующая грамматика:

$\mathbf{G''}$ $(\{a, b, c\}, \{A, B, C, S\}, \mathbf{P''}, S)$

$\mathbf{P''}$: $S \to aAB$

$A \to aA \mid bB$

$B \to ACb \mid b$

$C \to A \mid bA \mid cC$

ε-правила – все правила вида $A \to \varepsilon$, где $A \in \mathbf{VN}$. Грамматика без ε-правил – это грамматика, в которой не существует правил $A \to \varepsilon$, где $A \neq S$, и существует только одно правило $S \to \varepsilon$, при этом S не встречается в правой части ни одного правила.

Алгоритм устранения ε-правил:

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

Выход: КС-грамматика **G′** (**VT**, **VN′**, **P′**, S), не содержащая ε-правил, для которой **L (G)** = **L (G′)**.

Метод:

1. $\mathbf{W_0} = \{A: (A \to \varepsilon) \in \mathbf{P}\}$, $i = 1$.
2. $\mathbf{W_i} = \mathbf{W_{i-1}} \cup \{A: (A \to \alpha) \in \mathbf{P}, \alpha \in \mathbf{W_{i-1}}\}^*$.
3. Если $\mathbf{W_i} \neq \mathbf{W_{i-1}}$, $i = i + 1$, то перейти к шагу 2, иначе перейти к шагу 4.
4. **VN′** = **VN**, **VT′** = **VT**, в **P′** входят все правила из **P**, кроме правил вида $A \to \varepsilon$.
5. Если $A \to \alpha_0 B_1 \alpha_1 B_2 \ldots B_k \alpha_k \in \mathbf{P}$, где $k \geq 0$, $B_j \in \mathbf{W_i}$ и ни один из символов цепочек α_i не содержит символов из $\mathbf{W_i}$, то включить в **P′** все правила вида $A \to \alpha_0 X_1 \alpha_1 X_2 \ldots X_k \alpha_k$, где $X_j = B_j$ или $X = \varepsilon$.
6. Если $S \in \mathbf{W_i}$, то значит $\varepsilon \in \mathbf{L}$ (**G**), и тогда в **VN′** добавляется новый символ S', который становится целевым символом грамматики, а в **P′** добавляются правила: $S' \to \varepsilon \mid S$.

Пример 6.

Дана грамматика **G** $(\{a, b, c\}, \{A, B, C, S\}, \mathbf{P}, S)$ с правилами **P**:

$S \to AaB \mid aB \mid cC$

$A \to AB \mid a \mid b \mid B$

$B \to Ba \mid \varepsilon$

$C \to AB \mid c$

Удаление ε-правил:

1. Находим ε-правила в данной грамматике: $B \to \varepsilon$

$\mathbf{W_0} = \{B\}$, $i = 1$

2. Теперь ищем правила, содержащие нетерминал B в правой части: $A \to B$

$\mathbf{W_1} = \{B, A\}$, $\mathbf{W_1} \neq \mathbf{W_0}$, $i = 2$

3. Аналогично ищем правила с A и B в правой части: $C \to AB$

$\mathbf{W_2} = \{B, A, C\}$, $\mathbf{W_2} \neq \mathbf{W_1}$, $i = 3$

4. Правил, содержащих A, B и C в правой части, нет.

$\mathbf{W_3} = \{B, A, C\}$, $\mathbf{W_3} = \mathbf{W_2}$

5. Формируем правила приведенной грамматики $\mathbf{G}'$, исключая правило $B \to \varepsilon$.

$\mathbf{VN}' = \{A, B, C, S\}$, $\mathbf{VT}' = \{a, b, c\}$

$\mathbf{P}'$: $S \to AaB \mid aB \mid cC$

$A \to AB \mid a \mid b \mid B$

$B \to Ba$

$C \to AB \mid c$

6. Рассмотрим правила, которые включают себя нетерминалы из множества $\mathbf{W_3}$. Например, для правила $S \to AaB$ нужно включить во множество правил следующие: $S \to AaB \mid Aa\varepsilon \mid \varepsilon aB$ или, что то же самое, $S \to AaB \mid Aa \mid aB$.

$S \to AaB \mid aB \mid cC \Rightarrow S \to AaB \mid aB \mid cC \mid Aa \mid a \mid c$

$B \to Ba \Rightarrow B \to Ba \mid a$

$C \to AB \mid c \Rightarrow C \to AB \mid A \mid B \mid c$

Получаем грамматику без ε-правил, эквивалентную $\mathbf{G}$:

$\mathbf{G}'$ ($\{a, b, c\}$, $\{A, B, C, S\}$, $\mathbf{P}'$, S)

$\mathbf{P}'$: $S \to AaB \mid aB \mid cC \mid Aa \mid a \mid c$

$A \to AB \mid a \mid b \mid B$

$B \to Ba \mid a$

$C \to AB \mid A \mid B \mid c$

Вывод вида $A \Rightarrow^* A$, где $A \in \mathbf{VN}$ называется *циклом*. Циклы возможны в случае, если существуют цепные правила вида $A \to B$, где $A, B \in \mathbf{VN}$.

Алгоритм устранения цепных правил:

Для всех нетерминальных символов повторять шаги 2-4, затем перейти к 5.

1. $\mathbf{N}^X_0 = \{X\}$, $i = 1$.

2. $\mathbf{N}^X_i = \mathbf{N}^X_{i-1} \cup \{B: (A \to B) \in \mathbf{P}, A \in \mathbf{N}^X_{i-1}\}$.

Если $\mathbf{N}^X_i \neq \mathbf{N}^X_{i-1}$, то $i = i + 1$ и перейти к 3, иначе $\mathbf{N}^X = \mathbf{N}^X_i - \{X\}$ и продолжить цикл по шагу 1.

$\mathbf{VN}' = \mathbf{VN}$, $\mathbf{VT}' = \mathbf{VT}$, в $\mathbf{P}'$ входят все правила из $\mathbf{P}$, кроме правил вида $A \to B$, $S' = S$.

Для всех правил $(B \to \alpha) \in \mathbf{P}'$, если $B \in \mathbf{N}^A$, $B \neq A$, то в $\mathbf{P}'$ добавляются правила вида $A \to \alpha$.

Пример 7.

Дана грамматика $\mathbf{G}$ ($\{+, –, /, *, a, b, (,)\}$, $\{S, T, E\}$, $\mathbf{P}$, S) с правилами $\mathbf{P}$:

$S \to S + T \mid S – T \mid T$

$T \to T * E \mid T / E \mid E$

$E \to (S) \mid a \mid b$

Удаление цепных правил:

1. Для символа S:
$\mathbf{N}^{\mathbf{S}}_{\mathbf{0}} = \{S\}$, $i = 1$
К символу S добавляется нетерминал T, благодаря правилу $S \to T$
$\mathbf{N}^{\mathbf{S}}_{\mathbf{1}} = \{S, T\}$, $\mathbf{N}^{\mathbf{S}}_{\mathbf{0}} \neq \mathbf{N}^{\mathbf{S}}_{\mathbf{1}}$, $i = 2$
Во множество добавляется нетерминал E, благодаря правилу $T \to E$
$\mathbf{N}^{\mathbf{S}}_{\mathbf{2}} = \{S, T, E\}$, $\mathbf{N}^{\mathbf{S}}_{\mathbf{2}} \neq \mathbf{N}^{\mathbf{S}}_{\mathbf{1}}$, $i = 3$
$\mathbf{N}^{\mathbf{S}}_{\mathbf{3}} = \{S, T, E\}$, $\mathbf{N}^{\mathbf{S}}_{\mathbf{3}} = \mathbf{N}^{\mathbf{S}}_{\mathbf{2}}$
Результирующее множество:
$\mathbf{N}^{\mathbf{S}} = \{T, E\}$
2. Для символа T:
$\mathbf{N}^{\mathbf{T}}_{\mathbf{0}} = \{T\}$, $i = 1$
Во множество добавляется нетерминал E, благодаря правилу $T \to E$
$\mathbf{N}^{\mathbf{T}}_{\mathbf{1}} = \{T, E\}$, $\mathbf{N}^{\mathbf{T}}_{\mathbf{1}} \neq \mathbf{N}^{\mathbf{T}}_{\mathbf{0}}$, $i = 2$
$\mathbf{N}^{\mathbf{T}}_{\mathbf{2}} = \{T, E\}$, $\mathbf{N}^{\mathbf{T}}_{\mathbf{2}} = \mathbf{N}^{\mathbf{T}}_{\mathbf{1}}$
Результирующее множество:
$\mathbf{N}^{\mathbf{T}} = \{E\}$
Для символа E:
$\mathbf{N}^{\mathbf{E}}_{\mathbf{0}} = \{E\}$, $i = 1$
$\mathbf{N}^{\mathbf{E}}_{\mathbf{1}} = \{E\}$, $\mathbf{N}^{\mathbf{E}}_{\mathbf{1}} = \mathbf{N}^{\mathbf{E}}_{\mathbf{0}}$
Результирующее множество:
$\mathbf{N}^{\mathbf{E}} = \varnothing$
Результирующая грамматика получается следующей подстановкой:
1) Вместо правила $S \to T$ вставляются правила $S \to T * E \mid T / E$
2) Вместо правила $T \to E$ вставляются правила $T \to (S) \mid a \mid b$
$\mathbf{G'}$ ({+, –, /, *, a, b, (,)}, {S, T, E}, $\mathbf{P'}$, S)
$\mathbf{P'}$: $S \to S + T \mid S - T \mid T * E \mid T / E \mid (S) \mid a \mid b$
$T \to T * E \mid T / E \mid (S) \mid a \mid b$
$E \to (S) \mid a \mid b$

КС-грамматика **G** называется *приведённой*, если в ней нет недостижимых символов, циклов и правил с пустыми цепочками.

Алгоритм приведения грамматики:

1) обнаруживаются и удаляются все бесплодные нетерминалы;
2) обнаруживаются и удаляются все недостижимые символы;
3) удаляются правила с пустыми цепочками (ε-правила);
4) удаляются цепные правила.

Удаление символов сопровождается удалением правил вывода, содержащих эти символы.

Замечание: шаги преобразования должны выполняться в указанном порядке!

Нетерминал A называется *рекурсивным*, если для него существует цепочка вывода вида $A \Rightarrow^+ \alpha A \beta$, где $\alpha, \beta \in (\mathbf{VT} \cup \mathbf{VN})^*$. Если $\alpha = \varepsilon$ и $\beta \neq \varepsilon$, то рекурсия называется *левой*.

Алгоритм устранения левой рекурсии:

Вход: КС-грамматика **G** (**VT**, **VN**, **P**, S)
Выход: КС-грамматика **G′** (**VT′**, **VN′**, **P′**, S), не содержащая левой рекурсии, для которой **L** (**G**) = **L** (**G′**).
Метод:
1. **VN** = $\{A_1, A_2, \ldots, A_n\}$, $i = 1$
2. Правила для A_i.
Если в правилах нет левой рекурсии, то они записываются в **P′** без изменений, а A_i добавляется в **VN′**.
Иначе правила для A_i записываются в виде $A_i \to A_i\alpha_1 \mid A_i\alpha_2 \mid \ldots \mid A_i\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_p$, где ни одна из цепочек β_j не начинается с A_k таких, что $k \le i$.
Вместо этих правил в **P′** записываются правила вида:
$A_i \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_p \mid \beta_1 A_i' \mid \beta_2 A_i' \mid \ldots \mid \beta_p A_i'$
$A_i' \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m \mid \alpha_1 A_i' \mid \alpha 2 A_i' \mid \ldots \mid \alpha_m A_i'$
A_i и A_i' включаются в **VN′**.
3. Если $i = n$, то **G′** построена, перейти к 6, иначе $i = i + 1$, $j = 1$, перейти к 4.
4. Для A_i в **P′** заменить все правила вида $A_i \to A_j\alpha$, где $\alpha \in (\mathbf{VT} \cup \mathbf{VN})^*$, на правила вида $A_i \to \beta_1\alpha \mid \beta_2\alpha \mid \ldots \mid \beta_m\alpha$, причём $A_j \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$ – все правила для A_j.
5. Если $j = i - 1$, перейти к 2, иначе $j = j + 1$ и перейти к 4.
6. Целевым символом грамматики **G′** становится A_k, соответствующий символу S исходной грамматики.

Пример 8:
Рассмотрим грамматику из предыдущего примера:
G ({+, –, /, *, *a*, *b*, (,)}, {*S*, *T*, *E*}, **P**, *S*)
P: $S \to S + T \mid S - T \mid T$
$T \to T * E \mid T / E \mid E$
$E \to (S) \mid a \mid b$
1. Вводим новые нетерминальные символы вместо S, T и E:
VN' = $\{A_1, A_2, A_3\}$, $i = 1$
Переписываем правила грамматики с учётом введённых нетерминалов:
$A_1 \to A_1 + A_2 \mid A_1 - A_2 \mid A_2$
$A_2 \to A_2 * A_3 \mid A_2 / A_3 \mid A_3$
$A_3 \to (A_1) \mid a \mid b$
2. Правило для A_1 содержит левую рекурсию, поэтому перепишем его в соответствии с шагом 2 алгоритма:
$A_1 \to A_1 + A_2 \mid A_1 - A_2 \mid A_2$
$A_1 \to A_1\alpha_1 \mid A_2\alpha_2 \mid \beta_1$

$\alpha_1 = +A_2$, $\alpha_2 = -A_2$, $\beta_1 = A_2$

Во множество правил **P'** вносятся правила:

$A_1 \to A_2 \mid A_2A_1'$

$A_1' \to + A_2 \mid - A_2 \mid + A_2A_1' \mid - A_2A_1'$

3. Аналогично поступаем с правилами для A_2

$A_2 \to A_2 * A_3 \mid A_2 / A_3 \mid A_3$

$A_2 \to A_2\alpha_1 \mid A_2\alpha_2 \mid \beta_1$

$\alpha_1 = *A_3$, $\alpha_2 = / A_3$, $\beta_1 = A_3$

$A_2 \to A_3 \mid A_3A_2'$

$A_2' \to *A_3 \mid / A_3 \mid *A_3A_2' \mid / A_3A_2'$

Правило для A_3 не содержит левой рекурсии. Оно входит во множество **P′** без изменений:

$A_3 \to (A_1) \mid a \mid b$

Результирующая грамматика имеет вид:

G′ $(\{+, -, /, *, a, b, (,)\}, \{A_1, A_1', A_2, A_2', A_3\}, \mathbf{P}', A_1)$

P′: $A_1 \to A_2 \mid A_2A_1'$

$A_1' \to + A_2 \mid - A_2 \mid + A_2A_1' \mid - A_2A_1'$

$A_2 \to A_3 \mid A_3A_2'$

$A_2' \to *A_3 \mid / A_3 \mid *A_3A_2' \mid / A_3A_2'$

$A_3 \to (A_1) \mid a \mid b$

Для описания синтаксиса языков программирования стараются использовать однозначные приведённые КС-грамматики.

Контрольные вопросы

1. Какие операции можно выполнять над цепочками символов?
2. Какие существуют методы задания языков?
3. Что такое грамматика языка?
4. Как выглядит описание грамматики в форме Бэкуса-Наура? Какие ещё формы описания грамматик существуют?
5. На основе какого принципа классифицируются грамматики в классификации Н. Хомского?
6. Какие типы грамматик выделяют по классификации Н. Хомского?
7. Какие типы языков выделяют по классификации Н. Хомского? Как классификация языков соотносится с классификацией грамматик?
8. Что такое сентенциальная форма грамматики?
9. Что такое левосторонний и правосторонний выводы?
10. Для чего необходимо выполнять приведение грамматик?

ГЛАВА 3
РЕГУЛЯРНЫЕ ГРАММАТИКИ И КОНЕЧНЫЕ АВТОМАТЫ

Практически во всех трансляторах (и в компиляторах, и в интерпретаторах) в том или ином виде присутствует фаза лексического анализа. В основе лексических анализаторов лежат регулярные грамматики, поэтому рассмотрим грамматики этого класса более подробно. В дальнейшем под регулярной грамматикой будем понимать леволинейную грамматику.

Для грамматик этого типа существует алгоритм определения того, принадлежит ли анализируемая цепочка языку, порождаемому этой грамматикой (*алгоритм разбора*):

1) первый символ исходной цепочки $a_1a_2...a_n\perp$ заменяем нетерминалом A, для которого в грамматике есть правило вывода $A \to a_1$ (другими словами, производим «свёртку» терминала a_1 к нетерминалу A);
2) затем многократно (до тех пор, пока не считаем признак конца цепочки) выполняем следующие шаги: полученный на предыдущем шаге нетерминал A и расположенный непосредственно справа от него очередной терминал a_i исходной цепочки заменяем нетерминалом B, для которого в грамматике есть правило вывода $B \to Aa_i$, $i = 2, 3, ..., n$.

Это эквивалентно построению дерева разбора восходящим методом: на каждом шаге алгоритма строим один из уровней в дереве разбора, поднимаясь от листьев к корню.

При работе этого алгоритма возможны следующие ситуации:

1) прочитана вся цепочка; на каждом шаге находилась единственная нужная «свёртка»; на последнем шаге «свёртка» произошла к символу S. Это означает, что исходная цепочка $a_1a_2...a_n\perp \in \mathbf{L(G)}$;
2) прочитана вся цепочка; на каждом шаге находилась единственная нужная «свёртка»; на последнем шаге «свёртка» произошла к символу, отличному от S. Это означает, что исходная цепочка $a_1a_2...a_n\perp \notin \mathbf{L(G)}$;
3) на некотором шаге не нашлось нужной «свёртки», т.е. для полученного на предыдущем шаге нетерминала A и расположенного непосредственно справа от него очередного терминала a_i исходной цепочки не нашлось нетерминала B, для которого в грамматике было бы правило вывода $B \to Aa_i$. Это означает, что исходная цепочка $a_1a_2...a_n\perp \notin \mathbf{L(G)}$;
4) на некотором шаге работы алгоритма оказалось, что есть более одной подходящей «свёртки», т.е. в грамматике разные нетерминалы имеют правила вывода с одинаковыми правыми частями, и поэтому непонятно, к какому из них производить «свёртку». Это говорит о *недетерминированности разбора.* Анализ этой ситуации будет дан ниже.

Допустим, что разбор на каждом шаге детерминированный. Для того чтобы быстрее находить правило с подходящей правой частью, зафиксируем все возможные «свёртки». Это можно сделать в виде таблицы, строки которой помечены нетерминальными символами грамматики, столбцы – терминальными. Значение каждого элемента таблицы – это нетерминальный символ, к которому

можно свернуть пару «нетерминал-терминал», которыми помечены соответствующие строка и столбец.

Пример 9. Для грамматики **G** ({*a*, *b*, ⊥}, {*S*, *A*, *B*, *C*}, **P**, *S*) такая таблица будет выглядеть следующим образом:

P: $S \to C\perp$
$C \to Ab \mid Ba$
$A \to a \mid Ca$
$B \to b \mid Cb$

	a	***b***	⊥
C	*A*	*B*	*S*
A	–	*C*	–
B	*C*	–	–
S	–	–	–

Знак «–» ставится в том случае, если для пары «терминал-нетерминал» «свёртки» нет.

Но чаще информацию о возможных «свёртках» представляют в виде **диаграммы состояний** (ДС) – неупорядоченного ориентированного помеченного графа, который строится следующим образом:

1) строятся вершины графа, помеченные нетерминалами грамматики (для каждого нетерминала – одну вершину), и ещё одну вершину, помеченную символом, отличным от нетерминальных (например, *H*). Эти вершины называют *состояниями*. *H – начальное состояние*;
2) соединяем эти состояния дугами по следующим правилам:
   а) для каждого правила грамматики вида $W \to t$ ($W \in \mathbf{VN}^+$, $t \in \mathbf{VT}^+$) соединяем дугой состояния *H* и *W* (от *H* к *W*) и помечаем дугу символом *t*;
   б) для каждого правила $W \to Vt$ (*W*, $V \in \mathbf{VN}^+$, $t \in \mathbf{VT}^+$) соединяем дугой состояния *V* и *W* (от *V* к *W*) и помечаем дугу символом *t*.

Диаграмма состояний для грамматики **G** из примера 9 изображена на рисунке 2.

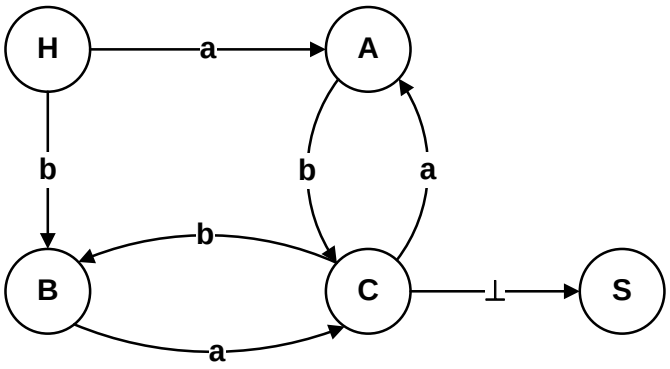


Рисунок 2. Диаграмма состояний для грамматики из примера 9.

Алгоритм разбора по диаграмме состояний:

1) объявляем текущим состояние *H*;
2) затем многократно (до тех пор, пока не считаем признак конца цепочки) выполняем следующие шаги: считываем очередной символ исходной цепочки и переходим из текущего состояния в другое состояние по

дуге, помеченной этим символом. Состояние, в которое мы при этом попадаем, становится текущим.

При работе этого алгоритма возможны ситуации, аналогичные тем, которые возникают при разборе непосредственно по регулярной грамматике.

Диаграмма состояний определяет конечный автомат, построенный по регулярной грамматике, который допускает множество цепочек, составляющих язык, определяемый этой грамматикой. Состояния и дуги ДС – это графическое изображение функции переходов конечного автомата из состояния в состояние при условии, что очередной анализируемый символ совпадает с символом-меткой дуги.

Среди всех состояний выделяется *начальное* (считается, что в начальный момент своей работы автомат находится в этом состоянии) и *конечное* (если автомат завершает работу переходом в это состояние, то анализируемая цепочка им допускается).

Таким образом, **конечный автомат** (КА) – это пятёрка (**K**, **VT**, **F**, H, S), где **K** – конечное множество состояний; **VT** – конечное множество допустимых входных символов; **F** – отображение множества $\mathbf{K} \times \mathbf{VT} \to \mathbf{K}$, определяющее поведение автомата; отображение **F** часто называют *функцией переходов*; $H \in \mathbf{K}$ – начальное состояние; $S \in \mathbf{K}$ – заключительное состояние (либо конечное множество заключительных состояний).

$F(A, t) = B$ означает, что из состояния A по входному символу t происходит переход в состояние B.

Конечный автомат *допускает цепочку* $a_1a_2...a_n$, если $F(H, a_1) = A_1$; $F(A_1, a_2) = A_2$; …; $F(A_{n-2}, a_{n-1}) = A_{n-1}$; $F(A_{n-1}, a_n) = S$, где $a_i \in \mathbf{VT}$, $A_j \in \mathbf{K}$, $j = 1, 2, ..., n-1$; $i = 1, 2, ..., n$; H – начальное состояние, S – одно из заключительных состояний.

Для более удобной работы с диаграммами состояний введём несколько соглашений:

а) если из одного состояния в другое выходит несколько дуг, помеченных разными символами, то будем изображать одну дугу, помеченную всеми этими символами;

б) непомеченная дуга будет соответствовать переходу при любом символе, кроме тех, которыми помечены другие дуги, выходящие из этого состояния;

в) введём *состояние ошибки* (ER); переход в это состояние будет означать, что исходная цепочка языку не принадлежит.

По диаграмме состояний можно написать анализатор для регулярной грамматики.

Для грамматики из примера 9 анализатор будет таким:

Листинг 1. Анализатор для грамматики из примера 9.

```
#include <stdio.h>
int scan_G()
{
  /* множество состояний */
  enum state {H, A, B, C, S, ER};
  enum state CS; /* CS - текущее состояние */
  FILE *fp;
  int c;
  /* текущее состояние = начальное состояние */
  CS = H;
  /* открываем файл и считываем первый символ */
  fp = fopen("data","r");
  c = fgetc(fp);
  do
  {
    switch(CS)
    {
     /* начальное состояние */
     case H:
        if(c == 'a')
        {
         c = fgetc(fp);
         CS = A;
        }
        else if(c == 'b')
        {
         c = fgetc(fp);
         CS = B;
        }
        else CS = ER;
        break;
     /* состояние A */
     case A:
        if(c == 'b')
        {
         c = fgetc(fp);
         CS = C;
        }
        else CS = ER;
        break;
     /* состояние B */
     case B:
        if(c == 'a')
        {
```

Листинг 1. Анализатор для грамматики из примера 9.

```
          c = fgetc(fp);
          CS = C;
        }
        else CS = ER;
        break;
      /* состояние C */
      case C:
        if(c =='a')
        {
          c = fgetc(fp);
          CS = A;
        }
        else if(c == 'b')
        {
          c = fgetc(fp);
          CS = B;
        }
        else if(c == '⊥') CS = S;
        else CS = ER;
        break;
    }
  }while(CS != S && CS != ER);
  if(CS == ER) return -1;
  else return 0;
}
```

При анализе по регулярной грамматике может оказаться, что несколько нетерминалов имеют одинаковые правые части, и поэтому неясно, к какому из них делать «свёртку» (см. описание алгоритма). В терминах диаграммы состояний это означает, что из одного состояния выходит несколько дуг, ведущих в разные состояния, но помеченных одним и тем же символом.

Пример 10. Для грамматики **G** ({*a*, *b*, ⊥}, {*S*, *A* ,*B*}, **P**, *S*), где

P:

$S \to A\perp$

$A \to a \mid Bb$

$B \to b \mid Bb$

разбор будет недетерминированным (т.к. у нетерминалов *A* и *B* есть одинаковые правые части – *Bb*). Такой грамматике будет соответствовать недетерминированный конечный автомат.

Недетерминированный конечный автомат (НКА) – это пятёрка (**K**, **VT**, **F**, **H**, **S**), где **K** – конечное множество состояний; **VT** – конечное множество допустимых входных символов; **F** – отображение множества $\mathbf{K} \times \mathbf{VT}$ в множество подмножеств **K**; $\mathbf{H} \subset \mathbf{K}$ – конечное множество начальных состоя-

ний; $\mathbf{S} \subset \mathbf{K}$ – конечное множество заключительных состояний. $F(A, t) = \{B_1, B_2, ..., B_n\}$ означает, что из состояния A по входному символу t можно осуществить переход в любое из состояний B_i, $i = 1, 2, ..., n$.

В этом случае можно предложить алгоритм, который будет перебирать все возможные варианты «свёрток» (переходов) один за другим; если цепочка принадлежит языку, то будет найден путь, ведущий к успеху; если будут просмотрены все варианты, и каждый из них будет завершаться неудачей, то цепочка языку не принадлежит. Однако такой алгоритм практически неприемлем, поскольку при переборе вариантов мы, скорее всего, снова окажемся перед проблемой выбора и, следовательно, будем иметь «дерево отложенных вариантов».

Один из наиболее важных результатов теории конечных автоматов состоит в том, что класс языков, определяемых недетерминированными конечными автоматами, совпадает с классом языков, определяемых детерминированными конечными автоматами.

Это означает, что для любого НКА всегда можно построить детерминированный КА, определяющий тот же язык.

Алгоритм построения детерминированного КА по НКА

Вход: **M (K, VT, F, H, S)** – недетерминированный конечный автомат.

Выход: **M′ (K′, VT, F′, H′, S′)** – детерминированный конечный автомат, допускающий тот же язык, что и автомат **M**.

Метод:

1. Множество состояний **K′** состоит из всех подмножеств множества **K**. Каждое состояние из **K′** будем обозначать $[A_1A_2...A_n]$, где $A_i \in \mathbf{K}$.

2. Отображение **F′** определим как $\mathbf{F'}([A_1A_2...A_n], t) = [B_1B_2...B_m]$, где для каждого $1 \le j \le m$, $F(A_i, t) = B_j$ для каких-либо $1 \le i \le n$.

3. Пусть $\mathbf{H} = \{H_1, H_2, ..., H_k\}$, тогда $\mathbf{H'} = \{H_1, H_2, ..., H_k\}$.

4. Пусть $\mathbf{S} = \{S_1, S_2, ..., S_p\}$, тогда **S′** – все состояния из **K′**, имеющие вид $[...S_i...]$, $S_i \in \mathbf{S}$ для какого-либо $1 \le i \le p$.

Во множестве **K′** могут оказаться состояния, которые недостижимы из начального состояния, их можно исключить.

Пример 11.

Пусть задан НКА M = ({*H, A, B, S*}, {0, 1}, **F**, {*H*}, {*S*}), где

$F(H, 1) = B$, $F(B, 0) = A$, $F(A, 1) = B$, $F(A, 1) = S$,

тогда соответствующий детерминированный конечный автомат будет таким:

K′ = {[*H*], [*A*], [*B*], [*S*], [*HA*], [*HB*], [*HS*], [*AB*], [*AS*], [*BS*], [*HAB*], [*HAS*], [*ABS*], [*HBS*], [*HABS*]}

$F'([A], 1) = [BS]$
$F'([H], 1) = [B]$
$F'([B], 0) = [A]$
$F'([HA], 1) = [BS]$
$F'([HB], 1) = [B]$
$F'([HB], 0) = [A]$
$F'([HS], 1) = [B]$
$F'([AB], 1) = [BS]$
$F'([AB], 0) = [A]$
$F'([AS], 1) = [BS]$
$F'([BS], 0) = [A]$

F' ([*HAB*], 0) = [*A*]
F' ([*HAB*], 1) = [*BS*]
F' ([*HAS*], 1) = [*BS*]
F' ([*ABS*], 1) = [*BS*]
F' ([*ABS*], 0) = [*A*]
F' ([*HBS*], 1) = [*B*]
F' ([*HBS*], 0) = [*A*]
F' ([*HABS*], 1) = [*BS*]
F' ([*HABS*], 0) = [*A*]

S′ = {[*S*], [*HS*], [*AS*], [*BS*], [*HAS*], [*ABS*], [*HBS*], [*HABS*]}

Достижимыми состояниями в получившемся КА являются [*H*], [*B*], [*A*] и [*BS*], поэтому остальные состояния удаляются.

Таким образом, **M′** ({[*H*], [*B*], [*A*], [*BS*]}, {0, 1}, **F′**, *H*, {[*BS*]}), где F'([*A*], 1) = [*BS*], F'([*H*], 1) = [*B*], F'([*B*], 0) = [*A*], F'([*BS*], 0) = [*A*].

Контрольные вопросы

1. Какие грамматики относятся к регулярным? Назовите два класса регулярных грамматик.
2. Можно ли граф переходов конечного автома использовать для однозначного определения данного автомата?
3. Всегда ли недетерминированный КА может быть преобразован к детерминированному виду?
4. Всякая ли регулярная грамматика является однозначной?
5. Если язык задан КА, то можно ли для него построить регулярное выражение?
6. Если язык задан КА, то может ли он быть задан КС-грамматикой?

ГЛАВА 4
ПРИНЦИПЫ ПОСТРОЕНИЯ ТРАНСЛЯТОРОВ

Транслятор (от англ. translate – переводить) – это программа, которая считывает текст программы, написанной на одном языке (входном), и транслирует (переводит) его в эквивалентный текст на другом языке (выходном).

Важным пунктом в определении транслятора является эквивалентность исходной и результирующей программ. Эквивалентность этих двух программ означает совпадение их смысла с точки зрения семантики входного языка и семантики выходного языка. Без выполнения этого требования сам транслятор теряет всякий практический смысл.

Одна из важных ролей транслятора состоит в сообщении об ошибках в исходной программе, обнаруженных в процессе трансляции.

Кроме понятия «транслятор» широко употребляется также близкое ему по смыслу понятие «компилятор».

Компилятор (от англ. compile – составлять, компоновать) – это транслятор, который осуществляет перевод исходной программы в эквивалентную ей результирующую программу на машинном языке или на языке ассемблера.

Результирующая программа компилятора называется *объектной программой*, или *объектным файлом*. Даже в том случае, когда результирующая программа порождается на машинном языке, между объектным файлом и исполняемым файлом есть существенная разница.

Для связывания между собой объектных файлов, порождаемых компилятором, а также файлов библиотек предназначен **компоновщик**, или редактор связей (от англ. link – связывать, сцеплять). Результатом его работы является единый файл (*исполняемый файл*), который содержит весь текст результирующей программы на машинном языке. Для выполнения этой задачи компоновщик проходит по объектному коду, начиная с точки входа (главной исполняемой функции), находит все вызовы внешних процедур и функций, обращений к внешним переменным и увязывает их с кодом тех модулей, в которых они описаны. Функции, описанные в разных модулях исходной программы, могут вызывать друг друга сколь угодно много раз. Компоновщик должен найти соответствие каждому из этих вызовов и определить («разрешить») соответствующую ссылку. Ссылки, которым компоновщик не смог найти соответствие, называются *неразрешёнными*. В этом случае компоновщик выдаёт сообщение об ошибке.

Кроме схожих между собой понятий «транслятор» и «компилятор», существует принципиально отличное от них понятие «интерпретатор».

Интерпретатор, так же как и транслятор, анализирует текст исходной программы. Однако он не порождает результирующую программу, а сразу же выполняет исходную. Скорость исполнения откомпилированного объектного кода обычно выше, чем при исполнении кода интерпретатором. Этот недостаток ограничивает применение интерпретаторов.

Существует вариант транслятора, сочетающий в себе и компиляцию, и интерпретацию. В этом случае результатом работы транслятора является не

объектный код, а некоторый промежуточный двоичный код, который не может быть непосредственно выполнен, а должен быть обработан специальным интерпретатором.

Такой *гибридный транслятор* используется для языка программирования Java. Исходная программа на Java сначала компилируется в промежуточный код, именуемый *байт-кодом* (англ. bytecode). Затем байт-код интерпретируется виртуальной машиной. Преимущество такого решения в том, что скомпилированный на одной машине байт-код может быть выполнен на другой, например, будучи передан по сети.

Для более быстрой обработки входных данных некоторые компиляторы Java, именуемые *динамическими* или *оперативными* (англ. just-in-time) *компиляторами*, транслируют байт-код в машинный язык непосредственно перед запуском промежуточной программы для обработки входных данных.

4.1 Схема работы компилятора

Основная функция компилятора – отображение входной программы в эквивалентную ей исполняемую программу. Это отображение разделяется на две части: анализ и синтез (рис. 3).

Анализ разбивает входную программу на составные части и накладывает на них грамматическую структуру. Затем он использует эту структуру для создания промежуточного представления входной программы. Если анализ обнаруживает, что входная программа неверно составлена синтаксически либо семантически, он должен выдать информативные сообщения об этом, чтобы пользователь мог исправить обнаруженные ошибки. Анализ также собирает информацию об исходной программе и сохраняет её в структуре данных, именуемой таблицей идентификаторов (символов), которая передаётся вместе с промежуточным представлением синтезу.

Синтез строит требуемую целевую программу на основе промежуточного представления и информации из таблицы идентификаторов.

Анализ часто называют начальной стадией, а синтез – заключительной.

Если рассмотреть процесс компиляции более детально, можно увидеть, что он представляет собой последовательность фаз, каждая из которых преобразует одно из представлений входной программы в другое. Типичное разложение компилятора на фазы приведено на рисунке 3. На практике некоторые фазы могут объединяться, а межфазное промежуточное представление может не строиться явно. Таблица идентификаторов, в которой хранится информация обо всей входной программе, используется всеми фазами компилятора.

Первая фаза компиляции называется **лексическим анализом** или **сканированием.** Лексический анализатор читает поток символов, составляющих входную программу, и группирует эти символы в значащие последовательности, называемые *лексемами*. Лексемы передаются последующей фазе компилятора, синтаксическому анализу.

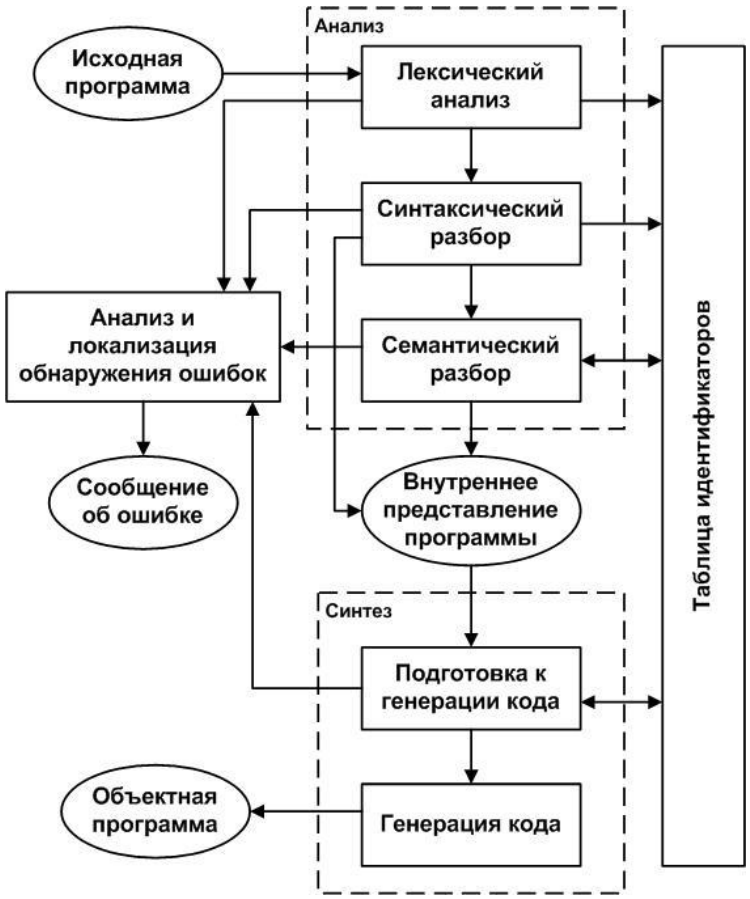


Рисунок 3. Схема работы компилятора

Вторая фаза компилятора – **синтаксический анализ** или **разбор**. Во время выполнения этой фазы используются лексемы, полученные от лексического анализатора, для создания древовидного промежуточного представления программы, которое описывает грамматическую структуру потока лексем. Типичным промежуточным представлением является *синтаксическое дерево*, в котором каждый внутренний узел представляет операцию, а дочерние узлы – аргументы этой операции.

Семантический анализатор использует синтаксическое дерево и информацию из таблицы идентификаторов для проверки входной программы на семантическую согласованность с определением языка программирования. Он также собирает информацию о типах и сохраняет её в синтаксическом дереве или в таблице идентификаторов для последующего использования в процессе генерации промежуточного кода.

После синтаксического и семантического анализа исходной программы компиляторы генерируют низкоуровневое промежуточное представление входной программы, которое можно рассматривать как программу для абстрактной вычислительной машины. Такое промежуточное представление должно обладать двумя важными свойствами: оно должно легко генерироваться и легко транслироваться в целевой машинный язык.

Некоторые компиляторы содержат фазу **машинно-независимой оптимизации**. Назначение этой оптимизации – улучшить промежуточный код, чтобы затем получить более качественный целевой код. Обычно «более качествен-

ный», «лучший» означает «более быстрый», но могут применяться и другие критерии сравнения, как, например, «более короткий код» или «код, использующий меньшее количество ресурсов».

Генератор кода получает в качестве входных данных промежуточное представление исходной программы и отображает его в целевой язык.

Важная функция компилятора состоит в том, чтобы записывать имена переменных входной программы и накапливать информацию о разных атрибутах каждого имени. Эти атрибуты могут предоставлять информацию о выделенной памяти для данного имени, его типе, области видимости (где именно в программе может использоваться его значение) и, в случае имён процедур, такие сведения, как количество и типы их аргументов, метод передачи каждого аргумента (например, по значению или по ссылке), а также возвращаемый тип.

Таблица идентификаторов представляет собой структуру данных, содержащую записи для каждого имени переменной с полями для атрибутов имени. Структура данных должна быть разработана таким образом, чтобы позволять компилятору быстро находить запись для каждого имени, а также быстро сохранять данные в записи и получать их из неё.

4.2 Многопроходные и однопроходные компиляторы

При реализации компилятора работа разных фаз может быть сгруппирована в *проходы* (англ. pass), которые считывают входной файл и записывают выходной. Например, фазы анализа – лексический анализ, синтаксический анализ, семантический анализ и генерация промежуточного кода – могут быть объединены в один проход. Оптимизация кода может представлять собой необязательный проход. Затем может быть ещё один проход, заключающийся в генерации кода для конкретной целевой архитектуры ЭВМ.

При разработке компилятора обычно стремятся максимально сократить количество проходов. При этом увеличивается скорость работы компилятора, сокращается объём необходимой ему памяти. Идеальным вариантом (но трудно достижимым) является однопроходный компилятор.

Реальные компиляторы выполняют, как правило, от двух до пяти проходов. Наиболее распространены двух- и трёхпроходные компиляторы, например: первый проход – лексический анализ, второй проход – синтаксический и семантический анализ, а третий – генерация и оптимизация кода.

Количество необходимых проходов определяется грамматикой и семантическими правилами входного языка. Чем сложнее грамматика и чем больше вариантов предполагают семантические правила, тем больше проходов будет выполнять компилятор.

4.3 Системы программирования

Компиляторы не существуют сами по себе, а решают свои задачи в рамках всего системного программного обеспечения. Основная цель компиляторов – обеспечивать разработку новых прикладных и системных программ с помо-

щью языков высокого уровня. Однако сами по себе компиляторы не решают эту задачу полностью. Поэтому они функционируют в тесном взаимодействии с другими программными средствами разработки программного обеспечения (ПО). Все эти средства вместе формируют комплекс, предназначенный для редактирования, тестирования, отладки программного обеспечения и называемый **системой программирования**. На рисунке 4 приведена общая структура системы программирования.

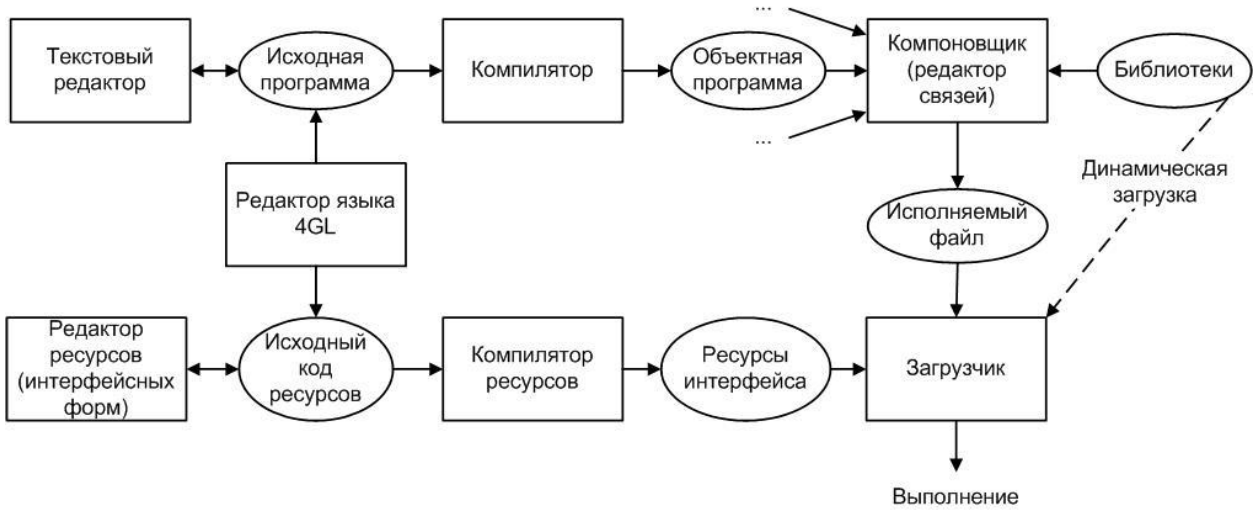


Рисунок 4. Структура системы программирования

Текстовый редактор предназначен для подготовки и внесения изменений в тексты входных программ. В современных системах программирования текстовые редакторы выполняют и другие функции:

- лексический анализ, выполняемый непосредственно в процессе редактирования входной программы;
- подсказка по вводимому программному коду (например, перечень и типы параметров функции, перечень методов и свойств класса и т.п.);
- справочная система по семантике и синтаксису используемого входного языка.

Редактор ресурсов предоставляет возможность разрабатывать ресурсы графического интерфейса пользователя (англ. Graphical User Interface, GUI). Как правило, подготовка ресурсов выполняется в графическом виде, а результатом является описание на специальном языке описания ресурсов.

Компиляторы – главная составляющая систем программирования. В состав системы программирования может входить несколько компиляторов: компилятор для входного языка программирования, компилятор ресурсов, компилятор с языка ассемблера и т.д.

Компоновщик обеспечивает объединение всех исходных модулей в единый файл. Функции этого средства разработки ПО рассматривались ранее в этой главе.

Библиотеки обеспечивают работоспособность системы программирования. Как минимум две библиотеки – библиотека функций входного языка и

библиотека функций целевой операционной системы – присутствуют в системе разработки ПО.

Загрузчик обеспечивает подготовку результирующей программы к выполнению.

Отладчик позволяет разработчику ПО находить и исправлять ошибки в программе. В этом случае речь идёт о семантических ошибках, о синтаксических ошибках разработчику сообщает компилятор.

В системы программирования внедряются также средства разработки на основе *языков четвёртого поколения* (англ. Fourth Generation Languages, 4GL). Эти языки представляют широкий набор средств для проектирования и разработки ПО на основе не синтаксических конструкций, а их графических образов. Описание программы, построенное с помощью 4GL, транслируется в исходный текст и файл описания ресурсов GUI. Этот текст можно корректировать и дополнять необходимыми функциями.

Такой подход позволяет разделить работу проектировщика ПО, дизайнера графического интерфейса пользователя и программиста, отвечающего непосредственно за создание исходного кода программного обеспечения.

Контрольные вопросы

1. Перечислите основные этапы и фазы компиляции.
2. Верно ли, что любой компилятор является транслятором? А наоборот?
3. Что такое интепретатор? В чём его отличие от транслятора и компилятора?
4. От чего зависит количество проходов, необходимых компилятору для построения результирующей объектной программы на основе исходной программы?
5. Что такое система программирования? Перечислите основные структурные элементы таких систем.

ГЛАВА 5
ТАБЛИЦЫ ИДЕНТИФИКАТОРОВ

Проверка правильности семантики и генерация кода требуют знания характеристик основных элементов исходной программы – переменных, констант, функций и других лексических единиц, встречающихся в программе на входном языке.

Главной характеристикой любого элемента исходной программы является его имя (идентификатор). Имя каждого элемента должно быть уникальным. Выделение идентификаторов и других элементов исходной программы происходит на фазе лексического анализа.

Компилятор должен иметь возможность хранить все найденные идентификаторы и связанные с ними характеристики. Для этой цели используются специальные хранилища данных, называемые **таблицами идентификаторов,** или **таблицами символов.**

Любая таблица идентификаторов состоит из набора полей, количество которых равно числу различных идентификаторов, найденных в исходной программе. Каждое поле содержит в себе полную информацию о данном элементе таблицы. Компилятор может работать с одной или несколькими таблицам идентификаторов – их количество зависит от реализации компилятора. Например, можно организовывать различные таблицы идентификаторов для различных модулей исходной программы или для различных типов элементов входного языка.

Состав информации, хранимой в таблице идентификаторов для каждого элемента исходной программы, зависит от семантики входного языка и типа элемента. Например, в таблицах идентификаторов может храниться следующая информация:

- *для переменных* – имя переменной; тип данных переменной; адрес ячейки памяти, связанной с переменной;
- *для констант* – имя константы (если оно имеется); значение константы; тип данных константы (если требуется);
- *для функций* – имя функции; количество и типы формальных аргументов функции; тип возвращаемого результата; адрес вызова кода функции.

Приведённый выше состав хранимой информации, конечно же, является только примерным. Конкретное наполнение таблиц идентификаторов зависит от реализации компилятора. Кроме того, не вся информация, хранимая в таблице идентификаторов, заполняется компилятором сразу – он может несколько раз выполнять обращение к данным в таблице идентификаторов на различных фазах компиляции.

Вне зависимости от реализации компилятора принцип его работы с таблицей идентификаторов остаётся одним и тем же – на различных фазах компиляции компилятор вынужден многократно обращаться к таблице для поиска информации и записи новых данных. Как правило, каждый элемент в исходной

программе однозначно идентифицируется своим именем. Поэтому компилятору приходится часто выполнять поиск необходимого элемента в таблице идентификаторов по его имени, в то время как процесс заполнения таблицы выполняется нечасто – новые идентификаторы описываются в программе гораздо реже, чем используются. Отсюда можно сделать вывод, что таблицы идентификаторов должны быть организованы таким образом, чтобы компилятор имел возможность максимально быстрого поиска нужного ему элемента.

Выделяют следующие способы организации таблиц идентификаторов:

- простые списки;
- упорядоченные списки;
- бинарное дерево;
- хэш-адресация с рехэшированием;
- хэш-адресация по методу цепочек;
- комбинация хэш-адресации со списком или бинарным деревом.

5.1 Простейшие методы построения таблиц идентификаторов

Простейший способ организации таблицы состоит в том, чтобы добавлять элементы в порядке их поступления. Тогда таблица идентификаторов будет представлять собой неупорядоченный список (или массив) информации, каждая ячейка которого будет содержать данные о соответствующем элементе таблицы.

Поиск нужного элемента в таблице будет в этом случае заключаться в последовательном сравнении искомого элемента с каждым элементом таблицы, пока не будет найден подходящий. И если время T_∂, требуемое на добавление элемента, не будет зависеть от числа N элементов в таблице, то для поиска элемента в неупорядоченной таблице из N элементов понадобится в среднем $N / 2$ сравнений.

Поскольку поиск в таблице идентификаторов является операцией, чаще всего выполняемой компилятором, а количество различных идентификаторов в исходной программе может быть достаточно большим (от нескольких сотен до нескольких тысяч элементов), то такой способ организации таблиц идентификаторов является неэффективным.

Поиск может быть выполнен более эффективно, если элементы таблицы упорядочены (отсортированы) согласно некоторому естественному порядку. Так как поиск выполняется по имени, естественно было бы расположить элементы таблицы в алфавитном порядке.

Эффективным методом поиска в упорядоченном списке из N элементов (рис. 3) является *бинарный* или *логарифмический поиск*, который осуществляется следующим образом:

1. Идентификатор, который требуется найти, сравнивается с элементом с номером $(N + 1) / 2$ (в середине таблицы). Если этот элемент не является искомым, то мы должны просмотреть только блок элементов, пронумерованных от 1 до $(N + 1) / 2 – 1$, или блок элементов от $(N + 1) / 2 + 1$ до N в зависимости от того, меньше или больше искомый элемент того, с которым его сравнили.

2. Затем процесс повторяется над нужным блоком в два раза меньшего размера. Так продолжается до тех пор, пока либо элемент не будет найден, либо алгоритм не дойдет до очередного блока, содержащего один или два элемента (с которыми уже можно выполнить прямое сравнение искомого элемента).

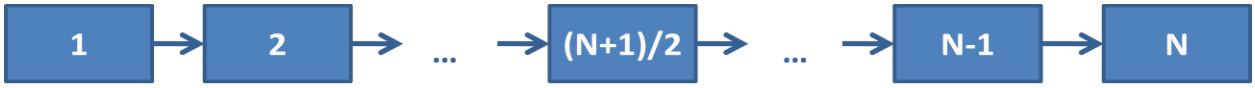


Рисунок 5. Упорядоченный список из N элементов

Так как на каждом шаге число элементов, которые могут содержать искомый элемент, сокращается в 2 раза, то максимальное число сравнений равно $1 + \log_2(N)$.

Тогда время поиска T_n элемента в таблице идентификаторов можно оценить как $T_n = O(\log_2 N)$. Для сравнения: при $N = 128$ бинарный поиск требует самое большее 8 сравнений, а поиск в неупорядоченной таблице – в среднем 64 сравнения. Метод называют «бинарным поиском», поскольку на каждом шаге объём рассматриваемой информации сокращается в два раза, а «логарифмическим» – поскольку время, затрачиваемое на поиск нужного элемента в массиве, имеет логарифмическую зависимость от общего количества элементов в нём.

Недостатком метода является требование упорядочивания таблицы идентификаторов. Так как массив информации, в котором выполняется поиск, должен быть упорядочен, то время его заполнения уже будет зависеть от числа элементов в массиве. Таблица идентификаторов зачастую просматривается ещё до того, как она заполнена полностью, поэтому требуется, чтобы условие упорядоченности выполнялось на всех этапах обращения к ней. Следовательно, для построения таблицы можно пользоваться только алгоритмом прямого упорядоченного включения элементов.

При добавлении каждого нового элемента в таблицу сначала надо определить место, куда поместить новый элемент, а потом выполнить перенос части информации в таблице, если элемент добавляется не в её конец.

В итоге при организации логарифмического поиска в таблице идентификаторов мы добиваемся существенного сокращения времени поиска нужного элемента за счёт увеличения времени на помещение нового элемента в таблицу. Поскольку добавление новых элементов в таблицу идентификаторов происходит существенно реже, чем обращение к ним, то этот метод следует признать более эффективным, чем метод организации неупорядоченной таблицы.

5.2 Построение таблиц идентификаторов по методу бинарного дерева

Можно сократить время поиска искомого элемента в таблице идентификаторов, не увеличивая значительно время, необходимое на её заполнение. Для этого надо отказаться от организации таблицы в виде непрерывного массива данных.

Существует метод построения таблиц, при котором таблица имеет форму бинарного дерева. Ориентированное дерево называется **бинарным**, если у него

в каждую вершину, кроме одной (корня), входит одна дуга, и из каждой вершины выходит не более двух дуг. Для определённости будем называть две дуги «правая» и «левая».

Каждый узел дерева представляет собой элемент таблицы, причём корневой узел является первым элементом, встреченным при заполнении таблицы.

Будем считать, что алгоритм заполнения бинарного дерева работает с потоком входных данных, содержащим идентификаторы. Пусть на множестве идентификаторов задан некоторый линейный (например, алфавитный) порядок. Таким образом, для произвольной пары идентификаторов *id*1 и *id*2 либо *id*1 < *id*2, либо *id*1 > *id*2, либо *id*1 совпадает с *id*2.

Первый идентификатор, как уже было сказано, помещается в вершину дерева. Все дальнейшие идентификаторы попадают в дерево по следующему алгоритму.

Шаг 1. Выбрать очередной идентификатор из входного потока данных. Если очередного идентификатора нет, то построение дерева закончено.

Шаг 2. Сделать текущей корневую вершину.

Шаг 3. Сравнить очередной идентификатор с идентификатором, содержащемся в текущей вершине дерева.

Шаг 4. Если очередной идентификатор меньше, то перейти к шагу 5, если равен – сообщить об ошибке и прекратить выполнение алгоритма (двух одинаковых идентификаторов быть не должно!), иначе – перейти к шагу 7.

Шаг 5. Если у текущей вершины существует левый потомок, то сделать его текущей вершиной и вернуться к шагу 3, иначе – перейти к шагу 6.

Шаг 6. Создать новую вершину, поместить в неё очередной идентификатор, сделать эту новую вершину левым потомком текущей и вернуться к шагу 1.

Шаг 7. Если у текущей вершины существует правый потомок, то сделать его текущей вершиной и вернуться к шагу 3, иначе – перейти к шагу 8.

Шаг 8. Создать новую вершину, поместить в неё очередной идентификатор, сделать эту новую вершину правым потомком текущей вершины и вернуться к шагу 1.

Рассмотрим в качестве примера последовательность идентификаторов *GA*, *D*1, *E*, *M*22, *A*12, *BC*, *F*. На рисунке 6 проиллюстрирован весь процесс построения бинарного дерева для этой последовательности идентификаторов.

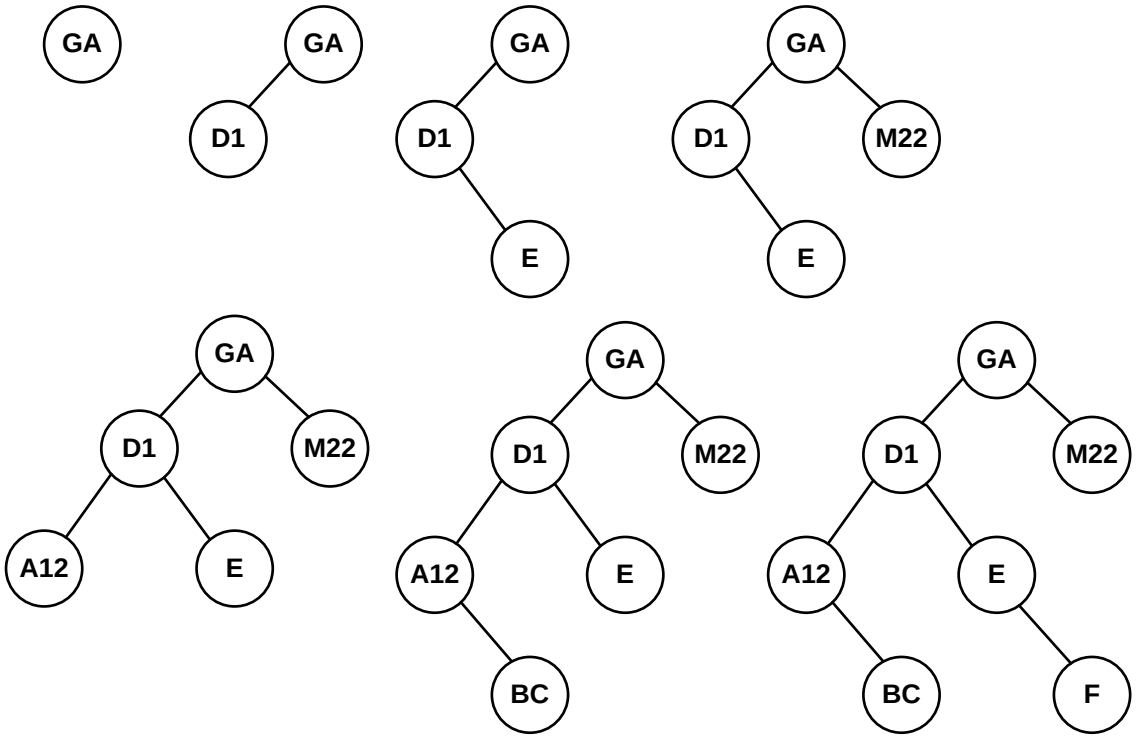


Рисунок 6. Пошаговое заполнение бинарного дерева для последовательности идентификаторов GA, D1, E, M22, A12, BC, F

Поиск нужного элемента в дереве выполняется по алгоритму, схожему с алгоритмом заполнения дерева:

Шаг 1. Сделать текущей корневую вершину.

Шаг 2. Сравнить искомый идентификатор с идентификатором, содержащемся в текущей вершине дерева.

Шаг 3. Если идентификаторы совпадают, то искомый идентификатор найден, алгоритм завершается, иначе перейти к шагу 4.

Шаг 4. Если очередной идентификатор меньше, то перейти к шагу 5, иначе – к шагу 6.

Шаг 5. Если у текущей вершины существует левый потомок, то сделать его текущей вершиной и вернуться к шагу 2, иначе искомый идентификатор не найден, алгоритм завершается.

Шаг 6. Если у текущей вершины существует правый потомок, то сделать его текущей вершиной и вернуться к шагу 2, иначе искомый идентификатор не найден, алгоритм завершается.

Например, произведём поиск идентификатора *A*12 в полностью сформированном дереве, изображённом на рисунке 6. Берём корневую вершину (она становится текущей), сравниваем идентификаторы *GA* и *A*12. Искомый идентификатор меньше, текущей становится левая вершина *D*1. Опять сравниваем идентификаторы. Искомый идентификатор меньше, текущей становится левая вершина *A*12. При следующем сравнении искомый идентификатор найден.

Если искать отсутствующий идентификатор, например, *A*11, то поиск опять пойдёт от корневой вершины. Сравниваем идентификаторы *GA* и *A*11.

Искомый идентификатор меньше, текущей становится левая вершина *D*1. Опять сравниваем идентификаторы. Искомый идентификатор меньше, текущей становится левая вершина *A*12. Искомый идентификатор меньше, но левая вершина-потомок у узла *A*12 отсутствует, поэтому в данном случае искомый идентификатор не найден.

Для данного метода число требуемых сравнений и форма получившегося дерева во многом зависят от того порядка, в котором поступают идентификаторы. Например, если в рассмотренном выше примере вместо последовательности идентификаторов *GA*, *D*1, *E*, *M*22, *A*12, *BC*, *F* взять последовательность *A*12, *GA*, *D*1, *M*22, *E*, *BC*, *F*, то полученное дерево будет иметь иной вид. А если в качестве примера взять последовательность идентификаторов *A, B, C, D, E, F*, то дерево выродится в упорядоченный однонаправленный связный список. Эта особенность является недостатком данного метода организации таблиц идентификаторов.

Если предположить, что последовательность идентификаторов в исходной программе является статистически неупорядоченной (что в целом соответствует действительности), то можно считать, что построенное бинарное дерево будет невырожденным. Тогда среднее время $T_д$ на заполнение дерева и на поиск $T_п$ элемента в нём можно оценить следующим образом:

$$T_д = N \cdot \mathrm{O}(\log_2 N);$$

$$T_п = \mathrm{O}(\log_2 N).$$

В целом метод бинарного дерева является довольно удачным механизмом для организации таблиц идентификаторов. Он нашёл своё применение в ряде компиляторов. Иногда компиляторы строят несколько различных деревьев для идентификаторов разных типов и разной длины.

5.3 Хэш-функции и хэш-адресация. Принципы работы хэш-функций

Логарифмическая зависимость времени поиска и времени заполнения таблицы идентификаторов – это хороший результат, которого можно достичь за счёт применения различных методов организации таблиц. Однако в реальных программах количество идентификаторов столь велико, что даже логарифмическую зависимость времени поиска от их числа нельзя считать удовлетворительной. Лучших результатов можно достичь, если применить методы, связанные с использованием хэш-функций и хэш-адресации.

Хэш-функцией *F* называется некоторое отображение множества входных элементов **R** на множество целых неотрицательных чисел **Z**: $F(r) = n$, $r \in \mathbf{R}$, $n \in \mathbf{Z}$. Сам термин «хэш-функция» происходит от английского термина «hash function» (hash — «мешать», «смешивать», «путать»).

Множество допустимых входных элементов **R** называют *областью определения* хэш-функции. *Множество значений* хэш-функции *F* – это подмножество **M** из множества целых неотрицательных чисел **Z**, содержащее все возможные значения, возвращаемые функцией *F*. Процесс отображения определения хэш-функции на множество значений называется *хэшированием*.

При работе с таблицей идентификаторов хэш-функция должна выполнять отображение имён идентификаторов на множество целых неотрицательных чисел. Областью определения хэш-функций будет множество всех возможных имён идентификаторов.

Хэш-адресация заключается в использовании значения, возвращаемого хэш-функцией, в качестве адреса ячейки из некоторого массива данных. Тогда размер массива данных должен соответствовать области значений используемой хэш-функций. Следовательно, в реальном компиляторе область значений хэш-функций никак не должна превышать размер доступного адресного пространства компьютера.

Метод организации таблиц идентификаторов, основанный на использовании хэш-адресации, заключается в размещении каждого элемента таблицы в ячейке, адрес которой возвращает хэш-функция, вычисленная для этого элемента. Тогда в идеальном случае для размещения любого элемента в таблице идентификаторов достаточно только вычислить его хэш-функцию и обратиться к нужной ячейке массива данных. Для поиска элемента в таблице необходимо вычислить хэш-функцию для искомого элемента и проверить, не является ли заданная ею ячейка массива пустой (если она не пуста – элемент найден, если пуста – не найден).

На рисунке 7 проиллюстрирован метод организации таблиц идентификаторов с использованием хэш-адресации. Трём различным идентификаторам A_1, A_2, A_3 соответствуют на рисунке три значения хэш-функций n_1, n_2, n_3. В ячейки, адресуемые n_1, n_2, n_3, помещается информация об идентификаторах A_1, A_2, A_3. При поиске идентификатора A_3 вычисляется значение адреса n_3 и выбираются данные из соответствующей ячейки таблицы.

Этот метод весьма эффективен, поскольку как время размещения элемента в таблице, так и время его поиска определяются только временем, затрачиваемым на вычисление хэш-функций, которое в общем случае несопоставимо меньше времени, необходимого на многократные сравнения элементов таблицы.

Метод имеет два очевидных недостатка. Первый из них – неэффективное использование объёма памяти под таблицу идентификаторов: размер массива для её хранения должен соответствовать области значений хэш-функции, в то время как реально хранимых в таблице идентификаторов может быть существенно меньше. Второй недостаток – необходимость соответствующего разумного выбора хэш-функции. Этому существенному вопросу посвящены следующие два подраздела.

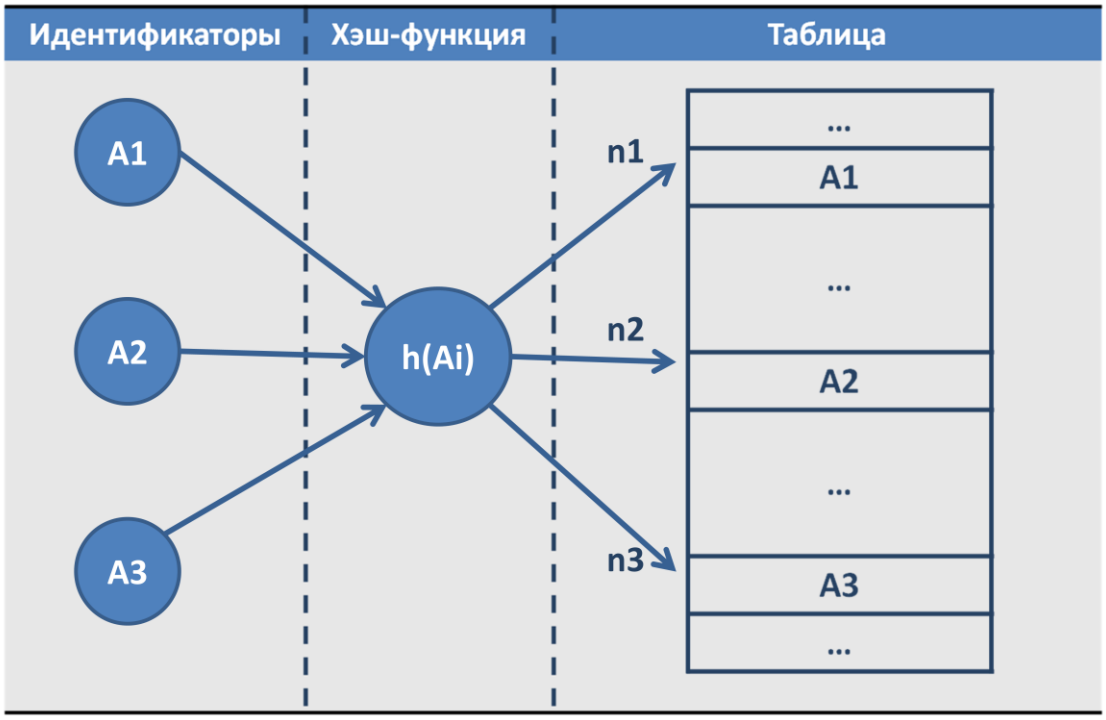


Рисунок 7. Организация таблицы идентификаторов с использованием хэш-адресации

5.4 Построение таблиц идентификаторов на основе хэш-функции

Существуют различные варианты хэш-функции. Получение результата хэш-функции (хэширование) – обычно достигается за счёт выполнения над цепочкой символов некоторых простых арифметических и логических операций. Самой простой хэш-функцией для символа является код внутреннего представления в ЭВМ литеры символа. Эту хэш-функцию можно использовать и для цепочки символов, выбирая первый символ в цепочке. Так, если ASCII код символа *A* есть 33_{10}, то результатом хэширования идентификатора *ATable* будет код 33_{10}.

Хэш-функция, предложенная выше, очевидно не удовлетворительна: при использовании такой хэш-функции возникнет новая проблема – двум различным идентификаторам, начинающимся с одной и той же буквы, будет соответствовать одно и то же значение хэш-функции. Тогда при хэш-адресации в одну ячейку таблицы идентификаторов по одному и тому же адресу должны быть помещены два различных идентификатора, что явно невозможно. Такая ситуация, когда двум или более идентификаторам соответствует одно и то же значение функции, называется **коллизией.**

Естественно, что хэш-функция, допускающая коллизии, не может быть напрямую использована для хэш-адресации в таблице идентификаторов. Достаточно получить хотя бы один случай коллизии на всём множестве идентификаторов, чтобы такой хэш-функцией нельзя было пользоваться непосредственно.

Очевидно, что для полного исключения коллизий хэш-функция должна быть *взаимно однозначной*. Тогда любым двум произвольным элементам из области определения хэш-функции будут всегда соответствовать два различных её значения. Теоретически для идентификаторов такую хэш-функцию постро-

ить можно, так как и область определения хэш-функции (все возможные имена идентификаторов), и область её значений (целые неотрицательные числа) являются бесконечными счётными множествами.

Но практически это сделать сложно, т.к. в реальности область значений любой хэш-функции ограничена размером доступного адресного пространства памяти ЭВМ. Если даже учесть, что длина принимаемой во внимание строки идентификатора в реальных компиляторах также практически ограничена – обычно она лежит в пределах от 32 до 128 символов (то есть и область определения хэш-функции конечна), то и тогда количество всех возможных идентификаторов всё равно больше количества допустимых адресов в современных компьютерах. Таким образом, создать взаимно однозначную хэш-функцию практически ни в каком варианте невозможно. Следовательно, **невозможно избежать возникновения коллизий.**

Одним из **способов решения проблемы коллизий** является метод *рехэширования* (или расстановки). Согласно этому методу, если для элемента A адрес $h(A)$, вычисленный с помощью хэш-функции, указывает на уже занятую ячейку, то необходимо вычислить значение функции $h_1(A)$ и проверить занятость ячейки по этому адресу. Если и она занята, то вычисляется значение $h_2(A)$, и так до тех пор, пока либо не будет найдена свободная ячейка, либо очередное значение $h_i(A)$ совпадёт с $h(A)$. В последнем случае считается, что таблица идентификаторов заполнена – выдаётся сообщение об ошибке размещения идентификатора в таблице. Особенностью метода является то, что первоначально таблица идентификаторов должна быть заполнена информацией, которая позволила бы говорить о том, что её ячейки являются пустыми (не содержат данных). Например, если используются указатели для хранения имён идентификаторов, то таблицу надо предварительно заполнить пустыми указателями.

Такую таблицу идентификаторов можно организовать по следующему алгоритму размещения элемента:

Шаг 1. Вычислить значение хэш-функции $n = h(A)$ для нового элемента A.

Шаг 2. Если ячейка по адресу n пустая, то поместить в неё элемент A и завершить алгоритм, иначе $i = 1$ и перейти к шагу 3.

Шаг 3. Вычислить $n_i = h_i(A)$. Если ячейка по адресу n_i пустая, то поместить в неё элемент A и завершить алгоритм, иначе перейти к шагу 4.

Шаг 4. Если $n = n_i$, то сообщить об ошибке и завершить алгоритм, иначе $i = i + 1$ и вернуться к шагу 3.

Тогда поиск элемента A в таблице идентификаторов, организованной таким образом, будет выполняться по следующему алгоритму:

Шаг 1. Вычислить значение хэш-функции $n = h(A)$ для нового элемента A.

Шаг 2. Если ячейка по адресу n пустая, то элемент не найден, алгоритм завершён, иначе сравнить имя элемента в ячейке n с именем искомого элемента A. Если они совпадают, то элемент найден и алгоритм завершён, иначе $i = 1$ и перейти к шагу 3.

Шаг 3. Вычислить $n_i = h_i(A)$. Если ячейка по адресу n_i пустая или $n = n_i$, то элемент не найден и алгоритм завершён, иначе сравнить имя элемента в ячейке n_i с именем искомого элемента A. Если они совпадают, то элемент найден и алгоритм завершён, иначе $i = i + 1$ и повторить шаг 3.

Итак, количество операций, необходимых для поиска или размещения в таблице элемента, зависит от заполненности таблицы. Естественно, функции $h_i(A)$ должны вычисляться единообразно на этапах размещения и поиска элемента. Вопрос только в том, как организовать вычисление функций $h_i(A)$.

Самым простым методом вычисления функции $h_i(A)$ является её организация в виде $h_i(A) = (h(A) + p_i) \bmod N_m$, где p_i – некоторое вычисляемое целое число, а N_m – максимальное число элементов в таблице идентификаторов. В свою очередь, самым простым подходом здесь будет положить $p_i = i$. Тогда получаем формулу $h_i(A) = (h(A) + i) \bmod N_m$. В этом случае при совпадении значений хэш-функции для каких-либо элементов поиск свободной ячейки в таблице начинается последовательно от текущей позиции, заданной хэш-функцией $h(A)$.

Этот способ нельзя признать особенно удачным, т.к. при совпадении хэш-адресов элементы в таблице начинают группироваться вокруг них, что увеличивает число необходимых сравнений при поиске и размещении. Среднее время поиска элемента в такой таблице в зависимости от числа операций сравнения можно оценить следующим образом:

$$T_n = O\left(\frac{1-\frac{Lf}{2}}{1-Lf}\right).$$

Здесь Lf – степень заполненности таблицы идентификаторов (отношение числа занятых ячеек таблицы к максимально допустимому числу элементов в ней).

Рассмотрим в качестве примера ряд последовательных ячеек таблицы n_1, n_2, n_3, n_4, n_5 и ряд идентификаторов, которые надо разместить в ней: A_1, A_2, A_3, A_4, A_5 при условии, что $h(A_1) = h(A_2) = h(A_5) = n_1$, $h(A_3) = n_2$, $h(A_4) = n_4$. Последовательность размещения идентификаторов в таблице при использовании простейшего метода рехэширования показана на рисунке 8. В итоге после размещения в таблице для поиска идентификатора A_1 потребуется 1 сравнение, для A_2 – 2 сравнения, для A_3 – 2 сравнения, для A_4 – 1 сравнение и для A_5 – 5 сравнений.

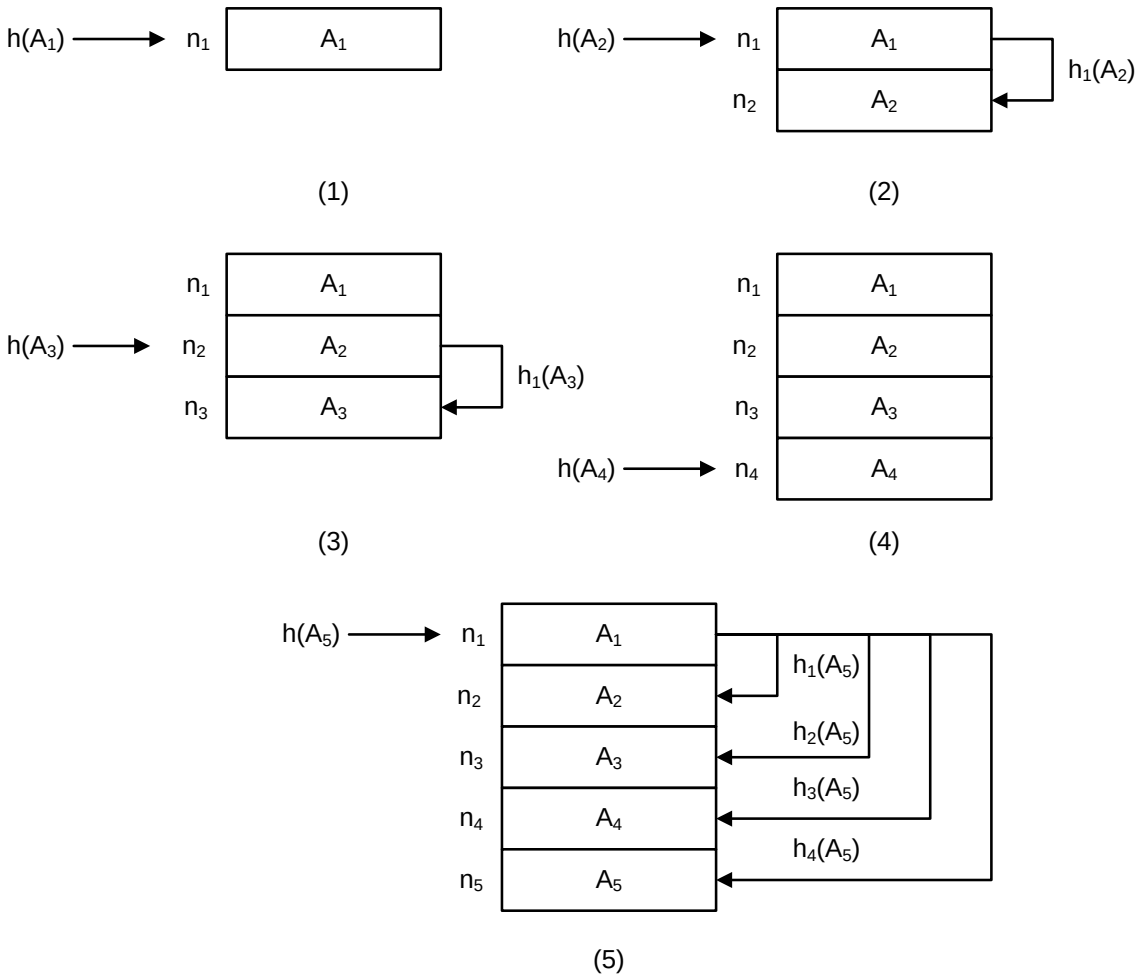


Рисунок 8. Заполнение таблицы идентификаторов при использовании простого рехэширования

Даже такой примитивный метод рехэширования является достаточно эффективным средством организации таблиц идентификаторов при неполном заполнении таблицы. Имея, например, даже заполненную на 90% таблицу, для 1024 идентификаторов в среднем необходимо выполнить 5,5 сравнений для поиска одного идентификатора, в то время как даже логарифмический поиск даёт в среднем от 9 до 10 сравнений. Сравнительная эффективность метода будет ещё выше при росте числа идентификаторов и снижении заполненности таблицы.

Среднее время на помещение одного элемента в таблицу и на поиск элемента в таблице можно снизить, если применить более совершенный метод рехэширования. Одним из таких методов является использование в качестве p_i для функции $h_i(A) = (h(A) + p_i) \bmod N_m$ последовательности *псевдослучайных* целых чисел p_1, p_2, ..., p_k. При хорошем выборе генератора псевдослучайных чисел длина k последовательности будет равна N_m. Тогда среднее время поиска одного элемента в таблице можно оценить следующим образом:

$$T_n = O\left(\frac{1}{Lf}\log_2(1-Lf)\right).$$

Существуют и другие методы организации функций рехэширования $h_i(A)$, основанные на квадратичных вычислениях или, например, на вычислении с помощью произведения по формуле: $h_i(A) = (h(A) \cdot i) \bmod N_m$, где N_m – простое число. В целом рехэширование позволяет добиться неплохих результатов для эффективного поиска элемента в таблице (лучших, чем бинарный поиск и бинарное дерево), но эффективность метода сильно зависит от заполненности таблицы идентификаторов и качества используемой хэш-функции – чем реже возникают коллизии, тем выше эффективность метода. Требование неполного заполнения таблицы ведёт к **неэффективному использованию объёма доступной памяти**.

5.5 Построение таблиц идентификаторов по методу цепочек

Неполное заполнение таблицы идентификаторов при применении хэш-функции ведёт к неэффективному использованию всего объёма памяти, доступного компилятору. Причём объём неиспользуемой памяти будет тем выше, чем больше информации хранится для каждого идентификатора. Этого недостатка можно избежать, если дополнить таблицу идентификаторов некоторой **промежуточной хэш-таблицей.**

В ячейках хэш-таблицы может храниться либо пустое значение, либо значение указателя на некоторую область памяти из основной таблицы идентификаторов. Тогда хэш-функция вычисляет адрес, по которому происходит обращение сначала к хэш-таблице, а потом уже через неё по найденному адресу – к самой таблице идентификаторов. Если соответствующая ячейка таблицы идентификаторов пуста, то ячейка хэш-таблицы будет содержать пустое значение. Тогда вовсе необязательно иметь в самой таблице идентификаторов ячейку для каждого возможного значения хэш-функции – таблицу можно сделать динамической, так чтобы её объём рос по мере заполнения (первоначально таблица идентификаторов не содержит ни одной ячейки, а все ячейки хэш-таблицы имеют пустое значение).

Такой подход позволяет добиться двух положительных результатов: во-первых, нет необходимости заполнять пустыми значениями таблицу идентификаторов – это можно сделать только для хэш-таблицы; во-вторых, каждому идентификатору будет соответствовать строго одна ячейка в таблице идентификаторов (в ней не будет пустых неиспользуемых ячеек). Пустые ячейки в таком случае будут только в хэш-таблице, и объём неиспользуемой памяти не будет зависеть от объёма информации, хранимой для каждого идентификатора – для каждого значения хэш-функции будет расходоваться только память, необходимая для хранения одного указателя на основную таблицу идентификаторов.

На основе этой схемы можно реализовать ещё один способ организации таблиц идентификаторов с помощью хэш-функции, называемый **методом цепочек**. Для метода цепочек в таблицу идентификаторов для каждого элемента добавляется ещё одно поле, в котором может содержаться ссылка на любой элемент таблицы. Первоначально это поле всегда пустое. Также

для этого метода необходимо иметь одну специальную переменную, которая всегда указывает на первую свободную ячейку основной таблицы идентификаторов (первоначально – указывает на начало таблицы).

Метод цепочек работает следующим образом:

Шаг 1. Во все ячейки хэш-таблицы поместить пустое значение, таблица идентификаторов не должна содержать ни одной ячейки, переменная *FreePtr* (указатель первой свободной ячейки) указывает на начало таблицы идентификаторов; $i = 1$.

Шаг 2. Вычислить значение хэш-функции n_i для нового элемента A_i. Если ячейка хэш-таблицы по адресу n_i пустая, то поместить в неё значение переменной *FreePtr* и перейти к шагу 5; иначе перейти к шагу 3.

Шаг 3. Положить $j = 1$, выбрать из хэш-таблицы адрес ячейки таблицы идентификаторов m_j и перейти к шагу 4.

Шаг 4. Для ячейки таблицы идентификаторов по адресу m_j проверить значение поля ссылки. Если оно пустое, то записать в него адрес из переменной *FreePtr* и перейти к шагу 5; иначе $j = j + 1$, выбрать из поля ссылки адрес m_j и повторить шаг 4.

Шаг 5. Добавить в таблицу идентификаторов новую ячейку, записать в неё информацию для элемента A_i (поле ссылки должно быть пустым), в переменную *FreePtr* поместить адрес за концом добавленной ячейки. Если больше нет идентификаторов, которые надо разместить в таблице, то выполнение алгоритма закончено, иначе $i = i + 1$ и перейти к шагу 2.

Поиск элемента в таблице идентификаторов, организованной таким образом, будет выполняться по следующему алгоритму:

Шаг 1. Вычислить значение хэш-функции n для искомого элемента A. Если ячейка хэш-таблицы по адресу n пустая, то элемент не найден и алгоритм завершён, иначе положить $j = 1$, выбрать из хэш-таблицы адрес ячейки таблицы идентификаторов $m_j = n$.

Шаг 2. Сравнить имя элемента в ячейке таблицы идентификаторов по адресу m_j с именем искомого элемента A. Если они совпадают, то искомый элемент найден и алгоритм завершён, иначе перейти к шагу 3.

Шаг 3. Проверить значение поля ссылки в ячейке таблицы идентификаторов по адресу m_j. Если оно пустое, то искомый элемент не найден и алгоритм завершён; иначе $j = j + 1$, выбрать из поля ссылки адрес m_j и перейти к шагу 2.

При такой организации таблиц идентификаторов в случае возникновения коллизии алгоритм размещает элементы в ячейках таблицы, связывая их друг с другом последовательно через поле ссылки. При этом элементы не могут попадать в ячейки с адресами, которые потом будут совпадать со значениями хэш-функции. Таким образом, дополнительные коллизии не возникают. В итоге в таблице возникают своеобразные цепочки связанных элементов, откуда происходит и название данного метода – «метод цепочек».

На рисунке 9 проиллюстрировано заполнение хэш-таблицы и таблицы идентификаторов для примера, который ранее был рассмотрен на рисунке 8 для метода простого рехэширования. После размещения в таблице для поиска

идентификатора A_1 потребуется 1 сравнение, для A_2 – 2 сравнения, для A_3 – 1 сравнение, для A_4 – 1 сравнение и для A_5 – 3 сравнения (сравните с результатами простого рехэширования).

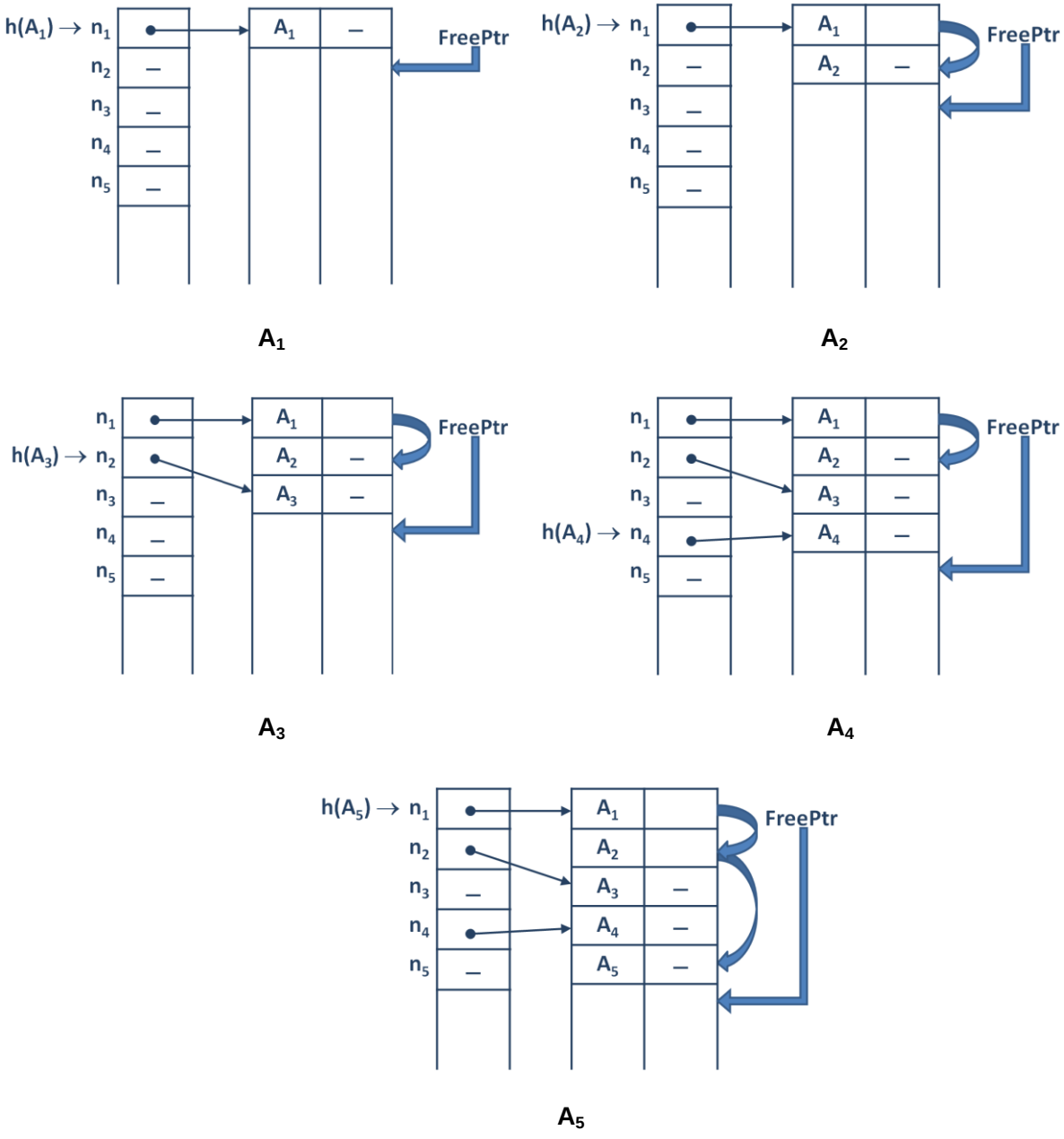


Рисунок 9. Заполнение хэш-таблицы и таблицы идентификаторов при использовании метода цепочек

Метод цепочек является очень эффективным средством организации таблиц идентификаторов. Среднее время на размещение одного элемента и на поиск элемента в таблице для него зависит только от среднего числа коллизий, возникающих при вычислении хэш-функции. Накладные расходы памяти, связанные с необходимостью иметь одно дополнительное поле указателя в таблице идентификаторов на каждый её элемент, можно признать вполне оправданными. Этот метод позволяет более экономно использовать память, но требует организации работы с динамическими массивами данных.

5.6 Выбор хэш-функции

Выше была рассмотрена весьма примитивная хэш-функция, которую никак нельзя назвать удовлетворительной. Хорошая хэш-функция распределяет поступающие на её вход идентификаторы равномерно на все имеющиеся в распоряжении адреса, так что коллизии возникают не столь часто.

Для хэш-адресации с рехэшированием в качестве хэш-функции возьмём функцию, которая будет получать на входе строку, а в результате выдавать сумму кодов первого, среднего и последнего элементов строки. Причём если строка содержит менее трёх символов, то один и тот же символ будет взят и в качестве первого, и в качестве среднего, и в качестве последнего.

Будем считать, что прописные и строчные буквы в идентификаторах различны. В качестве кодов символов возьмём коды таблицы ASCII. Тогда, если положить, что строка из области определения хэш-функции содержит только цифры и буквы английского алфавита, то минимальным значением хэш-функции будет сумма трёх кодов цифры «0», а максимальным значением – сумма трёх кодов литеры «z».

Таким образом, область значений выбранной хэш-функции может быть описана как:

$$'0' + '0' + '0' \ldots 'z' + 'z' + 'z'$$

Диапазон области значений составляет 223 элемента. Длина входных идентификаторов в данном случае ничем не ограничена. Для удобства пользования опишем две константы, задающие границы области значений хэш-функции:

```
HASH_MIN = '0' + '0' + '0';
HASH_MAX = 'z' + 'z' + 'z'.
```

Сама хэш-функция без учёта рехэширования будет вычислять следующее выражение:

```
HASH = sName[0] + sName[Length(sName) / 2]) +
          sName[Length(sName) - 1],
```

где `sName` – это входная строка (идентификатор).

Для рехэширования возьмём простейший генератор последовательности псевдослучайных чисел, построенный на основе формулы $F = i \cdot H_1 \bmod H_2$, где H_1 и H_2 – простые числа, выбранные таким образом, чтобы H_1 было в диапазоне от $H_2 / 2$ до H_2. Причём, чтобы этот генератор выдавал максимально длинную последовательность во всём диапазоне от `HASH_MIN` до `HASH_MAX`, H_2 должно быть максимально близко к величине `HASH_MAX - HASH_MIN + 1`. В данном случае диапазон содержит 223 элемента, и поскольку 223 – простое число, то возьмём $H_2 = 223$ (если бы размер диапазона не был простым числом, то в качестве H_2 нужно было бы взять ближайшее к нему меньшее простое число). В качестве H_1 возьмём 127.

Опишем соответствующие константы:

```
REHASH1 = 127;
REHASH2 = 223.
```

Тогда хэш-функция с учётом рехэширования будет иметь следующий вид:

```
Result = (HASH – HASH_MIN + iNum * REHASH1 mod REHASH2)
mod (HASH_MAX – HASH_MIN + 1) + HASH_MIN;
if Result < HASH_MIN Result = HASH_MIN;
```

Входные параметры этой функции: `iNum` – индекс рехэширования (если `iNum = 0`, то рехэширование отсутствует). Строка проверки величины результата (`Result < HASH_MIN`) добавлена, чтобы исключить ошибки в тех случаях, когда на вход функции подаётся строка, содержащая символы вне диапазона '0'...'z' (если контроль входных идентификаторов отсутствует, это имеет смысл).

В реальных компиляторах практически всегда, так или иначе, используется хэш-адресация. Алгоритм применяемой хэш-функции обычно составляет «ноу-хау» разработчиков компилятора. При разработке хэш-функции создатели компилятора стремятся свести к минимуму количество возникающих коллизий не на всём множестве возможных идентификаторов, а на тех их вариантах, которые наиболее часто встречаются во входных программах. Конечно, принять во внимание все допустимые исходные программы невозможно. Чаще всего выполняется статистическая обработка встречающихся имён идентификаторов на некотором множестве типичных исходных программ, а также принимаются во внимание соглашения о выборе имён идентификаторов, общепринятые для входного языка. Хорошая хэш-функция – это шаг к значительному ускорению работы компилятора, поскольку обращения к таблицам идентификаторов выполняются многократно на различных фазах компиляции.

Контрольные вопросы

1. Какая информация может храниться в таблице идентификаторов?
2. Исходя из каких характеристик оценивается эффективность того или иного метода организации таблицы идентификаторов?
3. Какие существуют методы организации таблиц идентификаторов?
4. Что такое коллизия? Почему она происходит при использовании хэш-функций для организации таблиц идентификаторов?
5. В чём заключаются преимущества и недостатки метода цепочек по сравнению с методом рехэширования?

ГЛАВА 6
ЛЕКСИЧЕСКИЙ АНАЛИЗАТОР

Первая фаза компиляции называется **лексическим анализом** или сканированием. **Лексический анализатор (сканер)** читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называющиеся *лексемами*.

Лексема – это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своём составе других структурных единиц языка. Лексемами языков программирования являются идентификаторы, константы, ключевые слова языка, знаки операций и т.п.

На вход лексического анализатора поступает текст исходной программы, а выходная информация передаётся для дальнейшей обработки синтаксическому анализатору. Для каждой лексемы сканер строит выходной *токен* (англ. token) вида

‹имя_токена, значение_атрибута›

Первый компонент токена, *имя_токена*, представляет собой абстрактный символ, использующийся во время синтаксического анализа, а второй компонент, значение атрибута, указывает на запись в таблице идентификаторов, соответствующую данному токену.

Предположим, например, что исходная программа содержит инструкцию присваивания

```
a = b + c * d
```

Символы в этом присваивании могут быть сгруппированы в следующие лексемы и отображены в следующие токены:

1) `a` представляет собой лексему, которая может отображаться в токен ‹*id*, 1›, где *id* – абстрактный символ, обозначающий идентификатор, а 1 указывает запись в таблице идентификаторов для `a`, в которой хранится такая информация как имя и тип идентификатора.
2) Символ присваивания = представляет собой лексему, которая отображается в токен ‹=›. Поскольку этот токен не требует значения атрибута, второй компонент данного токена опущен. В качестве имени токена может быть использован любой абстрактный символ, например такой, как «assign», но для удобства записи в качестве имени абстрактного символа можно использовать саму лексему.
3) `b` представляет собой лексему, которая отображается в токен ‹*id*, 2›, где 2 указывает на запись в таблице идентификаторов для `b`.
4) + является лексемой, отображаемой в токен ‹+›.
5) `c` – лексема, отображаемая в токен ‹*id*, 3›, где 3 указывает на запись в таблице идентификаторов для `c`.

6) `*` – лексема, отображаемая в токен ‹*›.

7) `d` – лексема, отображаемая в токен ‹*id*, 4›, где 4 указывает на запись в таблице идентификаторов для `d`.

Пробелы, разделяющие лексемы, лексическим анализатором пропускаются.

Представление инструкции присваивания после лексического анализа в виде последовательности токенов примет следующий вид:

‹*id*, 1›‹=›‹*id*, 2›‹+›‹*›‹*id*, 3›‹*id*, 4›.

В таблице 1 приведены некоторые типичные токены, неформальное описание их шаблонов и некоторые примеры лексем. *Шаблон* (англ. pattern) – это описание вида, который может принимать лексема токена. В случае ключевого слова шаблон представляет собой просто последовательность символов, образующих это ключевое слово.

Чтобы увидеть использование этих концепций на практике, рассмотрим инструкцию на языке программирования Си

```
printf("Total = %d\n", score);
```

в которой `printf` и `score` представляют собой лексемы, соответствующие токену *id*, а `"Total = %d\n"` является лексемой, соответствующей токену *literal*.

Таблица 1. Примеры токенов

Токен	Неформальное описание	Примеры лексем
if	Символы i, f	`if`
else	Символы e, l, s, e	`else`
comp	< или > или <= или >= или == или !=	`<=`
id	Буква, за которой следуют буквы и цифры	`score, D2`
number	Любая числовая константа	`3.14159`
literal	Последовательность любых символов, заключённая в кавычки (кроме самих кавычек)	`"Total = %d\n"`

С теоретической точки зрения лексический анализатор не является обязательной, необходимой частью компилятора. Его функции могут выполняться на этапе синтаксического разбора. Однако существует несколько причин, исходя из которых в состав практически всех компиляторов включают лексический анализ:

- упрощается работа с текстом исходной программы на этапе синтаксического разбора и сокращается объём обрабатываемой информации, так как лексический анализатор структурирует поступающий на вход исходный текст программы и удаляет всю незначащую информацию;
- для выделения в тексте и разбора лексем можно применять простую, эффективную и теоретически хорошо проработанную технику анализа,

в то время как на этапе синтаксического анализа конструкций исходного языка используются достаточно сложные алгоритмы разбора;

– сканер отделяет сложный по конструкции синтаксический анализатор от работы непосредственно с текстом исходной программы, структура которого может варьироваться в зависимости от версии входного языка – при такой конструкции компилятора при переходе от одной версии языка к другой достаточно только перестроить относительно простой сканер.

Функции, выполняемые лексическим анализатором, и состав лексем, которые он выделяет в тексте исходной программы, могут меняться в зависимости от версии компилятора. В основном лексические анализаторы выполняют исключение из текста исходной программы комментариев и незначащих пробелов, а также выделение лексем следующих типов: идентификаторов, строковых, символьных и числовых констант, ключевых (служебных) слов входного языка.

В большинстве компиляторов лексический и синтаксический анализаторы – это взаимосвязанные части. Лексический разбор исходного текста в таком варианте выполняется поэтапно, так что синтаксический анализатор, выполнив разбор очередной конструкции языка, обращается к сканеру за следующей лексемой. При этом он может сообщить информацию о том, какую лексему следует ожидать. В процессе разбора может даже происходить «откат назад», чтобы выполнить анализ текста на другой основе. В дальнейшем будем исходить из предположения, что все лексемы могут быть однозначно выделены сканером на этапе лексического разбора.

Работу синтаксического и лексического анализаторов можно изобразить в виде схемы, приведённой на рисунке 10.

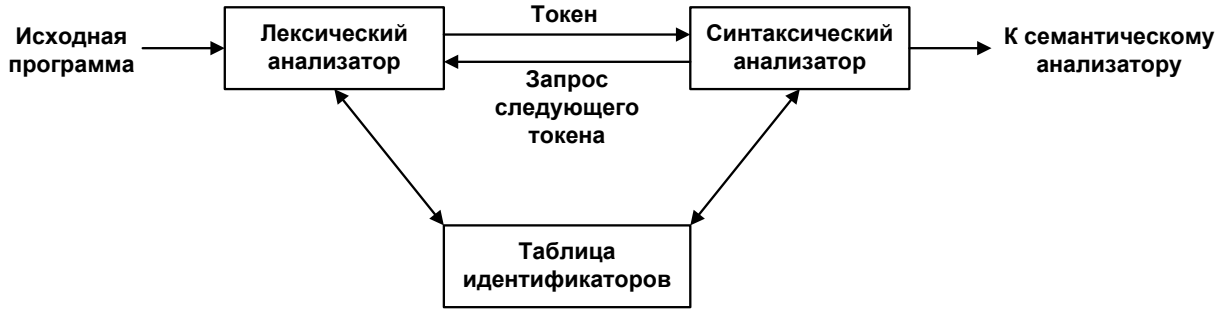


Рисунок 10. Взаимодействие лексического анализатора с синтаксическим

В простейшем случае фазы лексического и синтаксического анализа могут выполняться компилятором последовательно. Но для многих языков программирования информации на этапе лексического анализа может быть недостаточно для однозначного определения типа и границ очередной лексемы.

Иллюстрацией такого случая может служить пример оператора присваивания из языка программирования Си

```
k = i+++++j;
```

который имеет только одну верную интерпретацию (если операции разделить пробелами):

```
k = i++ + ++j;
```

Если невозможно определить границы лексем, то лексический анализ исходного текста должен выполняться поэтапно. Тогда лексический и синтаксический анализаторы должны функционировать параллельно. Лексический анализатор, найдя очередную лексему, передаёт её синтаксическому анализатору, тот пытается выполнить анализ считанной части исходной программы и может либо запросить у лексического анализатора следующую лексему, либо потребовать от него вернуться на несколько шагов назад и попробовать выделить лексемы с другими границами.

Очевидно, что параллельная работа лексического и синтаксического анализаторов более сложна в реализации, чем их последовательное выполнение. Кроме того, такая реализация требует больше вычислительных ресурсов и в общем случае большего времени на анализ исходной программы.

Чтобы избежать параллельной работы лексического и синтаксического анализаторов, разработчики компиляторов и языков программирования часто идут на разумные ограничения синтаксиса входного языка. Например, для языка Си принято соглашение, что при возникновении проблем с определением границ лексемы всегда выбирается лексема максимально возможной длины. Для рассмотренного выше оператора присваивания это приводит к тому, что при чтении четвёртого знака + из двух вариантов лексем (+ – знак сложения, ++ – оператор инкремента) лексический анализатор выбирает самую длинную, т.е. ++, и в целом весь оператор будет разобран как

```
k = i++ ++ +j;
```

что неверно. Компилятор gcc в этом случае выдаст сообщение об ошибке: lvalue required as increment operand – в качестве операнда оператора инкремента требуется l-значение. Любые неоднозначности при анализе данного оператора присваивания могут быть исключены только в случае правильной расстановки пробелов в исходной программе.

Вид представления информации после выполнения лексического анализа целиком зависит от конструкции компилятора. Но в общем виде её можно представить как **таблицу лексем**, которая в каждой строчке должна содержать информацию о виде лексемы, её типе и, возможно, значении. Обычно такая таблица имеет два столбца: первый – строка лексемы, второй – указатель на информацию о лексеме, может быть включён и третий столбец – тип лексем. Не следует путать таблицу лексем и таблицу идентификаторов – это две принципиально разные таблицы! Таблица лексем содержит весь текст исходной программы, обработанный лексическим анализатором. В неё входят все возможные типы лексем, при этом, любая лексема может в ней встречаться любое число раз. Таблица идентификаторов содержит только следующие типы лексем: идентификаторы и константы. В неё не попадают ключевые слова входного языка, знаки операций и разделители. Каждая лексема в таблице идентификаторов может встречаться только один раз.

Вот пример фрагмента текста программы на языке Паскаль и соответствующей ему таблицы лексем:

```
...
begin
 for i:=1 to N do
  fg := fg * 0.5
...
```

Таблица 2. Таблица лексем программы

Лексема	Тип лексемы	Значение
`begin`	Ключевое слово	X1
`for`	Ключевое слово	X2
`i`	Идентификатор	i : 1
`:=`	Знак присваивания	
`1`	Целочисленная константа	1
`to`	Ключевое слово	X3
`N`	Идентификатор	N : 2
`do`	Ключевое слово	X4
`fg`	Идентификатор	fg : 3
`:=`	Знак присваивания	
`fg`	Идентификатор	fg : 3
`*`	Знак арифметической операции	
`0.5`	Вещественная константа	0.5

В таблице 2 поле «значение» подразумевает некое кодовое значение, которое будет помещено в итоговую таблицу лексем. Значения, приведённые в примере, являются условными. Конкретные коды выбираются разработчиками при реализации компилятора. Связь между таблицей лексем и таблицей идентификаторов отражена в примере некоторым индексом, следующим после идентификатора за знаком «:». В реальном компиляторе эта связь определяется его реализацией.

6.1 Разработка лексического анализатора

В качестве примера возьмём входной язык, содержащий набор условных операторов **if … then … else** и **if … then**, разделённых символом ‘;’ (точка с запятой). Операторы условия содержат логические выражения, построенные с помощью круглых скобок и операций **or**, **xor**, **and**, операндами которых являются идентификаторы и целые десятичные константы без знака. В исполнительной части эти операторы содержат оператор присваивания (‘:=’) или другой условный оператор.

Описанный выше входной язык может быть задан с помощью КС-грамматики **G** ({**if**, **then**, **else**, ‘:=’, **or**, **xor**, **and**, ‘(‘, ‘)’, ‘;’, ‘_’, ‘a’, ‘b’, ‘c’, ..., ‘x’, ‘y’, ‘z’, ‘A’, ‘B’, ‘C’, ..., ‘X’, ‘Y’, ‘Z’, ‘0’, ‘1’, ‘2’, ’3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ⊥},

{*S, F, E, D, C, I, L, N, Z*}, **P**, *S*) с правилами **P** (правила представлены в расширенной форме Бэкуса-Наура (см. раздел 2.1)):

$S \to F$';'$\perp \mid F$';'S
$F \to$ **if** E **then** F **else** $F \mid$ **if** E **then** $F \mid I$ ':=' E
$E \to E$ **or** $D \mid E$ **xor** $D \mid D$
$D \to D$ **and** $C \mid C$
$C \to I \mid N \mid$ '(E)'
$I \to$ '_' $\mid L$ {'_' $\mid L \mid$ '0' $\mid Z$}
$L \to$ 'a' | 'b' | 'c' | ... | 'x' | 'y' | 'z' | 'A' | 'B' | 'C' | ... | 'X' | 'Y' | 'Z'
$N \to Z$ { '0' $\mid Z$}
$Z \to$ '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Целевым символом грамматики является символ *S*; символ $\perp$ – маркер конца текста программы.

Лексемы входного языка разделим на несколько классов:

- шесть ключевых слов языка (**if**, **then**, **else**, **or**, **xor**, **and**) – класс 1;
- разделители ('(', ')', ';') – класс 2;
- знак операции присваивания (':=') – класс 3;
- идентификаторы – класс 4;
- целые десятичные константы без знака – класс 5.

Внутреннее представление лексем – это пара вида: *номер_класса, номер_в_классе*. *Номер_в_классе* – это номер строки в таблице лексем соответствующего класса.

Границами лексем будут служить пробелы, знаки табуляции, знаки перевода строки и возврата каретки, круглые скобки, точка с запятой и знак двоеточия. При этом круглые скобки и точка с запятой сами являются лексемами, а знак двоеточия, являясь границей лексемы, в то же время является и началом другой лексемы – операции присваивания.

Введём следующие переменные:

1) `buf` – буфер для накопления символов лексемы;
2) `c` – очередной входной символ;
3) `d` – переменная для формирования числового значения константы;
4) `TW` – таблица ключевых слов входного языка;
5) `TD` – таблица разделителей входного языка;
6) `TID` – таблица идентификаторов анализируемой программы;
7) `TNUM` – таблица чисел-констант, используемых в программе.

Таблицы `TW` и `TD` заполнены заранее, т.к. их содержимое не зависит от исходной программы; `TID` и `TNUM` будут формироваться в процессе анализа; для простоты будем считать, что все таблицы одного типа; пусть `tab` – имя типа этих таблиц, `ptab` – указатель на `tab`.

Диаграмма состояний для лексического анализатора приведена на рисунке 11.

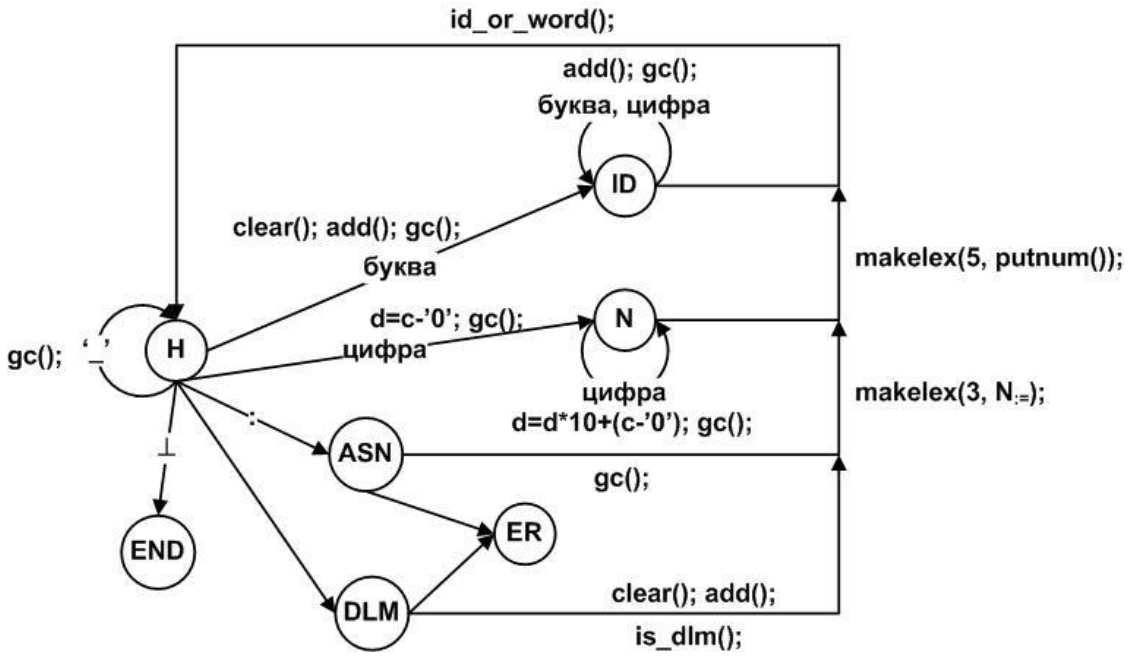


Рисунок 11. Диаграмма состояний для лексического анализатора

Символом N_x на диаграмме обозначен номер лексемы *x* в её классе.

Функции, используемые лексическим анализатором:

1) `void clear (void);` – очистка буфера `buf`;
2) `void add (void);` – добавление символа `c` в конец буфера `buf`;
3) `int look (ptab T);` – поиск в таблице `T` лексемы из буфера `buf`. Функция возвращает номер строки таблицы с информацией о лексеме либо 0, если такой лексемы в таблице `T` нет;
4) `int putl (ptab T);` – запись в таблицу `T` лексемы из буфера `buf`, если её там не было. Функция возвращает номер строки таблицы с информацией о лексеме;
5) `int putnum ();` – запись в `TNUM` константы из `d`, если её там не было. Функция возвращает номер строки таблицы `TNUM` с информацией о константе-лексеме;
6) `void makelex (int k, int i);` – формирование и вывод внутреннего представления лексемы; `k` – номер класса, `i` – номер в классе;
7) `void gc (void);` – функция, читающая из входного потока очередной символ исходной программы и заносящая его в переменную `c`;
8) `void id_or_word (void);` – функция, определяющая является ли слово в буфере `buf` идентификатором или ключевым словом и формирующая лексему соответствующего класса;
9) `void is_dlm (void);` – если символ в буфере `buf` является разделителем, то формирует соответствующую лексему, иначе производится переход в состояние ER.

Листинг 2. Лексический анализатор.

```
include <stdio.h>
#include <ctype.h>

#define BUFSIZE 80

extern ptab TW, TID, TD, TNUM;

char buf[BUFSIZE]; /* для накопления символов лексемы */
int c; /* очередной символ */
int d; /* для формирования числового значения константы */
int j; /* номер строки в таблице, где находится лексема,
          найденная функцией look */

enum state {H, ID, NUM, ASN, DLM, ER, END};
enum state TC; /* текущее состояние */

FILE* fp;

void clear(void); /* очистка буфера buf */
void add(void); /* добавление символа c в конец буфера buf */
int look(ptab); /* поиск в таблице лексемы из buf;
                   результат: номер строки таблицы либо 0 */
int putl(ptab); /* запись в таблицу лексемы из buf, если ее
                   там не было; результат: номер строки
                   таблицы */
int putnum();   /* запись в TNUM константы из d, если ее там
                   не было; результат: номер строки таблицы
                   TNUM */
void makelex(int,int); /* формирование и вывод внутреннего
                          представления лексемы */

void id_or_word(void)
{
    if (j=look(TW)) makelex(1,j);
    else
    {
       j=putl(TID); makelex(4,j);
    }
}

void is_dlm(void)
{
    if(j=look(TD))
    {
       makelex(2,j);
       gc();
       TC=H;
    }
    TC=ER;
}
```

Листинг 2. Лексический анализатор.

```
void gc(void)
{
    c = fgetc(fp);
}

void scan (void)
{
    TC = H;
    fp = fopen("prog","r"); /* в файле "prog" находится текст
                               исходной программы */
    gc();
    do
    {
       switch (TC)
       {
          case H:
             if (c == ' ') gc();
             else if (isalpha(c))
             {
                clear();
                add();
                gc();
                TC = ID;
             }
             else if (isdigit (c))
             {
                d = c - '0';
                gc();
                TC = NUM;
             }
             else if (c == ':')
             {
                gc();
                TC = ASN;
             }
             else if (c == '⊥')
             {
                makelex(2, N⊥);
                TC = END;
             }
             else TC = DLM;
             break;
          case ID:
             if (isalpha(c) || isdigit(c))
             {
                add();
                gc();
             }
             else
             {
                id_or_word();
```

Листинг 2. Лексический анализатор.

```
                TC = H;
             }
             break;
          case NUM:
             if (isdigit(c))
             {
                d=d*10+(c - '0');
                gc();
             }
             else
             {
                makelex (5, putnum());
                TC = H;
             }
             break;
          /* ........... */
       } /* конец switch */
    } /* конец тела цикла */
    while (TC != END && TC != ER);
    if (TC == ER) printf("ERROR !!!\n");
    else printf("O.K.!!!\n");
}
```

6.2 Генератор лексических анализаторов Flex

Существуют различные программные средства для решения задачи построения лексических анализаторов. Наиболее известным из них является Lex (в более поздних версиях – Flex).

Программный инструментарий Flex позволяет определить лексический анализатор с помощью регулярных выражений для описания шаблонов токенов. Входные обозначения для Flex обычно называют *языком Flex*, а сам инструмент – *компилятором Flex*. Компилятор Flex преобразует входные шаблоны в конечный автомат и генерирует код (в файле с именем lex.yy.c), имитирующий данный автомат.

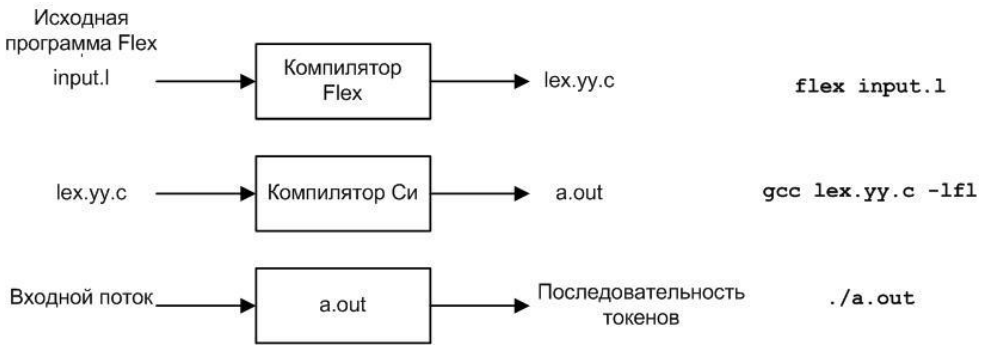


Рисунок 12. Схема использования Flex

На рисунке 12 показаны схема использования Flex и команды, соответствующие каждому этапу генерирования лексического анализатора. Входной файл `input.l` написан на языке Flex и описывает генерируемый лексический анализатор. Компилятор Flex преобразует `input.l` в программу на языке программирования Си (файл с именем `lex.yy.c`). При компиляции `lex.yy.c` необходимо прилинковать библиотеку Flex (`-lfl`). Этот файл компилируется в файл с именем `a.out` как обычно. Выход компилятора Си представляет собой работающий лексический анализатор, который на основе потока входных символов выдаёт поток токенов.

Обычно полученный лексический анализатор, используется в качестве подпрограммы синтаксического анализатора.

Структура программы на языке Flex имеет следующий вид:

```
Объявления
%%
Правила трансляции
%%
Вспомогательные функции
```

Обязательным является наличие правил трансляции, а, следовательно, и символов %% перед ними. Правила могут и отсутствовать в файле, но %% должны присутствовать всё равно.

Пример самого короткого файла на языке Flex:

```
%%
```

В этом случае входной поток просто посимвольно копируется в выходной. По умолчанию, входным является стандартный входной поток (stdin), а выходным – стандартный выходной (stdout).

Раздел объявлений может включать объявления переменных, именованные константы и регулярные определения (например, `digit [0-9]` – регулярное выражение, описывающее множество цифр от 0 до 9). Кроме того, в разделе объявлений может помещаться символьный блок, содержащий определения на Си. Символьный блок всегда начинается с `%{` и заканчивается `%}`. Весь код символьного блока полностью копируется в начало генерируемого файла исходного кода лексического анализатора.

Второй раздел содержит правила трансляции вида

```
Шаблон { Действие }
```

Каждый шаблон является регулярным выражением, которое может использовать регулярные определения из раздела объявлений. Действия представляют собой фрагменты кода, обычно написанные на языке программирования Си, хотя существуют и разновидности Flex для других языков программирования.

Третий раздел содержит различные дополнительные функции на Си, используемые в действиях. Flex копирует эту часть кода в конец генерируемого файла.

Листинг 3. Пример программы для подсчёта символов, слов и строк во введённом тексте.

```
%{
int chars = 0;
int words = 0;
int lines = 0;
%}
%%
[a-zA-Z]+ { words++; chars += strlen(yytext); }
\n        { chars++; lines++; }
.         { chars++; }
%%
int main(int argc, char **argv)
{
   yylex();
   printf("%8d%8d%8d\n", lines, words, chars);
   return 0;
}
```

В листинге 3 определены все три раздела программы на Flex.

В первом разделе объявлены три переменных-счётчика для символов, слов и строк, соответственно. Эта часть кода будет полностью скопирована в файл `lex.yy.c`.

Во втором разделе определены шаблоны токенов и действия, которые нужно выполнить при соответствии входного потока тому либо иному шаблону. **Перед шаблоном не должно быть пробелов, табуляций и т.п., поскольку Flex рассматривает любую строку, начинающуюся с пробела, как код, который нужно скопировать в файл `lex.yy.c`.**

В данном примере определены три шаблона:

1) `[a-zA-Z]+` соответствует слову текста. В соответствии с этим шаблоном слово может содержать прописные и заглавные буквы латинского алфавита. А знак + означает, что слово может состоять из одного или нескольких символов, описанных перед +. В случае совпадения входной последовательности и этого шаблона, увеличиваются счётчики для слов и символов. Массив символов `yytext` всегда содержит текст, соответствующий данному шаблону. В нашем случае он используется для расчёта длины слова;
2) `\n` соответствует символу перевода строки. В случае совпадения входного потока с данным шаблоном происходит увеличение счётчиков для символов и строк на 1;
3) `.` является шаблоном для любого входного символа.

В функции `main` вызывается `yylex()` – функция, непосредственно выполняющая лексический анализ входного текста.

Ниже приведены команды для компиляции и запуска программы на языке Flex для подсчёта символов, слов и строк в тексте, введённом с клавиатуры.

```
$ flex words.l
$ gcc lex.yy.c -lfl
$ ./a.out
To be, or not to be: that is the question
       1      10      42
```

В таблице 3 перечислены специальные символы, использующиеся в регулярных выражениях (шаблонах) Flex.

Таблица 3. Специальные символы, использующиеся в регулярных выражениях Flex.

Символ шаблона	Значение
`.`	Соответствует любому символу, кроме `\n`
`[]`	Класс символов, соответствующий любому из символов, описанных внутри скобок. Знак `'-'` указывает на диапазон символов. Например, `[0-9]` означает то же самое, что и `[0123456789]`, `[a-z]` – любая прописная буква латинского алфавита, `[A-z]` – все заглавные и прописные буквы латинского алфавита, а также 6 знаков пунктуации, находящихся между `Z` и `a` в таблице ASCII. Если символ `'-'` или `']'` указан в качестве первого символа после открывающейся квадратной скобки, значит он включается в описываемый класс символов. Управляющие (escape) последовательности языка Си также могут указываться внутри квадратных скобок, например, `\t`.
`^`	Внутри квадратных скобок используется как отрицание, например, регулярное выражение `[^\t\n]` соответствует любой последовательности символов, не содержащей табуляций и переводов строки. Если просто используется в начале шаблона, то означает начало строки.
`$`	При использовании в конце регулярного выражения означает конец строки.
`{}`	Если в фигурных скобках указаны два числа, то они интерпретируются как минимальное и максимальное количество повторений шаблона, предшествующего скобкам. Например, `A{1,3}` соответствует повторению буквы `A` от одного до трёх раз, а `0{5}` – `00000`. Если внутри скобок находится имя регулярного определения, то это просто обращение к данному определению по его имени.

Таблица 3. Специальные символы, использующиеся в регулярных выражениях Flex.

Символ шаблона	Значение
\	Используется в escape-последовательностях языка Си и для задания метасимволов, например, * – символ '*' в отличие от * (см. ниже).
*	Повторение регулярного выражения, указанного до *, 0 или более раз. Например, [\t]* соответствует регулярному выражению для пробелов и/или табуляций, отсутствующих или повторяющихся несколько раз.
+	Повторение регулярного выражения, указанного до +, один или более раз. Например, [0-9]+ соответствует строкам 1, 111 или 123456.
?	Соответствует повторению регулярного выражения, указанного до ?, 0 или 1 раз. Например, -?[0-9]+ соответствует знаковым числам с необязательным минусом перед числом.
|	Оператор «или». Например, true|false соответствует любой из двух строк.
()	Используются для группировки нескольких регулярных выражений в одно. Например, a(bc|de) соответствует входным последовательностям: abc или ade.
/	Так называемый присоединенный контекст. Например, регулярное выражение 0/1 соответствует 0 во входной строке 01, но не соответствует ничему в строках 0 или 02.
" "	Любое символы в кавычках рассматриваются как строка символов. Метасимволы, такие как *, теряют своё значение и интерпретируются как два символа: \ и *.

Лексический анализатор на языке Flex для грамматики из раздела 6.1 представлен в листинге 4.

Листинг 4. Лексический анализатор на языке Flex

```
%option noyywrap yylineno
%{
#include <stdio.h>
#include <string.h>
#define SIZE 6
char* filename;
char* keywords[SIZE] = {"if", "then", "else", "and",
                        "or", "xor"};
%}

letter [a-zA-Z]
digit [0-9]
delim [();]
ws [ \t\n]

%%

({letter}|"_")({letter}|{digit}|"_")* {
    if(resWord(yytext))
    {
      printf("%s:%d KEYWORD %s\n", filename, yylineno, yytext);
    }else{
      printf("%s:%d IDENTIFIER %s\n", filename, yylineno,
             yytext);
    }
}
{digit}+ {
      printf("%s:%d NUMBER %s\n", filename, yylineno, yytext);
}
":=" {
      printf("%s:%d ASSIGN %s\n", filename, yylineno, yytext);
}
{delim} {
      printf("%s:%d DELIMITER %s\n", filename, yylineno,
             yytext);
}
{ws}+ ;
. {
      printf("%s:%d Unknown character '%s'\n", filename,
             yylineno, yytext);
}
%%
int resWord(char* id)
{
   int i;
   for(i = 0; i < SIZE; i++)
   {
      if(strcmp(id, keywords[i]) == 0)
      {
         return 1;
      }
```

Листинг 4. Лексический анализатор на языке Flex

```
    }
    return 0;
}
int main(int argc, char** argv)
{
    if(argc < 2)
    {
        perror("Input file name is not specified");
        return 1;
    }
    yyin = fopen(argv[1], "r");
    if(yyin == NULL)
    {
        perror(argv[1]);
        return 1;
    }
    filename = strdup(argv[1]);
    yylineno = 1;
    yylex();
    return 0;
}
```

В первой строке данной программы указаны опции, которые должны быть учтены при построении лексического анализатора. Для этого используется формат

```
%option имя_опции
```

Те же самые опции можно было бы указать при компиляции в командной строке как

```
--имя_опции
```

Для отключения опции перед её именем следует указать «no», как в случае с `noyywrap`. Полный список допустимых опций можно найти в документации по Flex [6, 8].

Первые версии генератора лексических анализаторов Lex вызывали функцию `yywrap()` при достижении конца входного потока `yyin`. В случае, если нужно было продолжить анализ входного текста из другого файла, `yywrap` возвращала 0 для продолжения сканирования. В противном случае возвращалась 1.

В современных версиях Flex рекомендуется отключать использование `yywrap` с помощью опции `noyywrap`, если в программе на языке Flex есть своя функция `main`, в которой и определяется, какой файл и когда сканировать.

Использование опции `yylineno` позволяет вести нумерацию строк входного файла и в случае ошибки сообщать пользователю номер строки, в которой эта ошибка произошла. Flex определяет переменную `yylineno` и автоматически увеличивает её значение на 1, когда встречается символ `'\n'`. При этом Flex не инициализирует эту переменную. Поэтому в функции `main` перед

вызовом функции лексического анализа `yylex` переменной `yylineno` присваивается 1.

Flex, по умолчанию, присваивает переменной `yyin` указатель на стандартный поток ввода. Если предполагается сканировать текст из файла, то нужно присвоить переменной `yyin` результат вызова функции `fopen` до вызова `yylex`:

```
yyin = fopen(argv[1], "r");
```

В функции `main` в приведённом примере открывается файл, имя которого было указано пользователем при вызове лексического анализатора.

Входной текстовый файл содержит следующую программу, соответствующую правилам грамматики из раздела 6.1.

```
_A := 5;
B := 0;
if (_A and B) then
_A := B
else if (_A xor B) then
B := _A
else
_A := _A or B;
```

Для компиляции и запуска программы используются следующие команды:

```
flex example.l
gcc lex.yy.c -lfl
./a.out
```

Результат выполнения:

```
program:1 IDENTIFIER _A
program:1 ASSIGN :=
program:1 NUMBER 5
program:1 DELIMITER ;
program:2 IDENTIFIER B
program:2 ASSIGN :=
program:2 NUMBER 0
program:2 DELIMITER ;
program:3 KEYWORD if
program:3 DELIMITER (
program:3 IDENTIFIER _A
program:3 KEYWORD and
program:3 IDENTIFIER B
program:3 DELIMITER )
program:3 KEYWORD then
program:4 IDENTIFIER _A
program:4 ASSIGN :=
program:4 IDENTIFIER B
```

```
program:5 KEYWORD else
program:5 KEYWORD if
program:5 DELIMITER (
program:5 IDENTIFIER _A
program:5 KEYWORD xor
program:5 IDENTIFIER B
program:5 DELIMITER )
program:5 KEYWORD then
program:6 IDENTIFIER B
program:6 ASSIGN :=
program:6 IDENTIFIER _A
program:7 KEYWORD else
program:8 IDENTIFIER _A
program:8 ASSIGN :=
program:8 IDENTIFIER _A
program:8 KEYWORD or
program:8 IDENTIFIER B
program:8 DELIMITER ;
```

Контрольные вопросы

1. Чем различаются таблица лексем и таблица идентификаторов? В какую из этих таблиц лексический анализатор должен помещать ключевые слова, разделители и знаки операций?
2. Какую роль выполняет лексический анализ в процессе компиляции?
3. Как связаны лексический и синтаксический анализ?
4. Какие проблемы необходимо решить при построении лексического анализатора на основе конечного автомата?
5. Расскажите о структуре программы на языке Flex. Приведите пример самой короткой программы на этом языке.
6. Что представляют собой шаблоны и действия, использующиеся во втором разделе программ на языке Flex?

ГЛАВА 7
СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР

В иерархии грамматик Хомского выделено 4 основных группы языков (и описывающих их грамматик). При этом наибольший интерес представляют регулярные и контекстно-свободные (КС) грамматики и языки. Они используются при описании синтаксиса языков программирования. С помощью регулярных грамматик можно описать лексемы языка – идентификаторы, константы, служебные слова и прочие. На основе КС-грамматик строятся более крупные синтаксические конструкции: описания типов и переменных, арифметические и логические выражения, управляющие операторы, и, наконец, полностью вся программа на входном языке.

Входные цепочки регулярных языков распознаются с помощью конечных автоматов (КА). Они лежат в основе сканеров, выполняющих лексический анализ и выделение слов в тексте программы на входном языке. Результатом работы сканера является преобразование исходной программы в список или таблицу лексем. Дальнейшую её обработку выполняет другая часть компилятора – синтаксический анализатор. Его работа основана на использовании правил КС-грамматики, описывающих конструкции исходного языка.

Синтаксический анализатор – это часть компилятора, которая отвечает за выявление и проверку синтаксических конструкций входного языка. Синтаксический анализатор получает строку токенов от лексического анализатора, как показано на рисунке 13, и проверяет, может ли эта строка токенов порождаться грамматикой входного языка. Ещё одной функцией синтаксического анализатора является генерация сообщений обо всех выявленных ошибках, причём достаточно внятных и полных, а кроме того, синтаксический анализатор должен уметь обрабатывать обычные, часто встречающиеся ошибки и продолжать работу с оставшейся частью программы. В случае корректной программы синтаксический анализатор строит дерево разбора и передаёт его следующей части компилятора для дальнейшей обработки.

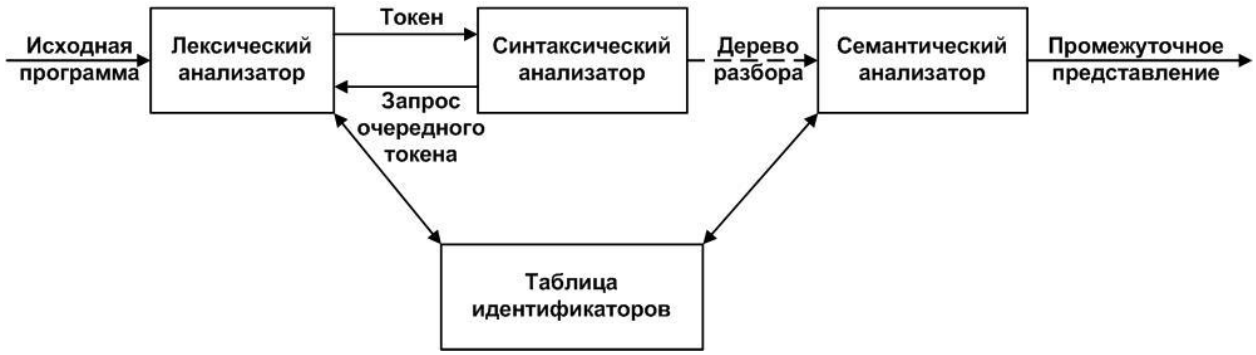


Рисунок 13. Место синтаксического анализатора в структуре компилятора

7.1 Распознавание цепочек КС-языков

Рассмотрим алгоритмы, лежащие в основе синтаксического анализа. Перед синтаксическим анализатором стоят две основные задачи: проверить пра-

вильность конструкций программы, которая представляется в виде уже выделенных слов входного языка, и преобразовать её в вид, удобный для дальнейшей семантической (смысловой) обработки и генерации кода. Одним из способов такого представления является дерево синтаксического разбора.

Основой для построения распознавателей КС-языков являются **автоматы с магазинной памятью** – МП-автоматы – односторонние недетерминированные распознаватели с линейно-ограниченной магазинной памятью.

МП-автомат в отличие от обычного КА имеет стек (магазин), в который можно помещать специальные «магазинные» символы (обычно это терминальные и нетерминальные символы грамматики языка). Переход МП-автомата из одного состояния в другое зависит не только от входного символа, но и от одного или нескольких верхних символов стека. Таким образом, конфигурация автомата определяется тремя параметрами: состоянием автомата, текущим символом входной цепочки (положением указателя в цепочке) и содержимым стека.

При выполнении перехода МП-автомата из одной конфигурации в другую из стека удаляются верхние символы, соответствующие условию перехода, и добавляется цепочка, соответствующая правилу перехода. Первый символ цепочки становится верхушкой стека. Допускаются переходы, при которых входной символ игнорируется (и тем самым он будет входным символом при следующем переходе). Эти переходы называются **ε-переходами**. Если при окончании цепочки автомат находится в одном из заданных конечных состояний, а стек пуст, цепочка считается принятой (после окончания цепочки могут быть сделаны ε-переходы). Иначе цепочка символов не принимается.

МП-автомат называется *недетерминированным,* если при одной и той же его конфигурации возможен более чем один переход. В противном случае (если из любой конфигурации МП-автомата по любому входному символу возможно не более одного перехода в следующую конфигурацию) МП-автомат считается *детерминированным* (ДМП-автоматом). ДМП-автоматы задают класс детерминированных КС-языков, для которых существуют однозначные КС-грамматики. Именно этот класс языков лежит в основе синтаксических конструкций всех языков программирования, так как любая синтаксическая конструкция языка программирования должна допускать только однозначную трактовку.

По произвольной КС-грамматике **G** (**VN**, **VT**, **P**, *S*), **V** = **VT** ∪ **VN** всегда можно построить недетерминированный МП-автомат, который допускает цепочки языка, заданного этой грамматикой. А на основе этого МП-автомата можно создать распознаватель для заданного языка.

Однако при алгоритмической реализации функционирования такого распознавателя могут возникнуть проблемы. Дело в том, что построенный МП-автомат будет, как правило, недетерминированным, а для МП-автоматов, в отличие от обычных КА, не существует алгоритма, который позволял бы преобразовать произвольный МП-автомат в ДМП-автомат. Поэтому программирование функционирования МП-автомата – нетривиальная задача. Если моделировать его функционирование по шагам с перебором всех возможных состояний, то может оказаться, что построенный для тривиального МП-автомата алгоритм

никогда не завершится на конечной входной цепочке символов при определённых условиях.

Поэтому для построения распознавателя для языка, заданного КС-грамматикой, рекомендуется воспользоваться соответствующим математическим аппаратом и одним из существующих алгоритмов.

7.2 Виды распознавателей для КС-языков

Существуют несложные преобразования КС-грамматик, выполнение которых гарантирует, что построенный на основе преобразованной грамматики МП-автомат можно будет промоделировать за конечное время на основе конечных вычислительных ресурсов.

Эти преобразования позволяют строить два основных типа простейших распознавателей:

- распознаватель с подбором альтернатив;
- распознаватель на основе алгоритма «сдвиг-свёртка».

Работу **распознавателя с подбором альтернатив** можно неформально описать следующим образом:

1) если на верхушке стека МП-автомата находится нетерминальный символ A, то его можно заменить на цепочку символов α при условии, что в грамматике языка есть правило $A \to \alpha$, не сдвигая при этом считывающую головку автомата (этот шаг работы называется «подбор альтернативы»);
2) если же на верхушке стека находится терминальный символ a, который совпадает с текущим символом входной цепочки, то этот символ можно выбросить из стека и передвинуть считывающую головку на одну позицию вправо (этот шаг работы называется «выброс»).

Данный МП-автомат может быть недетерминированным, поскольку при подборе альтернативы в грамматике языка может оказаться более одного правила вида $A \to \alpha$, тогда функция $\delta(q, \varepsilon, A)$ будет содержать более одного следующего состояния – у МП-автомата будет несколько альтернатив.

Решение о том, выполнять ли на каждом шаге работы МП-автомата выброс или подбор альтернативы, принимается однозначно. Моделирующий алгоритм должен обеспечивать выбор одной из возможных альтернатив и хранение информации о том, какие альтернативы и на каком шаге уже были выбраны, чтобы иметь возможность вернуться к этому шагу и подобрать другие альтернативы.

Распознаватель с подбором альтернатив является нисходящим распознавателем: он читает входную цепочку символов слева направо и строит левосторонний вывод. Название «нисходящий» дано ему потому, что дерево вывода в этом случае следует строить сверху вниз, от корня к концевым вершинам («листьям»).

Работу **распознавателя на основе алгоритма «сдвиг-свёртка»** можно описать так:

1) если на верхушке стека МП-автомата находится цепочка символов γ, то её можно заменить на нетерминальный символ A при условии, что в

грамматике языка существует правило вида $A \to \gamma$, не сдвигая при этом считывающую головку автомата (этот шаг работы называется «свёртка»);

2) с другой стороны, если считывающая головка автомата обозревает некоторый символ a входной цепочки, то его можно поместить в стек, сдвинув при этом головку на одну позицию вправо (этот шаг работы называется «сдвиг» или «перенос»).

Этот распознаватель потенциально имеет больше неоднозначностей, чем рассмотренный выше распознаватель, основанный на алгоритме подбора альтернатив. На каждом шаге работы автомата надо решать следующие вопросы:

- что необходимо выполнять: сдвиг или свёртку;
- если выполнять свёртку, то какую цепочку γ выбрать для поиска правил (цепочка γ должна встречаться в правой части правил грамматики);
- какое правило выбрать для свёртки, если окажется, что существует несколько правил вида $A \to \gamma$ (несколько правил с одинаковой правой частью).

Для моделирования работы этого расширенного МП-автомата надо на каждом шаге запоминать все предпринятые действия, чтобы иметь возможность вернуться к уже сделанному шагу и выполнить эти же действия по-другому. Этот процесс должен повторяться до тех пор, пока не будут перебраны все возможные варианты. Распознаватель на основе алгоритма «сдвиг-свёртка» является восходящим распознавателем: он читает входную цепочку символов слева направо и строит правосторонний вывод. Название «восходящий» дано ему потому, что дерево вывода в этом случае следует строить снизу вверх, от концевых вершин к корню.

Функционирование обоих рассмотренных распознавателей реализуется достаточно простыми алгоритмами. Однако оба они имеют один существенный недостаток – время их функционирования экспоненциально зависит от длины входной цепочки $n = |\alpha|$, что недопустимо для компиляторов, где длина входных программ составляет от десятков до сотен тысяч символов. Так происходит потому, что оба алгоритма выполняют разбор входной цепочки символов методом простого перебора, подбирая правила грамматики произвольным образом, а в случае неудачи возвращаются к уже прочитанной части входной цепочки и пытаются подобрать другие правила.

Существуют более эффективные табличные распознаватели, построенные на основе алгоритмов Эрли и Кока-Янгера-Касами. Они обеспечивают полиномиальную зависимость времени функционирования от длины входной цепочки (n^3 – для произвольного МП-автомата и n^2 – для ДМП-автомата). Это самые эффективные из универсальных распознавателей для КС-языков. Но и полиномиальную зависимость времени разбора от длины входной цепочки нельзя признать удовлетворительной.

Лучших универсальных распознавателей не существует. Однако среди всего типа КС-языков существует множество классов и подклассов языков, для которых можно построить распознаватели, имеющие линейную зависимость времени функционирования от длины входной цепочки символов. Такие распознаватели называют *линейными* распознавателями КС-языков.

В настоящее время известно множество линейных распознавателей и соответствующих им классов КС-языков. Каждый из них имеет свой алгоритм функционирования, но все известные алгоритмы являются модификацией двух базовых алгоритмов – алгоритма с подбором альтернатив и алгоритма «сдвиг-свёртка», рассмотренных выше. Модификации заключаются в том, что алгоритмы выполняют подбор правил грамматики для разбора входной цепочки символов не произвольным образом, а руководствуясь установленным порядком, который создаётся заранее на основе заданной КС-грамматики. Такой подход позволяет избежать возвратов к уже прочитанной части цепочки и существенно сокращает время, требуемое на её разбор.

Среди всего множества можно выделить следующие наиболее часто используемые распознаватели:

1. распознаватели на основе рекурсивного спуска (модификация алгоритма с подбором альтернатив);
2. распознаватели на основе LL(1)- и LL(k)-грамматик (модификация алгоритма с подбором альтернатив);
3. распознаватели на основе LR(0)- и LR(1)-грамматик (модификация алгоритма «сдвиг-свёртка»);
4. распознаватели на основе SLR(1)- и LALR(1)-грамматик (модификация алгоритма «сдвиг-свёртка»);
5. распознаватели на основе грамматик предшествования (модификация алгоритма «сдвиг-свёртка»).

7.3 Алгоритмы нисходящего синтаксического анализа

Нисходящий синтаксический анализ можно рассматривать как задачу построения дерева разбора для входной строки, начиная с корня и создавая узлы дерева разбора в прямом порядке обхода. Или, что то же самое, нисходящий синтаксический анализ можно рассматривать как поиск левого порождения входной строки.

7.3.1 Метод рекурсивного спуска

Программа синтаксического анализа методом рекурсивного спуска состоит из набора процедур, по одной для каждого нетерминала. Работа программы начинается с вызова процедуры для стартового символа и успешно заканчивается в случае сканирования всей входной строки. Псевдокод для типичного нетерминала приведён в листинге 5.

Рекурсивный спуск в общем случае может потребовать выполнения возврата, т.е. повторения сканирования входного потока. Однако при анализе синтаксических конструкций языков программирования возврат требуется редко, так что встреча с синтаксическим анализатором с возвратом – явление не частое.

Листинг 5. Псевдокод метода рекурсивного спуска

```
1.  void S()
2.  {
3.      Выбираем S-правило S → X1X2 … Xk;
4.      for (i от 1 до k )
5.      {
6.        if (Xi — нетерминал)
7.            Вызов процедуры Xi();
8.        else if (Xi равно текущему входному символу)
9.            Переходим к следующему входному символу;
10.       else /* Обнаружена ошибка */
11.     }
12. }
```

Чтобы разрешить возврат, псевдокод в листинге 5 должен быть немного модифицирован. Во-первых, невозможно выбрать единственное правило в строке 1, поэтому нужно проверять каждое из нескольких правил в некотором порядке. Во-вторых, ошибка в строке 10 не является окончательной и предполагает возврат к строке 1 и проверку другого правила. Объявлять о найденной во входной строке ошибке можно только в том случае, если больше не имеется непроверенных правил. Чтобы быть в состоянии проверить новое правило, нужно иметь возможность сбросить указатель входного потока в состояние, в котором он находился при первом достижении строки 1. Таким образом, для хранения этого указателя входного потока требуется локальная переменная.

Метод рекурсивного спуска применим к достаточно узкому подклассу КС-грамматик. Известны более широкие подклассы КС-грамматик, для которых существуют эффективные анализаторы, обладающие тем же свойством, что и анализатор, написанный методом рекурсивного спуска, – входная цепочка считывается один раз слева направо и процесс разбора полностью детерминирован. К таким грамматикам относятся LL(k)-грамматики, LR(k)-грамматики, грамматики предшествования и некоторые другие.

7.3.2 LL(k)-грамматики

Грамматика обладает свойством LL(k) (англ. Left Left – цепочка символов читается слева направо и формируется левосторонний вывод), $k > 0$, если на каждом шаге вывода для однозначного выбора очередной альтернативы достаточно знать символ на верхушке стека и рассмотреть первые k символов от текущего положения во входной цепочке символов.

Для LL(k)-грамматик известны следующие свойства:

- всякая LL(k)-грамматика для любого $k > 0$ является однозначной;
- существует алгоритм, позволяющий проверить, является ли заданная грамматика LL(k)-грамматикой для строго определённого k.

Все грамматики, допускающие разбор по методу рекурсивного спуска, являются подклассом LL(1)-грамматик.

Алгоритм разбора входных цепочек для LL(k)-грамматики носит название «k-предсказывающего алгоритма».

Рассмотрим алгоритм разбора входных цепочек на примере LL(1)-грамматик. Для каждого нетерминального символа LL(1)-грамматики не может быть двух правил, начинающихся с одного и того же терминального символа. Для удовлетворения этого условия необходимо удалить левую рекурсию в правилах грамматики (см. алгоритм устранения левой рекурсии в разделе 2.7).

Для построения распознавателя языка, заданного LL(1)-грамматикой, нам понадобятся два множества:

- **FIRST**(A) – множество терминальных символов, которыми начинаются цепочки, выводимые из A в грамматике **G** (**VT**, **VN**, **P**, S), т.е. **FIRST**(A) = $\{a \in \mathbf{VT} \mid A \to a\alpha, A \in \mathbf{VN}, \alpha \in (\mathbf{VT} \cup \mathbf{VN})^*\}$;
- **FOLLOW**(A) – множество терминальных символов, которые следуют за цепочками, выводимыми из A в грамматике **G** (**VT**, **VN**, **P**, S), т.е. **FOLLOW**(A) = $\{a \in \mathbf{VT} \mid S \to \alpha A \beta, \beta \to a\gamma, A \in \mathbf{VN}, \alpha, \beta, \gamma \in (\mathbf{VT} \cup \mathbf{VN})^*\}$.

Например, пусть в грамматике присутствуют следующие правила:

$S \to \alpha A a \beta$

$A \to c\gamma$

Тогда множество **FIRST**(A) = $\{c\}$, а множество **FOLLOW**(A) = $\{a\}$.

Чтобы вычислить **FIRST**(X) для всех нетерминальных символов грамматики X, будем применять следующие правила до тех пор, пока ни к одному из множеств **FIRST** не смогут быть добавлены ни терминалы, ни ε.

Если X – нетерминал и имеется правило $X \to Y_1\ Y_2\ ...\ Y_k$ для некоторого $k \geq 1$, то поместим a в **FIRST**(X), если для некоторого i $a \in$ **FIRST**(Y_i) и ε входит во все множества **FIRST**(Y_1), ..., **FIRST**(Y_{i-1}), т.е. $Y_1 ... Y_{i-1} \Rightarrow^*$ ε. Если ε входит в **FIRST**(Y_j) для всех j = 1, 2, ..., k, то добавляем ε к **FIRST**(X). Например, всё, что находится во множестве **FIRST**(Y_1), есть и во множестве **FIRST**(X). Если Y_1 не порождает ε, то больше мы ничего не добавляем к **FIRST**(X), но если $Y_1 \Rightarrow^*$ ε, то к **FIRST**(X) добавляется **FIRST**(Y_2) и т.д.

Если имеется правило $X \to$ ε, добавим ε к **FIRST**(X).

Чтобы вычислить **FOLLOW**(A) для всех нетерминалов A, будем применять следующие правила до тех пор, пока ни к одному множеству **FOLLOW** нельзя будет добавить ни одного символа.

1. Поместим ⊥ в **FOLLOW**(S), где S – целевой символ, а ⊥ – правый ограничитель входного потока.
2. Если имеется правило $A \to \alpha B \beta$, то все элементы множества **FIRST**(β), кроме ε, помещаются во множество **FOLLOW**(B).
3. Если имеется правило $A \to \alpha B$ или $A \to \alpha B \beta$, где **FIRST**(β) содержит ε, то все элементы из множества **FOLLOW**(A) помещаются во множество **FOLLOW**(B).

Приведённый далее алгоритм собирает информацию из множеств **FIRST** и **FOLLOW** в таблицу предиктивного синтаксического анализа $M[A, a]$ (представляющую собой двумерный массив), где A – нетерминал, а a – терминал или

символ ⊥ (маркер конца входного потока). Алгоритм основан на следующей идее: если очередной входной символ *a* находится во множестве **FIRST**(*A*), выбирается правило $A \to \alpha$. Единственная сложность возникает при $\alpha = \varepsilon$ или, в общем случае, когда $\alpha \Rightarrow^* \varepsilon$. В этом случае мы снова должны выбрать $A \to \alpha$, если текущий входной символ имеется в **FOLLOW**(*A*) или если из входного потока получен ⊥, который при этом входит в **FOLLOW**(*A*).

Алгоритм построения таблицы предиктивного синтаксического анализа

Вход: грамматика **G**.

Выход: таблица синтаксического анализа *M*.

Метод: для каждого правила грамматики $A \to \alpha$ выполняем следующие действия.

1. Для каждого терминала *a* из **FIRST**(*A*) добавляем $A \to \alpha$ в ячейку *M*[*A*, *a*].

2. Если $\varepsilon \in$ **FIRST**(*A*), то для каждого терминала *b* из **FOLLOW**(*A*) добавляем $A \to \alpha$ в *M*[*A*, *b*]. Если $\varepsilon \in$ **FIRST**(*A*) и $\perp \in$ **FOLLOW**(*A*), то добавляем $A \to \alpha$ также и в *M*[*A*, ⊥].

Если после выполнения этих действий ячейка *M*[*A*, *a*] осталась без правила, устанавливаем её значение равным *error* (это значение обычно представляется пустой записью таблицы).

Пример 12:

$E \to TE'$

$E' \to +TE' \mid \varepsilon$

$T \to FT'$

$T' \to *FT' \mid \varepsilon$

$F \to (E) \mid \mathbf{i}$

1. **FIRST**(*F*) = **FIRST**(*T*) = **FIRST**(*E*) = {(, ***i***}
2. **FIRST**(*E*') = {+, ε}
3. **FIRST**(*T*') = {*, ε}
4. **FOLLOW**(*E*) = **FOLLOW**(*E*') = {**)**, ⊥}
5. **FOLLOW**(*T*) = **FOLLOW**(*T*') = {+,), ⊥}
6. **FOLLOW**(*F*) = {+, *,), ⊥}

Таблица 4. Таблица предиктивного синтаксического анализа

Нетерминал	**Входной символ**					
	i	+	*	(	)	⊥
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to \mathbf{i}$			$F \to (E)$		

7.4 Алгоритмы восходящего синтаксического анализа

Восходящий синтаксический анализ соответствует построению дерева разбора для входной строки, начиная с листьев (снизу) и идя по направлению к корню (вверх).

Можно рассматривать восходящий синтаксический анализ как процесс «свёртки» строки ω к целевому символу грамматики. На каждом шаге свёртки определённая подстрока, соответствующая телу правила, заменяется нетерминалом из левой части этого правила.

Ключевые решения, принимаемые в процессе восходящего синтаксического анализа, – когда выполнять свёртку и какое правило применять.

Так как по определению свёртка представляет собой шаг, обратный порождению, то цель восходящего синтаксического анализа состоит в построении порождения в обратном порядке.

7.4.1 Алгоритм «сдвиг-свёртка»

Алгоритм «сдвиг-свёртка» представляет собой алгоритм восходящего анализа, в котором для хранения символов грамматики используется стек, а для хранения остающейся непроанализированной части входной строки – входной буфер.

Будем использовать символ ⊥ для маркирования дна стека и правого конца входной строки. При рассмотрении восходящего анализа удобно располагать вершину стека справа (а не слева, как это делалось при рассмотрении нисходящего синтаксического анализа). Изначально стек пуст, а во входном буфере находится строка ω:

Стек	Вход
⊥	ω⊥

В процессе сканирования входной строки слева направо синтаксический анализатор выполняет нуль или несколько переносов символов в стек, пока не будет готов выполнить свёртку строки β символов грамматики на вершине стека. Затем он выполняет свёртку β к заголовку соответствующего правила. Синтаксический анализатор повторяет этот цикл до тех пор, пока не будет обнаружена ошибка или пока стек не будет содержать только целевой символ, а входной буфер будет при этом пуст:

Стек	Вход
⊥S	⊥

Достигнув указанной конфигурации, синтаксический анализатор останавливается и сообщает об успешном завершении анализа.

Рассмотрим, например, грамматику со следующими правилами:

$E \to E + T \mid E - T \mid T$

$T \to T * F \mid T / F \mid F$

$F \to (E) \mid \mathbf{id}$

В таблице 5 показаны действия анализатора, выполняющего синтаксический анализ строки ***id* * *id*** по алгоритму «сдвиг-свёртка».

Анализатор, работающий по данному алгоритму, может выполнять четыре действия:

1) *сдвиг* (англ. shift). Перенос очередного входного символа на вершину стека;
2) *свёртка* (англ. reduction). Правая часть сворачиваемой строки должна располагаться на вершине стека. Определяется левый конец строки в стеке и принимается решение о том, каким нетерминалом будет заменена строка;
3) *принятие* (англ. accept). Объявление об успешном завершении синтаксического анализа;
4) *ошибка* (англ. error). Обнаружение синтаксической ошибки и выполнение восстановления после ошибки.

Таблица 5. Действия анализатора, выполняющего синтаксический анализ строки ***id* * *id***

Стек	**Вход**	**Действие**
$\perp$	***id* * *id***$\perp$	Сдвиг
$\perp$***id***	* ***id***$\perp$	Свёртка ($F \to$ ***id***)
$\perp F$	* ***id***$\perp$	Свёртка ($T \to F$)
$\perp T$	* ***id***$\perp$	Сдвиг
$\perp T$ *	***id***$\perp$	Сдвиг
$\perp T$ * ***id***	$\perp$	Свёртка ($F \to$ ***id***)
$\perp T * F$	$\perp$	Свёртка ($T \to T * F$)
$\perp T$	$\perp$	Свёртка ($E \to T$)
$\perp E$	$\perp$	Принятие

Имеются контекстно-свободные грамматики, для которых алгоритм «сдвиг-свёртка» неприменим. Любой анализатор для такой грамматики может достичь конфигурации, в которой синтаксический анализатор, обладая информацией о содержимом стека и очередных k входных символах, не может принять решение о том, следует ли выполнить сдвиг или свёртку (**конфликт «сдвиг-свёртка»**) либо какое именно из нескольких приведений должно быть выполнено (**конфликт «свёртка-свёртка»**). Эти грамматики не относятся к классу LR(k)-грамматик (см. раздел 7.4.2 об LR(k)-грамматиках).

Примером такой грамматики может служить следующая неоднозначная грамматика:

stmt $\to$ ***if*** *expr* ***then*** *stmt*
| ***if*** *expr* ***then*** *stmt* ***else*** *stmt*
| ***other***

Если анализатор находится в конфигурации

Стек	Вход
if *expr* ***then*** *stmt*	***else*** • • • ⊥

то мы сталкиваемся с конфликтом «сдвиг-свёртка». В зависимости от того, что следует за ***else*** во входном потоке, верным решением может оказаться свёртка ***if*** *expr* ***then*** *stmt в stmt* или перенос ***else*** и поиск ещё одного *stmt* для завершения альтернативы ***if*** *expr* ***then*** *stmt* ***else*** *stmt*. Решением данного конфликта может служить связывание ***else*** с предыдущим ***then***, которому ещё не найдено соответствующее ***else*** (что и реализовано, например, в генераторе синтаксических анализаторов Bison, см. раздел 7.5).

7.4.2 LR(*k*)-грамматики

КС-грамматика обладает свойством LR(*k*), $k \geq 0$, если на каждом шаге вывода для однозначного решения вопроса о выполняемом действии в алгоритме «сдвиг-свёртка» достаточно знать содержимое верхней части стека и рассмотреть первые *k* символов от текущего положения во входной цепочке символов. Название «LR(*k*)» имеет определённый смысл: «L» (от англ. left) обозначает порядок чтения входной цепочки символов слева направо, «R» (от англ. right) означает, что в результате работы распознавателя получается правосторонний вывод. Вместо «*k*» в названии стоит число, которое показывает, сколько символов входной цепочки надо рассмотреть, чтобы принять решение о сдвиге или свёртке. Существуют классы LR(0)-, LR(1)-грамматик и др.

Для LR(*k*)-грамматик известны следующие свойства:

- всякая LR(*k*)-грамматика для любого $k \geq 0$ является однозначной;
- существует алгоритм, позволяющий проверить, является ли заданная грамматика LR(*k*)-грамматикой для строго определённого *k*.

Обычно используемые в компиляции грамматики принадлежат к классу LR(1), т.е. выполняется предпросмотр не более одного символа.

7.4.2.1 LR(0)-грамматики

Простейшим случаем LR(*k*)-грамматик являются LR(0)-грамматики. При k = 0 распознаватель совсем не принимает во внимание текущий символ во входной последовательности. Решение о выполняемом действии принимается только на основании содержимого стека.

LR-анализатор принимает решение о выборе «сдвиг-свёртка», поддерживая состояния, которые отслеживают, где именно в процессе синтаксического анализа он находится. Состояния представляют собой множества *пунктов (ситуаций)*.

LR(0)-пункт грамматики – это правило этой грамматики с точкой в некоторой позиции правой части. Следовательно, правило $A \to XYZ$ даёт четыре пункта:

$A \to \bullet XYZ$

$A \to X\bullet YZ$

$A \to XY\bullet Z$

$A \to XYZ\bullet$

Правило $A \to \varepsilon$ генерирует единственный пункт $A \to \bullet$.

Можно сказать, что пункт указывает какую часть правила, мы уже просмотрели в данной точке в процессе синтаксического анализа.

Один набор множеств LR(0)-пунктов, именуемый **каноническим набором**, обеспечивает основу для построения детерминированного конечного автомата, который используется для принятия решений в процессе синтаксического анализа. Такой автомат называется **LR(0)-автоматом**. Каждое состояние LR(0)-автомата представляет собой множество пунктов в каноническом наборе.

Для построения канонического набора мы определяем расширенную грамматику и два множества: **CLOSURE** и **GOTO**. Если **G** – грамматика с целевым символом *S*, то расширенная грамматика **G'** представляет собой **G** с новым целевым символом *S'* и дополнительным правилом $S' \to S$. Назначение этого нового правила – указать синтаксическому анализатору, когда следует прекратить анализ и сообщить о принятии входной строки; т.е. принятие осуществляется тогда и только тогда, когда синтаксический анализатор выполняет свёртку с использованием правила $S' \to S$.

Замыкание пунктов

Пусть **I** – множество пунктов грамматики **G**, тогда **CLOSURE(I)** представляет собой множество пунктов, построенное из **I** согласно двум правилам.

1. Изначально в **CLOSURE(I)** добавляются все пункты из **I.**

2. Если $A \to \alpha\bullet B\beta$ входит в **CLOSURE(I)**, и существует правило $B \to \gamma$, то в **CLOSURE (I)** добавляется пункт $B \to \bullet\gamma$, если его там ещё нет. Это правило применяется до тех пор, пока не останется пунктов, которые могут быть добавлены в **CLOSURE (I)**.

Пример 13. Преобразуем ранее рассмотренную грамматику в расширенную грамматику:

$E' \to E$
$E \to E + T \mid E - T \mid T$
$T \to T * F \mid T / F \mid F$
$F \to (E) \mid \mathbf{id}$

Рассмотрим, как вычисляется замыкание. $E' \to \bullet E$ помещается в **CLOSURE(I)** согласно правилу 1. Поскольку непосредственно справа от точки находится *E*, мы добавляем правила для *E* с точками слева:
$E \to \bullet E + T$, $E \to \bullet E - T$ и $E \to \bullet T$.

Теперь справа от точки в последнем правиле находится *T*, так что следует добавить $T \to \bullet T * F$, $T \to \bullet T / F$ и $T \to \bullet F$. Далее, *F* справа от точки заставляет добавить $F \to \bullet(E)$ и $F \to \bullet\mathbf{id}$. Больше нельзя добавить никакие другие пункты.

Множество переходов

Вторым полезным множеством является **GOTO(I, *X*)**, где **I** – множество пунктов, а *X* – грамматический символ. **GOTO(I, *X*)** определяется как замыкание множества всех пунктов $A \to \alpha X\bullet\beta$, таких, что $A \to \alpha\bullet X\beta$ находится в **I.**

Можно сказать, что множество **GOTO** используется для определения переходов в LR(0)-автомате. Состояния автомата соответствуют множествам пунктов, и **GOTO(I**, *X*) указывает переход из состояния **I** при входном символе *X*.

Пример 14. Если **I** содержит пункт $E \to E\bullet + T$, то **GOTO(I**, +) включает в себя следующие пункты:

$E \to E + \bullet T$
$T \to \bullet T * F$
$T \to \bullet T / F$
$T \to \bullet F$
$F \to \bullet(E)$
$F \to \bullet \mathbf{id}$

GOTO(I, +) вычисляется путём рассмотрения пунктов **I**, в которых + следует непосредственно за точкой. Таковым является пункт $E \to E\bullet + T$. Поэтому мы переносим точку за +, получая пункт $E \to E + \bullet T$, а затем находим замыкание этого множества из одного элемента.

LR(0)-автомат для грамматики из примера 13 приведён на рисунке 14.

Таблица синтаксического анализа состоит из двух частей: функции действий синтаксического анализа ACTION и функции переходов GOTO.

1. Функция ACTION принимает в качестве аргумента состояние *i* и терминал *a* (или ⊥, маркер конца входной строки). Значение ACTION [*i*, *a*] может быть одного из следующих видов:

   а) сдвиг *j*, где *j* – состояние. Действие, предпринимаемое синтаксическим анализатором, эффективно переносит входной символ *a* в стек, но для представления *a* использует состояние *j*;

   б) свёртка $A \to \beta$. Действие синтаксического анализатора состоит в эффективной свёртке β на вершине стека в заголовок *A*;

   в) принятие. Синтаксический анализатор принимает входную строку и завершает анализ;

   г) ошибка. Синтаксический анализатор обнаруживает ошибку во входной строке.

2. Функция GOTO, определённая на множествах пунктов, распространяется на состояния: если GOTO [I_i, *A*] = I_j, то GOTO отображает состояние *i* и нетерминал *A* на состояние *j*.

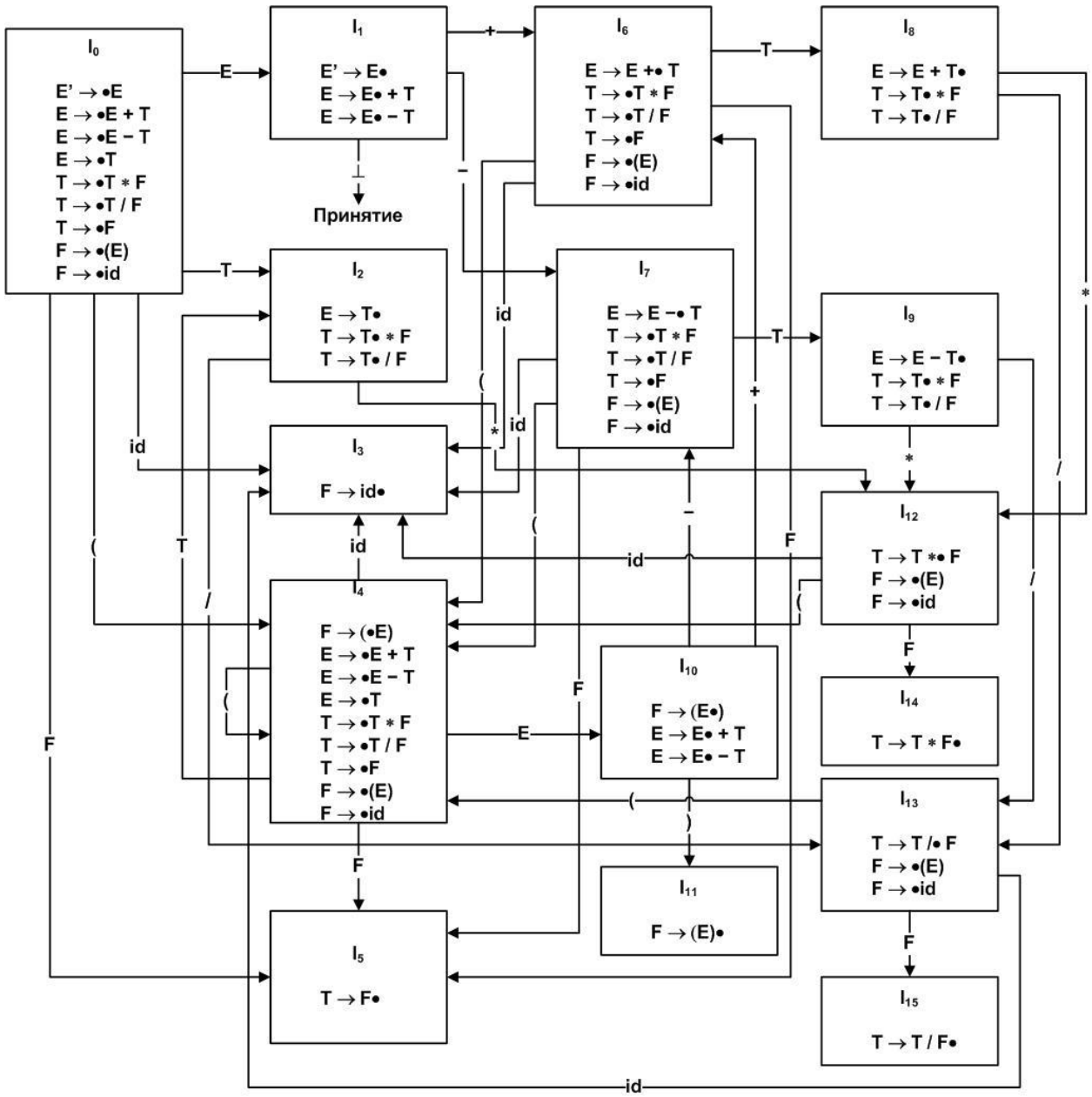


Рисунок 14. LR(0)-автомат для грамматики из примера 13

Пример 15. В таблице 6 показаны функции ACTION и GOTO из таблицы LR-анализа грамматики выражений (см. пример 13), повторенной ниже с пронумерованными правилами:

(1) $E \to E + T$ (5) $T \to T / F$
(2) $E \to E - T$ (6) $T \to F$
(3) $E \to T$ (7) $F \to (E)$
(4) $T \to T * F$ (8) $F \to \boldsymbol{id}$

Коды действий следующие:
1) *si* означает сдвиг и размещение в стеке состояния *i*.
2) *rj* означает свёртку в соответствии с правилом под номером *j*.
3) *acc* означает принятие.
4) Пустое поле означает ошибку.

Значение GOTO [*s*, *a*] для терминала *a* находится в поле ACTION, связанном с переносом для входного символа *a* и состояния *s*. Поле GOTO дает значения GOTO [*s*, *a*] для нетерминалов *A*.

Таблица 6. Таблица синтаксического анализа для грамматики выражений

Состояние	**ACTION**								**GOTO**		
	+	–	*	/	(	)	***id***	⊥	*E*	*T*	*F*
0					*s*4		*s*3		1	2	5
1	*s*6	*s*7						*acc*			
2	*r*3	*r*3	*s*12	*s*13		*r*3		*r*3			
3	*r*8	*r*8	*r*8	*r*8		*r*8		*r*8			
4					*s*4				10	2	5
5	*r*6	*r*6	*r*6	*r*6		*r*6		*r*6			
6					*s*4		*s*3			8	5
7					*s*4		*s*3			9	5
8	*r*1	*r*1	*s*12	*s*13		*r*1		*r*1			
9	*r*2	*r*2	*s*12	*s*13		*r*2		*r*2			
10	*s*6	*s*7				*s*11					
11	*r*7	*r*7	*r*7	*r*7		*r*7		*r*7			
12					*s*4		*s*3				14
13					*s*4		*s*3				15
14	*r*4	*r*4	*r*4	*r*4		*r*4		*r*4			
15	*r*5	*r*5	*r*5	*r*5		*r*5		*r*5			

Рассмотрим заполнение таблицы для состояния I_2:

$E \to T\bullet$

$T \to T\bullet * F$

$T \to T\bullet / F$

Поскольку FOLLOW(*E*) = {$, +,)}, первый пункт приводит к ACTION [2, $] = ACTION [2, +] = ACTION [2,)] = свёртка $E \to T$ (*r*3).

Второй пункт даёт ACTION [2, *] = сдвиг и переход в состояние 12. Третий пункт даёт ACTION [2, /] = сдвиг и переход в состояние 13. Продолжая работу таким образом, мы получим таблицу 6.

Для входной строки ***id*** * ***id*** + ***id*** последовательность содержимого стека и входной строки показана в таблице 7. Для ясности показана также последовательность грамматических символов, соответствующая хранящимся в стеке состояниям. Например, в строке (1) LR-анализатор находится в состоянии 0, начальном состоянии без грамматических символов и с ***id*** в качестве первого входного символа. Действие в строке 0 и столбце ***id*** поля ACTION в таблице 6 – *s*3; оно означает сдвиг и внесение в стек состояния 3. В строке (2) выполняется внесение в стек символа состояния 3 и удаление ***id*** из входного потока.

После этого текущим входным символом становится *; действие для состояния 3 и входного символа * – свёртка согласно правилу $F \to$ ***id*** (*r*8). Со стека при этом снимается один символ состояния, и на вершине стека появляется состояние 0.

Поскольку **GOTO** [0, *F*] равно 5, в стек вносится состояние 5. При этом получается конфигурация, показанная в строке (3). Остальные строки таблицы 7 получены аналогично.

Таблица 7. Действия LR-анализатора для строки ***id * id + id***

	Стек	**Символы**	**Вход**	**Действие**
(1)	0		***id** * **id** + **id**⊥*	Сдвиг
(2)	0 3	***id***	* ***id** + **id**⊥*	Свёртка по $F \rightarrow$ ***id***
(3)	0 5	*F*	* ***id** + **id**⊥*	Свёртка по $T \rightarrow F$
(4)	0 2	*T*	* ***id** + **id**⊥*	Сдвиг
(5)	0 2 12	*T* *	***id** + **id**⊥*	Сдвиг
(6)	0 2 12 3	*T* * ***id***	+ ***id**⊥*	Свёртка по $F \rightarrow$ ***id***
(7)	0 2 12 14	*T * F*	+ ***id**⊥*	Свёртка по $T \rightarrow T * F$
(8)	0 2	*T*	+ ***id**⊥*	Свёртка по $E \rightarrow T$
(9)	0 1	*E*	+ ***id**⊥*	Сдвиг
(10)	0 1 6	*E +*	***id**⊥*	Сдвиг
(11)	0 1 6 3	*E* + ***id***	⊥	Свёртка по $F \rightarrow$ ***id***
(12)	0 1 6 5	*E + F*	⊥	Свёртка по $T \rightarrow F$
(13)	0 1 6 8	*E + T*	⊥	Свёртка по $E \rightarrow E + T$
(14)	0 1	*E*	⊥	Принятие

7.4.2.2 LR(1)-грамматики

Можно хранить в состоянии LR-анализатора больший объём информации, который позволит отбрасывать некорректные свёртки. Разделяя при необходимости состояния, можно добиться того, что каждое состояние будет точно указывать, какие входные символы могут следовать за основой α, для которой возможна свёртка в *A*. Дополнительная информация вносится в состояние путём такого переопределения пунктов, чтобы они включали в качестве второго компонента терминальный символ. Общим видом пункта становится $[A \rightarrow \alpha\bullet\beta, a]$, где $A \rightarrow \alpha\beta$ – правило, а *a* – терминал или маркер конца входной строки ⊥. Такой объект называется **LR(1)-пунктом**. Здесь 1 означает длину второго компонента, именуемого *предпросмотром* (англ. lookahead) пункта. Предпросмотр не влияет на пункт вида $[A \rightarrow \alpha\bullet\beta, a]$, где β не равно ε, но пункт $[A \rightarrow \alpha\bullet, a]$ приводит к свёртке в соответствии с правилом $A \rightarrow \alpha$, только если очередной входной символ равен *a*.

Таким образом, свёртка в соответствии с продукцией $A \rightarrow \alpha$ применяется только при входном символе *a*, для которого $[A \rightarrow \alpha\bullet, a]$ является LR(1)-пунктом из состояния на вершине стека.

Метод построения набора множеств допустимых LR(1)-пунктов, по сути, тот же, что и для построения канонического набора множеств LR(0)-пунктов. Нужно только модифицировать формирование множеств **CLOSURE** и **GOTO**.

Замыкание пунктов

Пусть **I** – множество пунктов грамматики **G**, тогда **CLOSURE(I)** представляет собой множество пунктов, построенное из **I** согласно двум правилам.

1. Изначально в **CLOSURE(I)** добавляются все пункты из **I.**

2. Если $[A \to \alpha \bullet B\beta, a]$ входит в **CLOSURE(I)**, и существует правило $B \to \gamma$, то в **CLOSURE (I)** добавляется пункт $[B \to \bullet\gamma, b]$ для каждого терминала $b \in$ **FIRST**(βa), если такого пункта ещё нет во множестве замыкания. Это правило применяется до тех пор, пока не останется пунктов, которые могут быть добавлены в **CLOSURE (I).**

Пример 16. Рассмотрим следующую расширенную грамматику:

$S' \to S$

$S \to CC$

$C \to cC \mid d$

Начнём с вычисления замыкания **CLOSURE($\mathbf{I_0}$)** для пункта $[S' \to \bullet S, \perp]$. Сопоставим этот пункт с пунктом из описания алгоритма $[A \to \alpha \bullet B\beta, a]$, т.е. $A = S'$, $\alpha = \varepsilon$, $B = S$, $\beta = \varepsilon$ и $a = \perp$. В терминах рассматриваемой грамматики $B \to \gamma$ должно быть $S \to CC$, а поскольку β равно ε, а a равно $\perp$, то b тоже равно $\perp$. Таким образом, во множество замыкания добавляется пункт $[S \to \bullet CC, \perp]$.

На следующем шаге сопоставляем $[S \to \bullet CC, \perp]$ с шаблонным пунктом $[A \to \alpha \bullet B\beta, a]$. Получаем $A = S$, $\alpha = \varepsilon$, $B = C$, $\beta = C$ и $a = \perp$. Множество **FIRST**(C) содержит терминальные символы c и d, и мы добавляем пункты $[C \to \bullet cC, c]$, $[C \to \bullet cC, d]$, $[C \to \bullet d, c]$ и $[C \to \bullet d, d]$. Для удобства записи пункты вида $[C \to \bullet cC, c]$ и $[C \to \bullet cC, d]$ можно представить сокращённо $[C \to cC, c/d]$. Ни один из новых пунктов не имеет нетерминала непосредственно справа от точки, так что первое множество LR(1)-пунктов завершено. Аналогично строятся замыкания и для других пунктов (см. рис. 15).

Множество переходов

Множество переходов **GOTO(I**, X) определяется как замыкание множества всех пунктов $[A \to \alpha X \bullet \beta, a]$, таких что $[A \to \alpha \bullet X\beta, a]$ находится в **I.**

Построим множество **GOTO($\mathbf{I_0}$,** X) для множества LR(1)-пунктов из примера 16.

Для $X = S$ мы должны вычислить замыкание пункта $[S' \to S\bullet, \perp]$. Никакие дополнительные замыкания невозможны, поскольку точка располагается крайней справа. Таким образом, мы получаем следующее множество пунктов:
$\mathbf{I_1} = \{[S' \to S\bullet, \perp]\}$.

Для $X = C$ вычисляем замыкание $[S \to C\bullet C, \perp]$. Добавляем правила для C с символом предпросмотра $\perp$, после чего не можем добавить ничего более и, таким образом, получаем $\mathbf{I_2} = \{[S \to C\bullet C, \perp], [C \to \bullet cC, \perp], [C \to \bullet d, \perp]\}$.

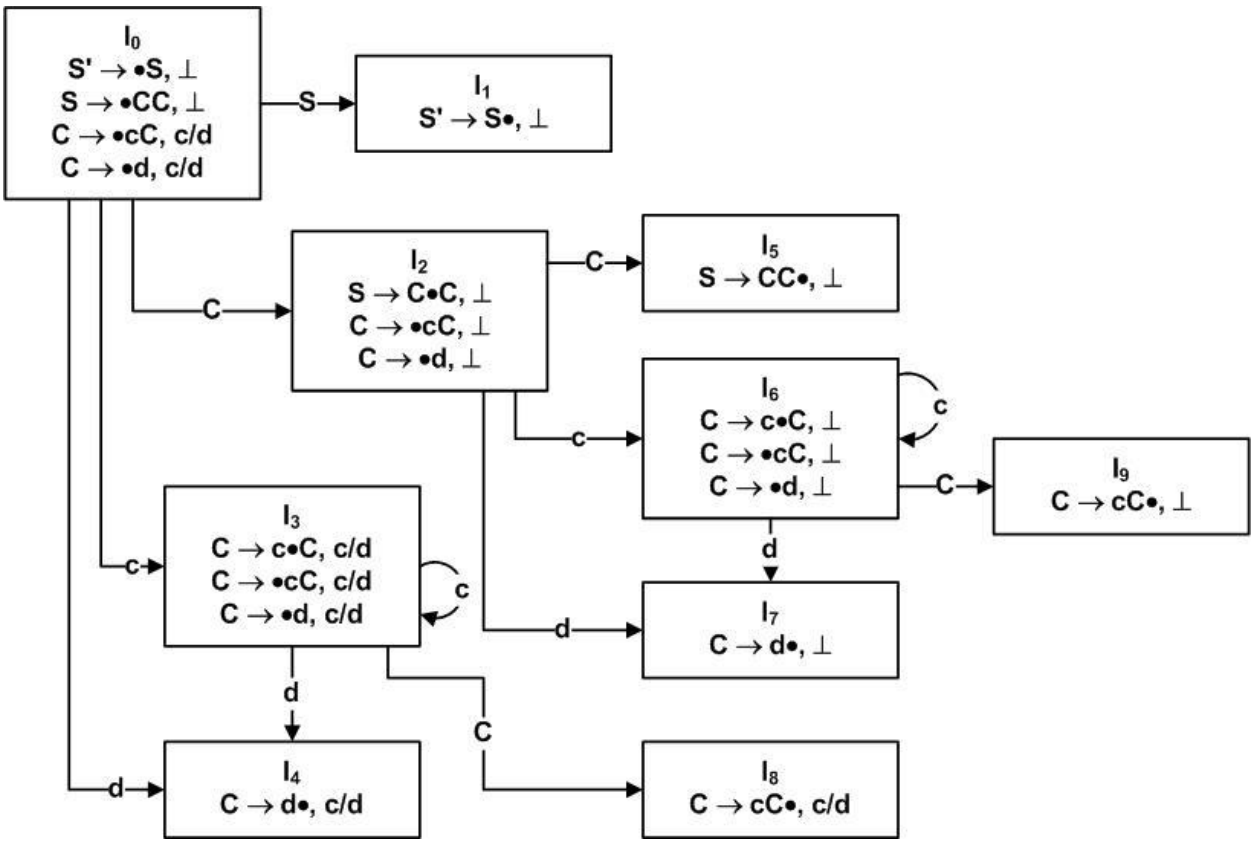


Рисунок 15. LR(1)-автомат для грамматики из примера 16

Далее положим $X = c$. Теперь надо выполнить замыкание $[C \to c\bullet C, c/d]$. Добавляем правила для C с символом предпросмотра c/d, что даёт
$\mathbf{I_3} = \{[C \to c\bullet C, c/d], [C \to \bullet cC, c/d], [C \to \bullet d, c/d]\}$.

Наконец, для $X = d$ получаем множество пунктов $\mathbf{I_4} = \{[C \to d\bullet, c/d]\}$. Этим завершается построение **GOTO** для $\mathbf{I_0}$.

LR(1)-автомат для грамматики из примера 16 показан на рисунке 15.

Теперь приведём правила построения функций ACTION и GOTO из множеств LR(1)-пунктов. Эти функции, как и в случае LR(0)-анализа, представлены таблицей. Единственное отличие – в значениях записей таблицы.

Действие синтаксического анализа для состояния i определяется следующим образом:

а) если $[A \to \alpha\bullet a\beta, b]$ входит во множество пункутов $\mathbf{I_i}$ и $\mathbf{GOTO}(\mathbf{I_i}, a) = \mathbf{I_j}$, установить **ACTION**$[i, a]$ равным «сдвиг j». Здесь a должно быть терминалом;

б) если $[A \to \alpha\bullet, a]$ входит в $\mathbf{I_i}$ и $A \neq S'$, то установить **ACTION**$[i, a]$ равным «свёртка $A \to \alpha$»;

в) если $[S' \to S\bullet, \perp]$ входит в $\mathbf{I_i}$, установить **ACTION**$[i, \perp]$ равным «принятие».

Если при применении указанных правил обнаруживаются конфликтующие действия, грамматика не принадлежит классу LR(1).

Переходы для состояния i строятся для всех нетерминалов A с использованием следующего правила: если $\mathbf{GOTO}(\mathbf{I_i}, A) = \mathbf{I_j}$, то **GOTO**$[i, A] = j$.

Таблица, образованная функциями действий и переходов, полученными при помощи такого алгоритма, называется *канонической таблицей* LR(1)-анализа.

Каноническая таблица синтаксического анализа для грамматики из примера 16 приведена в таблице 8.

Таблица 8. Каноническая таблица синтаксического анализа для грамматики из примера 16

Состояние	**ACTION**			**GOTO**	
	c	*d*	⊥	*S*	*C*
0	*s*3	*s*4		1	2
1			*acc*		
2	*s*6	*s*7			5
3	*s*3	*s*4			8
4	*r*1	*r*3			
5			*r*1		
6	*s*6	*s*7			9
7			*r*3		
8	*r*2	*r*2			
9			*r*2		

7.4.2.3 LALR(1)-грамматики

Идея распознавателей для LALR(*k*)-грамматик (англ. lookahead LR – LR с предпросмотром) – сократить множество рассматриваемых пунктов до множества пунктов LR(0)-грамматики и упростить построение таблицы синтаксического анализа. Все LALR(*k*)-грамматики для всех $k \geq 1$ образуют класс LALR-грамматик.

На практике используются только LALR(1)-грамматики, так как применяемый для них метод разрешения конфликтов («сдвиг-свёртка» или «свёртка-свёртка») трудно распространить на LALR(*k*)-грамматики для $k > 1$.

Сравнивая размеры синтаксических анализаторов, можно сказать, что количество состояний в LALR-таблицах обычно составляет несколько сотен для языков наподобие Си. Каноническая LR-таблица для языка такого типа обычно содержит несколько тысяч состояний.

В качестве примера формирования LALR-таблицы возьмём грамматику из примера 16, множества LR(1)-пунктов которой были приведены на рисунке 15. Возьмём пару похожих состояний, таких как $\mathbf{I_4}$ и $\mathbf{I_7}$. Эти состояния содержат только пункты с первым компонентом $C \to d\bullet$. В $\mathbf{I_4}$ предпросматриваемыми символами могут быть *c* и *d*, а в $\mathbf{I_7}$ – только ⊥. Заменим теперь состояния $\mathbf{I_4}$ и $\mathbf{I_7}$ состоянием $\mathbf{I_{47}}$, которое представляет собой объединение $\mathbf{I_4}$ и $\mathbf{I_7}$, состоящее из трёх пунктов: $[C \to d\bullet, c/d/\perp]$. Все переходы по *d* в $\mathbf{I_4}$ или $\mathbf{I_7}$ из $\mathbf{I_0}$, $\mathbf{I_2}$, $\mathbf{I_3}$ и $\mathbf{I_6}$ ведут теперь в $\mathbf{I_{47}}$. Действие в этом состоянии – свёртка при любом входном символе. Такой синтаксический анализатор в целом ведёт себя так же, как исходный, хотя и может свернуть *d* в *C* в условиях, когда исходный синтаксический анализатор объявил бы об ошибке, например, при входной строке *ccd* или *cdcdc*. В конце концов, ошибка будет обнаружена перед тем, как мы получим любой символ, вызывающий сдвиг.

Обобщая, мы можем рассмотреть множества LR(1)-пунктов, имеющих одно и то же *ядро*, т.е. множество первых компонентов, и объединить эти множества с общими ядрами в одно множество пунктов. Например, на рисунке 15 такую пару с ядром [$C \to d\bullet$], как уже упоминалось, образуют состояния $\mathbf{I_4}$ и $\mathbf{I_7}$. Аналогично множества $\mathbf{I_3}$ и $\mathbf{I_6}$ образуют другую пару – с ядром [$C \to c\bullet C$, $C \to \bullet cC$, $C \to \bullet d$]. Имеется и ещё одна пара – $\mathbf{I_8}$ и $\mathbf{I_9}$ – с общим ядром [C → cC•].

Поскольку ядро множества **GOTO** (**I**, *X*) зависит только от ядра множества **I**, значения функции **GOTO** объединяемых множеств также могут быть объединены. Таким образом, проблем вычисления функции **GOTO** при слиянии множеств не возникает. Функция же **ACTION** должна быть изменена, чтобы отражать неошибочные действия всех объединяемых множеств пунктов.

При объединении множеств пунктов возможно появление конфликта «свёртка-свёртка», как показано в следующем примере.

Пример 17. Рассмотрим грамматику

$S' \to S$

$S \to aAd \mid bBd \mid aBe \mid bAe$

$A \to c$

$B \to c$

Она генерирует четыре строки – *acd*, *ace*, *bed* и *bee*. Построив множества LR(1)-пунктов, мы обнаружим множество пунктов {[$A \to c\bullet, d$], [$B \to c\bullet, e$]}, допустимых для активного префикса *ac*, и {[$A \to c\bullet, e$], [$B \to c\bullet, d$]} – для префикса *bc*. Ни одно из этих множеств не вызывает конфликта; ядра их одинаковы. Однако их объединение:

$A \to c\bullet, d/e$

$B \to c\bullet, d/e$

вызывает конфликт «свёртка-свёртка», поскольку при входных символах *d* и *e* вызываются две свёртки: $A \to c$ и $B \to c$.

Основная идея алгоритма построения LALR-таблицы состоит в создании множеств LR(1)-пунктов. Для каждого ядра, имеющегося среди этих множеств, находим все множества, имеющие это ядро, и заменяем их объединением.

Пусть $\mathbf{C} = \{\mathbf{J_0}, \mathbf{J_1}, ..., \mathbf{J_m}\}$ – полученные в результате множества LR(1)-пунктов. Функцию ACTION для состояния *i* строим из $\mathbf{J_i}$ так же, как и в алгоритме для канонических LR(1)-таблиц. Если при этом обнаруживается конфликт, алгоритм не в состоянии построить синтаксический анализатор, а грамматика не является LALR(1)-грамматикой.

Функцию GOTO строим следующим образом. Если **J** – объединение одного или нескольких множеств LR(1)-пунктов, т.е. $\mathbf{J} = \mathbf{I_1} \cup \mathbf{I_2} \cup ... \cup \mathbf{I_k}$, то ядра множеств **GOTO** ($\mathbf{I_1}$, *X*), **GOTO** ($\mathbf{I_2}$, *X*), ..., **GOTO** ($\mathbf{I_k}$, *X*) одни и те же, поскольку $\mathbf{I_1}$, $\mathbf{I_2}$, ..., $\mathbf{I_k}$, имеют одно и то же ядро. Обозначим через **К** объединение всех множеств пунктов, имеющих то же ядро, что и **GOTO** (**J**, *X*). Тогда **GOTO** (**J**, *X*) = **К**.

Таблица, полученная при помощи описанного выше алгоритма, называется *таблицей LALR-анализа* для грамматики **G**. Если конфликты действий отсутствуют, то данная грамматика называется *LALR(1)-грамматикой*.

Пример 17. Вновь обратимся к грамматике из примера 16, граф **GOTO** которой показан на рисунке 15. Как уже упоминалось, имеется три пары множеств пунктов, которые могут быть объединены. $\mathbf{I_3}$ и $\mathbf{I_6}$ заменяются их объединением $\mathbf{I_{36}}$:

$C \to c\bullet C,\ c/d/\perp$

$C \to \bullet cC,\ c/d/\perp$

$C \to \bullet d,\ c/d/\perp$

$\mathbf{I_4}$ и $\mathbf{I_7}$ заменяются их объединением $\mathbf{I_{47}}$: $C \to d\bullet,\ c/d/\perp$

$\mathbf{I_8}$ и $\mathbf{I_9}$ заменяются их объединением $\mathbf{I_{89}}$: $C \to cC\bullet,\ c/d/\perp$

Функции действий и переходов LALR для объединенных множеств пунктов показаны в таблице 9.

Чтобы увидеть, каким образом вычисляется функция **GOTO**, рассмотрим **GOTO(**$\mathbf{I_{36}}$**,** *C*). В исходном множестве LR(1)-пунктов **GOTO(**$\mathbf{I_3}$**,** *C*) = $\mathbf{I_8}$, а $\mathbf{I_8}$ теперь является частью $\mathbf{I_{89}}$, так что мы определяем, что **GOTO(**$\mathbf{I_{36}}$**,** *C*) = $\mathbf{I_{89}}$. Тот же вывод можно сделать, рассматривая $\mathbf{I_6}$ – вторую часть $\mathbf{I_{36}}$. **GOTO(**$\mathbf{I_6}$**,** *C*) = $\mathbf{I_9}$, а $\mathbf{I_9}$ теперь также является частью $\mathbf{I_{89}}$. В качестве другого примера рассмотрим **GOTO(**$\mathbf{I_2}$**,** *c*), переход, выполняемый после переноса состояния $\mathbf{I_2}$ при входном символе *c*. В исходных множествах LR(1)-пунктов **GOTO(**$\mathbf{I_2}$**,** *c*) = $\mathbf{I_6}$. Поскольку теперь $\mathbf{I_6}$ является частью $\mathbf{I_{36}}$, **GOTO(**$\mathbf{I_2}$, *c*) становится равным $\mathbf{I_{36}}$; таким образом, запись в таблице 9 для состояния 2 и входного символа *c* – *s*36, что означает сдвиг и помещение в стек состояния 36.

Таблица 9. Таблица LALR-анализа для грамматики из примера 16

Состояние	**ACTION**			**GOTO**	
	c	*d*	⊥	*S*	*C*
0	*s*36	*s*47		1	2
1			*acc*		
2	*s*36	*s*47			5
36	*s*36	*s*47			89
47	*r*3	*r*3	*r*3		
5			*r*1		
89	*r*2	*r*2	*r*2		

При входной строке *c*dc*d* как LR-анализатор, так и LALR-анализатор выполняют одну и ту же последовательность сдвигов и свёрток, хотя имена состояний в стеке могут отличаться.

Однако при строке с ошибками LALR-анализатор может выполнить несколько свёрток после того, как LR-анализатор уже объявит об ошибке. Например, для входной строки *ccd* LR-анализатор поместит в стек

0 3 3 4

и в состоянии 4 обнаружит ошибку, поскольку следующий входной символ – ⊥, а для состояния 4 соответствующая запись таблицы – «ошибка».

В противоположность этому LALR-анализатор выполняет соответствующие действия, помещая в стек

0 36 36 47

Однако состояние 47 при входном символе ⊥ приводит к свёртке $C \to d$. Таким образом, LALR-анализатор изменит содержимое стека на

0 36 36 89

Действие в состоянии 89 для входного символа ⊥ – свёртка $C \to cC$, после чего содержимое стека приобретает вид

0 36 89

После этого та же свёртка выполняется повторно, что приводит к содержимому стека

0 2

Наконец, мы обнаруживаем ошибку в соответствии с записью таблицы для состояния 2 и входного символа ⊥.

7.4.3 Грамматики предшествования

Ещё одним распространённым классом КС-грамматик, для которых можно построить восходящий распознаватель без возвратов, являются **грамматики предшествования**. Они используются для синтаксического разбора цепочек с помощью модификации алгоритма «сдвиг-свёртка».

Принцип организации распознавателя на основе грамматики предшествования состоит в том, что для каждой упорядоченной пары символов в грамматике устанавливается отношение, называемое **отношением предшествования**. В процессе разбора МП-автомат сравнивает текущий символ входной цепочки с одним из символов, находящихся на верхушке стека автомата. В процессе сравнения проверяется, какое из возможных отношений предшествования существует между этими двумя символами. В зависимости от найденного отношения выполняется либо сдвиг, либо свёртка. При отсутствии отношения предшествования между символами алгоритм сигнализирует об ошибке.

Задача заключается в том, чтобы иметь возможность непротиворечивым образом определить отношения предшествования между символами грамматики. Если это возможно, то грамматика может быть отнесена к одному из классов грамматик предшествования.

Отношения предшествования будем обозначать знаками «=•», «<•» и «•>». Отношение предшествования единственно для каждой упорядоченной пары символов. При этом между какими-либо двумя символами может и не быть отношения предшествования – это значит, что они не могут находиться рядом ни в одном элементе разбора синтаксически правильной цепочки. Отношения предшествования зависят от порядка, в котором стоят символы, и в этом смысле их нельзя путать со знаками математических операций – они не обладают ни свойством коммутативности, ни свойством ассоциативности. Например, если известно, что $B_i \bullet\!> B_j$, то не обязательно выполняется $B_j <\!\bullet B_i$, (поэтому знаки предшествования помечают специальной точкой).

Метод предшествования основан на том факте, что отношения предшествования между двумя соседними символами распознаваемой строки соответствуют трём следующим вариантам:

– $B_i <\!\bullet B_{i+1}$, если символ B_{i+1} – крайний левый символ некоторой основы

(это отношение между символами можно назвать «предшествует основе» или просто «предшествует»);

- $B_i \bullet\!> B_{i+1}$, если символ B_i – крайний правый символ некоторой основы (это отношение между символами можно назвать «следует за основой» или просто «следует»);
- $B_i =\!\bullet\ B_{i+1}$, если символы B_i и B_{i+1} принадлежат одной основе (это отношение между символами можно назвать «составляют основу»).

Исходя из этих соотношений, выполняется разбор входной строки для грамматик предшествования.

Суть принципа такого разбора поясняет рисунок 16. На нём изображена входная цепочка символов αγβδ в тот момент, когда выполняется свёртка цепочки γ. Символ *a* является последним символом подцепочки α, а символ *b* – первым символом подцепочки β. Тогда, если в грамматике удастся установить непротиворечивые отношения предшествования, то в процессе выполнения разбора по алгоритму «сдвиг-свёртка» можно всегда выполнять сдвиг до тех пор, пока между символом на верхушке стека и текущим символом входной цепочки существует отношение <• или =•. А как только между этими символами будет обнаружено отношение •>, сразу надо выполнять свёртку. Причём для выполнения свёртки из стека надо выбирать все символы, связанные отношением =•. Все различные правила в грамматике предшествования должны иметь различные правые части – это гарантирует непротиворечивость выбора правила при выполнении свёртки.

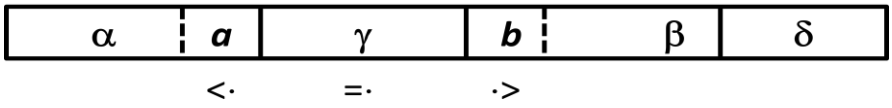


Рисунок 16. Отношения между символами входной цепочки в грамматике предшествования

Таким образом, установление непротиворечивых отношений предшествования между символами грамматики в комплексе с несовпадающими правыми частями различных правил даёт ответы на все вопросы, которые надо решить для организации работы алгоритма «сдвиг-свёртка» без возвратов.

На основании отношений предшествования строят *матрицу предшествования* грамматики. Строки матрицы предшествования помечаются первыми (левыми) символами, столбцы – вторыми (правыми) символами отношений предшествования. В клетки матрицы на пересечении соответствующих столбца и строки помещаются знаки отношений. При этом пустые клетки матрицы говорят о том, что между данными символами нет ни одного отношения предшествования.

Существует несколько видов грамматик предшествования. Они различаются по тому, какие отношения предшествования в них определены и между какими типами символов (терминальными или нетерминальными) могут быть установлены эти отношения. Кроме того, возможны незначительные модификации функционирования самого алгоритма «сдвиг-свёртка» в распознавателях

для таких грамматик (в основном на этапе выбора правила для выполнения свёртки, когда возможны неоднозначности).

Выделяют следующие виды грамматик предшествования:

- простого предшествования;
- расширенного предшествования;
- слабого предшествования;
- смешанной стратегии предшествования;
- операторного предшествования.

В данном учебном пособии рассматриваются структура правил и алгоритмы разбора для грамматик операторного предшествования.

Матрицу операторного предшествования КС-грамматики можно построить, опираясь непосредственно на определения отношений предшествования, но проще и удобнее воспользоваться двумя дополнительными типами множеств – множествами крайних левых и крайних правых символов, а также множествами крайних левых терминальных и крайних правых терминальных символов для всех нетерминальных символов грамматики.

Если имеется КС-грамматика **G (VT, VN, P**, *S*), **V = VT** $\cup$ **VN** то множества крайних левых и крайних правых символов определяются следующим образом:

- **L (***U*) = {*T* | $\exists$ *U* $\Rightarrow$* *Tz*} – множество крайних левых символов относительно нетерминального символа *U*;
- **R** (*U*) = {*T* | $\exists$ *U* $\Rightarrow$* *zT*} – множество крайних правых символов относительно нетерминального символа *U*,

где *U* – заданный нетерминальный символ (*U* $\in$ **VN**), *T* – любой символ грамматики (*T* $\in$ **V**), а *z* – произвольная цепочка символов (*z* $\in$ **V***, цепочка *z* может быть и пустой цепочкой).

Множества крайних левых и крайних правых терминальных символов определяются следующим образом:

- $\mathbf{L_t}$(*U*) = {*t* | $\exists$ *U* $\Rightarrow$* *tz* или $\exists$ *U* $\Rightarrow$* *Ctz*} – множество крайних левых терминальных символов относительно нетерминального символа *U*;
- $\mathbf{R_t}$(*U*) = {*t* | $\exists$ *U* $\Rightarrow$* *zt* или $\exists$ *U* $\Rightarrow$* *Czt*} – множество крайних правых терминальных символов относительно нетерминального символа *U*,

где *t* – терминальный символ (*t* $\in$ **VT**), *U* и *C* – нетерминальные символы (*U*, *C* $\in$ **VN**), а *z* – произвольная цепочка символов (*z* $\in$ **V***, цепочка *z* может быть и пустой цепочкой).

Множества **L** (*U*) и **R** (*U*) могут быть построены для каждого нетерминального символа *U* $\in$ **VN** по достаточно простому алгоритму:

1. Для каждого нетерминального символа *U* ищем все правила, содержащие *U* в левой части. Во множество **L** (*U*) включаем самый левый символ из правой части правил, а во множество **R** (*U*) – самый правый символ из правой части (то есть во множество **L** (*U*) записываем все символы, с которых начинаются правила для символа *U*, а во множество **R** (*U*) – символы, которыми эти правила заканчиваются). Если в правой части правила для символа *U* имеется только один символ, то он должен быть запи-

сан в оба множества – **L** (*U*) и **R** (*U*).

2. Для каждого нетерминального символа *U* выполняем следующее преобразование: если множество **L** (*U*) содержит нетерминальные символы грамматики *U′*, *U″*, ..., то его надо дополнить символами, входящими в соответствующие множества **L** (*U′*), **L** (*U″*), ... и не входящими в **L** (*U*). Ту же операцию надо выполнить для **R** (*U*). Фактически, если какой-то символ *U′* входит в одно из множеств для символа *U*, то надо объединить множества для *U′* и *U*, и результат записать во множество для символа *U*.
3. Если на предыдущем шаге хотя бы одно множество **L** (*U*) или **R** (*U*) для некоторого символа грамматики изменилось, то надо вернуться к шагу 2, иначе – построение закончено.

Для нахождения множеств $\mathbf{L_t}(U)$ и $\mathbf{R_t}(U)$ используется следующий алгоритм:

1. Для каждого нетерминального символа грамматики *U* строятся множества **L** (*U*) и **R** (*U*).
2. Для каждого нетерминального символа грамматики *U* ищутся правила вида $U \to tz$ и $U \to Ctz$, где $t \in \mathbf{VT}$, $C \in \mathbf{VN}$, $z \in \mathbf{V}^*$; терминальные символы *t* включаются во множество $\mathbf{L_t}(U)$. Аналогично для множества $\mathbf{R_t}(U)$ ищутся правила вида $U \to zt$ и $U \to Czt$. То есть, во множество $\mathbf{L_t}(U)$ записываются все крайние слева терминальные символы из правых частей правил для символа *U*, а во множество $\mathbf{R_t}(U)$ – все крайние справа терминальные символы этих правил. Не исключено, что один и тот же терминальный символ будет записан в оба множества.
3. Просматривается множество **L** (*U*), в которое входят символы *U′*, *U″*, ... Множество $\mathbf{L_t}(U)$ дополняется терминальными символами, входящими в $\mathbf{L_t}(U')$, $\mathbf{L_t}(U'')$, ... и не входящими в $\mathbf{L_t}(U)$. Аналогичная операция выполняется и для множества $\mathbf{R_t}(U)$ на основе множества **R** (*U*).

Для практического использования матрицу предшествования дополняют терминальными символами $\perp_н$ и $\perp_к$ (начало и конец цепочки, соответственно). Для них определены следующие отношения предшествования:

$$\perp_н <\bullet\ a, \text{ если } a \in \mathbf{L_t}(S);$$
$$\perp_к \bullet> a, \text{ если } a \in \mathbf{R_t}(S).$$

Имея построенные множества $\mathbf{L_t}(U)$ и $\mathbf{R_t}(U)$, заполнение матрицы операторного предшествования для КС-грамматики **G** (**VT, VN, P**, *S*) можно выполнить по следующему алгоритму:

1) Берём первый символ из множества терминальных символов грамматики **VT**: $a_i \in \mathbf{VT}$, $i = 1$. Будем считать этот символ текущим терминальным символом.
2) Во всём множестве правил **P** ищем правила вида $C \to \alpha a_i b_j \beta$ или $C \to \alpha a_i U b_j \beta$ где a_i – текущий терминальный символ, b_j – произвольный терминальный символ ($b_j \in \mathbf{VT}$), *U* и *C* – произвольные нетерминальные символы (*U*, *C* ∈ **VN**), а α и β – произвольные цепочки символов, возможно пустые (α, β ∈ **V***). Фактически производится поиск

таких правил, в которых в правой части символы a_i и b_j стоят рядом или же между ними есть не более одного нетерминального символа (причём символ a_i обязательно стоит слева от b_j).

3) Для всех символов b_j, найденных на шаге 2, выполняем следующее: ставим знак «=•» («составляет основу») в клетки матрицы операторного предшествования на пересечении строки, помеченной символом a_i, и столбца, помеченного символом b_j.
4) Во всём множестве правил **P** ищем правила вида $C \to \alpha a_i U_j \beta$, где a_i – текущий терминальный символ, U_j и C – произвольные нетерминальные символы (U_j, $C \in$ **VN**), а α и β – произвольные цепочки символов, возможно пустые (α, $\beta \in$ **V***). Фактически ищем правила, в которых в правой части символ a_i стоит слева от нетерминального символа U_j.
5) Для всех символов U_j, найденных на шаге 4, берём множество символов $\mathbf{L_t}(U_j)$. Для всех терминальных символов c_k, входящих в это множество, выполняем следующее: ставим знак «<•» («предшествует») в клетки матрицы операторного предшествования на пересечении строки, помеченной символом a_i, и столбца, помеченного символом c_k.
6) Во всём множестве правил **P** ищем правила вида $C \to \alpha U_j a_i \beta$, где a_i – текущий терминальный символ, U_j и C – произвольные нетерминальные символы (U_j, $C \in$ **VN**), а α и β – произвольные цепочки символов, возможно пустые (α, $\beta \in$ **V***). Фактически ищем правила, в которых в правой части символ a_i стоит справа от нетерминального символа U_j.
7) Для всех символов U_j, найденных на шаге 6, берём множество символов $\mathbf{R_t}(U_j)$. Для всех терминальных символов c_k, входящих в это множество, выполняем следующее: ставим знак «•>» («следует») в клетки матрицы операторного предшествования на пересечении строки, помеченной символом c_k, и столбца, помеченного символом a_i.
8) Если рассмотрены все терминальные символы из множества **VT**, то переходим к шагу 9, иначе – берём очередной символ $a_i \in$ **VT** из множества **VT**, $i := i + 1$, делаем его текущим терминальным символом и возвращаемся к шагу 2.
9) Берём множество $\mathbf{L_t}(S)$ для целевого символа грамматики S. Для всех терминальных символов c_k, входящих в это множество, выполняем следующее: ставим знак «<•» («предшествует») в клетки матрицы операторного предшествования на пересечении строки, помеченной символом $\perp_н$ (начало строки), и столбца, помеченного символом c_k.
10) Берём множество $\mathbf{R_t}(S)$ для целевого символа грамматики S. Для всех терминальных символов c_k, входящих в это множество, выполняем следующее: ставим знак «•>» («следует») в клетки матрицы операторного предшествования на пересечении строки, помеченной символом c_k, и столбца, помеченного символом $\perp_к$ (конец строки). Построение матрицы закончено.

Если на всех шагах алгоритма построения матрицы операторного предшествования не возникло противоречий, когда в одну и ту же клетку матрицы

надо записать два или три различных символа предшествования, то матрица построена правильно (в каждой клетке такой матрицы присутствует один из символов предшествования или же клетка пуста). Если на каком-то шаге возникло противоречие, значит, исходная КС-грамматика **G (VT, VN, P**, *S*) не является грамматикой операторного предшествования. В этом случае можно попробовать преобразовать грамматику так, чтобы она стала удовлетворять требованиям операторного предшествования (что не всегда возможно), либо необходимо использовать другой тип распознавателя.

Пример 18. Рассмотрим грамматику **G** ({***if***, ***then***, ***else***, *a*, :=, ***or***, ***xor***, ***and***, (,), ;}, {*S, F, E, D, C*}, **P**, *S*) с правилами **P**:

$S \to F$;

$F \to$ ***if*** E ***then*** T ***else*** $F \mid$ ***if*** E ***then*** $F \mid a := E$

$T \to$ ***if*** E ***then*** T ***else*** $T \mid a := E$

$E \to E$ ***or*** $D \mid E$ ***xor*** $D \mid D$

$D \to D$ ***and*** $C \mid C$

$C \to a \mid (E)$

Построим множества крайних левых и крайних правых символов согласно описанному ранее алгоритму.

На первом шаге возьмём все крайние левые и крайние правые символы из правил грамматики **G.** Получим следующие множества:

L (*S*) = {*F*, ***if***, *a*}
L (*F*) = {***if***, *a*}
L (*T*) = {***if***, *a*}
L (*E*) = {*E*, *D*, *C*, *a*, (}
L (*D*) = {*D*, *C*, *a*, (}
L (*C*) = {*a*, (}

R (*S*) = {;}
R (*F*) = {*F*, *E*, *D*, *C*, *a*,)}
R (*T*) = {*T*, *E*, *D*, *C*, *a*,)}
R (*E*) = {*D*, *C*, *a*,)}
R (*D*) = {*C*, *a*,)}
R (*C*) = {*a*,)}

Теперь построим множества крайних левых и крайних правых терминальных символов согласно описанному выше алгоритму. Получим следующие множества:

$\mathbf{L_t}$ (*S*) = {***if***, *a*, ;}
$\mathbf{L_t}$ (*F*) = {***if***, *a*}
$\mathbf{L_t}$ (*T*) = {***if***, *a*}
$\mathbf{L_t}$ (*E*) = {***or***, ***xor***, ***and***, *a*, (}
$\mathbf{L_t}$ (*D*) = {***and***, *a*, (}
$\mathbf{L_t}$ (*C*) = {*a*, (}

$\mathbf{R_t}$ (*S*) = {;}
$\mathbf{R_t}$ (*F*) = {***else***, ***then***, :=, ***or***, ***xor***, ***and***, *a*,)}
$\mathbf{R_t}$ (*T*) = {***else***, :=, ***or***, ***xor***, ***and***, *a*,)}
$\mathbf{R_t}$ (*E*) = {***or***, ***xor***, ***and***, *a*,)}
$\mathbf{R_t}$ (*D*) = {***and***, *a*,)}
$\mathbf{R_t}$ (*C*) = {*a*,)}

Для заполнения матрицы операторного предшествования необходимы множества крайних левых и крайних правых терминальных символов и правила исходной грамматики **G**. Заполненная матрица приведена в таблице 10.

Заполнение таблицы рассмотрим на примере лексем ***or*** и (.

Лексема ***or*** не стоит рядом с другими терминальными символами в правилах грамматики. Поэтому знак «=•» для неё не используется. Лексема ***or*** стоит слева от нетерминального символа *D* в правиле $E \to E$ **or** *D*. Во множество

$\mathbf{L_t}(D)$ входят терминалы ***and***, *a* и (. Поэтому в строке матрицы, помеченной символом ***or***, ставим знак «<•» в клетках на пересечении со столбцами, помеченными символами ***and***, *a* и (.

Кроме того, терминал ***or*** стоит справа от нетерминального символа *E* в том же правиле $E \rightarrow E$ ***or*** *D*. Во множество $\mathbf{R_t}(E)$ входят символы ***or***, ***xor***, ***and***, *a* и). Поэтому в столбце матрицы, помеченном символом ***or***, ставим знак «•>» в клетках на пересечении со строками, помеченными символами ***or***, ***xor***, ***and***, *a* и). Больше ни в каких правилах терминальный символ ***or*** не встречается, поэтому заполнение матрицы для него закончено.

Таблица 10. Матрица операторного предшествования для грамматики из примера 18

	;	***if***	***then***	***else***	*a*	:=	***or***	***xor***	***and***	(	)	$\perp_к$
;												•>
if			=•		<•		<•	<•	<•	<•		
then	•>	<•		=•	<•							
else	•>	<•		•>	<•							
a	•>		•>	•>		=•	•>	•>	•>		•>	
:=	•>			•>	<•		<•	<•	<•	<•		
or	•>		•>	•>	<•		•>	•>	<•	<•	•>	
xor	•>		•>	•>	<•		•>	•>	<•	<•	•>	
and	•>		•>	•>	<•		•>	•>	•>	<•	•>	
(					<•		<•	<•	<•	<•	=•	
)	•>		•>	•>			•>	•>	•>		•>	
$\perp_н$	<•	<•			<•							

Символ (стоит рядом с терминальным символом) в правиле $C \rightarrow$ *(E)* (между ними должно быть не более одного нетерминального символа – в данном случае один символ *E*). Поэтому в строке матрицы, помеченной символом (, ставим знак «=•» на пересечении со столбцом, помеченным символом). Терминал (также стоит слева от нетерминального символа *E* в том же правиле $C \rightarrow$ *(E)*. Во множество $\mathbf{L_t}(E)$ входят символы ***or***, ***xor***, ***and***, *a* и (. Поэтому в строке матрицы, помеченной символом (, ставим знак «<•» в клетках на пересечении со столбцами, помеченными символами ***or***, ***xor***, ***and***, *a* и (.

Больше ни в каких правилах символ (не встречается, поэтому заполнение матрицы для него закончено.

Повторяя описанные выше действия по заполнению матрицы для всех терминальных символов грамматики **G**, получим матрицу операторного предшествования (табл. 10). Останется только заполнить строку, соответствующую символу $\perp_н$, и столбец, соответствующий символу $\perp_к$.

Алгоритм «сдвиг-свёртка» для грамматик операторного предшествования выполняется МП-автоматом с одним состоянием. Для моделирования его работы необходима входная цепочка символов и стек символов, в котором автомат может обращаться не только к самому верхнему символу, но и к некоторой цепочке символов на вершине стека.

Этот алгоритм для заданной КС-грамматики **G (VT, VN, P**, *S*) при наличии построенной матрицы предшествования можно описать следующим образом:

1) Поместить в верхушку стека символ $\perp_н$ («начало строки»), считывающую головку МП-автомата поместить в начало входной цепочки (текущим входным символом становится первый символ входной цепочки). В конец входной цепочки надо дописать символ $\perp_к$ («конец строки»).
2) В стеке ищется самый верхний терминальный символ s_j (если на вершине стека лежат нетерминальные символы, они игнорируются и берётся первый терминальный символ, находящийся под ними), при этом сам символ s_j остаётся в стеке. Из входной цепочки берётся текущий символ a_i (справа от считывающей головки МП-автомата).
3) Если символ s_j – это символ $\perp_н$, а символ a_i – $\perp_к$, то алгоритм завершён, входная цепочка символов разобрана.
4) В матрице предшествования ищется клетка на пересечении строки, помеченной символом s_j, и столбца, помеченного символом a_i (выполняется сравнение текущего входного символа и терминального символа на верхушке стека).
5) Если клетка, найденная на шаге 3, пустая, то значит, входная строка символов не принимается МП-автоматом, алгоритм прерывается и выдаёт сообщение об ошибке.
6) Если клетка, найденная на шаге 3, содержит символ «=•» или «<•», то необходимо выполнить сдвиг. При выполнении сдвига текущий входной символ a_i помещается на верхушку стека, считывающая головка МП-автомата во входной цепочке символов сдвигается на одну позицию вправо (после чего текущим входным символом становится следующий символ a_{i+1}, $i := i + 1$). После этого надо вернуться к шагу 2.
7) Если клетка, найденная на шаге 3, содержит символ «•>», то необходимо произвести свёртку. Для выполнения свёртки из стека выбираются все терминальные символы, связанные отношением «=•», начиная от вершины стека, а также все нетерминальные символы, лежащие в стеке рядом с ними. Эти символы вынимаются из стека и собираются в цепочку γ (если в стеке нет символов, связанных отношением «=•», то из него вынимается один самый верхний терминальный символ и лежащие рядом с ним нетерминальные символы).
8) Во всём множестве правил **P** грамматики **G** ищется правило, у которого правая часть совпадает с цепочкой γ (по условиям грамматик предшествования все правые части правил должны быть различны, поэтому может быть найдено или одно такое правило, или ни одного). Если правило найдено, то в стек помещается нетерминальный символ из левой части правила, иначе, если правило не найдено, входная цепочка символов не принимается МП-автоматом, алгоритм прерывается и выдаёт сообщение об ошибке. Следует отметить, что при выполнении свёртки считывающая головка автомата не сдвигается и текущий

входной символ a_i остаётся неизменным. После выполнения свёртки необходимо вернуться к шагу 2.

Алгоритм «сдвиг-свёртка» для грамматики операторного предшествования игнорирует нетерминальные символы. Поэтому имеет смысл преобразовать исходную грамматику таким образом, чтобы оставить в ней только один нетерминальный символ. Это преобразование заключается в том, что все нетерминальные символы в правилах грамматики заменяются на один нетерминальный символ (чаще всего – целевой символ грамматики).

Построенная в результате такого преобразования грамматика называется *остовной грамматикой*, а само преобразование – *остовным преобразованием*. Остовное преобразование не ведёт к созданию эквивалентной грамматики и выполняется только для упрощения работы алгоритма (который при выборе правил всё равно игнорирует нетерминальные символы) после построения матрицы предшествования.

Для грамматики из примера 18 остовной грамматикой будет грамматика вида:

G' (**{*if, then, else,* ***a*****, :=, *or, xor, and,* (,), ;}, {*S*}, P',** *S*) с правилами **P'**:

$S \rightarrow S;$

$S \rightarrow$ ***if*** S ***then*** S ***else*** $S \mid$ ***if*** S ***then*** $S \mid a := S$

$S \rightarrow$ ***if*** S ***then*** S ***else*** $S \mid a := S$

$S \rightarrow S$ ***or*** $S \mid S$ ***xor*** $S \mid S$

$S \rightarrow S$ ***and*** $S \mid S$

$S \rightarrow a \mid (S)$

Полученная в результате остовного преобразования грамматика может не являться однозначной, но все необходимые данные о порядке применения правил содержатся в матрице предшествования и распознаватель остаётся детерминированным. Поэтому остовное преобразование может выполняться без потерь информации только после построения матрицы предшествования. При этом также необходимо следить, чтобы в грамматике не возникло неоднозначностей из-за одинаковых правых частей правил, которые могут появиться в остовной грамматике.

Вывод, полученный при разборе на основе остовной грамматики, называют *результатом остовного разбора* или *остовным выводом.*

По результатам остовного разбора можно построить соответствующий ему вывод на основе правил исходной грамматики. Однако эта задача не представляет практического интереса, поскольку остовной вывод отличается от вывода на основе исходной грамматики только тем, что в нём отсутствуют шаги, связанные с применением цепных правил, и не учитываются типы нетерминальных символов. Для компиляторов же распознавание цепочек входного языка заключается не в нахождении того или иного вывода, а в выявлении остовных синтаксических конструкций исходной программы с целью построения на их основе цепочек языка результирующей программы. В этом смысле типы нетерминальных символов и цепные правила не несут никакой полезной информации, а напротив, только усложняют обработку цепочки вывода. Поэтому для реального компилятора нахождение остовного вывода является даже более по-

лезным, чем нахождение вывода на основе исходной грамматики. Найденный остовной вывод в дальнейших преобразованиях уже не нуждается.

В общем виде последовательность построения распознавателя для КС-грамматики операторного предшествования **G** (**VT, VN, P**, *S*) можно описать следующим образом:

1) На основе множества правил грамматики **P** построить множества крайних левых и крайних правых символов для всех нетерминальных символов грамматики $U \in$ **VN**: **L** (*U*) и **R** (*U*).
2) На основе множества правил грамматики **P** и построенных на шаге 1 множеств **L** (*U*) и **R** (*U*) построить множества крайних левых и крайних правых терминальных символов для всех нетерминальных символов грамматики $U \in$ **VN**: $\mathbf{L_t}(U)$ и $\mathbf{R_t}(U)$.
3) На основе построенных на шаге 2 множеств $\mathbf{L_t}(U)$ и $\mathbf{R_t}(U)$ для всех терминальных символов грамматики заполняется матрица операторного предшествования.
4) Исходная грамматика **G** (**VT, VN, P,** *S*) преобразуется в остовную грамматику **G'(VT,** {*S*}**, P,** *S***)** с одним нетерминальным символом.
5) На основе построенной матрицы предшествования и остовной грамматики строится распознаватель на базе алгоритма «сдвиг-свёртка» для грамматик операторного предшествования.

7.5 Программный инструментарий Bison

Bison – программный инструментарий, предназначенный для генерации синтаксических анализаторов на основе LALR(1)-грамматик. Bison совместим с другим генератором распознавателей – Yacc (англ. Yet Another Compiler Compiler), разработанным в 1975-1978 гг.

В большинстве случаев сканер, сгенерированный Flex, является сопрограммой для синтаксического анализатора, генерируемого с помощью Bison (рис. 17). Лексический анализатор возвращает синтаксическому поток токенов. Задача распознавателя выяснить взаимоотношения между этими токенами. Обычно такие взаимоотношения отображаются в виде *дерева разбора*.

Например, для арифметического выражения `1 * 2 + 3 * 4 + 5` получится дерево разбора, изображённое на рисунке 18. Операция умножения имеет более высокий приоритет, чем операция сложения, поэтому первыми будут операции `1 * 2` и `3 * 4`. Затем результаты этих двух операций складываются вместе, и эта сумма складывается с `5`. Каждая ветвь дерева показывает взаимоотношение между токенами или поддеревьями.

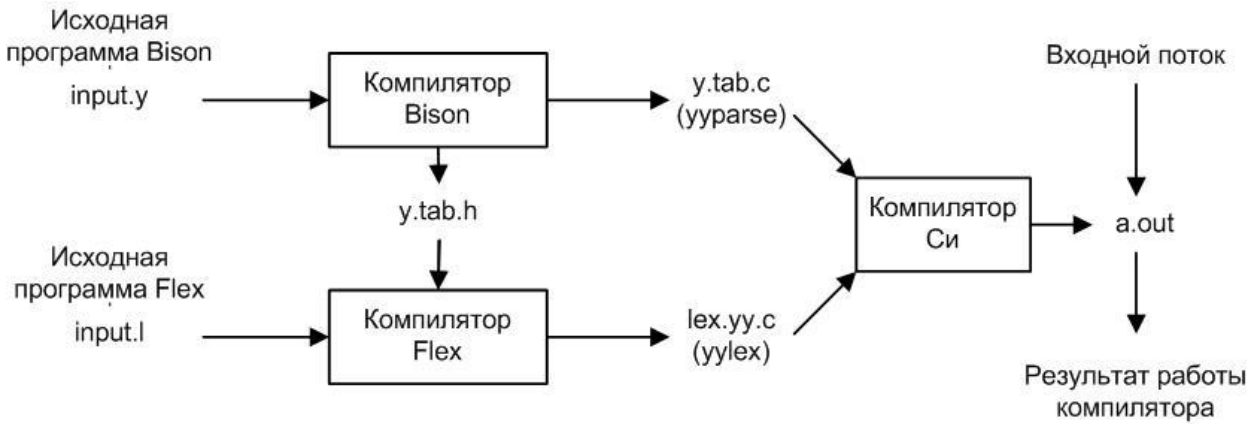


Рисунок 17. Схема совместной работы Flex и Bison

Любой синтаксический анализатор, сгенерированный с использованием Bison, формирует дерево разбора по мере анализа входных токенов.

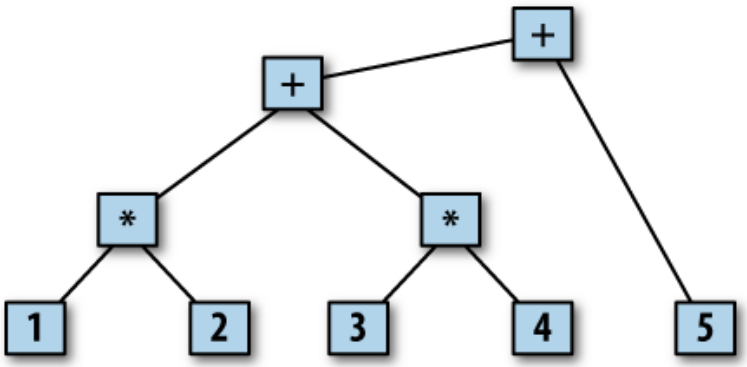


Рисунок 18. Дерево разбора для арифметического выражения 1 * 2 + 3 * 4 + 5

7.5.1 Как работает распознаватель Bison

Bison работает с грамматикой, которая определяется во входном файле, и создаёт синтаксический анализатор, распознающий «предложения», соответствующие этой грамматике. Для грамматики языка программирования такими предложениями будут синтаксически правильные программы на данном языке.

Синтаксически правильная программа не всегда является семантически правильной. Например, для языка Си присваивание строкового значения переменной типа `int` – семантически неверно, но удовлетворяет синтаксическим правилам языка. Bison проверяет только правильность синтаксиса.

Описание грамматики на языке Bison и его соответствие форме Бэкуса-Наура приведено в таблице 11. Вертикальная черта (|) показывает, что существует две возможности задания одного и того же нетерминального символа или что несколько правил могут иметь одинаковую левую часть. Символы в левой части правила – нетерминалы. Символы, возвращаемые лексическим анализатором, – терминалы или токены (в таблице 11 таким символом является `NAME`). Нельзя использовать одинаковые имена для терминальных и нетерминальных символов.

Таблица 11. Описание грамматики на языке Bison

Пример грамматики (Bison)	Форма Бэкуса-Наура

`statement: NAME'='expression`
`expression: NUMBER'+'NUMBER`
`| NUMBER'-'NUMBER`
	`statement → NAME'='expression`
	`expression → NUMBER'+'NUMBER`
	`| NUMBER'-'NUMBER`

Дерево разбора для предложения `fred = 12 + 13`, соответствующего грамматике, приведенной в таблице 11, показано на рисунке 19.

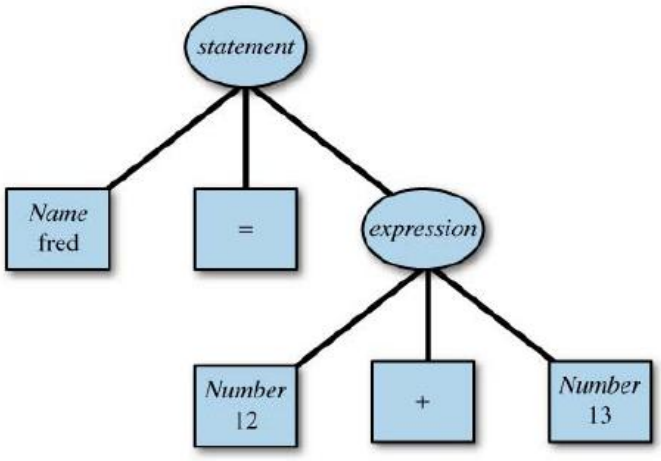


Рисунок 19. Дерево разбора для цепочки символов fred = 12 + 13

В этом примере `12 + 13` соответствует нетерминалу `expression`, а `fred = expression` формирует `statement`.

Каждая грамматика содержит начальный символ, который выступает в качестве корня дерева разбора. В данной грамматике `statement` является таким символом.

Правила могут явным или неявным образом ссылаться сами на себя. Такое свойство позволяет разбирать входные предложения любой длины. Грамматика из таблицы 11 может быть расширена для представления более длинных арифметических выражений:

```
statement: NAME'='expression
expression: expression'+'NUMBER
          | expression'-'NUMBER
          | NUMBER
```

Теперь последовательность вида `fred = 14 + 23 - 11 + 7` может быть успешно разобрана, как это показано на рисунке 20.

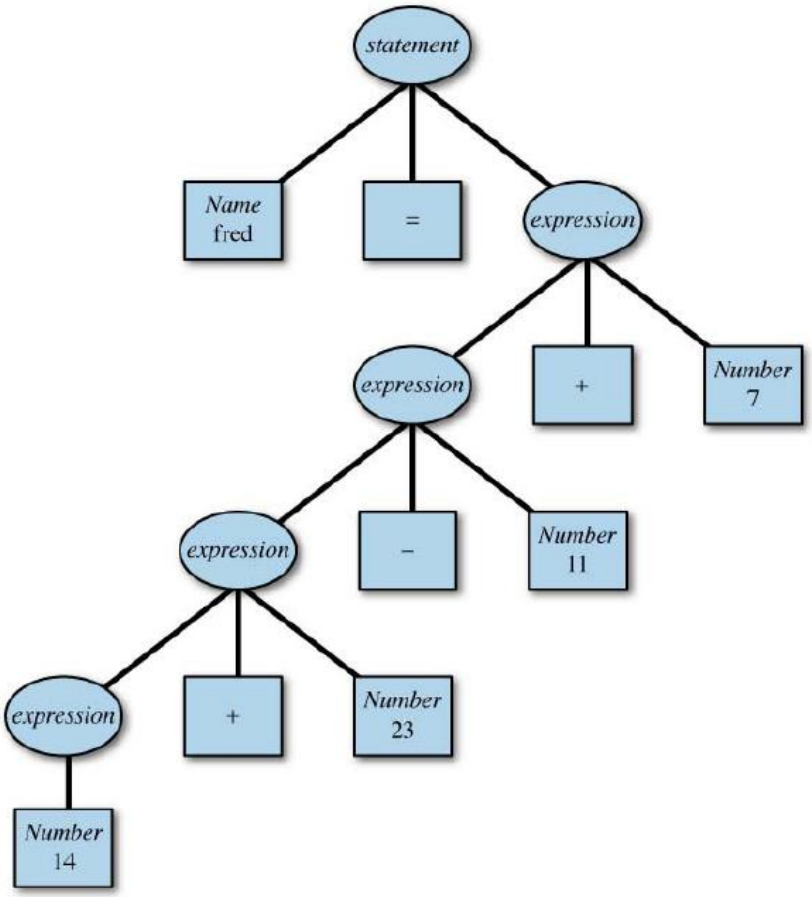


Рисунок 20. Дерево разбора для fred = 14 + 23 − 11 + 7

Распознаватель, генерируемый Bison, в процессе своей работы ищет правила, подходящие для получаемых им от лексического анализатора токенов. Он создает набор состояний, каждое из которых отражает возможную позицию в одном или нескольких частично распознанных правилах. Каждый раз при прочтении очередного токена, синтаксический анализатор помещает его во внутренний стек и переходит к новому состоянию, соответствующему этому токену. Это действие распознавателя называют *сдвигом*. Когда все символы, образующие правую часть правила, найдены, эти символы выталкиваются из стека, символ из левой части данного правила помещается в стек, а распознаватель переходит в состояние, соответствующее этому символу на верхушке стека. Такое действие называют *свёрткой*. При каждой свёртке Bison выполняет пользовательский программный код, связанный с этим правилом.

Рассмотрим пример разбора входной последовательности `fred = 12 + 13` в соответствии с правилами из таблицы 11. Синтаксический анализатор получает токены от лексического и последовательно заносит их в стек:

```
fred = 12 + 13
fred = 12 +
fred = 12
fred =
fred
```

После получения токена 13 распознаватель может применить правило expression: NUMBER + NUMBER для свёртки. В этом случае 12, + и 13 выталкиваются из стека и заменяются нетерминалом expression. В стеке остаётся: fred = expression.

Теперь применяется правило statement: NAME = expression. fred, =, и expression выталкиваются из стека, а statement наоборот туда помещается. Достигнут конец входной последовательности, и на вершине стека находится начальный символ грамматики. Следовательно, вход успешно распознан.

Распознаватели, генерируемые Bison, могут использовать один из двух известных методов: LALR(1) (англ. Look Ahead Left to Right) и GLR (англ. Generalized Left to Right). Большинство синтаксических анализаторов используют первый метод, потому что он быстрее и легче в реализации по сравнению с GLR.

Метод LALR(1) не может использоваться для распознавания:

– неоднозначных грамматик;
– грамматик, требующих более одного символа предпросмотра.

Рассмотрим следующий пример. Грамматика описана правилами:

```
phrase: cart_animal AND CART
      | work_animal AND PLOW
cart_animal: HORSE | GOAT
work_animal: HORSE | OX
```

Эта грамматика однозначна, так как можно построить только одно дерево вывода для любой входной цепочки символов. Но Bison не сможет правильно разобрать эти цепочки, потому что ему потребуется два символа предпросмотра, чтобы сделать это. Например, для входа HORSE AND CART он не сможет определить, по какому правилу нужно сворачивать HORSE: cart_animal: HORSE или work_animal: HORSE, пока не встретит символ CART. Можно изменить первое правило этой грамматики следующим образом:

```
phrase: cart_animal CART | work_animal PLOW
```

В этом случае у Bison не возникнет проблем с разбором.

На практике языки программирования обычно описываются однозначными грамматиками, требующими не более одного символа предпросмотра.

7.5.2 Структура программы на языке Bison

Программа, написанная на языке Bison, состоит из трёх основных частей, как и программа на Flex:

```
Объявления
%%
Правила
%%
Вспомогательные функции
```

Первые два раздела являются обязательными, хотя и могут быть пустыми. Третий раздел и предшествующие ему символы `%%` могут отсутствовать.

Первый раздел, раздел объявлений – это управляющая информация для синтаксического анализатора, и обычно он настраивает среду исполнения для анализа. Этот раздел может включать в себя программный код на Си, который полностью копируется в начало генерируемого файла. Обычно это объявления переменных и директивы `#include`, заключенные в `%{` и `%}`. В этом разделе также могут быть объявления вида `%union`, `%start`, `%token`, `%type`, `%left`, `%right` и `%nonassoc`. Раздел объявлений может содержать комментарии в `/*` и `*/`.

Второй раздел содержит правила грамматики и действия, связанные с ними. Действие представляет собой программный код на Си, который выполняется, когда Bison применяет правило для входных данных. Например:

```
date: month'/'day'/'year {printf("date found");};
```

В действиях можно ссылаться на нетерминальные символы из правой части правил, используя символ ‘$’ и номер нетерминала в правиле:

```
date: month'/'day'/'year {printf("date %d-%d-%d found",
                                  $1, $3, $5);};
```

Для использования нетерминала из левой части правила используется `$$`.

Пример описания правил и действий для следующей грамматики приведён в листинге 6:

G ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –, +, '\n'}, {program, expr, integer, digit}, **P**, program)
P:
program → program expr '\n' | ε
expr → integer | expr + expr | expr – expr
integer → digit | digit integer
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Листинг 6. Описание правил и действий на языке Bison

```
program:
          program expr '\n'     {printf("%d\n", $2);}
          |
          ;
expr:
          INTEGER               {$$ = $1;}
          | expr '+' expr       {$$ = $1 + $3;}
          | expr '-' expr       {$$ = $1 - $3;}
          ;
```

С помощью левой рекурсии мы определили, что программа (`program`) состоит из 0 или более выражений (`expr`). Каждое выражение заканчивается переходом на новую строку. Как только новая строка найдена, значение выражения выводится на экран.

Правило для нетерминального символа digit должно быть описано в разделе правил для лексического анализатора, например, так:

```
[0-9]+    {
              yylval = atoi(yytext);
              return INTEGER;
          }
```

Таким образом, INTEGER – это токен, возвращаемый лексическим анализатором синтаксическому. В области объявлений программы синтаксического анализатора нужно сделать следующее определение:

```
%token INTEGER
```

Третий раздел – это программный код на Си, который полностью копируется в генерируемую программу анализатора.

Bison по умолчанию генерирует файл с именем y.tab.c и заголовочный файл с именем y.tab.h. Для нашего примера заголовочный файл будет содержать:

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define INTEGER 258
extern YYSTYPE yylval;
```

Программа лексического анализатора должна включать этот файл. Для получения токена синтаксический анализатор вызывает функцию yylex(). Значения, связанные с объявленным токеном, помещаются сканером в переменную yylval.

При синтаксическом разборе Bison организует хранение в памяти двух стеков: стека разбора и стека значений. Стек разбора содержит терминалы и нетерминалы, представляющие текущее состояние разбора. Стек значений – это массив элементов типа YYSTYPE, который связывает значение с каждым элементом из стека разбора. Например, когда лексический анализатор возвращает токен INTEGER, Bison помещает этот токен в стек разбора. А соответствующее этому токену значение yylval помещается в стек значений. Таким образом, оба стека синхронизированы.

Когда применяется правило

```
expr: expr '+' expr {$$ = $1 + $3;},
```

происходит замещение в стеке разбора правой части правила на левую. В этом случае из стека удаляются три символа expr, '+' и expr, и сохраняется expr. Ссылка на позиции в стеке значений происходит при использовании $1 для первого символа из правой части правила, $2 – для второго и т.д. $$ обозначает верхушку стека после выполнения свёртки. Таким образом, действие в примере выполняет сложение значений, связанных с двумя символами expr, удаляет три элемента из стека значений и записывает туда полученную сумму. В результате оба стека остаются синхронизированными.

Для запуска синтаксического анализа на выполнение нужно вызвать функцию `yyparse()`.

Полный текст лексического и синтаксического анализаторов для грамматики, приведённой выше, представлен в следующих листингах.

Листинг 7. Лексический анализатор на языке Flex – input.l

```
%option noyywrap

%{
#include <stdlib.h>
void yyerror (char *);
#include "y.tab.h"
%}

%%

[0-9]+    {
              yylval = atoi(yytext);
              return INTEGER;
          }

[+-\n]    {return *yytext;}

[  \t]    ; /* пропускаем пробелы и табуляции */

.         {yyerror("Неизвестный символ!");}

%%
```

Листинг 8. Синтаксический анализатор на языке Bison – input.y

```
%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(char *);
%}

%token INTEGER

%%

program:
        program expr '\n'     {printf("%d\n", $2);}
        |
        ;

expr:
        INTEGER               {$$ = $1;}
        | expr '+' expr       {$$ = $1 + $3;}
        | expr '-' expr       {$$ = $1 - $3;}
        ;

%%

void yyerror(char *s)
{
    fprintf(stderr, "%s\n", s);
}

int main(void)
{
    yyparse();
    return 0;
}
```

Для компиляции и запуска программы используются следующие команды:

```
$ bison -dy input.y    (можно использовать yacc -d input.y)
$ flex input.l
$ gcc lex.yy.c y.tab.c
$ ./a.out
1+4
5
2+3
5
```

Контрольные вопросы

1. На каком алгоритме основана работа распознавателя для LL(k)-грамматик?
2. На каком алгоритме основана работа распознавателя для LR(k)-грамматик?
3. В чём отличие метода нисходящего синтаксического анализа от метода восходящего анализа?
4. В каких случаях алгоритм для LALR(1)-грамматик не применим?
5. Как определяются отношения предшествования? Как они используются при выполнении синтаксического анализа?
6. Опишите работу алгоритма синтаксического анализа для грамматик операторного предшествования.
7. Какая грамматика называется остовной?
8. Как организуется совместная работа Flex и Bison?
9. Расскажите о структуре программы на языке Bison.
10. Как описывается грамматика на языке Bison? Приведите пример.

СПИСОК ЛИТЕРАТУРЫ

1. Молчанов А.Ю. Системное программное обеспечение: Учебник для вузов. 3-е изд. – СПб.: Питер, 2010. – 400 с.
2. Ахо А.В., Лам М.С., Сети Р., Ульман Дж.Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд. : Пер. с англ. – М.: Изд. дом Вильямс, 2008. –1184 с.
3. Свердлов С.З. Языки программирования и методы трансляции: Учебное пособие. – СПб.: Питер, 2007. –638 с.
4. Опалева Э.А., Самойленко В.П. Языки программирования и методы трансляции. – СПб.: БХВ-Петербург, 2005. –480 с.
5. Волкова И.А., Руденко Т.В. Формальные грамматики и языки. Элементы теории трансляции: Учебное пособие. – М.: Издательский отдел факультета вычислительной математики и кибернетики МГУ им. М.В. Ломоносова, 1999. – 62 с.
6. Fast Lexical Analyzer. Режим доступа: http://flex.sourceforge.net/.
7. Bison – GNU parser generator. Режим доступа: http://www.gnu.org/software/bison/.
8. Levine J.R. Flex & bison. – O'Reilly Media, Inc., 2009. –274 p.
9. Niemann T. A Compact Guide To Lex & Yacc. Режим доступа: http://epaperpress.com/lexandyacc/download/lexyacc.pdf.

ПРИЛОЖЕНИЕ 1
ЗАДАНИЯ НА ЛАБОРАТОРНЫЕ РАБОТЫ

Лабораторная работа №1. Формальные языки, грамматики и их свойства

Задание на лабораторную работу

1. Дана грамматика. Построить вывод заданной цепочки.

а) $S \to T \mid T{+}S \mid T{-}S$
$T \to F \mid F{*}T$
$F \to a \mid b$
Цепочка $a{-}b{*}a{+}b$

b) $S \to aSBC \mid abC$
$CB \to BC$
$bB \to bb$
$bC \to bc$
$cC \to cc$
Цепочка $aaabbbccc$

2. Построить все сентенциальные формы для грамматики с правилами:

$S \to A{+}B \mid B{+}A$
$A \to a$
$B \to b$

3. Какой язык порождается грамматикой с правилами:

а) $S \to aaCFD$
$AD \to D$
$F \to AFB \mid AB$
$Cb \to bC$
$AB \to bBA$
$CB \to C$
$Ab \to bA$
$bCD \to \varepsilon$

б) $S \to aQb \mid accb$
$Q \to cSc$

в) $S \to A\perp \mid B\perp$
$A \to a \mid Ba$
$B \to b \mid Bb \mid Ab$

г) $S \to 1B$
$B \to B0 \mid 1$

4. Построить грамматику, порождающую язык:

а) $\mathbf{L} = \{a^n b^m c^k \mid n, m, k > 0\}$
б) $\mathbf{L} = \{(ab)^n (cb)^m \mid n, m \geq 0\}$
в) $\mathbf{L} = \{0^n (10)^m \mid n, m \geq 0\}$
г) $\mathbf{L} = \{(ac)^n \mid n > 0, a \in \{b, d\}, c \in \{+, -\}\}$
д) $\mathbf{L} = \{\perp(010)^n \perp \mid n > 0\}$
е) $\mathbf{L} = \{a_1 a_2 \ldots a_n a_n \ldots a_2 a_1 \mid a_i \in \{0, 1\}\}$
ж)* $\mathbf{L} = \{a_1 a_2 \ldots a_n a_1 a_2 \ldots a_n \mid a_i \in \{\mathrm{c}, \mathrm{d}\}\}$
з) * $\mathbf{L} = \{\omega c \omega c \omega \mid \omega \in \{a, b\}^+\}$

5. К какому типу по Хомскому относится грамматика с правилами:

а) $S \to a \mid Ba$
$B \to Bb \mid b$

б) $S \to Ab$
$A \to Aa \mid ba$

в) $S \to 0A1 \mid 01$
$0A \to 00A1$
$A \to 01$

г) $S \to AB$
$AB \to BA$
$A \to a$
$B \to b$

6. Эквивалентны ли грамматики с правилами:

а) $S \to AB$
$A \to a \mid Aa$
$B \to b \mid Bb$

и

$S \to AS \mid SB \mid AB$
$A \to a$
$B \to b$

б) $S \to aSL \mid aL$
$L \to Kc$
$cK \to Kc$
$K \to b$

и

$S \to aSBc \mid abc$
$cB \to Bc$
$bB \to bb$

7. Построить КС-грамматику, эквивалентную грамматике с правилами:

а) $S \to aAb$
$aA \to aaAb$
$A \to \varepsilon$

б) $S \to AB \mid ABS$
$AB \to BA$
$BA \to AB$
$A \to a$
$B \to b$

8. Построить регулярную грамматику, эквивалентную грамматике с правилами:

а) $S \to A \mid AS$
$A \to a \mid bb$

б) $S \to A.A$
$A \to B \mid BA$
$B \to 0 \mid 1$

9. Какие языки обозначаются следующими регулярными выражениями?

а) $(a^*b^*)^*$
б) $a(a|b)^*a$
в) $(aa|bb)^*((ab|ba)(aa|bb)^*(ab|ba)(aa|bb)^*)^*$
г) $a(ba|a)^*$
д) $ab(a|b^*c)^*bb^*a$

10. Постройте контекстно-свободную грамматику для:

а) выражения while в языке Си
б) выражения for в языке Си
в) выражения do-while в языке Си

11. Дана грамматика **G** с правилами $S \to (S)\,S \mid \varepsilon$

Сколько цепочек вывода с n открывающимися скобками можно вывести для этой грамматики при n, равном:

а) 1
б) 2

в) 3
г) 4
д) *m*?

12. Дана грамматика **G** с правилами $E \to E + E \mid E * E \mid (E) \mid \mathbf{id}$
Рассмотрим цепочку ***id* + *id* + ... + *id***, где количество знаков «+» равно *n*. Сколько деревьев вывода можно построить для этой цепочки при *n*, равном:
а) 1
б) 2
в) 3
г) 4
д) *m*?

13. Дана грамматика **G** с правилами $S \to aSb \mid \varepsilon$

Докажите, что $\mathbf{L(G)} = \{ a^n b^n \mid n \geq 0 \}$.

14. Дана грамматика **G**:

$S \to aSbS \mid bSaS \mid \varepsilon$

а) Какой язык порождает эта грамматика?
б) Постройте все возможные деревья вывода для цепочки *abab*.
в) Является ли эта грамматика неоднозначной?

15. Дана грамматика **G** с правилами:

$S \to SS+ \mid SS* \mid a$

а) Какой язык порождает данная грамматика?
б) Устраните левую рекурсию для этой грамматики.

Контрольные вопросы

1. Что такое трансляция, компиляция, транслятор, компилятор?
2. Из каких процессов состоит компиляция? Расскажите об общей структуре компилятора.
3. Дайте определение цепочки, языка. Что такое синтаксис и семантика языка?
4. Какие существуют методы задания языков? Какие дополнительные вопросы необходимо решить при задании языка программирования?
5. Что такое грамматика? Дайте определения грамматики.
6. Как выглядит описание грамматики в форме Бэкуса-Наура?
7. Какие типы грамматик существуют? Что такое регулярные грамматики?
8. Дайте определения контекстно-свободной грамматики, выводимости цепочки, непосредственной выводимости, длине вывода.

Лабораторная работа №2. Регулярные грамматики и конечные автоматы

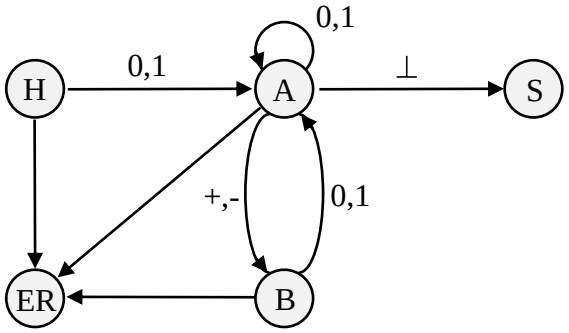
Задание на лабораторную работу

1. Дана регулярная грамматика с правилами:

$S \to S0 \mid S1 \mid P0 \mid P1$
$P \to N.$
$N \to 0 \mid 1 \mid N0 \mid N1$

Построить по ней диаграмму состояний и использовать ДС для разбора цепочек: 11.010 , 0.1 , 01. , 100
Какой язык порождает эта грамматика?

2. Дана ДС.



а) Осуществить разбор цепочек 1011⊥, 10+011⊥ и 0–101+1⊥.
б) Восстановить регулярную грамматику, по которой была построена данная ДС.
в) Какой язык порождает полученная грамматика?

3. Написать леволинейную регулярную грамматику, эквивалентную данной праволинейной, допускающую детерминированный разбор.

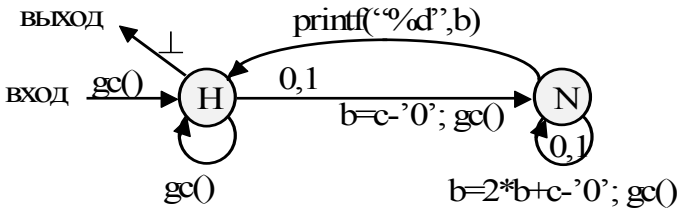
а) $S \to 0S \mid 0B$
$B \to 1B \mid 1C$
$C \to 1C \mid \perp$

б) $S \to aA \mid aB \mid bA$
$A \to bS$
$B \to aS \mid bB \mid \perp$

в) $S \to aB$
$B \to aC \mid aD \mid dB$
$C \to aB$
$D \to \perp$

г) $S \to 0B$
$B \to 1C \mid 1S$
$C \to \perp$

Варианты заданий

Вариант	**Задание**
1	Пусть имеется переменная *c* и функция *gc()*, считывающая в *c* очередной символ анализируемой цепочки. Дана ДС с действиями: 

	а) Определить, что будет выдано на печать при разборе цепочки 1+101//p11+++1000/5⊥? б) Написать на программу-анализатор по этой ДС.
2	Дана грамматика: **G** ({*Q, P, R, S*}, {0, 1, *, $, /}, **V**, *Q*), где **V**: $Q \to 1P$ $P \to *S \mid \$$ $S \to 0R$ $R \to /Q \mid \$$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.
3	Дана регулярная грамматика: $S \to A\perp$ $A \to Ab \mid Bb \mid b$ $B \to Aa$ Определить язык, который она порождает; построить ДС; написать на программу-анализатор.
4	Для данной грамматики: а) определить её тип; б) определить, какой язык она порождает; в) написать регулярную грамматику, почти эквивалентную данной; г) построить ДС и программу-анализатор. $S \to 0S \mid S0 \mid D$ $D \to DD \mid 1A \mid \perp$ $A \to 0B \mid \perp$ $B \to 0A \mid 0$
5	Дана грамматика: $S \to C\perp$ $B \to B1 \mid 0 \mid D0$ $C \to B1 \mid C1$ $D \to D0 \mid 0$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.
6	Дана грамматика: $S \to C\perp$ $C \to B1$ $B \to 0 \mid D0$ $D \to B1$ Определить язык, который она порождает; построить ДС; напи-

	сать программу-анализатор.
7	Дана грамматика: $S \to A0$ $A \to A0 \mid S1 \mid 0$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.
8	Дана грамматика: $S \to 0A \mid 1S$ $A \to 0A \mid 1B$ $B \to 0B \mid 1B \mid \perp$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.
9	Дана грамматика: $S \to B\perp$ $A \to B1 \mid 0$ $B \to A1 \mid C1 \mid B0 \mid 1$ $C \to A0 \mid B1$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.
10	Дана грамматика: $S \to 0\mathrm{S} \mid 0B$ $B \to 1B \mid 1C$ $C \to 1C \mid \perp$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.
11	Дана грамматика: **G** ($\{K, L, M, N\}$, $\{0, 1, \$\}$, **V**, K), где **V**: $K \to 1L \mid 0N$ $L \to 0M$ $N \to 1L \mid 1M$ $M \to \$$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.
12	Дана грамматика: **G** ($\{S, C, D\}$, $\{0, 1\}$, **P**, S), где **P**: $S \to 1C \mid 0D$ $C \to 0D \mid 0S \mid 1$ $D \to 1C \mid 1S \mid 0$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.

13	Дана грамматика: $S \to B0 \mid 0$ $B \to B0 \mid C1 \mid 0 \mid 1$ $C \to B0$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.
14	Дана грамматика: $S \to Sb \mid Aa \mid a \mid b$ $A \to Aa \mid Sb \mid a$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.
15	Дана грамматика: $S \to A0 \mid A1 \mid B1 \mid 0 \mid 1$ $A \to A1 \mid B1 \mid 1$ $B \to A0$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.
16	Дана грамматика: $S \to S0 \mid A1 \mid 0 \mid 1$ $A \to A1 \mid B0 \mid 0 \mid 1$ $B \to A0$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.
17	Дана грамматика: $S \to A\perp$ $A \to A0 \mid A1 \mid B1$ $B \to B0 \mid C0 \mid 0$ $C \to C1 \mid 1$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.
18	Даны две грамматики **G1** и **G2**, порождающие языки **L1** и **L2**. Построить регулярную грамматику для **L1** $\cup$ **L2**. Для полученной грамматики построить ДС и написать программу-анализатор. **G1**: $S \to S1 \mid A0$ $A \to A1 \mid 0$ **G2**: $S \to A1 \mid B0 \mid E1$ $A \to S1$ $B \to C1 \mid D1$ $C \to 0$ $D \to B1$ $E \to E0 \mid 1$
19	Даны две грамматики **G1** и **G2**, порождающие языки **L1** и **L2**.

	Построить регулярную грамматику для **L1** $\cap$ **L2**. Для полученной грамматики построить ДС и написать программу-анализатор. **G1**: $S \to S1 \mid A0$ $A \to A1 \mid 0$ **G2**: $S \to A1 \mid B0 \mid E1$ $A \to S1$ $B \to C1 \mid D1$ $C \to 0$ $D \to B1$ $E \to E0 \mid 1$
20	Дана грамматика: **G** ({*I, J, K, M, N*}, {0, 1, ~, !}, **P**, *I*), где **P**: $I \to 0J \mid 1K \mid 0M$ $J \to {\sim}K \mid 0M$ $K \to {\sim}M \mid 0J \mid 0N$ $M \to 1K \mid !$ $N \to 0I \mid 1I \mid !$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.
21	Дана грамматика: **G** ({*S, A, B, C*}, {*a, b, c*}, **P**, *S*), где **P**: $S \to aA \mid bB \mid aC$ $A \to bA \mid bB \mid c$ $B \to aA \mid cC \mid b$ $C \to bB \mid bC \mid a$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.
22	Дана грамматика: **G** ({*K, L, M, N*}, {*a, b*, +, –, $\perp$}, **P**, *K*), где **P**: $K \to aL \mid bM$ $L \to -N \mid -M$ $M \to +N$ $N \to aL \mid bM \mid \perp$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.
23	Дана грамматика: **G** ({*X, Y, Z, V, W*}, {0, 1, *x, y, z*}, **P**, *X*), где **P**: $X \to yY \mid zZ$ $Y \to 1V$ $Z \to 0W \mid 0Y$ $V \to xZ \mid xW \mid 1$ $W \to 1Y \mid 0$

	Определить язык, который она порождает; построить ДС; написать программу-анализатор.
24	Дана грамматика: **G** ({*S, A, B, C, D*}, {*a, b, c, d*, ⊥}, **P**, *S*), где **P**: $S \to aA \mid bB$ $A \to cC \mid \perp$ $C \to cC \mid cA$ $B \to dD \mid \perp$ $D \to dD \mid dB$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.
25	Дана грамматика: **G** ({*S, A, B, C, D*}, {*a, b, c, d*, #, ⊥}, **P**, *S*), где **P**: $S \to aA \mid bB \mid cC$ $A \to dD$ $B \to \#D$ $D \to dD \mid dB \mid \perp$ $C \to cD$ Определить язык, который она порождает; построить ДС; написать программу-анализатор.

Контрольные вопросы

1. Какие грамматики относятся к регулярным грамматикам? Назовите два класса регулярных грамматик.
2. Можно ли для языка, заданного леволинейной грамматикой, построить праволинейную грамматику, задающую эквивалентный язык?
3. Всякая ли регулярная грамматика является однозначной?
4. В чём заключается отличие автоматных грамматик от других регулярных грамматик? Всякая ли регулярная грамматика является автоматной?
5. Если язык, заданный регулярной грамматикой, содержит пустую цепочку, может ли он быть задан автоматной грамматикой?
6. Что такое конечный автомат (КА)? Дайте определение детерминированного и недетерминированного конечных автоматов.
7. Всегда ли недетерминированный КА может быть преобразован к детерминированному виду (если нет, то в каких случаях)?

Лабораторная работа №3. Таблицы идентификаторов

Задание на лабораторную работу

Написать программу, которая получает на входе набор идентификаторов, организует таблицу по заданному методу и позволяет осуществить многократ-

ный поиск идентификатора в этой таблице. Список идентификаторов задан в виде текстового файла. Длина идентификаторов ограничена 32 символами.

Варианты заданий

№ варианта	Первый метод организации таблицы	Второй метод организации таблицы
1	Простое рехэширование	Метод цепочек
2	Простое рехэширование	Простой список
3	Простое рехэширование	Упорядоченный список
4	Простое рехэширование	Бинарное дерево
5	Рехэширование с помощью псевдослучайных чисел	Простое рехэширование
6	Рехэширование с помощью псевдослучайных чисел	Метод цепочек
7	Рехэширование с помощью псевдослучайных чисел	Простой список
8	Рехэширование с помощью псевдослучайных чисел	Упорядоченный список
9	Рехэширование с помощью псевдослучайных чисел	Рехэширование с помощью произведения
10	Рехэширование с помощью псевдослучайных чисел	Бинарное дерево
11	Рехэширование с помощью произведения	Простое рехэширование
12	Рехэширование с помощью произведения	Метод цепочек
13	Рехэширование с помощью произведения	Простой список
14	Рехэширование с помощью произведения	Упорядоченный список
15	Рехэширование с помощью произведения	Бинарное дерево
16	Метод цепочек	Простой список
17	Метод цепочек	Упорядоченный список
18	Метод цепочек	Бинарное дерево

Контрольные вопросы

1. Что такое таблица идентификаторов, и для чего она предназначена?
2. Какая информация может храниться в таблице идентификаторов?
3. Какие цели преследуются при организации таблицы идентификаторов?
4. Какими характеристиками могут обладать константы, переменные?
5. Какие существуют способы организации таблиц идентификаторов?
6. Что такое коллизия? Почему она происходит?

7. В чём заключается алгоритм логарифмического поиска? Какие преимущества он дает по сравнению с простым перебором, и какие он имеет недостатки?
8. Что такое рехэширование? Какие методы рехэширования существуют?
9. Что такое хэш-функции и для чего они используются?
10. В чём заключается метод цепочек?
11. Расскажите о древовидной организации таблиц.
12. Как могут быть скомбинированы различные методы организации хэш-таблиц?

Лабораторная работа №4. Лексический анализатор

Задание на лабораторную работу

Для выполнения лабораторной работы необходимо:

1) написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием и порождает таблицу лексем с указанием их типов. Программа должна выдавать сообщения о наличии во входном тексте ошибок, которые могут быть обнаружены на этапе лексического анализа;
2) реализовать лексический анализатор с помощью Flex.

Варианты заданий

1. Входной язык содержит арифметические выражения, разделённые символом **;** (точка с запятой). Арифметические выражения состоят из идентификаторов, десятичных чисел с плавающей точкой (в обычной и экспоненциальной форме), знака присваивания (**:=**), знаков операций +, –, *, / и круглых скобок.
2. Входной язык содержит логические выражения, разделённые символом **;** (точка с запятой). Логические выражения состоят из идентификаторов, констант **0** и **1**, знака присваивания (**:=**), операций **or**, **xor**, **and**, **not** и круглых скобок.
3. Входной язык содержит операторы условия **if … then … else** и **if … then**, разделённые символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <, >, =, шестнадцатеричные числа, знак присваивания (**:=**). Шестнадцатеричными числами считать последовательность цифр и символов **a**, **b**, **c**, **d**, **e**, **f**, начинающуюся с цифры (например, 89, 45ac, 0abc).
4. Входной язык содержит операторы цикла **for (…; …; …) do …**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, строковые константы (последовательность символов в двойных кавычках), знак присваивания (**:=**).
5. Входной язык содержит операторы цикла **while (…) … done**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы,

знаки сравнения <, >, =, десятичные числа с плавающей точкой (в обычной и экспоненциальной форме), знак присваивания (**:=**).

6. Входной язык содержит операторы цикла **do … while (…)**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, римские числа, знак присваивания (**:=**). Римскими считать числа, записанные большими буквами **X**, **V** и **I**.
7. Входной язык содержит арифметические выражения, разделённые символом **;** (точка с запятой). Арифметические выражения состоят из идентификаторов, римских чисел, знака присваивания (**:=**), знаков операций +, –, *, / и круглых скобок. Римскими считать числа, записанные большими буквами **X**, **V** и **I**.
8. Входной язык содержит логические выражения, разделённые символом **;** (точка с запятой). Логические выражения состоят из идентификаторов, констант **true** и **false**, знака присваивания (**:=**), операций **or**, **xor**, **and**, **not** и круглых скобок.
9. Входной язык содержит операторы условия **if … then … else** и **if … then**, разделённые символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <, >, =, десятичные числа с плавающей точкой (в обычной и экспоненциальной форме), знак присваивания (**:=**).
10. Входной язык содержит операторы цикла **for (…; …; …) do …**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, шестнадцатеричные числа, знак присваивания (**:=**). Шестнадцатеричными числами считать последовательность цифр и символов **a**, **b**, **c**, **d**, **e**, **f**, начинающуюся с цифры (например, 89, 45ac, 0abc).
11. Входной язык содержит операторы цикла **while (…) … done**, разделённые символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <, >, =, строковые константы (последовательность символов в двойных кавычках), знак присваивания (**:=**).
12. Входной язык содержит операторы цикла **do … while (…)**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <=, =>, =, десятичные числа с плавающей точкой (в обычной и экспоненциальной форме), знак присваивания (**:=**).
13. Входной язык содержит арифметические выражения, разделённые символом **;** (точка с запятой). Арифметические выражения состоят из идентификаторов, шестнадцатеричных чисел, знака присваивания (**:=**), знаков операций +, –, *, / и круглых скобок. Шестнадцатеричными числами считать последовательность цифр и символов **a**, **b**, **c**, **d**, **e**, **f**, начинающуюся с цифры (например, 89, 45ac, 0abc).
14. Входной язык содержит логические выражения, разделённые символом **;** (точка с запятой). Логические выражения состоят из идентификаторов, символьных констант **'T'** и **'F'**, знака присваивания (**:=**), операций **or**, **xor**, **and**, **not** и круглых скобок.

15. Входной язык содержит операторы условия **if … then … else** и **if … then**, разделённые символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <, >, =, римские числа, знак присваивания (**:=**). Римскими считать числа, записанные большими буквами **X**, **V** и **I**.

16. Входной язык содержит операторы цикла **for (…; …; …) do …**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, десятичные числа с плавающей точкой (в обычной и экспоненциальной форме), знак присваивания (**:=**).

17. Входной язык содержит операторы цикла **do … while (…)**, разделённые символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <=, =>, =, строковые константы (последовательность символов в двойных кавычках), знак присваивания (**:=**).

18. Входной язык содержит операторы цикла **while (…) … done**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, римские числа, знак присваивания (**:=**). Римскими считать числа, записанные большими буквами **X**, **V** и **I**.

19. Входной язык содержит арифметические выражения, разделённые символом **;** (точка с запятой). Арифметические выражения состоят из идентификаторов, символьных констант (один символ в одинарных кавычках), знака присваивания (**:=**), знаков операций +, –, *, / и круглых скобок.

20. Входной язык содержит логические выражения, разделённые символом **;** (точка с запятой). Логические выражения состоят из идентификаторов, шестнадцатеричных чисел, знака присваивания (**:=**), операций **or**, **xor**, **and**, **not** и круглых скобок. Шестнадцатеричными числами считать последовательность цифр и символов **a**, **b**, **c**, **d**, **e**, **f**, начинающуюся с цифры (например, 89, 45ac, 0abc).

21. Входной язык содержит операторы цикла **for (…; …; …) do …**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, римские числа, знак присваивания (**:=**). Римскими считать числа, записанные большими буквами **X**, **V** и **I**.

22. Входной язык содержит операторы цикла **do … while (…)**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <=, =>, =, шестнадцатеричные числа, знак присваивания (**:=**). Шестнадцатеричными числами считать последовательность цифр и символов **a**, **b**, **c**, **d**, **e**, **f**, начинающуюся с цифры (например, 89, 45ac, 0abc).

23. Входной язык содержит операторы условия **if … then … else** и **if … then**, разделённые символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <, >, =, строковые константы (последовательность символов в двойных кавычках), знак присваивания (**:=**).

24. Входной язык содержит операторы цикла **while (…) … done**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы,

знаки сравнения <, >, =, шестнадцатеричные числа, знак присваивания (**:=**). Шестнадцатеричными числами считать последовательность цифр и символов **a**, **b**, **c**, **d**, **e**, **f**, начинающуюся с цифры (например, 89, 45ac, 0abc).

Контрольные вопросы

1. Какую роль выполняет лексический анализ в процессе компиляции?
2. Как связаны лексический и синтаксический анализ?
3. Какие проблемы необходимо решить при построении лексического анализатора на основе конечного автомата?
4. Чем отличаются таблица лексем и таблица идентификаторов? В какую из этих таблиц лексический анализатор не должен помещать ключевые слова, разделители и знаки операций?

Лабораторная работа №5. Синтаксический анализатор

Задание на лабораторную работу

Доработать программу лексического анализатора из лабораторной работы № 4 так, чтобы генерируемый ею поток токенов поступал на вход синтаксического анализатора. Выполнить программную реализацию синтаксического анализатора. Результаты работы программы представить в виде дерева разбора.

Синтаксический анализатор реализовать, используя:

1) алгоритм «сдвиг-свёртка» для грамматик операторного предшествования;
2) генератор синтаксических анализаторов Bison.

Контрольные вопросы

1. Какую роль выполняет синтаксический анализ в процессе компиляции?
2. Какие проблемы возникают при построении синтаксического анализатора и как они могут быть решены?
3. Какие типы грамматик существуют? Что такое КС-грамматики? Расскажите об их использовании в компиляторе.
4. Какие типы распознавателей для КС-грамматик существуют? Расскажите о недостатках и преимуществах различных типов распознавателей.
5. Поясните правила построения дерева вывода грамматики.
6. Что такое грамматики простого предшествования?
7. Как вычисляются отношения предшествования для грамматик простого предшествования?
8. Что такое грамматика операторного предшествования?
9. Как вычисляются отношения для грамматик операторного предшествования?
10. Что такое сдвиг, свёртка? Для чего необходим алгоритм «сдвиг-свёртка»?
11. Расскажите, как работает алгоритм «сдвиг-свёртка» в общем случае (с возвратами).
12. Как работает алгоритм «сдвиг-свёртка» без возвратов (объясните на своем примере)?

Ольга Владимировна Молдованова

Языки программирования и методы трансляции

Учебное пособие

Редактор: А.С.Родионов
Корректор: В.В.Сиделина

Подписано в печать 25.06.2012г.
Формат бумаги 60 х 84/16, отпечатано на ризографе, шрифт № 10,
изд. л. 8,4 заказ № 40 , тираж – 100 экз., СибГУТИ.
630102, г. Новосибирск, ул. Кирова, д. 86
134