# Лекція 7. Лексический анализатор прикладу языка программирования

## Транслитератор DPL

### Общая организация транслитератора

Транслитератор демонстрационного языка программирования используется для выделения следующих классов отдельных символов:

**Класс букв**: содержит прописные и строчные буквы латинского алфавита, используемые при создании разнообразных конструкций языка. Русские буквы в этот класс не включаются, так как используются только внутри строк и комментариев, допускающих почти все символы.

**Класс десятичных цифр**: объединяет арабские цифры от 0 до 9. Используется при формировании описаний действительных, а также некоторых из целых чисел.

**Класс двоичных цифр**: объединяет цифры 0 и 1. Используется при анализе целых двоичных чисел.

**Класс восьмеричных цифр**: объединяет цифры от 0 до 7. Используется при анализе целых восьмеричных чисел.

**Класс шестнадцатеричных цифр**: включает цифры от 0 до 9, а также прописные и строчные буквы: A, B, C, D, E, F, a, b, c, d, e f.

**Класс пропусков**: состоит из пробела, перевода строки, табуляции, перевода формата (разделяющего текст на отдельные страницы). Символы этого класса используются для разделения различных элементарных конструкций, слитное написание которых привело бы к неправильному восприятию (например, следующие друг за другом число и идентификатор "123E4 asdf" без пробела были бы восприняты как "123E4asdf", что является ошибкой).

**Класс игнорируемых символов**: включает все символы, которые, как предполагается, не отображаются на экране текстового редактора. В используемых кодовых таблицах к ним относятся символы, коды которых меньше кода пробела. Исключение составляют перевод строки, табуляция, перевод формата, уже отнесенные к предыдущему классу. В некоторых текстовых редакторов данные символы отображаются в виде специальных значков. Поэтому, выделение данного класса может являться спорным и зависит от различных факторов.

**Класс прочих символов**: включает все оставшиеся символы. Не смотря на то, что их тоже можно группировать в различные классы, в рассматриваемом языке нам, в большинстве ситуаций, достаточно использовать их непосредственные значения

Следует отметить, что классы символов пресекаются. Однако, вопрос принадлежности нужному классу можно решать, основываясь на текущем контексте. Класс символов можно

специально не хранить, а проверять тогда, когда потребуется.

## Программная реализация транслитератора

Реализация транслитератора, как и любого другого программного модуля, во многом зависит от стиля программирования. Несмотря на простоту, эта задача может быть рассмотрена и в более широком контексте. Транслитератор можно использовать как совокупность нескольких функций, проверяющих принадлежность символа к одному из классов. Такая реализация выглядит вполне логичной при процедурном программировании. Вместе с тем следует отметить, что транслитератор не используется сам по себе. Он является промежуточным звеном между функциями, обеспечивающими чтение символов из входного потока и лексическим анализатором. Это промежуточное звено обычно скрывает от лексического анализатора механизм чтения и преобразования символов. Сканеру так же ничего не надо знать об открытии входного потока и манипуляции с ним. Поэтому, при объектно-ориентированном подходе, получение значение нового символа, его класса, а также манипуляция входным потоком реализуются как единый объект (на основе класса). Этот класс инкапсулирует внутренние операции, а также объединяет разбросанные по программе структуры данных, используемые для ввода символов и их преобразования.

*Примечание. Одной из проблем, которая встала передо мной при подготовке данной темы, явилась проблема выбора реализации для демонстрационного примера. Что показать: простой по исполнению модуль сканера, построенный с использованием процедурного программирования, или его объектно-ориентированный аналог со всевозможными "наворотами"? Оба варианта я использовал при решении различных задач, поэтому проблем с самим кодом не было. Раздираемый мучительными противоречиями, я решил остановиться на процедурной версии в надежде на светлое будущее, в котором постараюсь добавить ОО реализацию в виде отдельного материала. В данном случае победило стремление к простоте представления материала. Не всех интересует объектно-ориентированное программирование (да и нужно ли оно?:-), а процедурное программирование (ПП) изучают практически все. Кроме того, я постарался скомпоновать модуль сканера таким образом, чтобы ОО гурманы могли легко переделать его в классы.*

В результате этого решения, транслитератор оказался представлен следующим кодом:

```
// Функции транслитератора, используемые для определения класса лексем
//
// Определяет принадлежность символа к классу букв
bool inline isLetter(int ch) {
  if((ch>='A' && ch<='Z') || (ch>='a' && ch<='z'))
    return true;
  else
    return false;
}
//
// Определяет принадлежность символа к классу двоичных цифр
bool inline isBin(int ch) {
  if((ch=='0' || ch=='1'))
    return true;
  else
    return false;
}
//
```

```
// Определяет принадлежность символа к классу восьмеричных цифр
bool inline isOctal(int ch) {
  if((ch >= '0' && ch <= '7'))
    return true;
  else
    return false;
}
//
// Определяет принадлежность символа к классу десятичных цифр
bool inline isDigit(int ch) {
  if((ch >= '0' && ch <= '9'))
    return true;
  else
    return false;
}
//
// Определяет принадлежность символа к классу шестнадцатеричных цифр
bool inline isHex(int ch) {
  if((ch >= '0' && ch <= '9') ||
     (ch >= 'A' && ch <= 'F') ||
     (ch >= 'a' && ch <= 'f'))
    return true;
  else
    return false;
}
//
// Определяет принадлежность символа к классу пропусков
bool inline isSkip(int ch) {
  if(ch == ' ' || ch == '\t' || ch == '\n' || ch == '\f')
    return true;
  else
    return false;
}
//
// Определяет принадлежность к классу игнорируемых символов
bool inline isIgnore(int ch) {
  if(ch>0 && ch<' ' && ch!='\t' && ch!='\n' && ch!='\f')
    return true;
  else
    return false;
}
//
// Читает следующий символ из входного потока
static void nxsi(void) {
  if((si = getc(infil)) == '\n') {
    ++line; column = 0;
  }
  else ++column;
    ++poz; // Переход к следующей позиции в файле
}
```

При использовании объектно-ориентированного подхода все эти данные можно инкапсулировать в один класс, обеспечив доступ к ним через соответствующий интерфейс. Следует также отдельно отметить последнюю строку функции nxl():

```
++poz; // Переход к следующей позиции в файле
```

Она присутствует только в непрямом лексическом анализаторе и предназначена для фиксации позиции в файле, что позволяет осуществлять откаты назад в том случае, если проверяемая

версия о значении лексемы не подтвердится. Взятие следующего символа в прямом сканере происходит без использования этого "довеска".

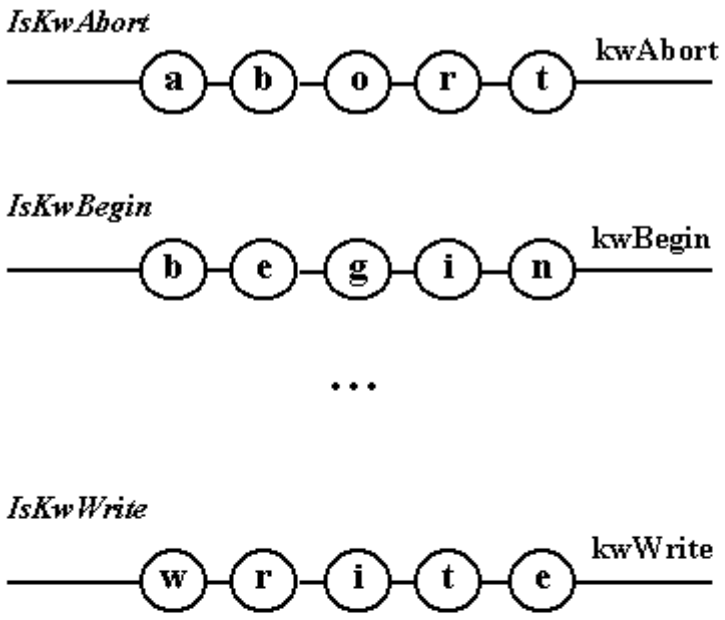
# Непрямой лексический анализатор DPL

Непрямой лексический анализатор, в соответствии с ранее описанной теорией, реализуется как совокупность независимых конечных автоматов, проверяющих принадлежность к отдельным лексемам. А эти автоматы, как было показано ранее, можно описать с использованием диаграмм Вирта. Практическая же целесообразность диктует следующие, используемые мною, технические решения:

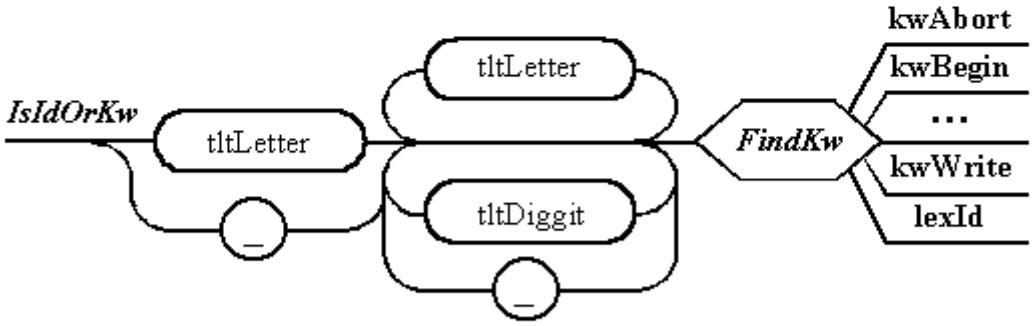
Конечный автомат, осуществляющий распознавание одной конкретной лексемы, может быть разбит на несколько, более мелких автоматов, каждый из которых распознает непересекающееся подмножество цепочек, принадлежащих искомому классу. Например, при распознавании целого числа можно построить отдельные автоматы для выявления двоичных, восьмеричных, десятичных, десятичных с префиксом и шестнадцатеричных чисел. Это упрощает реализацию отдельных автоматов. Для упрощения реализации можно также осуществлять создание автоматов, распознающих цепочки, являющиеся подцепочками других автоматов. В этом случае необходимо таким образом строить арбитраж, чтобы распознавание более длинных цепочек осуществлялось раньше. В качестве примера, я искусственно разбил распознаватель действительного числа аж на пять автоматов. Конечно, в реальной ситуации, "дробить" таким образом действительное число вряд ли имеет смысл. Однако трудно было удержаться от искушения и не продемонстрировать один из технических приемов.

Автоматы можно не только разделять, но и объединять, что ведет к использованию фрагментов прямого лексического анализа. Оставим эту технику для прямого лексического анализатора.

Можно также использовать и семантический анализ, что позволяет сократить код и ускорить разбор. На рис. 5.1а показан возможный вариант разбора ключевых слов.




а) Непосредственный анализ ключевых слов.




б) Семантическое выделение ключевых слов из анализа идентификатора

Рис. 5.1. Повешение эффективности анализа ключевых слов в непрямом лексическом анализаторе за счет использования семантической обработки.

Для распознавания каждого ключевого слова можно построить свой автомат. В этом случае возникает несколько проблем:

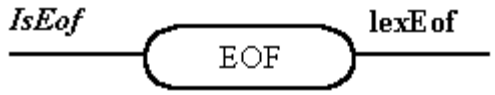
- замедляется разбор, что связано с постоянными откатами, используемыми в непрямом лексическом анализаторе (чем больше автоматов, тем медленнее разбор, а ключевых слов может быть много);
- состав ключевых слов может постоянно меняться (особенно, в новом языке), что ведет к необходимости модификации кода программы.

Гораздо проще и быстрее провести распознавание ключевых слов с использованием семантической обработки. Чаще всего (а в данном случае - это факт) ключевые слова являются

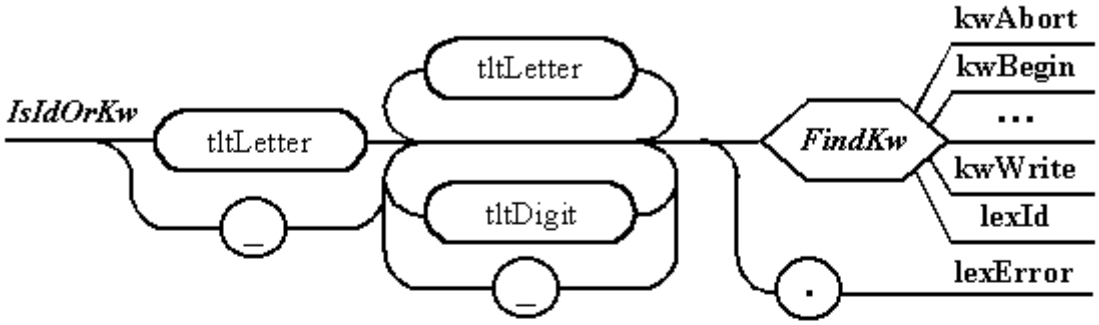
подмножеством идентификаторов. Поэтому, можно в начале осуществить выявление идентификатора, а затем провести его анализ на принадлежность к ключевому слову. Такой анализ можно осуществлять поиском (лучше всего двоичным) значения полученного идентификатора в таблице ключевых слов. При обнаружении совпадения формируется лексема, соответствующая выявленному ключевому слову. В противном случае выдается лексема - идентификатор. Соответствующая этому случаю диаграмма Вирта, вместе с блоком семантического разбора (представленного шестиугольником) приведена на рис. 5.1б. Аналогичную схему имеет смысл использовать и в прямом лексическом анализаторе.

## Диаграммы Вирта для отдельных автоматов непрямого лексического анализатора

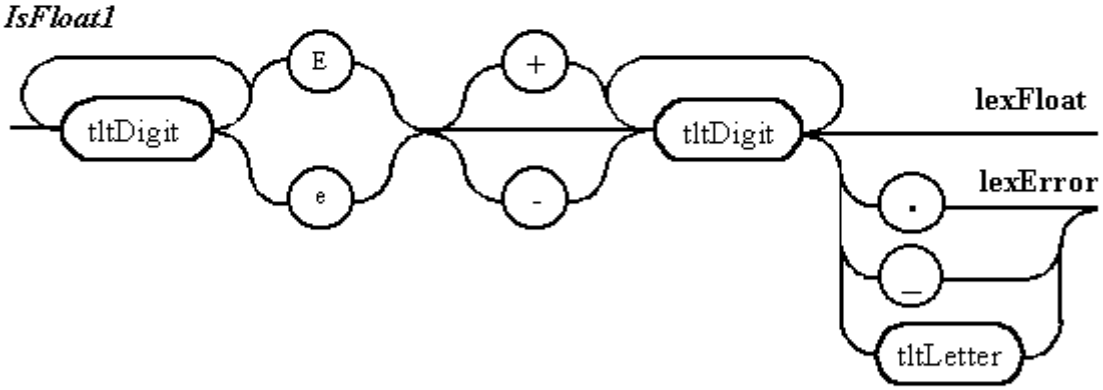
Диаграммы Вирта, описывающие отдельные независимые фрагменты непрямого лексического анализатора, представлены на рис. 5.2. В отличие от диаграмм, используемых для описания пользовательского синтаксиса, данные схемы помечены именами, которые предполагается использовать в программе. Выходы диаграмм идентифицируют порождаемые лексемы. Каждая из диаграмм непосредственно не связана с механизмом отката. Этим занимается сам анализатор.

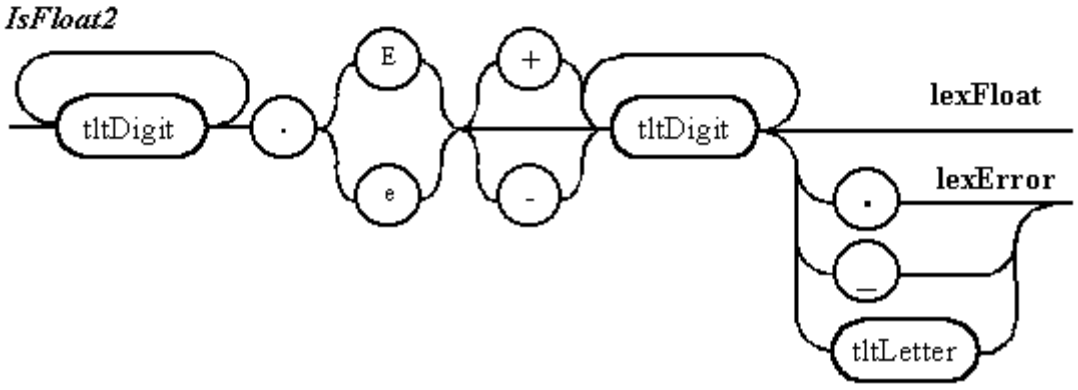


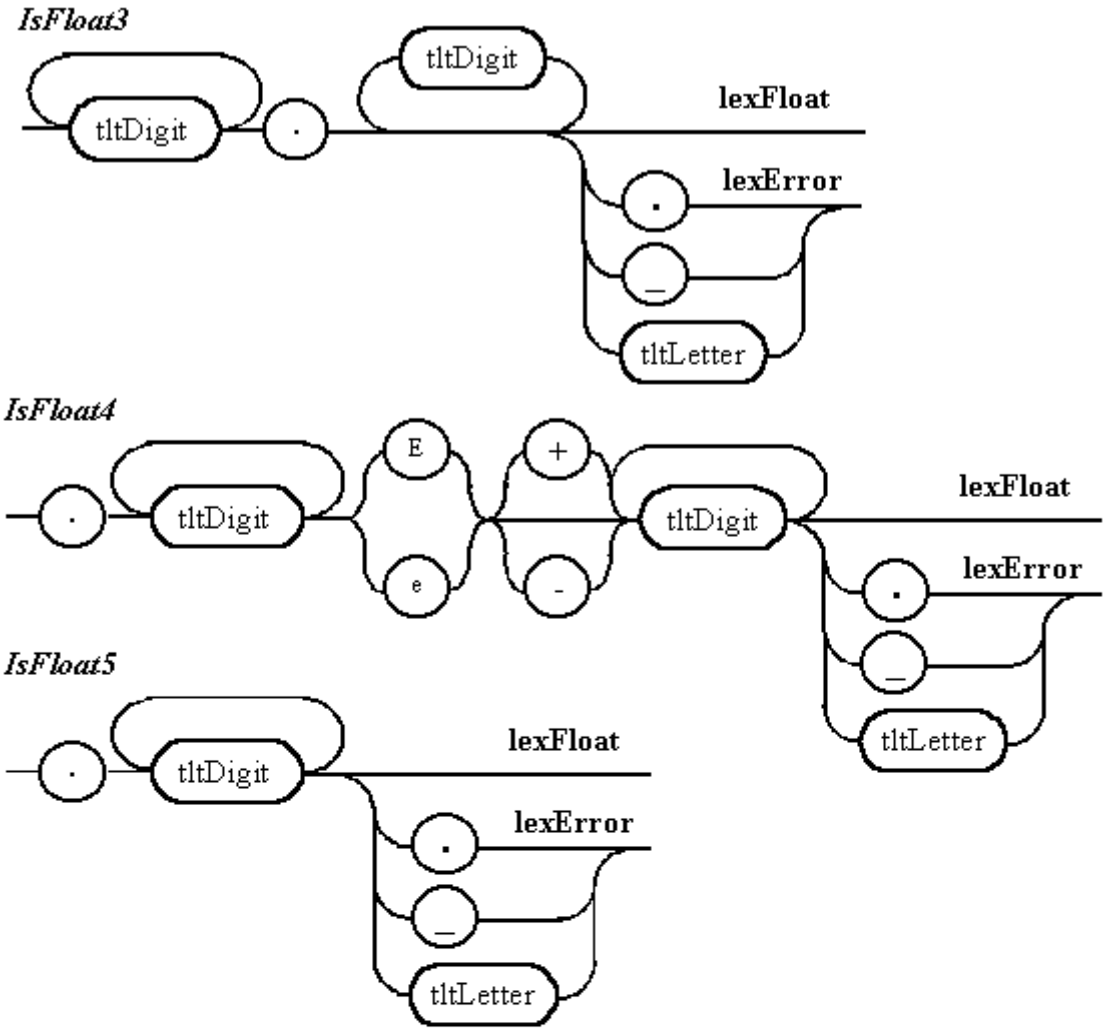

*Примечание 1.* Лексема, порождаемая при достижении конца обрабатываемого текста.



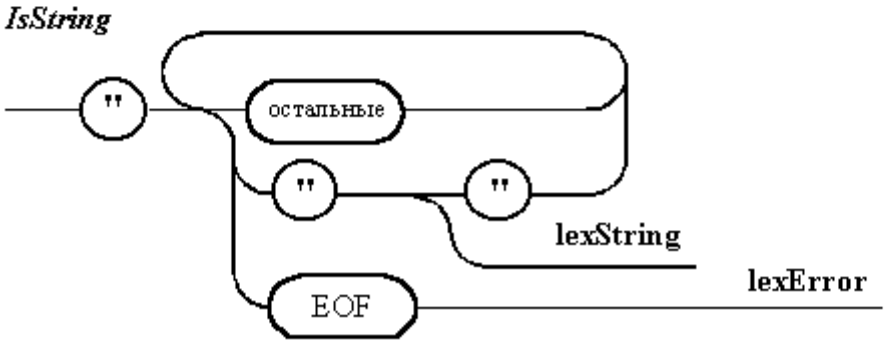

*Примечание 2.* Идентификатор и ключевые слова описываются правилом с семантической вставкой. Осуществляется также анализ на недопустимость возможного слияния идентификатора с действительным числом, начинающимся с точки.



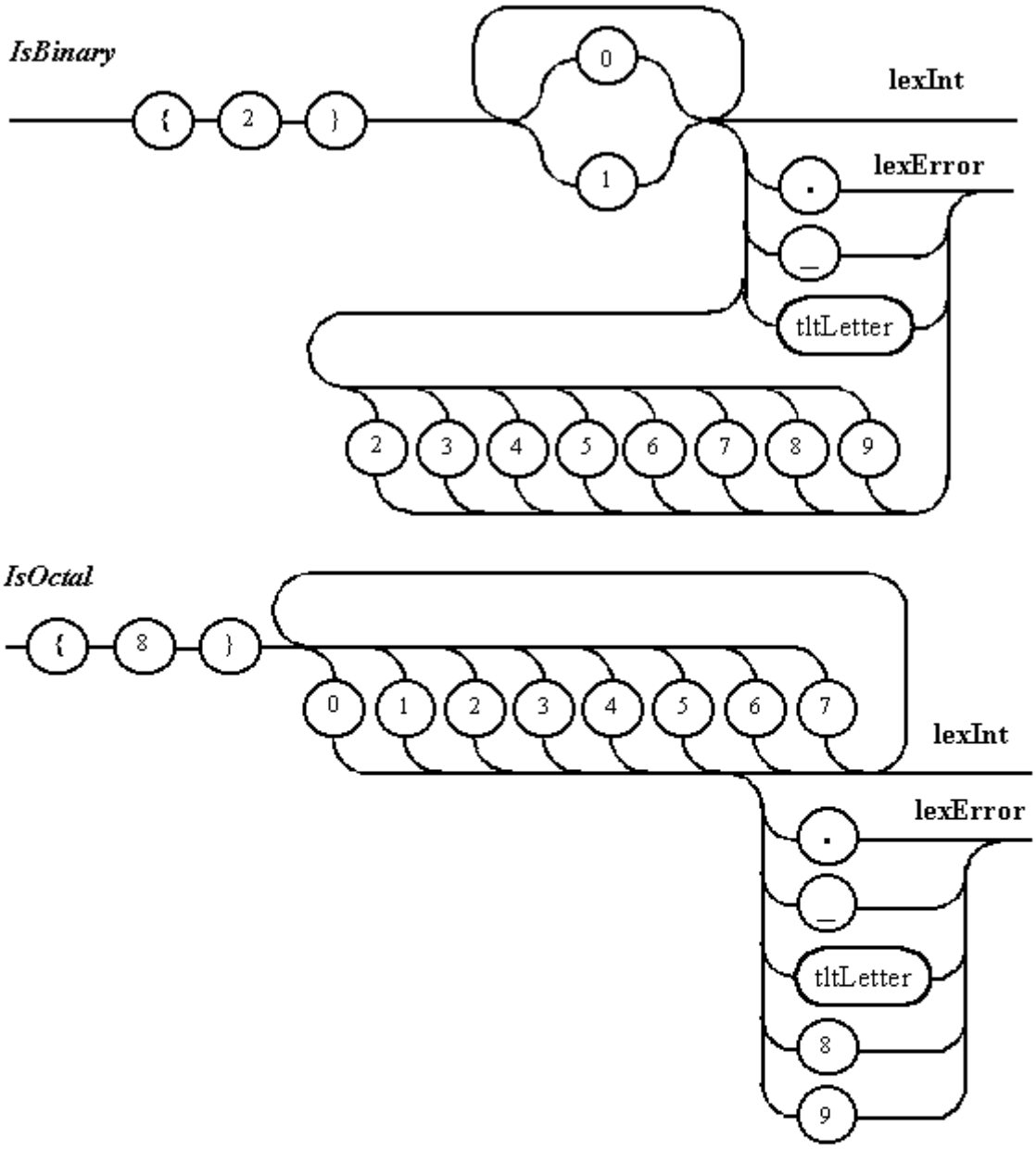





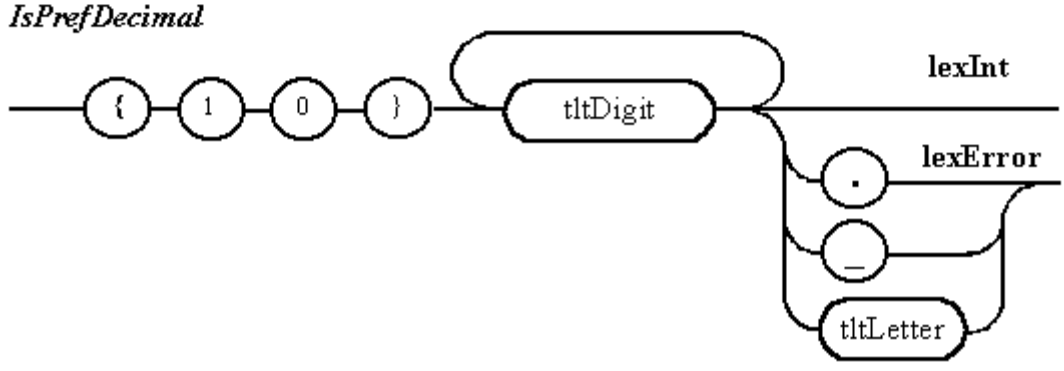

*Примечание 3.* Пять вариантов правил для распознавания действительного числа приводятся только для демонстрации арбитража при непрямом лексическом анализе. На практике легко можно обойтись одним правилом. Выдача ошибки происходит, если действительное число не отделяется разделителем от идентификатора или другого действительного числа, начинающегося с десятичной точки

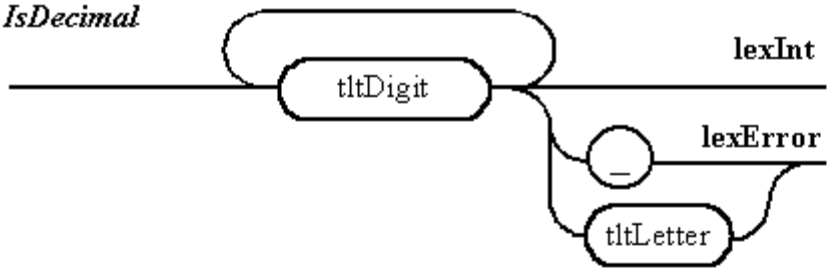



*Примечание 4.* Под остальными понимаются все символы, кроме апострофа (') и конца файла

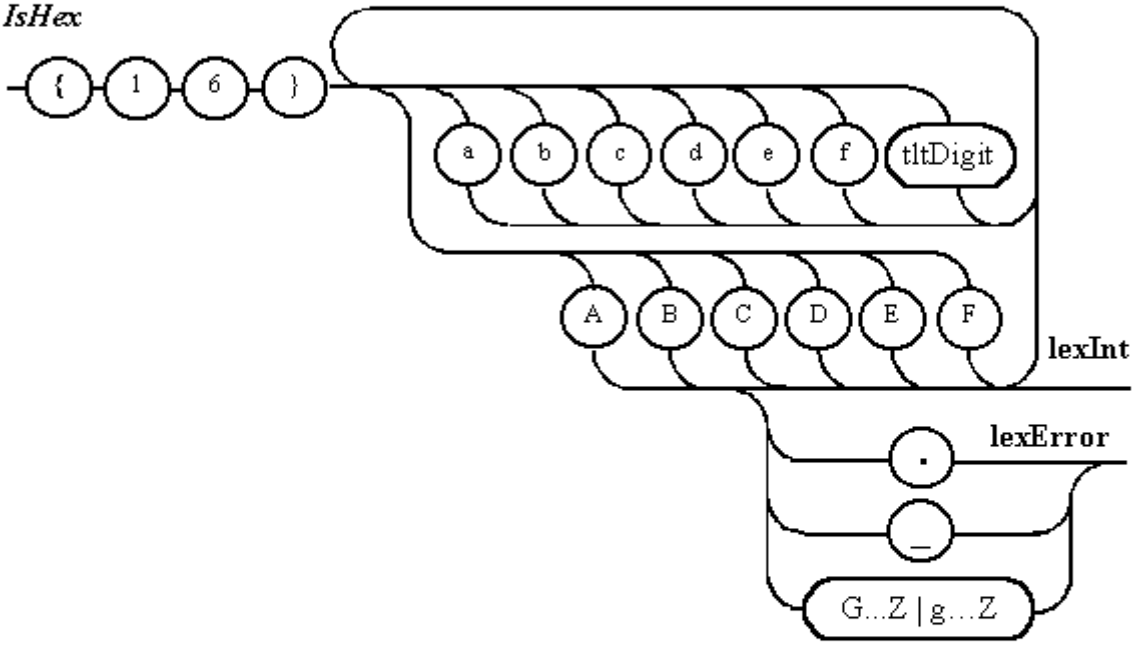



*Примечание 5.* Для двоичных и десятичных целых чисел необходима проверка того, что оно не сливается с теми цифрами, которые в них не содержатся.

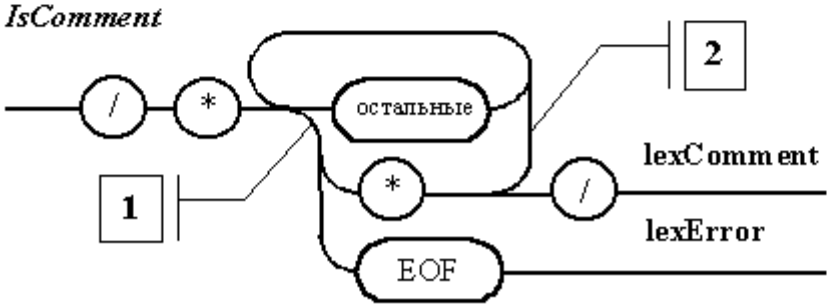




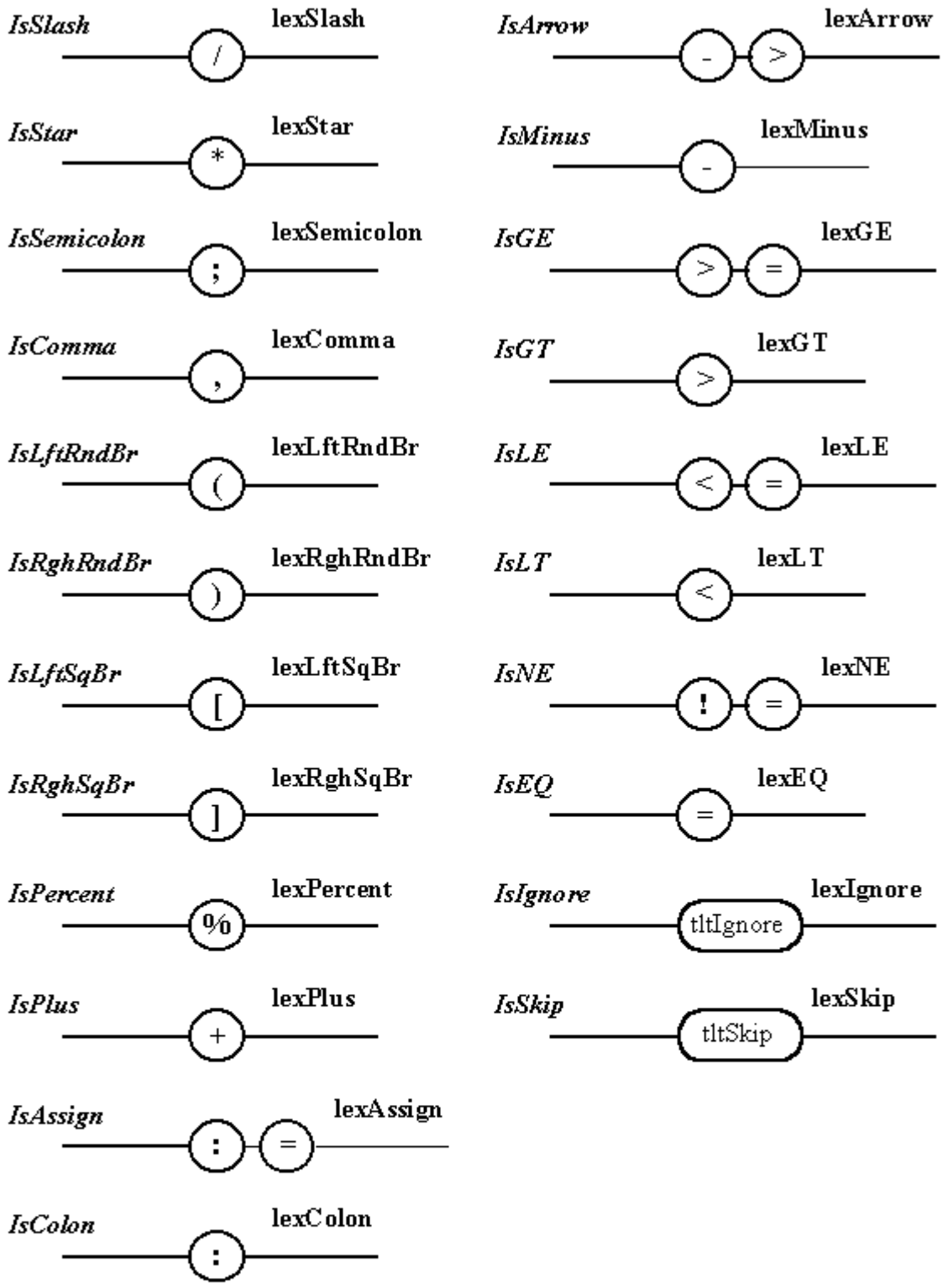

*Примечание 6.* Для целого десятичного числа без префикса анализ на недопустимость точки излишен, так как похожая ситуация должна была быть проанализирована раньше для действительного числа.




*Примечание 7.* Для целого шестнадцатеричного проверка на недопустимость должна исключать прописные и строчные буквы, используемые в самом числе. На представленных диаграммах это показано сокращенной записью путем задания диапазона. Это сделано для того, чтобы не загромождать диаграмму деталями.




*Примечание 8.* Под остальными понимаются символы не рассматриваемые непосредственно в текущей точке. В точке 1 – это не «*» и не конец файла; в точке 2 – это не «*», не конец файла и не «/»




*Примечание 9.* Лексемы, определяющие разделительные символы, расположены в соответствии с их приоритетом при анализе сверху вниз и слева направо.

Рис. 5.2. Описание с помощью диаграмм Вирта лексем, распознаваемых непрямым