Федеральное агентство по образованию

Сибирский федеральный университет

Легалов А.И., Швец Д.А., Легалов И.А.

ФОРМАЛЬНЫЕ ЯЗЫКИ И ТРАНСЛЯТОРЫ

Учебное пособие по циклу лабораторных работ

Красноярск 2007

УДК 681.3.06
ББК

Рецензенты:

Учебное пособие содержит описание 7 лабораторных работ по курсу «Формальные языки и трансляторы», направленных на освоение методов разработки трансляторов. В ходе выполнения лабораторных работ студенты получают практические навыки программирования компонент транслятора, разрабатывают синтаксические диаграммы и используют их для написания кода.

Предназначено для бакалавров, обучающихся по направлению 230100 «Информатика и вычислительная техника». Оно может быть также полезно начинающим и специалистам, занимающимся изучением методов разработки трансляторов.

ОГЛАВЛЕНИЕ

Введение

Дисциплина «Формальные языки и трансляторы» предназначена для изучения логики функционирования трансляторов, методов их разработки, используемого при разработке математического аппарата: теории формальных языков и формальных грамматик, метаязыков.

Знания, полученные при изучении дисциплины должны предоставить в распоряжение обучаемого набор методов, обеспечивающих проведение исследований в области создания языков программирования, а также послужить основой для практического освоения реальных инструментов и технических средств, применяемых при создании программного обеспечения, разработке и эксплуатации вычислительных систем. Учебное пособие по циклу лабораторных работ содержит описание 7 лабораторных работ.

В первой лабораторной работе изучаются методы построения таблицы имен используемой в трансляторе для хранения программных объектов. Рассматриваются различные способы построения таблиц и основные функции, обеспечивающие работу с ними.

Лабораторная работа 2 посвящена созданию пользовательского синтаксиса языка программирования в виде диаграмм Вирта по исходному представлению в виде расширенных форм Бэкуса-Наура. Каждый студент при этом выполняет свой вариант задания. Разработанные диаграммы в дальнейшем используются в последующих лабораторных работах, трансформируясь в соответствии с решаемыми задачами.

В лабораторной работе 3 осуществляется разработка и реализация лексического анализатора на основе диаграмм Вирта. На основе разрабатываемых тестов осуществляется проверка правильности работы лексического анализатора.

Лабораторная работа 4 посвящена созданию распознавателя. Обучаемые преобразуют диаграммы Вирта к КС(1) виду и разрабатывают по ним соответствующую программу.

В лабораторной работе 5 осуществляется подключение таблицы имен к распознавателю. Реализуется работа с таблицей имен, обеспечивающая корректное использование переменных разрабатываемого языка программирования.

Лабораторная работа 6 посвящена генерации промежуточного представления. Формируются структуры данных описывающих это представление. Пишется программа, формирующая промежуточное представление и сохраняющее его в файле.

В ходе выполнения лабораторной работы 7 осуществляется генерация кода на языке высокого уровня. Проводится окончательное тестирование полученного транслятора на разработанных примерах. Демонстрируются результаты выполнения.

Лабораторная работа №1
Разработка и использование таблицы имен

Цели и задачи: Изучение основных способов организации одноуровневых таблиц имен, функций работы с ними и эффективности использования различных методов их организации.

Время: 4 часа

Работа с таблицами имен проводится в различных программах обработки текстов, например, трансляторах, системах управления базами данных, макропроцессорах, гипертекстовых редакторах и так далее. Для успешного создания таких продуктов разработчику необходимо знать основные функции для работы с таблицами имен и структуры данных, используемые для их организации.

Порядок выполнения лабораторной работы

1. Ознакомиться с описанием лабораторной работы.
2. Получить задание у преподавателя. Тексты заданий выбирать на основе вариантов, представленных в описании лабораторной работы.
3. Разработать набор функций и структур данных, обеспечивающих работу с таблицами имен.
4. Составить программу, выполняющую работу с таблицами имен и сбор статистических данных.
5. Отладить программу и выполнить обработку для различных наборов данных.
6. Оценить эффективность программы. Сравнить полученную программу с программами, выполненными другими студентами.

Содержание отчета

1. Описание полученного задания.
2. Текст программы с необходимыми комментариями.
3. Результаты работы программы для различных наборов данных.
4. Графики и диаграммы, показывающие основные характеристики программы.
5. Выводы по результатам работы.

Структура элемента таблицы имен

Элемент таблицы имен содержит ключ, по которому осуществляется его поиск, и данные, обрабатываемые программой. В лабораторной работе ключом является символическое имя, состоящее из букв, цифр, знака подчеркивания, и начинающееся с буквы. Так как основной упор делается на изучение функций работы с таблицей, структура обрабатываемых данных

сводится к целочисленному счетчику, используемому для подсчета частоты встречаемости элемента с заданным именем в обрабатываемом файле.

Существуют различные методы хранения символьного ключа и обрабатываемых данных в элементе таблицы. Принципы хранения данных обычно определяются их организацией. В нашем случае имеющийся счетчик частоты встречаемости имени будет всегда храниться в скалярной целочисленной переменной. Из всего разнообразия методов хранения ключа выделим и рассмотрим следующие:

- непосредственное хранение в элементе;
- хранение в динамически выделенной памяти;
- хранение в общем символьном массиве;
- хранение в списке символьных массивов.

Непосредственное хранение ключа в элементе таблицы

Строка символов, определяющая значение ключа содержится в самом элементе (рис. 1. 1).

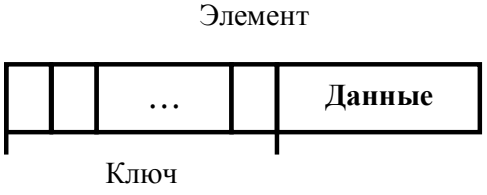


Рис. 1.1. Элемент таблицы с непосредственным хранением ключа

На языке Си элемент можно описать следующей структурой:

```
#define nameSize      10    // размер ключа
// Организация элемента таблицы имен при непосредственном
// хранении ключа в элементе.
struct element {
  char name[nameSize]; // ключ
  int  count;          // счетчик частоты встречаемости
};
```

Такой метод хранения удобен для ключей, имеющих одинаковую длину и небольшой размер (6-14 символов). Он часто используется в языках Ассемблера, Фортране, макропроцессорах.

Преимущества. Простота доступа к ключу, высокая скорость его обработки.

Недостатки. Неэффективное использование памяти при различной длине ключей и большом максимальном размере. Существование ограничителя длины имени может быть неприемлемо в ряде приложений.

Динамическое выделение памяти для ключа

Память для хранения строки символов выделяется динамически с использованием стандартных функций ОС и языков программирования (рис. 1.2).

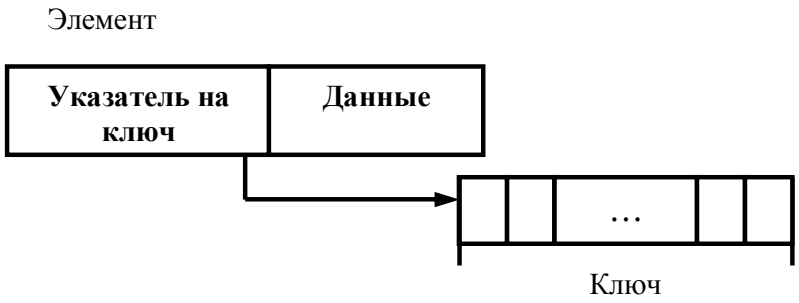


Рис. 1.2. Элемент таблицы с динамическим выделением памяти для ключа

В данном случае памяти выделяется столько, сколько нужно для хранения данного имени. В любой момент занимаемая память может быть легко освобождена. Описание структуры элемента на языке Си мало чем отличается от первого варианта.

```
// Организация элемента таблицы имен при хранении ключа
//  в динамически выделяемой памяти.
  struct element {
  char* name; // указатель на ключ
  int  count; // счетчик частоты встречаемости
};
```

Выделение памяти для ключа осуществляется с помощью оператора new. Кроме того, возможно использование функций malloc или calloc. Размер выделяемой памяти зависит от длины ключа и определяется в ходе работы программы.

Преимущества. Легкость управления доступной оперативной памятью. Динамическое выделение памяти позволяет организовать эффективное хранение ключей произвольных размеров.

Недостатки. Дополнительные затраты памяти на хранение указателей, замедление работы с ключами за счет введения дополнительных операций работы со ссылками (указателями), дополнительные временные затраты на динамическое выделение и освобождение памяти малыми порциями.

Хранение ключей в общем символьном массиве

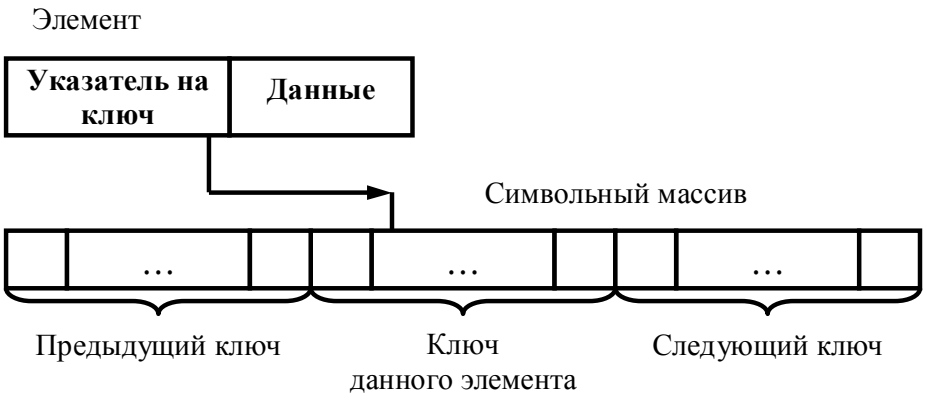
Выделяется одномерный символьный массив, в который заносятся все ключи. Обращение к элементам этого массива осуществляется по индексу или непосредственно по указателю (рис. 1.3).

Массив для хранения ключей может быть выделен как статически до начала выполнения программы, так и динамически в ходе ее работы. Вариант структуры элемента с обращением по указателю идентичен структуре, приведенной для предшествующего метода.

Использование индекса для обращения к ключу вместе с массивом для хранения ключей можно описать следующим образом:

```
#define nameArraySize  10000    // размер массива ключей
```

```
char nameArray[nameArraySize]; // массив ключей
int firstFree; // начало незанятой области массива ключей
// организация элемента таблицы имен при
// хранении ключа в специальном массиве.
struct element {
  int nameIndex;   // индекс ключа
  int  count;      // счетчик частоты встречаемости
```




```
};
```

Рис. 1.3. Элемент таблицы с хранением ключей в общем символьном массиве

Преимущества. Организовано хранение ключей произвольной длины. Отсутствуют затраты на динамическое распределение памяти. Простота вделения отдельного ключа.

Недостатки. Возможно переполнение массива и его неэффективное использование. В случае удаления элементов списка в массиве возникают «дыры», которые надо каким-либо образом ликвидировать (например, сдвигая все элементы массива) или перераспределять в дальнейшем. Обычно такая операция сжатия не реализуется, так как данный метод хранения используется только в системах массовым удалением всех элементов таблицы. Массовая же очистка массива ключей осуществляется простым обнулением индекса первого свободного символа.

Хранение ключей в списке символьных массивов

Организация этой структуры является комбинацией двух предшествующих. Память для ключей в этом случае выделяется динамически в виде достаточно длинного массива. Если выделенный массив заполнен до конца и требуется занесение нового ключа, динамически выделяется память для еще одного массива и т.д. Выделяемые массивы организуются в виде списка. Элемент таблицы содержит указатель на место хранения ключа в одном из таких массивов (рис. 1.4). Программная реализация данного метода опирается на задание сложного указателя ключа, так как, наряду с указанием местоположения в символьном массиве, необходимо еще задать и сам массив. При этом также возможно использование вариантов доступа как по индексу, так и по указателю. В последнем случае обращение к ключу может быть

осуществлено за один раз, а структура элемента будет соответствовать той, которая приведена для второго варианта.

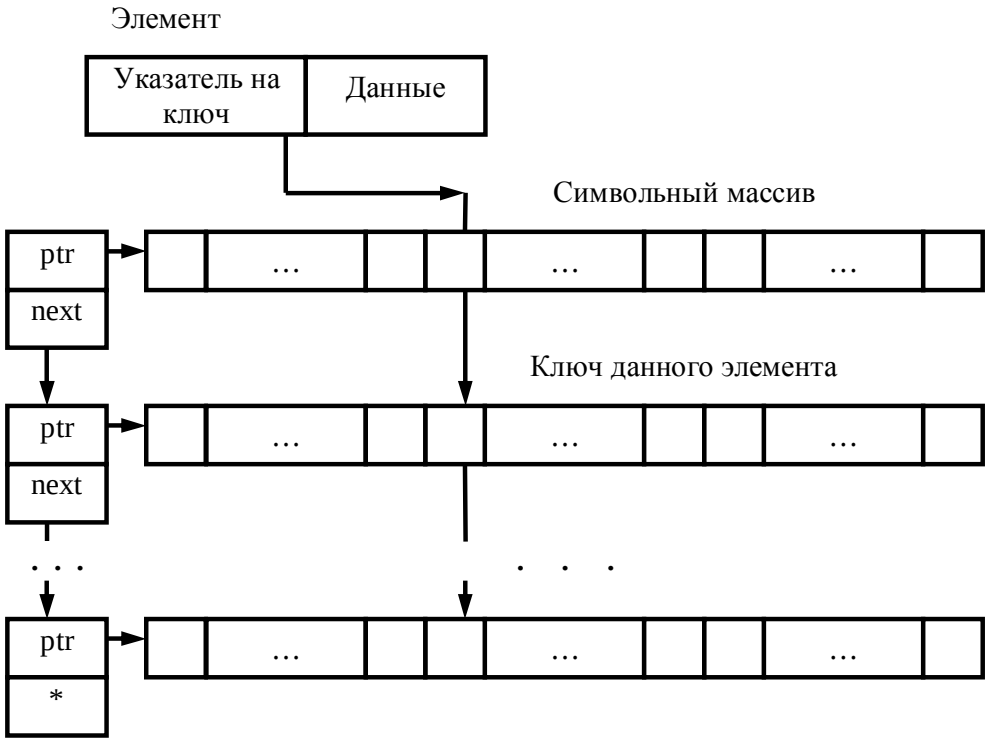


Рис. 1.4. Элемент таблицы с хранением ключей в списке символьных массивов

Пример структуры с комбинированным доступом к ключу:

```
#define nameArraySize  1000 // размер массива ключей
#define stringNumber   10   // количество строковых массивов
// список указателей на строки
char* stringArray[stringNumber];
int firstFree; // начало незанятой области массива ключей
// организация элемента таблицы имен при
// хранении ключа в специальном массиве
struct element {
  int stringIndex; // индекс символьного массива
  int nameIndex;   // индекс ключа
  int  count;      // счетчик частоты встречаемости
};
```

Преимущества. Динамическое распределение памяти позволяет эффективно использовать ресурсы. В то же время выделение памяти большими порциями обеспечивает уменьшение затрат памяти на хранение указателей и повышает эффективность алгоритмов работы с таблицей.

Недостатки. Усложняются алгоритмы работы со списком имен. Проблема «дыр», возникающих при удалении элементов, остается и для данного способа хранения ключей. Неэффективно используются последние элементы («хвосты») массивов.

Основные функции для работы с таблицами имен

При работе с таблицами имен обычно выполняются следующие действия:

- поиск в таблице имен заданного элемента с целью использования его параметров в дальнейшей обработке данных;
- поиск элемента для исключения его из таблицы;
- включение в таблицу после проведенного поиска, который удостоверяет отсутствие элемента с заданным именем;
- удаление всех элементов таблицы по завершении работы с ней (очистка таблицы).

Для выполнения этих действий используются следующие функции: поиск элемента в таблице имен, включение элемента в таблицу имен, исключение элемента из таблицы имен, очистка таблицы имен. Кроме этого, необходимо иметь функции вывода таблицы имен в файл и на экран для анализа и фиксации полученных результатов.

Упорядочение элементов таблицы

У каждого элемента имеется имя, являющееся ключом для его поиска в таблице. Метод организации хранения по этому ключу является одним из основных факторов, влияющих на эффективность работы программы. Элементы могут размещаться в таблице в произвольном порядке или быть каким-либо образом упорядочены. Произвольное размещение элементов резко снижает эффективность поиска и поэтому в реальных программах применяется крайне редко. При построении таблиц имен в данной лабораторной работе будет использоваться только упорядочение элементов по латинскому алфавиту (кириллица применяться не будет, чтобы обеспечить большую мобильность программ и упростить задачу сравнения имен).

Способы организации таблицы имен

Элементы могут объединяться в таблицы различными способами, имеющими определенные достоинства и недостатки. Из всего разнообразия выделим и рассмотрим построение таблиц на основе следующих структур данных:

- одномерных массивов;
- последовательных списков;
- хеш-таблиц;
- деревьев.

Эти структуры данных могут использоваться как в чистом виде, так и в комбинациях, образуя разнообразные варианты. Рассмотрим организацию только тех из них, которые используются в лабораторной работе.

Одномерный массив

Таблица на основе одномерного массива может быть представлена следующими структурами данных на языке Си:

```
#define tableSize 1000  // размер таблицы имен
int i_table = -1;       // индекс последнего занятого элемента
                        // для пустой таблицы равен -1
struct element table[tableSize]; // таблица имен
```

Поиск элемента. Может осуществляться по любому алгоритму. Однако наиболее эффективным для упорядоченных таблиц является двоичный поиск, при котором на каждом шаге пространство поиска делится пополам до тех пор, пока не будет найден требуемый элемент или зафиксировано его отсутствие.

Включение. Осуществляется после того, как в результате поиска подтверждено отсутствие элемента с заданным ключом. Завершение поиска обычно сопровождается нахождением индекса элемента, ключ которого предшествует исходному. Поэтому новый элемент можно вставить в таблицу вслед за ним. Вставка должна сопровождаться сдвигом всех последующих элементов на одну позицию в сторону возрастания индекса.

Удаление. После нахождения элемента в таблице его можно удалить путем сдвига всех последующих элементов на одну позицию в сторону уменьшения индекса. В связи с низкой эффективностью такого подхода можно вместо сдвига специальным образом маркировать такой элемент (например, задавать пустое имя), а само сжатие осуществлять после нескольких удалений или при полном заполнении пространства памяти, отведенного под таблицу.

Очистка таблицы. Осуществляется сбросом индекса последнего занятого элемента, то есть присваивания ему в данном случае значения, равного **-1**.

Преимущества. Организация таблицы в виде одномерного массива позволяет распределить его в непрерывной области памяти и обеспечивает непосредственный доступ к любому элементу по его индексу. Данному методу присуща простота организации, быстрота поиска элементов и очистки, наглядность структуры таблицы.

Недостатки. Максимальный размер таблицы необходимо фиксировать до ее заполнения, что не всегда удобно из-за отсутствия информации о количестве заносимых элементов. Это может привести к переполнению таблицы в процессе работы программы или наоборот к неэффективному использованию выделенного пространства памяти. При добавлении элементов в упорядоченную таблицу необходимо затрачивать дополнительное время на перемещение элементов, предшествующих вставляемому. При удалении элемента приходится тратить время на сжатие таблицы. Использование удаления с ус-

тановкой признака пустого элемента ведет к замедлению поиска и неэффективному использованию памяти (последнее, однако, не столь существенно, так как за таблицей фиксируется заранее отведенное пространство памяти).

Линейный однонаправленный список

Организация списка приведена на рис. 1.5.

Рис. 1.5. Организация однононаправленного линейного списка

Язык программирования Си позволяет описать данную структуру следующим образом:

```
struct node { // структура, определяющая элемент списка
              // и указатель на следующий элемент
  struct element val; // значение элемента списка
  struct node *next;  // указатель на следующий элемент
};
struct node *first = NULL; // указатель на первый элемент списка
                           // (обнулен, если список пуст)
```

Поиск элемента. Осуществляется от указателя на первый элемент путем последовательного перебора. Завершение поиска происходит при нахождении элемента с заданным ключом или при достижении первого элемента, ключ которого превышает искомый.

Включение. Осуществляется после завершения поиска с достижением первого элемента, ключ которого превышает искомый. Новый элемент включается перед найденным. Для выполнения операции включения необходимо иметь служебную переменную, в которой хранится указатель на предшествующий элемент списка.

Удаление. Осуществляется после успешного завершения поиска элемента с заданным ключом. Для выполнения этой операции так же необходимо хранить указатель на предшествующий элемент списка.

Массовое удаление. Осуществляется путем поэлементного удаления в цикле. При большом числе элементов в списке данный процесс может оказаться достаточно долгим.

Преимущества. Организация таблицы в виде динамического списка позволяет выделять память по мере надобности. При этом используются функции, предоставляемые операционной системой. При удалении элемента освобождающаяся область памяти легко может быть возвращена обратно в систему и повторно использоваться для других целей.

Недостатки. Поиск осуществляется только последовательно, что приводит к снижению его эффективности. Замедляет работу и процедура выделения (освобождения) памяти для элементов списка, которая осуществляется

мелкими порциями каждый раз при добавлении и удалении элементов. Медленно осуществляется массовое удаление элементов списка.

Кольцевой однонаправленный список

Этот способ организации таблиц сходен с предшествующим вариантом. Отличие заключается лишь в том, что последний элемент списка замыкается на первый, что позволяет обращаться через один вход к предшествующему и последующему элементам. Алгоритмы работы, достоинства и недостатки метода те же, что и для простого однонаправленного списка.

```
struct node { // структура, определяющая элемент списка
              // и указатель на следующий элемент
  struct element val; // значение элемента списка
  struct node *next;  // указатель на следующий элемент
};
struct node *last = NULL; // указатель на последний
                          // элемент списка
                          // (обнулен, если список пуст)
```

Однонаправленный список одномерных массивов

Для достижения компромисса между эффективностью и экономичностью динамического распределения памяти и затратами аппаратных и временных ресурсов на его организацию применяется выделение памяти сразу для группы элементов, которая организуется как одномерный массив (рис. 1.6). Этот вариант организации таблицы сходен со структурой организации хранения ключей в виде списка массивов. Выделение памяти осуществляется одновременно для группы элементов. Возможны разнообразные подходы к упорядочиванию и поиску информации. Например, сортировка данных и двоичный их поиск могут осуществляться только в пределах одномерных массивов. Переход же по списку от одного одномерного массива к другому организуется последовательно.

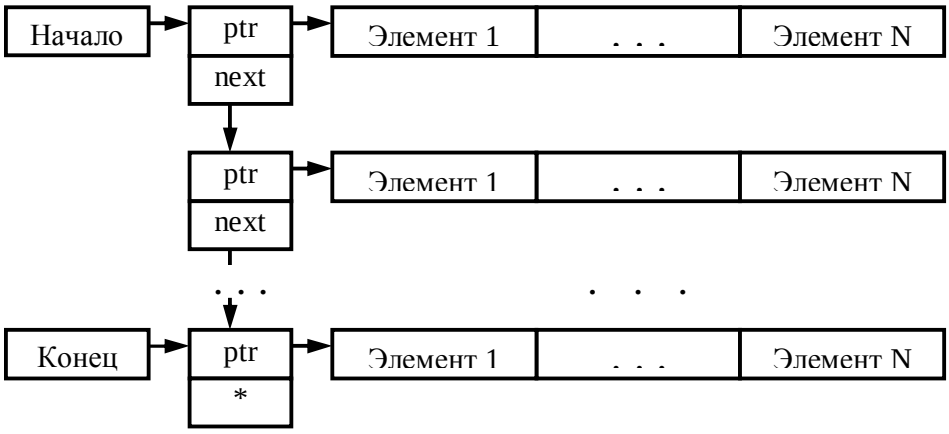


Рис. 1.6. Организация однононаправленного списка одномерных массивов

Однонаправленный кольцевой список массивов

Этот вариант сходен с предыдущим по способу организации таблицы, характеристикам, алгоритмам работы, достоинствам и недостаткам.

Хеширование

Хеширование - один из принципов идентификации информации на основе отождествления символьного ключа с числом. Занесение в таблицу, поиск и удаление элементов выполняется по данному числу. Его значение вычисляется в соответствии с выбранной хеш-функцией. Примеры хеш-функций:

- остаток от деления суммы всех кодов символов, составляющих ключ, на размер таблицы;

- остаток от деления суммы всех кодов символов по модулю 2 на размер таблицы.

Таблица строится на основе одномерного массива. Полученное число рассматривается как индекс, определяющий положение элемента в таблице. Значение хеш-функции в данном случае должно не превышать размер выделяемого под таблицу массива. Для этого, как правило, и выделяется остаток от деления полученного числа на размер таблицы.

Естественно, что при таком подходе к занесению элементов существует большая вероятность совпадения значений хеш-функции для разных имен (например, хеш-функция типа "суммирование кодов символов" будет иметь одинаковое значение для имен A2 и B1). В этом случае необходимо осуществлять эффективное разрешение конфликтов. Среди известных методов разрешения подобных конфликтов наиболее распространены следующие:

- специальный подбор формулы для хеш-функции, обеспечивающей более бесконфликтное размещение элементов в таблице (этот подбор является достаточно трудоемкой задачей);

- выбор в качестве размера таблицы простого числа, что позволяет улучшить распределение элементов, но увеличивает затраты на вычисление хеш-функции (если размер таблицы кратен двум в степени N, то вычисление модуля заменяется сдвигами числа).

- элемент таблицы, который соответствует полученному значению хеш-функции, является указателем на список элементов с одинаковыми значениями хеш-функции. Работа с этим списком ведется в соответствии с алгоритмами, описанными выше;

- если существует конфликт с уже имеющимся элементом, то новый элемент заносится в таблицу на ближайшее свободное место.

Каждый из приведенных методов разрешения конфликтов имеет свои преимущества. Однако им также присущи определенные недостатки, одним из которых является рост накладных расходов на организацию поиска,

включения и удаления. Например, если при поиске элемента по заданному ключу окажется, что искомая строка в таблице имен уже занята, необходимо первоначально убедиться в том, что в ней находится элемент с тем же ключом. Если это не так, то приходится в дальнейшем последовательно перебирать все элементы ниже найденного до тех пор, пока не будет найден требуемый ключ или первое свободное место. Проблема использования хеширования еще более усложняется, когда включение в таблицу и удаление из нее происходят в произвольных комбинациях. В этих случаях элементы с одним значением хеш-числа, для упрощения операций с ними, желательно объединять в подсписок путем введения дополнительного поля. Тогда переход от одного элемента к другому, после нахождения начала подсписка, будет осуществляться быстрее.

Преимущества. Быстрота формирования индексов элемента для разреженных таблиц.

Недостатки. Необходимость разрешения конфликтов, что замедляет в ряде случаев поиск элементов и их занесение в таблиц. Метод более удобен при последующем массовом удалении элементов.

Построение списковых структур на основе одномерных массивов

Применяется при использовании языков, не позволяющих организовать динамическое распределение памяти или при сочетании статического и динамического распределения памяти. Реализуется за счет добавления к элементу таблицы имен индекса, указывающего на следующий элемент списка (рис 1.7). Алгоритмы обработки в этом случае эквивалентны алгоритмам для моделируемого варианта организации списка.

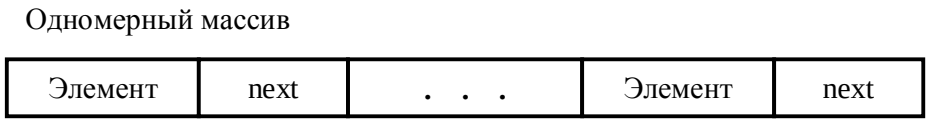


Рис. 1.7. Организация списковых структур на основе одномерных массивов

Однонаправленный линейный список

Вводятся два указателя - на первый и последний элемент списка. Кроме того, создается указатель на список свободных элементов.

Поиск элементов. Выполняется последовательно путем перебора от первого элемента списка к последнему.

Включение. Производится путем удаления элемента из списка свободных, его заполнения и внесения в список занятых.

Удаление. Выполняется следующим образом: элемент изымается из списка занятых, очищается и вносится в список свободных.

Преимущества. Простота работы, полный контроль и экономия памяти, используемой для хранения указателей.

Недостатки. Необходимость дополнительных указателей, ограничение объема используемой памяти размером массива.

Одномерный кольцевой список

Метод аналогичен предшествующему. Единственное отличие – замыкание списка ссылкой от последнего элемента на первый.

Деревья

Деревья являются одним из способов организации данных, который позволяет с высокой эффективностью осуществлять обработку таблицы имен. Элемент дерева содержит элемент таблицы и указатели на несколько потомков. Рассмотрим вариант организации таблицы в виде двоичного дерева, в котором каждый элемент может иметь не более двух потомков (рис. 1.8).

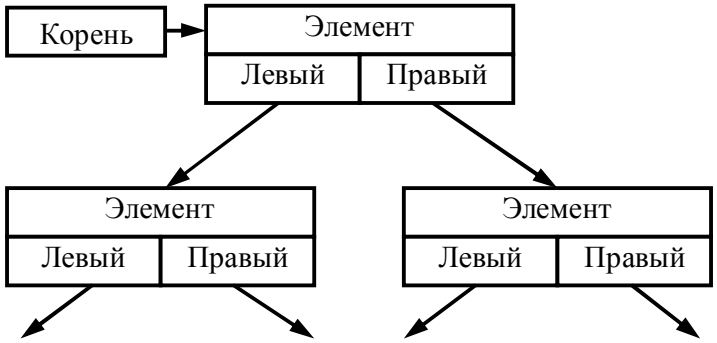


Рис. 1.8. Организация деревьев

Существуют различные классы двоичных деревьев, различающихся алгоритмами поиска, включения и удаления. Остановимся на обычном двоичном дереве.

Обычное двоичное дерево - это упорядоченная структура, в левом поддереве которого хранятся элементы, меньшие того, который хранится в корне, а в правом - большие.

Поиск осуществляется путем сравнения искомого элемента с элементом дерева и продвижением далее по правому или левому поддереву в зависимости от результата сравнения. Оптимальная организация алгоритма в этом случае - рекурсивная. Элемент не найден, если поиск заканчивается на элементе дерева, не имеющем потомков (такой элемент называется лист).

Включение. Если элемент не найден, т.е. поиск прекращен при достижении элемента дерева, не имеющего соответствующего потомка, то элемент включается на свободное место.

Удаление. Если элемент не содержит потомков, то он удаляется без каких-либо затруднений. Если у элемента один потомок, то этот потомок замещает удаляемый элемент. Если элемент содержит оба потомка, то удаляемый элемент заменяется на самый правый лист левого поддерева или на

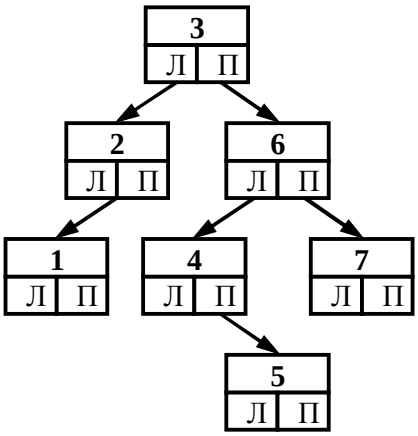
самый левый лист правого поддерева, т.е. достигаемый при движении по самой правой или самой левой ветви дерева.

Пример.

Заданная последовательность включений:

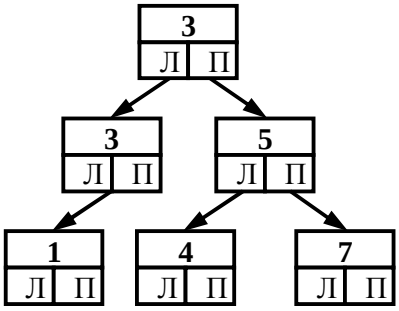
3 6 2 4 5 1 7

Полученное дерево:

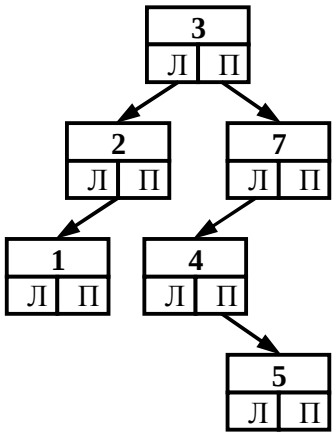


Удаление элемента со значением **6**.

1. С заменой на самый правый элемент левого поддерева:



2. С заменой на самый левый элемент правого поддерева:



Преимущества. Гибкость способа организации таблицы и простота алгоритмов обработки.

Недостатки. Частые операции с динамической памятью для выделения и освобождения элементов списка. Возможность вырождения дерева.

Возможно моделирование дерева через одномерный массив, которое осуществляется так же, как и для последовательных списков, но с двумя индексами для левого и правого поддерева.

Пример выполнения лабораторной работы

Приведем текст программы, раскрывающий специфику выполнения лабораторной работы. Кроме этих примеров, можно использовать и ряд других, хранящихся на машинных носителях и раскрывающих специфические особенности организации конкретных структур данных и функций их обработки.

Задание на лабораторную работу: разработать таблицу имен, обеспечивающую непосредственное хранение строкового ключа в элементе. Таблица должна быть построена на основе неупорядоченного одномерного массива.

```
// Файл lab1.hpp - содержит описания используемых
// структур данных и глобальных переменных

#define tableSize     1000
#define strBufSize    256
#define nameSize      10

char strBuf[strBufSize];
int i_strBuf = -1;

struct element {
  char name[nameSize];
  int  count;
};

struct element table[tableSize];
int i_table = -1;

int endfile = 0; // признак достижения конца файла сканером имен
FILE* infil;     // указатель на читаемый файл
int i_find = 0;  // индекс найденного элемента таблицы

// Файл lab1.cpp - содержит главную функцию, вспомогательные
// функции и функции для работы с таблицей имен
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
```

```
#include <sys\timeb.h>

#include "lab1.hpp"

void nextName(void); // чтение очередного имени из файла
int find(char*);     // нахождение элемента с заданным именем
void incl(char*);    // включение элемента в таблицу имен
void del(int); // удаление элемента из таблицы по его индексу
void output(char*);  // вывод таблицы имен в файл и на экран
void clear(void);    // очистка всей таблицы

void main(void) {
  // структуры для фиксации времени
  struct timeb t0, t1, t2, t3;
  int secIncl, msecIncl;
  int secDel,  msecDel;
  int secSum,  msecSum;
  printf("\nВведите имя файла для включения элементов: ");
  scanf("%s", strBuf);
  if((infil = fopen(strBuf, "rt"))==NULL) {
    printf("No infil\n"); return;
  }
  ftime(&t0);
  nextName();
  while(!endfile) {
    if(find(strBuf)) {
      table[i_find].count++;
    } else {
      incl(strBuf);
    }
    nextName();
  }
  ftime(&t1);
  output("incltab.txt");
  fclose(infil);
  printf("\nВведите имя файла для удаления элементов: ");
  scanf("%s", strBuf);
  if((infil = fopen(strBuf, "rt"))==NULL) {
    printf("No infil\n"); return;
  }
  ftime(&t2);
  nextName();
  while(!endfile) {
    if(find(strBuf)) {
      del(i_find);
    }
    nextName();
  }
  ftime(&t3);
```

```
  output("deltab.txt");
  fclose(infil);
  secIncl = t1.time - t0.time;
  msecIncl = t1.millitm - t0.millitm;
  if(msecIncl < 0) {msecIncl += 1000; --secIncl;}
  printf("Время построения таблицы: %d.%02d\n", secIncl, mse-
cIncl/10);
  secDel = t3.time - t2.time;
  msecDel = t3.millitm - t2.millitm;
  if(msecDel < 0) {msecDel += 1000; --secDel;}
  printf("Время удаления из таблицы: %d.%02d\n", secDel, msec-
Del/10);
  secSum = secIncl + secDel;
  msecSum = msecIncl + msecDel;
  if(msecSum > 1000) {msecSum -= 1000; ++secDel;}
  printf("Суммарное время: %d.%02d\n", secSum, msecSum/10);
  printf("Время вывода в файл результатов не учитывается\n");
  printf("\nNormal END of work!\n");
}

// сканер, обеспечивающий считывание имен в буфер
void nextName(void) {
  int c;
  i_strBuf = -1;
  strBuf[0] = '\0';
 while(1) {
  c = getc(infil);
  if(c==EOF) {endfile=1; return;}
  if(isalpha(c) || c=='_') {
    strBuf[++i_strBuf] = c;
    c = getc(infil);
    while(isalpha(c) || isdigit(c) || c=='_') {
      strBuf[++i_strBuf] = c;
      c = getc(infil);
    }
    // Ограничения размера ключа
    if(i_strBuf>nameSize-2) strBuf[nameSize-1] = '\0';
    else strBuf[++i_strBuf] = '\0';
    endfile=0;
    return;
  }
 }
}

// функция поиска элемента в таблице имен по заданному образцу
int find(char* n) { // n - образец для поиска
  int i;
  for(i=0; i<=i_table; ++i)
    if(!strcmp(table[i].name, n)) {
```

```
      i_find = i;
      return 1;
    }
  i_find = -1;
  return 0;
}

// Включение элемента в таблицу на последнне место,
// так как таблица неупорядочена.
void incl(char* n) {
  if(i_table == tableSize-1) {
    printf("\nTable OVERLOARD\n");
    exit(1);
  }
  strcpy(table[++i_table].name, n);
  table[i_table].count = 1;
}

// Удаление элемента из таблицы по его индексу
void del(int d) {
  int i;
  for(i=d; i<i_table; ++i) {
    strcpy(table[i].name, table[i+1].name);
    table[i].count = table[i+1].count;
  }
  --i_table;
}

// Вывод таблицы на экран и в файл
void output(char* n) { // n - имя файла
  FILE* outfil;
  int i;
  outfil = fopen(n, "wt");
  printf("\n");
  for(i = 0; i<=i_table; ++i) {
    printf("%s\t\t%d\n", table[i].name, table[i].count);
    fprintf(outfil, "%s\t\t%d\n", table[i].name, ta-
ble[i].count);
  }
  fclose(outfil);
}

// Очистка таблицы
void clear(void) {
  i_table = -1;
}
```

Варианты заданий

1. Альтернативные варианты организации элемента таблицы имен

1.1 Непосредственное хранение имени в элементе.

1.2. Динамическое выделение памяти под имена по мере надобности.

1.3. Выделение отдельного одномерного массива для хранения имен.

1.4. Использование списка одномерных массивов, выделяемых по мере надобности.

2. Альтернативные варианты организации таблицы

2.1. Одномерный массив элементов.

2.2. Однонаправленный линейный список элементов.

2.3. Однонаправленный кольцевой список.

2.4. Однонаправленный линейный список одномерных массивов.

2.5. Однонаправленный кольцевой список одномерных массивов.

2.6. Хеш-массив с разрешением конфликтов через однонаправленный линейный список.

2.7. Хеш-массив с разрешением конфликтов через однонаправленный кольцевой список.

2.8. Хеш-массив с разрешением конфликтов через ближайший свободный.

2.9. Однонаправленный линейный список, формируемый на основе одномерного массива элементов.

2.10. Однонаправленный кольцевой список, формируемый на основе одномерного массива элементов.

2.11. Двоичное дерево.

2.12. Двоичное дерево, построенное на основе одномерного массива.

Список вариантов

Организация таблицы	**Организация элемента**			
	1	***2***	***3***	***4***
1	1	13	25	37
2	2	14	26	38
3	3	15	27	39
4	4	16	28	40
5	5	17	29	41
6	6	18	30	42
7	7	19	31	43
8	8	20	32	44
9	9	21	33	45
10	10	22	34	46
11	11	23	35	47
12	12	24	36	48

Контрольные вопросы

1. Назначение и области использования таблиц имен.

2. Почему используются различные методы организации таблиц имен?

3. Как зависят используемые методы организации таблиц имен от выбранного языка программирования?

4. Каким образом влияет на поиск элементов в таблице имен их упорядочивание?

5. Для всех ли методов построения таблиц имен можно использовать упорядоченное расположение элементов?

6. Каким образом упорядочивание влияет на включение новых элементов в таблицу имен?

7. Каким образом упорядочивание влияет на удаление отдельных элементов из таблицы имен?

8. Каким образом упорядочивание влияет на очистку таблицы имен?

9. Каким образом влияет структура таблицы имен на методы ее очистки.

10. Каким образом влияет структура таблицы имен на методы поиска, включения и удаления отдельных элементов?11. Основные методы организации элементов таблицы имен.

12. Методы организации таблицы имен, использующие структуру, построенную на основе одномерных массивов.

13. Методы организации таблиц имен на основе динамических структур данных.

14. Основные способы поиска элементов в таблицах имен.

15. В каких таблицах и как осуществляется разрешение конфликтов между элементами?

Лабораторная работа №2
Описание синтаксиса языка программирования

Цели и задачи: Изучение основ теории языков и формальных грамматик, метаязыков, методов описания пользовательского синтаксиса, синтаксиса грамматик. Разработка синтаксических диаграмм, используемых для построения транслятора с учебного языка.

Время: 4 часа

Описание языков программирования во многом опирается на теорию формальных языков. Эта теория является фундаментом для организации синтаксического анализа и перевода. Теория формальных грамматик используется для описания синтаксиса языков программирования посредством специально разработанных метаязыков. Необходимо знать основные метаязыки и уметь применять их при разработке трансляторов.

Порядок выполнения лабораторной работы

1. Ознакомиться с описанием лабораторной работы и необходимым теоретическим материалом.
2. Получить вариант задания у преподавателя. Используя номер варианта, получить текст задания.
3. Перевести формальное описание разрабатываемого языка программирования из РБНФ в диаграммы Вирта.
4. Написать ряд содержательных примеров программ, раскрывающих особенности использования конструкций данных разрабатываемого языка, отразив в этих примерах все его функциональные возможности.

Содержание отчета

1. Синтаксис полученного от преподавателя варианта языка выполненный с использованием РБНФ. Предоставляется в виде электронного документа. Может быть сформирован выборкой своего варианта из общего задания на лабораторную работу.
2. Пользовательское описание синтаксиса разрабатываемого языка, построенное с использованием диаграмм Вирта. Может быть представлен как в электронном, так и рукописном виде. Требуется аккуратное оформление в соответствии с описанием диаграмм Вирта, представленном в лекционном курсе.
3. Пять содержательных примеров программ для работы с различными типами данных и с использованием различных операторов. Должны быть представлены в виде отдельных текстовых файлов. Предлагаемые для тестирования задачи:
   3.1. Сортировка одним из известных методов.

3.2. Нахождение наибольшего общего делителя.

3.3. Поиск координат максимального и минимального элементов.

3.4. Нахождение среднего арифметического для действительных чисел.

3.5. Нахождение корня квадратного по итерационной формуле.

Пример выполнения

В качестве примера рассмотрим преобразование к диаграммам Вирта синтаксических правил демонстрационного языка программирования DPL, представленного в учебном пособии. Исходный синтаксис представлен ниже в расширенных формах Бэкуса-Наура (РБНФ).

Синтаксические правила, задающие элементарные конструкции:

```
$ 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".
$ идентификатор = ( буква | "_" ) { буква | цифра | "_" }.
$ число = целое | действительное.
$ целое =  двоичное | восьмеричное |
           десятичное | шестнадцатеричное.
$ двоичное = "{2}" {/ "0" | "1" /}.
$ восьмеричное = "{8}"{ "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7" }.
$ десятичное = ["{10}"] {/ цифра /}.
$ шестнадцатеричное  = "{16}" {/ цифра |"A"|"B"|"C"|"D"|"E"|"F"
                                 |"a"|"b"|"c"|"d"|"e"|"f" /}.
$ действительное = числовая_строка порядок
                   |числовая_строка "." [числовая_строка] [порядок]
                   |"." числовая_строка [порядок].
$ числовая_строка = {/ цифра /}.
$ порядок = ("E"|"e")["+"|"-"] числовая_строка.
$ пробельный_символ = {/ пробел | табуляция
                         |перевод_строки | комментарий /}.
$ комментарий = "/*" { символ } "*/".
$ строка = "”" { символ | "””"} "”".
$ ключевое_слово = abort | begin | case | end | float | goto | int
                   | loop | or | read | skip | space | tab | var | write |.
$ разделитель = "(" | ")" | "[" | "]" | "," | ";" | ":" | ":=" | "*"
                | "/" | "%" |"+" | "-" | "->" |"<" | "=" | ">"
                | "<=" | ">=" | "!=".
```

Синтаксические правила, задающие структуру программы:

```
$ программа = begin ( описание | оператор )
              { ";" ( описание | оператор ) } end.
$ описание = var идентификатор [ размер ]
              { "," идентификатор [ размер ] } ":" тип.
$ тип = int | float.
```

```
$ размер = целое.
$ оператор = метка непомеченный.
$ метка  = идентификатор ":".
$ непомеченный = присваивания | условный | цикла | пустой |
                 ошибки | ввода | вывода | перехода.
$ присваивания = переменная ":=" выражение |
                 переменная "," присваивания "," выражение.
$ переменная = идентификатор [ "[" выражение "]" ].
$ выражение = [ "-" ] операнд { операция [ "-" ] операнд }.
$ операнд = "(" выражение ")" | число | переменная.
$ операция = "*" | "/" | "%" | "+" | "-" | "<" | "=" | ">" | ">=" |
                 "<=" | "!=".
$ условный = case [ набор_охраняемых ] end.
$ цикла = loop [ набор_охраняемых ] end.
$ набор_охраняемых = охраняемые [ or охраняемые ].
$ охраняемые = выражение "->" оператор { ";" оператор }.
$ пустой = skip |.
$ прерывания = abort [строка].
$ ввода = read переменная { "," переменная }.
$ вывода =   write ( выражение | спецификатор | строка )
             { "," ( выражение | спецификатор | строка ) }.
$ перехода = goto идентификатор.
$ спецификатор = ( space | tab | skip ) [ выражение ].
```

Полученные в результате диаграммы Вирта, описывающие пользовательский синтаксис языка, приведены в приложении А.

Варианты заданий

В соответствии с заданием на лабораторную работу необходимо разработать пользовательский синтаксис лексического и синтаксического анализаторов для проектируемого языка, описать его с применением РБНФ и диаграмм Вирта. Выбор правил для проектируемого языка осуществляется на основе таблицы вариантов. Каждая строка таблицы соответствует своему варианту. Необходимо из описаний альтернатив выбрать правила, соответствующие номерам, расположенным в колонках. Следует также отметить, что не все пункты заданного варианта необходимы для выполнения данной лабораторной работы, так как таблица определяет язык программирования, транслятор с которого разрабатывается в течение нескольких лабораторных работ. В частности, не используется информация о типе сканера. Варианты заданий представлены в таблице 2.1.

Таблица 2.1.

Варианты заданий на лабораторную работу

Сканер							**Распознаватель**				
Вариант	**Тип сканера**	**Набор символов**	**Регистр символов**	**Метка**	**Целое**	Действительное	**Программа**	**Описание**	**Присваивание**	**Условный**	**Цикла**
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	2	1	1	1	1	2
3	1	1	1	1	1	1	1	1	1	1	4
4	1	1	1	1	1	2	1	1	1	1	5
5	1	1	1	1	2	1	1	1	1	2	1
6	1	1	1	1	2	2	1	1	1	2	2
7	1	1	1	1	2	1	1	1	1	2	4
8	1	1	1	1	2	2	1	1	1	2	5
9	1	1	1	1	3	1	1	1	1	4	1
10	1	1	1	1	3	2	1	1	1	4	2
11	1	1	1	1	3	1	1	1	1	4	4
12	1	1	1	1	3	2	1	1	1	4	5
13	1	1	1	2	1	1	1	1	2	1	1
14	1	1	1	2	1	2	1	1	2	1	2
15	1	1	1	2	1	1	1	1	2	1	4
16	1	1	1	2	1	2	1	1	2	1	5
17	1	1	1	2	2	1	1	1	2	2	1
18	1	1	1	2	2	2	1	1	2	2	2
19	1	1	1	2	2	1	1	1	2	2	4
20	1	1	1	2	2	2	1	1	2	2	5
21	1	1	1	2	3	1	1	1	2	4	1
22	1	1	1	2	3	2	1	1	2	4	2
23	1	1	1	2	3	1	1	1	2	4	4
24	1	1	1	2	3	2	1	1	2	4	5
25	1	1	2	1	1	1	1	1	3	1	1
26	1	1	2	1	1	2	1	1	3	1	2
27	1	1	2	1	1	1	1	1	3	1	4
28	1	1	2	1	1	2	1	1	3	1	5
29	1	1	2	1	2	1	1	1	3	2	1
30	1	1	2	1	2	2	1	1	3	2	2
31	1	1	2	1	2	1	1	1	3	2	4
32	1	1	2	1	2	2	1	1	3	2	5
33	1	1	2	1	3	1	1	1	3	4	1
34	1	1	2	1	3	2	1	1	3	4	2

35	1	1	2	1	3	1	1	1	3	4	4
36	1	1	2	1	3	2	1	1	3	4	5
37	1	1	2	2	1	1	1	2	1	1	1
38	1	1	2	2	1	2	1	2	1	1	2
39	1	1	2	2	1	1	1	2	1	1	4
40	1	1	2	2	1	2	1	2	1	1	5
41	1	1	2	2	2	1	1	2	1	2	1
42	1	1	2	2	2	2	1	2	1	2	2
43	1	1	2	2	2	1	1	2	1	2	4
44	1	1	2	2	2	2	1	2	1	2	5
45	1	1	2	2	3	1	1	3	1	4	1
46	1	1	2	2	3	2	1	2	1	4	2
47	1	1	2	2	3	1	1	2	1	4	4
48	1	1	2	2	3	2	1	2	1	4	5
49	1	2	1	1	1	1	1	2	2	1	1
50	1	2	1	1	1	2	1	2	2	1	2
51	1	2	1	1	1	1	1	2	2	1	4
52	1	2	1	1	1	2	1	2	2	1	5
53	1	2	1	1	2	1	1	2	2	2	1
54	1	2	1	1	2	2	1	2	2	2	2
55	1	2	1	1	2	1	1	2	2	2	4
56	1	2	1	1	2	2	1	2	2	2	5
57	1	2	1	1	3	1	1	2	2	4	1
58	1	2	1	1	3	2	1	2	2	4	2
59	1	2	1	1	3	1	1	2	2	4	4
60	1	2	1	1	3	2	1	2	2	4	5
61	1	2	1	2	1	1	1	2	3	1	1
62	1	2	1	2	1	2	1	2	3	1	2
63	1	2	1	2	1	1	1	2	3	1	4
64	1	2	1	2	1	2	1	2	3	1	5
65	1	2	1	2	2	1	1	2	3	2	1
66	1	2	1	2	2	2	1	2	3	2	2
67	1	2	1	2	2	1	1	2	3	2	4
68	1	2	1	2	2	2	1	2	3	2	5
69	1	2	1	2	3	1	1	2	3	4	1
70	1	2	1	2	3	2	1	2	3	4	2
71	1	2	1	2	3	1	1	2	3	4	4
72	1	2	1	2	3	2	1	2	3	4	5
73	2	2	2	1	1	1	2	1	1	1	1
74	2	2	2	1	1	2	2	1	1	1	3
75	2	2	2	1	1	1	2	1	1	1	4
76	2	2	2	1	1	2	2	1	1	1	6
77	2	2	2	1	2	1	2	1	1	3	1
78	2	2	2	1	2	2	2	1	1	3	3
79	2	2	2	1	2	1	2	1	1	3	4
80	2	2	2	1	2	2	2	1	1	3	6

81	2	2	2	1	3	1	2	1	1	4	1
82	2	2	2	1	3	2	2	1	1	4	3
83	2	2	2	1	3	1	2	1	1	4	4
84	2	2	2	1	3	2	2	1	1	4	6
85	2	2	2	2	1	1	2	1	2	1	1
86	2	2	2	2	1	2	2	1	2	1	3
87	2	2	2	2	1	1	2	1	2	1	4
88	2	2	2	2	1	2	2	1	2	1	6
89	2	2	2	2	2	1	2	1	2	3	1
90	2	2	2	2	2	2	2	1	2	3	3
91	2	2	2	2	2	1	2	1	2	3	4
92	2	2	2	2	2	2	2	1	2	3	6
93	2	2	2	2	3	1	2	1	2	4	1
94	2	2	2	2	3	2	2	1	2	4	3
95	2	2	2	2	3	1	2	1	2	4	4
96	2	2	2	2	3	2	2	1	2	4	6
97	2	3	1	1	1	1	2	1	3	1	1
98	2	3	1	1	1	2	2	1	3	1	3
99	2	3	1	1	1	1	2	1	3	1	4
100	2	3	1	1	1	2	2	1	3	1	6
101	2	3	1	1	2	1	2	1	3	3	1
102	2	3	1	1	2	2	2	1	3	3	3
103	2	3	1	1	2	1	2	1	3	3	4
104	2	3	1	1	2	2	2	1	3	3	6
105	2	3	1	1	3	1	2	1	3	4	1
106	2	3	1	1	3	2	2	1	2	4	3
107	2	3	1	1	3	1	2	1	3	4	4
108	2	3	1	1	3	2	2	1	3	4	6
109	2	3	1	2	1	1	2	2	1	1	1
110	2	3	1	2	1	2	2	2	1	1	3
111	2	3	1	2	1	1	2	2	1	1	4
112	2	3	1	2	1	2	2	2	1	1	6
113	2	3	1	2	2	1	2	2	1	3	1
114	2	3	1	2	2	2	2	2	1	3	3
115	2	3	1	2	2	1	2	2	1	3	4
116	2	3	1	2	2	2	2	2	1	3	6
117	2	3	1	2	3	1	2	2	1	4	1
118	2	3	1	2	3	2	2	2	1	4	3
119	2	3	1	2	3	1	2	2	1	4	4
120	2	3	1	2	3	2	2	2	1	4	6
121	2	3	2	1	1	1	2	2	2	1	1
122	2	3	2	1	1	2	2	2	2	1	3
123	2	3	2	1	1	1	2	2	2	1	4
124	2	3	2	1	1	2	2	2	2	1	6
125	2	3	2	1	2	1	2	2	2	3	1
126	2	3	2	1	2	2	2	2	2	3	3

127	2	3	2	1	2	1	2	2	2	3	4
128	2	3	2	1	2	2	2	2	2	3	6
129	2	3	2	1	3	1	2	2	2	4	1
130	2	3	2	1	3	2	2	2	2	4	3
131	2	3	2	1	3	1	2	2	2	4	4
132	2	3	2	1	3	2	2	2	2	4	6
133	2	3	2	2	1	1	2	2	3	1	1
134	2	3	2	2	1	2	2	2	3	1	3
135	2	3	2	2	1	1	2	2	3	1	4
136	2	3	2	2	1	2	2	2	3	1	6
137	2	3	2	2	2	1	2	2	3	3	1
138	2	3	2	2	2	2	2	2	3	3	3
139	2	3	2	2	2	1	2	2	3	3	4
140	2	3	2	2	2	2	2	2	3	3	6
141	2	3	2	2	3	1	2	2	3	4	1
142	2	3	2	2	3	2	2	2	3	4	3
143	2	3	2	2	3	1	2	2	3	4	4
144	2	3	2	2	3	2	2	2	3	4	6

Альтернативные варианты правил лексического анализатора

Варианты синтаксических правил, определяющих лексический анализатор, называемый также сканером, задаются со второй по седьмую колонки таблицы вариантов. Однако, наряду с заданием альтернатив, необходимо также знать синтаксис общих для всех вариантов конструкций, к которым из объектов сканера относится идентификатор. Его синтаксис для данной лабораторной работы задается следующим правилом:

$ идентификатор = буква { буква | 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".

Тип лексического анализатора (сканера). Задается во второй колонке таблицы. Существуют два основных метода разработки лексических анализаторов, которые будут рассмотрены в следующей лабораторной работе. На основании заданного метода необходимо будет преобразовать пользовательский синтаксис языка и разработать блок лексического анализа (сканера).

Вариант 1. Прямой лексический анализатор.

Вариант 2. Непрямой лексический анализатор.

Наборы ключевых слов и операций. Определяются из третьей колонки табл.1. Задается один из трех наборов обозначений ключевых слов, операций и разделителей. первый вариант определяет данные конструкции в стиле языка программирования Паскаль, второй - в стиле Си, третий - в основном использует набор специальных символов. Заданный набор во многом определяет множество терминальных символов разрабатываемого языка.

Альтернативные наборы ключевых слов и операций соответствующие трем вариантам представлены в таблице 2.

Таблица 2.

Альтернативные наборы ключевых слов и операций

Понятие	Вариант 1	Вариант 2	Вариант 3	Обозначение
Сложение	+	+	PLUS	ADD
Вычитание	-	-	MINUS	MIN
Умножение	*	*	MULT	MULT
Деление	/	/	DIV	DIV
Остаток по модулю	mod	%	MOD	MOD
Равно	=	==	EQ	EQ
Не равно	<>	!=	NE	NE
Меньше	<	<	LT	LT
Больше	>	>	GT	GT
Меньше или равно	<=	<=	LE	LE
Больше или равно	>=	>=	GE	GE
Присваивание	:=	=	LET	ASS
Начало комментария	{	/*	//	COMMENT
Конец комментария	}	*/	Код “\n”	
Начало составного	begin	{	DO	BST
Конец составного	end	}	END	EST
Разделитель операторов	;	;	;	EOP

Идентичность верхнего и нижнего регистров (прописных и строчных букв). Задается в колонке четыре. Существуют языки программирования, в которых ключевые слова, идентификаторы, метки и другие конструкции можно задавать с любым сочетанием прописных и строчных букв, так как их регистр игнорируется. В этом случае перед блоком лексического анализа стоит специальный модуль, преобразующий поступающие буквы к одному регистру (верхний или нижний регистр выбираются произвольно или с учетом соглашений, принятых в операционной системе при работе с внешними устройствами). Игнорирование регистра букв широко используется в языке программирования Паскаль. В других языках, например Си, верхний и нижний регистр различаются. Поэтому увеличивается набор основных символов языка и накладываются ограничения на вид букв в ключевых словах.

Вариант 1. Прописные и строчные буквы различаются.

Вариант 2. Прописные и строчные буквы не различаются.

Организация метки. Вид метки задается в пятой колонке таблицы вариантов. Она состоит из имени метки и двоеточия, являющегося ограничителем данной конструкции. Имя метки может быть буквенно-цифровой строкой или строкой цифр. Предполагается, что во втором случае имя метки преобразуется в натуральное число в допустимом диапазоне. Поэтому, для од-

ной и той же метки ее представление может отличаться по числу незначащих нулей. Например, метка "1995:" эквивалентна "00001995:".

Вариант 1. Метка - буквенно-цифровая строка:

```
$ метка = имя_метки ":".
$ имя_метки = буква { буква | цифра }.
```

Вариант 2. Метка - строка десятичных цифр:

```
$ метка = имя_метки ":".
$ имя_метки = {/ цифра /}.
```

Организация целых чисел. Задается в шестой колонке таблицы вариантов. Существуют три варианта отличающихся друг от друга способом представления двоичных, восьмеричных, десятичных и шестнадцатеричных чисел.

Вариант 1. В стиле, аналогичном языку программирования Си.

```
$ двоичное = "0" ("B" | "b") {/ "0" | "1" /}.
$ восьмеричное = "0" { "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7" }.
$ десятичное = {/ цифра /}.
$ шестнадцатеричное = "0" ("X"|"x") {/ цифра|
                        "A"|"B"|"C"|"D"|"E"|"F"|"a"|"b"|"c"|"d"|"e"|"f" /}.
```

Вариант 2. В стиле ассемблера.

```
$ двоичное =  {/ "0" | "1" /} ( "B" | "b" ).
$ восьмеричное = {/ "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7" /} ( "C" | "c" ).
$ десятичное = {/ цифра /} [ "D" | "d" ].
$ шестнадцатеричное = цифра {цифра |
      "A"|"B"|"C"|"D"|"E"|"F"|"a"|"b"|"c"|"d"|"e"|"f" } ( "H" | "h" ).
```

Вариант 3. В произвольном стиле.

```
$ двоичное = "2#" {/ "0" | "1" /}.
$ восьмеричное = "8#" {/ "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7" /}.
$ десятичное = ["10#"] {/ цифра /}.
$ шестнадцатеричное = "16#" {/ цифра |
                "A"|"B"|"C"|"D"|"E"|"F"|"a"|"b"|"c"|"d"|"e"|"f" /}.
```

Организация действительных чисел. Задается в седьмой колонке таблицы вариантов. Наряду с общими правилами, определяющими понятия числовой строки и порядка, существуют две альтернативы.

```
$ числовая_строка = {/ цифра /}.
$ порядок = ("E"|"e")["+"|"-"] числовая_строка.
```

Вариант 1. В стиле Си.

```
$ действительное = числовая_строка порядок |
                числовая_строка "." [числовая_строка] [порядок] |
                        "." числовая_строка [порядок].
```

Вариант 2. В стиле Паскаля.
```
$ действительное =
        числовая_строка "." числовая_строка [порядок] |
        числовая_строка порядок.
```

Альтернативные варианты правил лексического анализатора

Альтернативы, задаваемые с восьмой по двенадцатую колонки таблицы вариантов, позволяют выбрать правила, определяющие синтаксис операторов, описаний и программы. Набор полученных правил определяет в дальнейшем структуру синтаксического анализатора разрабатываемого языка программирования.

Организация программы. Определяется из колонки восемь, задающей один из вариантов. Первый вариант определяет программы как список описаний и операторов, разделяемых точкой с запятой. Конец текста программы определяется концом файла. При втором варианте программа состоит из двух областей: описаний и операторов. За лексемой, определяющей конец программы в файле может быть что угодно, так как осуществлять разбор дальше в соответствии с синтаксисом не имеет смысла. Наряду с программой данный пункт определяет выбор составного оператора, синтаксис которого выдержан в аналогичном стиле.

Вариант 1.
```
$ программа = {/ (описание|оператор) ";" /} конец_файла.
$ составной = BST {/ оператор ";" /} EST.
```

Вариант 2.
```
$ программа = [ var описание { ";" описание } ]
BST оператор { ";" оператор } EST.
$ составной = BST оператор { ";" оператор } EST.
```

Описания. Задаются в девятой колонке таблицы вариантов. Первый вариант описания по стилю близок к языку программирования Паскаль, а второй напоминает описания Си.

Вариант 1.
```
$ описание = идентификатор { "," идентификатор } ":" [ vector "[" целое "]" of ] тип.
```

Вариант 2.
```
$ описание = тип идентификатор [ "[" целое "]" ]
                { "," идентификатор [ "[" целое "]" ] }.
```

Правило, определяющее тип, является общим для обоих вариантов:
```
$ тип = int | real.
```

Синтаксис операторов. Является одинаковым. Различия наблюдаются на уровне отдельных операторов. Одинаковым для всех является также синтаксис таких операторов, как пустой, перехода, ввода и вывода. Следует также отметить, что пустой оператор - это отсутствие каких бы то ни было конструкций, а не точка которая в данном случае является компонентой метаязыка, завершающей правило. Кроме операторов здесь же приведен еще ряд конструкций языка, являющихся общими для всех вариантов.

```
$ оператор = [метка] непомеченный.
$ непомеченный = составной | присваивания | перехода
                 условный | цикла | пустой | ввода | вывода.
$ пустой = .
$ перехода = goto имя_метки.
$ ввода = read переменная { "," переменная }.
$ вывода = write ( выражение | спецификатор )
                 { "," ( выражение | спецификатор ) }.
$ переменная = идентификатор [ "[" индекс "]" ].
$ индекс = идентификатор | целое.
$ спецификатор = skip | space | tab.
```

Оператор присваивания. Задается в десятой колонке таблицы вариантов. При этом определяется синтаксис не только данного оператора, но и выражения, которое ему присваивается. Это выражение может использоваться и в других конструкциях языка, например в операторе вывода. Существует три альтернативы для оператора присваивания и выражений: инфиксная форма, постфиксная скобочная форма и польская префиксная форма.

Вариант 1. Инфиксная форма.

```
$ присваивание = переменная ASS выражение.
$ выражение = слагаемое { (EQ|NE|LT|GT|LE|GE) слагаемое }.
$ слагаемое = множитель { (ADD|MIN) множитель }.
$ множитель = унарное { (MUL|DIV|MOD) унарное }.
$ унарное = [MIN] терм.
$ терм = переменная | число | “(“ выражение ”)”.
$ число = целое | действительное.
```

Вариант 2. Префиксная форма.

```
$ присваивание = ASS переменная "," выражение.
$ выражение = операция операнд операнд | "(" MIN операнд ")" | операнд .
$ операция = MUL | DIV | MOD | ADD | MIN | EQ | NE | LT | GT | LE | GE.
$ операнд = выражение | переменная | целое | действительное.
```

Вариант 3. Постфиксная форма.

```
$ присваивание = "(" выражение переменная ASS ")".
$ выражение = "(" операнд операнд операция ")" | "(" операнд MIN ")" | операнд .
$ операция = MUL | DIV | MOD | ADD | MIN | EQ | NE | LT | GT | LE | GE.
$ операнд = выражение | переменная | целое | действительное.
```

Условный оператор. Выбирается из одиннадцатой колонки. Приведенные альтернативы подобраны по стилю программы и определяют обычный условный оператор или переключатель на несколько направлений.

Вариант 1.

```
$ условный = if выражение then непомеченный [ else непомеченный ].
```

Вариант 2.

```
$ условный = if выражение then {/ оператор ";" /} [ else {/ оператор ";" /} ] end.
```

Вариант 3.

```
условный = if выражение then оператор { ";" оператор }
                [ else оператор { ";" оператор } ] end.
```

Вариант 4.

```
условный = case выражение of целое ":" непомеченный
                { or целое ":" непомеченный }
                [ else непомеченный ].
```

Оператор цикла. Выбирается из двенадцатой колонки таблицы вариантов. Синтаксис альтернатив в вариантах подобран под стиль программы, как и в случае с условными операторами.

Вариант 1.

```
$ цикла = while выражение do непомеченный.
```

Вариант 2.

```
$ цикла = while выражение do {/ оператор ";"/ } end.
```

Вариант 3.

```
$ цикла = while выражение do оператор { ";" оператор } end.
```

Вариант 4.

```
$ цикла = loop непомеченный.
```

Вариант 5.

```
$ цикла = loop {/ оператор ";" /} end.
```

Вариант 6.

```
$ цикла = loop оператор { ";" оператор } end.
```

Контрольные вопросы

1. Назовите основные способы определения формальных языков и их отличия.
2. Дайте определение формальной грамматики.
3. Для чего нужны метаязыки?
4. Чем является формальный язык, порождаемый грамматикой?

5. Определите отношения вывода и назовите отличия, существующие между ними.

6.Для грамматики G3 приведите пример вывода терминальной цепочки, содержащей три знака умножения и два знака сложения.

7. Приведите пример цепочки для грамматики G3, содержащей пять операндов. Осуществите вывод этой цепочки из начального нетерминала.

8. Напишите выражения, удовлетворяющие условиям, приведенным в заданиях 6 и 7, полученные при этом за минимальное число шагов.

9. Напишите выражения, удовлетворяющие условиям, приведенным в заданиях 6 и 7, полученные при этом за максимальное число шагов.

10. Дайте определение распознавателя. Приведите его структуру.

11. Назовите известные Вам классы грамматик с ограничениями на правила. Дайте их определения.

12. Чем отличается язык, определяемый формальной грамматикой, от языка, определяемого распознавателем?

13. Назовите эквивалентные соотношения между определениями формальных языков с помощью распознавателей и грамматик, заданных иерархией Хомского.

14. Опишите с помощью диаграмм Вирта синтаксис языка программирования, заданного вариантом лабораторной работы. Если возникнут проблемы, то переходите к изучению следующей темы. После чего повторите этот шаг.

Лабораторная работа №3
Построение лексического анализатора

Цели и задачи

Изучение подходов, используемых при разработке лексических анализаторов. Преобразование пользовательских диаграмм, описывающих элементарных конструкции в диаграммы Вирта, предназначенные для реализации. Закрепление изученных методов лексического анализа путем программной реализации сканера в рамках лабораторной работы.

Время: 4 часа

Лексический анализ – первая фаза процесса трансляции, предназначенная для группировки символов входной цепочки в более крупные конструкции, называемые лексемами. Его использование облегчает разработку синтаксического анализатора. Лексические анализаторы обычно используются для распознавания элементарных конструкций и могут быть реализованы различными способами. В рамках лабораторных работ, в зависимости от варианта задания, предлагается построить прямой или непрямой лексический анализатор.

Порядок выполнения лабораторной работы

1. В соответствии с вариантом задания провести разработку диаграмм Вирта лексического анализатора, опираясь на ранее созданное пользовательское описание элементарных конструкций.
2. Используя полученные диаграммы, разработать требуемую программу.
3. Провести комплексное тестирование разработанной программы на специально созданных тестах. Убедиться в правильном сканировании программой ранее разработанных на учебном языке примеров.

Содержание отчета

1. Диаграммы Вирта, разработанные для создания лексического анализатора.
2. Исходные тексты разработанной программы.
3. Комплексные тесты, используемые для проверки правильной работы программы.
4. Протоколы тестирования программы лексического анализатора применительно к созданным тестовым примерам и комплексным тестам.

Пример выполнения

Ниже рассматриваются особенности построения лексических анализаторов для демонстрационного языка DPL, описанного в учебном пособии.

Разработка непрямого лексического анализатора

Диаграммы Вирта DPL, полученные для непрямого лексического анализатора, приведены в приложении Б. Они опираются на отдельную реализацию каждой лексемы и возвраты назад. Основная диаграмма реализует централизованный арбитраж за счет обхода этих правил в нужном порядке. Каркас лексического анализатора образован функцией ***nxl()***:

```
//-------------------------------------------------------------
// Функция, формирующая следующую лексему
// Вызывается синтаксическим анализатором
//-------------------------------------------------------------
void nxl(void) {
  do {
    i_lv = -1;
    lv[0] = '\0';

    // Фиксируем начальную позицию
    oldpoz=ftell(infil)-1;
    oldline=line; oldcolumn=column;

    // Процесс пошел
    if(ch == EOF) {lc = lexEof; return;}
    // Игнорируемую лексему не возвращаем
    if(isSkip(ch)) {nxch(); lc = lexSkip; continue; /*return;*/}
    if(IsIdOrKw())       {return;} unset();
    if(IsString()) {return;} unset();
    if(IsFloat1())       {return;} unset();
    if(IsFloat2())       {return;} unset();
    if(IsFloat3())       {return;} unset();
    if(IsFloat4())       {return;} unset();
    if(IsFloat5())       {return;} unset();
    if(IsBinary())       {return;} unset();
    if(IsOctal())        {return;} unset();
    if(IsHex())          {return;} unset();
    if(IsPrefDecimal())     {return;} unset();
    if(IsDecimal())      {return;} unset();
    // Игнорируемую лексему не возвращаем
    if(comment())      {continue; /*return;*/} unset();
    // Игнорируемую лексему не возвращаем
    if(isIgnore(ch)) {
       nxch(); lc = lexIgnore; continue; /*return;*/}
    if(ch=='/') {nxch(); lc = lexSlash;return;}
    if(ch == ';') {nxch(); lc = lexSemicolon; return;}
    if(ch == ',') {nxch(); lc = lexComma; return;}
    if(ch == ':') {
```

```
      nxch();
      if(ch == '=') {nxch(); lc = lexAssign; return;}
    } unset();
    if(ch==':') {nxch(); lc = lexColon; return;}
    if(ch == '(') {nxch(); lc = lexLftRndBr; return;}
    if(ch == ')') {nxch(); lc = lexRghRndBr; return;}
    if(ch == '[') {nxch(); lc = lexLftSqBr; return;}
    if(ch == ']') {nxch(); lc = lexRghSqBr; return;}
    if(ch == '*') {nxch(); lc = lexStar; return;}
    if(ch == '%') {nxch(); lc = lexPercent; return;}
    if(ch == '+') {nxch(); lc = lexPlus; return;}
    if(ch == '-') {
      nxch();
      if(ch == '>') {nxch(); lc = lexArrow; return;}
    } unset();
    if(ch=='-') {nxch(); lc=lexMinus; return;}
    if(ch == '=') {nxch(); lc = lexEQ; return;}
    if(ch == '!') {
      nxch();
      if(ch == '=') {nxch(); lc = lexNE; return;}
    } unset();
    if(ch == '>') {
      nxch();
      if(ch == '=') {nxch(); lc = lexGE; return;}
    } unset();
    if(ch=='>') {nxch(); lc=lexGT; return;}
    if(ch == '<') {
      nxch();
      if(ch == '=') {nxch(); lc = lexLE; return;}
    } unset();
    if(ch=='<') {nxch(); lc=lexLT; return;}
    lc = lexError; er(0); nxch();
  } while (lc == lexComment || lc == lexSkip
         || lc == lexIgnore);
}
```

Для выполнения отката, в начале обработки лексемы запоминаются текущие позиции в файле, строке и столбце. При неудаче они восстанавливаются и происходит анализ следующей диаграммы. Функция отката unset() реализована следующим образом.

```
// откат назад при неудачной попытке распознать лексему
static void unset() {
  fseek(infil, oldpoz, 0);
  nxch();
  i_lv=-1;
  lv[0]='\0';
  poz = oldpoz;
  line=oldline;
  column=oldcolumn;
```

```
}
```

С реализацией отдельных диаграмм, процедурой обработки ошибок и тестовыми процедурами можно ознакомиться по демонстрационным примерам, сопровождающим изучаемую дисциплину.

Разработка прямого лексического анализатора

При разработке прямого лексического анализатора создается единое правило, формировние которого начинается с размещения лексем, начинающихся с одинаковых начальных символов, по группам. Эти символы образуют набор альтернатив, анализируемых на первом шаге, то есть, из начального состояния объединенного конечного автомата. Далее, аналогичный анализ осуществляется в каждой из подгрупп. При этом, достаточно длинные правила можно оформлять как отдельные автоматы, что, в конце концов, приведет в созданию соответствующих процедур. Сформированный общий автомат, обеспечивающий прямой лексический анализ (представленный диаграммой Вирта), приведен в приложении В. Программная реализация функции, реализующей этот замысел, выглядит следующим образом.

```
//-------------------------------------------------------------
// Функция, формирующая следующую лексему
// Вызывается синтаксическим анализатором
//-------------------------------------------------------------
void nxl(void) {
  do {
    i_lv = -1;
    lv[0] = '\0';
    if(ch == EOF) {lc = lexEof;}
    else if(isSkip(ch)) {nxch(); lc = lexSkip;}
    else if(isLetter(ch) || ch == '_'){
      lv[++i_lv]=ch; nxch();IsContinueIdOrKw();
    }
    else if(isDigit(ch)) { IsNumber();}
    else if(isIgnore(ch)) {nxch(); lc = lexIgnore;}
    else if(ch == '/') {nxch();IsSlashComment();}
    else if(ch == '\"') {nxch(); string_const();}
    else if(ch == ';') {nxch(); lc = lexSemicolon;}
    else if(ch == ',') {nxch(); lc = lexComma;}
    else if(ch == ':') {
      nxch();
      if(ch == '=') {nxch(); lc = lexAssign;}
      else lc = lexColon;
    }
    else if(ch == '(') {nxch(); lc = lexLftRndBr;}
    else if(ch == ')') {nxch(); lc = lexRghRndBr;}
    else if(ch == '[') {nxch(); lc = lexLftSqBr;}
    else if(ch == ']') {nxch(); lc = lexRghSqBr;}
    else if(ch == '*') {nxch(); lc = lexStar;}
    else if(ch == '%') {nxch(); lc = lexPercent;}
```

```
    else if(ch == '+') {nxch(); lc = lexPlus;}
    else if(ch == '-') {
      nxch();
      if(ch == '>') {nxch(); lc = lexArrow;}
      else lc = lexMinus;
    }
    else if(ch == '=') {nxch(); lc = lexEQ;}
    else if(ch == '!') {
      nxch();
      if(ch == '=') {nxch(); lc = lexNE;}
      else {lc = lexError; er(1);}
    }
    else if(ch == '>') {
      nxch();
      if(ch == '=') {nxch(); lc = lexGE;}
      lc = lexGT;
    }
    else if(ch == '<') {
      nxch();
      if(ch == '=') {nxch(); lc = lexLE;}
      lc = lexLT;
    }
    else if(ch == '{') {nxch();IsPrefNumber();}
    else if(ch == '.') {lv[++i_lv]=ch; nxch();IsFloatNumber();}
    else {lc = lexError; er(0); nxch();}
  } while (lc == lexComment || lc == lexSkip
         || lc == lexIgnore);
}
```

Диаграммы Вирта, описывающие отдельные подавтоматы прямого лексического анализатора, представлены на после описания единого правила. Их программную реализацию тоже проще всего изучить на предлагаемых для загрузки исходных текстах

Контрольные вопросы

1. Для чего нужен лексический анализатор?
2. Что порождает лексический анализатор?
3. Можно ли обойтись без сканера?
4. Назначение транслитератора.
5. Какая связь между сканером и конечным автоматом?
6. Существует ли связь между конечным автоматом и диаграммами Вирта?
7. Существует ли связь между конечным автоматом и праволинейными грамматиками?
8. Существует ли связь между конечным автоматом и грамматиками с левой рекурсией?

9. Как преобразовать грамматику с правой рекурсией в итеративную диаграмму Вирта?

10. Как преобразовать грамматику с левой рекурсией в итеративную диаграмму Вирта?

11. Назовите основные методы лексического анализа.

12. Приведите обобщенную структуру непрямого лексического анализатора.

13. Достоинства и недостатки непрямого лексического анализатора.

14. Можно ли повысит производительность непрямого лексического анализатора?

15. Приведите обобщенную структуру прямого лексического анализатора.

16. Достоинства и недостатки прямого лексического анализатора.

17. Перечислите конструкции конкретного языка программирования, которые целесообразно распознать на фазе лексического анализа.

18. Подготовьте список конструкций Вашего учебного языка программирования, которые будут распознаваться на фазе лексического анализа.

19. Число встречается в нескольких правилах непрямого лексического анализатора. В какой последовательсти необходимо проверять числа языка DPL? Допустимы ли перестановки в этой последовательности?

20. В код какого сканера (прямого или непрямого) удобнее добавлять новые лексемы? Поясните свой ответ.

21. Какими сообщениями о лексических ошибках можно расширить массив сообщений об ошибках в учебном языке?

Лабораторная работа №4
Построение распознавателя

Цели и задачи:

Закрепление принципов синтаксического разбора. Изучение методов анализа и преобразования контекстно-свободных грамматик. Приведение заданной вариантом грамматики к КС(1) виду. Написание и отладка программы, реализующей функции распознавателя.

Время: 6 часов

Синтаксический разбор (распознавание) является основополагающим этапом синтаксического анализа. Именно при его выполнении осуществляется подтверждение того, что входная цепочка символов является программой, а отдельные подцепочки составляют синтаксически правильные программные объекты. Вслед за распознаванием отдельных подцепочек осуществляется анализ их семантической корректности на основе накопленной информации. Разбор служит каркасом для остальных блоков фазы синтаксического анализа. Именно на него «навешиваются» процессы, формирующие или дерево разбора, или объектную модель языка, или промежуточное представление. Диаграммы Вирта позволяют достаточно просто решать ряд практических проблем, связанных с построением распознавателей. Во многом это обуславливается тем, что решение переходит из разряда символических преобразований в область манипуляции с графами. Потому, в лабораторной работе сделан упор на практическое использование методов построения распознавателей с применением диаграмм Вирта.

Порядок выполнения лабораторной работы

1. В соответствии с вариантом задания провести переработку диаграмм Вирта, определяющих структуру программы. Ввести вместо обозначений нетерминалов, удобных для пользователя идентификаторы, используемые в разрабатываемой программе.
2. Провести анализ и преобразование синтаксических диаграмм к КС(1) виду. Расставить альтернативные точки, соответствующие состояниям динамически порождаемых конечных автоматов.
3. Разработать программу распознавателя. Провести ее комплексное тестирование и отладку.

Содержание отчета

1. Диаграммы Вирта, разработанные для создания распознавателя.
2. Исходные тексты разработанной программы.
3. Тесты, используемые для проверки правильной работы программы.

4. Протоколы тестирования программы распознавателя применительно к созданным тестовым примерам и комплексным тестам.

Пример выполнения

Ниже рассматриваются особенности построения распознавателя для демонстрационного языка DPL, описанного в учебном пособии [1].

На первоначальном этапе происходит избавление от пустых правил. Затем выделяются альтернативные правила, начинающиеся с одинаковых терминалов, и осуществляется их преобразование, приводящее к получению КС(1) грамматики. В ходе этого преобразования проводится доказательство принадлежности путем проверки альтернативных точек и пустых связей, ведущих к перетеканию альтернатив. Диаграммы Вирта, полученные в результате анализа и используемые в дальнейшем для разработки распознавателя, приведены в приложении Г.

Разработка программного кода, реализующего распознаватель осуществляется по простой методике, заключающейся практически в механическом отображении построенных диаграмм Вирта в функции используемого языка программирования. Нюансы такого преобразования исходной модели в реально работающий программный модуль определяются языком и избранной технологией программирования. Ниже рассматривается прямое отображение диаграмм Вирта в соответствующие конечные автоматы. В качестве примера приводятся отдельные диаграммы и функции написанные по ним.

На рис. 4.1 приведена диаграмма правила описывающего программу.

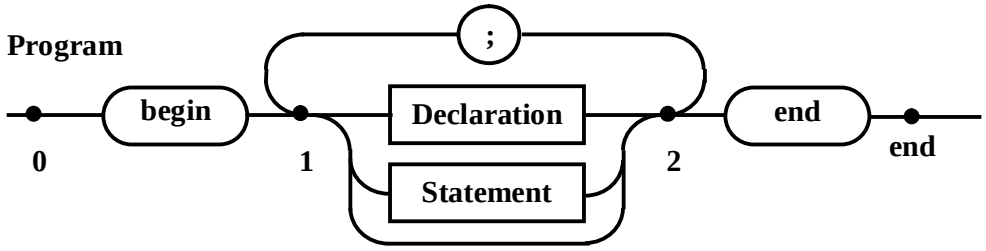


Рис. 4.1. Диаграмма Вирта, задающая правило ***Program***

Функция на языке C++, реализующая распознаватель для этого правила, выгляди следующим образом.

```
bool Program() {
//_0:
    if(lc==kwBegin) {nxl(); goto _1;}
    return false;
  _1:
    if(Declaration()) {goto _2;}
    if(Statement()) {goto _2;}
    if(lc==lexSemicolon) {nxl(); goto _1;}
    if(lc==kwEnd) {nxl(); goto _end;}
    er(7); return false;
```

```
  _2:
    if(lc==lexSemicolon) {nxl(); goto _1;}
    if(lc==kwEnd) {nxl(); goto _end;}
    er(7); return false;
  _end:
    er(7); return false;
  _end:
    return true;
}
```

Приведенный код показывает, что каждое правило преобразуется в отдельную функцию-автомат, возвращающую булевое значение, означающее допуск обрабатываемой подцепочки языка (***true***) или отказ (***false***). Сосотяния автоматов, задаваемые числами на диаграммах Вирта преобразуются в соответствующие метки. Конечное состояние преобразовано в метку ***end***.

После метки, задающей состояние, расположены условные операторы, каждый из которых задает один из переходов. Условие перехода определяется внутри условного оператора. Сам переход осуществляется оператором безусловного перехода размещаемым в его теле. Переход осуществляется по выполнению условия в котором проверяется совпадение с лексемой или нетерминалом.

Если переход из состояния должен происходить по лексеме, то перед ним осуществляется взятие следующей лексемы, так как предыдущая оказывается правильно обработанной. При переходе по нетерминалу, задаваемому вызовом функции, в которой реализована соответствующая диаграмма Вирта, взятие следующей лексемы не происходит. Это объясняется тем, что проверка лексемы, а следоваетльно и замена ее другой осуществляются внутри функций вложенных в вызывающую функцию только в том случае, когда происходит непосредственная проверка значения лексемы.

Следует отметить, что в общепринятой практике программирование с использованием оператора ***goto*** считается дурным тоном. Однако в данном случае построение кода осуществляется механически по диаграммам Вирта, которые служать основным документам. Именно эта особенность позволяет на наш взгляд более эффективно работать с синтаксисом, особенно при разработке новых языков программирования.

При нежелании использовантъ оператор безусловного перехода можно струкутрировать имеющиеся диаграммы Вирта или использовать в качестве базового метаязыка расширенные формы Бэкуса-Наура. Однако в этом случае затрудняется исследование свойств грамматики разрабатываемого языка.

С более подробным описанием процесса разработки распознавателя можно ознакомиться в учебном пособии [1].

Контрольные вопросы

1. Назначение синтаксического разбора.
2. Что является результатом синтаксического разбора?
3. Назовите критерии классификации синтаксического разбора.
4. Какие существуют методы разбора?
5. Связь методов разбора с выводом входной цепочки.
6. Особенности нисходящего разбора.
7. Особенности восходящего разбора.
8. Особенности комбинированного разбора.
9. Какие существуют последовательности разбора?
10. Связь между методами разбора и последовательностью разбора.
11. Особенности разбора с просмотром вперед.
12. Дополнительная классификация контекстно свободных грамматик.
13. Особенности разбора с возвратами.
14. Связь между сложностью языка и его трансляцией.
15. Зачем, при синтаксическом разборе нужны автоматы с магазинной памятью?
16. Как организован автомат с магазинной памятью?
17. Основные операции автомата с магазинной памятью.
18. Каким образом ограничения, накладываемые на грамматику, определяют реализацию автомата?
19. В чем заключается семантический разрыв между грамматиками и автоматами с магазинной памятью?
20. Дайте определение модели распознавателя на основе динамически порождаемых конечных автоматов.

Лабораторная работа №5
Использование таблицы имен

Цели и задачи:

Изучить применение семантического анализа при построении и использовании таблицы имен. Подключить таблицу имен, разработанную в лабораторной работе 1 к программе распознавателя созданного в лабораторной работе 4. Разобраться с особенностями синтаксически управляемого перевода.

Время: 4 часа

Синтаксический разбор определяет принадлежность входной цепочки заданному языку. Однако одного разбора недостаточно для генерации кода или его интерпретации. Необходим дополнительный набор действий, которые:

- определяют смысловую нагрузку объектов анализируемой программы;
- порождают из синтаксического представления программы (в виде цепочки символов) ее семантическую (смысловую) модель;
- анализируют корректность использования элементов построенной семантической модели перед построением выходного представления (кода объектной машины).

Эти действия определяются как семантический анализ, заключающийся в формировании семантической модели программы.

Создание и использование имен (идентификаторов) – неотъемлемая часть написания программы на любом языке программирования. Транслятор должен запоминать вводимые пользователем имена, определять конфликты, связанные с использованием объектов, обозначенных именами, запрещать повторное определение имен в пределах единой области их существования. При этом учитывается и то, что в некоторых языках программирования допустимо использование одного и того же имени в различных контекстах в рамках единой области действия.

Порядок выполнения лабораторной работы

1. В соответствии с вариантом задания провести добавление в программу модулей, обеспечивающих работу с таблицей имен.
2. Добавить в программу распознавателя код, определяющий взаимодействие с таблицей имен.
3. Провести тестирование программы, перебрав при этом различные корректные и некорректные ситуации.

4. Оттранслировать примеры, написанные при выполнении лабораторной работы 2. Проанализировать результаты работы транслятора.

Содержание отчета

1. Исходные тексты разработанной программы.
2. Тесты, используемые для проверки правильной работы программы.
3. Протоколы тестирования работы программы с таблицей имен.

Пример выполнения

Рассматривается построение таблицы имен для DPL [1]. Основные действия производимы в программе можно разделить на построение таблице имен при описании переменных и использование таблицы имен при появлении переменных в операторах.

Построение таблицы имен осуществляется в ходе синтаксического анализа при обработке описаний новых переменных.

1. Ищется имя в таблице имен, соответствующее текущей лексеме, определяющей идентификатор.

2. Если такое имя найдено, то выдается сообщение об ошибке: "Повторное определение имени", в противном случае имя заносится в таблицу и осуществляется формирование значений атрибутов очередного элемента таблицы.

После того, как, для вновь введенного имени, определяются значения всех атрибутов, задающих контекст, становится возможным его использование при дальнейшем анализе программы. Это использование происходит в те моменты, когда имя вновь встречается в тексте программы. В элементарном случае использование контекста имени достаточно просто.

1. Осуществляется поиск имени в таблице имен.

2. Если оно найдено, то сопоставляются имя, указанное в таблице и текущая его семантика.

   2.1. При отсутствии противоречий происходит дальнейший разбор и использование имени в контексте программы.

   2.2. Наличие противоречий ведет к формированию сообщений об ошибке.

3. Если же имя не найдено в таблице, то формируется сообщение об использовании неописанного имени.

Применение синтаксически управляемого перевода при построении таблицы имен базируется на внедрении в синтаксические диаграммы семантических вставок. Рассмотрим, каким образом этот прием используется при построении таблицы.

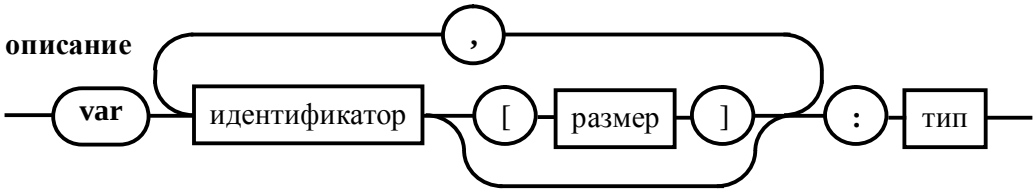
В демонстрационном языке программирования новые имена впервые вводятся в программу только в описаниях. Поэтому, соответствующую семантическую обработку необходимо добавить в правило, определяющее опи-

сание. Исходное правило представлено на рис. 5.1а. Его вид после добавления семантических элементов представлен на рис. 5.1б. Следует отметить, что появление семантических вставок меняет количество терминалов и нетерминалов в правиле, так как, даже после одного и того же символа возможны различные семантические вставки. Однако, организация связей остается неизменной. Поэтому не надо менять код, определяющий реализацию правил. На данном рисунке семантические вставки пронумерованы и связаны со следующими действиями.

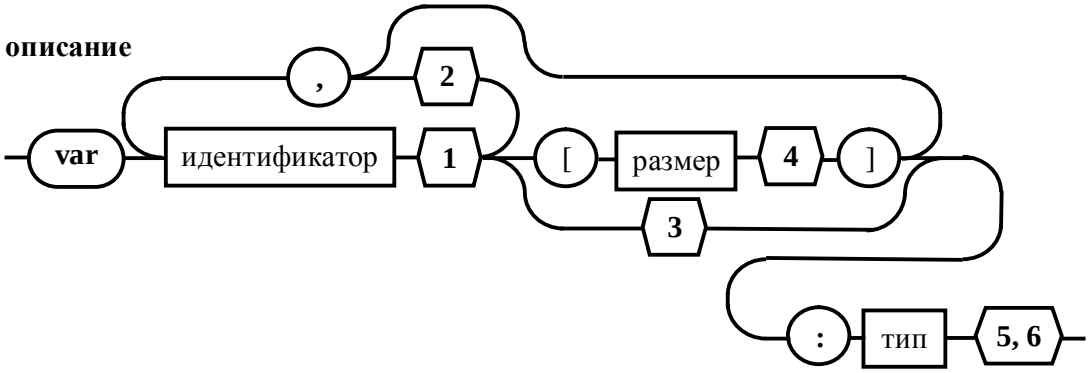
Вставка 1. Поиск в таблице имен.

1.1. Если имя отсутствует, то: оно включается в таблицу. Кроме этого, указатель на данный элемент в таблице фиксируется во временном списке, предназначенном для хранения всех имен данного описания. Это необходимо сделать для того, чтобы в дальнейшем доопределить семантику имен, так как структура правила не позволяет сразу определить тип переменной и занимаемый ею объем памяти.

1.2. Если же переменная с данным именем уже имеется в таблице, то фиксируется ошибка и выдается сообщение о повторном описании имени переменной. Дальнейший разбор не проводится или осуществляется нейтрализация ошибки.



а) Исходное правило, задающее описание переменных



б) Правило, задающее описание переменных,
с включенными семантическими вставками

Рис. 5.1. Использование синтаксически управляемого перевода
для построения таблицы имен при описании переменных

Вставка 2. Установка признака скаляра для текущей переменной списка. Параметр длины первоначально устанавливается в нулевое значение. То есть, предполагается, что описываемая переменная является скалярной.

Вставка 3. Установка признака скаляра для последней переменной списка. Параметр длины первоначально устанавливается в нулевое значение. Данное действие эквивалентно действию, описанному в пункте 3, но выполняется по другой условной ветви. Если бы объем вычислений в этих семантических элементах был большим, то можно было бы реализовать отдельную функцию, вызываемую в нужных точках.

Вставка 4. Установка длины вектора. При наличии описателя размерности сформированного целое число определяет длину вектора. Его значение, преобразованное сканером во внутреннее представление, заносится в параметр, определяющий длину векторной переменной.

Вставка 5. Фиксация типа и распределение памяти для описанных целочисленных переменных. После того, как определен целочисленный тип переменных, имеются все данные, чтобы завершить формирование параметров переменных. Для этого осуществляется перебор элементов временного списка, в ходе которого устанавливается целый тип и вычисляется размещение в памяти, зависящее не только от типа, но и размерности.

Вставка 6. Фиксация типа и распределение памяти для описанных действительных переменных. После того, как определен действительный тип переменных, имеются все данные, чтобы завершить формирование параметров переменных. Для этого осуществляется перебор элементов временного списка, в ходе которого устанавливается целый тип и вычисляется размещение в памяти, зависящее не только от типа, но и размерности.

Вставки 5, 6 выполняются альтернативно после анлиза ранее сформированных сведений о переменных.

Следует отметить, что даже для этого правила могут быть другие варианты обработки, обеспечивающие построение эквивалентной таблицы имен. Можно осуществить дополнительную оптимизацию, сокращающую код, использовать вспомогательные процедуры и функции и т.д.

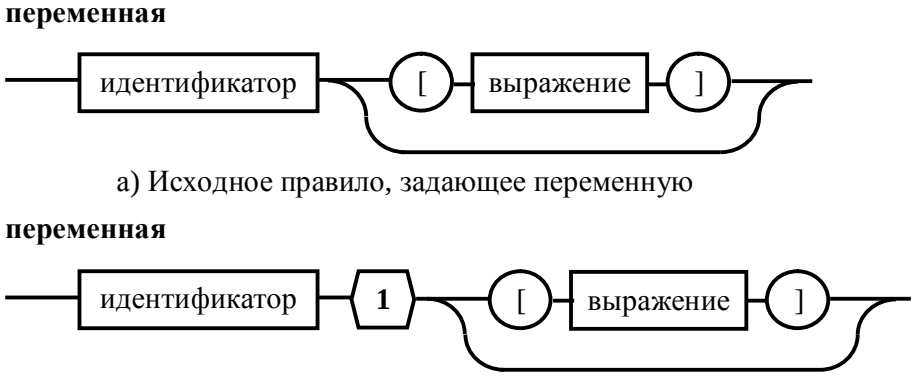
Ниже приводится исходный текст функции, осуществлющей проверку описания переменной, в которую встроен код по работе с элементами таблицы имен.

```
int decl() {
  // Временный список на заносимые переменные.
  // Реализован без дополнительных проверок на переполнение.
  struct node* varyable[256];
  int i = -1; // указатель на первый элемент массива
varyable
_0:
  if(lc==KWVAR) {nxl(); goto _1;}
  return 0; // Не описание
```

```
 _1:
   if(lc==ID) { // Идентификатор описываемой переменной
if(!find(lv)) {
  // При отсутствии описания переменной формируем новое
  varyable[++i] = incl(lv);
} else {
  er(19); return 0; // Ошибка: повторное определение имени
}
nxl(); goto _2;
   }
   // Ошибка: недопустима лексема внутри описания
   er(5); return 0;
 _2:
   if(lc==LSB) {nxl(); goto _3;}
   if(lc==CL) {
// Определяется скалярная переменная. Размер = 0
varyable[i]->val.appl.len = 0;
nxl(); goto _1;
   }
   if(lc==DVT) { // Определяется скалярная переменная. Размер
 = 0
     varyable[i]->val.appl.len = 0;
     nxl(); goto _6;
   }
   // Ошибка: недопустима лексема внутри описания
   er(5); return 0;
 _3:
   if(lc==INT) {
 // Фиксируется длина для  массива
 varyable[i]->val.appl.len = unum;
 nxl(); goto _4;
   }
   // Ошибка: недопустима лексема внутри описания
   er(5); return 0;
 _4:
   if(lc==RSB) {nxl(); goto _5;}
   // Ошибка: недопустима лексема внутри описания
   er(5); return 0;
 _5:
   if(lc==CL) {nxl(); goto _1;}
   if(lc==DVT) {nxl(); goto _6;}
   // Ошибка: недопустима лексема внутри описания
   er(5); return 0;
 _6:
   if(lc==KWINT) { // Все переменные целого типа
       // Размер целого равен два байта (задан явно)
int j; // Индекс для перебора всех переменных с данным типом
for(j = 0; j <= i; ++j) {
 varyable[j]->val.appl.typ = INTTYP;
```

```
varyable[j]->val.appl.addr = memAddr;
if(varyable[j]->val.appl.len == 0)
memAddr += 2;
else
memAddr += 2*varyable[j]->val.appl.len;
  }
nxl(); goto _end;
  }
  if(lc==KWFLOAT) { // Все переменные действительного типа
// Размер действительного равен четыре байта (задан явно)
int j; // индекс для перебора всех переменных с данным типом
for(j = 0; j <= i; ++j) {
varyable[j]->val.appl.typ = FLOATTYP;
varyable[j]->val.appl.addr = memAddr;
if(varyable[j]->val.appl.len == 0)
     memAddr += 4;
else
memAddr += 4*varyable[j]->val.appl.len;
  }
  nxl(); goto _end;
  }
  // Ошибка: недопустима лексема внутри описания
  er(5); return 0;
_end:
  return 1;
}
```

В демонстрационном языке программирования существует только одно правило, задающее использование имени в программе. Это правило, описывающее переменную (рис. 5.2а). Добавление семантического элемента слегка изменяет его вид (рис. 5.2б). Сами же действия дополнительной вставки 1 описываются следующим образом.



а) Исходное правило, задающее переменную

б) Правило, задающее описание переменных,
с включенными семантическими вставками

Рис. 11.15. Использование синтаксически управляемого перевода
для работы с таблицей имен при использовании переменных

Вставка 1. Анализ нахождения имени переменной в таблице. Если имя, совпадающее с анализируем идентификатором находится в таблице, то возвращается указатель на его назначение (или на весь элемент таблицы). Если же имя не найдено, то формируется сообщение о появлении в программе имени с неизвестным контекстом, которое предварительно не определено, и производится обработка ошибки.

Необходимо обратить внимание на отсутствие семантических вставок, анализирующих корректность использования переменных. В частности, нет проверки на использование скалярной переменной с квадратными скобками или векторной переменной без квадратных скобок. В принципе, такие проверки легко можно было бы добавить уже сейчас, но будем считать, что семантика языка в этом вопросе пока до конца не определена. О есть, в дальнейшем мы вполне можем допустить использование только имени векторной переменной в векторных операциях (как это сделано в PL/1) или выбор бита из скалярной переменной по указанному индексу. Кроме того, что данные действия определяются семантикой языка, их реализацию можно рассматривать тогда, когда идет вопрос о генерации кода и осуществляется комплексная проверка семантики. Это еще одна причина отложить решение данного вопроса.

Ниже приводится исходный текст функции, осуществляющей работу с таблицей имен при разборе правила, описывающего переменную.

```
 int varying() {
 _0:
   if(lc==ID) {
// имя переменной д.б. в таблице имен
if(find(lv)) {
  // Использование элемента таблицы имен.
  // Должно добавиться при формировании объектной модели
} else {
er(20); return 0; // Имя предварительно не определено
}
nxl(); goto _1;
   }
   // Это не переменная. Выход на анализ другой версии.
   return 0;
 _1:
   if(lc==LSB) {nxl(); goto _2;}
   goto _end;
 _2:
   if(expr()) {goto _3;}
   // Ошибка: в этом месте допустимо только выражение
   er(15); return 0;
 _3:
   if(lc==RSB) {nxl(); goto _end;}
```

```
  // Ошибка: допустима только закрывающаяся квадратная скоб-
ка.
  er(15); return 0;
_end:
  return 1; // Правило успешно распознано.
}
```

Контрольные вопросы

1. Как используется синтаксически управляемый перевод при построении таблицы имен?

2. Как используется синтаксически управляемый перевод при построении таблицы имен?

3. Какие семантические действия должны выполняться, на ваш взгляд при объявлении в программе метки?

4. Какие семантические действия должны выполняться, на ваш взгляд при появлении в программе оператора безусловного перехода на метку?

5. Какие программные объекты используются при работе с таблицей имен.

6. Каким образом реализованы в Вашей программе объекты, представляющие таблицу имен?

7. Каким образом реализованы в Вашей программе объекты, представляющие назначения имен?

Лабораторная работа №6
Семантический анализ и генерация промежуточного представления

Цели и задачи:

Изучить применение семантического анализа при генерации промежуточного представления. Включить в программу дополнительные модули, обеспечивающие работу с операндами и командами. Добавить в распознаватель код, осуществляющий генерацию промежуточного представления.

Время: 8 часов

Промежуточное представление (ПП) - это программа в кодах виртуальной машины (ВМ) на языке, эквивалентном или близком к исходному. Однако, это уже машинная программа, и из нее убран текстовый (языковый) контекст. При разработке команд ВМ необходимо учитывать специфику исходного языка. Перед тем, как начать формирование промежуточного представления, необходимо ответить на следующие вопросы: каким образом оно должно выглядеть и каким должен быть состав команд ВМ. Состав команд определяется семантикой языка программирования. При этом возможны два пути.

Порядок выполнения лабораторной работы

1. В соответствии с вариантом задания провести добавление в программу модулей, обеспечивающих порождение операндов и цепочки команд промежуточного представления.
2. Добавить в программу, полученную после выполнения лабораторной работы 5, код, определяющий генерацию промежуточного представления.
3. Провести тестирование программы.
4. Оттранслировать примеры, написанные при выполнении лабораторной работы 2. Проанализировать результаты работы транслятора.

Содержание отчета

1. Исходные тексты разработанной программы.
2. Тесты, используемые для проверки правильной работы программы.
3. Протоколы тестирования работы программы с таблицей имен.

Пример выполнения

Для разработки промежуточного представления необходимо проанализировать особенности операций и операторов демонстрационного языка, преобразуемых в команды промежуточного представления. Можно выделить в нем отдельные группы.

1. Команды, используемые в выражениях. Это операции различного вида, применяемые в арифметических выражениях и отношениях. Они делятся на:

1.1. унарные. (это только унарный минус);

1.2. бинарные (плюс, бинарный минус, умножить и т.д.).

2. Команды для работы с переменной. Данная категория команд определяет доступ к внутренней структуре переменных. Выделяется только одна команда данной группы:

2.1. выделить (взять) элемент из вектора (квадратные скобки "[…]").

3. Команды в операторах. Эти команды обеспечивают организацию вычислений на уровне различных операторов исходного языка. К ним относятся:

3.1. команда присваивания, обеспечивающая присваивание значение некоторой переменной, что эквивалентно пересылке данных из одной ячейки в другую;

3.2. команда условного перехода по истинному условию на указанную в нем метку, что позволяет выполнять ветвления и циклы;

3.3. команда безусловного перехода на указанную метку, которая может дополнять условный переход, чтобы обеспечить требуемые ветвления;

3.4. команда ввода значения переменной (в отличие от оператора, позволяет вводить только одно значение, несколько значений вводятся последовательностью команд);

3.5. команда вывода значения переменной (выводит только одно значение);

3.6. команды вывода форматирующих символов (каждая команда выводит только один заданный символ);

3.7. команда прерывания;

3.8. пустая команда, которая ничего не выполняет;

3.9. команда завершения программы, передающая управление системе или останавливающая выполняемую программу;

3.10. команда- метка предназначенная для установке позиции метки в поток команд.

Анализ рассмотренных команд показывает, что большинство из них имеет три операнда (чаще всего, два аргумента и один результат), а отношение управлении можно совместить с отношением расположения, так как в основном используется переход на следующую команду. В тех случаяз, когда требуется переход, не совпадающий с отношением управления можно использовать поле операнда, в котором указывается специальный операнд-метка.

Формат полученной обобщенной команды промежуточного представлен на рис. 6.1.

Рис. 6.1. Формат обобщенной команды промежуточного представления

Использование подобных команд упрощает ее представление в виде структуры данных, написанной на используемом языке программирования. Она имеет следующий вид:

```
struct INSTRUCTION {
     opcType   opc;
     // отношение расположения/управления
     struct INSTRUCTION  *next;
     // Операндная часть инструкции
     struct OPERAND *arg1;
     struct OPERAND *arg2;
     struct OPERAND *rez;
     // Место встречи в тексте программы
     int       line;
     int       column;
};
```

Дополнительные поля **line** и **column** используются для сохранения местоположения данной команды в исходном тексте программы, что может оказаться удобным при кодогенерации и отладке.

Поле opc хранит код операции, который может быть задан в виде перечислимого тип, например, следующим образом:

```
// Разновидности команд промежуточного представления.
typedef enum {
     addOpc,        assOpc,        divOpc,        eqOpc,
     emptyOpc,      exitOpc,       geOpc,         gotoOpc,
     gtOpc,         ifOpc,         inOpc,         indexOpc,
     labelOpc,      leOpc,         ltOpc,         minOpc,
     modOpc,        multOpc,       neOpc,         outOpc,
     skipOutOpc,    spaceOutOpc,   subOpc,        tabOutOpc
} opcType;
```

Для окончатльного определения списка команд необходимо сформировать структуру операнда. Операндами команд в промежуточном языке являются:

- переменные заданные в описания и хранящиеся в памяти по выделенным адресам;
- константы, обрабатываемые размещаемые в памяти транслятором;
- временные переменны, формируемые транслятором как промежуточные элементы для связи команд по общим данным;

- операнды-метки, применяемые в промежуточном представлении для обеспечения гибкости в обработки условных и безусловных переходов.

Обобщенное представление структуры операнда промежуточного представления демонстрационного языка программирования приведено на рис. 6.2.

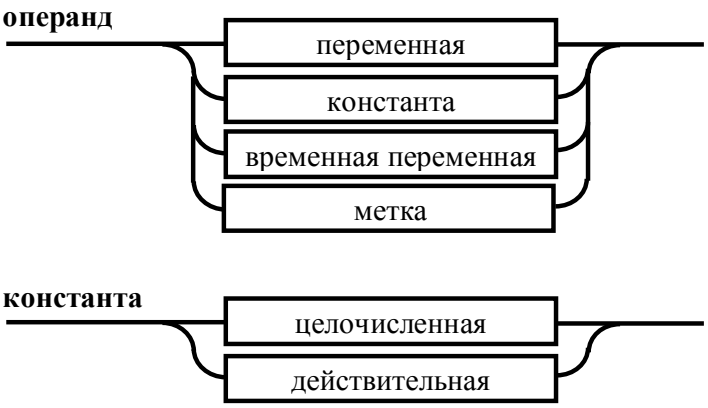


Рис. 6.2. Описание обощенной структуры операнда промежуточного представления демонстрационного языка программирования

Наличие целочисленных и действительных данных соответствующим образом отражается на структуре константы. Представленные описании могут быть реализованы на языке программирования следующим образом:

```
// Структуры данных, обеспечивающие
// хранение и обработку операндов
// Организация константы
struct CONST {
     scalType typ; // тип константы
     union {
          int  unum;     // целая константа
          float     fnum;     // действительная константа
     } val;    // значение константы
};

// Разновидности операнда
typedef enum {
     nameVarOpd, // именованная переменная из таблицы имен
     tmpVarOpd,  // промежуточная переменная в выражениях
     constOpd,   // константный операнд
     labelOpd    // операнд - метка
} opdType;

// Организация операнда
struct OPERAND {
     opdType typ;   // тип операнда
     union {
          // ссылка на именованную или промежуточную переменную:
```

```
        struct application *var;
        struct CONST *cons;        // ссылка на константу
        struct INSTRUCTION *label; // ссылка на метку
    } val;    // значение операнда
    // Дополнительные параметры, обеспечивающие тестирование
    int       ident;      // уникальный идентификатор операнда
    struct OPERAND *next; // для организации списка операндов
};
```

Набор функций, обеспечивающих построение списка команд, отличается от набора функций, применяемых при построении и использовании таблицы имен. Он определяется решаемой задачей и сводится к поддержке следующих действий:

1) добавление команды к уже существующему списку команд;

2) слияние двух отдельных списков команд в один в заданной последовательности;

3) добавление отельной команды в голову списка.

Генерация структур, представляющих константные операнды

Генерация структур данных, представляющих операнды, осуществляется в тех функциях распознавателя, которые реализуют работу с операндами. В частности порождение констант осуществляется в правиле, задающем число. В соответствии с синтаксически управляемым переводом оно модифицируется за счет добавления семантических вставок (рис. 6.3).

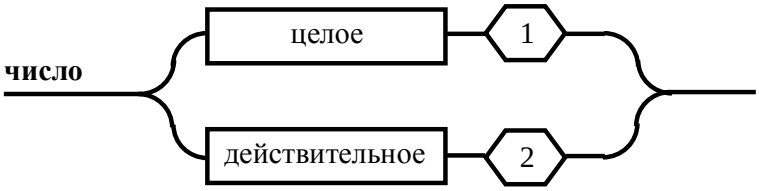


Рис. 6.3. Добавление в число семантических вставок, обеспечивающих генерацию константного операнда

Введенные в правило семантические вставки выполняют следующие действия:

Встака 1. Генерируется константа целочисленного типа путем заполнения полей структуры ***CONST*** соответствующим признаком целочисленной константы ***INTTYP*** и значением этой константы.

Втавка 2. Генерируется константа действительного типа путем заполнения полей структуры ***CONST*** соответствующим признаком целочисленной константы ***FLOATTYP*** и значением этой константы.

Этот нетерминал число входит в правило, описывающее терм. В этом правиле полученная константа включается в операнд и подсоединяется к дополнительной оболчке, содержащей список команд. Формирование списка команд протекает и по другим альтернативам терма: переменной и выражению. Однако для числа характерно то, что список команд является пустым,

так как формируется только операнд при отсутствии константы. Семантическая вставка (рис. 6.4) описывающая построение списка команд, реализует следующее действие:

Вставка 1. При распознавании числа формируется константный операнд. Создается также пустой список команд, указатель на операнд которого ссылается на сформированный константный операнд. Полученный список команд и является возвращаемым параметром.

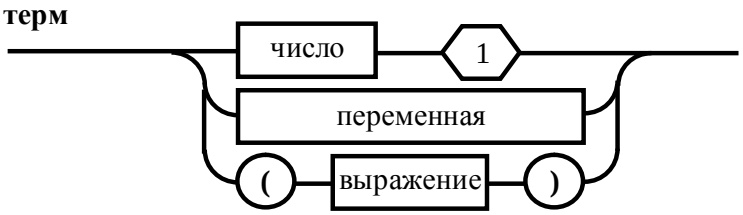


Рис. 6.4. Семантическая вставка в терм, порождающая пустой список команд с присоединенным константным операндом

Следует отметить, что список команд имеет следующую структуру:

```
// Список команд
struct INSTR_LIST {
     INSTRUCTION    *firstInstr;   // Первая команда в списке.
     INSTRUCTION    *lastInstr;    // Последняя команда в списке.
     OPERAND   *lastOpd; // Последний полученный операнд.
};
```

Сохранение операнда, полученного в ходе генерации списка команд позволяет передавать последний полученных результат для вновь генерируемой команды. Таким образом может проходить формирование отношений между командой и операндами, полученными в различных подвыражениях. Более подробно этот процесс рассматривается при генерации кода для выражений.

Генерация промежуточного представления для переменной

Порождение операнда на основе переменной, размещаемой в таблице имен, осуществлятся при распознавании переменной в соответствующем правиле. Для этого в него добавляются две семантические вставки (рис. 6.5).

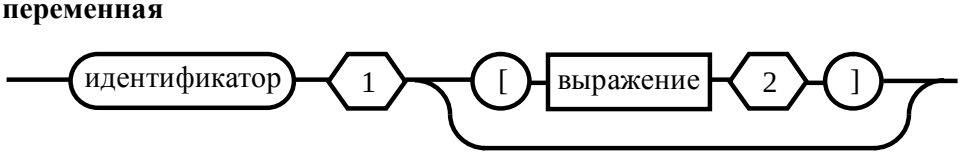


Рис. 6.5. Включение семантических вставок в правило, описывающее переменную

Они выполняют следующие действия:

Вставка1. Расширяет ранее выполняемые операции по поиску значения идентификатора в таблице имен дополнительными действиями, связанными с формированием структуры операнда-переменной. Создается также

пустой список команд, в котором операнд-переменная фиксируется в качестве последнего операнда.

Вставка 2. При наличии индексного выражения генерируется команда выделения значения из вектора по индексу переменной. Первым операндом этой команды является операнд, сформированный в семантической вставке 1, а второй операнд - это последний операнд, зафиксированный в списке команд, возвращенном из выражения. После этого осуществляется конкатенация списков команд содержащих идентификатор и выражение. К нему добавляется команда вычисления индекса. Последним операндом вновь сформированного списка команд становится результат команды взятия индекса.

Вторая вставка демонстрирует использование операндов, включенных в объединяемые списки команд

Генерация промежуточного представления для выражений

В правилах, определяющих выражение происходит иерархическое порождение и слияние списков команд. Наиболее простая генерация происходит в множетеле, так как там возможно только добавление унарного минуса (рис. 6.6).

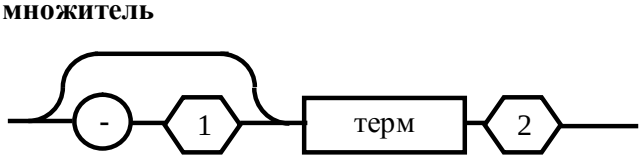


Рис. 6.6. Включение семантических вставок в правило, описывающее множитель

Семантические вставки правила выполняют следующие действия:

Вставка 1. Формируется команда, вычисляющая унарный минус. Ее результатом является промежуточная переменная, а аргумент пока не определен.

Вставка 2. Если команда, вычисляющая унарный минус, сформировалась, то она добавляется возвращаемому из терма списку команд. Последний операнд возвращенного списка становится аргументом унарного минуса. Результат унарного минуса становится последним операндом вновь сформированного списка.

Генерация промежуточного представления для правил описывающих вычисления посредством бинарных операций, происхоит по одинаковой схеме. Ниже этот процесс рассматривается на примере правила, определяющего слагаемого и содержащего операции умножения, деления и вычисления остатка (рис. 6.7).

Правило содержит семантические вставки для каждой операции, что позволяет сгенерировать нужные команды обработки данных.

Вставка 1. Запоминается список команд и его последний операнд, возвращенные из множителя.

Вставка 2. Генерируется команда умножения. Возвращенный из первого множителя последний операнд становится ее первым операндом.

Вставка 3. Генерируется команда деления. Возвращенный из первого множителя последний операнд становится ее первым операндом.

Вставка 4. Генерируется команда вычисления остатка. Возвращенный из первого множителя последний операнд становится ее первым операндом.

Вставка 5. Последний операнд, возвращенный из второго множителя, становится вторым операндом команды, сформированной в пунктах 2, 3, 4. Осуществляется конкатенация списков команд, определяющих первый и второй множители. К полученному списку добавляется команда, сформированная в пунктах 2, 3, 4. Ее результат, являющийся промежуточной переменной, становится последним операндом сформированного списка.

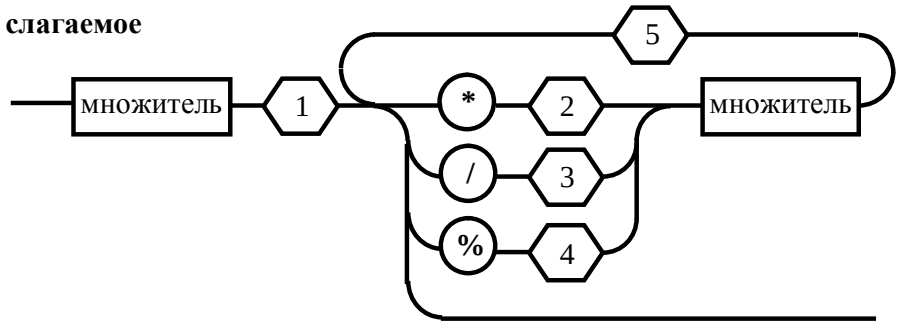


Рис. 6.7. Включение семантических вставок в правило, описывающее слагаемое

Аналогичным образом осуществляется генерация промежуточного представления и для операторов демонстрационного языка программирования. Более подробно с этим можно ознакомиться по исходным текстам программы реализующей транслятор с DPL, которые прилагаются к учебному пособию.

Контрольные вопросы

1. Назовите способы задания семантического представления.
2. В чем заключается особенность промежуточного представления программы?
3. Как формируются команды виртуальной машины?
4. Какие отношения существуют между командами?
5. Что задает операнд промежуточного представления?
6. Опишите семантические вставки для различных синтаксических правил.

Лабораторная работа №7
Генерация кода объектной машины

Цели и задачи:

Используя в качестве исходного промежуточное представление, построенного в ходе трансляции исходной программы, перевести его в программу на одном из языков программирования высокого уровня.

Время: 8 часов

Генерация кода является одной из наиболее сложных задач, решаемых при разработке трансляторов. Особенно много проблем возникают при генерации кода для низкоуровневых систем команд, которые характерны для реальных архитектур вычислительных систем. Вместе с тем, следует отметить, что для понимания основных принципов этого процесса, можно осуществлять генерацию в язык высокого уровня. Именно такой подход используется в рамках данной дисциплины.

Порядок выполнения лабораторной работы

1. В соответствии с вариантом задания провести добавление в программу модулей, обеспечивающих перевод промежуточного представления в требуемый код на языке высокого уровня.
2. Оформить создаваемый код в виде отдельного модуля, который содержит функцию, читающую промежуточное представление и преобразующую его в нужный вариант кода объектной машины.
3. Разработать программу, обеспечивающую решения поставленной задачи
4. Сформировать технологическую цепочку, состоящую из разработанного компилятора и компилятора с языка, в который осуществляется трансляция. Осуществить многоэтапный процесс, завершающийся выполнением программы, написанной на исходном языке и преобразованной в ходе многоэтапной трансляции в исполняемый код.
5. Осуществить тестовый прогон программ, написанных на учебном языке и проанализировать полученные результаты.

Содержание отчета

1. Исходные тексты разработанной программы.
2. Тесты, используемые для проверки правильной работы программы.
3. Протоколы тестирования работы программы с таблицей имен.

Выполнение

Для решения поставленной задачи необходимо рассмотреть ряд особенностей генерации высокоуровневого представления для ранее выделенных команд промежуточного представления. В качестве выходного языка выберем C++.

Существует ряд проблем, которые необходимо решить при генерации высокоуровневого выходного представления.

1. Необходимо выбрать, какие имена будут использоваться для переменных исходного языка в выходном представлении. В частности, возможны следующие варианты: полное совпадение имен, генерация собственных имен, комбинированный вариант.

2. Нужно выбрать метод для представления промежуточных (временных) переменных, которые получаются при обработке выражений. Здесь также возможны несколько вариантов:

- генерация специальных имен с добавлением номера, определяемого номером операнда;
- использование массивов для хранения временных переменных, которых номером служит индекс элемента массива.
- можно не использовать промежуточные переменные, а порождать на выходе скобочные выражения, непосредственно вложенные друг в друга в соответствии с порядком выполнения операций.

3. Определение соответствия между командами промежуточного представления и генерируемыми по ним операциями и операторами высокоуровневого языка.

Предполагается, что ответы на эти вопросы каждый из студентов находит самостоятельно, реализуя ту схему, которая кажется ему наиболее целесообразной. При защите лабораторной работы он должен мотивировать принятые решения.

Контрольные вопросы

1. В чем отличие генерации кода в высокоуровневый язык от генерации в язык машинного уровня?

2. Покажите, какие операторы высокоуровневого языка будут соответствовать различным командам промежуточного представления.

3. Какие достоинства и недостатки именования переменных высокоуровневого выходного представления именами, используемыми в исходном тексте программы.

3. Какие достоинства и недостатки именования переменных высокоуровневого выходного представления именами, не связанными с теми, что используются в исходном тексте программы.

4. Каковы возможны стратегии формирования имен промежуточных (временных) переменных.

Литература

1. Легалов А.И., Швец Д.А., Легалов И.А. Формальные языки и трансляторы. Учебное пособие. – Красноярск. – 2008 (в процессе разработки и рецензирования).

2. Карпов, Ю.Г. Теория и технология программирования. Основы построения трансляторов. / Ю.Г. Карпов. – СПб.: БХВ-Петербург. – 2006. – 272 с.

2. Свердлов, С.З. Языки программирования и методы трансляции. / С.З. Свердлов. – СПб.: Питер. – 2007. – 638 с.

3. Молчанов, А.Ю. Системное программное обеспечение. / А.Ю. Молчанов. – СПб.: Питер. – 2003. – 396 с.

4. Ахо А. В, Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии, инструменты.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 768 с.

5. Легалов А.И., Сиротинина Н.Ю. Организация таблиц имен: методические указания по лабораторной работе для студентов специальностей 220100, 220400, 220600. КГТУ, Красноярск, 1996.

6. Вирт Н. Алгоритмы + структуры данных = программы. - М.: Мир, 1985.

7. Льюис Ф., Розенкранц Д., Стирнз Р. Теоретические основы проектирования компиляторов. - М.: Мир, 1979.

8. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. - М.: Мир, 1978.

9. Дейкстра Э. Дисциплина программирования. М.: Мир, 1978.

10. Грис Д. Наука программирования. М.: Мир, 1984.

11. Вирт Н. Алгоритмы и структуры данных. М.: Мир, 1989. 360 с.

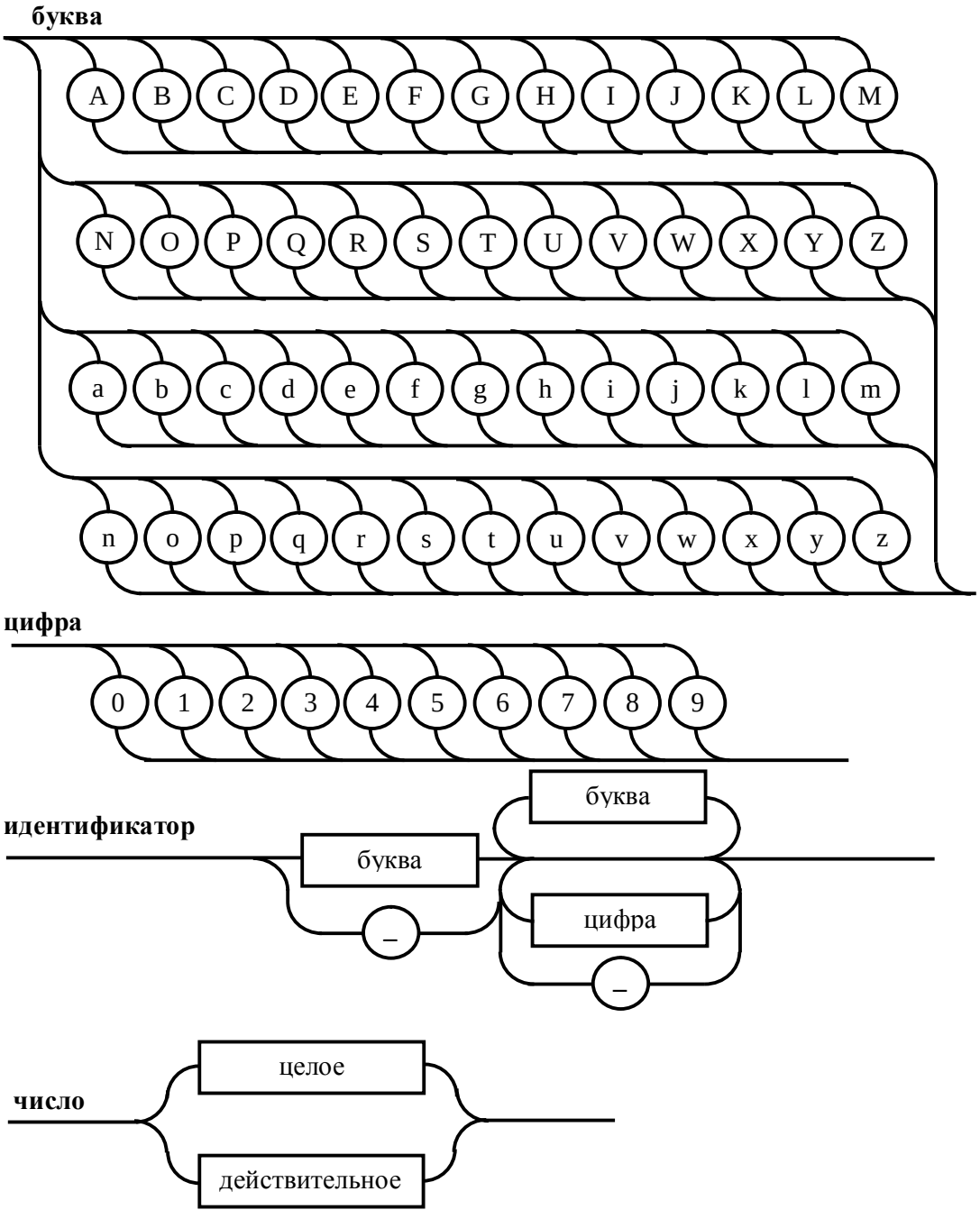
12. Кнут Д. Искусство программирования для ЭВМ: В 5 т.. Т. 3. Сортировка и поиск. М.: Мир, 1978. 846 с.

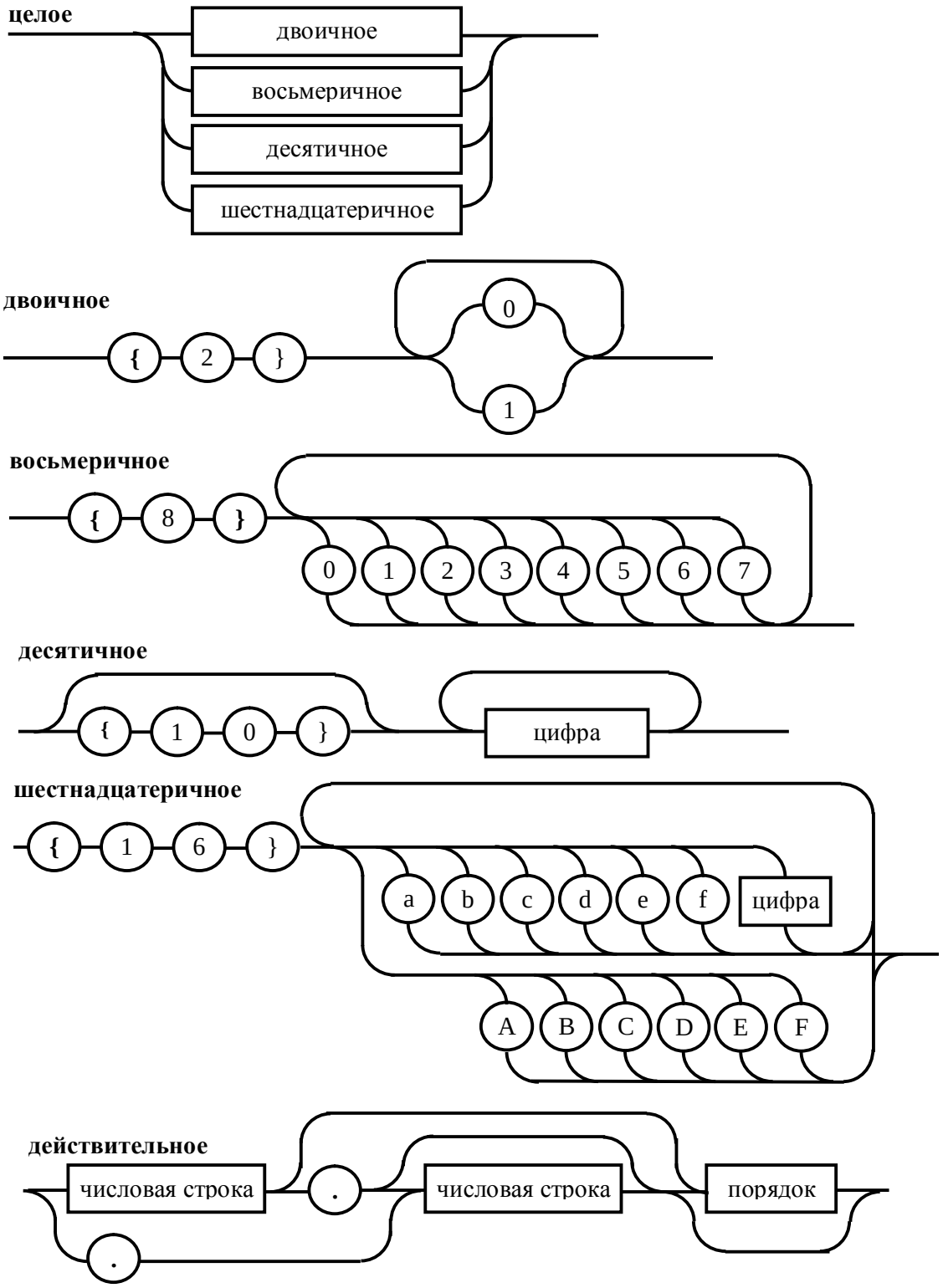
12. Сибуя М., Ямомото Т. Алгоритмы обработки данных. М.: Мир, 1986. 218 с.

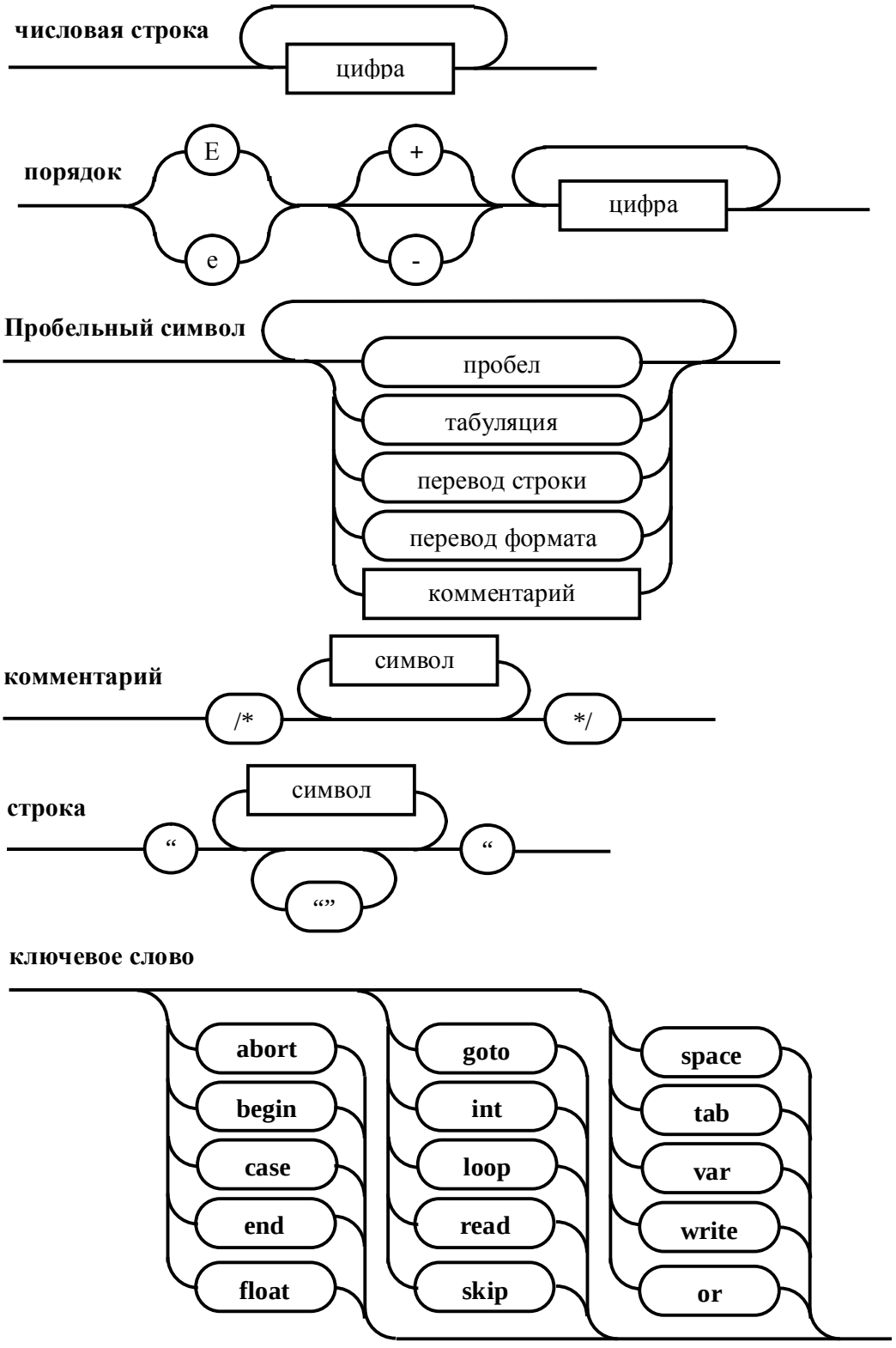
13. Сайт А.И. Легалова с публикациями по методам разработки трансляторов, изучаемым в рамках данной дисциплины: http://www.softcraft.ru

Приложение А. Описание синтаксиса DPL с использованием диаграмм Вирта

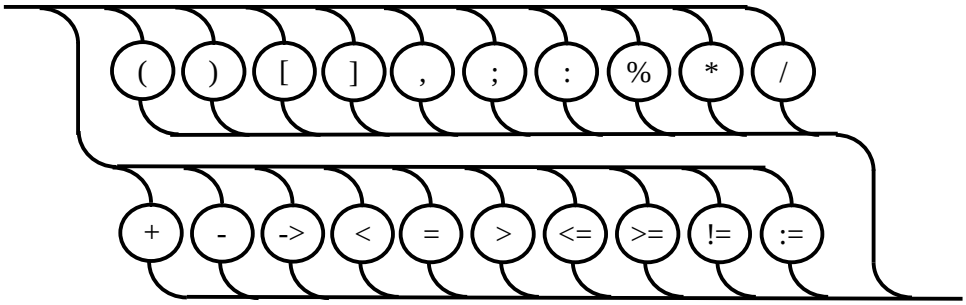
Элементарные конструкции




целое
двоичное
восьмеричное
десятичное
шестнадцатеричное
двоичное
{
2
}
0
1
восьмеричное
{
8
}
0
1
2
3
4
5
6
7
десятичное
{
1
0
}
цифра
шестнадцатеричное
{
1
6
}
a
b
c
d
e
f
цифра
A
B
C
D
E
F
действительное
числовая строка
.
числовая строка
порядок
.

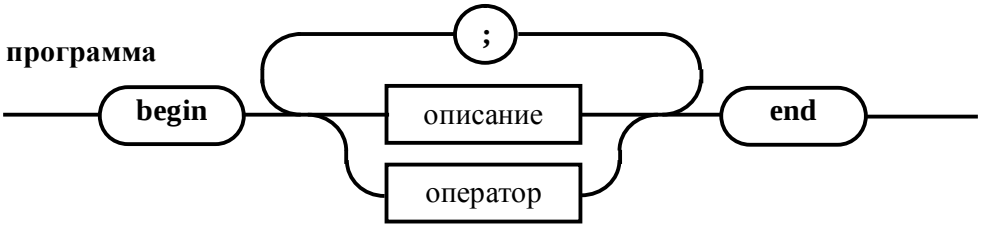

числовая строка
цифра
порядок
E
e
+
-
цифра
Пробельный символ
пробел
табуляция
перевод строки
перевод формата
комментарий
комментарий
/*
символ
*/
строка
“
символ
“
“”
ключевое слово
abort
begin
case
end
float
goto
int
loop
read
skip
space
tab
var
write
or

разделитель

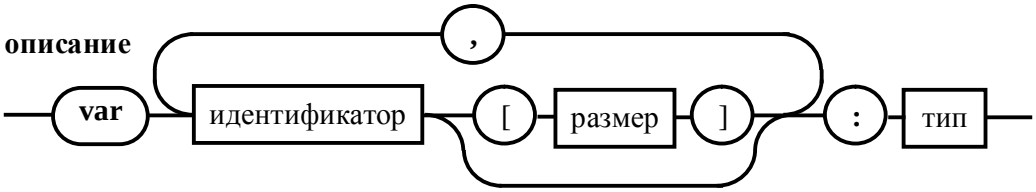


СОСТАВНЫЕ КОНСТРУКЦИИ

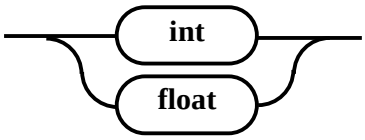
программа



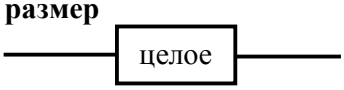
описание



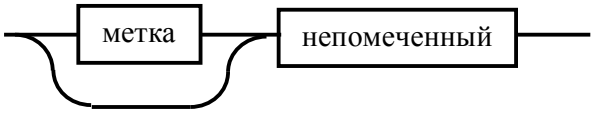
Тип



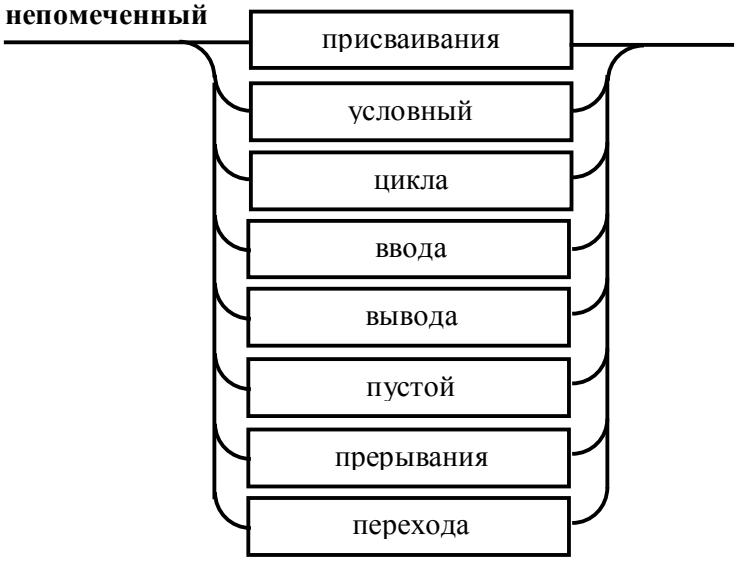
размер



оператор

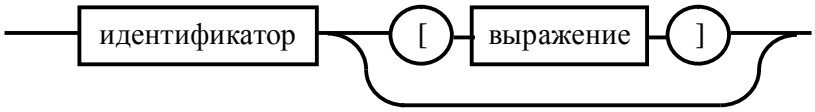


непомеченный

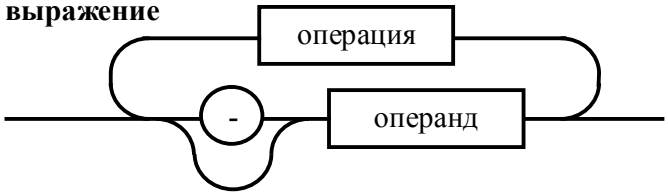


присваивания

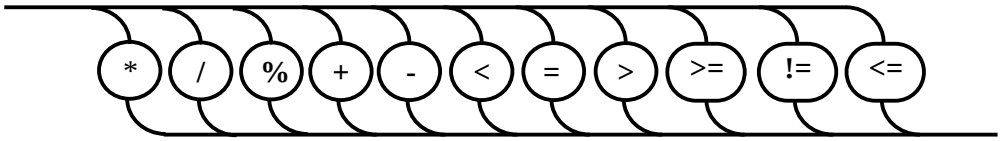
переменная



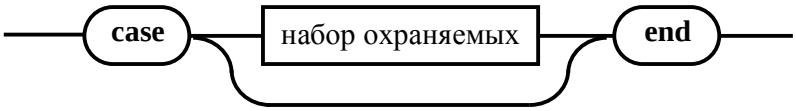
выражение



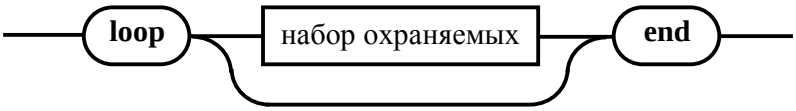
операция



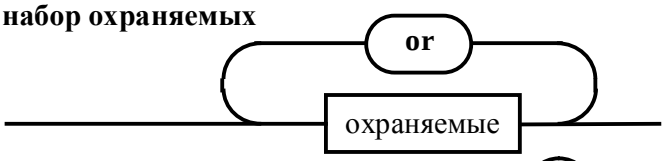
условный



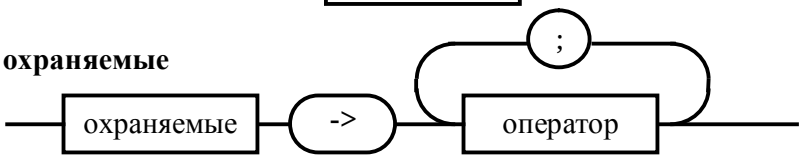
цикла



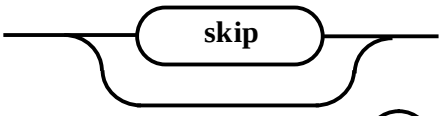
набор охраняемых



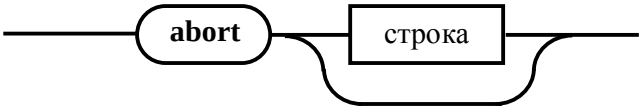
охраняемые



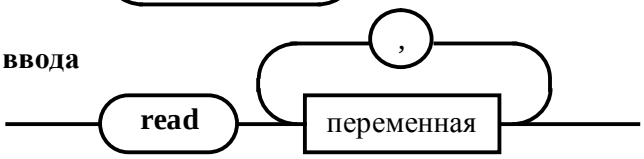
пустой



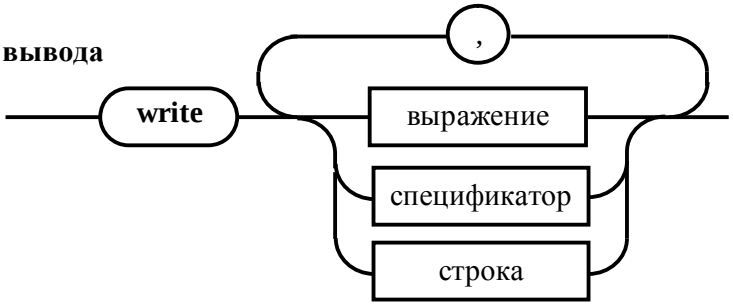
ошибки



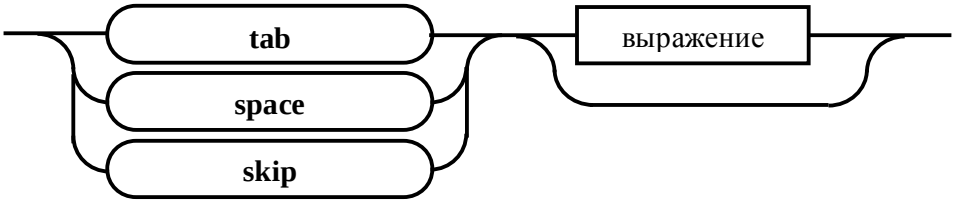
ввода



вывода



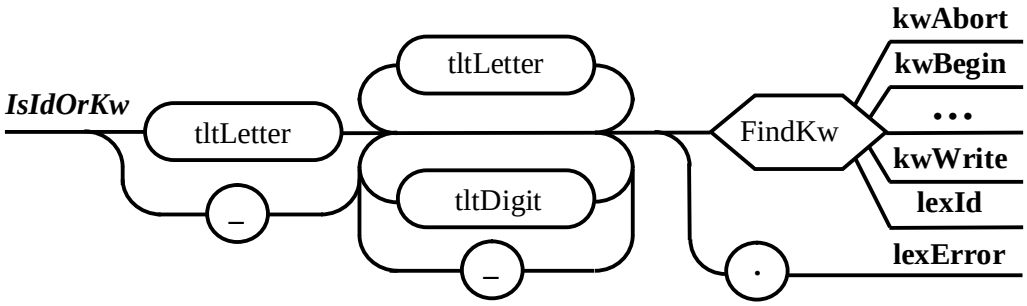
спецификатор



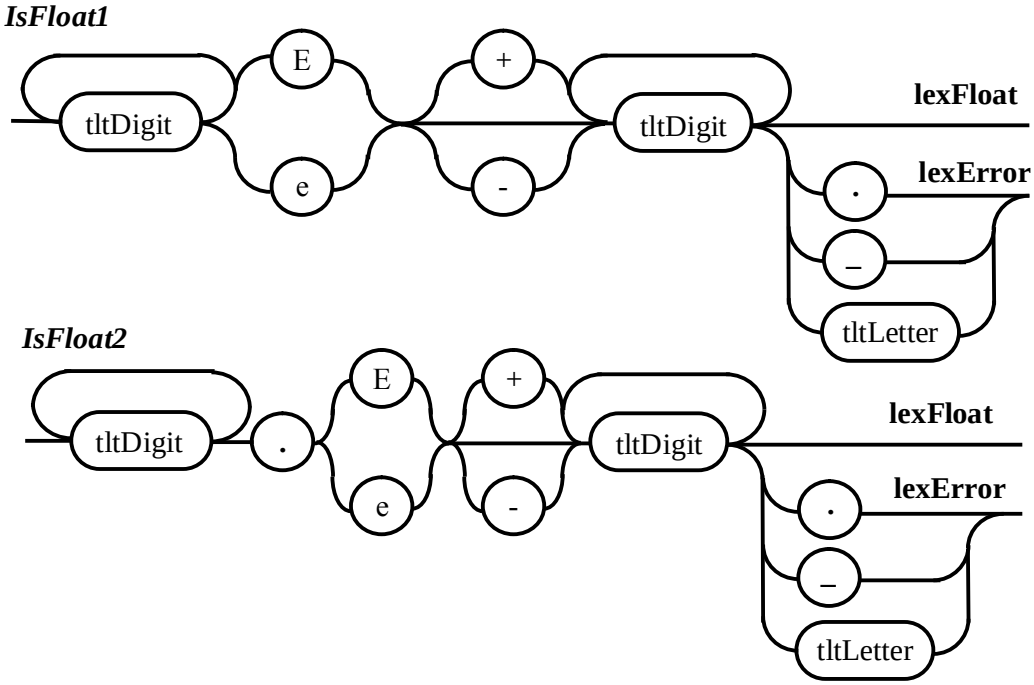
перехода

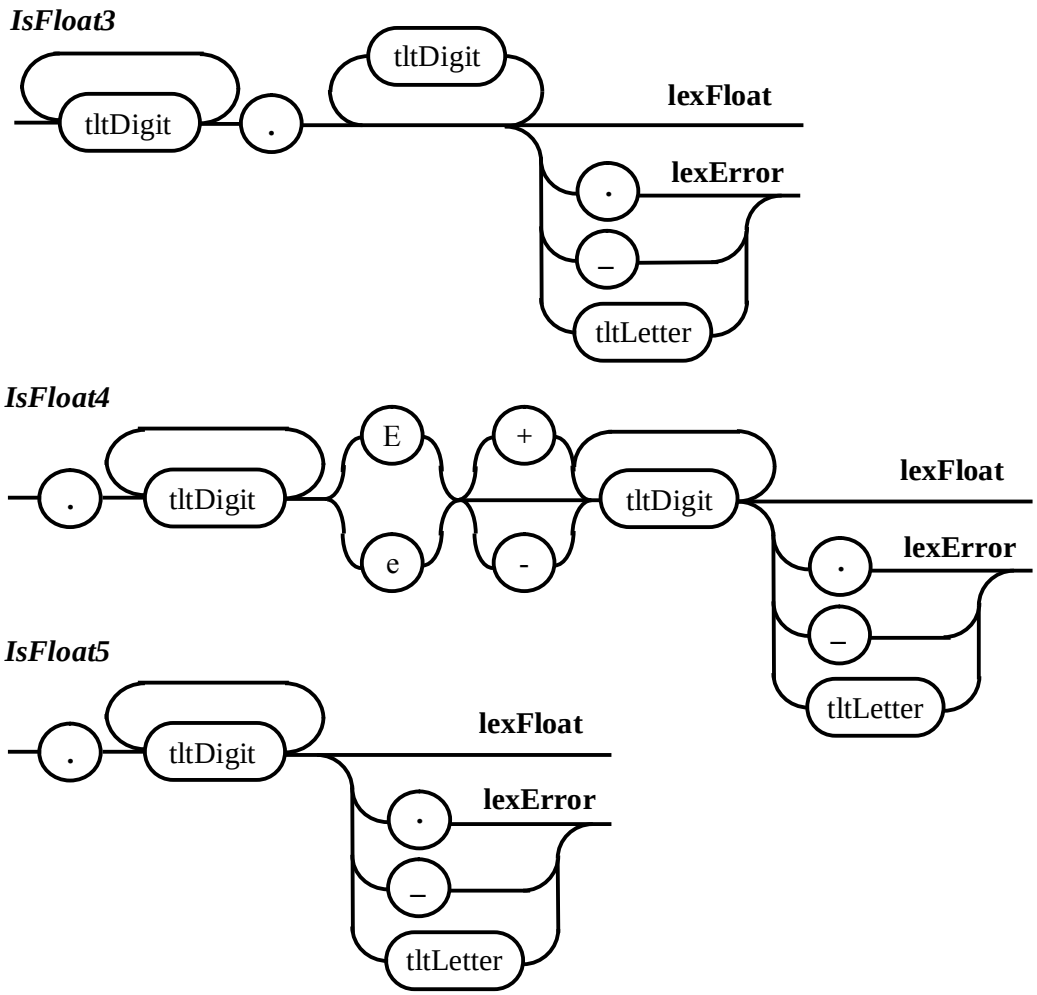
Приложение Б. Диаграммы Вирта, используемые при построении непрямого лексического анализатора

Примечание 1. Лексема, порождаемая при достижении конца обрабатывае-

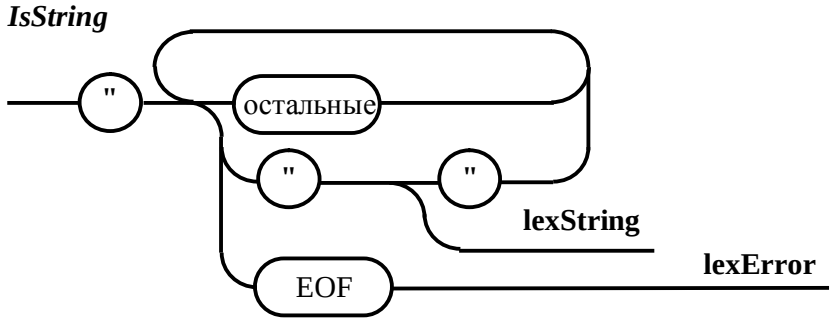


Примечание 2. Идентификатор и ключевые слова описываются правилом с семантической вставкой. Осуществляется также анализ на недопустимость возможного слияния идентификатора с действительным числом, начинаю-

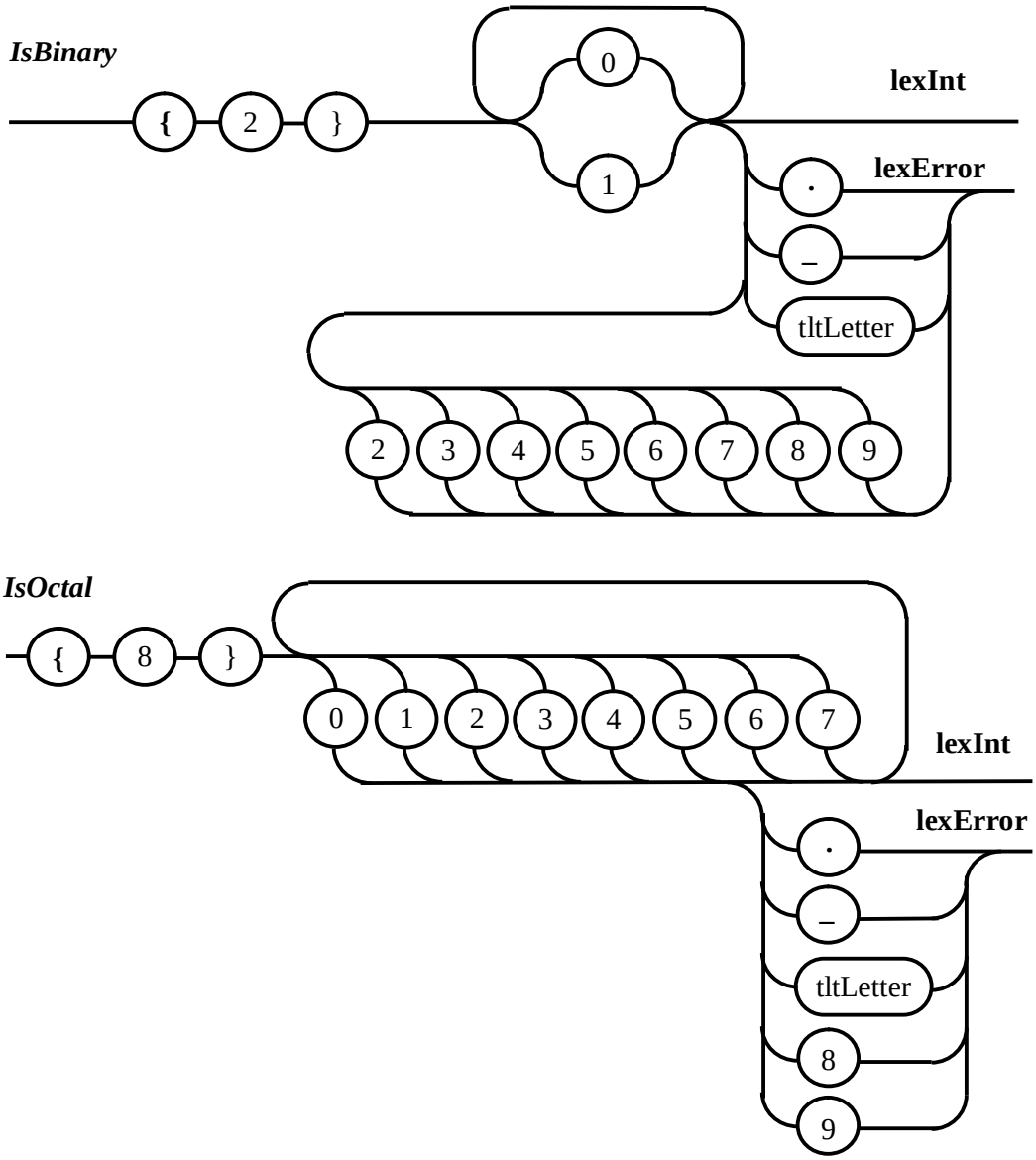




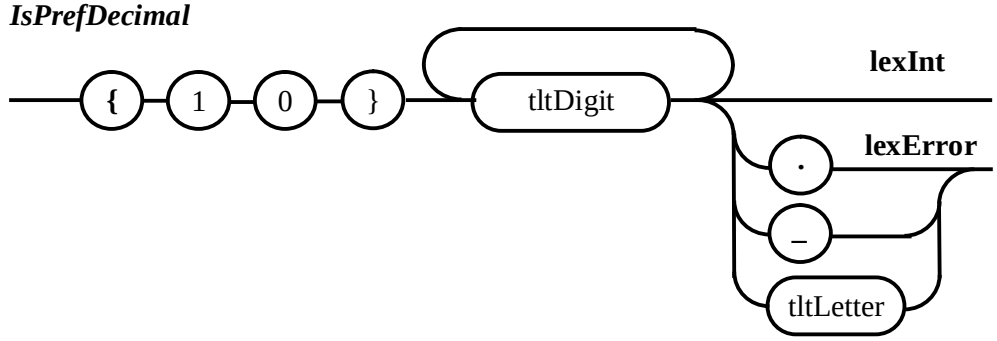
Примечание 3. Пять вариантов правил для распознавания действительного числа приводятся только для демонстрации арбитража при непрямом лексическом анализе. На практике легко можно обойтись одним правилом. Выдача ошибки происходит, если действительное число не отделяется разделителем от идентификатора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

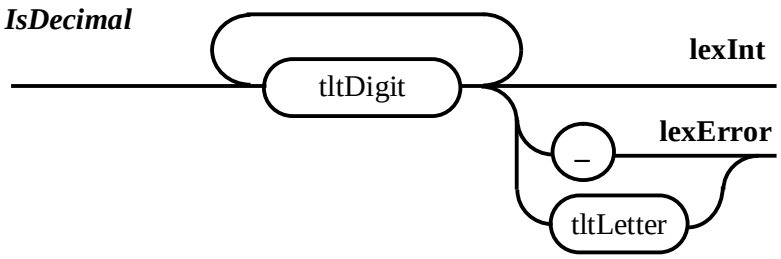


Примечание 4. Под остальными понимаются все символы, кроме апострофа (') и конца файла

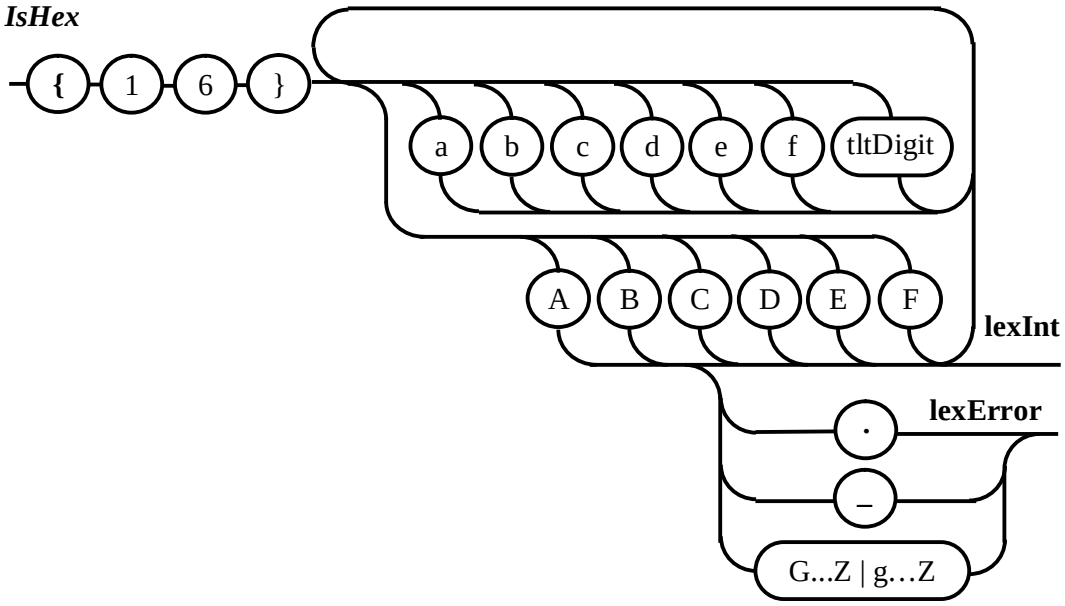


Примечание 5. Для двоичных и десятичных целых чисел необходима проверка того, что оно не сливается с теми цифрами, которые в них не содержатся.

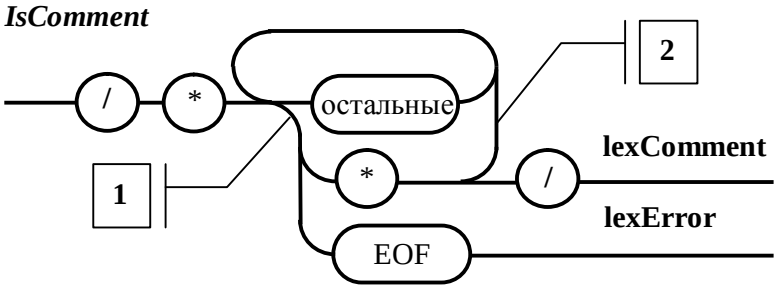




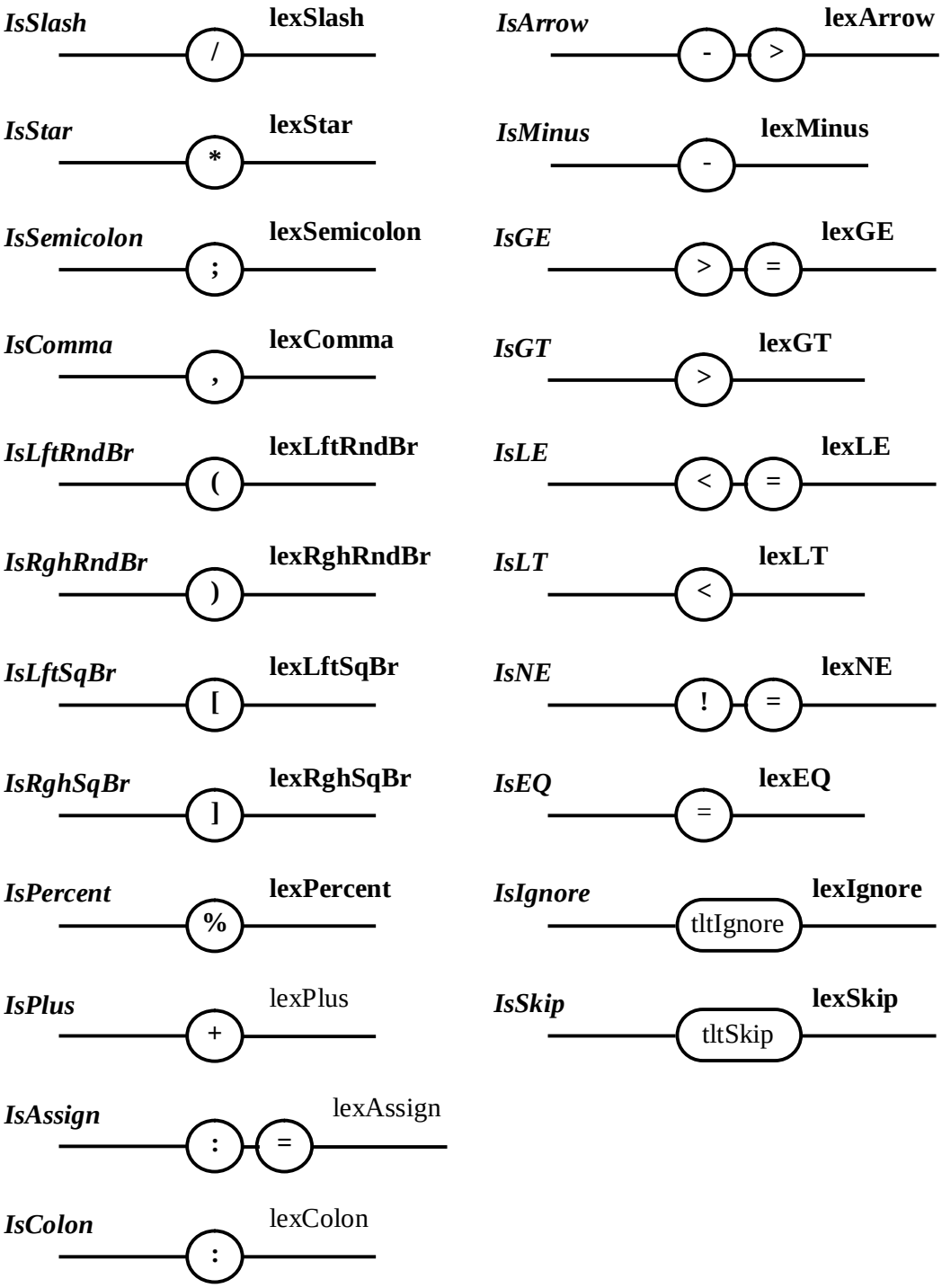
Примечание 6. Для целого десятичного числа без префикса анализ на недопустимость точки излишен, так как похожая ситуация должна была быть проанализирована раньше для действительного числа.



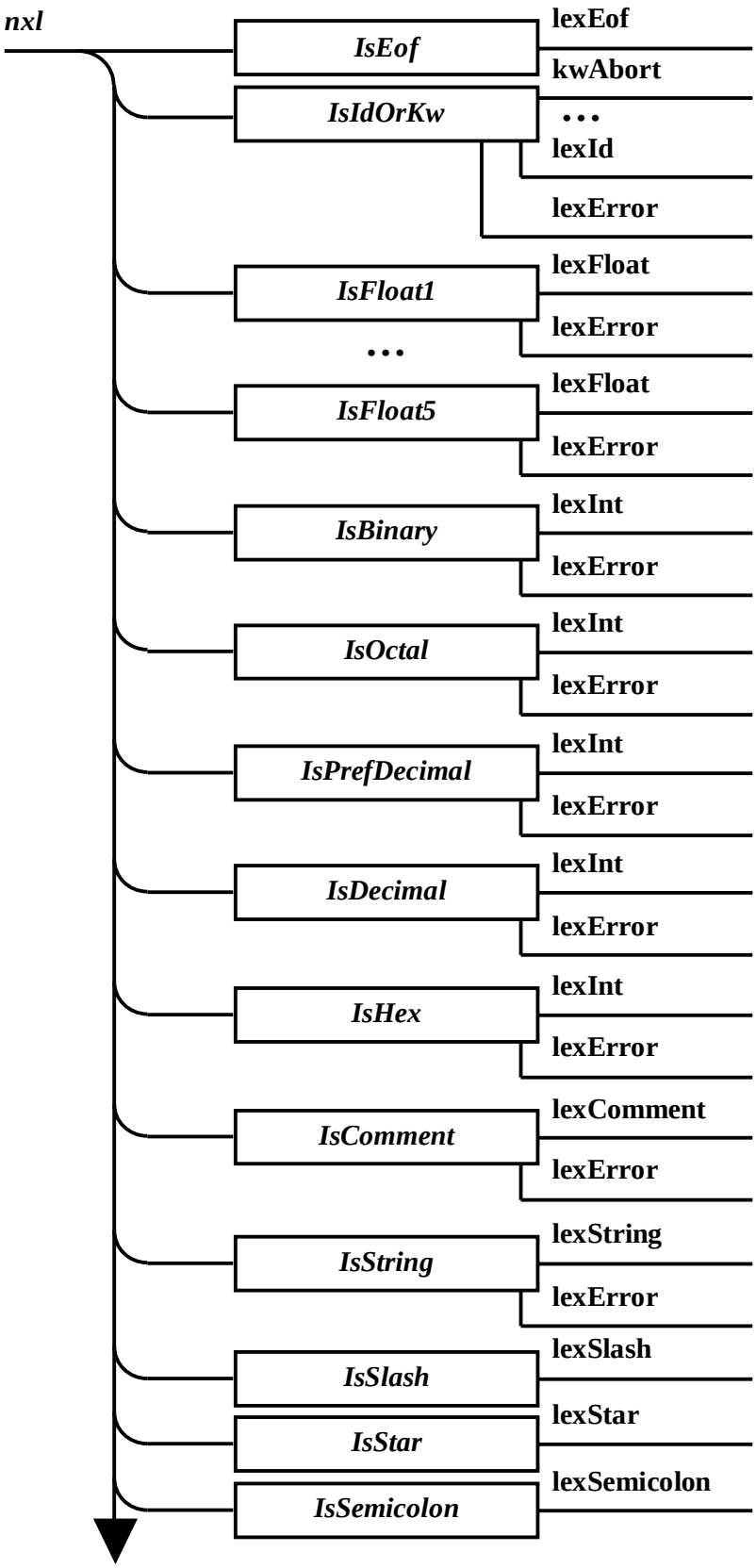
Примечание 7. Для целого шестнадцатеричного проверка на недопустимость должна исключать прописные и строчные буквы, используемые в самом числе. На представленных диаграммах это показано сокращенной записью путем задания диапазона. Это сделано для того, чтобы не загромождать диа-

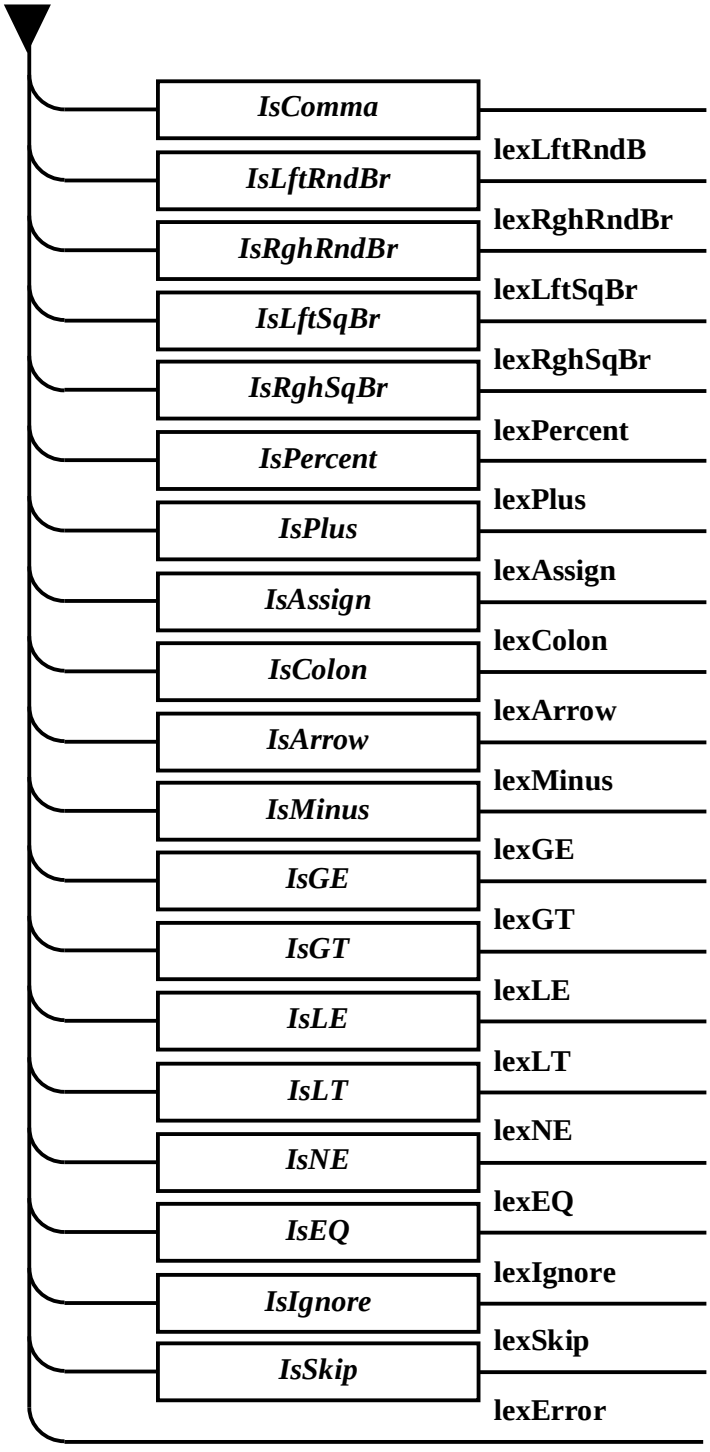


Примечание 8. Под остальными понимаются символы не рассматриваемые непосредственно в текущей точке. В точке 1 – это не «*» и не конец файла; в точке 2 – это не «*», не конец файла и не «/»

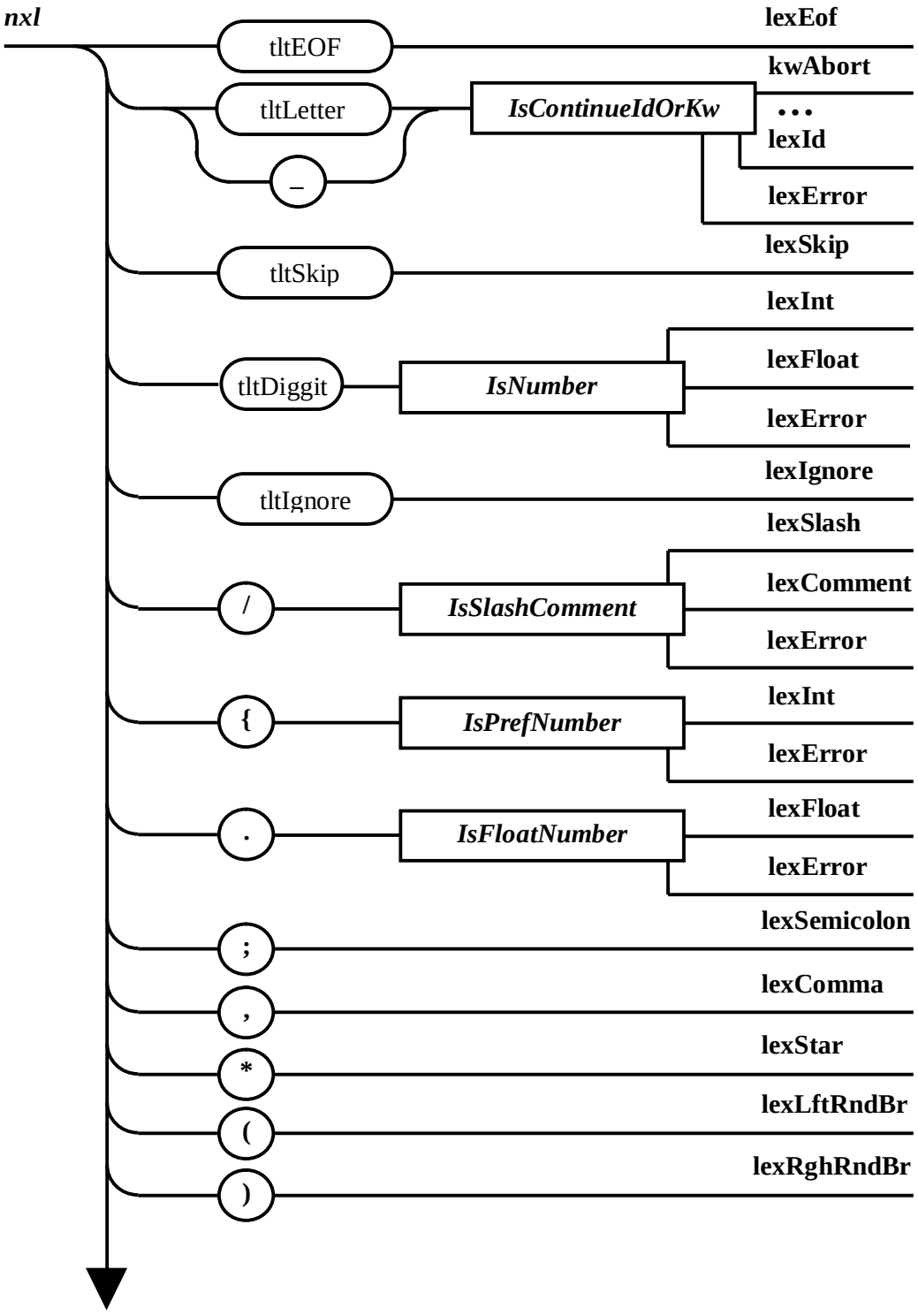


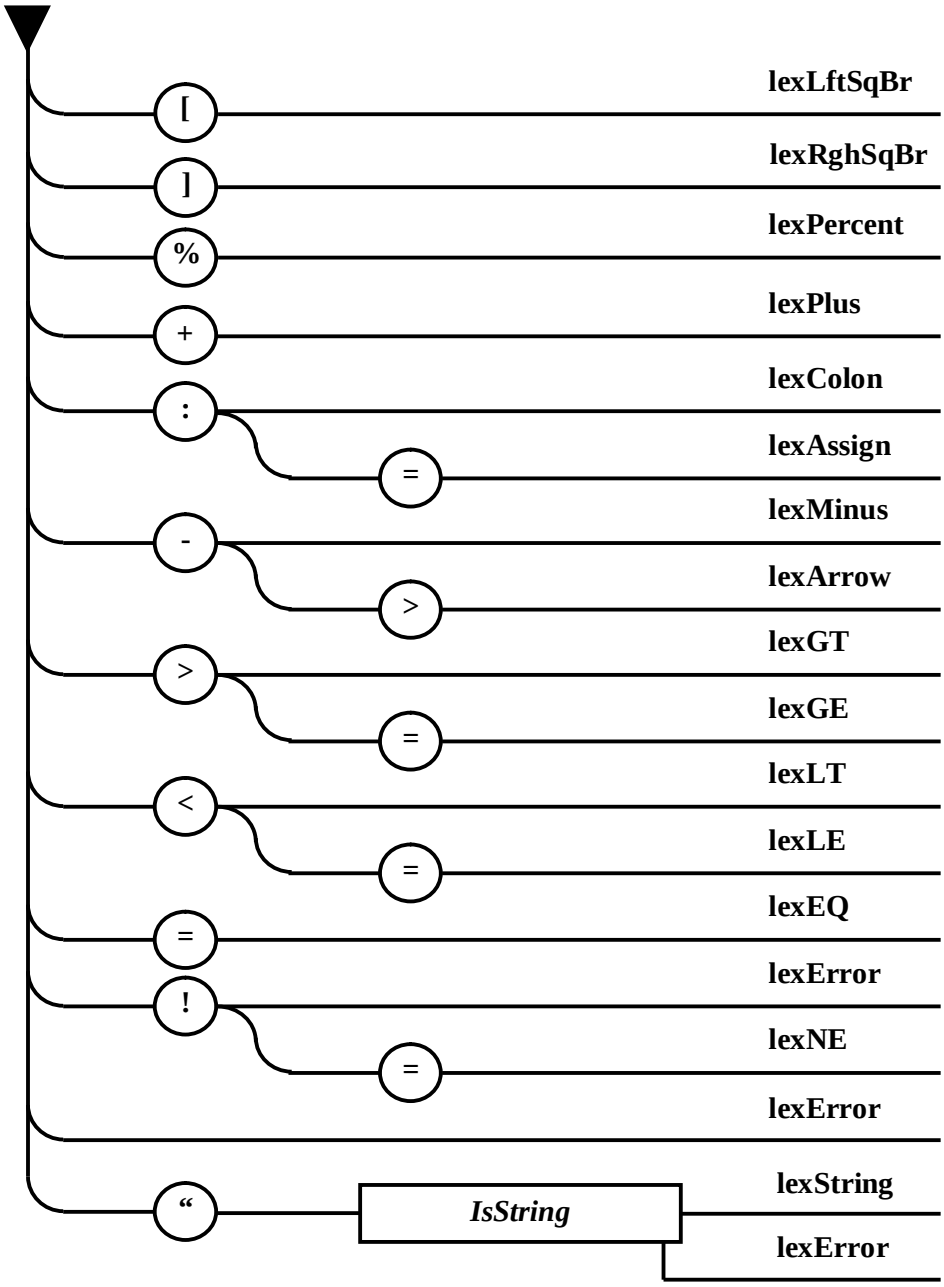
Примечание 9. Лексемы, определяющие разделительные символы, расположены в соответствии с их приоритетом при анализе сверху вниз и слева направо.

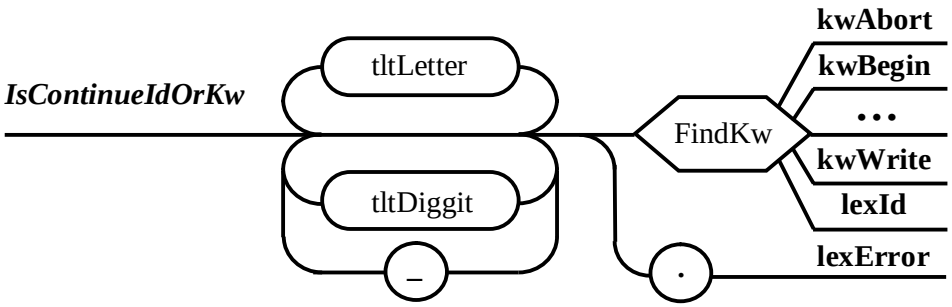

nxl
IsEof
lexEof
kwAbort
IsIdOrKw
• • •
lexId
lexError
IsFloat1
lexFloat
lexError
• • •
IsFloat5
lexFloat
lexError
IsBinary
lexInt
lexError
IsOctal
lexInt
lexError
IsPrefDecimal
lexInt
lexError
IsDecimal
lexInt
lexError
IsHex
lexInt
lexError
IsComment
lexComment
lexError
IsString
lexString
lexError
IsSlash
lexSlash
IsStar
lexStar
IsSemicolon
lexSemicolon

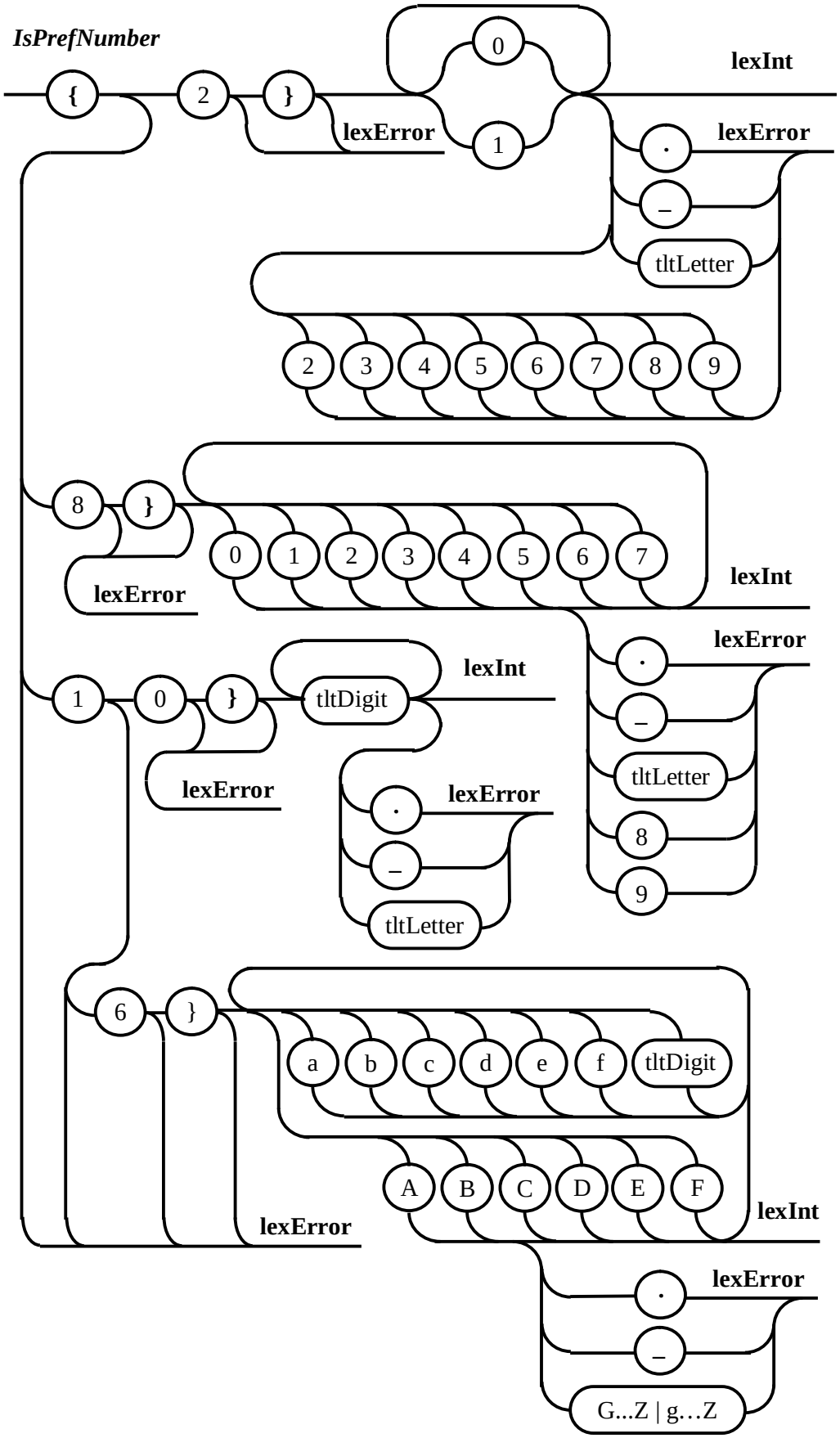

IsComma
IsLftRndBr
IsRghRndBr
IsLftSqBr
IsRghSqBr
IsPercent
IsPlus
IsAssign
IsColon
IsArrow
IsMinus
IsGE
IsGT
IsLE
IsLT
IsNE
IsEQ
IsIgnore
IsSkip
lexLftRndB
lexRghRndBr
lexLftSqBr
lexRghSqBr
lexPercent
lexPlus
lexAssign
lexColon
lexArrow
lexMinus
lexGE
lexGT
lexLE
lexLT
lexNE
lexEQ
lexIgnore
lexSkip
lexError

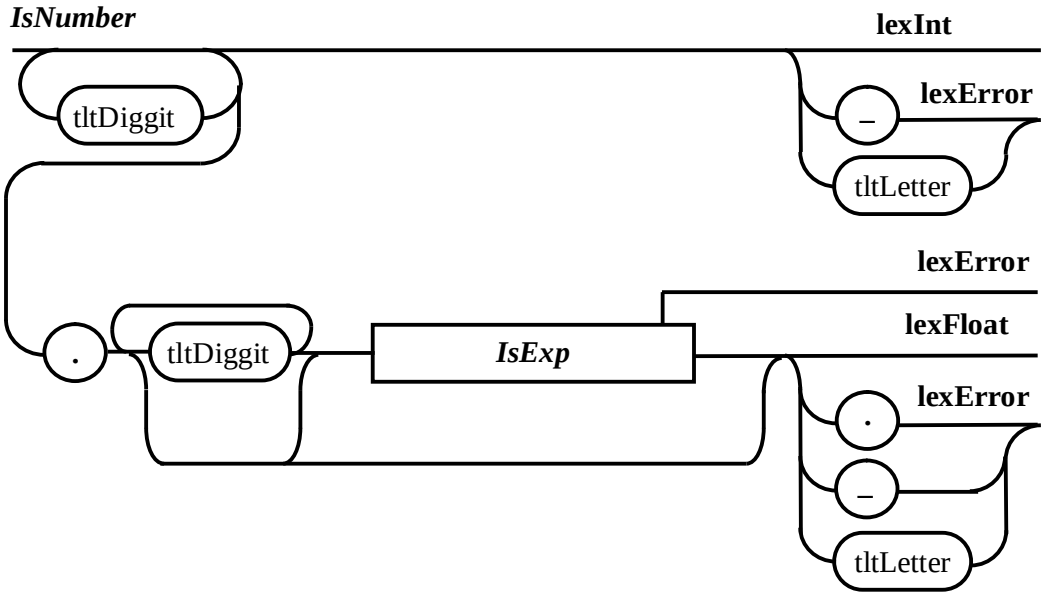
Приложение В. Диаграммы Вирта, используемые при построении прямого лексического анализатора

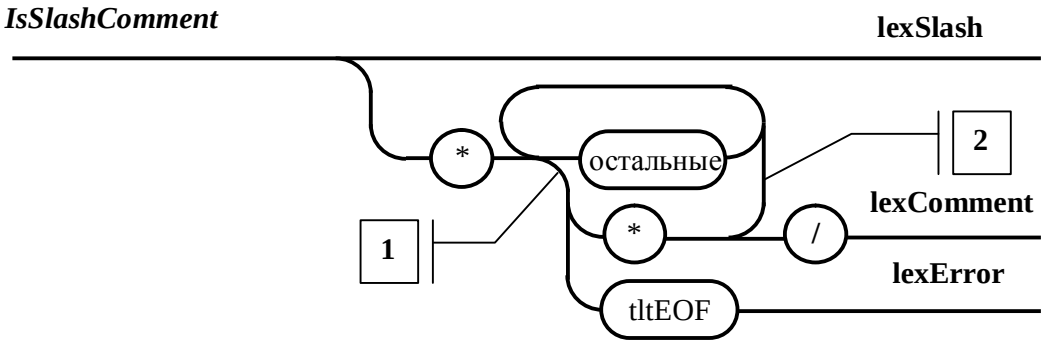



[
lexLftSqBr
]
lexRghSqBr
%
lexPercent
+
lexPlus
:
lexColon
=
lexAssign
-
lexMinus
>
lexArrow
>
lexGT
=
lexGE
<
lexLT
=
lexLE
=
lexEQ
!
lexError
=
lexNE
lexError
“
IsString
lexString
lexError

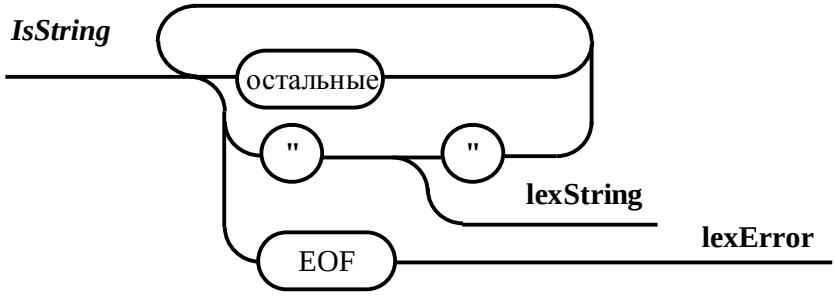

IsContinueIdOrKw
tltLetter
tltDiggit
_
FindKw
.
kwAbort
kwBegin
• • •
kwWrite
lexId
lexError


IsPrefNumber
{
2
}
lexError
0
1
lexInt
.
lexError
_
tltLetter
2
3
4
5
6
7
8
9
8
}
lexError
0
1
2
3
4
5
6
7
lexInt
.
lexError
_
tltLetter
8
9
1
0
}
lexError
tltDigit
lexInt
.
lexError
_
tltLetter
6
}
lexError
a
b
c
d
e
f
tltDigit
A
B
C
D
E
F
lexInt
.
lexError
_
G...Z | g…Z

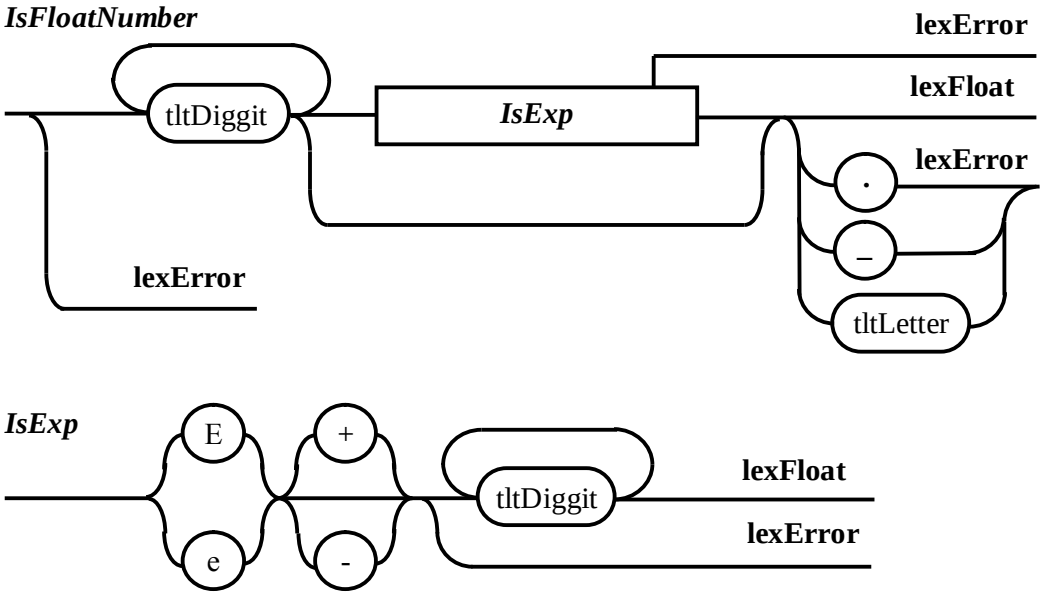




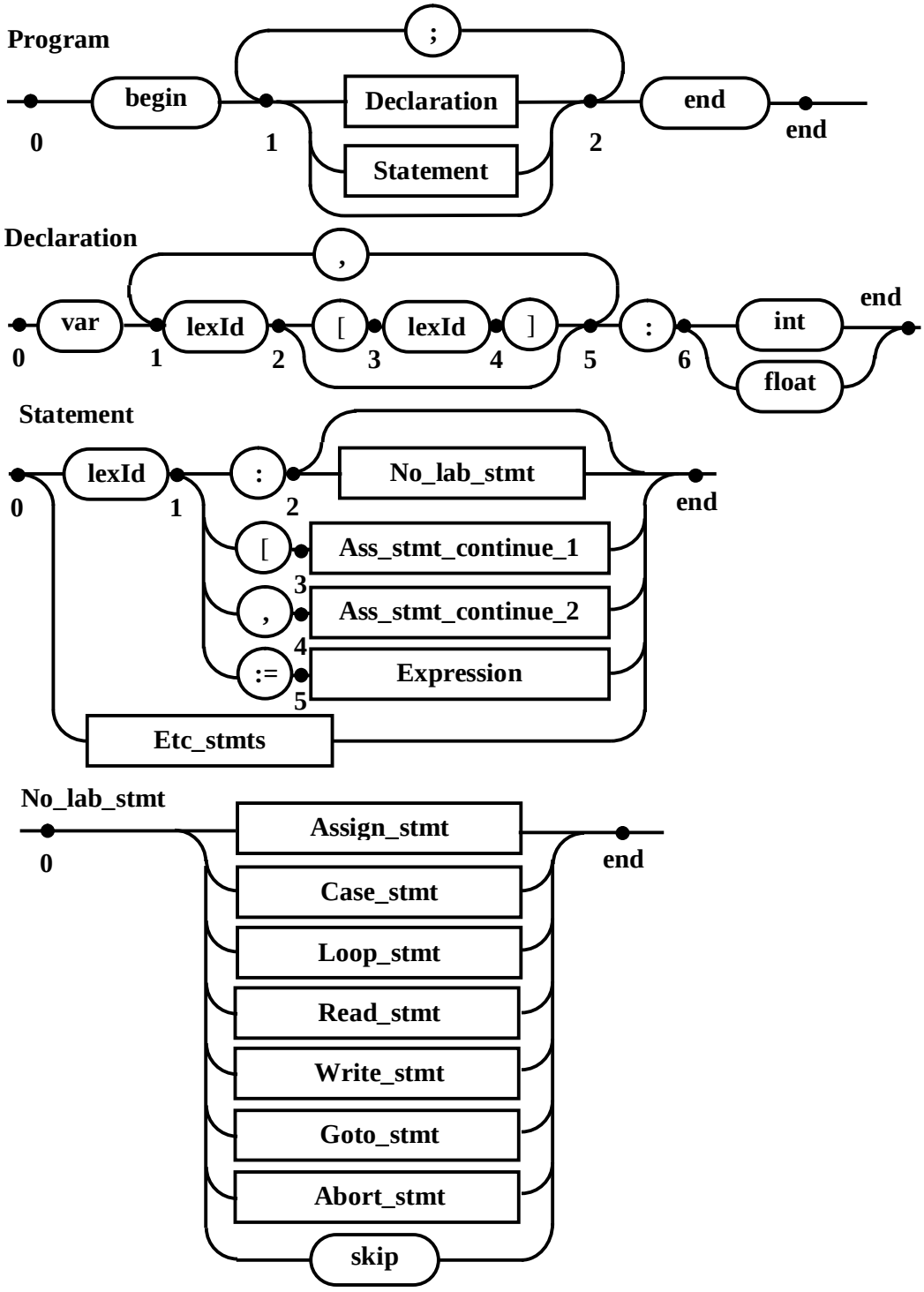
Примечание 1. Под остальными понимаются символы не рассматриваемые непосредственно в текущей точке. В точке 1 – это не «*» и не конец файла; в точке 2 – это не «*», не конец файла и не «/»

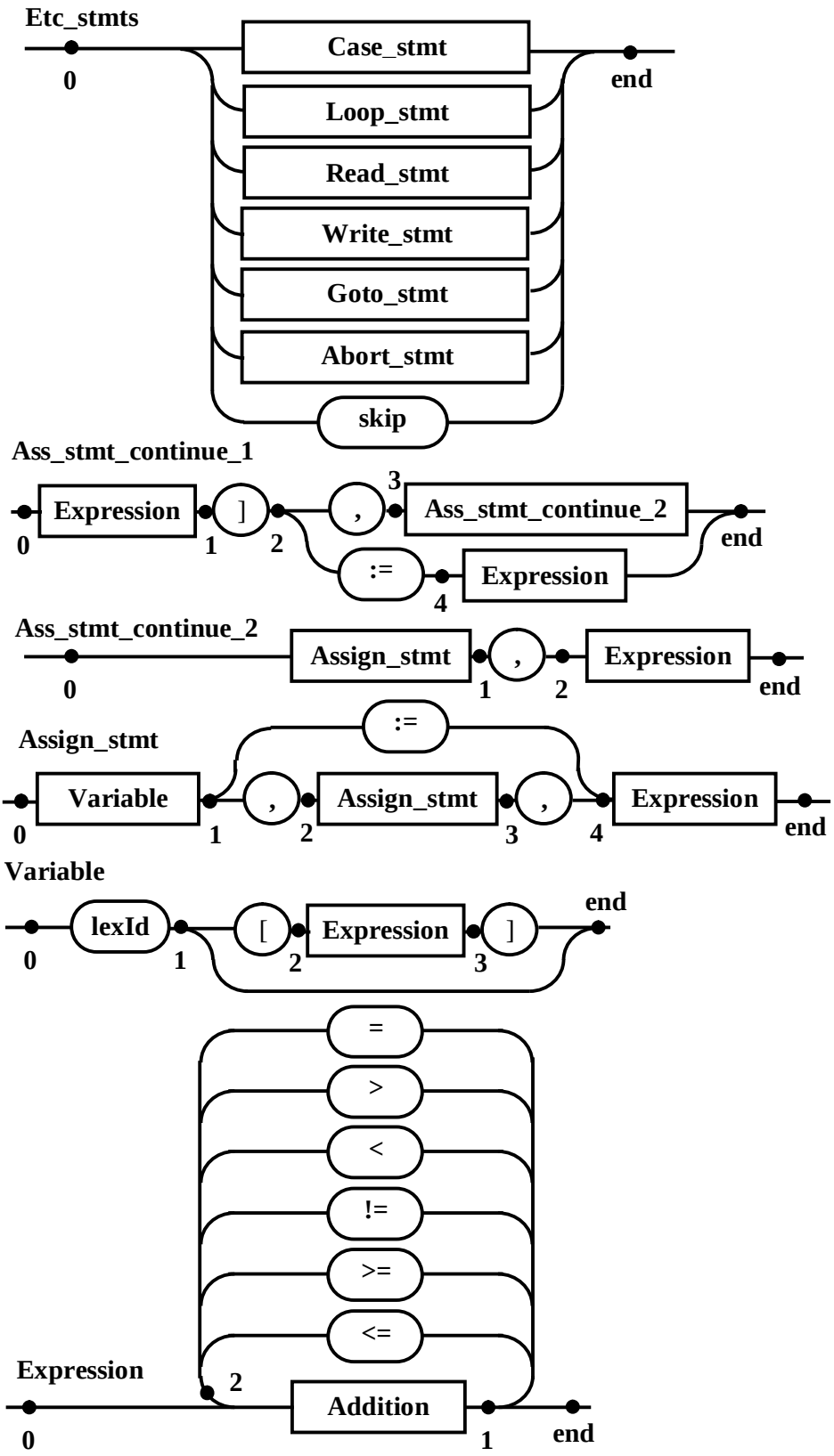


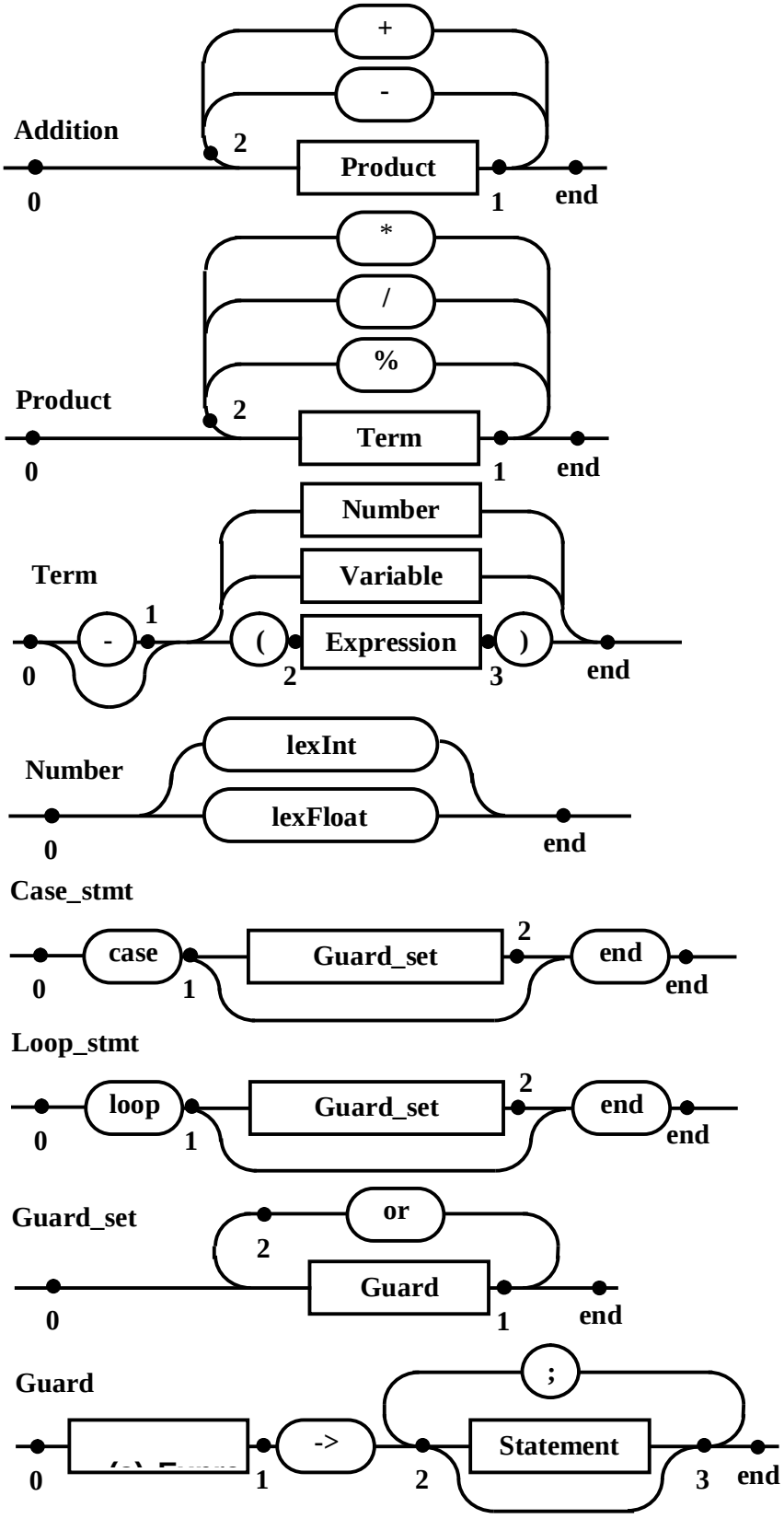
Примечание 2. Под остальными понимаются все символы, кроме апострофа (') и конца файла

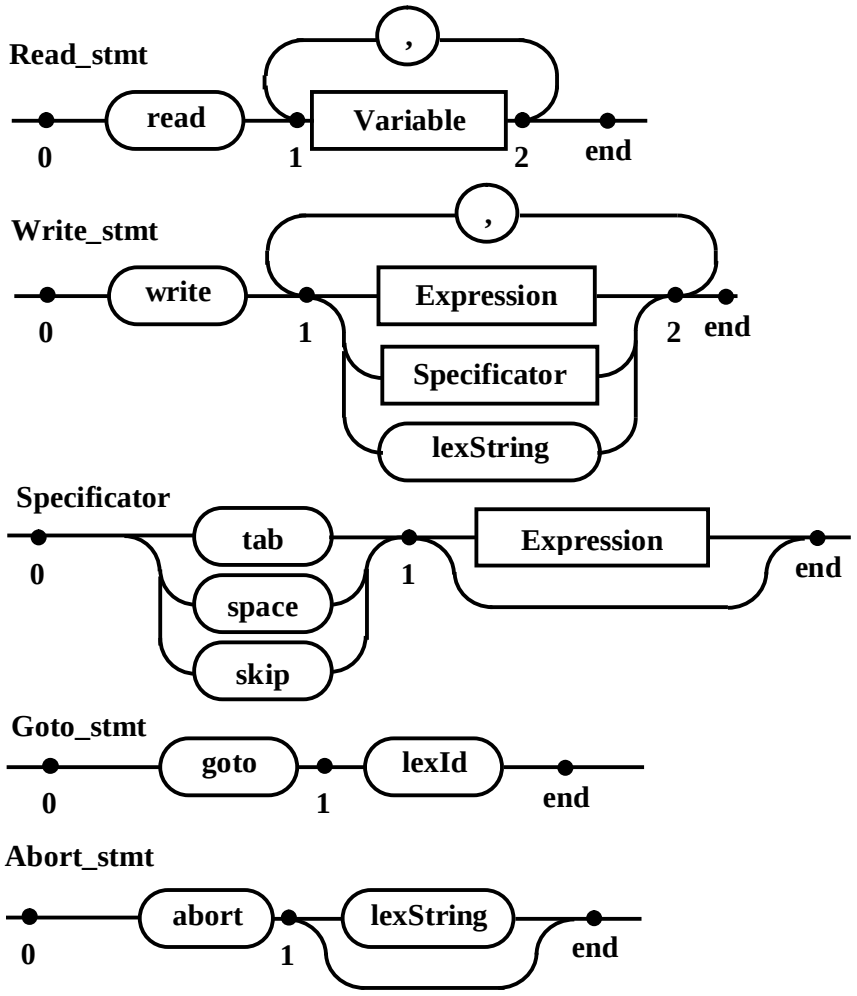

IsFloatNumber
lexError
tltDiggit
IsExp
lexFloat
lexError
.
_
tltLetter
lexError
IsExp
E
e
+
-
tltDiggit
lexFloat
lexError

Приложение Г. Диаграммы Вирта, предназначенные для написания программы распознавателя




Etc_stmts
0
Case_stmt
Loop_stmt
Read_stmt
Write_stmt
Goto_stmt
Abort_stmt
skip
end
Ass_stmt_continue_1
Expression
]
,
:=
Ass_stmt_continue_2
Expression
0
1
2
3
4
end
Ass_stmt_continue_2
Assign_stmt
,
Expression
0
1
2
end
Assign_stmt
:=
Variable
,
Assign_stmt
,
Expression
0
1
2
3
4
end
Variable
lexId
[
Expression
]
0
1
2
3
end
=
>
<
!=
>=
<=
Expression
2
Addition
0
1
end


Addition
+
-
2
Product
0
1
end
Product
*
/
%
2
Term
0
1
end
Term
Number
Variable
1
-
(
Expression
)
0
2
3
end
Number
lexInt
lexFloat
0
end
Case_stmt
case
Guard_set
2
end
0
1
end
Loop_stmt
loop
Guard_set
2
end
0
1
end
Guard_set
or
2
Guard
0
1
end
Guard
;
->
Statement
0
1
2
3
end


Read_stmt
,
read
Variable
0
1
2
end
Write_stmt
,
write
Expression
Specificator
lexString
0
1
2
end
Specificator
tab
space
skip
Expression
0
1
end
Goto_stmt
goto
lexId
0
1
end
Abort_stmt
abort
lexString
0
1
end