Титульный лист материалов по дисциплине

ДИСЦИПЛИНА *Теория формальных языков*

ИНСТИТУТ *Информационных технологий*

КАФЕДРА *Вычислительной техники*

ВИД УЧЕБНОГО МАТЕРИАЛА *Лекция*

ПРЕПОДАВАТЕЛЬ *Унгер Антон Юрьевич*

СЕМЕСТР *3 семестр*

8. СИНТАКСИЧЕСКИЙ АНАЛИЗ

Синтаксический анализатор является сердцем транслятора, по крайней мере той его части, который ответственен на анализ исходного кода. Его вход соединяется с выходом лексического анализатора таким образом, что синтаксический анализатор может запросить от лексического требуемое количество лексем. В целом, именно структура синтаксического анализатора определяет структуру всего транслятора. Это становится очевидным. если вспомнить, что именно в процессе синтаксического разбора появляется универсальное представление – *абстрактное синтаксическое дерево*, которое является основой для дальнейшего анализа.

8.1. 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:

$$S \to XY; \quad (5.1)$$
$$X \to xX; \quad (5.2)$$
$$X \to x; \quad (5.3)$$
$$Y \to yY; \quad (5.4)$$
$$Y \to y. \quad (5.5)$$

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

$$S \Rightarrow XY \Rightarrow xXY \Rightarrow xxXY \Rightarrow xxxY \Rightarrow xxxyY \Rightarrow xxxyy. \quad (5.6)$$

Разбор производится слева направо. Здесь мы на каждом шаге заменяем крайний левый нетерминал. Первый шаг очевиден, поскольку для нетерминала *S* есть только одна продукция (5.1).

Следующий шаг не так очевиден, поскольку для нетерминала *X* существует более одной продукции (5.2) и (5.3). 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 (5.2).

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 (5.4). После этого в строке остается последний символ *y*, поэтому для него используется продукция (5.5).

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 5. Здесь в первом столбце уже просмотренные символы зачеркнуты. Первый не зачёркнутый символ и является символом предпросмотра.

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

8.2. 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.

Таблица 5

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

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 таблице 6.

Таблица 6

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

В самом деле, рассмотрим грамматику

$$D \to Dx|y. \tag{5.7}$$

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.

8.3. Метод рекурсивного спуска

Прежде чем определить анализатор рекурсивного спуска (англ. *recursive descent parser*) [15], 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 $\perp$. При этом концом ввода может служить как конец исходной программы (конец файла), так и конец текущей строки.

Пример 43. Рассмотрим грамматику

$$S \to ABd; \tag{5.8}$$

$$A \to a|cA; \tag{5.9}$$

$$B \to bA. \tag{5.10}$$

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

$$S \Rightarrow ABd \Rightarrow cABd \Rightarrow caBd \Rightarrow cabAd \Rightarrow cabad. \tag{5.11}$$

Цепочка принадлежит языку. Построим дерево вывода для данной цепочки (рис. 23).

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

$$M \to S \perp. \tag{5.12}$$

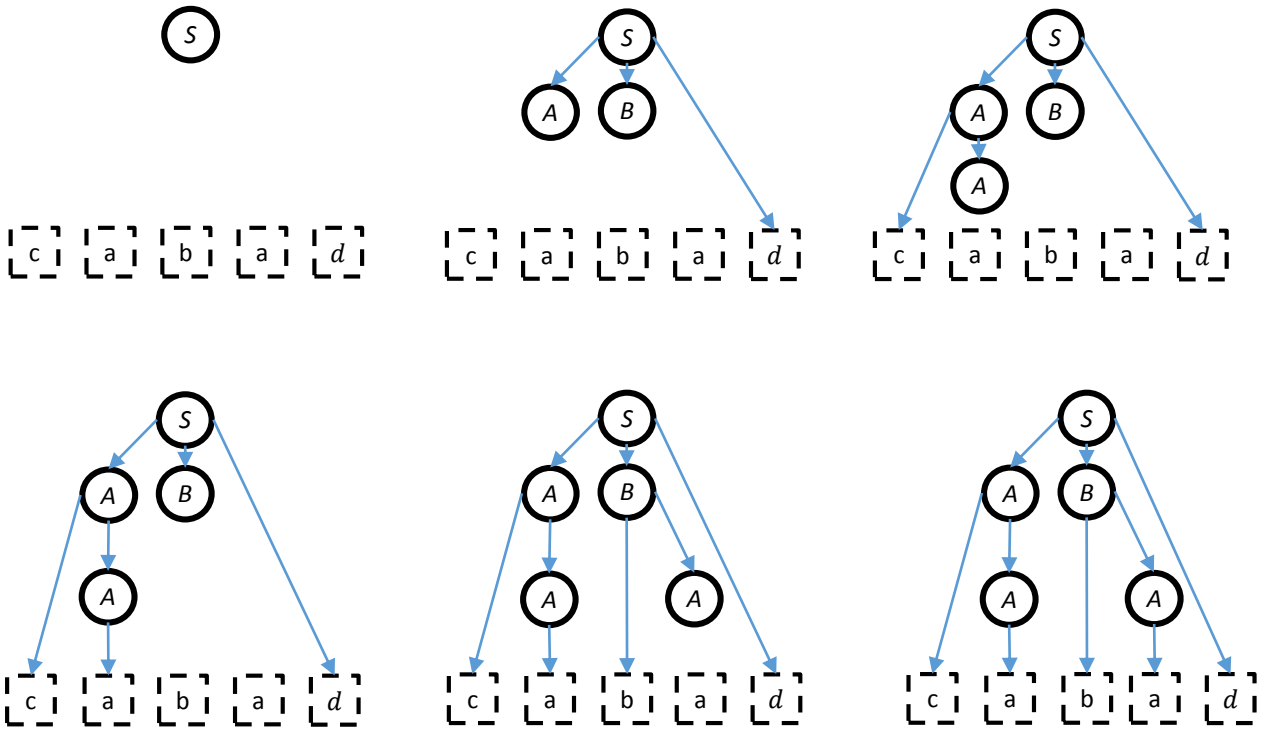

Рисунок 23. Дерево вывода для примера 43

В результате функция *main* будет соответствовать продукции (5.12). 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();
    }
```

Окончание листинга 11

```
    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 (5.8) – (5.10). Следует отметить, данная программа лишь *распознает* язык, т.е. проверяет выводится ли цепочка на входе из заданных грамматических правил. Синтаксический анализатор в каждой рекурсивной 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.

8.3.1. Достаточное условие применимости метода

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

$$X \to \alpha, \alpha \in (T \cup N)^*; \qquad (5.13)$$

$$X \to a_1\alpha_1 | a_2\alpha_2 | \dots | a_n\alpha_n, a_i \in T, \alpha_i \in (T \cup N)^*. \quad (5.14)$$

Если правило вывода для нетерминала имеет вид (5.13), то оно должно быть единственным.

Правил вида (5.14) 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 (5.13), (5.14), 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. 7).

Таблица 7

	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. 7 (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.24).

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.

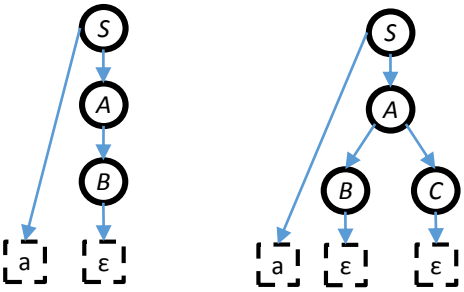

Рисунок 24. Дерево вывода для примера 46

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 8

	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();
```

Окончание листинга 12

```
      }
      // 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.

$$follow(X) = \{a \in T | S \Rightarrow \alpha X \beta, \beta = a\gamma\}. \quad (5.15)$$

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

$$first(\alpha) \cap first(\beta) = \emptyset; \quad (5.16)$$

$$\text{Только одно из правил } \alpha \Rightarrow \varepsilon \text{ или } \beta \Rightarrow \varepsilon \text{ верно}; \quad (5.17)$$

$$\text{Если } \alpha \Rightarrow \varepsilon, first(\beta) \cap follow(A) = \emptyset. \quad (5.18)$$

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

8.3.2. 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 (5.16) – (5.18) соблюдены. Построим таблицу прогнозов.

Таблица 9

	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					$E \to e$

8.3.3. Канонические КС-грамматики.

Грамматика называется канонической, если

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

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

3. $X \to a_1\alpha_1|a_2\alpha_2|\ldots|a_n\alpha_n|\varepsilon$, 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.

8.4. Синтаксический анализатор для модельного языка

Здесь мы построим систему рекурсивных процедур для распознавания грамматики модельного языка программирования из раздела 4.2.

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 → program D1 B⊥
D1 → var D {,D}
```

```
D → I {,I}:[int|bool]
B → begin S {;S} end
S → I ≔E| if E then S else S | while E do S | B | read(I)
| write(E)
E → E1{[=|>|<|>=|<=|!=]E1}
E1 → T{[+|-|or] T}
T → F{[*|/|and] F}
F → I | N | L| not F | (E)
L → true | false
I → C | IC | IR
N → R | NR
C → a | b |…| z | A | B |...| Z
R → 0 | 1 |…| 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 [16]. 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*.