ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Несмотря на более чем полувековую историю вычислительной техники, рождение теории формальных языков ведет отсчет с 1957 года. В этот год американский ученый Джон Бэкус [1] разработал первый компилятор языка Фортран. Он применил теорию формальных языков, во многом опирающуюся на работы известного ученого-лингвиста Н. Хомского [2] – автора классификации формальных языков. Хомский в основном занимался изучением естественных языков, Бекус применил его теорию для разработки языка программирования. Это дало толчок к разработке сотен языков программирования.

Несмотря на наличие большого количества алгоритмов, позволяющих автоматизировать процесс написания транслятора для формального языка, создание нового языка требует творческого подхода. В основном это относится к синтаксису языка, который, с одной стороны, должен быть удобен в прикладном программировании, а с другой, должен укладываться в область контекстно-свободных языков, для которых существуют развитые методы анализа.

Основы теории формальных языков и практические методы разработки распознавателей формальных языков составляют неотъемлемую часть образования современного инженера-программиста.

Предлагаемое учебное пособие служит целям освоения основных методов анализа языков, построенных на базе формальных грамматик. С практической точки зрения анализ любого формального языка эквивалентен разработке распознавателя данного языка с помощью одного из существующих высокоуровневых средств программирования.

Теория формальных языков неразрывно связана с теорией трансляции (от англ. *translation* – перевод). В этом смысле пособие может быть полезно тем, кто желает приобрести практические навыки написания транслятора языка программирования.

Пособие насчитывает более 50 практических примеров, начиная с самых простых и заканчивая более сложными. Все примеры в тексте имеют сквозную нумерацию. Помимо этого, приводится анализ лексики и синтаксиса модельного языка программирования, включающего стандартный набор управляющих инструкций, арифметических и логических операторов.

1. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ И ПОНЯТИЯ

Алфавит. Цепочка символов в алфавите. Длина цепочки. Пустая цепочка. Конкатенация. Реверс. Формальный язык. Распознаватель. Порождающая грамматика.

Современная теория формальных языков опирается на строгий математический аппарат. Для изложения дальнейшего материала нам потребуется ряд определений. Предполагается, что читатель знаком с основными понятиями теории множеств.

Определение 1. *Алфавитом* V называется конечное множество символов. Здесь под «символом» подразумевается некий *атомарный* элемент множества.

Определение 2. Любая последовательность символом конечной длины в алфавите V называется *цепочкой символов*.

Определение 3. *Пустая цепочка* – цепочка, не содержащая ни одного символа. Пустая цепочка обозначается греческой буквой ε («эпсилон»).

Здесь необходимо отметить, что сама по себе пустая цепочка и буква, которой она обозначается ε, не входят в алфавит V.

Определение 4. *Конкатенацией* называется бинарная операция, операндами, которой служат две цепочки α и β.

Пусть, например, $\alpha = ab$, $\beta = cd$, тогда $\alpha\beta = abcd$ есть конкатенация этих двух цепочек.

Для любой цепочки справедливы следующие равенства:

1. $\alpha\varepsilon = \varepsilon\alpha = \alpha$

2. Для любых цепочек α, β, γ действует свойство ассоциативности, т.е. $(\alpha\beta)\gamma = \alpha(\beta\gamma) = \alpha\beta\gamma$.

Определение 5. *Реверсом* цепочки называется операция обращения цепочка, в результате которой получается цепочка, все символы которой записаны в обратном порядке. Реверс обозначается так $R(\alpha)$.

Очевидны следующие свойства реверса:

1. Реверс пустой цепочки приводит к пустой цепочке, т.е. $R(\varepsilon) = \varepsilon$.

2. Реверс цепочки, состоящей из одного символа приводит к той же самой цепочке.

Определение 6. α^n – возведение в степень n цепочки α – называется -кратная конкатенация цепочки с самой собой, т.е.

$$\alpha^n = \alpha\alpha \ldots \alpha \ (n \text{ раз})$$

Свойствами возведения в степень являются:

1. $\alpha^0 = \varepsilon$

2. $\alpha^n = \alpha^{n-1}\alpha = \alpha\alpha^{n-1}$

Определение 7. Количество символов в цепочке определяет *длину цепочки*. Длина цепочки обозначается так $|\alpha|$.

Очевидно, длина пустой цепочки $|\varepsilon| = 0$.

Определение 8. Множество V^* над алфавитом V называется множество всех цепочек, включая пустую цепочку.

Например, пусть алфавит состоит из символов 0 и 1, $V = \{0,1\}$. Тогда, $V^* = \{\varepsilon, 0,1,00,01,10,11,000,001 \dots\}$.

Определение 9. Множество V^+ над алфавитом V называется множество всех цепочек, не включая пустую цепочку. Аналогично рассмотренному выше примеру $V^+ = \{0,1,00,01,10,11,000,001, \dots\}$.

Приведенные определения дают возможность ввести понятие *язык*.

Определение 10. *Я*зыком L в алфавите V называется подмножество всех цепочек, определенных для данного алфавита, т.е. $L \subseteq V$.

Согласно данному определению в язык могут быть включены не все возможные цепочки символов данного алфавита. Если множество всех цепочек, составляющих язык конечно, то язык можно определить простым перечислением всех возможных цепочек. Однако, большинство языков включают бесконечное множество цепочек, которые составляются по определенным правилам. Поэтому существует два подходя для определения бесконечных языков.

Первый подход основывается на процедуре распознавания. Данная процедура отвечает на вопрос о принадлежности цепочки языку. Говорят, что цепочка принадлежит языку или допускается языком, если распознаватель признает её, т.е. процедура распознавания завершается с ответом «да». Иначе, распознаватель завершается с ответом «нет». Таким образом, определение языка таково: язык – это множество всех цепочек, которые он допускает.

Распознаватель можно схематично представить в виде совокупности входной ленты, читающей головки, которая указывает на очередной символ на ленте, устройства управления (УУ) и дополнительной памяти (стек).

Конфигурацией распознавателя является:

- состояние УУ;
- содержимое входной ленты;
- положение читающей головки;
- содержимое дополнительной памяти (стека).

Существует множество примеров распознавателей. Вот некоторые из них:

1. Машина Тьюринга [3], допускает все рекурсивно-перечислимые языки. Это наиболее общий вид распознавателей.

2. Линейно-ограниченный автомат, допускает контекстно-зависимые языки. Автомат отличается от машины Тьюринга тем, что его лента не бесконечна.

3. Автомат со стековой памятью, допускает все контекстно-свободные языки. Данный автомат в отличие от линейно-ограниченного автомата, не позволяет читающей головке перемещаться по ленте влево и не позволяет изменять входное слово.

4. Конечный автомат допускает регулярные языки и является наиболее ограниченным типом распознавателей формальных языков.

Данные примеры следуют классификации Хомского [4], о которой будет идти речь ниже.

Вторым способом описания бесконечных языков является механизм порождения, основанный на порождающих грамматиках. Использование порождающих грамматик является самым распространенным средством описания формальных языков в целом и языков программирования в частности.

1.1. Порождающие грамматики

Для дальнейшего нам потребуется следующее определение.

Определение 11. Декартово произведение двух множеств, обозначаемое $A \times B$, есть множество, элементами которого являются все пары (a, b), такие что $a \in A$ и $b \in B$.

Определение 12. Порождающая грамматика есть четверка элементов $G =< T, N, P, S >$, где

T – множество терминальных символов (иначе, терминалов);

N – множество нетерминальных символов (иначе, нетерминалов);

P – множество правил вывода, вида $\alpha \to \beta$. Здесь α называют левой частью правила, а β – правой. Символ «→» читается, как «может иметь вид».

S – начальный символ (иначе, цель) грамматики.

Рассмотрим эти множества подробнее. *Терминалами* называют конечные неделимые элементы языка, которые нельзя вывести из других элементов. Другими словами, терминалы составляют алфавит, из которого строятся все цепочки, принадлежащие языку, определяемому данной порождающей грамматикой.

Нетерминалы служат для описания того, как можно из одних цепочек по определенным правилам получить другие цепочки, принадлежащие данному языку. Множества терминальных и нетерминальных символов не пересекаются, т.е. $T \cap N = \emptyset$.

Правила вывода, как раз и являются этими правилами перевода одних допустимых цепочек в другие. Множество P представляет собой декартово про-

изведение множеств $(T \cup N)^+ \times (T \cup N)^*$. Здесь левая часть каждого правила должна включать хотя бы один нетерминал.

Грамматика используется для генерации последовательностей символов, которые составляют строки языка. Процесс, в ходе которого к каждому нетерминалу, начиная с S, последовательно применяется правило из числа продукций P, продолжается до тех пор, пока в строке не останутся одни терминальные символы.

Правила вывода по-другому называют *продукциями*. Продукции используются для генерации возможных строк языка по следующему алгоритму:

1. Начать с символа S, заменить его строкой справа от →.

2. Продолжать замену до тех пор, пока в строке есть хотя бы один символ нетерминальный символ.

Из множества всех нетерминалов выделяют один $S \in N$ – *цель грамматики*, с которого начинается вывод (порождение) всех цепочек данного языка.

Введем следующие удобные сокращения. Одной и той же левой части некоторого правила вывода может соответствовать несколько альтернативных правых частей. Вместо записи

$$\alpha \to \beta_1$$
$$\alpha \to \beta_2$$
$$\dots$$
$$\alpha \to \beta_n$$

мы будем использовать сокращенную запись

$$\alpha \to \beta_1 | \beta_2 | \dots | \beta_n \qquad (1.1)$$

Пример 1. Простейшая грамматика, состоящая из трех правил вывода, имеет вид: $G_1 =< \{0,1\}, \{A, S\}, P, S >$, где правила вывода таковы:

$$S \to 0A1$$
$$0A \to 00A1$$
$$A \to \varepsilon$$

Забегая вперед, скажем, что данная грамматика является неукорачивающей контекстно-зависимой грамматикой.

Определение 13: если в грамматике существует правило $\gamma \to \delta$ и $\alpha \to f(\gamma)$ и $\beta \to g(\delta)$, то говорят, что цепочка β *непосредственно выводится* из α.

Например, в приведенной выше грамматике цепочка $00A11$ непосредственно выводима из цепочки $0A1$. В самом деле, если к левой и правой частям второго правила прибавить символ 1, то из цепочки $0A1$ непосредственно получаем $00A11$.

Определение 14. Если при последовательном применении правил вывода можно из цепочки α за конечное число шагов получить цепочку β, т.е. $\alpha \to \gamma_1 \to \gamma_2 \to \cdots \to \gamma_n \to \beta$, то говорят, что цепочка β *выводима* из цепочки α. Последовательность правил $\gamma_1, \gamma_2, \ldots, \gamma_n$ называется выводом длины n.

На примере грамматики G_1 цепочку $000A111$ можно получить из цели грамматики за 3 шага: $S \to 0A1 \to 00A11 \to 000A111$.

Пришло время связать формально эти два понятия – язык и грамматику.

Определение 15. язык L, порождаемый заданной грамматикой G, есть множество цепочек, которые можно вывести из стартового символа S. Это значит, что с помощью правил вывода P можно получить все строки этого языка. И обратно, нельзя породить ни одну строку, не принадлежащую данному языку.

Пример 2. Пусть язык формально определен так $L = \{x^n | n > 0\}$. Данный язык допускает все строки вида: *x, xx, xxx* и т.д., т.е. строки, состоящие из произвольного количества символов *x*. Данному языку соответствует грамматика $G =< \{x\}, \{S\}, P, S >$, где правило вывода P таково:

$$S \to xS|x$$

Пример 3. Пусть язык определен так $L = \{x^n y^n | n > 0\}$. В данном случае язык определяет строки, состоящие из произвольного количества символов *x*, за которым следует точно такое же количество символов *y*. Данному языку соответствует грамматика $G =< \{x, y\}, \{S\}, P, S >$ с одним правилом вывода:

$$S \to xSy|xy$$

Пример 4. Пусть язык определен так $L = \{x^n y^m | n, m \geq 0\}$. Данные язык отличается от предыдущего, так как определяет строки, состоящие из произвольного количества символов *x*, за которым следует произвольное количество символов *y*, причем n не зависит от m. Кроме того, данный язык допускает пустую строку ε. Грамматика, порождающая данный язык $G =< \{x, y\}, \{S, B\}, P, S >$. Здесь правила вывода таковы:

$$S \to xS$$
$$S \to yB$$
$$S \to x$$
$$S \to y$$
$$S \to \varepsilon$$
$$B \to yB$$
$$B \to y$$

Определение 16. *Сентенциальная форма* представляет собой некоторую цепочку символов, состоящую из терминалов и нетерминалов $\alpha \in (T \cup N)^*$, та-

кую что ее можно получить, последовательно применяя правила вывода. Другими словами $S \Rightarrow \alpha$. Итоговая форма, состоящая из одних терминалов, называется *сентенцией* (от англ. *sentence* – предложение).

Множество сентенциальных форм, в которых присутствуют только терминальные символы, представляет собой все строки данного языка.

Определение 17. Грамматики G_1 и G_2 называются *эквивалентными*, если они порождают один и тот же язык (одно и то же множество строк).

Пример 5. Рассмотренная выше грамматик G_1 эквивалентна следующей грамматике

$$S \to 0S1|01$$

В самом деле, и та и другая грамматика порождают язык $L = \{0^n 1^n | n > 0\}$.

Определение 18. Если две грамматики G_1 и G_2 порождают языки L_1 и L_2, которые отличаются не более чем на пустую строку, то такие грамматики называют *почти эквивалентными*.

Пример 6. Рассмотренная выше грамматика G_1 почти эквивалентна следующей грамматике

$$S \to 0S1|\varepsilon$$

В самом деле, в данном случае получается язык $L = \{0^n 1^n | n \geq 0\}$, который допускает пустую строку. Грамматика G_1 пустую строку не допускает.

1.2. Классификация грамматик по Хомскому

Сделаем одно замечание. Строчные буквы мы будем использовать для обозначения терминалов, а заглавные (как в примерах выше) для обозначения нетерминалов.

Рассмотрим более сложный пример чем те, что мы рассматривали ранее.

Пример 7. Пусть дана грамматика $G_3 = (\{a\}, \{S, N, Q, R\}, P, S)$, для которой правила вывода следующие:

$$S \to QNQ$$
$$QN \to QR$$
$$RN \to NNR$$
$$RQ \to NNQ$$
$$N \to a$$
$$Q \to \varepsilon$$

Согласно этим правилам можно подменить нетерминал N на R, но только в случае, если перед N стоит Q. Далее, R можно заменить на NN, только если после R следует Q.

Данная грамматика относится к классу *контекстно-зависимых* грамматик. Для данного примера возможны следующие порождения:

$S \Rightarrow QNQ \Rightarrow QRQ \Rightarrow QNNQ \Rightarrow aa$

$S \Rightarrow QNQ \Rightarrow QRQ \Rightarrow QNNQ \Rightarrow QRNQ \Rightarrow QNNRQ \Rightarrow QNNNNQ \Rightarrow aaaa$

Можно показать [5], что число терминалов (a) в получающихся строках составляет степень двойки. Другими словами, данная грамматика генерирует следующий язык:

$$\{a^m | m = 2,4, \dots, 2^n; n > 0\}$$

Согласно общепринятой классификации (*иерархии Хомского*) все грамматики подразделяются на 4 типа в зависимости от ограничений, накладываемых на правила вывода.

К грамматике **0-го типа** относятся все грамматики без каких бы то ни было ограничений. Данный тип грамматик эквивалентен множеству рекурсивно-перечислимых языков.

1.2.1. Грамматики с ограничениями на правила вывода

К грамматике **1-го типа** относятся грамматики, у которых для всех продукций $\alpha \to \beta$ длина строки α не больше длины строки β. Математически это можно выразить так:

$$|\alpha| \leq |\beta| \qquad (1.2)$$

У данного правила есть одно исключение: допускается продукция вида $S \to \varepsilon$, хотя формально здесь длина строки S больше длины строки ε.

Подобные грамматики называются *контекстно-зависимыми*. Языки, порождаемые контекстно-зависимыми грамматиками, называют контекстно-зависимыми языками.

В литературе контекстно-зависимые грамматики еще называют неукорачивающими, благодаря правилу (1.2). Можно показать, что класс контекстно-зависимых грамматик (и языков) совпадает с классом неукорачивающих грамматик (языков) [6].

К грамматикам **2-го типа** относятся все грамматики 1-го типа, у которых в левых частях продукций может находиться только один нетерминал. Такие грамматики называются *контекстно-свободными* (КС). Согласно определению, правила вывода для КС-грамматик имеют вид:

$$A \to \beta$$

Здесь $A \in N, \beta \in (T \cup N)^*$.

Так же как и контекстно-зависимые грамматики КС-грамматики допускают правило вывода вида $S \to \varepsilon$.

Язык, порождаемый КС-грамматикой, называется *контекстно-свободным* языком.

К грамматике **3-го типа** относят все КС-грамматики, для которых все продукции могут быть вида:

$$A \to a|bC \tag{1.3}$$

или

$$A \to a|Cb \tag{1.4}$$

В первом случае, грамматика называется *праволинейной*, а во втором *леволинейной*. Праволинейные и леволинейные грамматики образуют класс *регулярных* грамматик. Важно отметить, что, если грамматика включает в себя и праволинейные и леволинейные продукции, она уже не будет являться регулярной.

Утверждение 1. Если для языка L существует леволинейная порождающая грамматика $G_l(L)$, то для него существует эквивалентная праволинейная грамматика $G_r(L)$.

Таким образом, праволиненые и леволинейные грамматики порождают один и тот же класс *регулярных* языков.

По-другому регулярные грамматики еще называют *автоматными*.

1.2.2. Иерархия Хомского

Следующие соотношения справедливы для любой формальной грамматики.

1. Любая регулярная грамматика является одновременно КС-грамматикой.

2. Любая КС-грамматик является одновременно контекстно-зависимой грамматикой.

3. Любая контекстно-зависимая грамматика является одновременно грамматикой типа 0.

Данные соотношения мы приводим без доказательств, однако они напрямую вытекают из определений типов грамматик, представленных выше. Приведем несколько примеров.

Пример 8. Язык

$$L = \{a^n b^n | n > 0\}$$

является КС-языком, однако не является регулярным. Доказательство этому будет дано после того, как будут введены такие понятия, как регулярное множество и регулярное выражение.

Пример 9. Язык

$$L = \{a^n b^n c^n | n > 0\}$$

является контекстно-зависимым языком, однако не является КС-языком. Доказательство этому можно найти, например, в [7].

Язык, генерируемый регулярной грамматикой, называется регулярным. Между регулярным языком, регулярными выражениями и конечными автоматами существует тесная связь. С регулярными выражениями мы познакомимся ниже. Таким образом, любой регулярный язык можно описать регулярным выражением, а любое регулярное выражение можно описать *конечным автоматом*. Схематически это представлено на рис. 1.1.

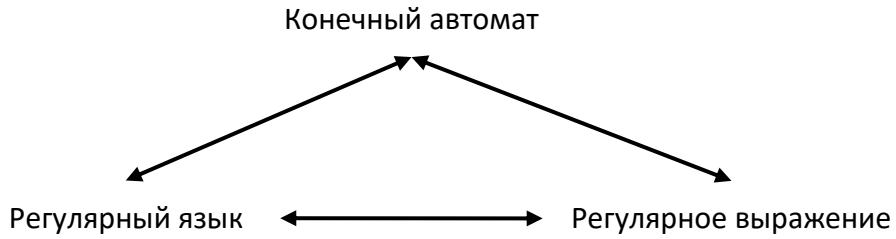

Рис. 1.1

Аналогично, грамматики 2-го типа допускаются автоматами с магазинной (стековой) памятью, грамматики 1-го типа – линейной ограниченными автоматами, а грамматики 0-го типа – полной машиной Тьюринга.

Для справки, **машина Тьюринга** является обобщением конечного автомата, содержащего бесконечную ленту, в которой хранятся данные (символы) необходимые для выполнения некоторого алгоритма. Можно показать, что на машине Тьюринга можно вычислить любой алгоритм, который можно разложить на последовательность элементарных действий (полнота по Тьюрингу).

Иерархия типов грамматик, введенная американским лингвистом Н. Хомским, схематически может быть представлена так, как показано на рис. 1.2.

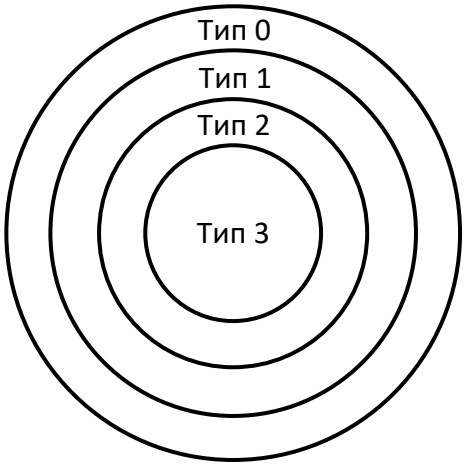

Рис. 1.2

Для нас наибольший интерес представляют уровни 2 и 3 на данной схеме, так как большинство языков программирования относятся именно к типу 2, т.е. являются контекстно-свободными (КС). На практике, даже если в языке присутствуют зависящие от контекста структуры, для него строят КС-грамматику, а эти особенности учитывают на этапе семантического анализа.

Отметим, что язык данного типа можно описать грамматикой данного типа или грамматикой более общего типа.

Пример 10. Выше мы определили язык $L = \{x^n y^m | x, y \geq 0\}$, как регулярный, т.е. 3-го типа. Соответствующая регулярная грамматика такова:

$$S \to xS$$
$$S \to yB$$
$$S \to x$$
$$S \to y$$
$$S \to \varepsilon$$
$$B \to yB$$
$$B \to y$$

В то же время данный язык можно описать и контекстно-свободной грамматикой, т.е. грамматикой 2-го типа:

$$S \to XY$$
$$X \to xX$$
$$X \to \varepsilon$$
$$Y \to yY$$
$$Y \to \varepsilon$$

Утверждение 2. Если для заданного языка существует грамматика типа k, то не существует метода, позволяющего сказать, существует ли для данного языка эквивалентной грамматики типа k+1. Другими словами, если язык является языком типа k (k=1,2,3), то он автоматически является языком типа k−1, однако обратное утверждение неверно. Это утверждение является прямым следствием диаграммы на рис. 1.2.

Вообще, чем выше тип языка, тем легче построить для него распознаватель. Наиболее простыми для анализа являются регулярные языки. Поэтому, естественно попытаться ответить на вопрос, можно ли описать заданный язык возможно более высоким типом грамматики.

Пример 11. Пусть дана грамматика G_1 со следующими правилами вывода:

$$S \to 0A1$$
$$A \to 0A0$$
$$A \to \varepsilon$$

Необходимо отнести данную грамматику к некоторому типу. Очевидно, что данная грамматика удовлетворяет ограничениям, накладываемым на грамматики типа 2, т.е. КС-грамматики. Таким образом, грамматика G_1 является грамматикой типа 2. Однако, она не является регулярной.

С другой стороны язык, описываемый КС-грамматикой G_1 является регулярным. В самом деле, язык включает все строки, начинающие с нуля, за которым следует число нулей, кратное 2, и оканчивающиеся на 1. Данное описание позволяет задать язык следующей грамматикой:

$$S \to 0A$$
$$A \to 0B|1$$
$$B \to 0A$$

Данная грамматика является регулярной. Следовательно, язык, который мы хотим описать, является языком типа 3, т.е. регулярным языком.

Пример 12. Пуст дана грамматика G_2 со следующей продукцией:

$$S \to aSb|\varepsilon$$

Данная грамматика, очевидно, является грамматикой типа 2. Язык, который она порождает таков $L = \{a^n b^n | n \geq 0\}$. Данный язык, как будет показано ниже, невозможно описать регулярной грамматикой, следовательно, язык является языком типа 2.

1.2.3. Примеры формальных языков и грамматик

Рассмотрим несколько примером.

Пример 13. Язык, описываемый грамматикой

$$S \to aS|a \tag{1.5}$$

включает все строки вида $L = \{a^n | n > 0\}$. Данный язык является регулярным (автоматным) языком, и может быть порожден как праволинейной грамматикой (1.5), так и леволинейной

$$S \to Sa|a \tag{1.6}$$

Пример 14. Грамматика из предыдущего примера, допускающая пустые строки, может быть переписана следующим образом

$$S \to aS|\varepsilon \tag{1.7}$$

Данная грамматика является праволинейной регулярной грамматикой, и язык, который ей соответствует, может быть описан так $L = \{a^n | n \geq 0\}$.

Пример 15. Рассмотрим следующую грамматику

$$\begin{aligned} S &\to A \perp | B \perp \\ A &\to a|Ba \\ B &\to b|Bb|Ab \end{aligned} \tag{1.8}$$

Данная грамматика леволинейная регулярная. Здесь символ $\perp$ является символом конца строки. Таким образом, все строки языка, порождаемого данной грамматикой, заканчиваются на символ конца. Эти строки включают все цепочки, состоящие и символов a и b, которые не содержат двух, идущих подряд символов aa.

Пример 16. Рассмотрим грамматику со следующими правилами вывода

$$S \to aQb|accb$$
$$Q \to cSc$$

Данная грамматика является контекстно-свободной, т.е. относится к типу 2. Язык, который она порождает может быть описан так $L = \{(ac)^n(cb)^n|n > 0\}$. Для данного языка не существует порождающей его регулярной грамматики.

Пример 17. Рассмотрим грамматику

$$S \to aSBC|abC \quad (1.9)$$
$$CB \to BC \quad (1.10)$$
$$bB \to bb \quad (1.11)$$
$$bC \to bc \quad (1.12)$$
$$cC \to cc \quad (1.13)$$

Данная грамматика является примером контекстно-зависимой грамматики, о чем свидетельствуют правила (1.10) – (1.13). Данная грамматика порождает следующий язык

$$L = \{a^n b^n c^n|n > 0\} \quad (1.14)$$

Данный язык уже встречался нам выше. В [7] доказывается, что для данного языка не существует КС-грамматики, поэтому он является языком типа 1.

Пример 18. Рассмотрим грамматику

$$S \to SS$$
$$SS \to \varepsilon$$

Эта, несложная не первый взгляд грамматика относится к грамматикам типа 0, равно как и язык, ею порождаемый. Интересно отметить, что несмотря на то, что данная грамматика не имеет никаких ограничений на правила вывода, порождаемый ею язык состоит из одной единственной цепочки $L = \{\varepsilon\}$.

1.2.4. Связь между языком и грамматикой

Несмотря на множество приведенных примеров языков и грамматик, для доказательства того, что язык может быть порожден данной грамматикой необходимо ответить на следующие вопросы:

1. Правила вывода позволяют получить любую цепочку, принадлежащую данному языку.

2. Правила вывода не позволяют получить ни одну цепочку, которая не принадлежит данному языку.

Пример 19. Покажем, что следующая грамматика с правилами вывода (1.9) – (1.13) порождает язык (1.14).

Все цепочки, принадлежащие языку, содержат равное количество символов a, b и c, следующих друг за другом. Докажем методом математической индукции, что любая такая цепочка может быть получена применением правил (1.9) – (1.13).

1) цепочка abc получается применением правила (1.9) и (1.12). В самом деле, $S \to abC \to abc$. Этот случай соответствует $n = 1$.

2) для случая $n > 1$: $S \to aSBC \to aaSBCBC \to \cdots \to a^{n-1}S(BC)^{n-1} \to a^nbC(BC)^{n-1} \to \cdots \to a^nbB^{n-1}C^n \to a^nbbB^{n-2}C^n \to \cdots \to a^nb^nC^n \to a^nb^ncC^{n-1} \to a^nb^nccC^{n-2} \to \cdots \to a^nb^nc^n$.

Мы показали, что любая цепочка вида $a^nb^nc^n$ может быть получена путем применения правил грамматики, но мы не показали, что какая-нибудь другая цепочка, не принадлежащая языку, не может быть получена, путем применения данных правил. Покажем это.

1. Новый символ a, b или B, а также c или C, могут появиться в сентенциальной форме только в равном количестве.

2. Нетерминал B можно заменить только на терминал b.

3. Нетерминал C можно заменить только на терминал c.

4. В любой сентенциальной форме терминал a всегда расположен левее терминала b, который, в свою очередь, расположен левее терминала c.

5. Терминал b может появиться только в результате применения правила (1.9).

6. Согласно правилу (1.11) нетерминал B заменяется на терминал b, тогда и только тогда, когда слева от него уже стоит терминал b. Другими словами, при применении правил грамматики терминал b появляется только справа от уже прочитанной строки.

7. Правило (1.12) применяется только после того, как в сентенциальной форме все нетерминалы B уже заменены на терминал b. Из этого следует, что символы c могут появляться только справа от символов b.

Таким образом, доказано, любая цепочка, выводимая из данных грамматических правил, содержит равное количество символов a, b и c, причем символы b следуют за a, символы c следуют за b, т.е. язык имеет вид (1.14). Что и требовалось доказать.

1.2. Разбор цепочек

Основной задачей анализа формального языка является построение вывода некоторой цепочки путем последовательного применения грамматических правил вывода. Это процесс называется *разбором*. Разбор строки можно осуществлять двумя способами. Первый способ состоит в том, что в зависимости от текущего символа строки применяется то или иное правило, начиная с правила для стартового символа S.

Второй способ состоит в том, что в исходной строке ищется подстрока, соответствующая правой части некоторого грамматического правила, и эта подстрока заменяется (*сворачивается*) в левую часть данного правила. Этот процесс повторяется до тех пор, пока сентенциальная форма не свернется к цели грамматики S.

Грамматики 0-го и 1-го типов практически не используются для анализа языков программирования, поскольку для них не существует стандартных алгоритмов разбора. Это не значит, что язык 0-го и 1-го типов невозможно проанализировать. Это означает только, что процесс анализа требует индивидуального «ручного» подхода.

Для целей трансляции языков программирования ключевой интерес представляют контекстно-свободные и регулярные грамматики, так как для них разработаны эффективные алгоритмы разбора. КС-грамматик как правило достаточно для того, чтобы описать все основные особенности современных языков программирования.

Рассмотрим подробнее процесс разбора цепочки по КС-грамматике. Напомним, что в КС-грамматике в левой части каждого правила вывода стоит один нетерминал.

Определение 19. Если каждая следующая сентенциальная форма отличается от предыдущей тем, что в нем заменен самый левый нетерминал, то такой вывод называется *левосторонним*.

Определение 20. Если каждая следующая сентенциальная форма отличается от предыдущей тем, что в нем заменен самый правый нетерминал, то такой вывод называется *правосторонним*.

Важно отметить, что при заданной грамматики вывод данной цепочки можно получить по-разному, применяя продукции в разном порядке. Например, пусть дана цепочка $a + b + a$. Получить вывод данной цепочки в грамматике G_4

$$S \to T|T + S$$
$$T \to a|b$$

можно так:

1) $S \to T + S \to T + T + S \to T + T + T \to a + T + T \to a + b + T \to a + b + a$

2) $S \to T + S \to a + S \to a + T + S \to a + b + S \to a + b + T \to a + b + a$

3) $S \to T + S \to T + T + S \to T + T + T \to T + T + a \to T + b + a \to a + b + a$

Здесь вывод (2) – левосторонний, вывод (3) – правосторонний, а вывод (1) – не является ни правосторонним, ни левосторонним. Можно видеть, что каким бы ни был вывод, на каждом шаге согласном правилам грамматики заменяется один единственный нетерминал. Введем важное понятие.

Определение 21. *Деревом вывода* для заданной грамматики $G =< T, N, P, S >$ называется древовидная структура данных, которая удовлетворяет набору условий:

1. Каждая вершина дерева – это либо терминал T, либо нетерминал N, либо, если грамматика допускает пустую строку, ε.

2. Корень дерева есть стартовый символ – цель грамматики S.

3. Листьями дерева могут быть только терминальные символы.

4. Потомки $a_1, a_2, \dots, a_i$ каждого узла A дерева соответствуют правилу вывода $A \to a_1 a_2 \dots a_i$.

5. Если грамматика содержит правило вывода $A \to \varepsilon$, то этому правилу соответствует узел дерева, единственным потомком которого является лист ε.

Пример дерева вывода для цепочки $a + b + a$ в грамматике G_4 приведен на рис. 1.3.

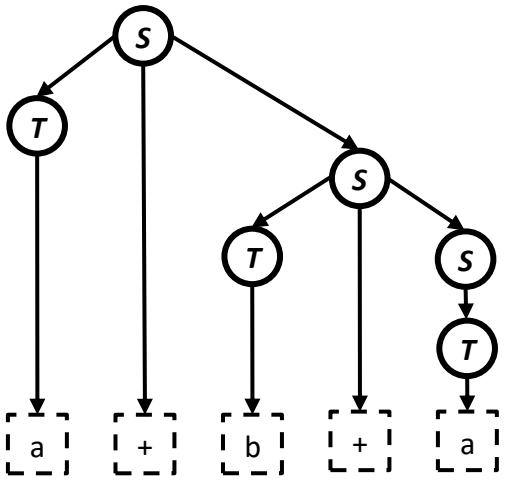

Рис. 1.3

Определение 22. *Неоднозначная* грамматика есть грамматика, в которой для одной и той же цепочки существует несколько различных деревьев вывода.

Определение 23. Язык, порожденный неоднозначной грамматикой, называется *неоднозначным*.

Рассмотрим пример неоднозначной грамматики, которая встречается едва ли не в каждом языке программирования.

Пример 20. Дана грамматика

$$S \rightarrow \boldsymbol{if}\ b\ \boldsymbol{then}\ S\ \boldsymbol{else}\ S \mid \boldsymbol{if}\ b\ \boldsymbol{then}\ S \mid a \qquad (1.15)$$

Данная грамматика представляет собой условный оператор. Здесь терминалами являются ключевые слова ***if***, ***then***, ***else***, а также два предиката a и b.

Покажем, что данная грамматика неоднозначна. Для этого необходимо для одной и той же цепочки построить два различных дерева вывода. Два эквивалентных дерева вывода для цепочки $\boldsymbol{if}\ b\ \boldsymbol{then}\ \boldsymbol{if}\ b\ \boldsymbol{then}\ a\ \boldsymbol{else}\ a$ показаны на рис 1.4 (а) и (б).

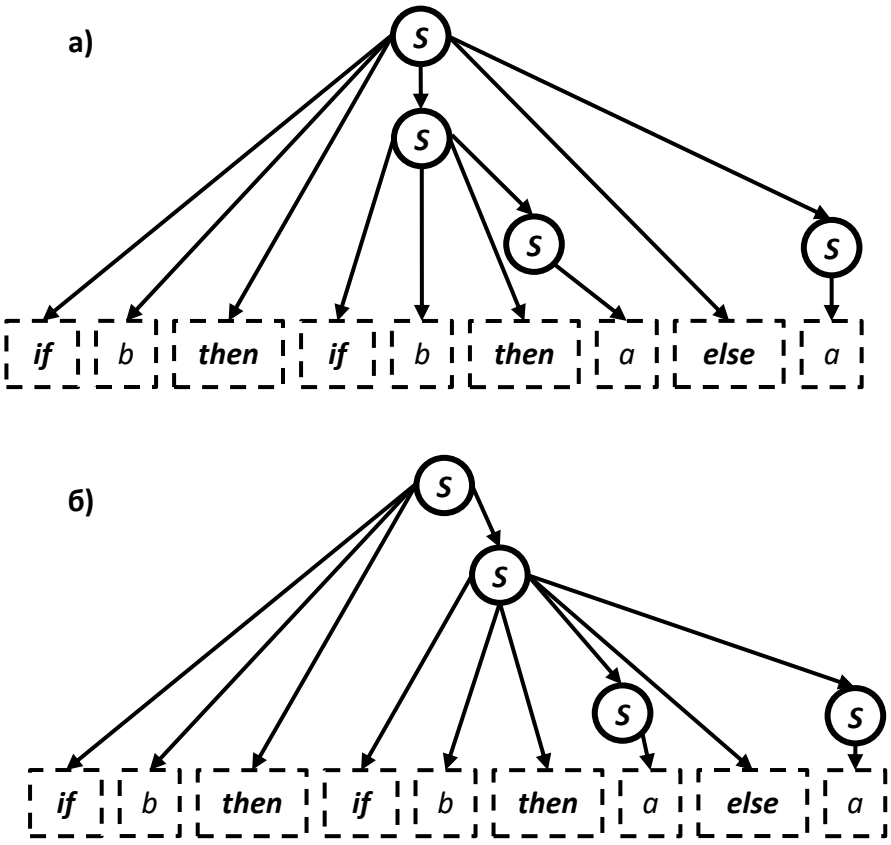

Рис. 1.4

Важно отметить, что в данном случае неоднозначность грамматики не говорит о неоднозначности языка. Другими словами, можно найти эквивалентную данному языку грамматику, которая будет однозначной. Естественно, что грамматики, применяемые для языков программирования, должны быть однозначными.

Для грамматики из примера 20 эквивалентной однозначной грамматикой будет следующая

$$S \rightarrow \boldsymbol{if}\ b\ \boldsymbol{then}\ S \mid \boldsymbol{if}\ b\ \boldsymbol{then}\ S'\boldsymbol{else}\ S \mid a$$
$$S' \rightarrow \boldsymbol{if}\ b\ \boldsymbol{then}\ S'\boldsymbol{else}\ S' \mid a$$

Заметим, что в этом, как и в большинстве прочих случаев, неоднозначность устраняется за счет введения в грамматику дополнительных нетерминалов.

Утверждение 3. Не существует алгоритма, который по заданным грамматическим правилам даст ответ на вопрос «однозначна грамматика или нет».

Для доказательства неоднозначности грамматики достаточно найти цепочку с двумя различными деревьями вывода. С другой стороны, для доказательства однозначности грамматики, необходимо доказать, что таких цепочек не существует.

Дерево вывода можно строить двумя различными способами. Первый способ отличается тем, что строит дерево, начиная с корня и следует к листьям. Данный способ построения называется *нисходящим* методом разбора. Суть метода заключается в том, чтобы для каждого нетерминала (каждой вершины дерева) найти подходящее грамматические правило вывода, которое бы проецировалось на входную цепочку.

Другой способ заключается в том, что разбор начинается с листьев и продвигается вверх, к корню дерева (цели грамматики *S*). Данный метод построения называется *восходящим* разбором и подразумевает использование *сверток*, заменяющих символы исходной цепочки на терминалы в соответствии с правилами вывода.

Каким бы ни был способ разбора, если грамматика однозначна, должно получиться единственное дерево разбора.

2. ЭЛЕМЕНТЫ ТЕОРИИ ТРАНСЛЯЦИИ

Трансляция. Компиляция. Интерпретация. Лексический анализ. Синтаксический анализ. Семантический анализ. Регулярные выражения. Регулярные множества. Конечный автомат. Детерминированных конечный автомат. Недетерминированный конечный автомат.

Трансляция представляет собой перевод исходного текста программы, написанной на некотором языке программирования, во внутреннее представление, понятное машине. В общем случае, к трансляции относят компиляцию, при которой внутренним представлением является последовательность машинных команд, и интерпретацию, при которой программа выполняется на некоторой виртуальной машине инструкция за инструкцией. Мы не будет разделять эти понятия и будем пользоваться одним термином *трансляция*. Транслятор обычно выполняет свою работу в два этапа:

1. Этап *анализа* исходного текста.

2. Этап *синтеза*, в результате которого генерируется машинно-ориентированное представление.

Процесс трансляции схематично показан на рис. 2.1.

Рис. 2.1

Важно отметить, что первый этап трансляции – этап анализа исходного текста – успешно поддается автоматизации. Другими словами, существуют программные средства – лексические и синтаксические анализаторы – которые умеют генерировать компиляторы. Среди подобных систем наибольшую известность имеют *Lex* и *Yacc*, которые называют компиляторами компиляторов [8].

Они компилируют описания сканера и парсера, записанные на специальном языке, в программу на языке *C*, которая способна по заданному описанию языка выполнять лексический и синтаксический анализ исходных текстов.

Транслятор языка программирования представляет собой компьютерную программу, которая сама может быть написана на языке программирования высокого уровня. Хорошо спроектированный компилятор имеет модульную структуру (рис. 2.2).

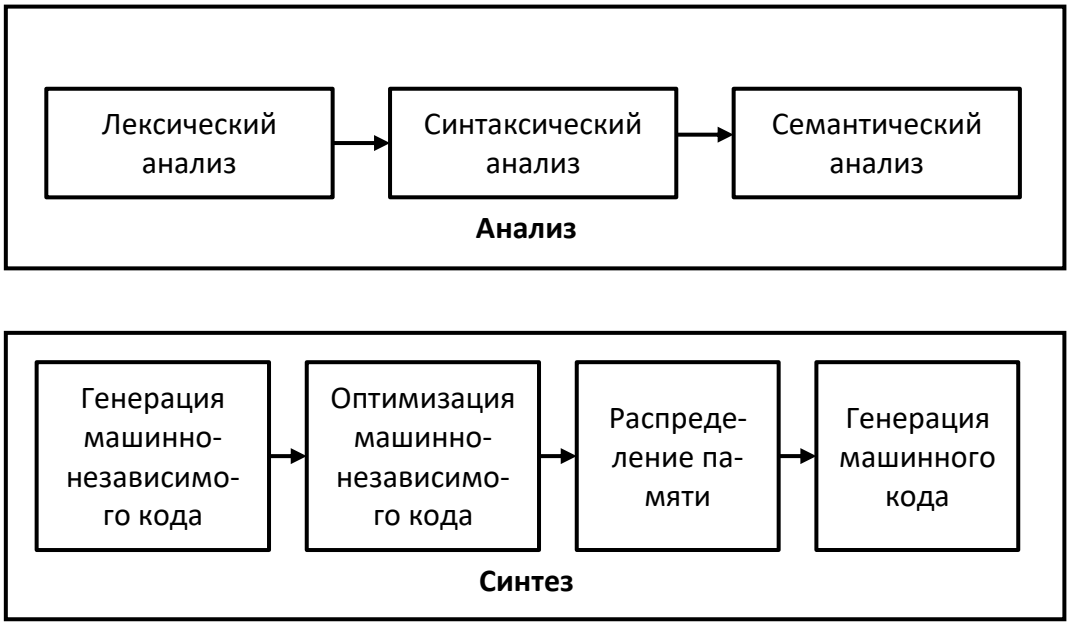

Рис. 2.2

Перечислим основные этапы трансляции:

1. Лексический анализ – оперирует с исходным текстом программы.

2. Синтаксический анализ – соединен с выходом лексического анализатора и входом семантического анализатора.

3. Семантический анализ – генерирует абстрактное синтаксическое дерево разбора.

4. Генерация машинно-независимого кода – переводит абстрактное дерево разбора в последовательность инструкция, не привязанную к конкретной архитектуре.

5. Оптимизация машинно-независимого кода – служит для исключения из кода бесполезных инструкций.

6. Распределение памяти – выделяет для каждой переменной в программе область памяти и закрепляет за ней определенный адрес.

7. Генерация машинного кода – выдает последовательность инструкций, пригодную для исполнения на ЭВМ.

Остановимся подробнее на фазе анализа.

Лексический анализ является наиболее простой фазой, в ходе которой вырабатываются так называемые *лексемы* языка. Например, ключевые слова

$$for\ do\ while$$

или пользовательские идентификаторы

$$name\ sprintf$$

или знаки операций

$$+ = * - /$$

Эти цепочки удобнее представлять в виде одного символа, принадлежащего некоторому типу, и имеющего некоторое *ассоциированное значение*.

Задача лексического анализа – перевести исходное текстовое представление из последовательности знаков в последовательность лексем. Эта последовательность символов затем передается на вход синтаксического анализатора.

Важно отметить, что лексический анализатор никак не воспринимает переданную ему на вход последовательность знаков. Например, он не примет никакого решения относительно корректности или некорректности следующей программы на языке *C*:

$$; \text{ number int return do} == + +$$

Полная независимость от контекста программы (его синтактической нагрузки) делает лексический анализ относительно простой фазой трансляции. Вместе с тем, как правило лексический анализ занимает продолжительное время, так именно на этом этапе осуществляется посимвольное чтение исходной программы.

Синтаксический анализ (*парсинг*, от англ parse – *разбирать*) производит построение общей структуры программы. Здесь имеет значение, какой символ следует за текущим, а какой ему предшествует. Результат работы синтаксического анализатора имеет структура дерева – синтаксического дерева разбора.

Например, алгебраическое выражение

$$\big((1 - (2 * 3)) + 4\big)$$

имеет следующую синтаксическую структура (рис. 2.3).

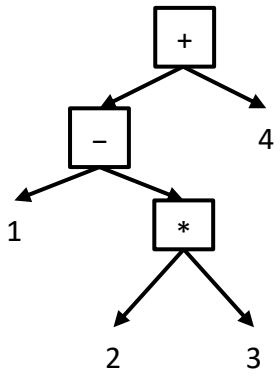

Рис. 2.3

Здесь отсутствуют скобки, поскольку сама структура дерева содержит их в неявном виде. Подобным же образом в виде *абстрактного синтаксического дерева* можно представить всю программу. Фаза синтаксического анализа является ключевой фазой анализа исходного текста программы.

Некоторые характеристики реализуемого языка не могут быть проверены в ходе синтаксического анализа. В частности, информация о типах переменных

не может быть получена простым сканирование символов исходного кода слева направо. Для этого необходимо иметь возможность перемещаться по тексту программы в границах всего исходного кода.

Семантический анализ необходим для устранения особенностей языка, не поддающихся описанию средствами формальной грамматики. В частности, проверка типов и область видимости переменных происходит на семантической фазе анализа.

Распределение памяти занимается выделением для каждой переменной или константы некоторого места для хранения своего значения. При этом память может быть следующего типа:

1. Статическая память, в которой хранятся переменные, время жизни которых равно времени жизни программы.

2. Динамическая память, хранящая переменные некоторой функции, по завершение которой их можно удалить.

3. Глобальная память используется в том случае, если время жизни переменной можно определить только на этапе выполнения программы (*runtime*).

Стоит отметить, что в отличие от анализа, синтез трудно поддается автоматизации.

Несколько слов следует сказать о средствах автоматизированной разработки анализаторов.

1. Генераторах лексических анализаторов.

2. Генераторах синтаксических анализаторов.

На вход генератора лексических анализаторов поступает информация о *лексической структуре* языка (каким образом из знаков составляются лексемы). Выход представляет собой программу (например, на языке *C*), которая представляет собой лексический анализатор для данного языка.

На вход генератора синтаксических анализаторов подается информация о *синтаксический структуре* языке (используемой им грамматике). Результатом работы является синтаксический анализатор в виде программы (например, на том же языке *C*).

Вход и выход обеих программ соединяют для получения окончательного результата – дерева синтаксического анализа. Процесс схематически изображен на рис. 2.4.

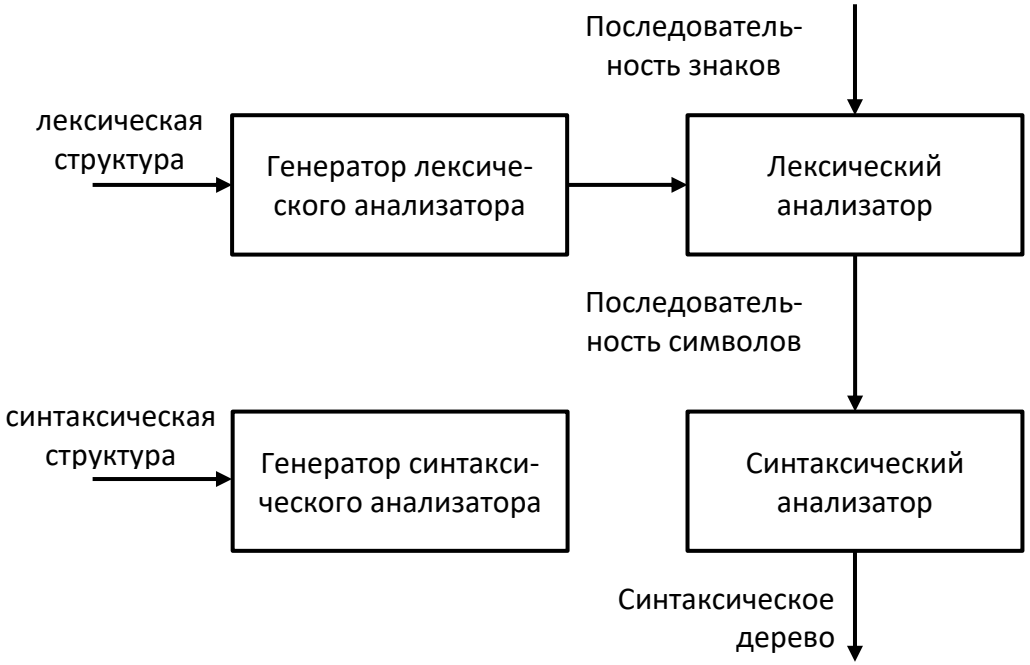

Рис. 2.4

2.1. Регулярные выражение

Первой фазой анализа исходного текста программы является лексический анализ. Эта фаза достаточно проста и для нее достаточно мощности, предлагаемой регулярными грамматиками. Для описания регулярных языков существует другое, очень удобное представление, а именно, *регулярные выражения*. Для того, чтобы математически ввести это новое понятие, определим следующие операции.

Определение 24. *Объединением* двух языков L_1 и L_2 является язык $L_1 \cup L_2$, включающий все строки языка L_1 и все строки языка L_2.

Определение 25. *Конкатенацией* языков L_1 и L_2 является новый язык L_1L_2, все строки которого получены соединением каждой строки языка L_1 с каждой строкой языка L_2.

Определение 26. *Итерацией* языка L является новый язык, полученный в результате возведения каждой строки языка L в степень $n = 0,1, \dots \infty$.

Для записи перечисленных операций существуют удобные обозначения. Так, например, для обозначения операции итерации служит символ «*» (*звездочка Клини*). Запись x^* читается так «во входной строке символ *x* может встретиться ноль и более раз».

Символ «+» в выражении x^+y^+ читается, как «*x* может встретиться один и более раз, за которым следует *y*, который также может встретиться один или

более раз». Строго говоря этот оператор необязателен, так как может быть заменен xx^*yy^*.

Определение 27. *Регулярное выражение*, определенное для алфавита *V*, представляет собой:

1. Пустую строку – ε.

2. Любой символ из алфавита *V*.

3. Если P и Q являются регулярными выражениями, то регулярными также являются выражения:

1. PQ

2. $P|Q$

3. P^*

т.е. выражения полученные с помощью *конкатенации*, *объединения* и *итерации*, соответственно.

Пример 21. Регулярное выражение $(aab|ab)^*$ допускает строки вида:

$$\varepsilon$$
$$aabaabab$$
$$ababab$$
$$aababaabab$$

и т. д.

Оператор итерации имеет самый высокий приоритет, а оператор объединения самый низкий.

Далеко не каждый язык программирования можно определить с помощью регулярных выражений. Более того, большинство существующих языков программирования не поддаются описанию с помощью регулярных выражений. Однако механизм регулярных выражений часто используется на фазе лексического анализа для определения символа языка через составляющие его знаки. Например, регулярное выражение

$$I(I|d)^* \qquad (2.1)$$

определяет идентификатор для многих языков программирования. В самом деле, если считать, что I – любая буква, а d – любая цифра, то данное регулярное можно расшифровать так: одна буква за которой следует произвольное количество букв или цифр.

Регулярные выражения широко используются для целей разбора текстов не только в задачах трансляции, но и в задачах обработки графики, аудио и видео файлов. Существуют расширенные описания регулярных выражений, которые включают операции, не перечисленные выше. Эти описания входят в состав стандартных библиотек практически всех языков программирования и

операционных систем. К наиболее популярным расширениям относятся *POSIX* и *PCRE*.

2.1.1. Отличия регулярных и контекстно-свободных грамматик

При разработке компиляторов наиболее часто используются языки 2-го и 3-го типов, т.е. контекстно-свободные (КС) и регулярные. При этом, как уже говорилось выше, КС-языки включают в себя все регулярные языки в том смысле, что грамматика 3-го типа одновременно является грамматикой 2-го типа (но не наоборот). КС-языки более сложны для анализа, чем регулярные языки. Следовательно, где только возможно следует использовать регулярные языки. Для этого необходимо уметь определять, является ли язык регулярным.

Введем понятие *рекурсии* в языке. Следующие продукции:

$$A \to Aa \tag{2.2}$$

$$B \to cB \tag{2.3}$$

$$C \to dCf \tag{2.4}$$

содержат рекурсию, так как нетерминал, стоящий в левой части, присутствует также и в правой. Различают *левую* (2.2), *правую* (2.2) и *среднюю* (2.4) рекурсии, в зависимости от того, где располагается терминал в правой части продукции.

Необходимо отметить, что грамматики практически всех языков содержат рекурсию, так как это единственный способ описать язык, допускающий произвольно большие предложения.

Также отметим, что существует еще и *непрямая* рекурсия, например,

$$A \to Bc$$
$$B \to Cd$$
$$C \to Ae$$

Первый шаг в определении того, является ли язык регулярным, состоит в том, содержит ли грамматика рекурсию. Если – нет, следовательно, язык содержит конечное множество предложений, и является регулярным.

Утверждение 4. Если язык не содержит среднюю рекурсию, то он является регулярным.

Мы примем это утверждение без доказательства. Таким образом, язык, описываемый следующими грамматическими правилами, является регулярным

$$S \to XY|x|y|\varepsilon$$
$$X \to xX|x$$
$$Y \to yY|y$$

С этим языком мы уже встречались $L = \{x^n y^m | n, m \geq 0\}$.

С другой стороны, язык, описываемый следующими продукциями, не является регулярным, поскольку содержит среднюю рекурсию

$$S \to xSy|xy$$

С этим языком мы также уже встречались $L = \{x^n y^n | n > 0\}$.

Поговорим о связи только что введенных понятий с принципами построения компиляторов языка программирования. Лексика большинства языков программирования, к которой относятся имена переменных, литералы, константы, многознаковые операнды (например, ++), практически всегда можно описать языком регулярных выражений.

С другой стороны, выражения (например, арифметические выражения), которые составляют синтаксис языка, описать с помощью регулярных выражений невозможно. Для примера рассмотрим встречающуюся во многих языках возможность вкладывать одно выражение в другое с помощью скобок. Грамматически это можно описать так:

$$S \to (S)$$
$$S \to SS$$
$$S \to \varepsilon$$

Здесь присутствует средняя рекурсия, что делает грамматику нерегулярной.

2.1.2. Ограничения КС-грамматик

Как уже говорилось основой для построения трансляторов большинства современных языков программирования являются грамматики 2-го типа или *контекстно-свободные* грамматики. Существуют языки, достаточно простые, но для которых не существует КС-грамматики.

Пример 22. $\{a^m | m - \text{положительная степень двойки}\}$.

Пример 23. $\{a^n b^n c^n | n \geq 0\}$.

Учитывая это, естественно возникает вопрос – насколько КС-грамматики подходят для генерации языков программирования?

Утверждение 5. В общем случае невозможно создать КС-грамматику, которая учитывала бы все возможные конструкции языка.

Это связано с тем, что многие выражения таких языков требуют явного знания *контекста применения*.

Пример 24. Ниже приведен листинг фрагмента программы на языке *C*

Листинг 1

```
1) int f(int, int);
2) int main() {
3)     return f(4,5,6);
4) }
```

Здесь функция *f* вызывается в тремя параметрами вместо двух. Это пример того, как информация, которая требуется для правильной интерпретации кода в строке 3, содержится в строке 1.

Таким образом, использование КС-грамматики требует введения дополнительных ограничений на вход. Трансляция КС-языков программирования производится в два этапа:

1. Проверка синтаксиса на соответствие контекстно-свободной грамматике.

2. Синтаксически корректные выражения проверяются на соответствие дополнительным ограничениям.

2.2. Разбор по регулярной грамматике

Как уже было сказано, в основе лексического анализа формальных языков лежат регулярные грамматики. Напомним, что к регулярным грамматикам относятся леволинейные и праволинейные грамматики. Одним из основных положений регулярных грамматик является то, что для любой леволинейной грамматки всегда существует эквивалентная праволинейная грамматика.

Правила вывода для леволинейной регулярной грамматики имеют вид

$$A \to aB|a \tag{2.5}$$

Здесь, напомним, $A, B \in N; a \in T$. Данная грамматика не содержит ε-правила, т.е. правил с пустой правой частью вида $A \to \varepsilon$. Это ограничение не существенно, поскольку мы рассматриваем разбор по регулярным грамматикам применительно к лексическому анализу языка программирования. Лексемы языка не могут быть пустыми.

Регулярные выражения являются основой лексического анализа языка программирования.

Пример 25. Идентификатор языка определяется регулярным выражением вида

$$letter(letter|digit)^* \tag{2.6}$$

Реализация лексического анализатора – простейшая программа на любом языке высокого уровня. В листинге 2 приведен фрагмент программы на языке C, реализующий проверку введенного идентификатора на соответствие шаблону (2.6).

Листинг 2

```
#include <stdio.h>
#include <ctype.h>
int main() {
    char in;
    in = getchar();
```

Окончание листинга 2

```
    if (isalpha(in)) in = getchar();
    else return 1;
    while (isalpha(in) || isdigit(in)) in = getchar();
    return 0;
}
```

Приведенная программа лишь проверяют вводимые данные на соответствие шаблону, но не извлекают из них никакой информации.

Приведенный пример показывает насколько просто реализовать разбор выражения по регулярной грамматике, однако в общем случае применяют аппарат *конечных автоматов*. Известно (см. выше), что регулярные выражения и конечные автоматы тесно связаны друг с другом.

Конечный автомат может находиться в одном их конечного множества состояний и может переходить из одного состояния в другое в результате считывания символа из входного потока.

2.2.1. Конечный автомат

Определение 28. *Детерминированный конечный автомат (ДКА)* – это пятерка элементов $M = (K, \Sigma, \delta, S, F)$, где

K – множество состояний;

Σ – входной алфавит;

δ – множество переходов между состояниями;

S – начальное состояние ($S \in K$);

F – одно или несколько конечных состояний ($F \in K$).

Стоит сделать несколько уточняющих замечаний. Во-первых, может быть только одно конечное состояние, поскольку все цепочки регулярного языка должны заканчиваться на символ конца $\perp$. Это замечание относится исключительно к лексическому анализу языков программирования и не связано с теорией автоматов, которая допускает у конечного автомата несколько конечных состояний.

Во-вторых, множество переходов $\delta(A, t) = B$ есть функция, которая ставит в соответствие некоторому состоянию A и текущему символу t на входе другое состояние B. Если для текущего состояния A_i и текущего прочитанного символа t_i значение $\delta(A_i, t_i)$ не определено, автомат останавливается с ошибкой.

Определение 29. Говорят, что ДКА *допускает* цепочку символов $a_1 a_2 \dots a_n$, если определены $\delta(H, a_1) = A_1, \delta(A_1, a_2) = A_2, \dots, \delta(A_{n-1}, a_n) = S$, где $a_i \in T, A_j \in K$, H – начальное состояние, S – конечное состояние.

Пример 26. На рис. 2.5 приведен конечный автомат для распознавания идентификатора (2.6).

Рис. 2.5

Здесь при считывании первой буквы происходит переход из *начального* состояния 1 в состояние 2 (*конечное*). Далее, любая буква или цифра переводят автомат снова в состояние 2.

Пример 27. Рассмотрим регулярное выражение для распознавания действительного числа

$$(+|-|\varepsilon)digit^*.digit\ digit^* \qquad (2.7)$$

Необходимо отметить, что данное выражение не вполне соответствует лексике языка *C*, которая допускает отсутствие дробной части (после десятичной точки).

Конечный автомат, соответствующий регулярному выражению (2.7) приведен на рис. 2.6.

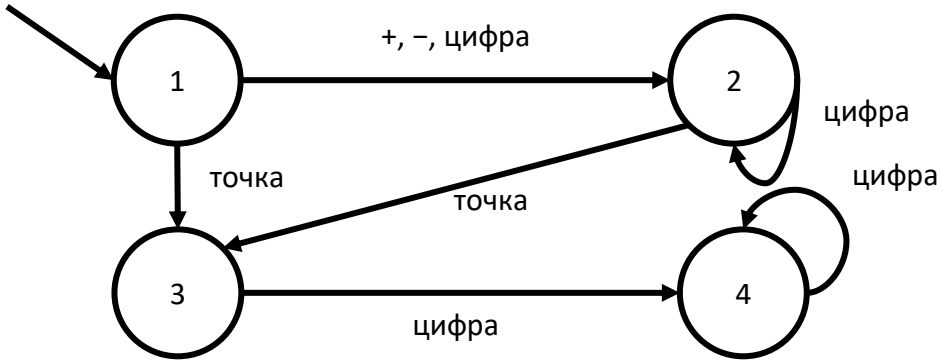

Рис. 2.6

В листинге 3 приведен фрагмент программы на языке С, который реализует автоматный разбор входной строки на соответствие регулярному выражению.

Листинг 3

```
#include <stdio.h>
#include <ctype.h>
int issign(char in) {return (in == '+' || in == '-');}
int main() {
```

```
    int state = 1;
    char in = getchar();
    while (isdigit(in) || issign(in) || in == '.') {
        switch (state) {
        case 1:
            if (isdigit(in) || issign(in)) state = 2;
            else if (in == '.') state = 3;
            break;
        case 2:
            if (isdigit(in)) state = 2;
            else if (in == '.') state = 3;
            else return 1;
            break;
        case 3:
            if (isdigit(in)) state = 4;
            else return 1; // Error!
            break;
        case 4:
            if (isdigit(in)) state = 4;
            else return 1;
            break;
        }
        in = getchar();
    }
    return (state == 4);
```

Обращает на себя внимание тот факт, что в блоке ***while*** мы фактически указываем алфавит всех допустимых символов.

2.2.2. Недетерминированный конечный автомат

Недетерминизм конечного автомата состоит в том, что из текущего символа, находясь в некотором состоянии, автомат может перейти в несколько возможных состояний.

Определение 30. *Недетерминированный конечный автомат (НКА)* – это пятерка $< K, T, D, H, S >$, в которой

K – конечное множество состояний;

T – конечное множество возможных символов на входе;

D – функция разрешенных переходов между состояниями;

H – начальное состояние;

S – множество заключительных состояний.

Графически работу НКА можно представить так (рис. 2.7).

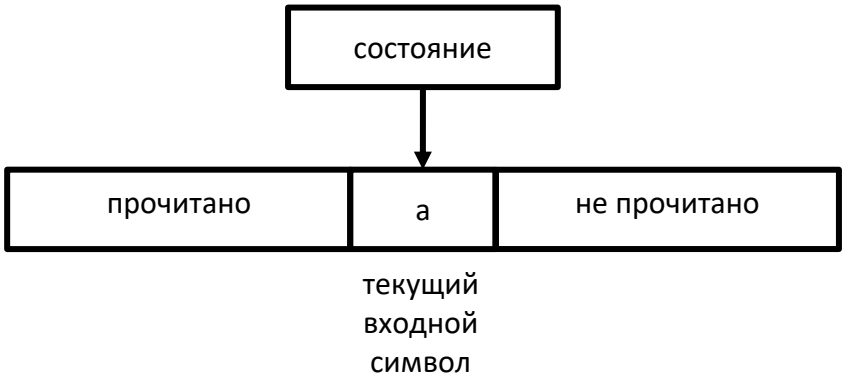

Рис. 2.7

2.2.3. Детерминированный конечный автомат

В отличие от НКА из каждого состояния детерминированный конечный автомат (ДКА) может перейти только в одно допустимое состояние.

ДКА является частным случаем НКА. Другими словами, класс языков, определяемых НКА, совпадает с классом языков, определяемых ДКА. Из этого следует важный вывод о том, что НКА может быть преобразован в ДКА и наоборот. Покажем это.

Пример 28. Построим конечный автомат для следующего регулярного выражения

$$(a|b)^*a(a|b)(a|b) \tag{2.8}$$

Недетерминированный конечный автомат содержит 4 состояния и определяется так

$$M = \{\{1,2,3,4\},\{a,b\},D,1,\{4\}\} \tag{2.9}$$

Здесь D – правила переходов, который удобно разместить в табл. 2.1.

Таблица 2.1

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

Данному конечному автомату соответствует следующая диаграмма состояний (рис. 2.8).

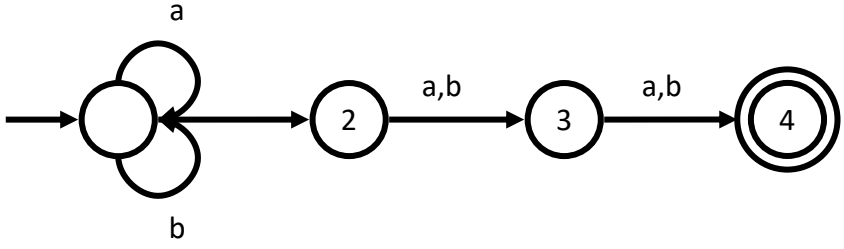

Рис. 2.8

Эту же задачу можно решить с помощью детерминированного конечного автомата. ДКА в данном случае будет иметь 8 состояний и будет определяться так

$$M = \{\{1,2,3,4,5,6,7,8\}, \{a, b\}, D, 1, \{3,5,6,8\}\} \quad (2.10)$$

Автомату будет соответствовать следующая диаграмма состояний (рис. 2.9).

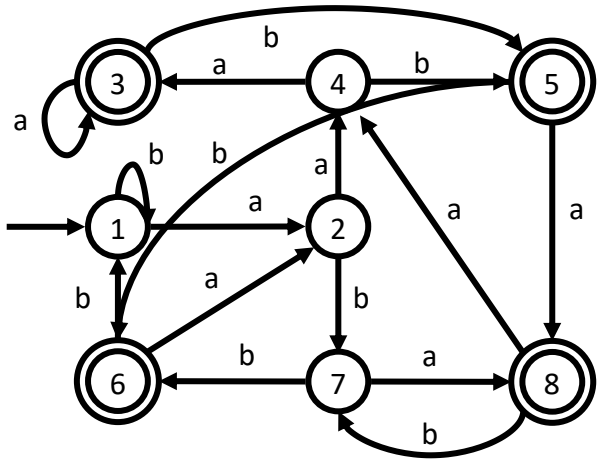

Рис. 2.9

Заключительными будут состояния 3, 5, 6 и 8 (они показаны кружками с двойной границей).

2.2.4. Построение НКА по регулярному выражению

Напомним, что в регулярном выражении допустимы следующие подвыражения:

1. Пустое множество $\emptyset$, которому соответствует диаграмма (рис. 2.10а).
2. Пустая строка ε (рис. 2.10б).
3. Любой символ из алфавита a (рис. 2.10в).
4. Объединение $s|t$, где s и t – регулярные выражения (рис. 2.10г).
5. Конкатенация двух подвыражений st (рис. 2.10д).
6. Итерация (звездочка Клини) s^* (рис. 2.10е).

Рассмотрим случай леволинейной грамматики $G =< T, N, P, S >$ с правилами вывода $A \to Bt|t$, где $A, B \in N; t \in T$.

2.2.5. Алгоритм разбора

Разбор по регулярной грамматике заключается в том, чтобы определить принадлежит ли строка $a_1 a_2 \ldots a_n$ $\perp$ языку, задаваемому данной грамматикой, и состоит из следующей последовательности шагов:

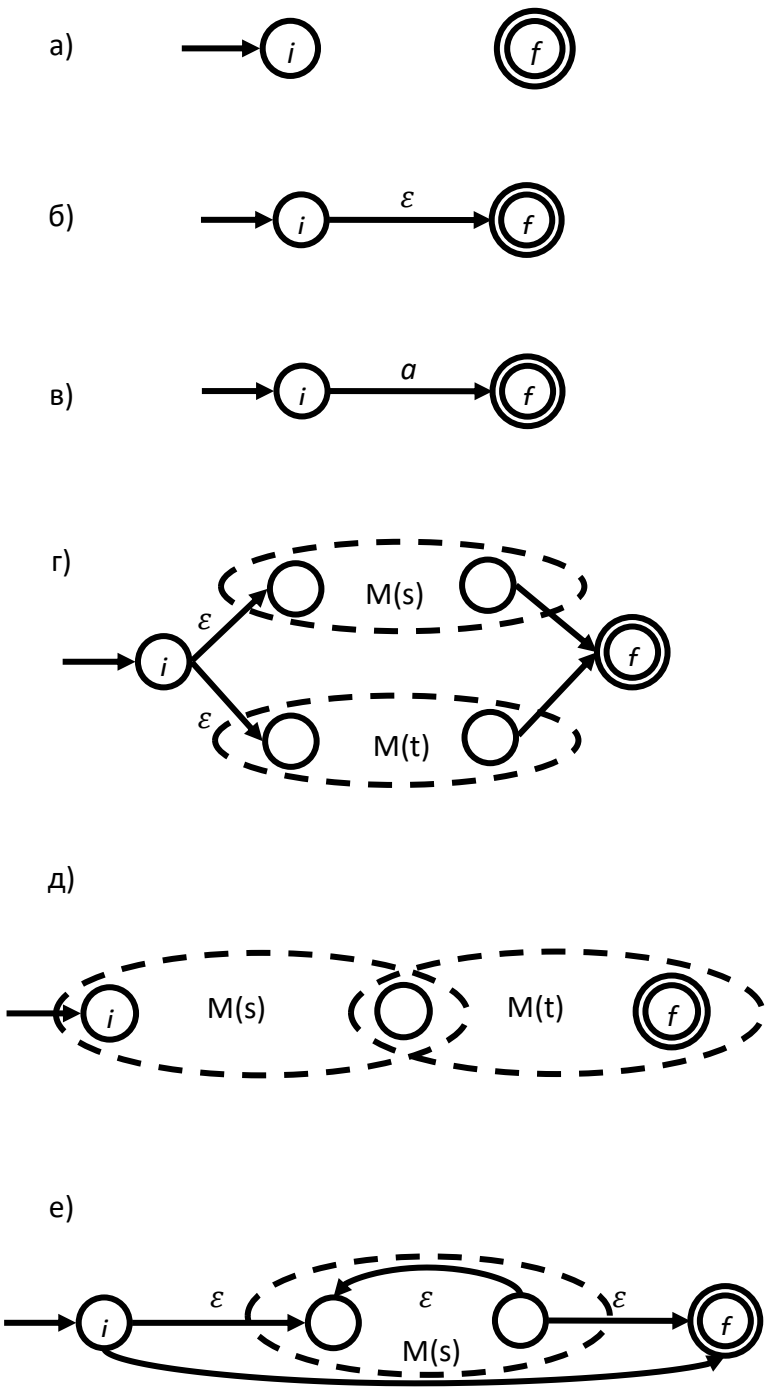

Рис. 2.10

1. Первый символ исходной строки заменить нетерминалом, для которого есть правило $A \to a_1$. Свертка терминала a_1 к нетерминалу A.

2. Повторять следующие действия до считывания символа конца строки: свернуть очередной входной символ с a_i с нетерминалом A и получить нетерминал B, для которого есть правило вывода $B \to Aa_i$ $(i = 2 \ldots n)$.

При этом возможны следующие ситуации:

1. Прочитана вся цепочка. На каждом шаге имелась нужная свертка. На последнем шаге свертка привела к символу S. Следовательно, цепочка принадлежит языку $a_1 a_2 \dots a_n \in L(G)$.

2. Прочитана вся цепочка. На каждом шаге находилась нужная свертка. Последняя свертка не привела к символу S. Следовательно, цепочка не принадлежит языку $a_1 a_2 \dots a_n \notin L(G)$.

3. На некотором шаге не нашлось нужной свертки. Следовательно, $a_1 a_2 \dots a_n \notin L(G)$.

4. На некотором шаге нашлось более одной нужной свертки. Эта *недетерминированная* ситуация должна рассматриваться отдельно.

Для того, чтобы автоматизировать процесс разбора, построим таблицу возможных переходов. Это позволит облегчить процесс нахождения нужного правила вывода с подходящей правой частью. Пусть столбцы таблицы содержат алфавит допустимых терминалов. Каждая строка таблицы разбора соответствует нетерминалу в левой части грамматических правил. При этом первая строка содержит начальный (стартовый) символ H. На пересечение каждой строки и столбца расположен нетерминал, к которому можно свернуть соответствующий терминал, помечающий столбец, и соответствующий нетерминал, помечающий строку.

Пример 29. Рассмотрим грамматику $G_l =< \{a, b, \perp\}, \{S, A, B, C\}, P, S >$, где

$$S \to C \perp \tag{2.11}$$
$$C \to Ab|Ba$$
$$A \to a|Ca$$
$$B \to b|Cb$$

Таблица разбора, которая определяется правилами (2.11) и не зависит от вида анализируемой цепочки приведена ниже. Здесь символ «-» означает, что для соответствующей пары «терминал-нетерминал» нет подходящей свертки.

Таблицу разбора удобно комбинировать с *диаграммой состояний* (ДС). Диаграмма представляет собой *ориентированный граф*, такой что:

1. Каждому нетерминалу грамматики соответствует вершина (состояние). Еще одна вершина, помеченная символом H, служит для обозначения начального состояния.

2. Для правил вывода $W \to t$ проводим дугу, помеченную t, ведущую из состояния H в состояние W.

3. Для правил вывода $W \to Vt$ соединяем дугой, помеченной t, состояния V и W.

Таблица 2.2

	$\boldsymbol{a}$	$\boldsymbol{b}$	$\perp$
$\boldsymbol{H}$	A	B	-
$\boldsymbol{C}$	A	B	S
$\boldsymbol{A}$	-	C	-
$\boldsymbol{B}$	C	-	-
$\boldsymbol{S}$	-	-	-

Полученная таким образом диаграмма состояний является конечным автоматом для заданной регулярной грамматики.

Пример 30. На рис. 2.11 представлена диаграмма состояний для грамматики (2.11).

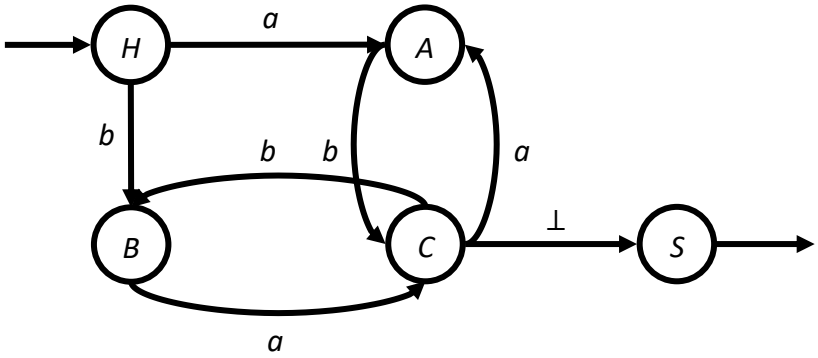

Рис. 2.11

Приведем алгоритм разбора входной строки $a_1 a_2 \ldots a_n$ по диаграмме состояний. Алгоритм включает следующие шаги:

1. Сделать текущее состояние начальным *H*.

2. Считать первый символ входной строки a_1.

3. Перейти в состояние *W*, ведущее из *H* по символу a_1.

4. Сделать состояние *W* текущим.

5. Повторять шаги 2-4 до считывания маркера конца строки или ошибки ввода.

Если все строки языка заканчиваются маркером ⊥, до конечное состояние единственно *S*.

Результатом разбора могут быть следующие ситуации:

1. Последний считанный символ привел в состояние *S*, следовательно, строка принадлежит языку *L*.

2. Последний считанный символ не привел в состояние *S*, следовательно, строка не принадлежит языку *L*.

3. Для некоторого символа a_i не нашлось перехода в новое состояние, следовательно, строка не принадлежит языку L.

4. Для некоторого символа a_i нашлось более одной дуги ведущей в разные состояния. В этом случае разбор *недетерминированный*.

2.2.6. Анализатор регулярной грамматики

Для регулярной грамматики (2.11) нами была составлена таблица разбора, нарисована диаграмма состояний, однако конечной целью разбора является написание программы анализатора на выбранном языке программирования. Для этого введем следующее соглашение: дополним множество возможных состояний конечного автомата дополнительным состоянием *ER*, переход в которое соответствует ошибке ввода.

В листинге 4 приведен фрагмент программы на языке C++, который реализует разбор введенной строки согласно грамматике (2.11). В данном случае конец ввода определяется стандартным для C++ способом – по «нулевому» символу.

Листинг 4

```
#include <iostream>
using namespace std;
char c; // текущий символ
void gc () { cin >> c; }
bool scan_G () {
    enum state { H, A, B, C, S, ER };
    state CS; // текущее состояние
    CS = H;
    do {
        gc ();
        switch (CS) {
        case H:
            if ( c == 'a' ) CS = A;
            else if ( c == 'b' ) CS = B;
            else CS = ER;
            break;
        case A:
            if ( c == 'b' ) CS = C;
            else CS = ER;
            break;
        case B:
            if ( c == 'a' ) CS = C;
            else CS = ER;
            break;
```

```
            case C:
                if ( c =='a' ) CS = A;
                else if ( c == 'b' ) CS = B;
                else if ( c == '\0' ) CS = S;
                else CS = ER;
                break;
            }
        }
        while ( CS != S && CS != ER);
        return CS == S;
}
```

Можно видеть, что здесь применен стандартный «автоматный» подход, которому в C++ соответствует оператор ***switch-case***.

Рассмотрим работу алгоритма подробнее. Для примера построим дерево разбора для строки $abba\perp$. Данной строке соответствует следующая последовательность *сверток*

$$abba\perp \rightarrow Abba\perp \rightarrow Cba\perp \rightarrow Ba\perp \rightarrow C\perp \rightarrow S \qquad (2.12)$$

Как видим, строка сворачивается в стартовый символ *S*, следовательно, она принадлежит описываемому языку. На рис. 2.12 показаны этапы построения дерева разбора.

Вспомним, что для леволинейной грамматики применим подход построения дерева снизу-вверх. При таком подходе правила вывода применяются в обратную сторону, т.е. каждому терминалу t и нетерминалу N, для который в грамматике есть правило $L \rightarrow Nt$, ставится в соответствие нетерминал L. Для рассматриваемой цепочки последовательность замен будет такой

$$abba\perp \leftarrow \boldsymbol{A}bba\perp \leftarrow \boldsymbol{C}ba\perp \leftarrow \boldsymbol{B}a\perp \leftarrow \boldsymbol{C}\perp \leftarrow \boldsymbol{S} \qquad (2.13)$$

Эта последовательность соответствует перевернутому правому выводу для данной цепочки в грамматике (2.11).

Выше мы рассмотрели леволинейную грамматику и осуществили по ней разбор цепочки. Леволинейной грамматике соответствует построение дерева разбора снизу-вверх.

Пример 31. Рассмотрим праволинейную грамматику $G_r =< \{a, b, \perp\}, \{H, A, B, C\}, P, H >$, у которой правила вывода таковы

$$H \rightarrow aA|bB \qquad (2.14)$$
$$A \rightarrow bC$$
$$C \rightarrow bB|aA|\perp$$
$$B \rightarrow aC$$

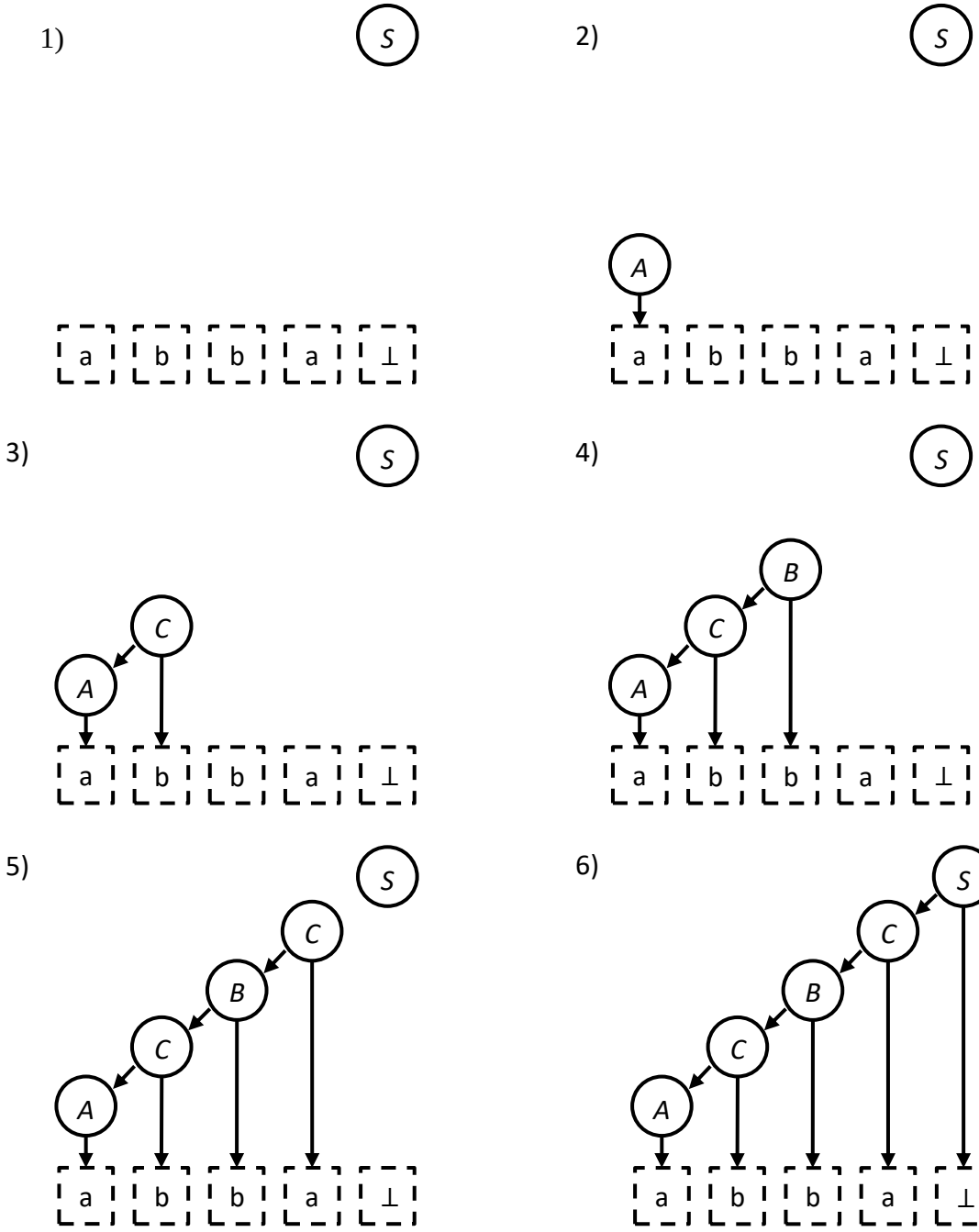

Рис. 2.12

Цепочка $abba\perp$, которую мы уже рассматривали, здесь выводится путем последовательно замены каждого нетерминала сентенциальной формы на левую часть подходящего правила вывода. Таким образом, левый вывод указанной цепочки имеет вид

$$\boldsymbol{H} \rightarrow a\boldsymbol{A} \rightarrow ab\boldsymbol{C} \rightarrow abb\boldsymbol{B} \rightarrow abba\boldsymbol{C} \rightarrow abba\perp \qquad (2.15)$$

Соответствующее данному выводу дерево разбора теперь строится сверху-вниз и показано на рис. 2.13.

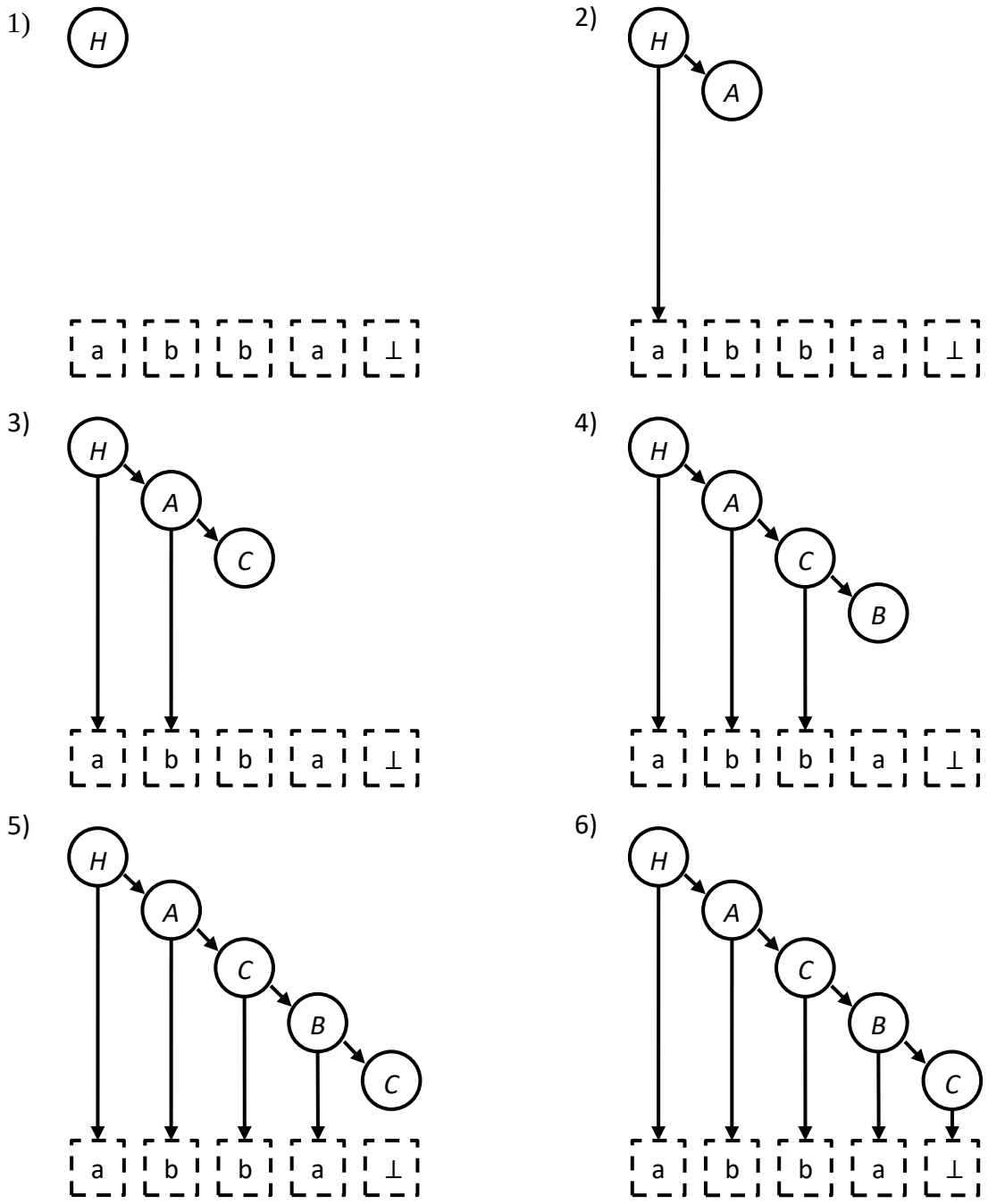

Рис. 2.13

Заметим, что грамматики G_l и G_r эквивалентны в том смысле, что описывают один и тот же язык.

Пример 32. Рассмотрим грамматику со следующими правилами вывода

$$S \to A \perp$$
$$A \to a|Bb$$
$$B \to b|Bb$$

Разбор строки по данной грамматике будет недетерминированным, поскольку разным нетерминалам слева (A и B) соответствуют одинаковые правые правила справа. Данной грамматике соответствует недетерминированный конечный автомат.

Разбор строки по НКА предполагает поиск всех возможных путей вплоть до нахождения успешного пути, где *успешный путь* – тот, который соответствует строке, принадлежащей языку. Однако, поиск всех успешных путей приводит к экспоненциальной сложности алгоритма. К счастью, как уже было сказано выше, НКА всегда можно преобразовать в ДКА.

Утверждение 6. Если язык L порожден некоторой регулярной грамматикой, то справедливо:

1) существует НКА, который допускает язык L;

2) существует ДКА, который допускает язык L.

2.3. Алгоритм преобразования НКА в ДКА

Вход: НКА $M = \{K, T, D, H, S\}$

Выход: ДКА $M_1 = \{K_1, T, D_1, H_1, S_1\}$

Как видим, четыре элемента в определении конечного автомата претерпели изменения. Неизменным остался только первичный алфавит T.

Принцип: объединять состояния НКА в одно состояние ДКА.

Алгоритм:

1. Все начальные состояния M объединить в одно начальное состояние M_1.

2. Для каждого терминала $t \in T$ добавить в множество состояний K_1 и множество переходов D_1 новое состояние, соответствующее переходу по символу t и начального состояния H_1 в множество возможных состояний M.

3. Выбрать заключительные состояния H_1 из множества заключительных состояний H.

Замечание. Следуя описанному алгоритму, может получиться несколько заключительных состояний. Это вызвано тем, что не всякий регулярный язык допускает детерминированный разбор с единственным заключительным состоянием. Поэтому во всех случаях в алфавите должен присутствовать маркер конца ввода, которым заканчивается каждая входная цепочка. Этим маркером может служить, например, признак конца файла (*eof*, от англ. *end-of-file* – конец файла), символ конца строки и т.д. Обязательное наличие маркера конца требует дополнительного состояния Q, в которое автомат должен может перейти из всех заключительных состояний S_1. Следовательно, все состояния из множества S_1 больше не считаются заключительными, а заключительным становится

единственное состояние Q. Грамматика, равно как и перестроенный автомат, дополняются новым переходом

$$S_1 \to Q \perp \tag{2.16}$$

и допускает детерминированный разбор.

Пример 33. Построить ДКА по НКА $M = \{\{H, A, B, S\}, \{0,1\}, D, \{H\}, \{S\}\}$, где

$$D(H, 1) = \{B\}$$
$$D(A, 1) = \{B, S\}$$
$$D(B, 0) = \{A\}$$

Данный НКА допускает язык

$$L(M) = \{1(01)^n | n \geq 1\} \tag{2.17}$$

и описывается регулярным выражением

$$101(01)^* \tag{2.18}$$

Прежде всего выпишем правила вывода для данного НКА

$$S \to A1$$
$$A \to B0$$
$$B \to A1|1$$

Получившаяся грамматика леволинейная. Очевидно, что разбор по данной грамматике будет недетерминированным, так как последовательности $A1$ соответствуют разные нетерминалы S и B. Диаграмма состояний для данного НКА представлена на рис. 2.14.

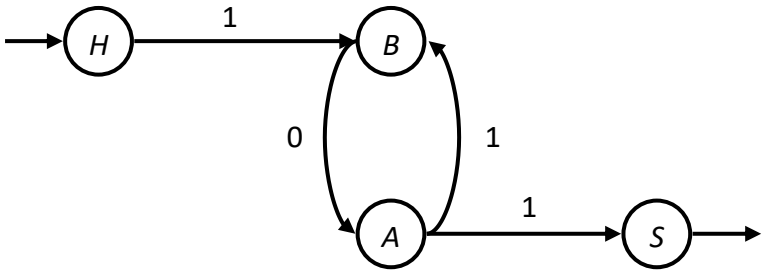

Рис. 2.14

Преобразуем недетерминированный конечный автомат в детерминированный по приведенному алгоритму.

Начальное состояние в НКА единственно, следовательно, оно становится начальным состоянием ДКА. Обозначим его H_1. Переход по единице (1) из состояния H в B детерминированный, следовательно, состояние B становится состоянием B_1. Переход по 0 из состояния B в состояние A также детерминированный. Из состояния A по символу 1 можно перейти, как в состояние B, так и в состояние S. Следовательно, введем новое состояние BS. Переход по 1 из со-

стояния A ведет в BS. Переход по 0 из состояния BS ведет в A. Новая грамматика, соответствующая детерминированному автомату, такова:

$$S_1 \to A_1 1$$
$$A_1 \to S_1 0 | B_1 0$$
$$B_1 \to 1$$

Здесь состояние BS переименовано в S_1. Получившейся грамматике соответствует конечный автомат $M_1 =< \{H_1, B_1, A_1, S_1\}, \{0,1\}, D_1, H_1, S_1 >$. Здесь функции переходов

$$D(H_1, 1) = B_1$$
$$D(B_1, 0) = A_1$$
$$D(A_1, 1) = S_1$$
$$D(S_1, 0) = A_1$$

Заключительным состоянием является S_1. Диаграмма состояний ДКА такова (рис. 2.15).

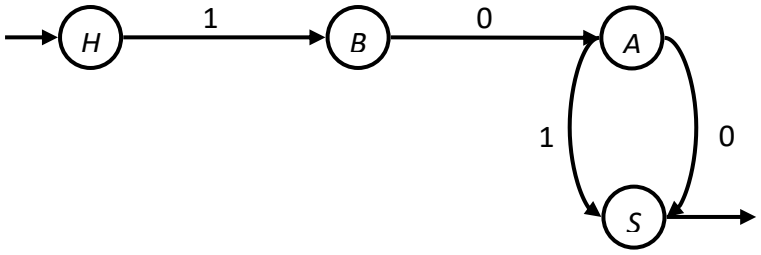

Рис. 2.15

Пример 34. Пусть дана недетерминированная грамматика

$$S \to Sb | Aa | a$$
$$A \to Aa | Sb | b$$

Требуется построить эквивалентную детерминированную грамматику и соответствующий ей ДКА.

Выпишем все разрешенные переходы

$$D(H, a) = \{S\}$$
$$D(H, b) = \{A\}$$
$$D(A, a) = \{A, S\}$$
$$D(S, b) = \{A, S\}$$

Воспользуемся таблицей для нахождения функции переходов ДКА. Построение таблицы начнем с состояния H. Начав с состояния H мы вводим два новых состояния S и A, в которые по символам a и b, соответственно, возможен переход.

Таблица 2.3

	$\boldsymbol{a}$	$\boldsymbol{b}$
$\boldsymbol{H}$	S	A
$\boldsymbol{S}$	$\emptyset$	AS
$\boldsymbol{A}$	AS	$\emptyset$
$\boldsymbol{AS}$	AS	AS

Введем новые обозначения состояний: $A \Rightarrow A, H \Rightarrow H, AS \Rightarrow Y, S \Rightarrow X$. Таким образом, получаем два заключительным состояния.

Добавим еще одно состояние S и введем маркер конца строки $\perp$. Это позволит свести два заключительных состояния в одно. Итоговая функция переходов ДКА такова:

$$D(H, a) = X$$
$$D(H, b) = A$$
$$D(X, b) = Y$$
$$D(X, \perp) = S$$
$$D(A, a) = Y$$
$$D(Y, a) = Y$$
$$D(Y, b) = Y$$
$$D(Y, \perp) = S$$

Данный конечный автомат допускает детерминированный разбор строк. Соответствующая ему детерминированная грамматика такова:

$$S \to X \perp | Y \perp$$
$$Y \to Ya|Yb|Aa|Xb$$
$$X \to a$$
$$A \to b$$

3. ЛЕКСИЧЕСКИЙ АНАЛИЗ

Лексема. Ассоциированное значение. Модельный язык. Форма Бэкуса-Наура. Токен. Ключевые слова. Разделители. Идентификаторы. Перечисления. Строковый литералы. Целочисленная константа. Вещественная константа.

В этом разделе мы применим полученные знания для построения лексического анализа некоторого модельного языка программирования, допускающего объявление переменных, стандартные арифметические действия, а также инструкции управления потоком выполнения программы (циклы, условия).

Определение 31. *Лексема* – совокупность символов исходного кода, принадлежащая некоторому типу.

Определение 32. *Токен* – кортеж, включающий в себя тип лексемы и информацию о ней – *ассоциированное значение*.

Определение 33. *Вход* лексического анализатора есть строка символов.

Определение 34. *Выход* – последовательность токенов.

Лексический анализ является первой фазой анализа исходного текста программы. Эта наиболее простая фаза, которая одновременно может является наиболее продолжительной, поскольку именно на этой фазы происходит посимвольное чтение исходного текста программы с устройства ввода.

К основным задачам лексического анализатора относятся:

1. Анализатор должен выделить в тексте программы последовательности символов и отнести их к лексеме определенного типа.

2. Анализатор должен проверить правильность записи каждой лексемы.

3. Анализатор должен зафиксировать факт нахождения лексемы в заданной позиции в исходном тексте программы.

4. Анализатор должен извлечь информацию о лексему из текста программы.

5. Анализатор должен удалить символы разделители и комментарии в тексте программы.

В этом разделе мы построим лексический анализатор для модельного языка программирования, но для начала введем несколько новых понятий.

3.1. Форма Бэкуса-Наура

Описание грамматики с помощью набора продукций наиболее правильный способ описания языка, однако по мере увеличения количества правил вывода он становится несколько громоздким.

Ученые Бэкус и Наур предложили улучшенную форму описания грамматический правил, известную как *форма Бэкуса-Наура (БНФ)*.

БНФ включает следующие специальные обозначения:

1. Левая и правая части каждого грамматического правила разделяются с помощью символа «::=».

2. Нетерминалы определяются произвольной символьной строкой, заключенной в угловые скобки, например $<$ выражение $>$.

3. Все другие символы обозначают терминалы.

4. Несколько альтернативных грамматических правил для одного и того же нетерминала разделяются вертикальной чертой «|».

Пример 35. С помощью БНФ определим стандартный *идентификатор* языка программирования, который должен начинаться с буквы, за которой может следовать произвольное количество букв или цифр.

$<$ идентификатор $>$::=$<$ буква $>$

$|<$ идентификатор $><$ буква $>$

$|<$ идентификатор $><$ цифра $>$

$<$ буква $>$::= $a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z$

$|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z$

$<$ цифра $>$::= $0|1|2|3|4|5|6|7|8|9$

3.1.1. Расширенная форма Бэкуса-Наура

Формы Бэкуса-Наура с точностью до условных обозначений практически полностью копируют основные правила формальных грамматик. Для повышения удобства БНФ расширяют следующими дополнительными конструкциями:

1. Синтаксические конструкции, которые могут отсутствовать, обрамляются слева и справа квадратными скобками «[» и «]».

2. Синтаксические конструкции, которые могут повторяться произвольное количество раз (*итерация*) заключаются в фигурные скобки «{» и «}».

3. Альтернативные синтаксическое конструкции заключаются в круглые скобки «(» и «)».

Пример 36. Расширенная форма Бэкуса-Наура (РБНФ) для записи идентификатора с учетом введенных ранее форм для $<$ буквы $>$ и $<$ цифры $>$ принимает вид

$<$ идентификатор $>$::=$<$ буква $>$ {$<$ буква $>$ | $<$ цифра $>$}

Пример 37. С помощью РБНФ запишем форму записи произвольного действительного числа, которое может включать знак, целую часть, десятичную точку, дробную часть, порядок и знак порядка.

$<$ действительное $>::=<$ число $><$ порядок $>$

$$|[<\text{число}>].<\text{число}>[<\text{порядок}>]$$

$<$ число $>::=<$ цифра $>\{<$ цифра $>\}$

$<$ порядок $>::=(E|e)[+|-]<$ число $>$

3.1.2. Диаграммы Вирта

Альтернативой записи грамматических правил в форме БНФ является графическое представление в виде диаграмм Вирта. Диаграммы Вирта используют следующие графические обозначения:

1. Каждый нетерминал грамматики заключается в прямоугольник.

2. Каждый односимвольный терминал первичного алфавита обозначается кружочком с начертанием символа в центре.

3. Каждый многосимвольный терминал (например, ключевое слово) заключается в прямоугольник с закругленными краями.

4. Перечисленные графические примитивы имеют вход и выход.

5. Грамматическим правилам соответствуют соединениям примитивов посредством дуг. Стрелки на дугах обычно не ставятся.

6. Альтернативные синтаксическое конструкции выделяются посредством ветвления дуг, а итерации – посредством слияния.

7. Должная быть одна входная дуга, входящая в стартовый нетерминал грамматики.

С учетом сказанного идентификатор языка программирования (пример 36) представляется следующей диаграммой Вирта (рис. 3.1).

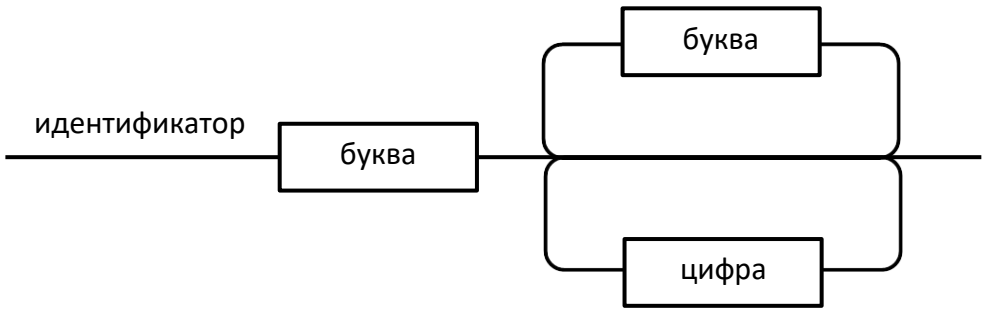

Рис. 3.1

Здесь графические примитивы буква и цифра не показаны ввиду их громоздкости.

3.2. Лексический анализатор для модельного языка

Лексический анализатор – подпрограмма, которая принимает на вход исходный текст программы и выдает последовательность *лексем* – минимальных элементов программы, несущих смысловую нагрузку.

В модельном языке программирования выделяют следующие типы лексем:

- ключевые слова;
- разделители;
- числа;
- идентификаторы.

Кстати, данные типы характерны практически для всех существующих языков программирования.

При разработке лексического анализатора, ключевые слова и разделители известны заранее, идентификаторы и числовые константы – вычисляются в момент разбора исходного текста.

Для каждого типа лексем предусмотрена отдельная таблица. Таким образом, внутреннее представление лексемы – пара чисел (*n*, *k*), где *n* – номер таблицы лексем, *k* – номер лексемы в таблице.

Кроме того, в исходном коде программы кроме ключевых слов, идентификаторов и числовых констант может находиться произвольное число пробельных символов («пробел», «табуляция», «перенос строки», «возврат каретки») и комментариев, заключенных в фигурные скобки.

3.2.1. Грамматика модельного языка

Ниже приводится полное описание грамматики рассматриваемого модельного языка в нотации РБНФ.

$$< \text{программа} > ::= \boldsymbol{program\ var}\ < \text{описание} > \{, < \text{описание} >\} \quad (3.1)$$
$$\boldsymbol{begin} < \text{оператор} > \{; < \text{оператор} >\} \boldsymbol{end}$$

$$< \text{описание} > ::= < \text{идентификатор} > \{, < \text{идентификатор} >\} \quad (3.2)$$
$$: < \text{тип} >$$

$$< \text{тип} > ::= \boldsymbol{int} \mid \boldsymbol{bool} \quad (3.3)$$

$$< \text{оператор} > ::= < \text{составной} > \mid < \text{присваивания} > \quad (3.4)$$
$$\mid < \text{условный} > \mid < \text{цикла} > \mid < \text{ввода} > \mid < \text{вывода} >$$

$$< \text{составной} > ::= \boldsymbol{begin}\ < \text{оператор} > \{; < \text{оператор} >\} \boldsymbol{end} \quad (3.5)$$

$$< \text{присваивания} > ::= < \text{идентификатор} > := < \text{выражение} > \quad (3.6)$$

$$< \text{условный} > ::= \boldsymbol{if}\ < \text{выражение} > \boldsymbol{then} < \text{оператор} > \quad (3.7)$$
$$[\boldsymbol{else} < \text{оператор} >]$$

$$< \text{цикла} > ::= \boldsymbol{while} < \text{выражение} > \boldsymbol{do} < \text{оператор} > \quad (3.8)$$

< ввода >::= ***read*** (< идентификатор >) (3.9)

< вывода >::= ***write*** (< выражение >) (3.10)

< выражение >::= < логическое > (3.11)

| < выражение > ***or*** < логическое >

< логическое >::=< равенство > (3.12)

| < логическое > ***and*** < равенство >

< равенство >::=< сравнение > (3.13)

| < равенство > = < сравнение >

| < равенство > ! = < сравнение >

< сравнение >::=< сложение > (3.14)

| < сравнение > < < сложение >

| < сравнение > > < сложение >

| < сравнение > <= < сложение >

| < сравнение > >= < сложение >

< сложение >::=< умножение > (3.15)

| < сложение > +< умножение >

| < сложение > −< умножение >

< умножение >::=< унарное > (3.16)

| < умножение >∗< унарное >

| < умножение >/< унарное >

< унарное >::=< первичное > (3.17)

|***not*** < унарное >

< первичное >::=< идентификатор > (3.18)

| < константа >

| (< выражение >)

< константа >::=< число > (3.19)

|(***true***|***false***)

Правила для < идентификатор > и < число > приведены в примерах 36 и 37, соответственно.

Данная грамматика является *контекстно-свободной*. Однако, часть правил являются *регулярными*, следовательно, по ним возможен автоматный разбор. К таковым относятся:

- выделение идентификаторов;
- выделение целочисленных констант;
- выделение ключевых слов (обозначены в правилах жирным шрифтом);
- выделение одно и двухсимвольных операторов;
- выделение разделителей.

Лексический анализ текста проводится по регулярной грамматике. Известно, что регулярная грамматика эквивалентна конченому автомату, следовательно, для написания лексического анализатора необходимо построить диаграмму состояний, соответствующего конечного автомата (рис. 3.2).

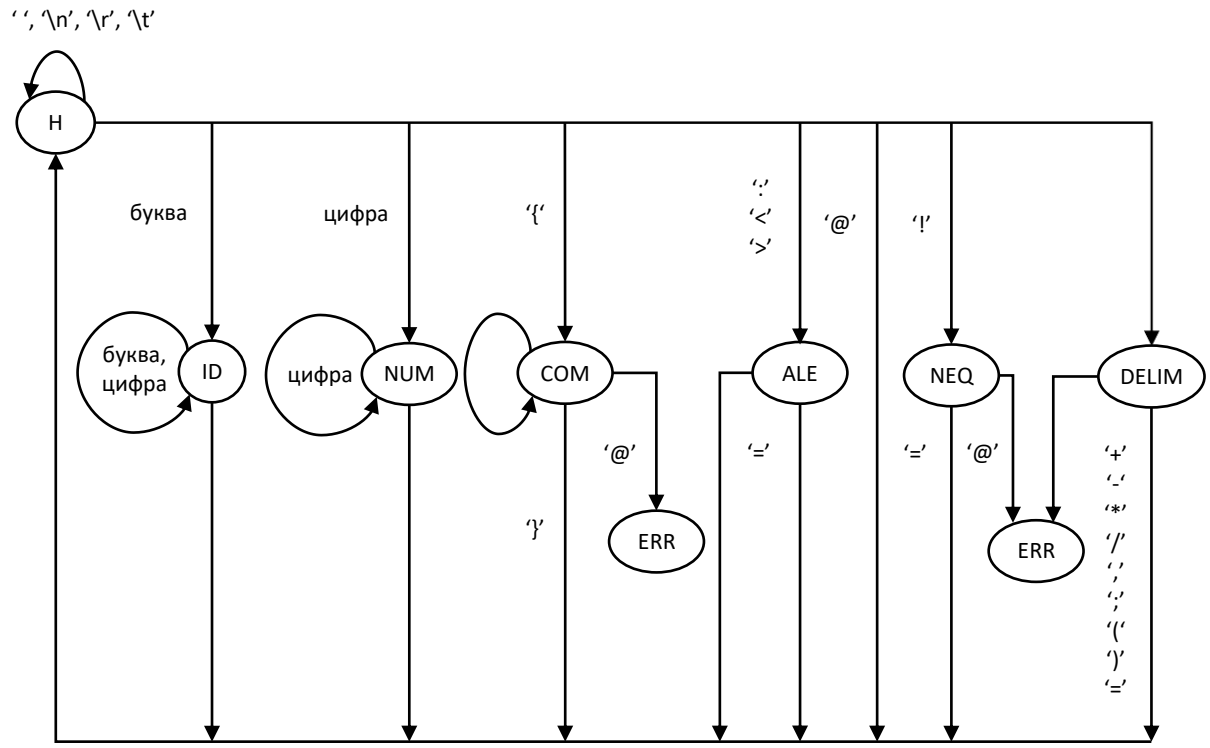

Рис. 3.2

Следует отметить, что приведенная на рис. 3.2 диаграмма состояния является не полной. В самом деле, автомат, помимо распознавания, т.е. определения того, принадлежит ли цепочка символов описываемому языку, должен в результате каждого перехода в новое состояние совершать некоторые действия. К подобным действиям относятся, например, сохранение очередного символа в некотором буфере, принятие решения о том, какому типу лексемы соответствует последовательность символов, сохраненная в буфере, перевод числа из строкового представления в целочисленное и т.д.

Пример 38. Рассмотрим конечный автомат, распознающий целые беззнаковые числа. Леволинейная грамматика, соответствующая поставленной задаче такова:

$$S \to N \perp$$
$$N \to N0|N1| \dots |N9|1|2| \dots |9$$

В данном случае автомат включает 3 состояния: H – начальное состояние, N – состояние, в котором автомат будет находиться до тех пор, пока на входе присутствует цифра и S – конечное состояние, в которое автомат перейдет, если

за последней цифрой следует маркер конца ввода. Соответствующая диаграмма состояний показана на рис. 3.3.

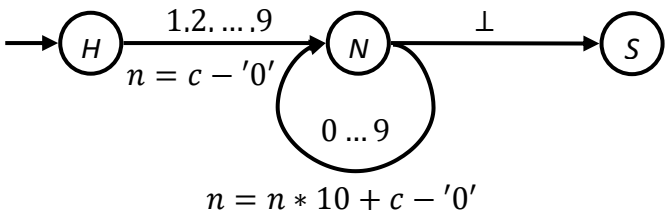

Рис. 3.3

Здесь действиями, которые должны выполняться, являются:

1. При переходе из состояния *H* в состояние *N* – сохранение в переменной *n* типа *int* значения целого значения первой цифры вводимого числа.

2. При переходе из состояния *N* в состояние *N* – добавление следующего разряда к числу, хранящемуся в переменной *n*.

3. При переходе из состояния *N* в состояние *S* – сохранение значения переменной *n*, как ассоциированного значения новой лексемы типа < число >.

Подробнее диаграмма состояний (рис. 3.2) рассматривается в листингах ниже, которые содержат фрагмент подпрограммы лексического анализатора модельного языка.

3.2.2. Типы лексем

Разработку лексического анализатора будем проводить на языке C++. Для определения всех типов лексем введем перечислимый тип данных (листинг 5).

Листинг 5

```
enum lex_type
{
    LEX_NULL,        // 0
    // КЛЮЧЕВЫЕ СЛОВА
    LEX_AND, LEX_BEGIN,..., LEX_WRITE, // 1,2,...,18
    // МАРКЕР КОНЦА
    LEX_FIN,         // 19
    // ОПЕРАТОРЫ И РАЗДЕЛИТЕЛИ
    LEX_SEMICOLON,   // 20
    LEX_COMMA,       // 21
    LEX_COLON,       // 22
    LEX_ASSIGN,      // 23
    LEX_LPAREN,      // 24
    LEX_RPAREN,      // 25
    LEX_EQ,          // 26
    LEX_LSS,         // 27
    LEX_GTR,         // 28
    LEX_PLUS,        // 29
```

Окончание листинга 5

```
    LEX_MINUS,       // 30
    LEX_TIMES,       // 31
    LEX_SLASH,       // 32
    LEX_LEQ,         // 33
    LEX_NEQ,         // 34
    LEX_GEQ,         // 35
    // ЧИСЛОВАЯ КОНСТАНТА
    LEX_NUM,         // 36
    // ИДЕНТИФИКАТОР
    LEX_ID           // 37
};
```

Итак, мы получили 37 типов лексем. Однако здесь перечислены только сами типы. Лексемы хранятся в специальных таблицах. В нашем случае достаточно трех таких таблиц:

1. **TW** – таблица ключевых слов.
2. **TD** – таблица операторов и разделителей.
3. **TID** - таблица идентификаторов.

Первые две таблицы известны в момент компиляции, поскольку данные, которые в них хранятся известны априори. Таблица **TID** заполняется информацией об встречающихся идентификаторах по мере прочтения программы.

Конкретизируем понятие лексема классом C++ (листинг 6).

Листинг 6

```
class Lex
{
    lex_type type;
    int value;
public:
    Lex(lex_type t = LEX_NULL, int v = 0) {
        type = t;
        value = v;
    }
    lex_type getType() { return type; }
    int getValue() { return value; }
    friend ostream& operator << (ostream & s, Lex l)
    {
        s << "(" << l.type << ',' << l.value << ");";
        return s;
    }
};
```

Здесь оператор «<<» перегружен для удобства вывода лексемы на печать.

В листинге 7 приведен класс «Идентификатор». Переменные данного типа хранятся в таблице **TID**.

Листинг 7

```
class Id
{
    char * name;
    lex_type type;
    int value;
public:
    char * getName() { return name; }

    void setName(const char * name) {
        this->name = new char[strlen(name) - 1];
        strcpy(this->name, name);
    }
    lex_type getType() { return type; }

    void setType(lex_type type) {
        this->type = type;
    }
    int getValue() { return value; }

    void setValue(int value) { this->value = value; }
};
```

Данный класс достаточно тривиален и не нуждается в дополнительных пояснениях.

Таблица **TID** также реализована, как класс (листинг 8).

Листинг 8

```
class TID
{
    Id * p;
    int size;
    int top;

public:
    TID(int max_size) {
        size = max_size;
        p = new Id[size];
        top = 1;
    }
    ~TID() {
        delete [] p;
    }
    Id& operator [] (int k) {
```

Окончание листинга 8

```
        return p[k];
    }
    int put(const char * name) {
        for(int i = 1; i < top; ++i) {
            if (!strcmp(name, p[i].getName())) {
                return i;
            }
        }
        p[top].setName(name);
        top++;
        return top - 1;
    }
};
```

Таблица идентификаторов хранится в куче рабочего процесса. Единственная функция, заслуживающая внимания – функция *put*, которая сравнивает переданную в качестве параметра строку с каждой строкой таблицы и, либо сохраняет ее, как новый идентификатор, либо, если строка уже присутствует в таблице, возвращает номер соответствующего ей идентификатора.

Сам разбор исходного текста на лексемы осуществляется в классе *Lexer* (Лексический анализатор). В классе происходит посимвольное чтение исходного текста программы из файла. Конечный автомат, согласно диаграмме состояний (рис. 3.2) включает 7 состояний (листинг 9)

Листинг 9

```
class Lexer
{
    enum state {H, ID, NUM, COM, ALE, NEQ, DELIM};
    ...
};
```

и реализован в функции *gettok* (листинг 10).

Листинг 10

```
Lex Lexer::gettok()
{
    int d, j;
    CS = H;
    do {
        switch(CS) {
        case H:
            if (c==' '||c=='\n'||c=='\r'||c=='\t') {
                gc();
            }
            else if (isalpha(c)) {
```

```
                clear();
                add();
                gc();
                CS = ID;
            }
            else if (isdigit(c)) {
                d = c - '0';
                gc();
                CS = NUM;
            }
            else if (c == '{') {
                gc();
                CS = COM;
            }
            else if (c == ':' || c == '<' || c == '>') {
                clear();
                add();
                gc();
                CS = ALE;
            }
            else if (c == '@') {
                return Lex(LEX_FIN);
            }
            else if (c == '!') {
                clear();
                add();
                gc();
                CS = NEQ;
            }
            else
                CS = DELIM;
            }
            break;
        case ID:
            if (isalpha(c) || isdigit(c)) {
                add();
                gc();
            }
            else {
                j = look(buf, TW);
                if (j) {
                    return Lex(words[j], j);
                }
                else {
```

```
                j = TID.put(buf);
                return Lex(LEX_ID, j);
            }
        }
        break;
    case NUM:
        if (isdigit(c)) {
            d = d * 10 + (c - '0');
            gc();
        }
        else {
            return Lex (LEX_NUM, d);
        }
        break;
    case COM:
        if (c == '}') {
            gc();
            CS = H;
        }
        else if (c == '@' || c == '{') {
            throw c;
        }
        else {
            gc();
        }
        break;
    case ALE:
        if (c == '=') {
            add();
            gc();
            j = look(buf, TD);
            return Lex(dlms[j], j);
        }
        else {
            j = look(buf, TD);
            return Lex(dlms[j], j);
        }
        break;
    case NEQ:
        if (c == '=') {
            add();
            gc();
            j = look(buf, TD);
            return Lex(LEX_NEQ, j);
```

Окончание листинга 10

```
            }
            else {
                throw '!';
            }
            break;
        case DELIM:
            clear();
            add();
            j = look(buf, TD);
            if (j) {
                gc();
                return Lex(dlms[j], j);
            }
            else {
                throw c;
            }
            break;
        }
    }
    while(true);
}
```

Здесь применен следующий подход. Поскольку основная работа по разбору текста будет сосредоточена в синтаксическом анализаторе, то лексический анализатор будет выдавать очередную лексему и дожидаться следующего вызова. Обработка ошибок также будет сосредоточена в синтаксическом анализаторе, поэтому в листинге мы просто выбрасываем исключение с сообщением того символа, который привел к ошибке.

В листинге 10 встречаются функции, исходные тексты которых мы не приводим в силу их простоты, а именно:

1. *clear* – очищает внутренний буфер и готовит его для принятия следующей лексемы.

2. *add* – добавляет очередной прочитанный символ в буфер.

3. *gc* – считывает очередной символ из потока ввода.

4. *look* – построчно сравнивает содержимое буфера с соответствующей таблицей лексем и, в случае нахождения совпадения, выдает номер лексемы.

Также класс *Lexer* использует вспомогательные таблицы, которые хранят символьные представления всех ключевых слов, операторов и символов разделителей. Эти таблицы синхронизированы с перечислимым типом *lex_type* из листинга 5, поскольку тип лексемы – это значение соответствующей перечислимой константы.

4. СИНТАКСИЧЕСКИЙ АНАЛИЗ

Нисходящий синтаксический анализ. Восходящий синтаксический анализ. LL(1)-грамматики. Метод рекурсивного спуска. Таблица прогнозов. Прогнозируемый выбор альтернатив. Множество first(α). Множество follow(A). Канонические грамматики.

4.1. Основная задача синтаксического анализа

Основная задача синтаксического анализа состоит в нахождении порождения для заданного выражения (или принятия решения о том, что порождения не существует). В основе анализа лежит формальная грамматика используемого языка.

Таким образом, задача синтаксического анализа разбивается на две подзадачи:

1. Необходимо определить соответствует ли последовательность лексем на входе анализатора синтаксису языка, задаваемому формальной грамматикой.

2. Необходимо построить для данной последовательности дерево разбора.

В основе синтаксического анализа большинства языков программирования лежат КС-грамматики. Их выразительной мощности вполне хватает для того, чтобы охватить все ключевые составляющие вычислительного процесса, такие как переменные, функции, встроенные типы данных, пользовательские типы данных, условные операторы, операторы выбора, операторы циклов, арифметические и логические операторы и т.д. Напомним, для любой КС-грамматики справедливы правила вида

$$A \to \alpha, A \in N, \alpha \in (T \cup N)^*$$

При *нисходящем* синтаксическом анализе в основном ищутся *левые* порождения.

При *восходящем* синтаксическом анализе в основном ищутся *правые* порождения.

Нисходящий анализ исходит из символа предложения S (цели грамматики) и генерирует предложение.

При восходящем анализе разбор начинается с предложения, которое необходимо свернуть в S.

Пример 39. Рассмотрим язык, с которым мы уже встречались выше $L = \{x^m y^n | m, n > 0\}$. Данному языку соответствуют следующие правила вывода:

$$S \to XY \quad (4.1)$$

$$X \to xX \quad (4.2)$$

$$X \to x \quad (4.3)$$
$$Y \to yY \quad (4.4)$$
$$Y \to y \quad (4.5)$$

Следующую строку $xxxyy$ можно получить с помощью левого порождения:

$$S \Rightarrow XY \Rightarrow xXY \Rightarrow xxXY \Rightarrow xxxY \Rightarrow xxxyY \Rightarrow xxxyy \quad (4.6)$$

Разбор производится слева направо. Здесь мы на каждом шаге заменяем крайний левый нетерминал. Первый шаг очевиден, поскольку для нетерминала *S* есть только одна продукция (4.1).

Следующий шаг не так очевиден, поскольку для нетерминала *X* существует более одной продукции (4.2) и (4.3). Помощь в выборе нужной продукции нам окажет следующее наблюдение: **при синтаксическом разборе результат (исходная строка) всегда известен**. В данном случае это $xxxyy$.

На следующем шаге мы видим в исходной строке более одного символа *x*, следовательно, необходимо использовать продукцию (4.2).

Далее эта продукция используется еще один раз. Наконец, в исходной строке остается только один символ *x*, за которым следует символ *y*.

Для первого символа y используется продукция (4.4). После этого в строке остается последний символ *y*, поэтому для него используется продукция (4.5).

Из рассмотренного примера можно сделать вывод: **для синтаксического анализа требуется просмотр не только текущего символа, но и нескольких последующих**. Таким символов в общем случае может быть произвольно много.

Еще раз обратимся к примеру 39 и зафиксируем последовательность применения правил в таблице.

Таблица 4.1

Входная строка	**Правило вывода**	**Сентенциальная форма**
$xxxyy$	$S \to XY$	XY
$xxxyy$	$X \to xX$	xXY
$\sout{x}xxyy$	$X \to xX$	$xxXY$
$\sout{xx}xyy$	$X \to x$	$xxxY$
$\sout{xxx}yy$	$Y \to yY$	$xxxyY$
$\sout{xxxy}y$	$Y \to y$	$xxxyy$
$\sout{xxxyy}$		

Здесь в первом столбце уже просмотренные символы зачеркнуты. Первый незачеркнутый символ и является символом предпросмотра.

Таким образом, вывод о том какую продукцию применять, делается на основе текущего символа предпросмотра.

4.2. LL(1)-грамматики

Большой класс формальных языков может быть проанализирован с помощью не более одного символа предпросмотра на каждом этапе порождения. Синтаксической анализатор обычно следует за лексическим анализатором, так что здесь под символом предпросмотра подразумевается лексема.

В первом приближении будем рассматривать только *однозначные* грамматики, такие что каждой строке языка соответствует единственное левое порождение.

Метод нисходящего разбора заключается в следующем. При левом порождении мы заменяем крайний левый нетерминал. Если нетерминал стоит в левых частях *нескольких* продукций, требуется найти множества символов предпросмотра так, чтобы каждое множество соответствовало одной продукции. Поскольку грамматика по определению однозначна, эти множества не пересекаются.

Далее, мы просматриваем, к какому множеству принадлежит данных символ предпросмотра, и выбираем соответствующую продукцию.

Если символ предпросмотра не принадлежит ни одному множеству, значит имеет место *синтаксическая ошибка*.

Определение 35. *Стартовый символ* – любой символ, с которого начинается данная продукция.

Определение 36. *Символ-последователь* – любой символ, который может следовать за данным нетерминалом в любой сентенциальной форме.

Пример 40. Грамматика содержит следующие правила вывода:

$$T \to aG|bG$$

Здесь множество стартовых символов для каждой продукции – это просто терминалы, с который начинается правая часть $\{a, b\}$. Если правая часть начинается с нетерминала, множество стартовых символов необходимо вычислять.

Пример 41. Расширим грамматику из предыдущего примера следующими правилами вывода:

$$T \to aG|bG$$
$$R \to BG|CH$$
$$B \to cD|TV$$

Для продукции $R \to BG$ множество стартовых символов есть $\{a, b, c\}$. В самом деле, правая часть начинается в нетерминала B, следовательно, множество стартовых символов для R совпадает с множеством для B. Множество стартовых символов для нетерминала B содержит собственный терминал c, а

также терминалы a и b, унаследованные от множества стартовых символов нетерминала T.

Из сказанного можно сделать вывод о том, что в общем случае множество стартовых символов для каждого правила вывода может быть вычислительно сложным процессом.

Определение 37. *Множество первых порождаемых символов* для каждой продукции является множеством всех терминалов, которые указывают на использование данной продукции в порождении.

Пример 42. Дана грамматика с правилами вывода

$$S \to Ty$$
$$T \to AB|sT$$
$$A \to aA|\varepsilon$$
$$B \to bB|\varepsilon$$

Множества первых порождаемых символов для данного примера приведены в табл. 4.2.

Таблица 4.2

Продукция	**Множество первых порождаемых символов**
$T \to AB$	$\{a, b, y\}$
$T \to sT$	$\{s\}$
$A \to aA$	$\{a\}$
$A \to \varepsilon$	$\{b, y\}$
$B \to bB$	$\{b\}$
$B \to \varepsilon$	$\{y\}$

Множество первых порождаемых символов для продукции $B \to \varepsilon$ есть $\{y\}$, поскольку y может следовать за B. Здесь B генерирует пустую строку.

Определение 38. *LL(1)-грамматика* – это грамматика, для которой множества первых порождаемых символов для каждой продукций являются непересекающимися.

Термин $LL(1)$ означает следующее: первая буква L означает чтение слева направо (*Left*), вторая буква L означает крайнее левое порождение (от англ. *leftmost*), цифра 1 означает использование одного символа предпросмотра. Более общие $LL(k)$-грамматики используют k символов предпросмотра.

Грамматика из примера 42 является $LL(1)$ грамматикой, поскольку множества первых порождаемых символов для нетерминалов A, B и T не пересекаются.

$LL(1)$-грамматики весьма ограничены в том смысле, что они подходят к ограниченному множеству КС-грамматик. Одним из фундаментальных ограни-

чений является следующее. Если грамматика включает левую рекурсию, она не является $LL(1)$-грамматикой.

В самом деле, рассмотрим грамматику

$$D \to Dx|y \tag{4.7}$$

Здесь для нетерминала есть две возможные альтернативы. Однако, у этих альтернатив есть общие стартовые терминалы. Для второй альтернативы это, очевидно, $\{y\}$, а для первой – множество стартовых символов совпадает с множеством стартовых символов нетерминала *D*, т.е. $\{y\}$.

Рассмотрим один из фундаментальных методов анализа, который применим к широкому классу КС-грамматик.

4.3. Метод рекурсивного спуска

Прежде чем определить алгоритм рекурсивного спуска (англ. *recursive descent parser*), зафиксируем основные требования к синтаксическому анализатору. Первым требованием является время работы алгоритма разбора, оно должно быть пропорциональным длине входной цепочки. Кроме того, алгоритм должен быть *корректным*, т.е.:

1. Должен быть способен распознать любую цепочку, принадлежащую языку.

2. Должен отвергать цепочки, не принадлежащие языку.

3. Не должен зацикливаться при любом вводе.

Метод рекурсивного спуска (РС) реализует нисходящий синтаксический разбор (сверху-вниз) с помощью рекурсивных функций.

Для каждого нетерминала создается отдельная функция. Функция называется так же, как и нетерминал. Функция должна найти во входной строке подцепочку, выводимую из этого нетерминала (или сообщить об ошибке).

Для написания исходного кода каждой такой функции используются правила вывода соответствующей грамматики. При этом нетерминал в правой части правила вывода эквивалентен вызову процедуры, соответствующей данному нетерминалу.

Работа алгоритма может быть описана следующим образом. Нисходящий разбор (сверху-вниз) всегда начинается с символа предложения *S*. В точке входа в программу (функция *main*) вызывается функция *S*. Задача функции определить, выводится ли входная цепочка из символа *S*.

В теле функции *S* могут вызваться другие функции, поскольку в процессе вывода могут встречаться нетерминалы. Предполагается, что все цепочки обязательно заканчиваются маркером конца ввода ⊥. При этом концом ввода может служить как конец исходной программы (конец файла), так и конец текущей строки.

Пример 43. Рассмотрим грамматику

$$S \to ABd \tag{4.8}$$

$$A \to a|cA \tag{4.9}$$

$$B \to bA \tag{4.10}$$

Данная грамматика, очевидно, относится к классу КС-грамматик. Для начала проверим принадлежность цепочки $cabad$ языку, описываемому данной грамматикой. Построим левый вывод:

$$S \Rightarrow ABd \Rightarrow cABd \Rightarrow caBd \Rightarrow cabAd \Rightarrow cabad \tag{4.11}$$

Цепочка принадлежит языку. Построим дерево вывода для данной цепочки (рис. 4.1).

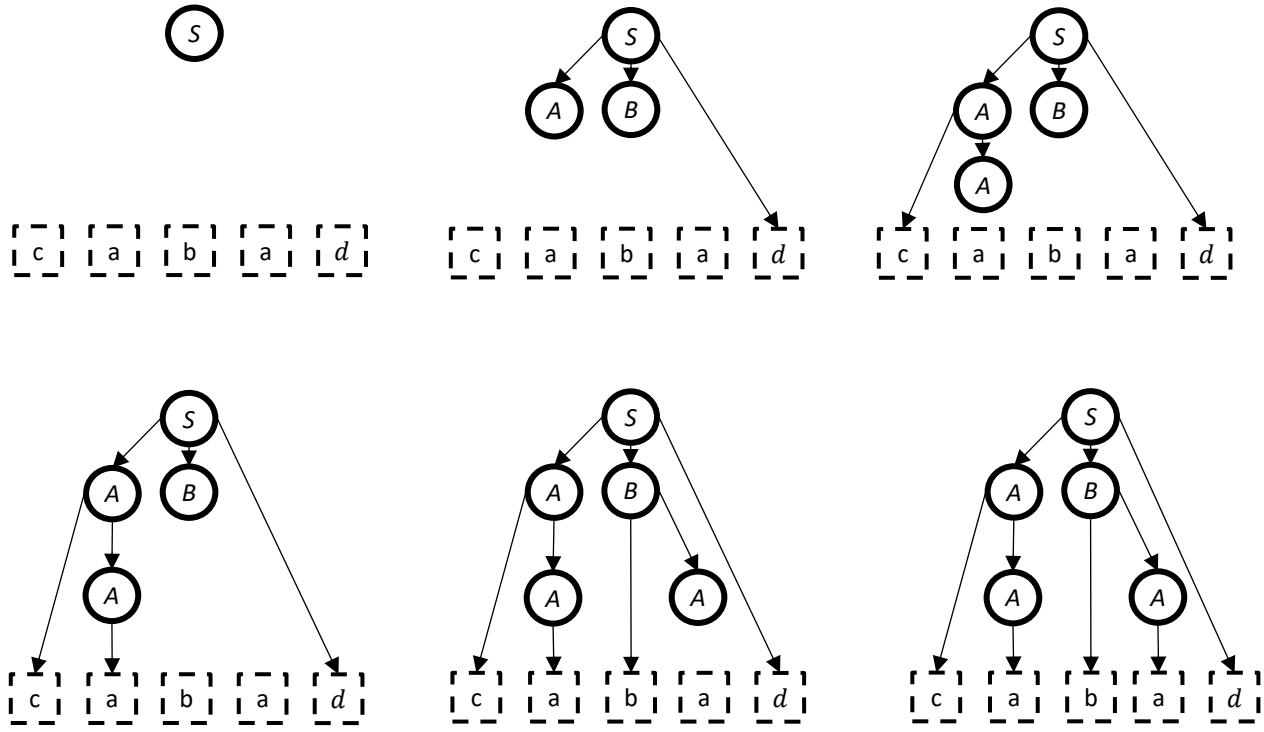

Рис. 4.1

Рассматриваемая цепочка не заканчивается на символ конца ввода, поэтому добавим в правила вывода следующее

$$M \to S \perp \tag{4.12}$$

В результате функция main будет соответствовать продукции (4.12). Опишем РС-анализатор на языке высокого уровня C++ (листинг 11).

Листинг 11

```
#include <iostream>
using namespace std;

int c; // текущий символ просмотра
void A();
```

```
void B();
void gc() { cin >> c; }

void S() {
     A();
     B();
     if (c != 'd') throw c;
     gc();
}
void A() {
     if (c == 'a') gc();
     else if (c == 'c') {
          gc();
          A();
     }
     else throw c;
}
void B() {
     if (c == 'b') {
          gc();
          A();
     }
     else throw c;
}
int main() {
     try {
          gc();
          S();
          if (c != '\0') {
               throw c;
          }
          cout << "Success" << endl;
          return 0;
     }
     catch (int c) {
          cout << "Error" << endl;
          return 1;
     }
}
```

Можно видеть, что рекурсивные процедуры практически в точности соответствуют правилам грамматики (4.8) – (4.10). Следует отметить, данная программа лишь *распознает* язык, т.е. проверяет выводится ли цепочка на входе из заданных грамматических правил. Синтаксический анализатор в каждой рекур-

сивной процедуре должен строить дерево разбора – структуру, которая отражает ход исполнения программы. Данное дерево может представлять собой множество узлов, связанных указателями.

В основе метода рекурсивного спуска лежит тот факт, что правила вывода однозначно определяются по текущему символу предпросмотра. Другими словами, для того чтобы применять метод, необходимо определить условия его применимости.

4.3.1. Достаточное условие применимости метода

Метод рекурсивного спуска применим тогда, когда каждое правило вывода имеет вид:

$$X \to \alpha, \alpha \in (T \cup N)^* \tag{4.13}$$

$$X \to a_1\alpha_1 | a_2\alpha_2 | \dots | a_n\alpha_n, a_i \in T, \alpha_i \in (T \cup N)^* \tag{4.14}$$

Если правило вывода для нетерминала имеет вид (4.13), то оно должно быть единственным.

Правил вида (4.14) может быть несколько, но все они должны начинаться с разных терминалов.

Алгоритм РС относится к нисходящим методам анализа с *прогнозируемым выбором альтернатив*. Грамматику, которая удовлетворяет требованиям (4.13), (4.14), называют *s*-грамматикой.

Вернемся к примеру 43. Выше мы рассмотрели разбор заданной цепочки. Определим процесс вывода любой цепочки языка, задаваемого данной грамматикой. Процесс построения вывода любой цепочки можно упростить, если построить таблицу прогнозов (табл. 4.3).

Таблица 4.3

	a	***b***	***c***	***d***
S	$S \to ABd$	$S \to ABd$	$S \to ABd$	$S \to ABd$
A	$A \to a$		$A \to cA$	
B		$B \to bA$		

По приведенной таблице последовательность действий при разборе строки такова:

1. По данной входной строке начать построение вывода с начального символа S.

2. Если получается сентенциальная форма вида $wY\alpha$, то:

2.1. Если начало формы w не совпадает с началом цепочки, сообщить об ошибке.

2.2. Иначе, если следующим за w символом в цепочке является символ z, заменить нетерминал Y на правую часть соответствующего правила из табл. 4.3 (ячейка на пересечении строки Y и столбца z).

2.3. Если ячейка пуста, сообщить об ошибке.

Метод рекурсивного спуска применяется, если для грамматики существует таблица однозначных прогнозов.

Более общим является метод, при котором в случае неоднозначного прогноза рассматриваются все возможные альтернативы. Ошибка фиксируется только в том случае, если ни одна из альтернатив не привела к успеху. Данный метод на практике стараются не применять, так как он приводит к экспоненциальному времени, затрачиваемому на разбор текста программы.

Пример 44. Рассмотрим грамматику

$$S \to aA|B|d$$
$$A \to d|aA$$
$$B \to aA|a$$

Данная грамматика неоднозначна, поскольку невозможно дать однозначный прогноз, если анализируемая цепочка начинается с символа a. Здесь возможны альтернативы: $S \to aA$ или $S \to B$.

Пример 45. Грамматика вида

$$S \to A|B$$
$$A \to aA|d$$
$$B \to aB|b$$

не позволяет сделать однозначный прогноз. Каждая цепочка выводится из S единственным способом, следовательно, грамматика однозначна. Однако, нельзя по одному символу предпросмотра сказать, с какого из правил вывода следует начинать анализ. Для того, чтобы сделать выбор необходимо просмотреть всю входную цепочку до конца. К грамматикам из примеров 44 и 45 РС-метод не применим.

Определение 39. $first(\alpha)$ – множество терминалов, с которых начинаются цепочки $\alpha \in (T \cup N)^*$ заданной грамматики $G = \{T, N, P, S\}$.

Утверждение 7. Если в грамматике присутствуют правила вывода вида $X \to \alpha|\beta$, где α и β начинаются с одних и тех же символов (т.е. $first(\alpha) \cap first(\beta) \neq \emptyset$), то метод рекурсивного спуска для данной грамматике неприменим.

Пример 46. Пусть дана грамматика

$$S \to aA$$
$$A \to BC|B$$

$$C \to b|\varepsilon$$
$$B \to \varepsilon$$

Здесь пересечение множества стартовых символов для каждой альтернативы пусто, однако одна и та же цепочка может иметь несколько деревьев вывода. В самом деле, для цепочки, состоящей из одного символа a, получаем два разных дерева вывода (рис. 4.2).

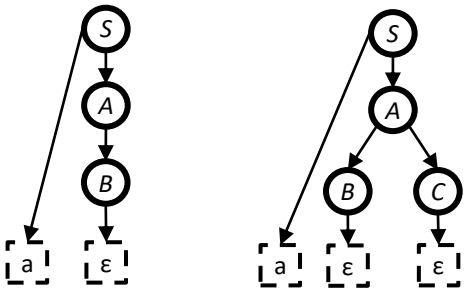

Рис. 4.2

Утверждение 8. Если в грамматике присутствуют правила $X \to \alpha|\beta$, такие что $\alpha \to \varepsilon$ и $\beta \to \varepsilon$, то метод рекурсивного спуска неприменим.

Пример 47. Рассмотрим грамматику

$$S \to cAd|d$$
$$A \to aA|\varepsilon$$

Здесь пересечение множеств стартовых символов, с которых начинается каждая альтернатива $first(\alpha)$, пусто, но в правилах присутствует пустая строка. Для ответа на вопрос, можно ли в данном случае применять метод рекурсивного спуска, построим таблицу прогнозов.

Таблица 4.4

	a	***c***	***d***
S		$S \to cAd$	$S \to d$
A	$A \to aA$	$A \to \varepsilon$	$A \to \varepsilon$

По таблице видно, что нескольким входным символам (c и d) соответствуют одинаковые правила. Однако, в данном случае неоднозначность можно разрешить следующим образом. Пусть текущий анализируемый символ a. В это случае применяется правило $A \to aA$, иначе правило $A \to \varepsilon$.

Известно, что за любой подцепочкой, выводимой из A, должен следовать символ d. Рекурсивная функция A в случае применения правила $A \to \varepsilon$, возвращает управление в точку вызова (функцию S). Функция S считывает очередной символ. Если этот символ не является символом d, генерируется ошибка.

На языке высокого уровня эта ситуация разрешается так, как показано в листинге 12.

Листинг 12

```
void A() {
     if (c == 'a') {
          gc();
          A();
     }
     // Возврат в точку вызова
}
```

В данном случае метод рекурсивного спуска применим. Рассмотрим еще один пример.

Пример 48. Пусть дана грамматика

$$S \to Bd$$
$$B \to cAa|a$$
$$A \to aA|\varepsilon$$

Здесь для нетерминала A правила вывода точно такие же как в предыдущем примере. Однако написать рекурсивную функцию A не получится. В самом деле, данная грамматика допускает сентенциальную форму вида $cAad$. Здесь, когда управление находится в функции A, по текущему символу a, невозможно выбрать альтернативу. Если воспользоваться функцией из листинга 12, то она считает следующий символ d и передаст управление самой себе, хотя символ d уже не является частью ни одной из продукций для нетерминала A.

Определение 40. Множество $follow(X)$ включает все терминалы, которые могут появляться при разборе цепочки непосредственно справа от X, т.е.

$$follow(X) = \{a \in T | S \Rightarrow \alpha X \beta, \beta = a\gamma\} \quad (4.15)$$

Утверждение 9. Метод рекурсивного спуска применим для КС-грамматик, таких что для любой пары альтернатив $A \to \alpha|\beta$, справедливо:

$first(\alpha) \cap first(\beta) = \emptyset$ (4.16)

Только одно из правил $\alpha \Rightarrow \varepsilon$ или $\beta \Rightarrow \varepsilon$ верно (4.17)

Если $\alpha \Rightarrow \varepsilon$, $first(\beta) \cap follow(A) = \emptyset$ (4.18)

Сделаем важное замечание. В общем случае не существует алгоритма, способного по данной грамматике построить эквивалентную грамматику, для которой метод рекурсивного спуска применим.

4.3.2. Построение таблицы прогнозов

Метод рекурсивного спуска в общем случае применим, если для грамматики существует таблица однозначных прогнозов. Следовательно, для корректного синтаксического разбора необходимо составить таблицу прогнозов. Она строится следующим образом.

1. Для каждого правила вида $X \to \alpha$ и для каждого терминала, с которого начинается правая часть $a \in first(\alpha)$, поместить в ячейку $[X, a]$ запись $X \to \alpha$.

2. Для каждого правила $X \to \alpha$, такого что $\alpha \Rightarrow \varepsilon$, заполнить все незаполненные на шаге 1 строки записями $X \to \alpha$.

3. Для каждого правила $X \to Y\beta$, где Y – нетерминал, поместить $X \to Y\beta$ во все оставшиеся ячейки строки X.

Пример 49. Построить таблицу прогнозов для грамматики

$$S \to A|BS|cS$$
$$B \to bB|d$$
$$A \to aA|E|\varepsilon$$
$$E \to e$$

Прежде всего необходимо убедиться в применимости метода рекурсивного спуска. Определим пересечения множеств $first$ и $follow$.

$first(A) = \{a, e\}, first(BS) = \{b, d\}, first(cS) = \{c\}, follow(S) = \emptyset$

$first(bB) = \{b\}, first(d) = \{d\}$

$first(aA) = \{a\}, first(E) = \{e\}, follow(A) = \emptyset$

$first(e) = \{e\}$

Как видим условия (4.16) – (4.18) соблюдены. Построим таблицу прогнозов.

Таблица 4.5

	a	***b***	***c***	***d***	***e***
S	$S \to A$	$S \to BS$	$S \to cS$	$S \to BS$	$S \to A$
A	$A \to aA$	$A \to \varepsilon$	$A \to \varepsilon$	$A \to \varepsilon$	$A \to E$
B		$B \to bB$		$B \to d$	
E					$E \to e$

4.3.3. Канонические КС-грамматики.

Грамматика называется канонической, если

1. Правило $X \to \alpha$ – это единственное правило для нетерминала X.

2. $X \to a_1\alpha_1| \dots |a_n\alpha_n$, где все терминалы a_i попарно различны.

3. $X \to a_1\alpha_1|a_2\alpha_2| \dots |a_n\alpha_n|\varepsilon$, где все терминалы a_i попарно различны и $first(X) \cap follow(X) = \emptyset$.

Для канонических грамматик фазу построения таблицы прогнозов можно опустить. Алгоритм разбора приобретает вид: если текущий символ равен a_i, выбирается правило, начинающееся с a_i, если присутствует пустая альтернатива (ε), выбирается она, иначе фиксируется ошибка.

При описании языков программирования множество терминалов T фиксировано, а множество нетерминалов N может варьироваться. Это дает возможность для преобразования грамматики в более удобную для анализа форму. Например, правило вывода

$$X \to \alpha\{\beta\}\gamma$$

можно преобразовать в

$$X \to \alpha Y \gamma$$
$$Y \to \beta Y | \varepsilon$$

Здесь фигурные скобки обозначают *итерацию* (повторения ноль или более раз).

В языках программирования часто встречаются итеративные конструкции: списки параметров функции, объявления нескольких переменных и т.д. Правило вывода для таких конструкций может иметь вид:

$$L \to a\{, a\}$$

Это правило может быть преобразовано к виду, для которого применим *метод рекурсивного спуска*.

$$L \to aM$$
$$M \to , aM | \varepsilon$$

Рекурсивная процедура, которая реализует итерацию имеет вид (листинг 13).

Листинг 13

```
void L() {
     if (c != 'a') throw c;
     gc();
     while (c == ',') {
          gc();
          if (c != 'a') throw c;
          gc();
     }
}
```

Рассмотрим еще один пример.

Пример 50. Дана грамматика с правилами вывода

$$S \to LB \perp$$
$$L \to a\{, a\}$$
$$B \to , b$$

которая допускает строки вида a, a, a, b. Использовать для разбора анализатор из листинга 13 нельзя, поскольку функция L(), захватит чужую запятую (которая присутствует в правиле для B), и далее не обнаружив символа a, сообщит об ошибке. Попробуем преобразовать грамматику следующим образом

$$S \to LB \perp$$
$$L \to aM$$
$$M \to , aM | \varepsilon$$
$$B \to , b$$

Здесь условия применимости метода рекурсивного спуска не выполняются, ибо $first(, aM) \cap follow(M) = \{,\} \neq \emptyset$.

На практике в данном случае последовательность терминалов a следует отделять от терминала b символом отличным от запятой (например, точка с запятой). В этом случае пересечение множеств $first$ и $follow$ стало бы пустым и метод рекурсивного спуска стал бы применим.

4.4. Синтаксический анализатор для модельного языка

Здесь мы построим систему рекурсивных процедур для распознавания грамматики модельного языка программирования из раздела 3.2.

Будем считать, что лексический и синтаксической анализаторы взаимодействуют следующим образом. Если синтаксическому анализатору для анализа требуется очередная лексема, он запрашивает ее у лексического анализатора. Таким образом, разбор исходного текста программы идет под управлением подпрограммы синтаксического анализатора, который реализован, как C++ класс *Parser*.

Для того, чтобы в явном виде представить множество рекурсивных функций, перепишем грамматические правила следующим образом:

$P \to \boldsymbol{program}\ D1\ B \perp$
$D1 \to \boldsymbol{var}\ D\ \{, D\}$
$D \to I\ \{, I\}: [\boldsymbol{int}|\boldsymbol{bool}]$
$B \to \boldsymbol{begin}\ S\ \{; S\}\ \boldsymbol{end}$
$S \to I \coloneqq E | \boldsymbol{if}\ E\ \boldsymbol{then}\ S\ \boldsymbol{else}\ S \mid \boldsymbol{while}\ E\ \boldsymbol{do}\ S \mid B \mid \boldsymbol{read}(I) \mid \boldsymbol{write}(E)$
$E \to E1\{[= \mid > \mid < \mid >= \mid <= \mid !=]E1\}$
$E1 \to T\{[+ \mid - \mid \boldsymbol{or}\ ]\ T\}$
$T \to F\{[* \mid / \mid \boldsymbol{and}\ ]\ F\}$
$F \to I \mid N \mid L \mid \boldsymbol{not}\ F \mid (E)$
$L \to \boldsymbol{true} \mid \boldsymbol{false}$
$I \to C \mid IC \mid IR$
$N \to R \mid NR$
$C \to a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z$
$R \to 0 \mid 1 \mid \ldots \mid 9$

Правила для нетерминалов *L*, *I*, *N*, *C* и *R* описаны на этапе лексического разбора. Следовательно, остается описать функции для нетерминалов *P*, *D1*, *D*, *B*, *S*, *E*, *E1*, *T*, *F*. В листинге 14 приведен пример функции для нетерминала *D*, который отвечает за объявление переменных в программе.

Листинг 14

```
void Parser::D()
{
    reset();
    if (c_type != LEX_ID) throw curr_lex;
    push(c_val);
    gl();
    while (c_type == LEX_COMMA) {
        gl();
        if (c_type != LEX_ID) throw curr_lex;
        else {
            push(c_val);
            gl();
        }
    }
    if (c_type !=LEX_COLON) throw curr_lex;
    gl();
    if (c_type == LEX_INT) {
        this->dec(LEX_INT);
        gl();
    }
    else if (c_type == LEX_BOOL) {
        this->dec(LEX_BOOL);
        gl();
    }
    else throw curr_lex;
}
```

Здесь переменная *cur_lex* – текущая считанная лексема, *c_type* хранит тип текущей лексемы, *c_val* – ее значение. Кроме того, в теле функции *D* используются методы, которые непосредственного отношения к синтаксическому анализу не имеют, а именно: *push()* и *reset()*.

Эти методы являются примером дополнительных *семантических проверок*. Некоторые особенности языка не могут быть описаны контекстно-свободной грамматикой. К таким правилам относятся, например,

- любой идентификатор, используемый в теле программы должен быть описан;

- повторное описание одного и того же идентификатора не разрешается;

- в операторе присваивания типы идентификаторов должны совпадать;

- в условном операторе и операторе цикла в качестве условия допустимы только логические выражения и др.

Указанные особенности языка разбираются на этапе *семантического анализа*. Удобно процедуры семантического анализа совместить с процедурами синтаксического анализа. На практике это означает, что в рекурсивные функции встраиваются дополнительные контекстно-зависимые проверки. Например, на этапе лексического анализа в таблицу **TID** заносятся данные обо всех лексемах-идентификаторах, которые встречаются в тексте программы. На этапе синтаксического анализа в ту же таблицу заносятся данные о типе идентификатора (поле *type*) и о наличии для него описания (поле *declared*).

С учетом сказанного, правила вывода для нетерминала *D* принимают вид:

$$D \rightarrow stack.reset()\ I\ stack.push(c\_val)$$
$$\{, I\ stack.push(c\_val)\}$$
$$: [\boldsymbol{int}\ dec(LEX\_INT)$$
$$|\ \boldsymbol{bool}\ dec(LEX\_BOOL)]$$

Здесь *stack* – структура данных, в которую запоминаются идентификаторы (номера строк в таблице **TID**), *dec* – функция, задача которой заключается в занесении информации об идентификаторах (поля *type* и *declared*), а также контроль повторного объявления идентификатора.

Пример семантической проверки на повторное объявление одного и того же идентификатора приведен в листинге 14 в теле рекурсивной процедуры *D*.

СПИСОК ЛИТЕРАТУРЫ

1. Свердлов С. З. Языки программирования и методы трансляции: учебное пособие. – Санкт-Петербург: Лань, 2019. – 564 с.
2. Малявко А. А. Формальные языки и компиляторы: учебное пособие для вузов. – М.: Юрайт, 2018. – 429 с.
3. Миронов С. В. Формальные языки и грамматики: учебное пособие для студентов факультета компьютерных наук и информационных технологий. – Саратов: СГУ, 2019. – 80 с.
4. Антик М. И., Казанцева Л. В. Теория формальных языков в проектировании трансляторов: учебное пособие. – М.: МИРЭА, 2020. – 57 с.
5. Ахо А. В., Лам М. С., Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии и инструментарий. – М.: Вильямс, 2008. – 1184 с.
6. Ишакова Е.Н. Теория языков программирования и методов трансляции: учебное пособие. – Оренбург: ИПК ГОУ ОГУ, 2007. – 137 с.
7. Гинзбург С. Математическая теория контекстно-свободных языков. – М.: Мир, 1970. – 328 с.
8. Хантер Р. Основные концепции компиляторов. – М.: Вильямс, 2002. – 252 с.
9. Рейуорд-Смит В. Теория формальных языков. Вводный курс. – М.: Радио и связь, 1988. – 128 с.
10. Пратт Т. Языки программирования: разработка и реализация. – СПб.: Питер принт, 2002. – 688 с.

СВЕДЕНИЯ ОБ АВТОРЕ

Унгер Антон Юрьевич, к.т.н., доцент кафедры вычислительной техники Института информационных технологий РТУ МИРЭА