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*

7. ЛЕКСИЧЕСКИЙ АНАЛИЗ

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, так называемое 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.

7.1. Формы Бэкуса-Наура

Описание грамматики с помощью набора продукций с теоретической точки зрения 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**

7.1.1. Расширенные формы 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> |
[<число>].<число> [<порядок>]
<число> ::= <цифра>{<цифра>}
<порядок>::=(E|e)[+|-]<число>
```

7.1.2. 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. 20).

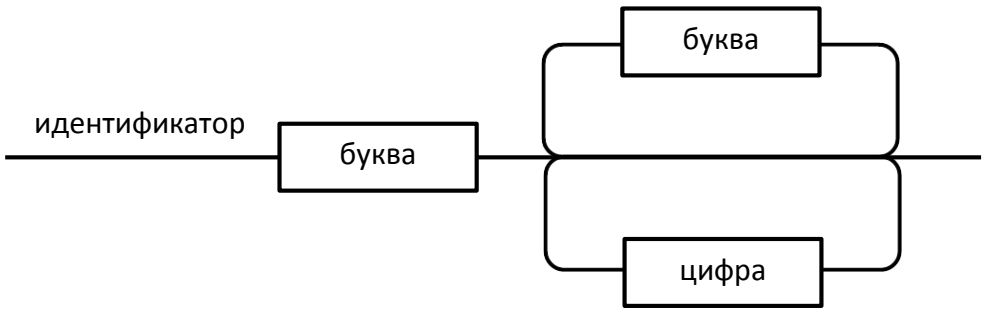

Рисунок 20. Диаграмма Вирта для идентификатора

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

7.2. 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.

7.2.1. Грамматика модельного языка

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

$$< \text{программа} > ::= \boldsymbol{program\ var}\ < \text{описание} > \{, < \text{описание} >\} \quad (4.1)$$
$$\boldsymbol{begin} < \text{оператор} > \{; < \text{оператор} >\} \boldsymbol{end}$$

$$< \text{описание} > ::= < \text{идентификатор} > \{, < \text{идентификатор} > \} \quad (4.2)$$
$$: < \text{тип} >$$

$$< \text{тип} > ::= \boldsymbol{int} \mid \boldsymbol{bool} \quad (4.3)$$

$$< \text{оператор} > ::= < \text{составной} > \mid < \text{присваивания} > \quad (4.4)$$

| < условный > | < цикла > | < ввода > | < вывода >

< составной >::= ***begin*** < оператор > {; < оператор > }***end*** (4.5)

< присваивания >::= < идентификатор >:=< выражение > (4.6)

< условный >::= ***if*** < выражение > ***then*** < оператор > (4.7)

[***else*** < оператор >]

< цикла >::= ***while*** < выражение > ***do*** < оператор > (4.8)

< ввода >::= ***read*** (< идентификатор >) (4.9)

< вывода >::= ***write*** (< выражение >) (4.10)

< выражение >::= < логическое > (4.11)

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

< логическое >::=< равенство > (4.12)

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

< равенство >::=< сравнение > (4.13)

| < равенство > = < сравнение >

| < равенство > ! = < сравнение >

< сравнение >::=< сложение > (4.14)

| < сравнение > < < сложение >

| < сравнение > > < сложение >

| < сравнение > <= < сложение >

| < сравнение > >= < сложение >

< сложение >::=< умножение > (4.15)

| < сложение > +< умножение >

| < сложение > −< умножение >

< умножение >::=< унарное > (4.16)

| < умножение >∗< унарное >

| < умножение >/< унарное >

< унарное >::=< первичное > (4.17)

|***not*** < унарное >

< первичное >::=< идентификатор > (4.18)

| < константа >

| (< выражение >)

< константа >::=< число > (4.19)

|(***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. 21).

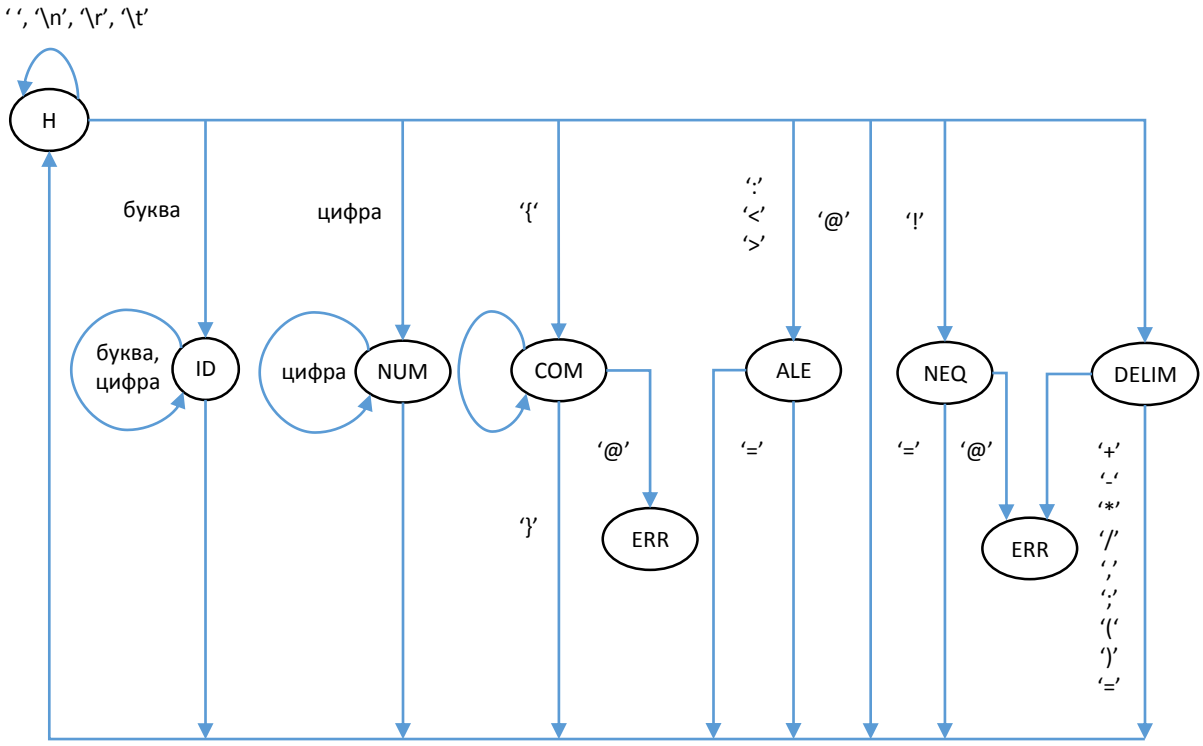

Рисунок 21. Диаграмма состояний лексического анализатора модельного языка

Следует отметить, что приведенная на рис. 21 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 числа без знака. 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. 22.

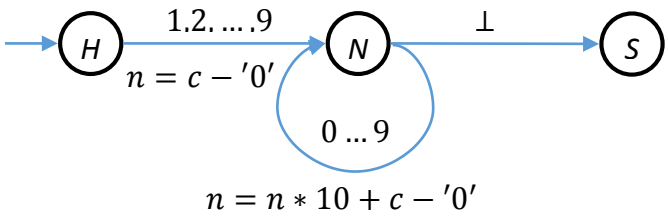

Рисунок 22. Диаграмма состояний для распознаваний беззнаковых чисел

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. 21) рассматривается в листингах ниже, которые содержат фрагмент подпрограммы лексического анализатора модельного языка.

7.2.2. 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
```

Окончание листинга 5

```
    LEX_PLUS,        // 29
    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;
```

```
    }
```

Окончание листинга 8

```
    ~TID() {
        delete [] p;
    }
    Id& operator [] (int k) {
        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. 21) включает 7 состояний (листинг 9).

Листинг 9

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

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

Листинг 10

```
Lex Lexer::gettok()
{
```

```
    int d, j;
```

Продолжение листинга 10

```
    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();
```

Продолжение листинга 10

```
                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);
```

Окончание листинга 10

```
            }
            break;
        case NEQ:
            if (c == '=') {
                add();
                gc();
                j = look(buf, TD);
                return Lex(LEX_NEQ, j);
            }
            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.