

## Тема 1. Общие сведения о трансляторах

### Содержание темы

Цели и задачи дисциплины. Основные понятия и определения. Общие особенности языков программирования и трансляторов. Обобщенная структура транслятора. Варианты взаимодействия блоков транслятора.

### Цели и задачи дисциплины

В настоящее время искусственные языки, использующие для описания предметной области текстовое представление, широко применяются не только в программировании, но и в других областях. С их помощью описывается структура всевозможных документов, трехмерных виртуальных миров, графических интерфейсов пользователя и многих других объектов, используемых в моделях и в реальном мире. Для того, чтобы эти текстовые описания были корректно составлены, а затем правильно распознаны и интерпретированы, используются специальные методы их анализа и преобразования. В основе методов лежит теория языков и формальных грамматик, а также теория автоматов. Программные системы, предназначенные для анализа и интерпретации текстов, называются трансляторами.

Несмотря на то, что к настоящему времени разработаны тысячи различных языков и их трансляторов, процесс создания новых приложений в этой области не прекращается. Это связано как с развитием технологии производства вычислительных систем, так и с необходимостью решения все более сложных прикладных задач. Кроме того, элементы теории языков и формальных грамматик применимы и в других разнообразных областях, например, при описании структур данных, файлов, изображений, представленных не в текстовом, а двоичном формате. Эти методы полезны и при разработке своих трансляторов даже там, где уже имеются соответствующие аналоги. Такая разработка может быть обусловлена различными причинами, в частности, функциональными ограничениями, отсутствием локализации, низкой эффективностью. Например, одной из последних разработок компании Microsoft является язык программирования C#, а одной из причин его создания является стремление к снижению популярности языка программирования Java. Можно привести множество других примеров, когда разработка своего транслятора может оказаться актуальной. Поэтому, основы теории языков и формальных грамматик, а также практические методы разработки трансляторов лежат в фундаменте инженерного образования по информатике и вычислительной технике.

Предлагаемый материал затрагивает основы методов разработки трансляторов и содержит сведения, необходимые для изучения логики их функционирования, используемого математического аппарата (теории формальных языков и формальных грамматик, метаязыков). Он используется в рамках семестровых лекционных курсов, читаемых для различных специальностей, на факультете информатики и вычислительной техники Красноярского государственного технического университета. В ходе лабораторных работ осуществляется непосредственное знакомство с отдельными методами создания трансляторов.

Цель дисциплины: предоставить знания по основам теории языков и формальных грамматик, теории автоматов, методам разработки трансляторов.

Для достижения поставленной цели в ходе преподавания дисциплины решаются следующие задачи:

1. В ходе лекционного курса рассматриваются общие принципы организации процесса трансляции и структуры трансляторов. Изучаются основы теории построения трансляторов.

На лабораторных занятиях и в ходе самостоятельной работы осуществляется практическое закрепление полученных теоретических знаний: разрабатывается транслятор для простого языка программирования.

# Основные понятия и определения

Большинство рассматриваемых определений заимствовано из [[АРНФТС](#)].

**Транслятор** - *обслуживающая программа, преобразующая исходную программу, предоставленную на входном языке программирования, в рабочую программу, представленную на объектном языке.*

Приведенное определение относится ко всем разновидностям транслирующих программ. Однако у каждой из таких программ могут иметься свои особенности по организации процесса трансляции. В настоящее время трансляторы разделяются на три основные группы: ассемблеры, компиляторы и интерпретаторы.

**Ассемблер** - *системная обслуживающая программа, которая преобразует символические конструкции в команды машинного языка.* Специфической чертой ассемблеров является то, что они осуществляют дословную трансляцию одной символической команды в одну машинную. Таким образом, язык ассемблера (еще называется автокодом) предназначен для облегчения восприятия системы команд компьютера и ускорения программирования в этой системе команд. Программисту гораздо легче запомнить мнемоническое обозначение машинных команд, чем их двоичный код. Поэтому, основной выигрыш достигается не за счет увеличения мощности отдельных команд, а за счет повышения эффективности их восприятия.

Вместе с тем, язык ассемблера, кроме аналогов машинных команд, содержит множество дополнительных директив, облегчающих, в частности, управление ресурсами компьютера, написание повторяющихся фрагментов, построение многомодульных программ. Поэтому выразительность языка намного богаче, чем просто языка символического кодирования, что значительно повышает эффективность программирования.

**Компилятор** - *это обслуживающая программа, выполняющая трансляцию на машинный язык программы, записанной на исходном языке программирования.* Также как и ассемблер, компилятор обеспечивает преобразование программы с одного языка на другой (чаще всего, в язык конкретного компьютера). Вместе с тем, команды исходного языка значительно отличаются по организации и мощности от команд машинного языка. Существуют языки, в которых одна команда исходного языка транслируется в 7-10 машинных команд. Однако есть и такие языки, в которых каждой команде может соответствовать 100 и более машинных команд (например, Пролог). Кроме того, в исходных языках достаточно часто используется строгая типизация данных, осуществляемая через их предварительное описание. Программирование может опираться не на кодирование алгоритма, а на тщательное обдумывание структур данных или классов. Процесс трансляции с таких языков обычно называется компиляцией, а исходные языки обычно относятся к языкам программирования высокого уровня (или высокоуровневым языкам). Абстрагирование языка программирования от системы команд компьютера привело к независимому созданию самых разнообразных языков, ориентированных на решение конкретных задач. Появились языки для научных расчетов, экономических расчетов, доступа к базам данных и другие.

**Интерпретатор** - *программа или устройство, осуществляющее пооператорную трансляцию и выполнение исходной программы.* В отличие от компилятора, интерпретатор не порождает на выходе программу на машинном языке. Распознав команду исходного языка, он тут же выполняет ее. Как в компиляторах, так и в интерпретаторах используются одинаковые методы анализа исходного текста программы. Но интерпретатор позволяет начать обработку данных после написания даже одной команды. Это делает процесс разработки и отладки программ более гибким. Кроме того, отсутствие выходного машинного кода позволяет не "захламлять" внешние устройства дополнительными файлами, а сам интерпретатор можно достаточно легко адаптировать к любым машинным архитектурам, разработав его только один раз на широко распространенном языке программирования. Поэтому, интерпретируемые языки, типа Java Script, VB Script, получили широкое распространение. Недостатком интерпретаторов является низкая скорость выполнения программ. Обычно интерпретируемые программы выполняются в 50-100 раз медленнее программ, написанных в машинных кодах.

**Эмулятор** - *программа или программно-техническое средство, обеспечивающее возможность без перепрограммирования выполнять на данной ЭВМ программу, использующую коды или способы выполнения операция, отличные от данной ЭВМ.* Эмулятор похож на интерпретатор тем, что непосредственно исполняет программу, написанную на некотором языке. Однако, чаще всего это

машинный язык или промежуточный код. И тот и другой представляют команды в двоичном коде, которые могут сразу исполняться после распознавания кода операций. В отличие от текстовых программ, не требуется распознавать структуру программы, выделять операнды.

Эмуляторы используются достаточно часто в самых различных целях. Например, при разработке новых вычислительных систем, сначала создается эмулятор, выполняющий программы, разрабатываемые для еще несуществующих компьютеров. Это позволяет оценить систему команд и наработать базовое программное обеспечение еще до того, как будет создано соответствующее оборудование.

Очень часто эмулятор используется для выполнения старых программ на новых вычислительных машинах. Обычно новые компьютеры обладают более высоким быстродействием и имеют более качественное периферийное оборудование. Это позволяет эмулировать старые программы более эффективно по сравнению с их выполнением на старых компьютерах. Примером такого подхода является разработка эмуляторов домашнего компьютера ZX Spectrum с микропроцессором Z80. До сих пор находятся любители поиграть на эмуляторе в устаревшие, но все еще не утратившие былой привлекательности, игровые программы. Эмулятор может также использоваться как более дешевый аналог современных компьютерных систем, обеспечивая при этом приемлемую производительность, эквивалентную младшим моделям некоторого семейства архитектур. В качестве примера можно привести эмуляторы IBM PC совместимых компьютеров, реализованные на более мощных компьютерах фирмы Apple. Ряд эмуляторов, написанных для IBM PC, с успехом заменяют различные игровые приставки.

Эмулятор промежуточного представления, как и интерпретатор, могут легко переноситься с одной компьютерной архитектуры на другую, что позволяет создавать мобильное программное обеспечение. Именно это свойство предопределило успех языка программирования Java, с которого программа транслируется в промежуточный код. Исполняющая этот код виртуальная Java машина, является ни чем иным как эмулятором, работающим под управлением любой современной операционной системы.

**Перекодировщик** - программа или программное устройство, переводящие программы, написанные на машинном языке одной ЭВМ в программы на машинном языке другой ЭВМ. Если эмулятор является менее интеллектуальным аналогом интерпретатора, то перекодировщик выступает в том же качестве по отношению к компилятору. Точно также исходный (и обычно двоичный) машинный код или промежуточное представление преобразуются в другой аналогичный код по одной команде и без какого-либо общего анализа структуры исходной программы. Перекодировщики бывают полезны при переносе программ с одних компьютерных архитектур на другие. Они могут также использоваться для восстановления текста программы на языке высокого уровня по имеющемуся двоичному коду.

**Макропроцессор** - программа, обеспечивающая замену одной последовательности символов другой [Браун]. Это разновидность компилятора. Он осуществляет генерацию выходного текста путем обработки специальных вставок, располагаемых в исходном тексте. Эти вставки оформляются специальным образом и принадлежат конструкциям языка, называемого макроязыком. Макропроцессоры часто используются как надстройки над языками программирования, увеличивая функциональные возможности систем программирования. Практически любой ассемблер содержит макропроцессор, что повышает эффективность разработки машинных программ. Такие системы программирования обычно называются макроассемблерами.

Макропроцессоры используются и с языками высокого уровня. Они увеличивают функциональные возможности таких языков как PL/I, C, C++. Особенно широко макропроцессоры применяются в C и C++, позволяя упростить написание программ. Примером широкого использования макропроцессоров является библиотека классов Microsoft Foundation Classes (MFC). Через макровставки в ней реализованы карты сообщений и другие программные объекты. При этом, макропроцессоры повышают эффективность программирования без изменения синтаксиса и семантики языка.

**Синтаксис** - совокупность правил некоторого языка, определяющих формирование его элементов. Иначе говоря, это совокупность правил образования семантически значимых последовательностей символов в данном языке. Синтаксис задается с помощью правил, которые описывают понятия некоторого языка. Примерами понятий являются: переменная, выражение, оператор, процедура. Последовательность понятий и их допустимое использование в правилах определяет синтаксически правильные структуры, образующие программы. Именно иерархия объектов, а не то, как они

взаимодействуют между собой, определяются через синтаксис. Например, оператор может встречаться только в процедуре, а выражение в операторе, переменная может состоять из имени и необязательных индексов и т.д. Синтаксис не связан с такими явлениями в программе как "переход на несуществующую метку" или "переменная с данным именем не определена". Этим занимается семантика.

**Семантика** - правила и условия, определяющие соотношения между элементами языка и их смысловыми значениями, а также интерпретацию содержательного значения синтаксических конструкций языка. Объекты языка программирования не только размещаются в тексте в соответствии с некоторой иерархией, но и дополнительно связаны между собой посредством других понятий, образующих разнообразные ассоциации. Например, переменная, для которой синтаксис определяет допустимое местоположение только в описаниях и некоторых операторах, обладает определенным типом, может использоваться с ограниченным множеством операций, имеет адрес, размер и должна быть описана до того, как будет использоваться в программе.

**Синтаксический анализатор** - компонента компилятора, осуществляющая проверку исходных операторов на соответствие синтаксическим правилам и семантике данного языка программирования. Несмотря на название, анализатор занимается проверкой и синтаксиса, и семантики. Он состоит из нескольких блоков, каждый из которых решает свои задачи. Более подробно будет рассмотрен при описании структуры транслятора.

## Общие особенности языков программирования и трансляторов

Языки программирования достаточно сильно отличаются друг от друга по назначению, структуре, семантической сложности, методам реализации. Это накладывает свои специфические особенности на разработку конкретных трансляторов.

Языки программирования являются инструментами для решения задач в разных предметных областях, что определяет специфику их организации и различия по назначению. В качестве примера можно привести такие языки как Фортран, ориентированный на научные расчеты, С, предназначенный для системного программирования, Пролог, эффективно описывающий задачи логического вывода, Лисп, используемый для рекурсивной обработки списков. Эти примеры можно продолжить. Каждая из предметных областей предъявляет свои требования к организации самого языка. Поэтому можно отметить разнообразие форм представления операторов и выражений, различие в наборе базовых операций, снижение эффективности программирования при решении задач, не связанных с предметной областью. Языковые различия отражаются и в структуре трансляторов. Лисп и Пролог чаще всего выполняются в режиме интерпретации из-за того, что используют динамическое формирование типов данных в ходе вычислений. Для трансляторов с языка Фортран характерна агрессивная оптимизация результирующего машинного кода, которая становится возможной благодаря относительно простой семантике конструкций языка - в частности, благодаря отсутствию механизмов альтернативного именования переменных через указатели или ссылки. Наличие же указателей в языке С предъявляет специфические требования к динамическому распределению памяти.

Структура языка характеризует иерархические отношения между его понятиями, которые описываются синтаксическими правилами. Языки программирования могут сильно отличаться друг от друга по организации отдельных понятий и по отношениям между ними. Язык программирования PL/1 допускает произвольное вложение процедур и функций, тогда как в С все функции должны находиться на внешнем уровне вложенности. Язык С++ допускает описание переменных в любой точке программы перед первым ее использованием, а в Паскале переменные должны быть определены в специальной области описания. Еще дальше в этом вопросе идет PL/1, который допускает описание переменной после ее использования. Или описание можно вообще опустить и руководствоваться правилами, принятыми по умолчанию. В зависимости от принятого решения, транслятор может анализировать программу за один или несколько проходов, что влияет на скорость трансляции.

Семантика языков программирования изменяется в очень широких пределах. Они отличаются не только по особенностям реализации отдельных операций, но и по парадигмам программирования, определяющим принципиальные различия в методах разработки программ. Специфика реализации

операций может касаться как структуры обрабатываемых данных, так и правил обработки одних и тех же типов данных. Такие языки, как PL/1 и APL поддерживают выполнение матричных и векторных операций. Большинство же языков работают в основном со скалярами, предоставляя для обработки массивов процедуры и функции, написанные программистами. Но даже при выполнении операции сложения двух целых чисел такие языки, как С и Паскаль могут вести себя по-разному.

Наряду с традиционным процедурным программированием, называемым также императивным, существуют такие парадигмы как функциональное программирование, логическое программирование и объектно-ориентированное программирование. Надеюсь, что в этом ряду займет свое место и предложенная мною процедурно-параметрическая парадигма программирования [Легалов2000]. Структура понятий и объектов языков сильно зависит от избранной парадигмы, что также влияет на реализацию транслятора.

Даже один и тот же язык может быть реализован несколькими способами. Это связано с тем, что теория формальных грамматик допускает различные методы разбора одних и тех же предложений. В соответствии с этим трансляторы разными способами могут получать один и тот же результат (объектную программу) по первоначальному исходному тексту.

Вместе с тем, все языки программирования обладают рядом общих характеристик и параметров. Эта общность определяет и схожие для всех языков принципы организации трансляторов.

1. Языки программирования предназначены для облегчения программирования. Поэтому их операторы и структуры данных более мощные, чем в машинных языках.
2. Для повышения наглядности программ вместо числовых кодов используются символические или графические представления конструкций языка, более удобные для их восприятия человеком.
3. Для любого языка определяется:
  - Множество символов, которые можно использовать для записи правильных программ (алфавит), основные элементы.
  - Множество правильных программ (синтаксис).
  - "Смысл" каждой правильной программы (семантика).

Независимо от специфики языка любой транслятор можно считать функциональным преобразователем  $F$ , обеспечивающим однозначное отображение  $X$  в  $Y$ , где  $X$  - программа на исходном языке,  $Y$  - программа на выходном языке. Поэтому сам процесс трансляции формально можно представить достаточно просто и понятно:

$$Y = F(X)$$

Формально каждая правильная программа  $X$  - это цепочка символов из некоторого алфавита  $A$ , преобразуемая в соответствующую ей цепочку  $Y$ , составленную из символов алфавита  $B$ . Язык программирования, как и любая сложная система, определяется через иерархию понятий, задающую взаимосвязи между его элементами. Эти понятия связаны между собой в соответствии с синтаксическими правилами. Каждая из программ, построенная по этим правилам, имеет соответствующую иерархическую структуру.

В связи с этим для всех языков и их программ можно дополнительно выделить следующие общие черты: каждый язык должен содержать правила, позволяющие порождать программы, соответствующие этому языку или распознавать соответствие между написанными программами и заданным языком.

*Связь структуры программы с языком программирования называется **синтаксическим отображением**.*

В качестве примера рассмотрим зависимость между иерархической структурой и цепочкой символов, определяющей следующее арифметическое выражение:

$$a + (b + c) * d$$

В большинстве языков программирования данное выражение определяет иерархию программных

объектов, которую можно отобразить в виде дерева (рис. 1.1.):

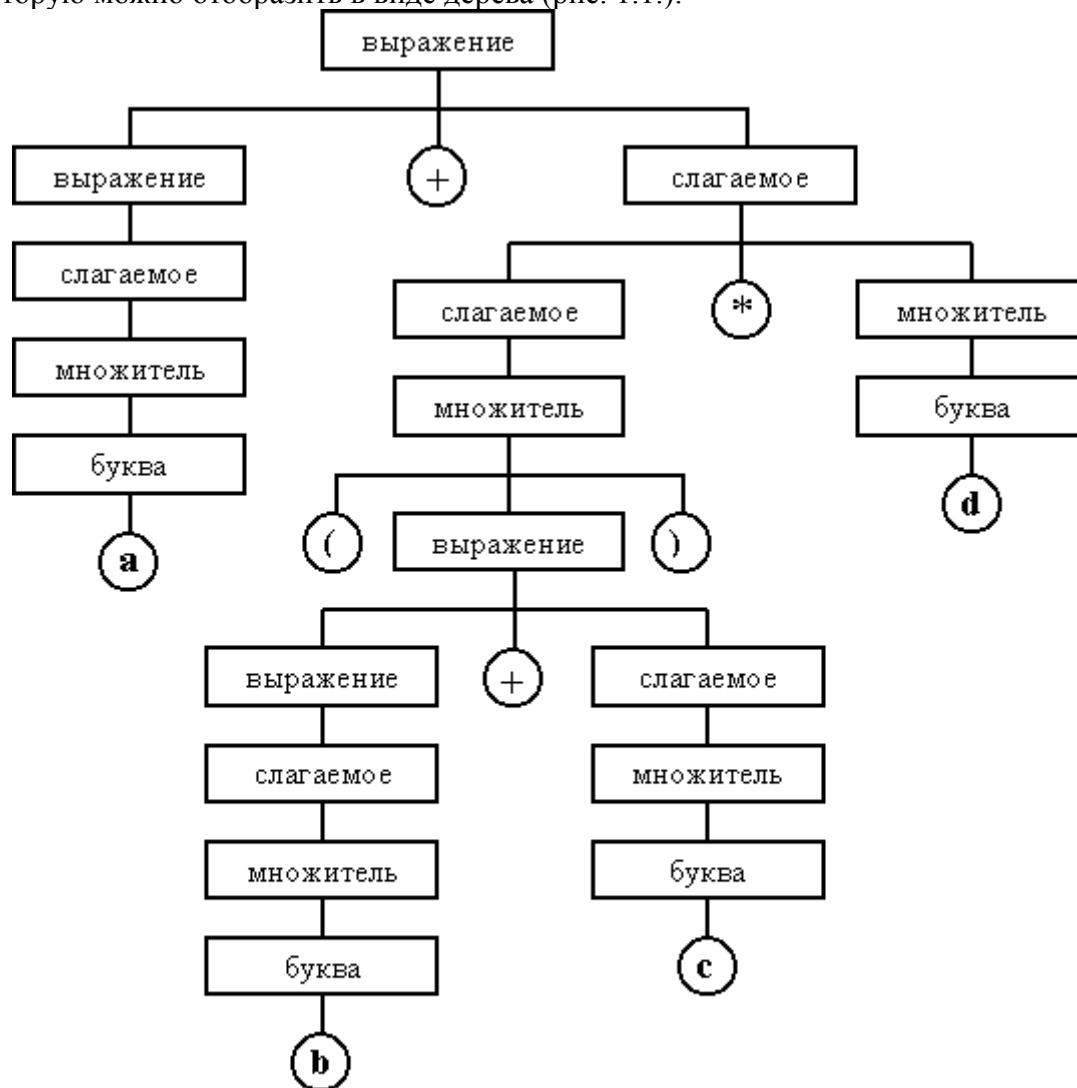


Рис. 1.1. Синтаксическая структура выражения  $a + (b + c) * d$   
построенная по первому описанию

В кружках представлены символы, используемые в качестве элементарных конструкций, а в прямоугольниках задаются составные понятия, имеющие иерархическую и, возможно, рекурсивную структуру. Эта иерархия определяется с помощью синтаксических правил, записанных на специальном языке, который называется метаязыком (подробнее метаязыки будут рассмотрены при изучении формальных грамматик):

```

<выражение> ::= <слагаемое> | <выражение> + <слагаемое>
<слагаемое> ::= <множитель> | <слагаемое> * <множитель>
<множитель> ::= <буква> | ( <выражение> )
<буква> ::= a | b | c | d | i | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

```

*Примечание.* Знак " $::=$ " читается как "это есть". Вертикальная черта "|" читается как "или".

Если правила будут записаны иначе, то изменится и иерархическая структура. В качестве примера можно привести следующие способ записи правил:

$$\begin{aligned} \langle \text{выражение} \rangle &::= \langle \text{операнд} \rangle \mid \langle \text{выражение} \rangle + \langle \text{операнд} \rangle \mid \langle \text{выражение} \rangle * \langle \text{операнд} \rangle \\ \langle \text{операнд} \rangle &::= \langle \text{буква} \rangle \mid ( \langle \text{выражение} \rangle ) \\ \langle \text{буква} \rangle &::= a \mid b \mid c \mid d \mid i \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z \end{aligned}$$

В результате синтаксического разбора того же арифметического выражения будет построена

иерархическая структура, представленная на рис. 1.2.

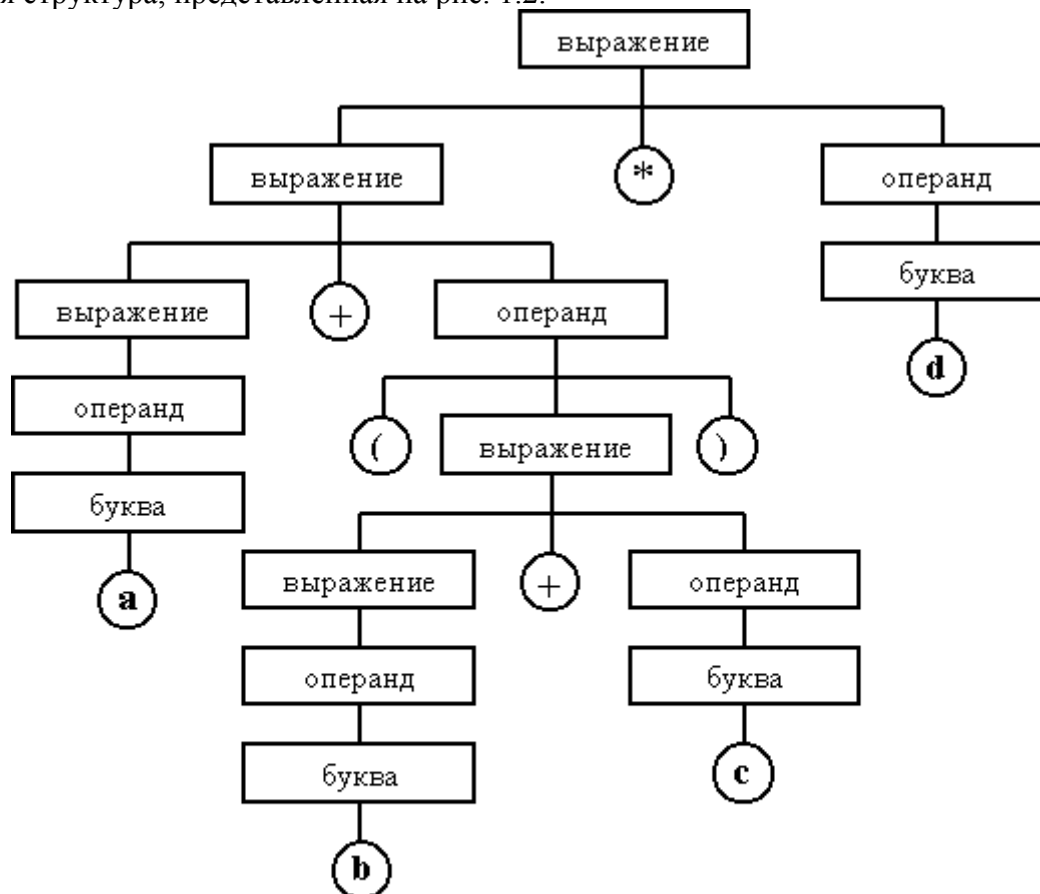


Рис. 1.2. Синтаксическая структура выражения  $a + (b + c) * d$  построенная по второму описанию

Следует отметить, что иерархическая структура в общем случае может быть никоим образом не связана с семантикой выражения. И в том и другом случае приоритет выполнения операций может быть реализован в соответствии с общепринятыми правилами, когда умножение предшествует сложению (или наоборот, все операции могут иметь одинаковый приоритет при любом наборе правил). Однако первая структура явно поддерживает дальнейшую реализацию общепринятого приоритета, тогда как вторая больше подходит для реализации операций с одинаковым приоритетом и их выполнению справа налево.

*Процесс нахождения синтаксической структуры заданной программы называется **синтаксическим разбором**.*

Синтаксическая структура, правильная для одного языка, может быть ошибочной для другого. Например, в языке Форт приведенное выражение не будет распознано. Однако для этого языка корректным будет являться постфиксное выражение:

$a \ b \ c \ + \ d \ * \ +$

Его синтаксическая структура описывается правилами:

$\langle \text{выражение} \rangle ::= \langle \text{буква} \rangle \mid \langle \text{операнд} \rangle \langle \text{операнд} \rangle \langle \text{операция} \rangle$

$\langle \text{операнд} \rangle ::= \langle \text{буква} \rangle \mid \langle \text{выражение} \rangle$

$\langle \text{операция} \rangle ::= + \mid *$

$\langle \text{буква} \rangle ::= a \mid b \mid c \mid d \mid i \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$

Иерархическое дерево, определяющее синтаксическую структуру, представлено на рис. 1.3.

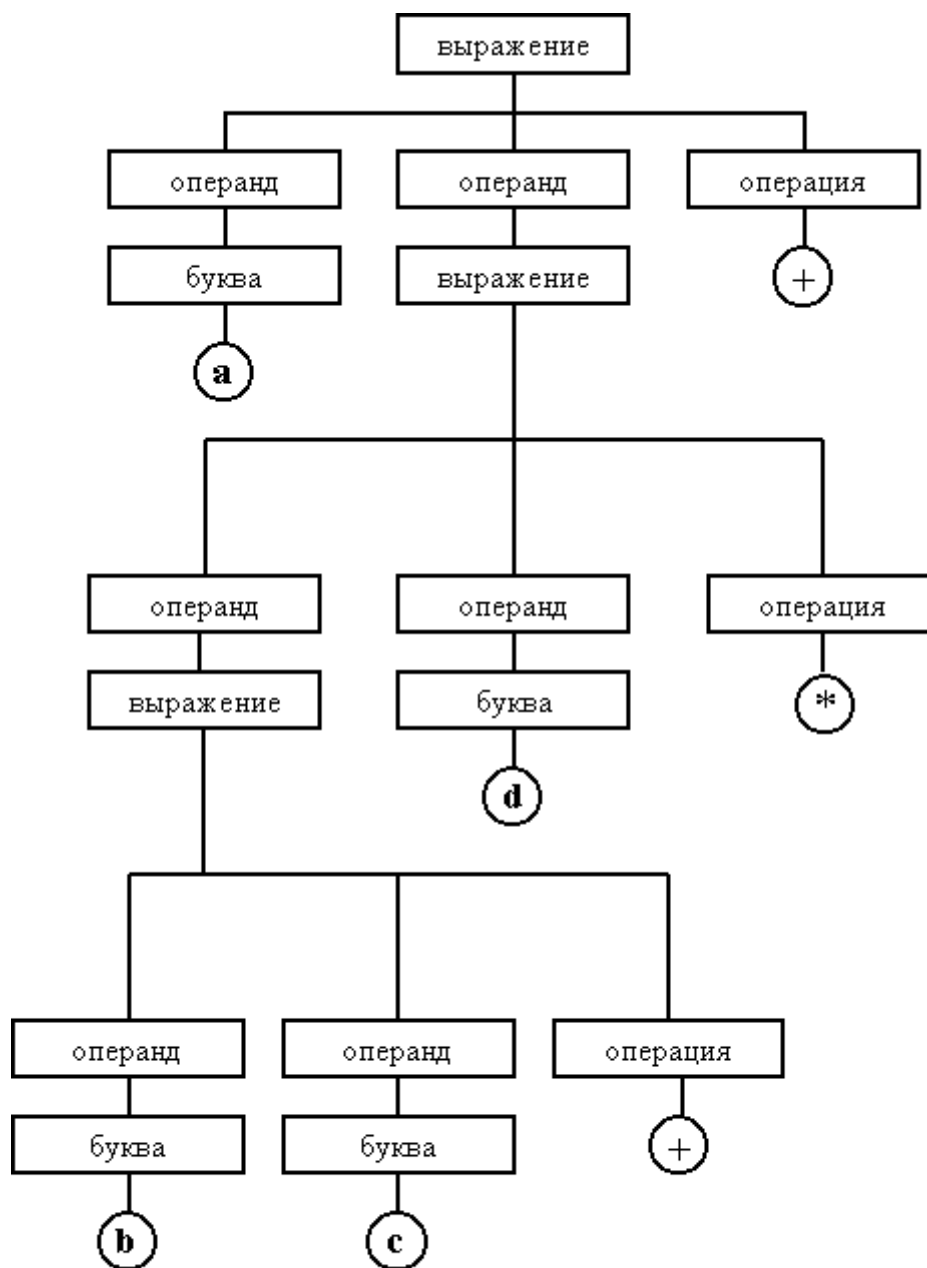


Рис. 1.3. Синтаксическая структура выражения  $a b c + d * +$ , построенная по постфиксным правилам

Другой характерной особенностью всех языков является их семантика. Она определяет смысл операций языка, корректность операндов. Цепочки, имеющие одинаковую синтаксическую структуру в различных языках программирования, могут различаться по семантике (что, например, наблюдается в C++, Pascal, Basic для приведенного выше фрагмента арифметического выражения).

Знание семантики языка позволяет отделить ее от его синтаксиса и использовать для преобразования в другой язык (осуществить генерацию кода).

Описание семантики и распознавание ее корректности обычно является самой трудоемкой и объемной частью транслятора, так как необходимо осуществить перебор и анализ множества вариантов допустимых комбинаций операций и операндов.



# Обобщенная структура транслятора

Общие свойства и закономерности присущи как различным языкам программирования, так и трансляторам с этих языков. В них протекают схожие процессы преобразования исходного текста. Не смотря на то, что взаимодействие этих процессов может быть организовано различным путем, можно выделить функции, выполнение которых приводит к одинаковым результатам. Назовем такие функции фазами процесса трансляции.

Учитывая схожесть компилятора и интерпретатора, рассмотрим фазы, существующие в компиляторе. В нем выделяются:

1. Фаза лексического анализа.
2. Фаза синтаксического анализа, состоящая из:
  - распознавания синтаксической структуры;
  - семантического разбора, в процессе которого осуществляется работа с таблицами, порождение промежуточного семантического представления или объектной модели языка.
3. Фаза генерации кода, осуществляющая:
  - семантический анализ компонент промежуточного представления или объектной модели языка;
  - перевод промежуточного представления или объектной модели в объектный код.

Наряду с основными фазами процесса трансляции возможны также дополнительные фазы:

2а. Фаза исследования и оптимизации промежуточного представления, состоящая из:

- 2а.1. анализа корректности промежуточного представления;
- 2а.2. оптимизации промежуточного представления.

3а. Фаза оптимизации объектного кода.

Интерпретатор отличается тем, что фаза генерации кода обычно заменяется фазой эмуляции элементов промежуточного представления или объектной модели языка. Кроме того, в интерпретаторе обычно не проводится оптимизация промежуточного представления, а сразу же осуществляется его эмуляция.

Кроме этого можно выделить единый для всех фаз процесс анализа и исправление ошибок, существующих в обрабатываемом исходном тексте программы.

Обобщенная структура компилятора, учитывающая существующие в нем фазы, представлена на рис. 1.4.

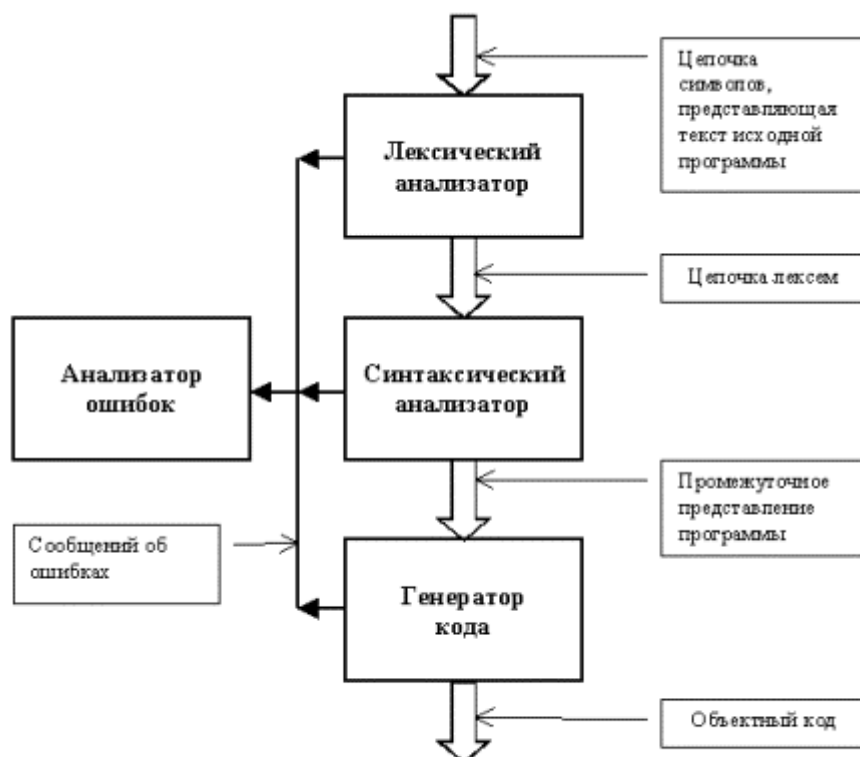


Рис. 1.4. Обобщенная структура компилятора.

Он состоит из лексического анализатора, синтаксического анализатора, генератора кода, анализатора ошибок. В интерпретаторе вместо генератора кода используется эмулятор (рис. 1.5), в который, кроме элементов промежуточного представления, передаются исходные данные. На выход эмулятора выдается результат вычислений.

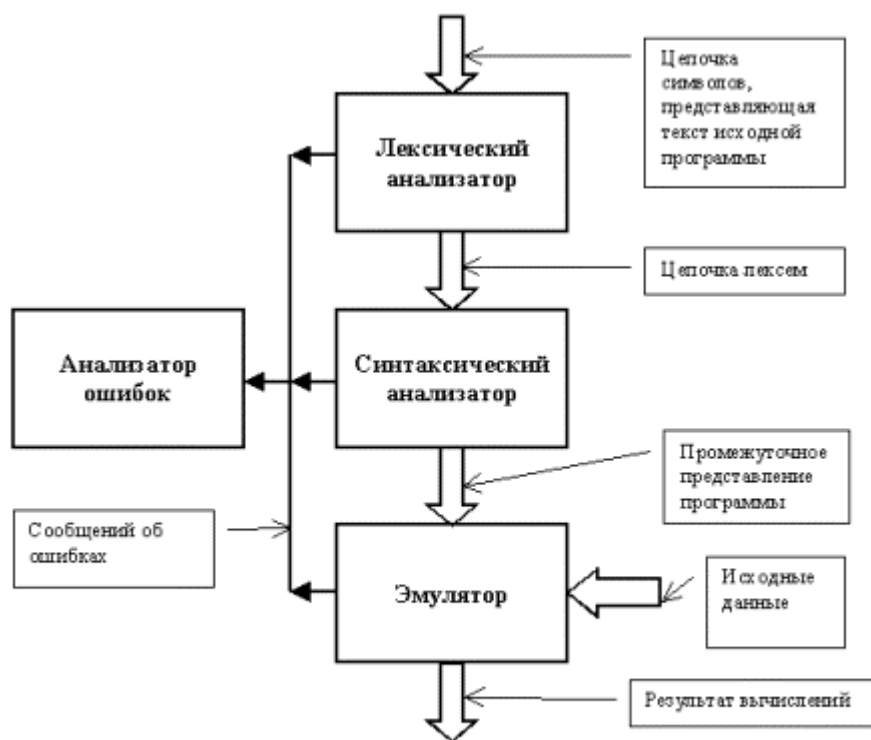


Рис. 1.5. Обобщенная структура интерпретатора.

Лексический анализатор (известен также как сканер) осуществляет чтение входной цепочки символов и их группировку в элементарные конструкции, называемые лексемами. Каждая лексема имеет класс и значение. Обычно претендентами на роль лексем выступают элементарные конструкции языка, например, идентификатор, действительное число, комментарий. Полученные лексемы передаются синтаксическому анализатору. Сканер не является обязательной частью транслятора. Однако, он позволяет повысить эффективность процесса трансляции. Подробнее лексический анализ рассмотрен в теме: "Организация лексического анализа".

Синтаксический анализатор осуществляет разбор исходной программы, используя поступающие лексемы, построение синтаксической структуры программы и семантический анализ с формированием объектной модели языка. Объектная модель представляет синтаксическую структуру, дополненную семантическими связями между существующими понятиями. Этими связями могут быть:

- ссылки на переменные, типы данных и имена процедур, размещаемые в таблицах имен;
- связи, определяющие последовательность выполнения команд;
- связи, определяющие вложенность элементов объектной модели языка и другие.

Таким образом, синтаксический анализатор является достаточно сложным блоком транслятора. Поэтому его можно разбить на следующие составляющие:

- распознаватель;
- блок семантического анализа;
- объектную модель, или промежуточное представление, состоящие из таблицы имен и синтаксической структуры.

Обобщенная структура синтаксического анализатора приведена на рис. 1.6.



Рис. 1.6. Обобщенная схема синтаксического анализатора

Распознаватель получает цепочку лексем и на ее основе осуществляет разбор в соответствии с используемыми правилами. Лексемы, при успешном разборе правил, передаются семантическому анализатору, который строит таблицу имен и фиксирует фрагменты синтаксической структуры. Кроме этого, между таблицей имен и синтаксической структурой фиксируются дополнительные семантические связи. В результате формируется объектная модель программы, освобожденная от привязки к синтаксису языка программирования. Достаточно часто вместо синтаксической структуры, полностью копирующей иерархию объектов языка, создается ее упрощенный аналог, который называется промежуточным представлением.

Анализатор ошибок получает информацию об ошибках, возникающих в различных блоках транслятора. Используя полученную информацию, он формирует сообщения пользователю. Кроме этого, данный блок может попытаться исправить ошибку, чтобы продолжить разбор дальше. На него также возлагаются действия, связанные с корректным завершением программы в случае, когда дальнейшую трансляцию продолжать невозможно.

Генератор кода строит код объектной машины на основе анализа объектной модели или промежуточного представления. Построение кода сопровождается дополнительным семантическим анализом, связанным с необходимостью преобразования обобщенных команд в коды конкретной вычислительной машины. На этапе такого анализа окончательно определяется возможность преобразования, и выбираются эффективные варианты. Сама генерация кода является перекодировкой одних команд в другие.

## Варианты взаимодействия блоков транслятора

Организация процессов трансляции, определяющая реализацию основных фаз, может осуществляться различным образом. Это определяется различными вариантами взаимодействия блоков транслятора: лексического анализатора, синтаксического анализатора и генератора кода. Несмотря на одинаковый конечный результат, различные варианты взаимодействия блоков транслятора обеспечивают различные варианты хранения промежуточных данных. Можно выделить два основных варианта взаимодействия блоков транслятора:

- многопроходную организацию, при которой каждая из фаз является независимым процессом, передающим управление следующей фазе только после окончания полной обработки своих данных;
- однопроходную организацию, при которой все фазы представляют единый процесс и передают друг другу данные небольшими фрагментами.

На основе двух основных вариантов можно также создавать их разнообразные сочетания.

### Многопроходная организация взаимодействия блоков транслятора

Данный вариант взаимодействия блоков, на примере компилятора, представлен на рис 1.7.

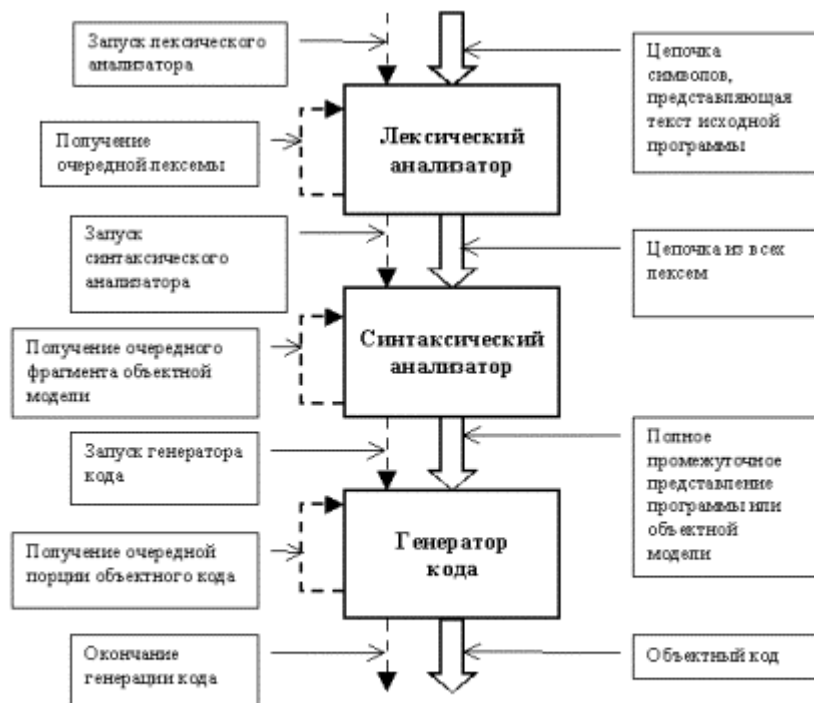


Рис. 1.7. Многопроходное взаимодействие блоков компилятора.

Лексический анализатор полностью обрабатывает исходный текст, формируя на выходе цепочку, состоящую из всех полученных лексем. Только после этого управление передается синтаксическому анализатору. Синтаксический анализатор получает сформированную цепочку лексем и на ее основе формирует промежуточное представление или объектную модель. После получения всей объектной модели он передает управление генератору кода. Генератор кода, на основе объектной модели языка, строит требуемый машинный код. К достоинствам такого подхода можно отнести:

1. Обособленность отдельных фаз, что позволяет обеспечить их независимую друг от друга реализацию и использование.
2. Возможность хранения данных, получаемых в результате работы каждой из фаз, на внешних запоминающих устройствах и их использования по мере надобности.
3. Возможность уменьшения объема оперативной памяти, требуемой для работы транслятора, за счет последовательного вызова фаз.

К недостаткам следует отнести.

1. Наличие больших объемов промежуточной информации, из которой в данный момент времени требуется только небольшая часть.
2. Замедление скорости трансляции из-за последовательного выполнения фаз и использования для экономии оперативной памяти внешних запоминающих устройств.

Данный подход может оказаться удобным при построении трансляторов с языков программирования, обладающей сложной синтаксической и семантической структурой (например, PL/I). В таких ситуациях трансляцию сложно осуществить за один проход, поэтому результаты предыдущих проходов проще представлять в виде дополнительных промежуточных данных. Следует отметить, что сложные семантическая и синтаксическая структуры языка могут привести к дополнительным проходам, необходимым для установления требуемых зависимостей. Общее количество проходов может оказаться более десяти. На число проходов может также влиять использование в программе конкретных возможностей языка, таких как объявление переменных и процедур после их использования, применение правил объявления по умолчанию и т. д.

### Однопроходная организация взаимодействия блоков транслятора

Один из вариантов взаимодействия блоков компилятора при однопроходной организации представлено на рис. 1.8.

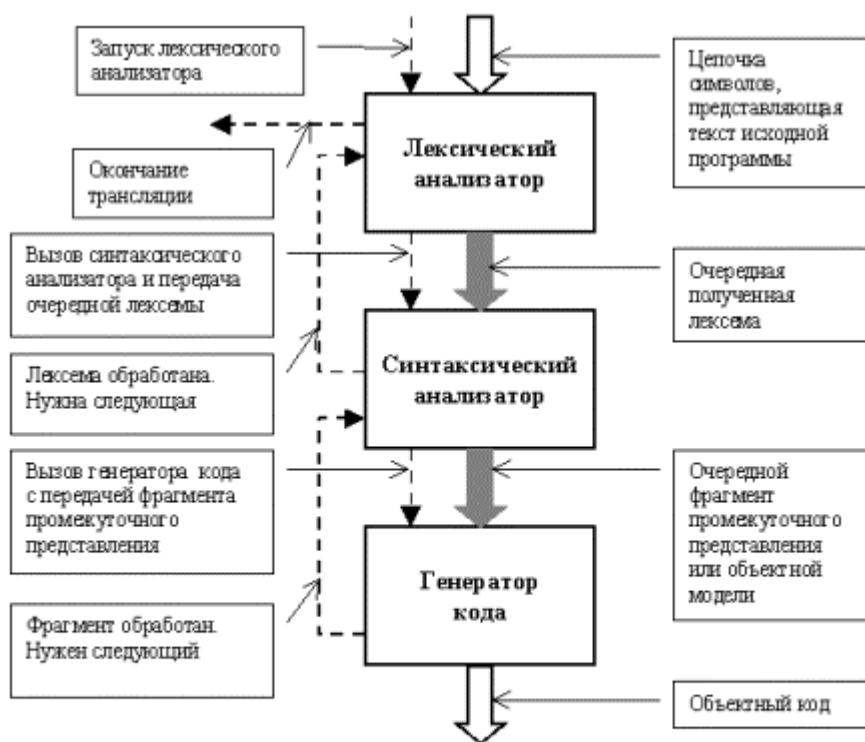


Рис. 1.8. Однопроходное взаимодействие блоков компилятора при управлении, инициируемом лексическим анализатором.

В этом случае процесс трансляции протекает следующим образом. Лексический анализатор читает фрагмент исходного текста, необходимый для получения одной лексемы. После формирования лексемы им осуществляется вызов синтаксического анализатора и передача ему созданной лексемы в качестве параметра. Если синтаксический анализатор может построить очередной элемент промежуточного представления, то он делает это и передает построенный фрагмент генератору кода. В противном случае синтаксический анализатор возвращает управление сканеру, давая, тем самым, понять, что очередная лексема обработана и нужны новые данные.

Генератор кода функционирует аналогичным образом. По полученному фрагменту промежуточного представления он создает соответствующий фрагмент объектного кода. После этого управление возвращается синтаксическому анализатору.

По окончании исходного текста и завершении обработки всех промежуточных данных каждым из блоков лексический анализатор инициирует процесс завершения программы.

Чаще всего в однопроходных трансляторах используется другая схема управления, в которой роль основного блока играет синтаксический анализатор (рис. 1.9).

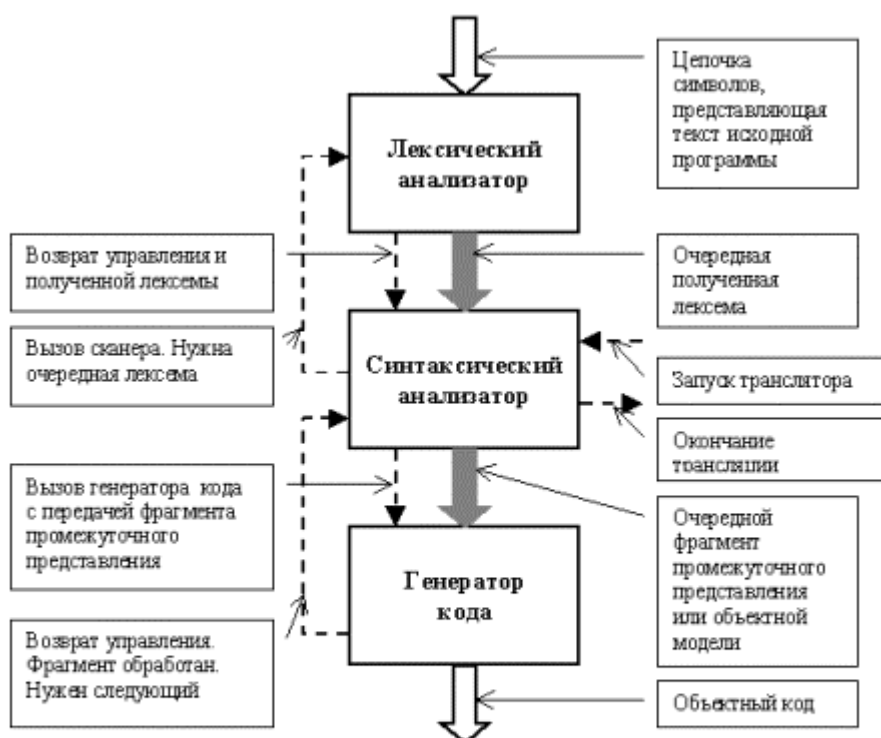


Рис. 1.9. Однопроходное взаимодействие блоков компилятора при управлении, инициируемом синтаксическим анализатором

Лексический анализатор и генератор кода выступают в роли вызываемых им подпрограмм. Как только синтаксическому анализатору нужна очередная лексема, он вызывает сканер. При получении фрагмента промежуточного представления осуществляется обращение к генератору кода. Завершение процесса трансляции происходит после получения и обработки последней лексемы и инициируется синтаксическим анализатором.

К достоинствам однопроходной схемы следует отнести отсутствие больших объемов промежуточных данных, высокую скорость обработки из-за совмещения фаз в едином процессе и отсутствие обращений в внешним запоминающим устройствам.

К недостаткам относятся: невозможность реализации такой схемы трансляции для сложных по структуре языков и отсутствие промежуточных данных, которые можно использовать для комплексного анализа и оптимизации.

Такая схема часто применяется для простых по семантической и синтаксической структурам языков программирования, как в компиляторах, так и в интерпретаторах. Примерами таких языков могут служить Basic и Pascal. Классический интерпретатор обычно строится по однопроходной схеме, так как непосредственное исполнение осуществляется на уровне отдельных фрагментов промежуточного представления.

Организация взаимодействия блоков такого интерпретатора представлена на рис. 1.10.



Рис. 1.10. Однопроходное взаимодействие блоков интерпретатора.

## Комбинированные взаимодействия блоков транслятора

Сочетания многопроходной и однопроходной схем трансляции порождают разнообразные комбинированные варианты, многие из которых успешно используются. В качестве примера можно рассмотреть некоторые из них.

На рис. 1.11 представлена схема взаимодействия блоков транслятора, разбивающая весь процесс на два прохода. На первом проходе порождается полное промежуточное представление программы, а на втором осуществляется генерация кода. Использование такой схемы позволяет легко переносить транслятор с одной вычислительной системы на другую путем переписывания генератора кода.

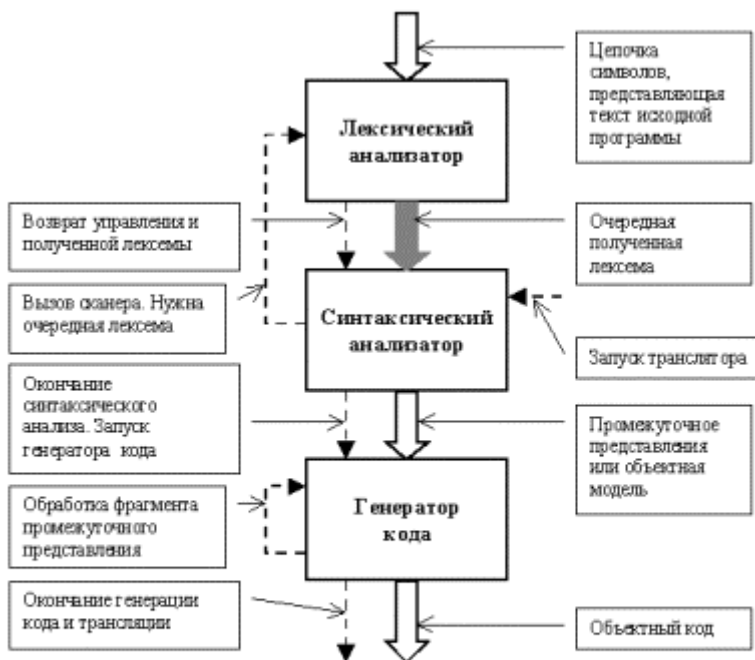


Рис. 1.11. Двухпроходная схема с генерацией полного промежуточного представления.

Кроме этого, вместо генератора кода легко подключить эмулятор промежуточного представления, что достаточно просто позволяет разработать систему программирования на некотором языке, ориентированную на различные среды исполнения. Пример подобной организации взаимодействия блоков транслятора представлен на рис. 1.12.



Рис. 1.12. Эмуляция промежуточного представления.

Наряду со схемами, предполагающими замену генератора кода на эмулятор, существуют трансляторы, допускающие их совместное использование. Одна из таких схем представлена на рис. 1.13.

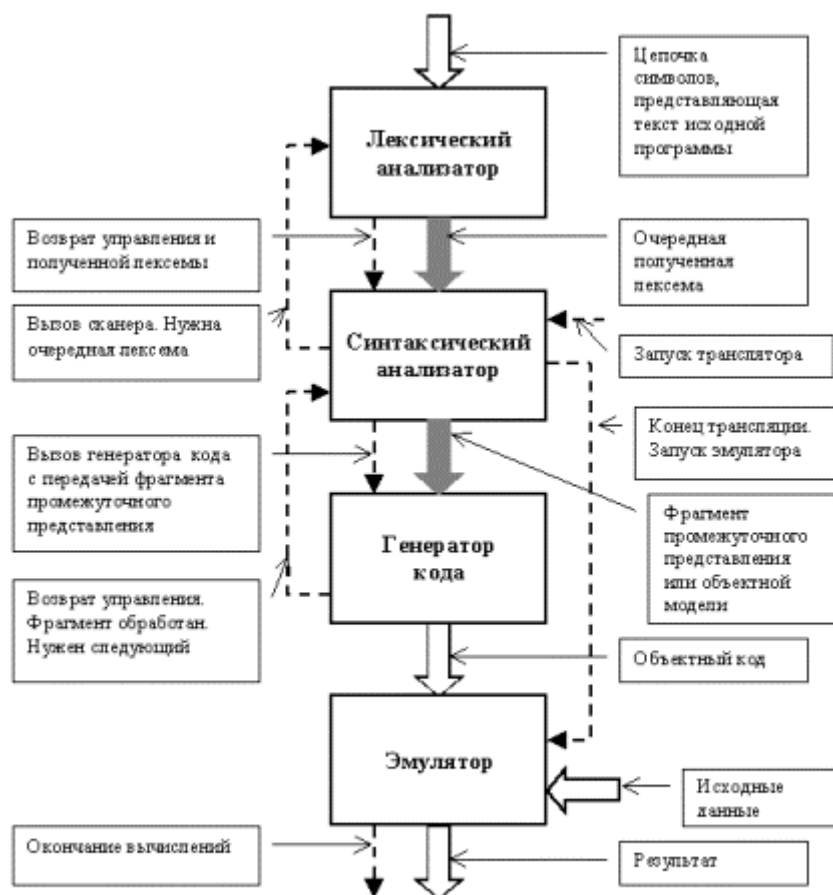


Рис. 1.13. Эмуляция скомпилированного объектного кода.



Процесс трансляции, включая и генерацию кода, может быть выполнен за любое число проходов (в примере используется однопроходная трансляция, представленная ранее на [рис 1.9](#)).

Однако сформированный объектный код не исполняется на соответствующей ему вычислительной системе, а эмулируется на компьютере с другой архитектурой. Такая схема применяется в среде построенной вокруг языка программирования Java. Сам транслятор генерирует код виртуальной Java-машины, эмуляция которого осуществляется специальными средствами как автономно, так и в среде Internet браузера.

Представленная схема может иметь и более широкое толкование применительно к любому компилятору, порождающему машинный код. Все дело в том, что большинство современных вычислительных машин реализованы с использованием микропрограммного управления. А микропрограммное устройство управления можно рассматривать как программу, эмулирующую машинный код. Это позволяет говорить о повсеместном использовании последней представленной схемы.

## **Контрольные вопросы и задания**

1. Назовите отличия:

- интерпретатора от компилятора;
- компилятора от ассемблера;
- перекодировщика от транслятора;
- эмулятора от интерпретатора;
- синтаксиса от семантики.

2. Расскажите об известных Вам последних разработках языков программирования. Приведите основные характеристики названных языков.

3. Приведите конкретные примеры использования методов трансляции в областях, не связанных с языками программирования.

4. Приведите конкретные примеры компилируемых языков программирования.

5. Приведите конкретные примеры интерпретируемых языков программирования.

6. Приведите конкретные примеры языков программирования, для которых имеются как компиляторы, так и интерпретаторы.

7. Основные достоинства и недостатки компиляторов.

8. Основные достоинства и недостатки интерпретаторов.

9. Опишите основные различия в синтаксисе двух известных Вам языков программирования.

10. Опишите основные различия в семантике двух известных Вам языков программирования.

11. Назовите основные фазы процесса трансляции и их назначение.

12. Назовите специфические особенности однопроходной трансляции.

13. Назовите специфические особенности многопроходной трансляции.

14. Приведите примеры возможных комбинаций однопроходной и многопроходной трансляции.

Расскажите о практическом использовании этих схем.

# Тема 2. Основы теории языков и формальных грамматик

## Содержание темы

Способы определения языков. Формальные грамматики. Грамматики с ограничениями на правила. Способы записи синтаксиса языка. Распознаватели. Контрольные вопросы.

## Способы определения языков

Описание языков программирования во многом опирается на теорию формальных языков. Эта теория является фундаментом для организации синтаксического анализа и перевода. Существует два основных способа определения языков:

- механизм порождения или генератор;
- механизм распознавания или распознаватель.

Они тесно связаны. Первый обычно используется для описания языков, а второй для их реализации. Оба способа позволяют описать языки конечным образом, несмотря на бесконечное число порождаемых ими цепочек.

Неформально язык  $L$  - это множество цепочек конечной длины в алфавите  $T$ . Механизм порождения позволяет описать языки с помощью системы правил, называемой грамматикой. Цепочки (предложения) языка строятся в соответствии с этими правилами. Достоинство определения языка с помощью грамматик в том, что операции, производимые в ходе синтаксического анализа и перевода, можно делать проще, если воспользоваться структурой, предписываемой цепочкам с помощью этих грамматик.

Механизм распознавания использует алгоритм, который для произвольной входной цепочки остановится и ответит "да" после конечного числа шагов, если эта цепочка принадлежит языку. Если цепочка не принадлежит языку, алгоритм ответит "нет". Распознаватели используются непосредственно при построении синтаксических анализаторов и являются как бы их формальной моделью. Распознаватели строятся на основе теорий конечных автоматов и автоматов с магазинной памятью.

## Формальные грамматики

Грамматикой называется четверка  $G = (N, T, P, S)$ , где  $N$  - конечное множество нетерминальных символов (нетерминалов),  $T$  - множество терминалов (не пересекающихся с  $N$ ),  $S$  - символ из  $N$ , называемый начальным,  $P$  - конечное подмножество множества:

$$(N \cup T)^* N (N \cup T)^* x (N \cup T)^*,$$

называемое множеством правил. Множество правил  $P$  описывает процесс порождения цепочек языка. Элемент  $p_i = (\alpha, \beta)$  множества  $P$  называется правилом (продукцией) и записывается в виде  $\alpha \Rightarrow \beta$ . Здесь  $\alpha$  и  $\beta$  - цепочки, состоящие из терминалов и нетерминалов. Данная запись может читаться одним из следующих способов:

- цепочка  $\alpha$  порождает цепочку  $\beta$ ;
- из цепочки  $\alpha$  выводится цепочка  $\beta$ .

Таким образом, правило  $P$  имеет две части: левую, определяемую, и правую, подставляемую. То есть правило  $p_i$  - это двойка  $(p_{i1}, p_{i2})$ , где  $p_{i1} = (N \cup T)^* N (N \cup T)^*$  - цепочка, содержащая хотя бы один нетерминал,  $p_{i2} = (N \cup T)^*$  - произвольная, возможно пустая цепочка ( $\epsilon$  - цепочка).

Если цепочка  $\alpha$  содержит  $p_{i1}$ , то, в соответствии с правилом  $p_i$ , можно образовать новую цепочку  $\beta$ , заменив одно вхождение  $p_{i1}$  на  $p_{i2}$ . Говорят также, что цепочка  $\beta$  выводится из  $\alpha$  в данной грамматике.

Для описания абстрактных языков в определениях и примерах будем пользоваться следующими обозначениями:

- терминалы обозначим буквами a, b, c, d или цифрами 0, 1, ..., 9;
- нетерминалы будем обозначать буквами A, B, C, D, S (причем нетерминал S - начальный символ грамматики);
- буквы U, V, ..., Z используем для обозначения отдельных терминалов или нетерминалов;
- через  $\alpha, \beta, \gamma$ ... обозначим цепочки терминалов и нетерминалов;
- u, v, w, x, y, z - цепочки терминалов;
- для обозначения пустой цепочки (не содержащей ни одного символа) будем использовать знак  $\epsilon$ ;
- знак " $\rightarrow$ " будет отделять левую часть правила от правой и читаться как "порождает" или "есть по определению". Например,  $A \rightarrow cd$ , читается как "A порождает cd".

Эти обозначения определяют некоторый язык, предназначенный для описания правил построения цепочек, а значит, для описания других языков. Язык, предназначенный для описания другого языка, называется **метаязыком**.

### Пример грамматики G1:

$G1 = (\{A, S\}, \{0, 1\}, P, S)$ ,

где P:

1.  $S \rightarrow 0A1$ ;
1.  $0A \rightarrow 00A1$ ;
2.  $A \rightarrow \epsilon$ .

Выводимая цепочка грамматики **G**, не содержащая нетерминалов, называется **терминальной цепочкой**, порождаемой грамматикой **G**.

Язык **L(G)**, порождаемый грамматикой **G**, - это множество терминальных цепочек, порождаемых грамматикой **G**.

Введем отношение  $\Rightarrow_G$  непосредственного вывода на множестве  $(N \cup T)^*$ , которое будем записывать следующим образом:

$$\Phi \Rightarrow_G \Psi.$$

Данная запись читается:  $\Psi$  непосредственно выводима из  $\Phi$  для грамматики  $G = (N, T, P, S)$  и означает: если  $\alpha\beta\gamma$  - цепочка из множества  $(N \cup T)^*$  и  $\beta \rightarrow \delta$  - правило из **P** то  $\alpha\beta\gamma \Rightarrow_G \alpha\delta\gamma$ .

Через  $\Rightarrow_G^+$  обозначим транзитивное замыкание (нетривиальный вывод за один и более шагов). Тогда

$\Phi \Rightarrow_G^+ \Psi$  читается как:  $\Psi$  выводима из  $\Phi$  нетривиальным образом.

Через  $\Rightarrow_G^*$  - обозначим рефлексивное и транзитивное замыкание (вывод за ноль и более шагов). Тогда

$\Phi \Rightarrow_G^* \Psi$  означает:  $\Psi$  выводима из  $\Phi$ .

Пусть  $\Rightarrow^k$  k - я степень отношения  $\Rightarrow$ . То есть, если  $\alpha \Rightarrow^k \beta$ , то существует последовательность  $\alpha_0\alpha_1\alpha_2\alpha_3 \dots \alpha_k$  из k+1 цепочек

$$\alpha = \alpha_0, \alpha_1, \dots, \alpha_{i-1} \Rightarrow \alpha_i, 1 \leq i \leq k \text{ и } \alpha_k = \beta.$$

Пример выводов для грамматики G1:

$S \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 0011$ ;

$S \Rightarrow^1 0A1$ ;  $S \Rightarrow^2 00A11$ ;  $S \Rightarrow^3 0011$ ;

$S \Rightarrow^+ 0A1$ ;  $S \Rightarrow^+ 00A11$ ;  $S \Rightarrow^+ 0011$ ;

$S \Rightarrow^* S$ ;  $S \Rightarrow^* 0A1$ ;  $S \Rightarrow^* 00A11$ ;  $S \Rightarrow^* 0011$ ;

где  $0011 \subset L(G1)$ .

# Грамматики с ограничениями на правила

Несмотря на большое разнообразие грамматик, при построении трансляторов нашли широкое применение только ряд из них, имеющих некоторые ограничения. Это связано с практической целесообразностью использования определенных типов правил, так как сложность их построения непосредственно влияет на сложность построения трансляторов. По виду правил выделяют несколько классов грамматик. В соответствии с классификацией Хомского грамматика **G** называется:

- **праволинейной**, если каждое правило из **P** имеет вид:  $A \rightarrow xB$  или  $A \rightarrow x$ , где **A**, **B** - нетерминалы, **x** - цепочка, состоящая из терминалов;
- **контекстно-свободной** (КС) или **бесконтекстной**, если каждое правило из **P** имеет вид:  $A \rightarrow \alpha$ , где  $A \in N$ , а  $\alpha \in (N \cup T)^*$ , то есть является цепочкой, состоящей из множества терминалов и нетерминалов, возможно пустой;
- **контекстно-зависимой** или **неукорачивающей**, если каждое правило из **P** имеет вид:  $\alpha \rightarrow \beta$ , где  $|\alpha| \leq |\beta|$ . То есть, вновь порождаемые цепочки не могут быть короче, чем исходные, а, значит, и пустыми (другие ограничения отсутствуют);
- **грамматикой свободного вида**, если в ней отсутствуют выше упомянутые ограничения.

Пример праволинейной грамматики:

$G_2 = (\{S\}, \{0,1\}, P, S)$ , где

P:

1.  $S \rightarrow 0S$ ;
2.  $S \rightarrow 1S$ ;
3.  $S \rightarrow \epsilon$ ,

определяет язык  $\{0, 1\}^*$ .

Пример КС-грамматики:

$G_3 = (\{E, T, F\}, \{a, +, *, ()\}, P, E)$  где

P:

1.  $E \rightarrow T$
2.  $E \rightarrow E + T$
3.  $T \rightarrow F$
4.  $T \rightarrow T * F$
5.  $F \rightarrow (E)$
6.  $F \rightarrow a$ .

Данная грамматика порождает простейшие арифметические выражения.

Пример КЗ-грамматики:

$G_4 = (\{B, C, S\}, \{a, b, c\}, P, S)$  где

P:

1.  $S \rightarrow aSBC$ ;
2.  $S \rightarrow abc$ ;
3.  $CB \rightarrow BC$ ;
4.  $bB \rightarrow bb$ ;
5.  $bC \rightarrow bc$ ;
6.  $cC \rightarrow cc$ ,

порождает язык  $\{a^n b^n c^n\}, n \geq 1$ .

Примечание 1. Согласно определению каждая праволинейная грамматика является контекстно-свободной.

Примечание 2. По определению КЗ-грамматика не допускает правил:  $A \rightarrow \epsilon$ , где  $\epsilon$  - пустая цепочка. Т.е. КС-грамматика с пустыми цепочками в правой части правил не является контекстно-зависимой. Наличие пустых цепочек ведет к грамматике без ограничений.

Соглашение. Если язык L порождается грамматикой типа G, то L называется языком типа G.

Пример: L(G3) - КС язык типа G3.

Наиболее широкое применение при разработке трансляторов нашли КС-грамматики и порождаемые ими КС языки. В процессе изучения КС языков остановимся только на тех, которые будут полезны для нас с практической точки зрения (теория языков обширна и для детального ее изучения необходимо много времени). Те, кто желает приобрести более глубокие познания в данной области, могут обратиться к монографии Ахо и Ульмана [[Ахо78](#)].

## Способы записи синтаксиса языка

Существуют различные способы записи синтаксических правил, что в основном определяется условными обозначениями и ограничениями на структуру правил, принятыми в используемых метаязыках. Метаязыки используются для задания грамматики языков программирования со времен Алгола 60. Еще раньше они начали использоваться при описании небольших языков в статьях, посвященных формальным грамматикам. Кратко рассмотрим основные вехи становления и развития метаязыков. Во всех случаях будем определять идентификатор.

### Метаязык Хомского

Метаязык Хомского вышел из недр математической логики. Он имеет следующую систему обозначений:

- символ " $\rightarrow$ " отделяет левую часть правила от правой (читается как "порождает" и "это есть");
- нетерминалы обозначаются буквой **A** с индексом, указывающим на его номер;
- терминалы - это символы используемые в описываемом языке;
- каждое правило определяет порождение одной новой цепочки, причем один и тот же нетерминал может встречаться в нескольких правилах слева.

Описание идентификатора на метаязыке Хомского будет выглядеть следующим образом:

1. $A_1 \rightarrow A$	23. $A_1 \rightarrow W$	45. $A_1 \rightarrow s$
2. $A_1 \rightarrow B$	24. $A_1 \rightarrow X$	46. $A_1 \rightarrow t$
3. $A_1 \rightarrow C$	25. $A_1 \rightarrow Y$	47. $A_1 \rightarrow u$
4. $A_1 \rightarrow D$	26. $A_1 \rightarrow Z$	48. $A_1 \rightarrow v$
5. $A_1 \rightarrow E$	27. $A_1 \rightarrow a$	49. $A_1 \rightarrow w$
6. $A_1 \rightarrow F$	28. $A_1 \rightarrow b$	50. $A_1 \rightarrow x$
7. $A_1 \rightarrow G$	29. $A_1 \rightarrow c$	51. $A_1 \rightarrow y$
8. $A_1 \rightarrow H$	30. $A_1 \rightarrow d$	52. $A_1 \rightarrow z$
9. $A_1 \rightarrow I$	31. $A_1 \rightarrow e$	53. $A_2 \rightarrow 0$
10. $A_1 \rightarrow J$	32. $A_1 \rightarrow f$	54. $A_2 \rightarrow 1$
11. $A_1 \rightarrow K$	33. $A_1 \rightarrow g$	55. $A_2 \rightarrow 2$

12. $A_1 \rightarrow L$	34. $A_1 \rightarrow h$	56. $A_2 \rightarrow 3$
13. $A_1 \rightarrow M$	35. $A_1 \rightarrow i$	57. $A_2 \rightarrow 4$
14. $A_1 \rightarrow N$	36. $A_1 \rightarrow j$	58. $A_2 \rightarrow 5$
15. $A_1 \rightarrow O$	37. $A_1 \rightarrow k$	59. $A_2 \rightarrow 6$
16. $A_1 \rightarrow P$	38. $A_1 \rightarrow l$	60. $A_2 \rightarrow 7$
17. $A_1 \rightarrow Q$	39. $A_1 \rightarrow m$	61. $A_2 \rightarrow 8$
18. $A_1 \rightarrow R$	40. $A_1 \rightarrow n$	62. $A_2 \rightarrow 9$
19. $A_1 \rightarrow S$	41. $A_1 \rightarrow o$	63. $A_3 \rightarrow A_1$
20. $A_1 \rightarrow T$	42. $A_1 \rightarrow p$	64. $A_3 \rightarrow A_3 A_1$
21. $A_1 \rightarrow U$	43. $A_1 \rightarrow q$	65. $A_3 \rightarrow A_3 A_2$
22. $A_1 \rightarrow V$	44. $A_1 \rightarrow r$	

### Метаязык Хомского-Щутценберже

Приведенный в предыдущем разделе пример описания идентификатора показывает громоздкость метаязыка Хомского, что позволяет эффективно использовать его только для описания небольших абстрактных языков. Более компактное описание возможно с применением метаязыка Хомского-Щутценберже, использующего следующие обозначения метасимволов:

- символ “=” отделяет левую часть правила от правой (вместо символа “ $\rightarrow$ ”);
- нетерминалы обозначаются буквой A с индексом, указывающим на его номер;
- терминалы - это символы используемые в описываемом языке;
- каждое правило определяет порождение нескольких альтернативных цепочек, отделяемых друг от друга символом “+”, что позволяет, при желании, использовать в левой части только разные нетерминалы.

Введение возможности альтернативного перечисления позволило сократить описание языков. Описание идентификатора будет выглядеть следующим образом:

1.  $A_1 = A+B+C+D+E+F+G+H+I+J+K+L+M+N+O+P+Q+R+S+T+U+V+W+X+Y+Z+a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z$
2.  $A_2 = 0+1+2+4+5+6+7+8+9$
3.  $A_3 = A_1 + A_3 A_1 + A_3 A_2$

### Бэкуса-Наура формы (БНФ)

Метаязыки Хомского и Хомского-Щутценберже использовались в математической литературе при описании простых абстрактных языков. Метаязык, предложенный Бэкусом и Науром, впервые

использовался для описания синтаксиса реального языка программирования Алгол 60. Наряду с новыми обозначениями метасимволов, в нем использовались содержательные обозначения нетерминалов. Это сделало описание языка нагляднее и позволило в дальнейшем широко использовать данную нотацию для описания реальных языков программирования. Были использованы следующие обозначения:

- символ "::<=" отделяет левую часть правила от правой;
- нетерминалы обозначаются произвольной символьной строкой, заключенной в угловые скобки "<" и ">;
- терминалы - это символы, используемые в описываемом языке;
- каждое правило определяет порождение нескольких альтернативных цепочек, отделяемых друг от друга символом вертикальной черты "|".

Пример описания идентификатора с использованием БНФ:

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

Правила можно задавать и раздельно:

3. <идентификатор> ::= <буква>
4. <идентификатор> ::= <идентификатор> <буква>
5. <идентификатор> ::= <идентификатор> <цифра>

## Расширенные Бэкуса-Наура формы (РБНФ)

Метаязыки, представленные выше, позволяют описывать любой синтаксис. Однако, для повышения удобства и компактности описания, целесообразно вести в язык дополнительные конструкции. В частности, специальные метасимволы были разработаны для описания необязательных цепочек, повторяющихся цепочек, обязательных альтернативных цепочек. Существуют различные расширенные формы метаязыков, незначительно отличающиеся друг от друга. Их разнообразие зачастую объясняется желанием разработчиков языков программирования по-своему описать создаваемый язык. К примерам таких широко известных метаязыков можно отнести: метаязык PL/I, метаязык Вирта, используемый при описании Модуль-2, метаязык Кернигана-Ритчи, описывающий Си. Зачастую такие языки называются расширенными формами Бэкуса-Наура (РБНФ).

В частности, РБНФ, используемые Виртом, имеют следующие особенности:

- Квадратные скобки "[" и "]" означают, что заключенная в них синтаксическая конструкция может отсутствовать;
- фигурные скобки "{" и "}" означают ее повторение (возможно, 0 раз);
- круглые скобки "(" и ")" используются для ограничения альтернативных конструкций;
- сочетание фигурных скобок и косой черты "{" и "}" используется для обозначения повторения один и более раз. Нетерминальные символы изображаются словами, выражающими их интуитивный смысл и написанными на русском языке.

Если нетерминал состоит из нескольких смысловых слов, то они должны быть написаны слитно. В этом случае для повышения удобства в восприятии фразы целесообразно каждое ее слово начинать с заглавной буквы или разделять слова во фразах символом подчеркивания. Терминальные символы изображаются словами, написанными буквами латинского алфавита (зарезервированные слова) или цепочками знаков, заключенными в кавычки. Синтаксическим правилам предшествует знак "\$" в начале строки. Каждое правило оканчивается знаком "." (точка). Левая часть правила отделяется от правой знаком "=" (равно), а альтернативы - вертикальной чертой "|". Этот вариант РБНФ и будет использоваться для описания синтаксиса языков в лабораторной работе. В соответствии с данными правилами синтаксис идентификатора будет выглядеть следующим образом:

```
$ 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}.
```

## Диаграммы Вирта

Наряду с текстовыми способами описания синтаксиса языков широко используются и графические метаязыки, среди которых наиболее широкую известность получил язык диаграмм Вирта, впервые примененный для описания языка Паскаль. Метасимволы заменены следующими графическими обозначениями (рис. 2.1):



Рис. 2.1. Графические примитивы, используемые при построении диаграмм Вирта.

- терминальные символы и их постоянные группы располагаются в окружностях или прямоугольниках со скругленным вертикальными сторонами;
- нетерминальные символы заносятся внутрь прямоугольников;
- каждый графический элемент, соответствующий терминалу или нетерминалу, имеет по одному входу и выходу, которые обычно рисуются на противоположных сторонах;
- каждому правилу соответствует своя графическая диаграмма, на которой терминалы и нетерминалы соединяются посредством дуг;
- альтернативы в правилах задаются ветвлением дуг, а итерации - их слиянием;
- должна быть одна входная дуга (располагается обычно слева и сверху), задающая начало правила и помеченная именем определяемого нетерминала, и одна выходная, задающая его конец (обычно располагается справа и снизу).

Пример описания идентификатора с использованием диаграмм Вирта представлен на рис 2.2.



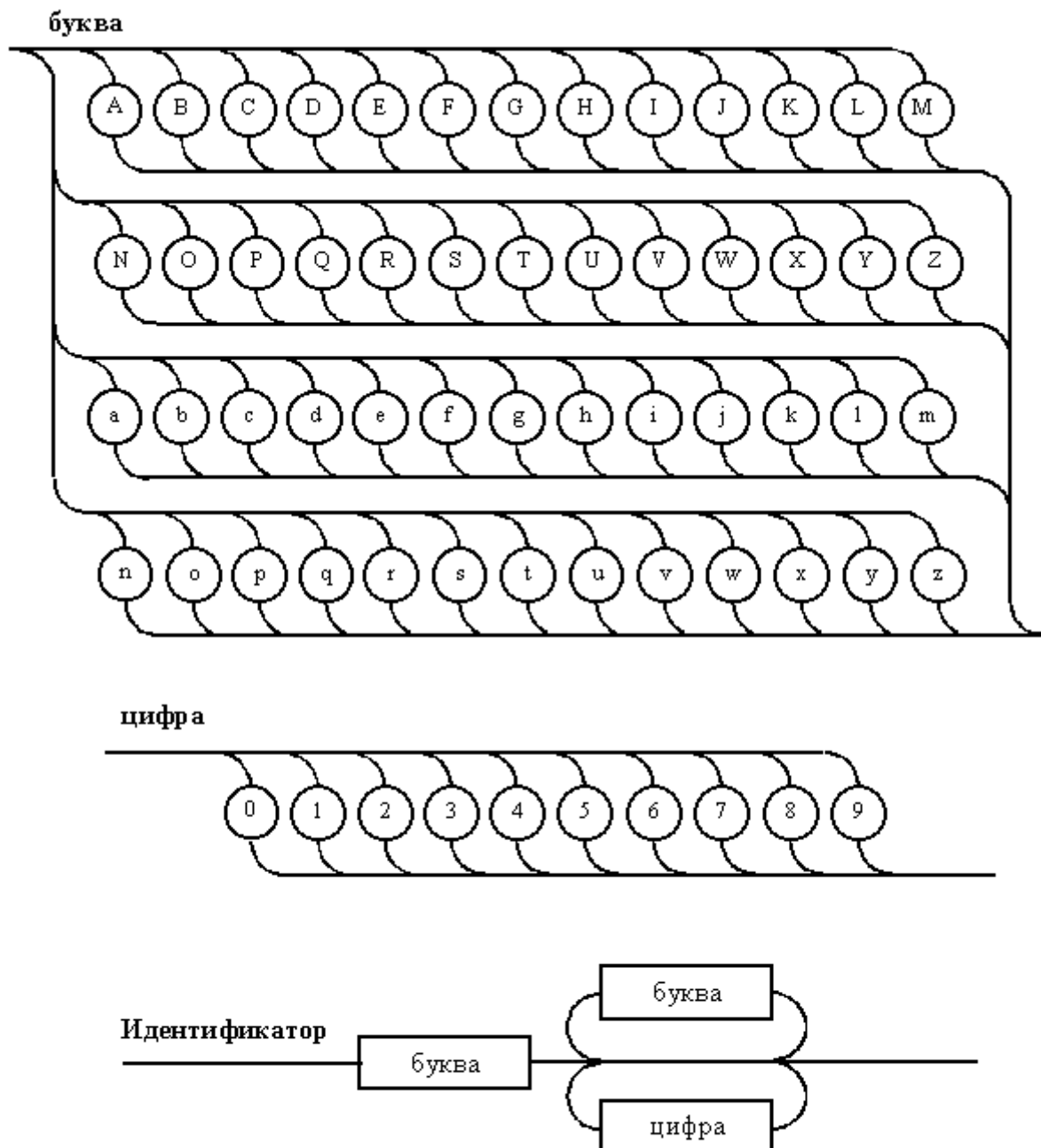


Рис. 2.2. Описание идентификатора с использованием диаграмм Вирта.

Обычно стрелки на дугах диаграмм не ставятся, а направления связей отслеживаются движением от начальной дуги в соответствии с плавными изгибами промежуточных дуг и ветвлений. Таким же образом определяются входы и выходы терминалов и нетерминалов. Специальных стандартов на диаграммы Вирта нет, поэтому графические обозначения могут меняться в зависимости от средств рисования. Можно, например, использовать псевдографику или просто текстовые символы, связи со стрелками. Однако такой вид правил менее удобен для восприятия и поэтому применяется крайне редко.

Диаграммы Вирта позволяют задавать альтернативы, рекурсии, итерации и по изобразительной мощности эквивалентны РБНФ. Но графическое отображение правил более наглядно. Кроме этого допускается произвольное проведение дуг, что уменьшает количество элементов в правиле за счет его неструктурированности. Диаграммы Вирта являются удобным исходным документом для построения лексического и синтаксического анализаторов.

# Распознаватели

**Распознаватель** – это очень схематизированный алгоритм, определяющий некоторое множество. Его можно представить в виде устройства (автомата). Этот автомат состоит из трех частей: входной ленты, устройства управления с конечной памятью и вспомогательной (рабочей) памяти (рис 2.3).

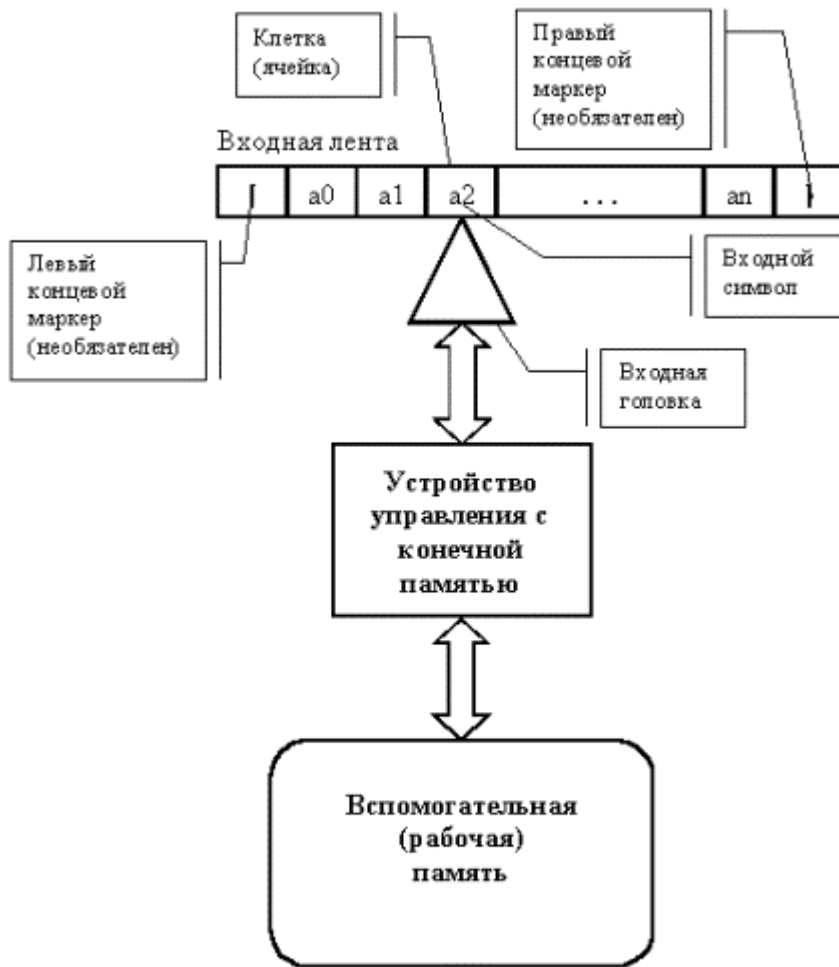


Рис. 2.3. Обобщенная структура распознавателя

**Входная лента** – линейная последовательность клеток (ячеек), каждая из которых содержит один входной символ из конечного входного алфавита. Могут присутствовать левый и правый концевые маркеры, может присутствовать только один концевой маркер (левый или правый), могут отсутствовать оба маркера.

**Входная головка** – в каждый момент читает одну входную ячейку. За один шаг входная головка может сдвинуться на одну ячейку влево, вправо и остаться неподвижной.

Распознаватель, никогда не передвигающий входную головку влево, называется **односторонним**. Обычно предполагается, что входная головка только читает. Но могут быть такие распознаватели, у которых входная головка и читает, и пишет.

**Память** – хранит информацию, построенную только из символов конечного алфавита памяти. Может иметь различную структуру: очередь, стек (магазин) и т. д. Можно читать из вспомогательной памяти и писать в нее. Для стека и очереди используются специфические операции (вталкивание, выталкивание).

**Устройство управления с конечной памятью** – программа, управляющая поведением распознавателя. Может являться аналогом конечного автомата. Определяет перемещение входной головки и работу с памятью на каждом шаге (такте). Переходит за шаг из одного состояния в другое.

**Конфигурация распознавателя** – мгновенный снимок, на котором изображены:

1. состояние устройства управления;
2. содержимое входной ленты;
3. содержимое памяти.

**Начальная конфигурация** – устройство управления находится в заданном начальном состоянии, входная головка читает самый левый символ на входной ленте, память имеет заранее установленное начальное содержимое.

**Заключительная конфигурация** – устройство управления находится в одном из состояний, принадлежащем заранее выделенному множеству заключительных состояний, входная головка обзореваает правый концевой маркер или, если маркер отсутствует, сошла с конца входной ленты. Иногда требуется, чтобы заключительная конфигурация памяти удовлетворяла некоторым условиям.

Распознаватель **допускает** входную цепочку  $w$ , если, начиная с начальной конфигурации, в которой цепочка  $w$  записана на входной ленте, распознаватель может проделать последовательность шагов, заканчивающуюся заключительной конфигурацией.

**Язык, определяемый распознавателем** – это множество цепочек, которые он допускает.

Для каждой из грамматик, приведенных выше в соответствии с иерархией Хомского, существуют распознаватели определяющие один и тот же класс языков.

1. Язык  $L$  **праволинейный** тогда и только тогда, когда он определяется конечным, односторонним детерминированным автоматом.
2. Язык  $L$  **контекстно-свободный** тогда и только тогда, когда он определяется односторонним недетерминированным автоматом с магазинной памятью.
3. Язык  $L$  **контекстно-зависимый** тогда и только тогда, когда он определяется двухсторонним недетерминированным линейно ограниченным автоматом.
4. Язык  $L$  **рекурсивно перечислимый** тогда и только тогда, когда он определяется машиной Тьюринга.

Данные определения показывают, что теория языков и формальных грамматик продвинулась достаточно далеко, чтобы служить самостоятельным предметом изучения. Не будем вдаваться в детали и выяснять смысл представленных понятий. Зафиксируем только сам факт эквивалентности между механизмами порождения и распознавания.

## Контрольные вопросы и задания

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  $G_3$  приведите пример вывода терминальной цепочки, содержащей три знака умножения и два знака сложения.
7. Приведите пример цепочки для грамматики  $G_3$ , 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.

# Тема 3. Демонстрационный язык программирования

## Содержание темы

Источники вдохновения для создания демонстрационного языка. Синтаксис и семантика DPL. Примеры программ. Описание пользовательского синтаксиса с использованием диаграмм Вирта.

## Источники вдохновения для создания демонстрационного языка

Чтобы осуществить разработку транслятора, необходимо иметь язык программирования. Можно взять уже существующий язык программирования. Однако, создаваемый транслятор будет большим по объему, что затруднит изучение методов его разработки. Можно упростить любой из существующих языков программирования до уровня, удобного для изучения основных методов разработки трансляторов. Такой подход используется достаточно часто. Он удобен и позволяет легко разобрать различные методы построения трансляторов. Вместе с тем, организация таких языков программирования обладает определенными ограничениями, определяемыми "настоящим" языком, что не позволяет изучить ряд специфических приемов. Введение в упрощенный язык своих дополнительных конструкций нарушает общее гармоничное восприятие от урезанного первоисточника и заставляет относиться к нему как к помеси бульдога с носорогом. Поэтому, проще разработать собственный, достаточно простой язык, в котором можно определить все конструкции, необходимые для демонстрации различных аспектов разработки трансляторов.

История программирования полна идеями, которые могут служить источниками для создания своего языка. Мы же остановимся на тех из них, которые были предложены Дейкстрой в его книге "Дисциплина программирования" [[Дейкстра78](#)]. В ней излагаются концепции безошибочного программирования и предлагаются языковые конструкции, поддерживающие такой подход. Этот язык также описан в книге Гриса [[Грис84](#)], посвященной изучению разработке правильных программ и методов доказательства правильности программ. Назовем разрабатываемый язык DPL (Deijkstra Programming Language). Учитывая, что разработка языка носит учебный характер, внесем в него ряд изменений, противоречащих позициям автора, но позволяющих изучить необходимые методы разработки трансляторов.

## Синтаксис и семантика DPL

DPL содержит основные операторы обработки данных и управления, которые позволяют строить простые программы. Вместе с тем, в нем отсутствуют конструкции, широко применяемые в развитых языках. В частности, нет процедур и функций, блочной структуры, типов данных, классов. Это не мешает изучению основных принципов разработки трансляторов, которые можно с успехом использовать и при разработке более сложных языков программирования. Описание языка построим по традиционному принципу. В начале рассмотрим элементарные конструкции, а затем структуру программы. Для описания синтаксиса DPL будем использовать РБНФ.

## Элементарные конструкции

К элементарным конструкциям языка обычно относятся его понятия, состоящие из терминальных символов, принадлежащих алфавиту языка. Выделение элементарных конструкций обусловлено целым рядом причин, среди которых можно отметить:

- мы имеем в своем распоряжении набор базовых "кирпичиков", опираясь на которые, легче изучать более сложные понятия;
- в большинстве языков программирования смысл и представление элементарных конструкций совпадают, поэтому, поняв их в ходе изучения одного языка, легче перейти к следующему.

К элементарным относятся такие понятия, как идентификатор, числа (целые, действительные, двоичные, десятичные), комментарии, метки, знаки операций, разделители, строки символов. Список можно продолжить и дальше. Эти понятия уточняются и конкретизируются при описании семантики языка. Например, идентификатор может служить в качестве имени переменной, процедуры, функции или типа. Элементарные конструкции обычно описываются с позиции, удобной для их изучения пользователем. Они могут распознаваться как во время лексического, так и синтаксического анализа, хотя большая их часть обычно распознается сканером.

Ниже приводится синтаксис элементарных конструкций DPL. Их описанию предшествует определение групп основных символов в качестве отдельных понятий, разделяющих эти символы на отдельные категории (классы). Аналогичные элементарные конструкции присутствуют в любом языке, поэтому дополнительное пояснение отсутствует.

\$ 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 | восьмиричное |

десятичное | шестнадцатиричное.

\$ двоичное = "{2}" {/ "0" | "1" /}.

\$ восьмиричное = "{8}" { "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7" }.

\$ десятичное = ["{10}"] {/ цифра /}.

\$ шестнадцатиричное = "{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 | "\"" } "\"".

Понятие **пробел** обозначает пустое место, которое можно задать пропуском " ". Однако, обычно это прямо не делается из-за опасения неправильной интерпретации или слияния кавычек при наборе текста и форматировании. Понятия **перевод\_строки** и **табуляция** определяют невидимые символы, ASCII коды которых меньше кода пробела. Они могут присутствовать в тексте программы, обеспечивая его форматирование, но только в виде специальных знаков, и на экране или бумаге не отображаются. Понятие символа определяет все видимые символы кодовой таблицы, а также символы перевода строки и табуляции.

Следует отметить, что ряд понятий определен неформально. Эти определения ориентированы на то, чтобы раскрыть пользователю языка смысл написания и использования основных конструкций. Поэтому данное описание будем называть **пользовательским синтаксисом** языка. Для более полного понимания оно обычно снабжается дополнительным пояснительным текстом, определяющим и уточняющим семантику. Этим дополнительным описанием в данном случае является приведенный выше абзац, поясняющий понятия: **пробел**, **перевод\_строки**, **табуляция**. Обычно пользовательский синтаксис напрямую не может быть использован для построения сканера или распознавателя. Его необходимо преобразовать к виду, удобного для эффективной реализации этих блоков транслятора.

Дополнительным описанием следует также снабдить понятия **комментария** и **строки**. Комментарий может задаваться в любом месте программы, где можно поставить пробел или разделитель. Он начинается парой символов "/\*" и заканчивается другой парой "\*/". Между ними могут быть любые печатаемые символы, включая отдельные группы звездочек и наклонных линий, не образующих завершающую комбинацию, а также символы табуляции, перевода строки. Строка может содержать только видимые символы, заключенные между двумя кавычками. Если в строке необходимо поставить кавычку, то она дублируется при написании:

**"Строка, в которой следующее ""слово"" взято в кавычки".**

Ключевые слова и разделители используются для формирования выражений, описаний и операторов. В DPL они определяются следующим образом:

**\$ ключевое\_слово = abort | begin | case | end | float | goto | int | loop |  
or | read | skip | space | tab | var | write |.**

**\$ разделитель = "(" | ")" | "[" | "]" | "," | ";" | ":" | "=" | "\*" | "/" | "%" |  
"+" | "-" | "->" | "<" | "=" | ">" | "<=" | ">=" | "!=".**

## **Составные конструкции. Организация программы**

К составным конструкциям относятся понятия, определяющие структуру программы, ее операторов, описаний и выражений.

**\$ 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 ].**

Приведенные правила требуют некоторых дополнительных пояснений, раскрывающих особенности семантики языка и мотивирующих решения, принятые из чисто субъективных соображений.

## **Краткое описание семантики языка.**

Оператор присваивания имеет нетрадиционную форму и, в общем случае, обеспечивает одновременное присваивание нескольким переменным, расположенным слева от знака "!=" значений предварительно вычисленных выражений, расположенных в правой части. Каждой переменной соответствует свое выражение. Присваивания начинаются только после вычисления всех выражений, результаты которых временно сохраняются. Это позволяет произвести обмен значений переменных с использованием только одного оператора присваивания:

**x, y := y, x**

В обычных языках программирования необходимо написать три отдельных оператора:

**t := x; x := y; y := t**

Выражения задаются в традиционной инфиксной форме. Порядок выполнения операций определяется их приоритетом и скобками. В начале выполняются выражения в скобках. Наивысший

приоритет имеет унарный минус "-", далее следуют мультипликативные операции "\*", "/", "%" (вычисление остатка), затем аддитивные "+", "-" и, наконец операции отношения "<", "=", ">", "<=", ">=", "!=".

Для представления пустого оператора Дейкстры намеренно использовал специальное ключевое слово **skip** (что мотивировал соответствующим текстом), хотя в большинстве языков программирования пустой оператор - это пустое место:

**\$ пустой = .**

Для того, чтобы продемонстрировать решения проблемы неоднозначности синтаксиса языка, возникающие из-за наличия "традиционного" пустого оператора, я ввел оба варианта написания. Но это не самый большой мой грех по отношению к Дейкстре.

Для экстренного выхода из любой точки программы в языке используется оператор прерывания **abort**. Необязательная строка символов предназначена для пояснения причины выхода из программы.

В соответствии с концепциями безошибочного программирования, разработанными Дейкстрой, определены условный оператор и оператор цикла. Их тела содержат наборы операторов, выполнение которых возможно только при истинности условий, задаваемых предваряющими их охраняющими выражениями. Выражения отделяются от охраняемых ими операторов стрелками "->" и, начиная с первого, последовательно анализируются до тех пор, пока не встретится "истинное". Истинным считается ненулевое значение выражения. Предполагается, что в рассматриваемой версии языка операции отношения возвращают в качестве результата целое число, равное 1, при выполнении условия и, равное 0, если условие не выполняется. Если в условном операторе все охраняющие выражения дают ложь, то он выполняется как оператор ошибки (**abort**). Оператор цикла в данной ситуации эквивалентен пустому оператору (**skip**). Возникновение такой ситуации обеспечивает выход из цикла. При наличии истинного охраняющего выражения происходит выполнения охраняемых операторов и повторное выполнение оператора цикла. Оператор **abort** также эквивалентен конструкции **case end** (пустое тело в условном операторе), а оператор **skip** - оператору **loop end**.

Спецификаторы **space**, **tab** и **skip** используются в операторе вывода для форматирования выходного потока данных и означают пробел, табуляцию и перевод строки. Выражение, следующее за спецификатором, определяет количество его повторений. Строка символов используется для вывода пояснительного текста.

И, наконец, прокомментирую акт вандализма, осуществленный мною по отношению к Дейкстре: включение в язык оператора перехода **goto**. И это после того, как он воздвиг фундамент структурного программирования и написал целый ряд обличительных статей о вреде данного рудимента и атавизма одновременно! Однако использование данного оператора в учебном языке осуществляется только для того, чтобы *продемонстрировать ряд особенностей, связанных с построением транслятора*. Поэтому, после нескольких лет сомнений и метаний (когда **goto** отсутствовал), я решился на столь тяжкий грех (хорошо, что Дейкстра не видел моих исходных текстов синтаксического анализатора, а не то гореть мне в аду).

# Примеры программ на DPL

Ниже приведены примеры программ, раскрывающие особенности использования языка.

---

## Пример 1. Алгоритм Евклида (нахождение наибольшего общего делителя).

```
begin
  var x, y: int; /* описание переменных */
  read x, y; /* ввод операндов */
  /* выполнять до равенства аргументов */
  loop x != y ->
    case
      x > y -> x := x - y
    or
      y > x -> y := y - x
    end
  end;
  write x /* полученный НОД */
end
```

---

## Пример 2. Одновременное нахождение наибольшего общего делителя (НОД) и наименьшего общего кратного (НОК).

```
begin
  var x, y, u, v: int; /* описание переменных */
  read x, y; /* ввод операндов */
  u, v := y, x;
  /* выполнять до равенства аргументов */
  loop
    x > y -> x, v := x - y, v + u
  or
    y > x -> y, u := y - x, u + v
  end;
  write "НОД = ", x; /* НОД */
  write skip, "НОК = ", (u + v) / 2 /* НОК */
end
```

---

## Пример 3. Суммирование n элементов из входного потока.

```
begin
  var a, i, s, n: int;
  i, s := 0, 0;
  read n;
  loop
    i < n -> read a; s, i := s+a, i+1
```



```
end;  
write s  
end
```

---

#### Пример 4. Сортировка элементов вектора.

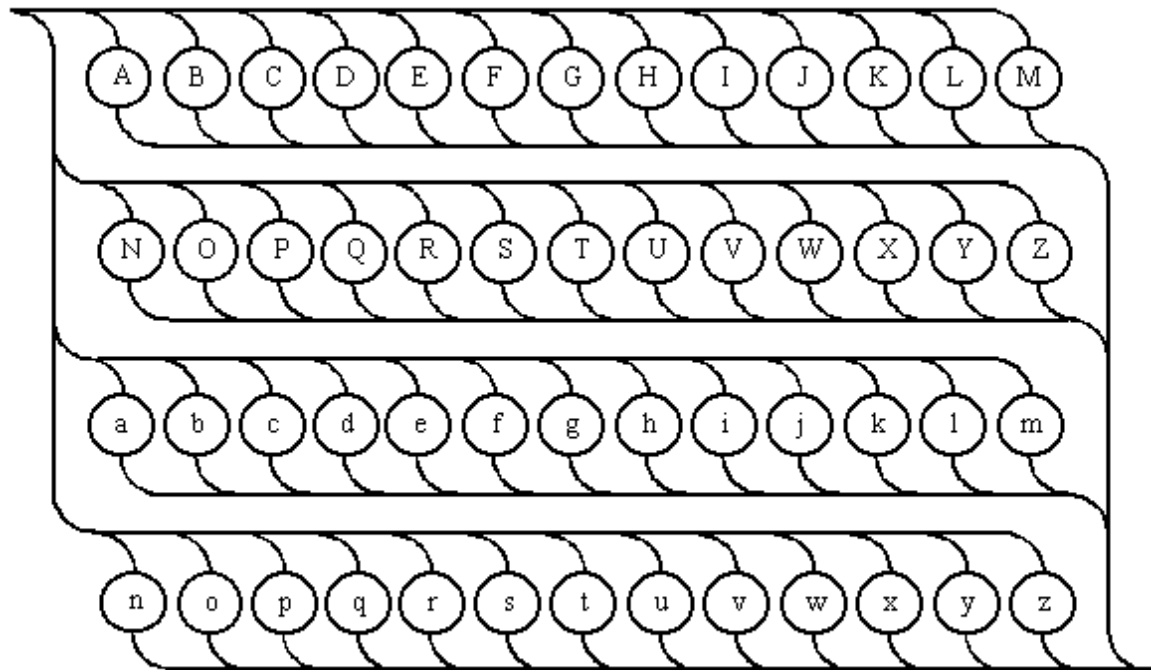
```
begin  
  var A[100], i, n: int;  
  i := 0;  
  read n; /* чтение числа элементов */  
  /* ввод вектора */  
  loop  
    i < n -> read A[ i ];  
    i := i + 1  
  end;  
  i := 0;  
  /* сортировка методом пузырька */  
  loop i < n-1 ->  
    case  
      A[ i ] > A[ i+1 ] ->  
        A[ i ], A[ i+1 ], i := A[ i + 1 ], A[ i ], 0  
    or  
      A[ i ] <= A[ i+1 ] ->  
        i := i + 1  
    end  
  end;  
  /* вывод вектора */  
  i := 0;  
  loop  
    i < n ->  
      write A[ i ], skip;  
      i := i + 1  
  end  
end
```

### Описание пользовательского синтаксиса с использованием диаграмм Вирта

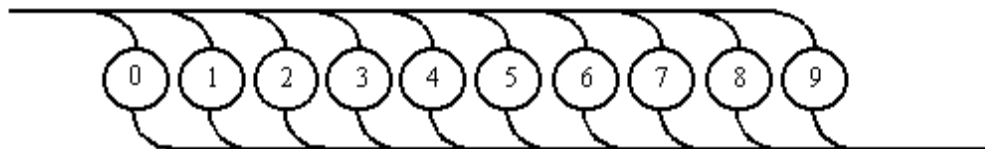
Разработка транслятора, рассматриваемая в последующих разделах, опирается на описание синтаксиса DPL с использованием диаграмм Вирта. Поэтому, необходимо осуществить перевод с одного метаязыка на другой. Сам синтаксис пока по-прежнему остается пользовательским, а его изменение будет осуществляться в дальнейшем по мере надобности. Окончательное представление синтаксиса DPL с применением диаграмм Вирта приведено на рис. 3.1 без каких либо дополнительных комментариев.

## Элементарные конструкции

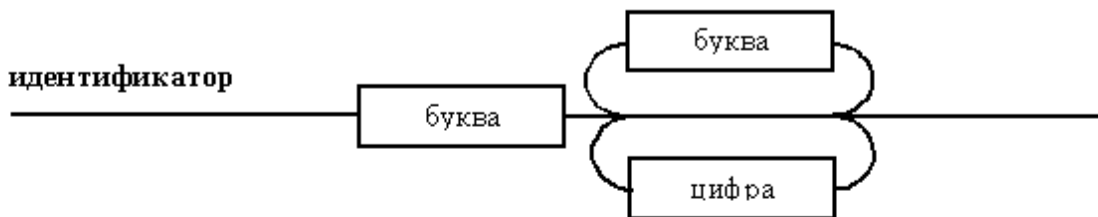
буква



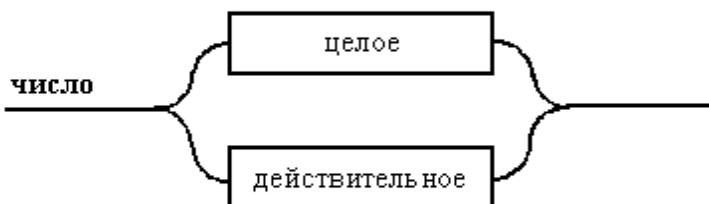
цифра



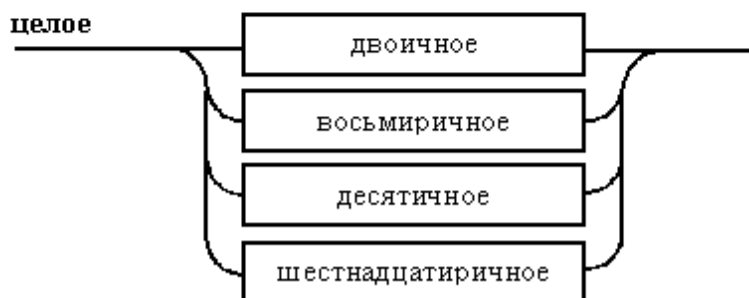
идентификатор



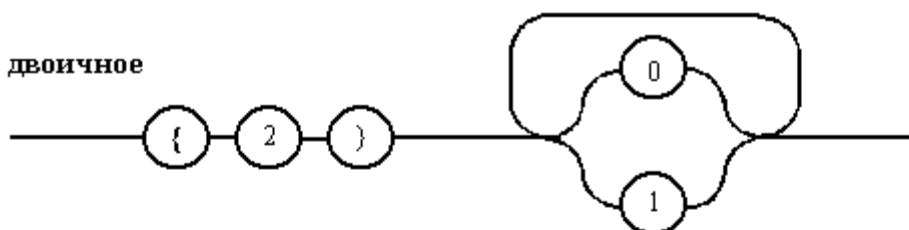
число



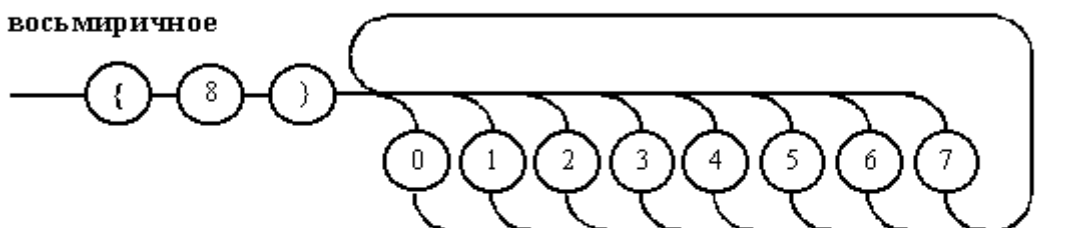
целое



двоичное



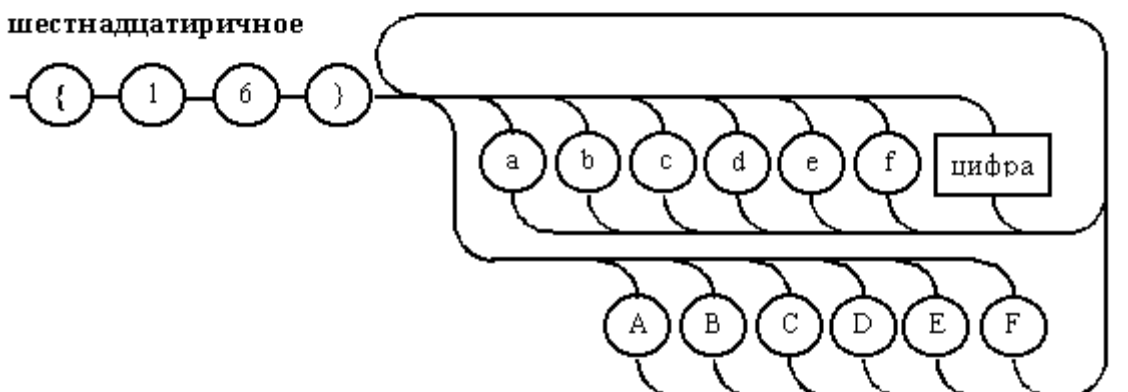
восьмиричное



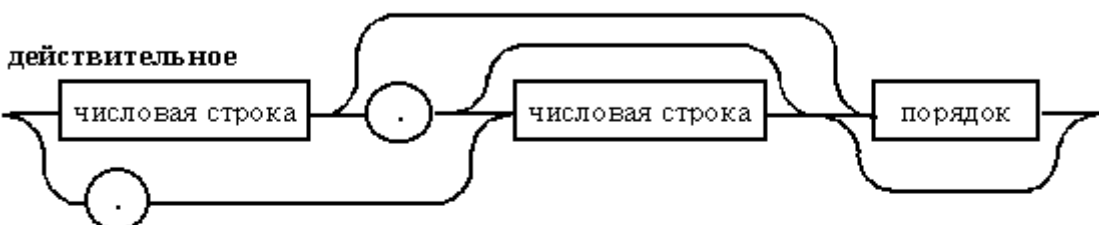
десятичное



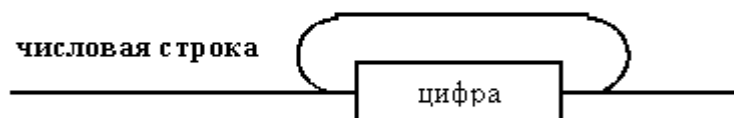
шестнадцатиричное



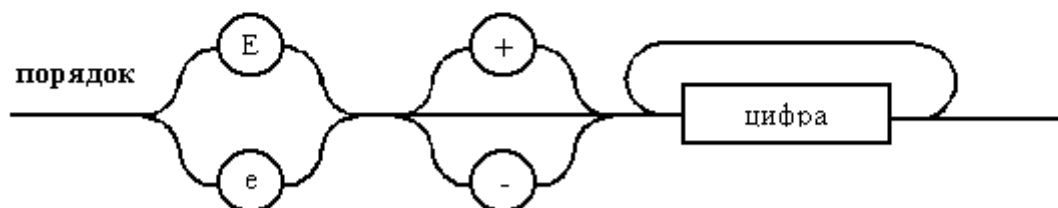
действительное



числовая строка



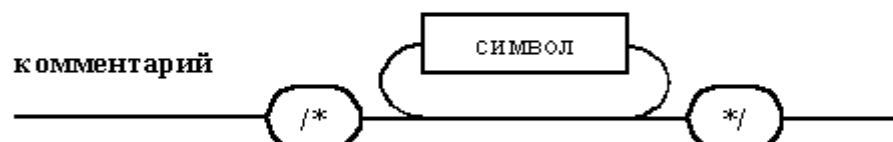
порядок



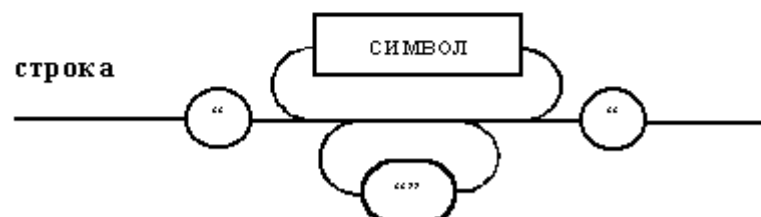
Пробельный символ



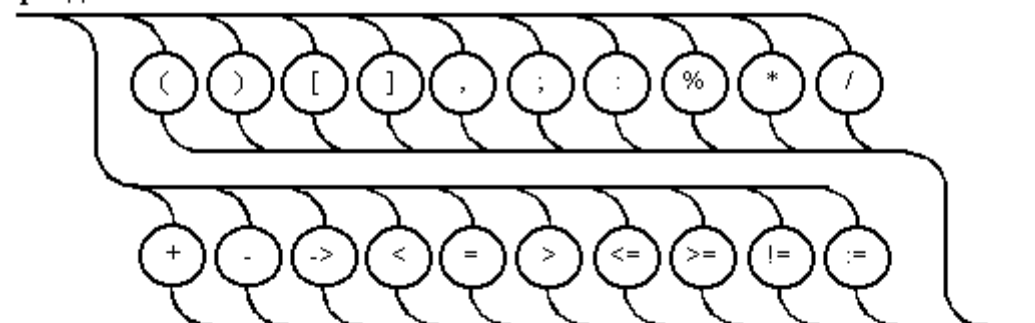
комментарий



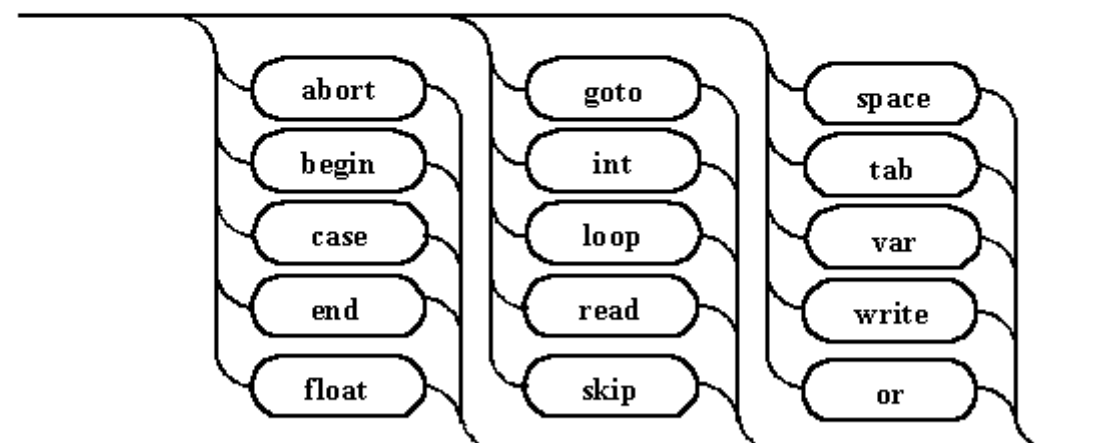
строка



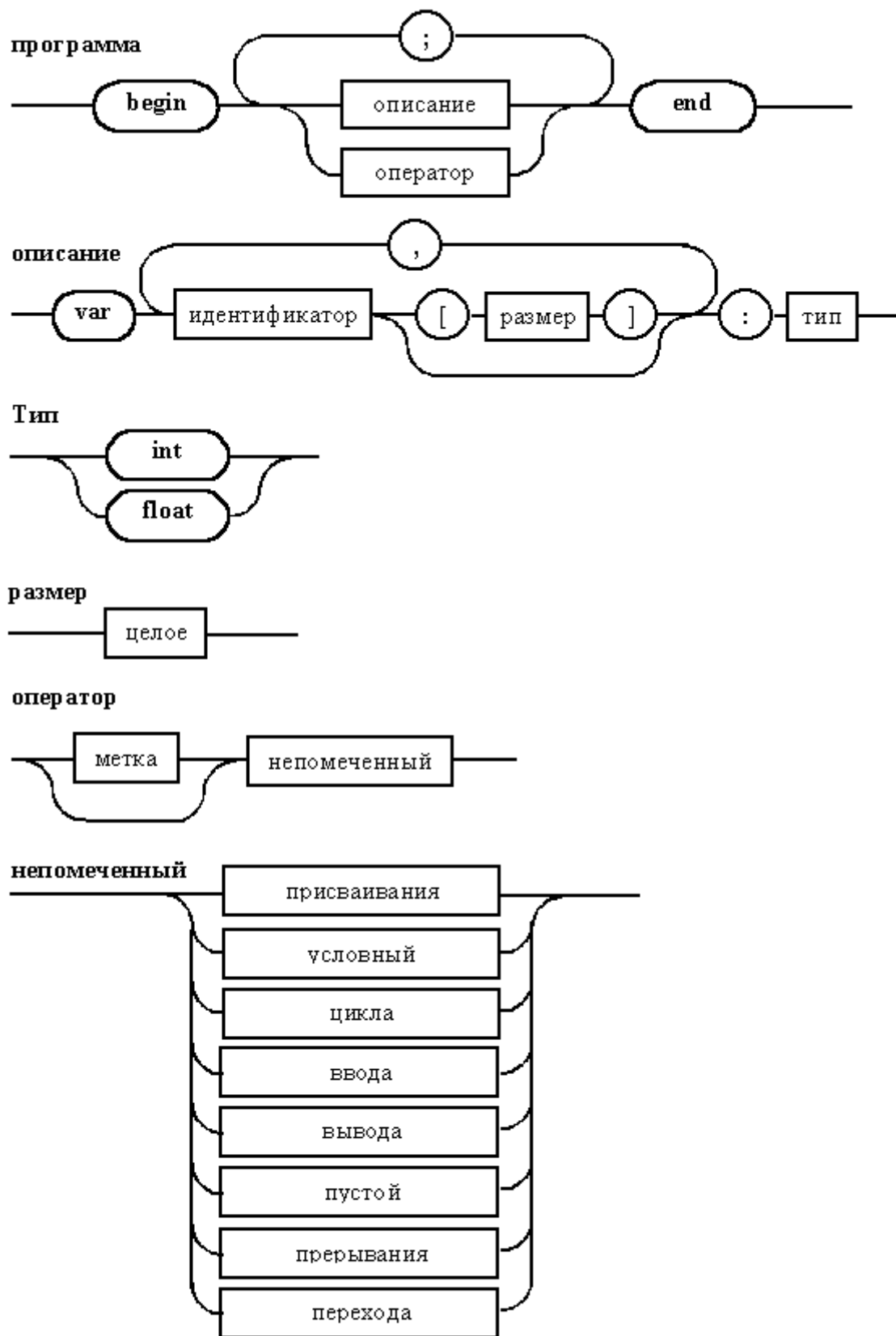
разделитель



ключевое слово



## Составные конструкции



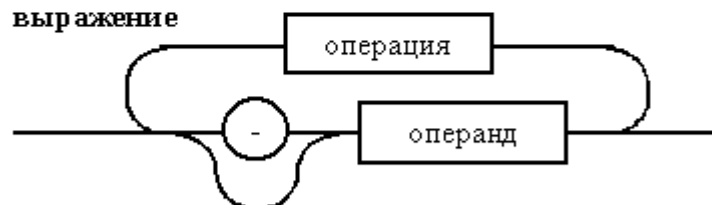
### присваивания



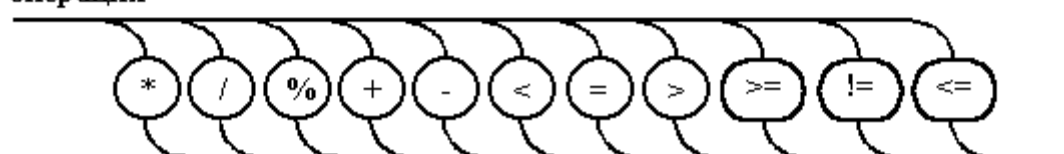
### переменная



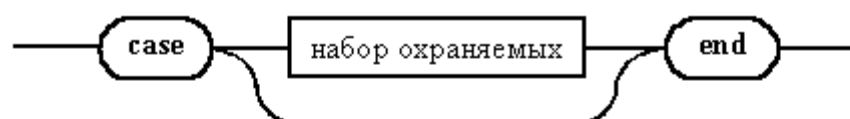
### выражение



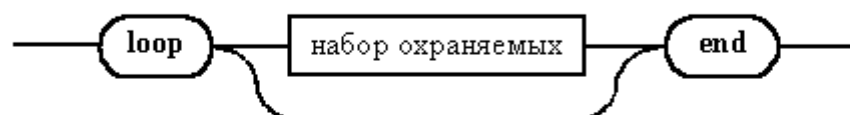
### операция



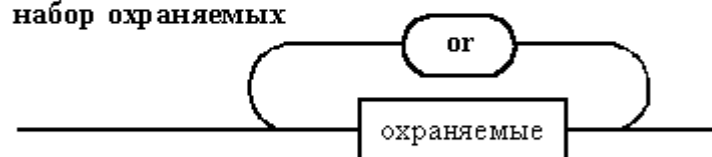
### условный



### цикла



### набор охраняемых



### охраняемые



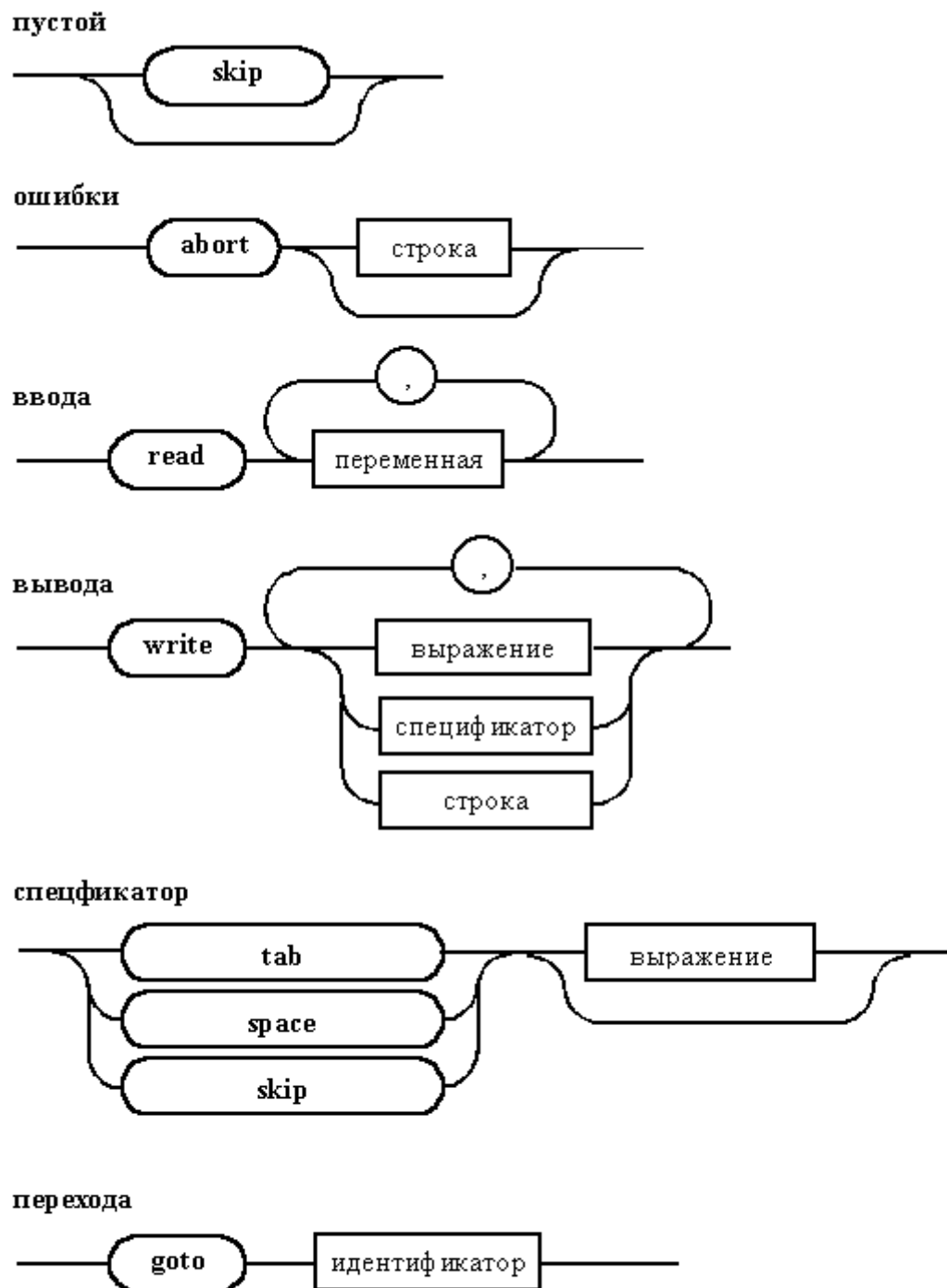


Рис. 3.1. Описание синтаксиса DPL с использованием диаграмм Вирта.

## Контрольные вопросы и задания

1. Завершить [лабораторную работу](#), написав аналогичные примеры программ для языка программирования, заданного вариантом.

# Тема 4. Организация лексического анализа

## Содержание темы

Назначение и необходимость фазы лексического анализа. Транслитератор. Грамматики и распознаватели для лексического анализа. Методы лексического анализа.

### Назначение и необходимость фазы лексического анализа

Лексический анализ - первая фаза процесса трансляции, предназначенная для группировки символов входной цепочки в более крупные конструкции, называемые лексемами. С каждой лексемой связано два понятия:

- **класс лексемы**, определяющий общее название для категории элементов, обладающих общими свойствами (например, идентификатор, целое число, строка символов и т. д.);
- **значение лексемы**, определяющее подстроку символов входной цепочки, соответствующих распознанному классу лексемы. В зависимости от класса, значение лексемы может быть преобразовано во внутреннее представление уже на этапе лексического анализа. Так, например, поступают с числами, преобразуя их в двоичное машинное представление, что обеспечивает более компактное хранение и проверку правильности диапазона на ранней стадии анализа.

В принципе, задачи, решаемые сканером, можно возложить на синтаксический анализатор. Но такой подход неудобен по следующим причинам:

1. Замена, цепочек символов, представляющих элементарные конструкции языка, делает внутреннее представление программы более удобным для дальнейшего анализа распознавателем. Последний манипулирует не отдельными символами, а законченными элементарными конструкциями, что облегчает их общее восприятие и дальнейший семантический анализ. Кроме этого, при построении лексем может осуществляться простейшая семантическая обработка. Например, преобразование и проверка числовых констант.
2. Уменьшается длина программы, поступающей в синтаксический анализатор, за счет устранения из нее несущественных для дальнейшего анализа пробелов, комментариев, игнорируемых символов. Уменьшение размера текста повышает производительность распознавателей, особенно для многопроходных компиляторов.
3. Один и тот же язык программирования может иметь различные внешние представления элементарных конструкций. Поэтому, наличие нескольких лексических анализаторов, порождающих на выходе одно и то же множество лексем, позволяет не переписывать синтаксический анализатор. Написать новый лексический анализатор намного проще, чем синтаксический. Примером языка, имеющего различные входные наборы символов, является PL/1. Для него определены 48-символьный и 60-символьный алфавиты.
4. Лексический анализатор использует более простые, по сравнению с синтаксическим анализатором, методы разбора. Следовательно, на одних и тех же цепочках и при выполнении разбора одних и тех же конструкций его производительность будет выше.
5. Блок лексического анализа естественным образом вписывается в иерархическую структуру компилятора, что тоже немаловажно при системном подходе к разработке трансляторов.



# Транслитератор

Несмотря на то, что лексический анализатор обрабатывает входную цепочку, удобнее на его вход подавать не просто отдельные символы, а символы, сгруппированные по категориям. Поэтому, перед лексическим анализатором осуществляется дополнительная обработка, сопоставляющая с каждым символом его класс, что позволяет сканеру манипулировать единым понятием для целой группы символов, иногда достаточно большой. Устройство, осуществляющее сопоставление класса с каждым отдельным символом, называется **транслитератором**. Наиболее типичными классами символов являются:

- **буква** - класс, с которым сопоставляется множество букв, причем необязательно только одного алфавита;
- **цифра** - множество символов, относящихся к цифрам, чаще всего от 0 до 9;
- **разделитель** - пробел, перевод строки, возврат каретки перевод формата;
- **игнорируемый** - может встречаться во входном потоке, но игнорируется и поэтому просто отфильтровывается из него (например, невидимый код звукового сигнала и другие аналогичные коды);
- **запрещенный** - символы, который не относятся к алфавиту языка, но встречается во входной цепочке;
- **прочие** - символы, не вошедшие ни в одну из определенных категорий.

Пара, класс символа и его значение, поступают на вход сканера, который выбирает для анализа тот ее элемент, который наиболее удобен в данный момент. Например, при анализе идентификатора удобнее манипулировать понятием "буква", тогда как при анализе действительного числа можно сразу смотреть значение буквы "Е" или "е", означающей начало порядка. Следует также отметить, что во всех дальнейших рассуждениях будем считать, что понятия "буква" и "цифра" являются терминалами, как полученные в транслитераторе до начала лексического анализа. В предыдущей трактовке эти понятия считались нетерминальными символами.

## Грамматики и распознаватели для лексического анализа

Лексические анализаторы обычно используются для распознавания элементарных конструкций, синтаксическое описание которых можно осуществить с применением праволинейных грамматик. Это наиболее простой класс грамматик, эквивалентных по мощности детерминированным конечным автоматам. Использование праволинейных грамматик для построения автоматов широко освещается в литературе [Ахо78, Льюис]. Простота лексического анализа заключается также в том, что лексемы - это единственные нетерминалы, используемые в ходе распознавания. Даже, если в грамматике, используемой для описания лексем, существуют промежуточные нетерминалы, они исчезают при построении конечного автомата, преобразуясь в его состояния. Входными символами конечного автомата являются терминальные символы и их классы символов, формируемые транслитератором.

### Связь между диаграммой Вирта и конечным автоматом

Построение конечного автомата можно осуществить также с использованием ряда грамматик с левой рекурсией и диаграмм Вирта. На практике, вместо праволинейной грамматики, удобнее использовать диаграммы Вирта. Они намного нагляднее при описании правил. Кроме этого, между диаграммами Вирта, не содержащими нетерминалов, и конечными автоматами существует однозначное соответствие. Практически это два эквивалентных способа представления одной модели, которая одновременно может служить как механизмом порождения, так и механизмом распознавания.

Конечный автомат, эквивалентный диаграмме Вирта, состоящей только из терминалов, строится следующим образом:

1. Начальная дуга диаграммы преобразуется в начальное состояние конечного автомата.
2. Конечная дуга диаграммы образует заключительное состояние конечного автомата.
3. Выходы отдельных дуг, соединяющих символы, и точки ветвления остальных дуг диаграммы образуют множество остальных состояний конечного автомата.
4. Конечные состояния диаграммы являются допускающими состояниями конечного автомата.
5. Терминальный символ диаграммы Вирта, расположенный между дугами, преобразуется в связь между соответствующими состояниями, помеченную входным символом, равным этому терминалу.
6. Связи, обеспечивающие переход в допускающие состояния, помечаются множеством оставшихся символов. Это множество не пересекается с множеством символов, обеспечивающих другие переходы из текущего в другие состояния (обозначены как "прочие").

Построение конечного автомата в соответствии с данными правилами можно рассмотреть на примере описания идентификатора. Диаграмма Вирта для идентификатора и полученный по ней конечный автомат приведены на рис. 4.1.

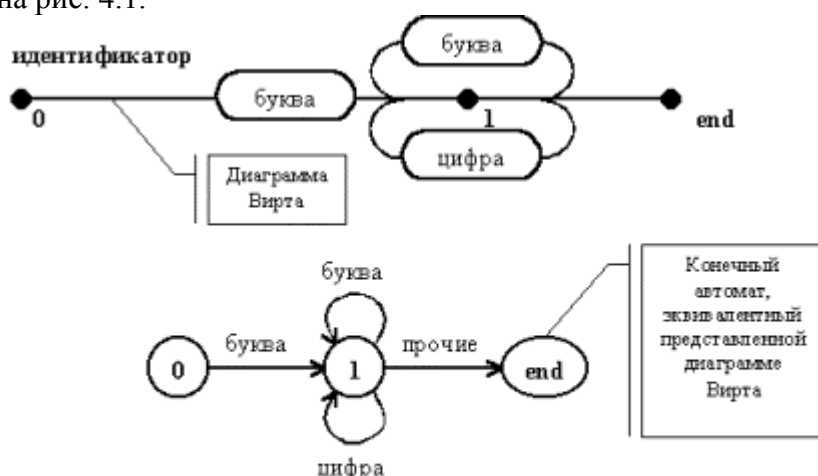


Рис. 4.1. Преобразование диаграммы Вирта в эквивалентный конечный автомат.

Для наглядности на дугах отмечены точки, соответствующие состояниям конечного автомата. Они обозначены теми же номерами, что и состояния. Заключительное состояние отмечено меткой "end". Следует еще раз отметить, что буква и цифра на диаграмме рассматриваются как терминалы, так как эти понятия формируются транслитератором.

Диаграммы Вирта могут использоваться и для того, чтобы минимизировать общее представление правил, что соответствует методам минимизации, применяемым к конечным автоматам. Эта минимизация осуществляется за счет объединения одинаковых подграфов диаграмм. Пример возможной минимизации понятия "идентификатор" представлен на рис. 4.2.

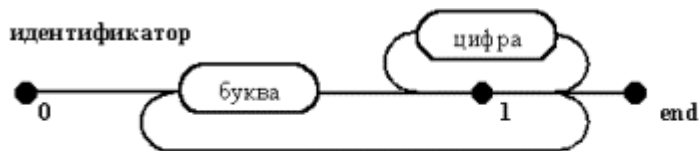


Рис. 4.2. Минимизированная диаграмма Вирта, задающая идентификатор.

Следует отметить, что не всегда минимизация диаграммы Вирта ведет к минимизации соответствующего ей конечного автомата. Приведенный рисунок показывает, что, несмотря на сокращение общего числа символов, разметка, соответствующая состояниям конечного автомата, осталась неизменной.

Наличие однозначного и легко понятного соответствия между диаграммами Вирта и конечными автоматами позволяет не строить последние, а переходить к написанию программы лексического анализатора непосредственно от диаграмм Вирта. Это сокращает технологическую цепочку, связывающую формальное описание элементов языка программирования и их программную реализацию.

## Связь между диаграммами Вирта и праволинейными грамматиками. Преобразование правой рекурсии в итерацию

Использование диаграмм Вирта для непосредственного написания программы позволяет говорить о том, что, при наличии праволинейной грамматики, можно не строить конечный автомат, а преобразовать исходную грамматику в диаграммы Вирта. Непосредственное представление праволинейной грамматики диаграммами Вирта не вызывает каких-либо проблем (рис. 4.3), но полученный результат не позволяет переходить к написанию программы из-за наличия в правых частях правил нетерминалов.

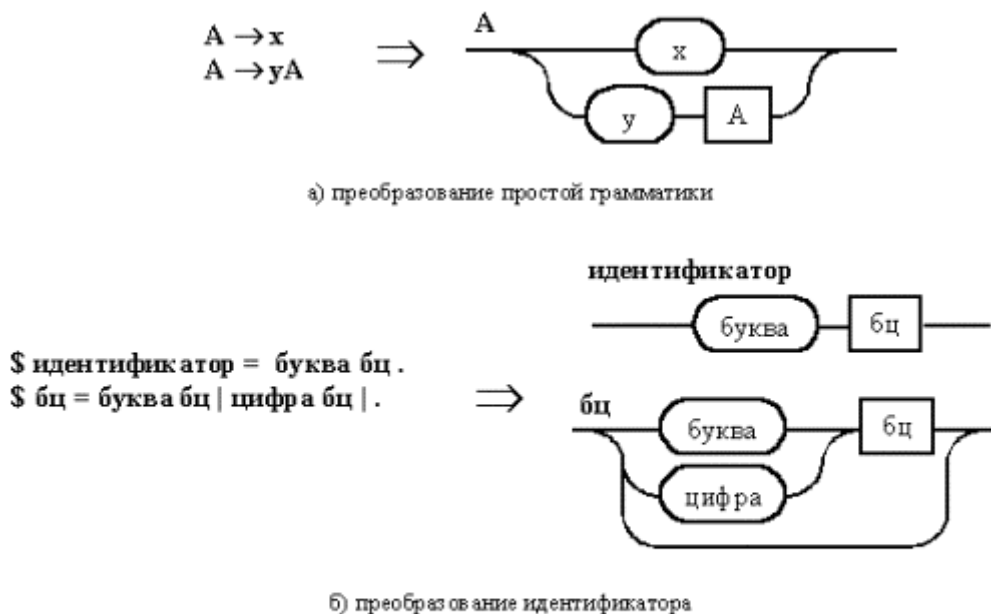


Рис. 4.3. Непосредственное преобразование праволинейных грамматик в диаграммы Вирта.

В ряде случаев от нетерминалов можно избавиться простой подстановкой. Но, если в правиле имеется рекурсия, обычную подстановку осуществить не удастся. Однако можно воспользоваться тем, что в праволинейной грамматике допускается только правая рекурсия, которая легко заменяется итерацией.

Замена правой рекурсии на итерацию в диаграммах Вирта производится следующим простым способом:

- дуга, **входящая** в рекурсивно определяемый нетерминал, замыкается своим **выходом** на самое начало правила, образуя, таким образом, цикл;
- сам нетерминал, оказавшийся в подвешенном состоянии вычеркивается, а выходящая из него дуга убирается.

Пример, иллюстрирующий, общие принципы преобразования правой рекурсии, представлен на рис. 4.4.

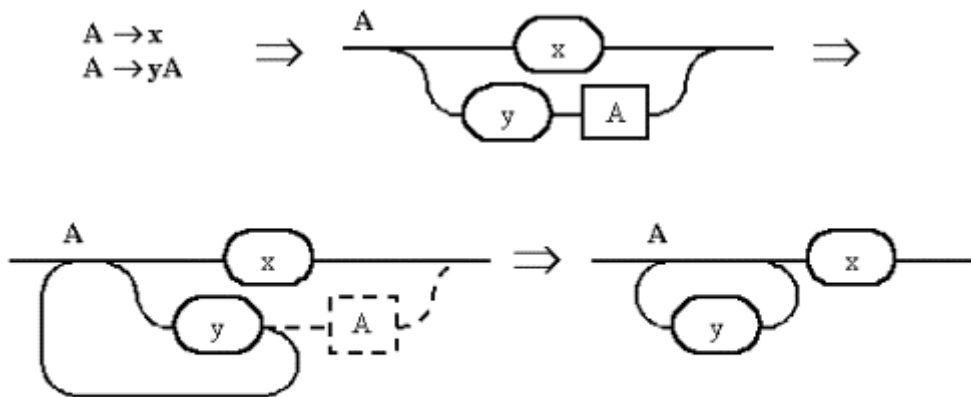


Рис. 4.4. Преобразование праволинейной грамматики в итеративную диаграмму Вирта.

Он показывает возможность такого преобразования, исходя из следующих индуктивных рассуждений. Нетерминал **A** на первом шаге порождает терминальную цепочку **x** или терминальную цепочку **y**, за которой следует нетерминал **A**, из которого на следующем шаге можно породить терминальную цепочку **x** или терминальную цепочку **y**, за которой следует нетерминал **A**, из которого... Эти бесконечные рассуждения на тему "У попа была собака..." можно заменить одной структурированной фразой: Нетерминал **A** порождает терминальную цепочку **y**, повторяющуюся ноль или большее число раз, за которой следует терминальная цепочка **x**. На рис. 4.5 приведено преобразование в итеративную диаграмму Вирта правил праволинейной грамматики, описывающих идентификатор. После избавления от лишних нетерминалов мы получаем хорошо знакомый результат.

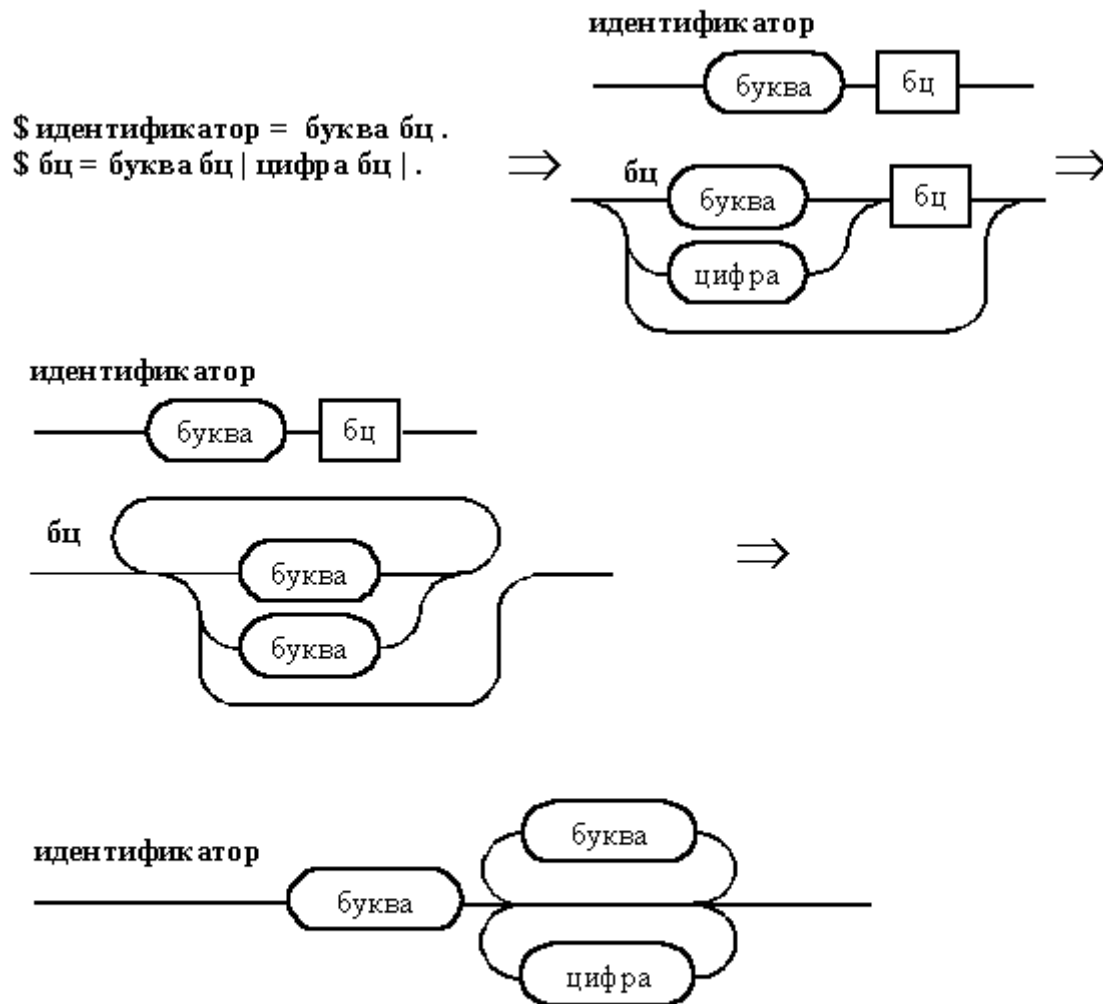


Рис. 4.5. Преобразование праволинейной грамматики идентификатора в итеративную диаграмму Вирта.

## Связь между диаграммами Вирта и грамматиками с левой рекурсией. Преобразование левой рекурсии в итерацию

Аналогичные преобразования в итерацию могут осуществляться и для правил, содержащих левую рекурсию:

- дуга, **выходящая** из рекурсивно определяемого нетерминала, замыкается своим **входом** на самый конец правила, образуя, таким образом, цикл;
- сам нетерминал, оказавшийся без адресата, вычеркивается, а входившая в него дуга убирается.

Таким образом, мы имеем зеркальную интерпретацию по отношению к правой рекурсии, что вполне очевидно. Пример, иллюстрирующий, общие принципы преобразования левой рекурсии, представлен на рис. 4.6.

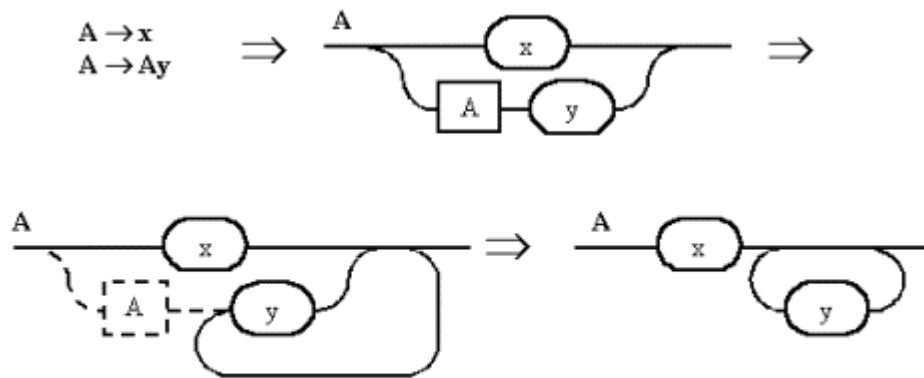


Рис. 4.6. Преобразование грамматики с левой рекурсией в итеративную диаграмму Вирта.

Он показывает возможность такого преобразования, исходя из следующих индуктивных рассуждений. Нетерминал  $A$  на первом шаге порождает терминальную цепочку  $x$  или нетерминал  $A$  (из которого на следующем шаге можно породить терминальную цепочку  $x$  или нетерминал  $A$  (из которого...), за которой следует терминальная цепочка  $y$ ), за которым следует терминальная цепочка  $y$ . А эти бесконечные рассуждения на тему "И надпись написал, что..." можно заменить еще одной структурированной фразой: Нетерминал  $A$  порождает терминальную цепочку  $x$ , за которой следует терминальная цепочка  $y$ , повторяющаяся ноль или большее число раз. На рис. 4.7 приведено преобразование в итеративную диаграмму Вирта правил грамматики с левой рекурсией, описывающих идентификатор. Налицо опять тот же результат.

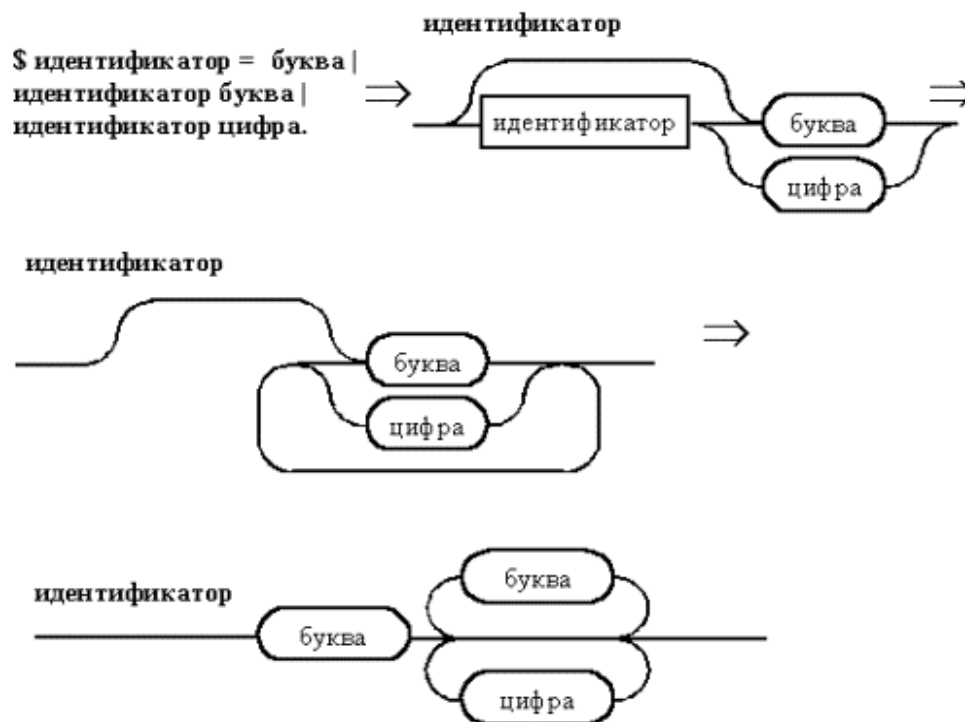


Рис. 4.7. Преобразование грамматики идентификатора с левой рекурсией в итеративную диаграмму Вирта.

Приведенные примеры показывают, что на практике, для описания лексем, удобнее пользоваться итеративными диаграммами Вирта, содержащими терминалы. Они могут быть легко получены из любого другого представления синтаксиса языка. Итеративные диаграммы Вирта, состоящие из терминальных символов практически эквивалентны конечным автоматам, что позволяет использовать их непосредственно для программной реализации. Для описания лексем можно использовать не только праволинейную грамматику (что рекомендуется теорией), но и грамматики с левой рекурсией (на практике дают более наглядное представление). И те, и другие легко сводятся к итеративным диаграммам Вирта, являющимся удобной основой для программной реализации лексического анализатора.

# Методы лексического анализа

Выделяются методы непрямого и прямого лексического анализа.

*Непрямой* лексический анализ, или лексический анализ с возвратами, заключается в последовательной проверке версий о классах лексем. Если проверка текущей версии не подтверждается, то происходит откат назад по цепочке символов и осуществляется проверка следующей версии.

*Прямой* лексический анализ позволяет определить значение лексемы без откатов назад по цепочке символов.

## Организация непрямого лексического анализатора

Непрямой лексический анализатор состоит из отдельных автоматов, каждый из которых распознает одну заданную лексему. Все автоматы имеют одинаковую структуру и отличаются только внутренними состояниями, что связано с различием распознаваемых лексем. Общая структура непрямого лексического анализатора приведена на рис. 4.8.

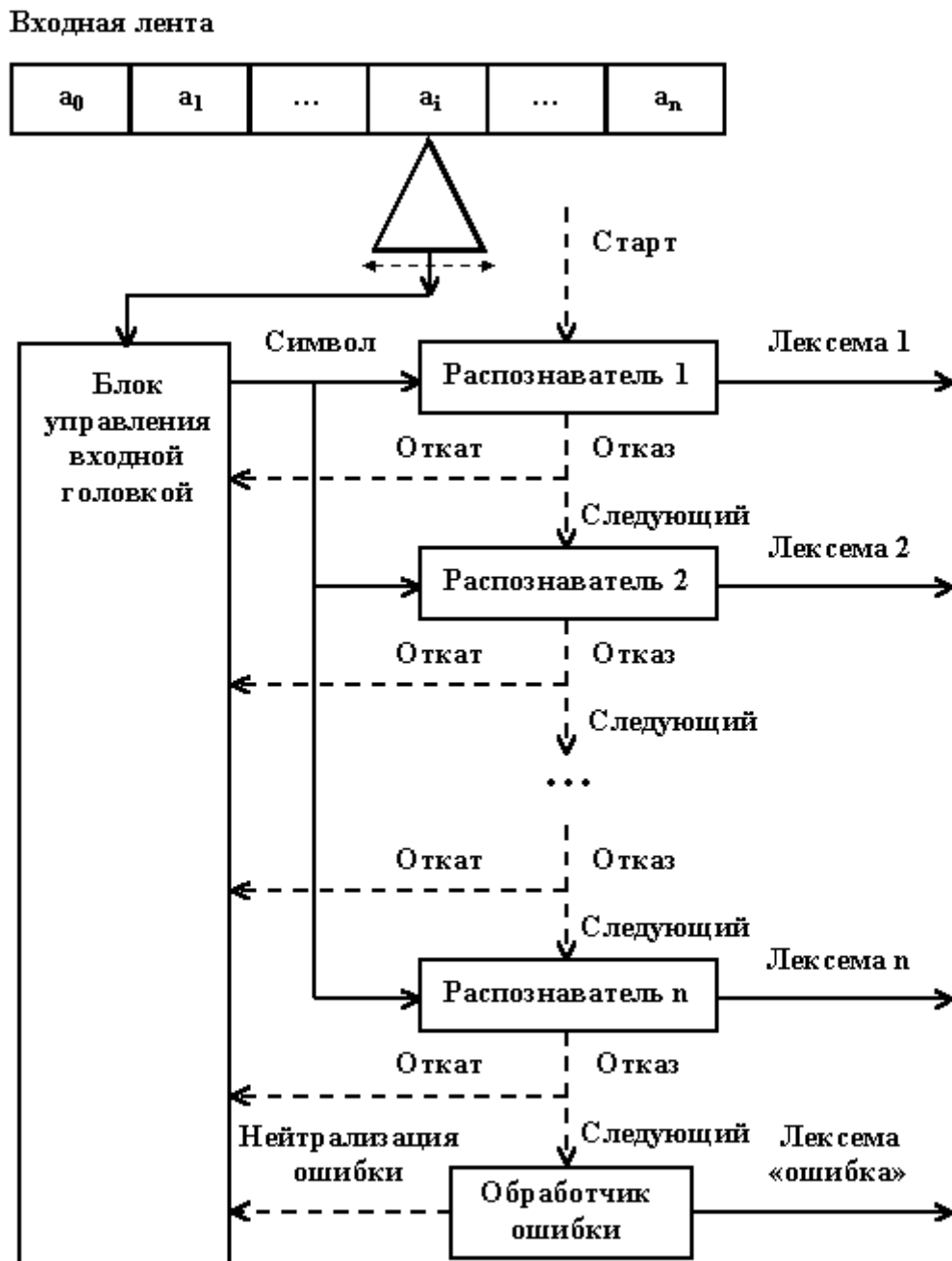


Рис. 4.8. Обобщенная структура непрямого лексического анализатора

Он имеет входную головку, читающую символы с входной ленты. Символы анализируются блоком управления входной головкой и передаются активному автомату. Затем входная головка смещается на один символ вправо.

Автоматы связаны между собой в последовательную цепь, что определяет порядок передачи управления между ними в случае, если активный автомат отвергает входную цепочку символов, то есть, не может распознать предписанную ему лексему. Последовательность автоматов в цепи определяется возможным предпочтительным анализом одних лексем перед другими. Наличие приоритета связано с тем, что некоторые лексемы имеют сходные начальные подцепочки и одна из лексем может включать другую. Например, действительное число может начинаться с цепочки цифр. С нее же может начинаться и целое число. Поэтому, в начале необходимо провести проверку на действительное число, а затем, если версия не подтвердится, попытаться распознать целое число. Существуют различные группы лексем, требующие приоритетной расстановки распознавателей. Они образуют группы автоматов, выстроенных в соответствии с приоритетом. Порядок размещения отдельных групп обычно не является существенным.

Анализ очередной лексемы начинается с запуска автомата, расположенного первым. Он читает первый символ  $a_i$  обрабатываемой подцепочки. Далее идет распознавание лексемы, в результате чего читаются  $n$  символов. Текущим символом становится  $a_{i+n}$ . Если версия о лексеме подтверждается, то сканер выдает ее и завершает работу. Его последующий запуск начинается с чтения символа  $a_{i+n}$ , который вновь передается первому автомату. Если же автомат не может распознать лексему, то он выдает сигнал отказа, в блок управления входной головкой и активизирует следующий по приоритету автомат. Блок управления осуществляет возврат входной головки к символу  $a_i$  и передает его новому активному автомату. Процесс возврата входной головки осуществляется до тех пор, пока лексема не будет распознана. Если не один из автоматов не распознает лексему, то управление передается обработчику ошибки. Тот выдает лексему "*ошибка*" и осуществляет ее нейтрализацию. В простейшем случае нейтрализация заключается в сдвиге входной головки на следующий символ  $a_{i+1}$  перед повторным запуском лексического анализатора.

К достоинствам непрямого лексического анализатора следует отнести:

- прозрачность общей регулярной структуры, которая легко может изменяться и наращиваться;
- простота каждого отдельно автомата, распознающего одну, достаточно элементарную, конструкцию;
- практическая применимость подхода в самых разнообразных языках, независимо от сложности выделения в нем тех или иных лексем.

Иллюстрацией последнего пункта может служить следующий фрагмент оператора цикла, являющийся правильной конструкцией в языке программирования FORTRAN-4:

***DO I = 3, 10, 3***

Так как в данном языке пробелы могут не использоваться, то рассмотренная запись будет корректной, если ее записать следующим образом:

***DOI=3,10,3***

Поэтому, при анализе сразу не ясно, что перед нами: рассмотренный оператор цикла или оператор присваивания, начинающийся с переменной ***DOI***:

***DOI=3***

Только возврат назад позволяет корректно просканировать эту и множество других изоциренных конструкций Фортрана.

К недостаткам непрямого лексического анализатора следует отнести низкую скорость распознавания. Это связано с постоянными возвратами входной головки к исходному состоянию, если версия о лексеме не подтверждается.

Теоретически данный недостаток можно исправить, если перестроить структуру непрямого лексического анализатора таким образом, чтобы все автоматы работали одновременно. Это вполне допустимо, так как отдельные автоматы являются вполне автономными устройствами. Общая структура такого лексического анализатора приведена на рис. 4.9.

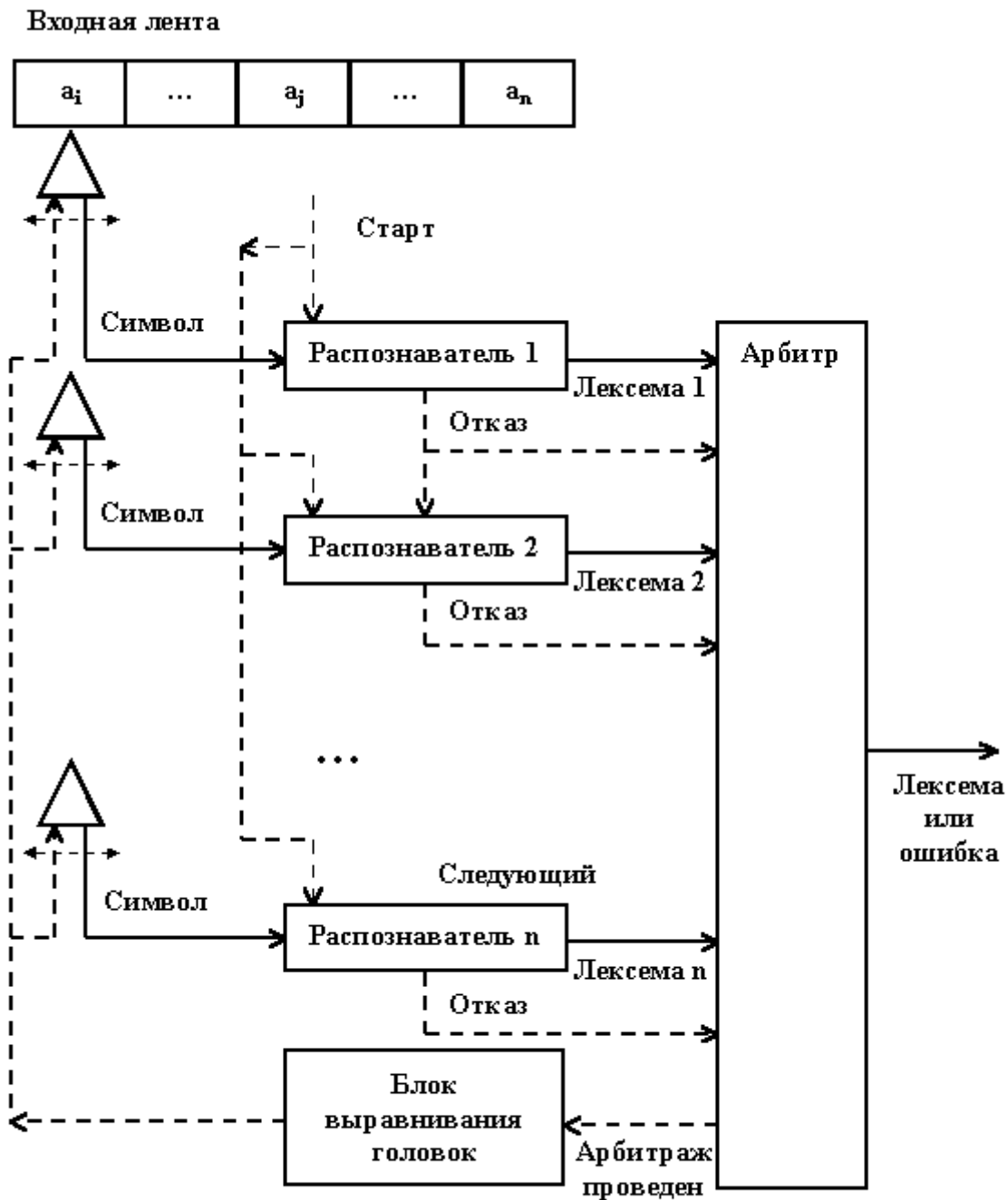


Рис. 4.9. Структура лексического анализатора с параллельной работой распознавателей

Каждый из распознавателей имеет свою входную головку. Перед запуском лексического анализатора все они читают один и тот же символ  $a_i$ . После запуска каждый из распознавателей анализирует входную цепочку одновременно с другими, допуская или отвергая ее к концу своей работы.

Распознанные лексемы и отказы передаются арбитру, который определяет нужную выходную лексему в соответствии с избранной схемой их приоритетов. Перед завершением цикла распознавания текущей лексемы входные головки всех автоматов выравниваются по положению входной головки автомата, породившего выходную лексему. В том случае, если все автоматы выдадут отказ, арбитр формирует на выходе ошибочную лексему и выравнивает все входные головки в соответствии с избранным способом нейтрализации ошибки (можно поступить так же, как и в предыдущем анализаторе: осуществить выравнивание на символе  $a_{i+1}$ ).

Не смотря на повышение скорости распознавания, такой метод не находит, в настоящее время, практического применения. Это связано с отсутствием вычислительных систем, обеспечивающих реальный параллелизм на уровне коротких последовательностей команд. Кроме того, низким является объем полезных вычислений, так как на каждую распознанную лексему приходится большой объем бесполезных вычислений в автоматах, порождающих другие лексемы и отказы



## Организация прямого лексического анализатора

Прямой лексический анализатор строится на основе одного детерминированного автомата, объединяющего множество автоматов, распознающих отдельные лексемы. Такой автомат на каждом шаге читает один входной символ и переходит в следующее состояние, приближающее его к распознаванию текущей лексемы или формированию ошибки. Для лексем, имеющих одинаковые подцепочки, автомат имеет общие фрагменты, реализующие единое множество состояний. Отличающиеся части реализуются своими фрагментами. Обобщенная структура прямого лексического анализатора приведена на рис. 4.10.

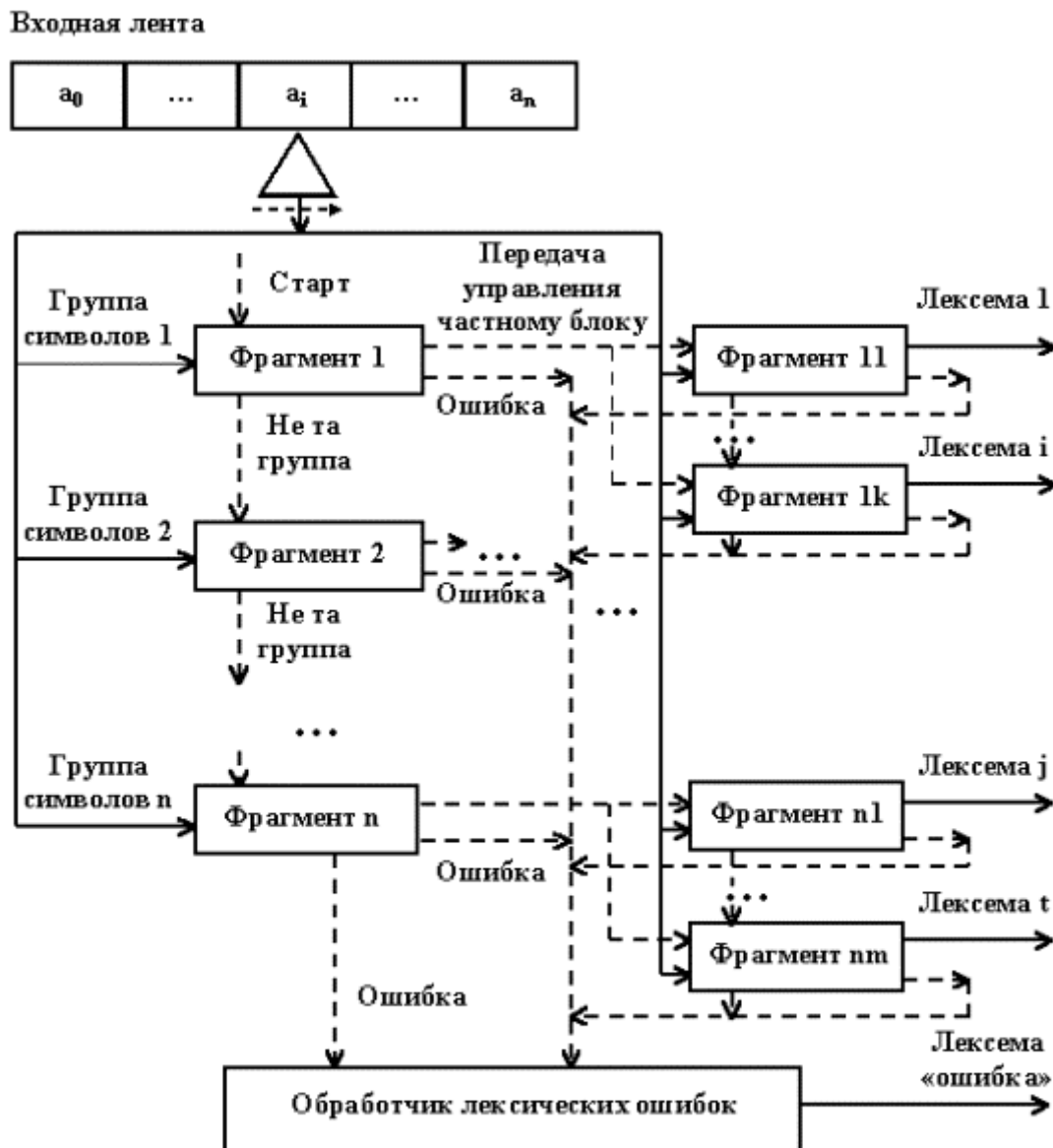


Рис. 4.10. Обобщенная структура прямого лексического анализатора

Он содержит входную головку, которая передает текущий входной символ в первое состояние одного из начальных блоков, реализующего фрагмент детерминированного автомата. Каждое из таких состояний распознает множество входных символов, не пересекающихся с множеством входных символов других состояний. Разбиение автомата на фрагменты проводится на основе выделения состояний, специализирующихся на распознавании общих или частных подцепочек отдельных лексем. В самом общем случае прямой лексический анализатор может рассматриваться как единый неструктурированный блок. Поэтому его дробление на отдельные блоки достаточно условно. Очередной символ читается обычно каждым состоянием автомата. В каждом фрагменте условно можно выделить начальное состояние, с которого начинается разбор подцепочки, закрепленной за фрагментом. Если символ не входит в группу, распознаваемую начальным состоянием фрагмента, то происходит отказ и передача управления начальному состоянию следующего альтернативного фрагмента, анализирующего

другую группу входных символов. Если входной символ не может быть распознан другими состояниями фрагмента, то обработка отказа передается обработчику ошибок, который формирует ошибочную лексему и нейтрализует ошибку принятым способом. Передача управления обработчику ошибок осуществляется и в том случае, если все начальные состояния связанных между собой фрагментов формируют отказ. Разбор заканчивается выдачей распознанной или ошибочной лексемы.

К достоинствам прямого лексического анализатора относятся:

- высокая производительность за счет отсутствия возвратов входной головки;
- меньшее общее число состояний, что получается за счет слияния одинаковых фрагментов отдельных автоматов и проведения дополнительной минимизации.

К недостаткам следует отнести:

- на разработку прямого лексического анализатора требуется больше времени;
- данный анализатор практически невозможно реализовать для некоторых языков программирования. Примером такого языка может служить FORTRAN-4.

## **Контрольные вопросы и задания**

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 повысить 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.

# Тема 5. Лексический анализатор демонстрационного языка программирования

## Содержание темы

Транслитератор DPL. Непрямой лексический анализатор DPL. Прямой лексический анализатор DPL.  
*Примечание. Если Вам интересно взглянуть на исходные тексты непрямого и прямого лексических анализаторов и результаты их работы, то [около 52 Кб только и ждут этого момента](#).*

## Транслитератор DPL

### Общая организация транслитератора

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

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

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

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

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

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

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

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

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

Следует отметить, что классы символов пересекаются. Однако, вопрос принадлежности нужному классу можно решать, основываясь на текущем контексте. Класс символов можно специально не хранить, а проверять тогда, когда потребуется.

### Программная реализация транслитератора

Реализация транслитератора, как и любого другого программного модуля, во многом зависит от стиля программирования. Несмотря на простоту, эта задача может быть рассмотрена и в более широком контексте. Транслитератор можно использовать как совокупность нескольких функций, проверяющих принадлежность символа к одному из классов. Такая реализация выглядит вполне логичной при процедурном программировании. Вместе с тем следует отметить, что транслитератор не используется сам по себе. Он является промежуточным звеном между функциями, обеспечивающими чтение

символов из входного потока и лексическим анализатором. Это промежуточное звено обычно скрывает от лексического анализатора механизм чтения и преобразования символов. Сканеру так же ничего не надо знать об открытии входного потока и манипуляции с ним. Поэтому, при объектно-ориентированном подходе, получение значение нового символа, его класса, а также манипуляция входным потоком реализуются как единый объект (на основе класса). Этот класс инкапсулирует внутренние операции, а также объединяет разбросанные по программе структуры данных, используемые для ввода символов и их преобразования.

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

```
// Функции транслитератора, используемые для определения класса лексем
```

```
//
```

```
// Определяет принадлежность символа к классу букв
```

```
bool inline isLetter(int ch) {  
    if((ch>='A' && ch<='Z') || (ch>='a' && ch<='z'))  
        return true;  
    else  
        return false;
```

```
}
```

```
//
```

```
// Определяет принадлежность символа к классу двоичных цифр
```

```
bool inline isBin(int ch) {  
    if((ch=='0' || ch=='1'))  
        return true;  
    else  
        return false;
```

```
}
```

```
//
```

```
// Определяет принадлежность символа к классу восьмеричных цифр
```

```
bool inline isOctal(int ch) {  
    if((ch >= '0' && ch <= '7'))  
        return true;  
    else  
        return false;
```

```
}
```

```
//
```

```
// Определяет принадлежность символа к классу десятичных цифр
```

```
bool inline isDigit(int ch) {  
    if((ch >= '0' && ch <= '9'))
```

```

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

```

При использовании объектно-ориентированного подхода все эти данные можно инкапсулировать в один класс, обеспечив доступ к ним через соответствующий интерфейс. Следует также отдельно отметить последнюю строку функции **nxl()**: **++poz; // Переход к следующей позиции в файле**

Она присутствует только в непрямом лексическом анализаторе и предназначена для фиксации позиции в файле, что позволяет осуществлять откаты назад в том случае, если проверяемая версия о значении лексемы не подтвердится. Взятие следующего символа в прямом сканере происходит без использования этого "довеска".

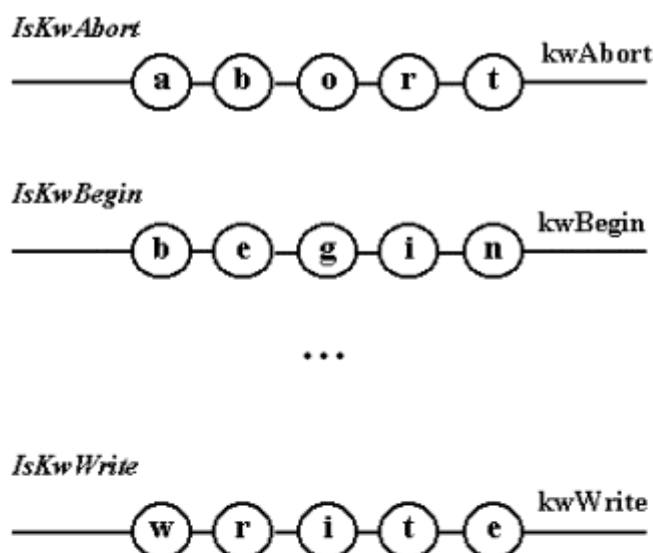
# Непрямой лексический анализатор DPL

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

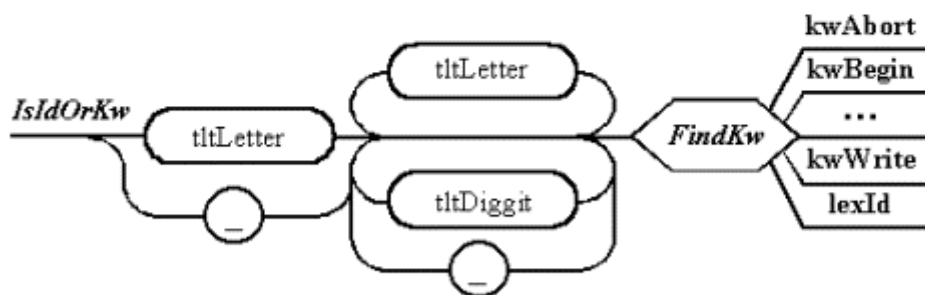
Конечный автомат, осуществляющий распознавание одной конкретной лексемы, может быть разбит на несколько, более мелких автоматов, каждый из которых распознает непересекающееся подмножество цепочек, принадлежащих искомому классу. Например, при распознавании целого числа можно построить отдельные автоматы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

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

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



а) Непосредственный анализ ключевых слов.



б) Семантическое выделение ключевых слов из анализа идентификатора

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

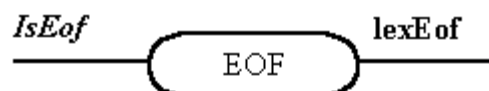
Для распознавания каждого ключевого слова можно построить свой автомат. В этом случае возникает несколько проблем:

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

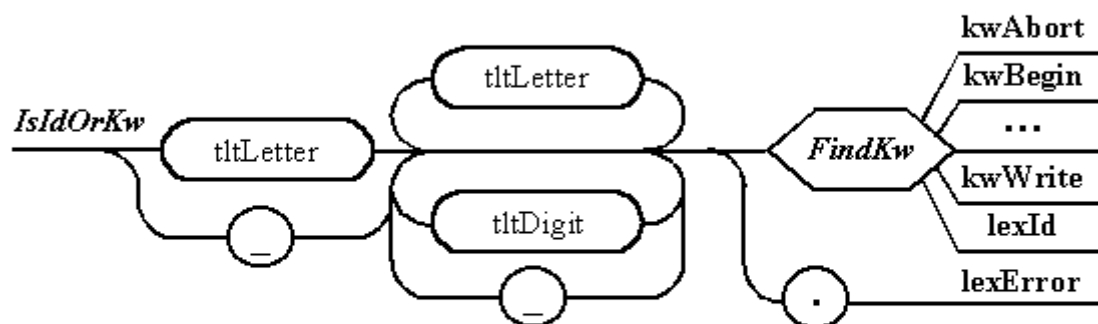
Гораздо проще и быстрее провести распознавание ключевых слов с использованием семантической обработки. Чаще всего (а в данном случае - это факт) ключевые слова являются подмножеством идентификаторов. Поэтому, можно в начале осуществить выявление идентификатора, а затем провести его анализ на принадлежность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

### **Диаграммы Вирта для отдельных автоматов непрямого лексического анализатора**

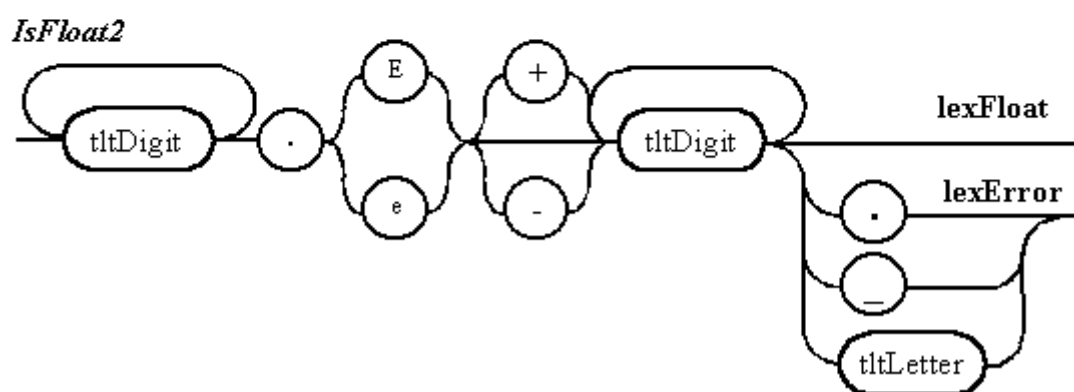
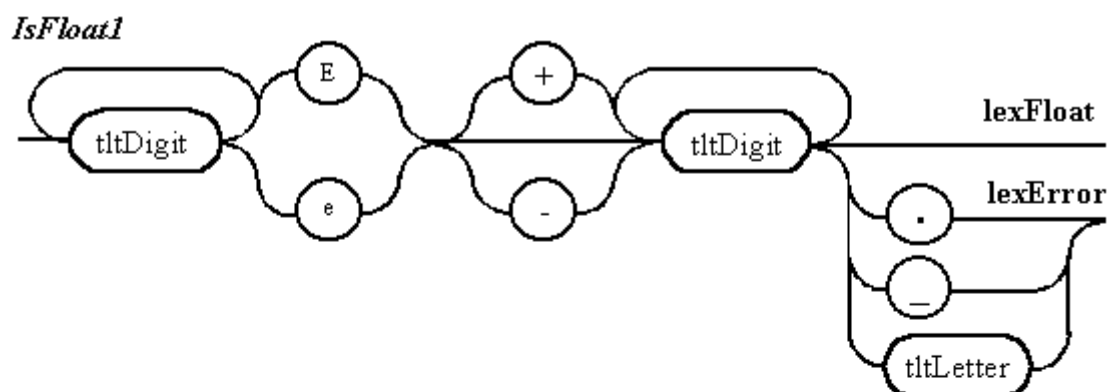
Диаграммы Вирта, описывающие отдельные независимые фрагменты непрямого лексического анализатора, представлены на рис. 5.2. В отличие от диаграмм, используемых для описания пользовательского синтаксиса, данные схемы помечены именами, которые предполагается использовать в программе. Выходы диаграмм идентифицируют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



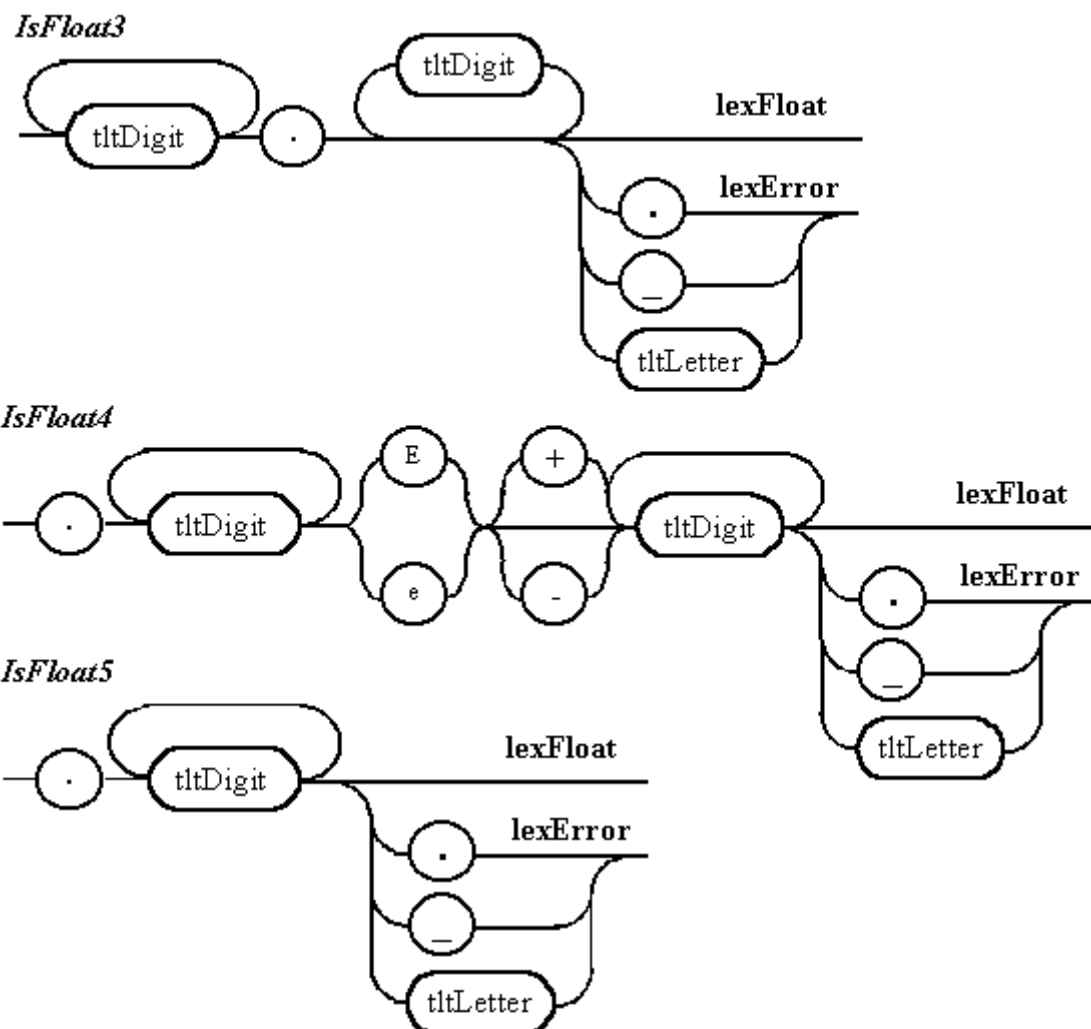
Примечание 1. Лексема, порождаемая при достижении конца обрабатываемого текста.



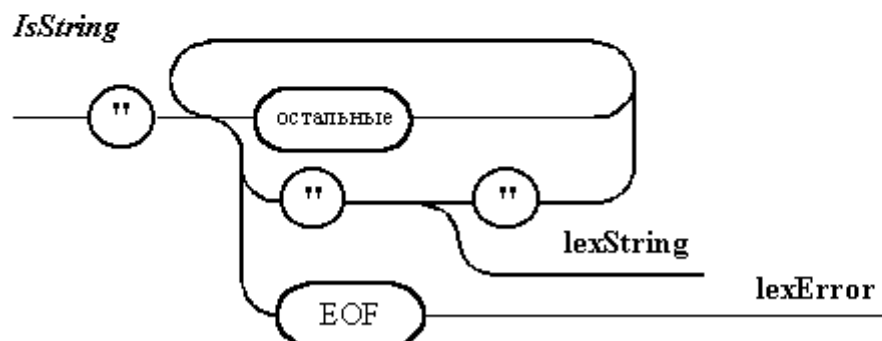
Примечание 2. Идентификатор и ключевые слова описываются правилом с семантической вставкой. Осуществляется также анализ на недопустимость возможного слияния идентификатора с действительным числом, начинающимся с точки.



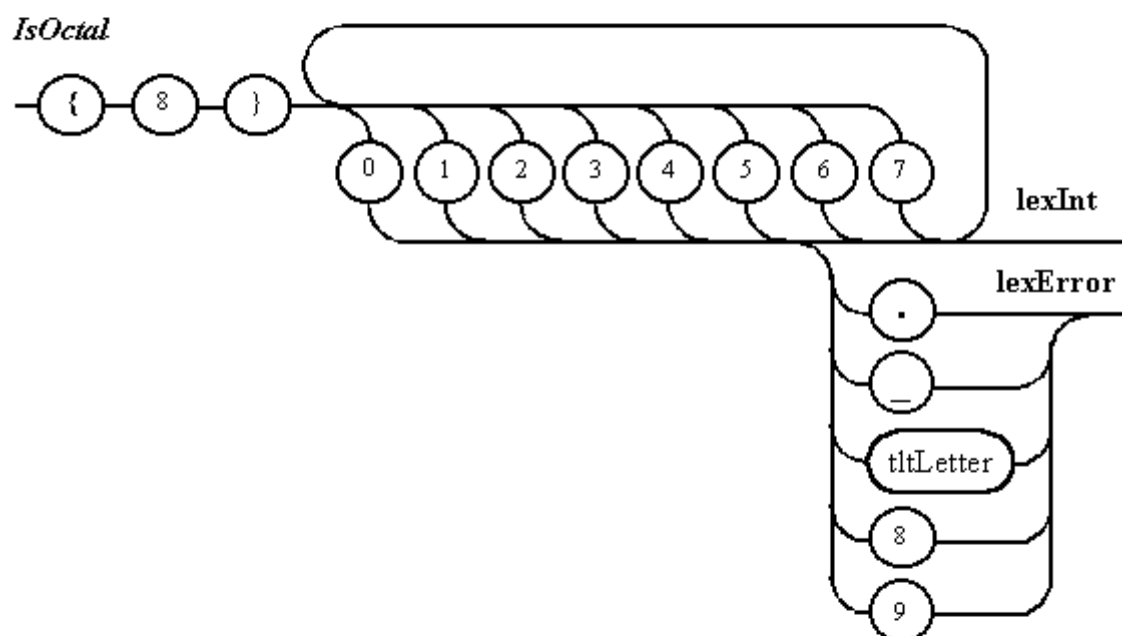
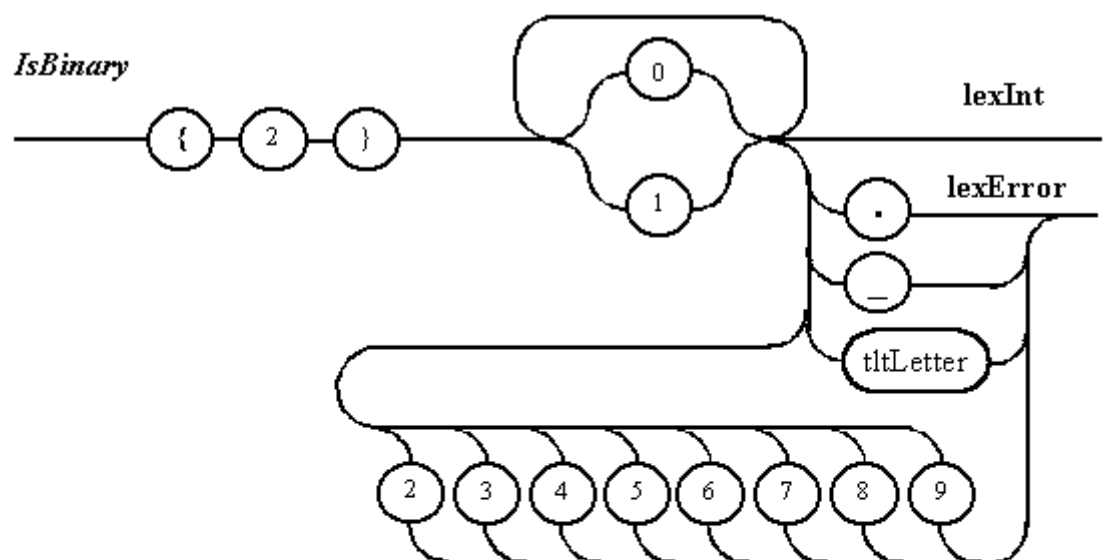




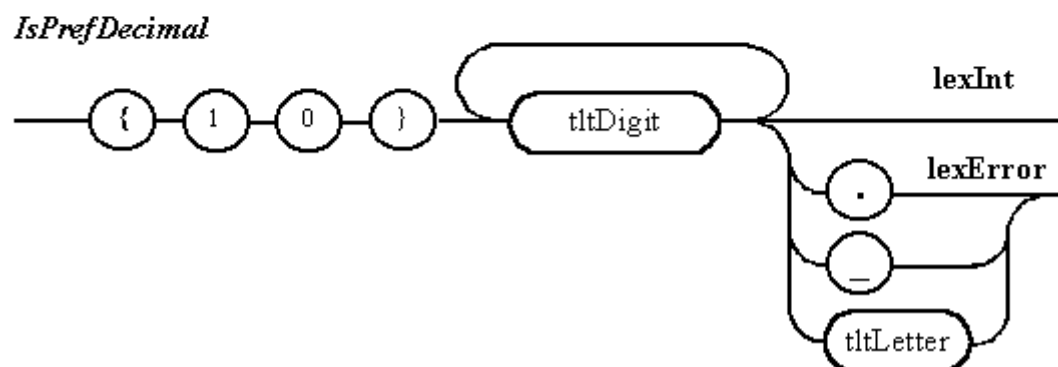
*Примечание 3.* Пять вариантов правил для распознавания действительного числа приводятся только для демонстрации арбитража при непрямом лексическом анализе. На практике легко можно обойтись одним правилом. Выдача ошибки происходит, если действительное число не отделяется разделителем от идентификатора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



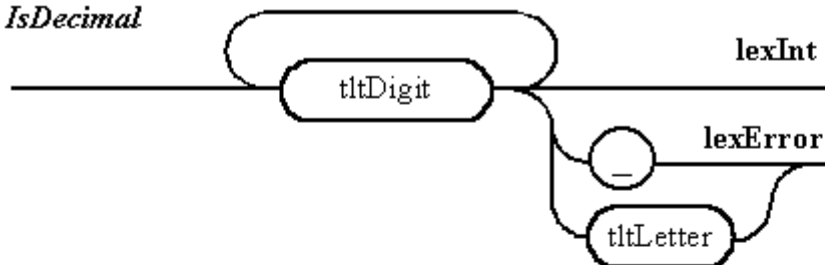
*Примечание 4.* Под остальными понимаются все символы, кроме апострофа (') и конца файла



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

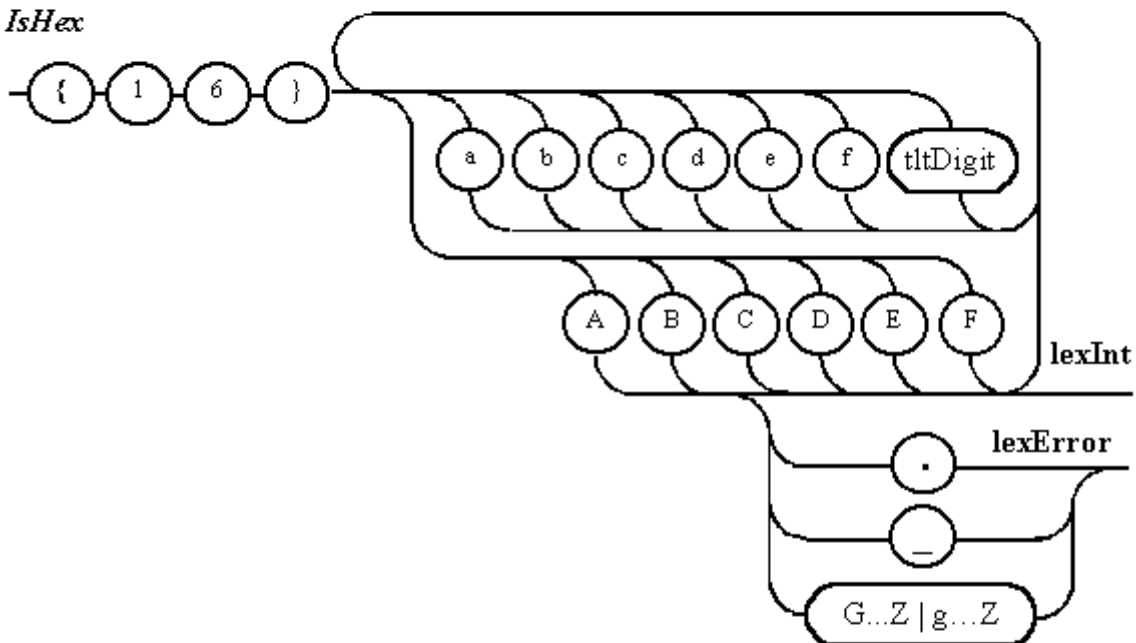


### IsDecimal



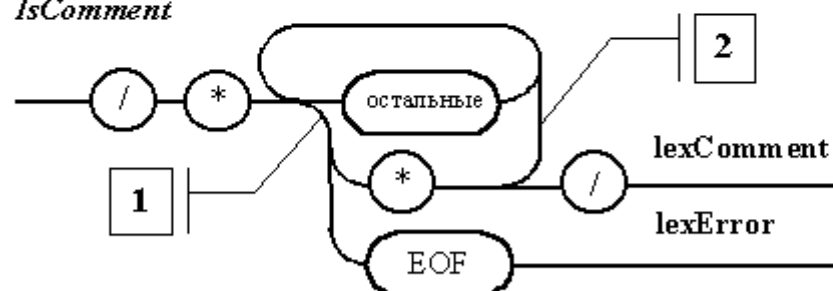
Примечание 6. Для целого десятичного числа без префикса анализ на недопустимость точки излишен, так как похожая ситуация должна была быть проанализирована раньше для действительного числа.

### IsHex

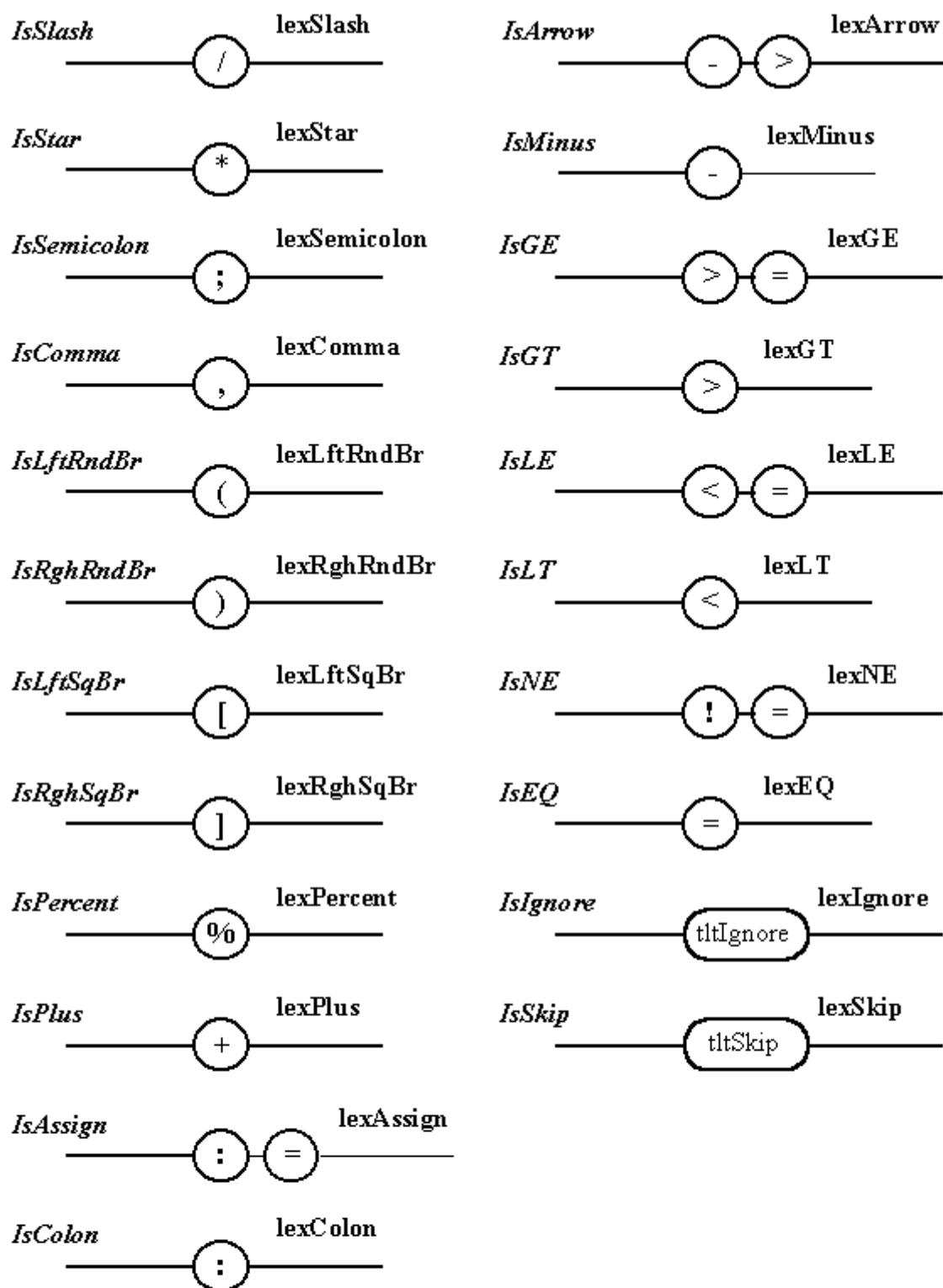


Примечание 7. Для целого шестнадцатеричного проверка на недопустимость должна исключать прописные и строчные буквы, используемые в самом числе. На представленных диаграммах это показано сокращенной записью путем задания диапазона. Это сделано для того, чтобы не загромождать диаграмму деталями.

### IsComment



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



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

Рис. 5.2. Описание с помощью диаграмм Вирата лексем, распознаваемых непрямым лексическим анализатором

## Программная реализация отдельных автоматов

Каждый автомат представлен соответствующей процедурой (функцией), осуществляющей имитацию переходов от начального состояния к одному из заключительных. Разметка диаграмм Вирта, соответствующая состояниям конечного автомата, была описана ранее. Принципы программной реализации в общем случае тоже достаточно просты:

- каждому состоянию ставится в соответствие метка;
- анализируемые альтернативы проверяются условными операторами (на мой взгляд, они подходят лучше переключателей из-за того, что проверяться могут значения символов и их классы);
- если результат проверки является истиной, проводится обработка контекста, берется следующий символ и осуществляется переход на новую метку (в новое состояние);
- процесс повторяется до тех пор, пока не произойдет переход в одно заключительных состояний.

Да, Вы правы, я без зазрения совести использую операторы безусловного перехода! И это после работы Дейкстры, заклеившей `goto`!! Как можно посягать на каноны программирования, когда в современных языках этот оператор просто не существует!!! А ведь Вы еще не видели программной реализации диаграмм, используемых в синтаксическом анализаторе!!!! Я не собираюсь с пеной у рта отстаивать используемое решение, но робко мотивировать его все же попробую.

**Во-первых**, основным документом, по которому разрабатываются программные модули, является диаграмма Вирта, которая, в общем случае является неструктурированным графом. А такие графы невозможно представить без `goto`, если только не осуществить соответствующие преобразования. Применение же этих преобразований может снизить наглядность исходных диаграмм, привести к их избыточности, что ведет к соответствующему "захламлению" и формируемого кода. Тогда проще сразу отказаться от диаграмм Вирта и перейти на РБНФ. Именно неструктурированные диаграммы Вирта диктуют мне ранее описанную регулярную схему моделирования каждой ее отдельной связи.

**Во-вторых**, используя данный подход, можно, с помощью экранного редактора, достаточно быстро набирать фрагменты новых правил, перенося отдельные связи из ранее набранных диаграмм. Особенно удобно такое редактирование при написании синтаксического анализатора.

**В третьих**, достаточно часто мне приходилось заниматься разработкой новых языков программирования, синтаксис которых изменялся уже во время написания программы. При использовании `goto` модификация связей диаграммы достаточно просто отображалась на изменение программного кода. Кроме того, изменение диаграмм возможно и при разработке трансляторов с уже существующих языков. Это, например, может быть связано со стремлением преобразовать грамматику к определенному виду.

**В четвертых**, формируемые синтаксические правила обычно порождают небольшой код, большая часть которого размещается в пределах страницы (экрана). Его достаточно легко охватить единым взглядом и сопоставить с нарисованной диаграммой. Поэтому, на мой взгляд, нагромождение операторов безусловного перехода в глаза особо не бросается. :-)

**В пятых**, одно время я увлеченно прорабатывал средства визуальной разработки трансляторов. А для таких средств особого значения не имеет вид выходного представления. Представленный выше метод порождения кода достаточно легко ложился на разрабатываемые структуры. Проект умер, а `goto` остались.

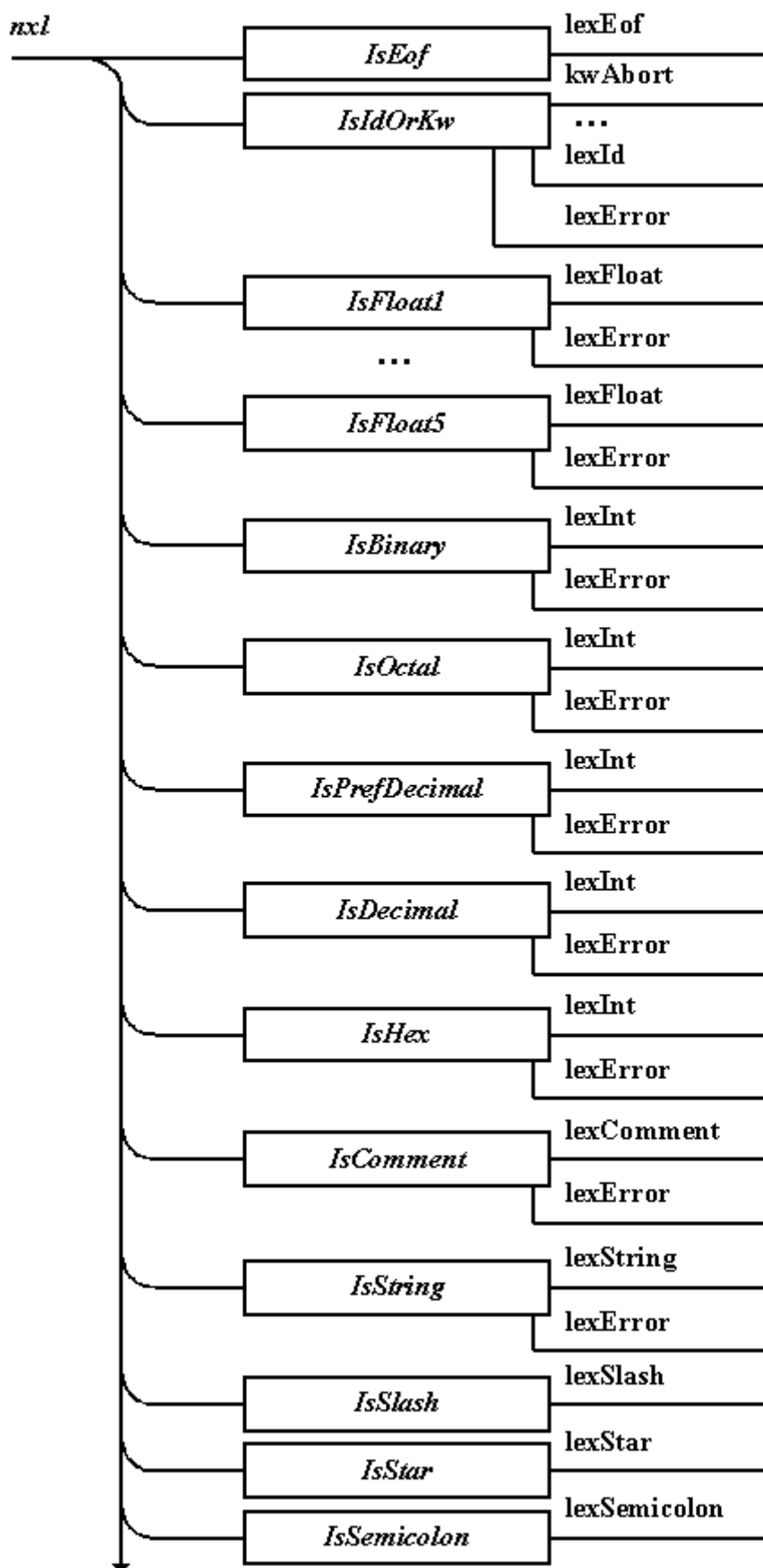
**В шестых**, даже Дональд Кнут указывал на полезность `goto` в ряде случаев (статья: Structured programming with GOTO statements. ACM Computing Surveys, 6, №4, 1974). Почему же мне не использовать его в том случае, когда код функции изначально может быть неструктурным?

Уф! Столько оправданий из-за какого-то маленького оператора! Надеюсь, что с Дейкстрой разобрался. Кто следующий? Гради Буч?! Я считаю, что все выше сказанное поясняет, почему я не считаю языками программирования те из них, в которых отсутствует `GOTO`. Еще лучше, если используется управляющий оператор перехода, как в Фортране, или переменные-метки, как в PL/1. :(Ой! Больно! Уж и пошутить нельзя!)

Ну а если серьезно, то каждый эволюционно вырабатывает свой стиль (о вкусах не спорят) или, при работе в коллективе, адаптируется к принятым групповым требованиям (в чужой монастырь...). Такова жизнь!

## Общая структура непрямого лексического анализатора

В завершение всего можно привести объединенную диаграмму Вирта, определяющую взаимосвязи между отдельными, ранее представленными автоматами (рис. 5.3). На ней отражается альтернативный порядок следования разбираемых лексем с учетом приоритетов, определяемых схемой арбитража.



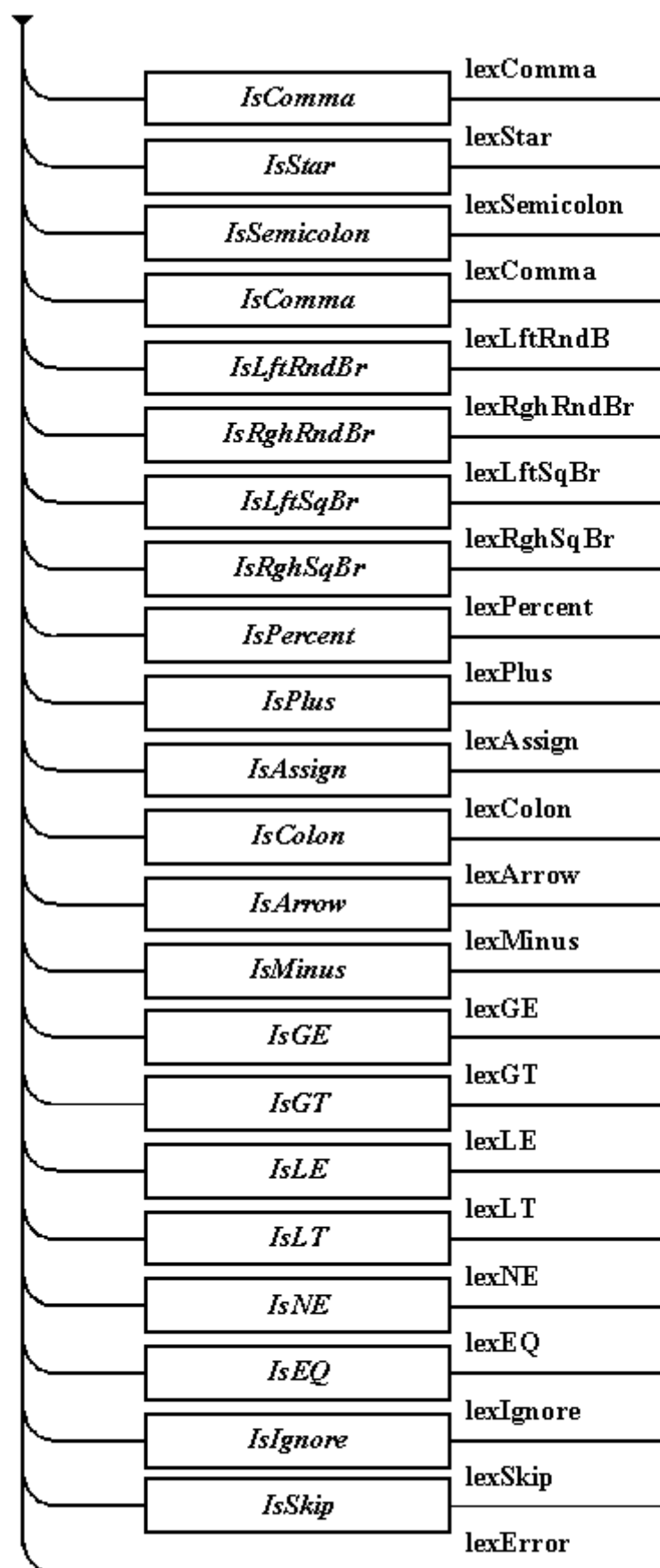


Рис. 5.3. Объединение диаграмм Вирта для непрямого лексического анализа в соответствии с приоритетом

В основной программе данной диаграмме соответствует функция nxl(), порождающая лексемы.

```
//-----  
// 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(si == EOF) {lc = lexEof; return;}  
        // Игнорируемую лексему не возвращаем  
        if(isSkip(si)) {nxsi(); lc = lexSkip; continue; /*return;*/}  
        if(id_etc()) {return;} unset();  
        if(string_const()) {return;} unset();  
        if(float1()) {return;} unset();  
        if(float2()) {return;} unset();  
        if(float3()) {return;} unset();  
        if(float4()) {return;} unset();  
        if(float5()) {return;} unset();  
        if(binary()) {return;} unset();  
        if(octal()) {return;} unset();  
        if(hex()) {return;} unset();  
        if(pdecimal()) {return;} unset();  
        if(decimal()) {return;} unset();  
        // Игнорируемую лексему не возвращаем  
        if(comment()) {continue; /*return;*/} unset();  
        // Игнорируемую лексему не возвращаем  
        if(isIgnore(si)) {nxsi(); lc = lexIgnore; continue; /*return;*/}  
        if(si=='/') {nxsi(); lc = lexSlash;return;}  
        if(si == ';') {nxsi(); lc = lexSemicolon; return;}  
        if(si == ',') {nxsi(); lc = lexComma; return;}  
        if(si == ':') {  
            nxsi();  
            if(si == '=') {nxsi(); lc = lexAssign; return;}  
        } unset();  
        if(si==':') {nxsi(); lc = lexColon; return;}  
        if(si == '(') {nxsi(); lc = lexLftRndBr; return;}  
        if(si == ')') {nxsi(); lc = lexRghRndBr; return;}  
        if(si == '[') {nxsi(); lc = lexLftSqBr; return;}  
        if(si == ']') {nxsi(); lc = lexRghSqBr; return;}  
        if(si == '*') {nxsi(); lc = lexStar; return;}  
    }
```



```

if(si == '%') {nxsi(); lc = lexPercent; return;}
if(si == '+') {nxsi(); lc = lexPlus; return;}
if(si == '-') {
    nxsi();
    if(si == '>') {nxsi(); lc = lexArrow; return;}
} unset();
if(si=='-') {nxsi(); lc=lexMinus; return;}
if(si == '=') {nxsi(); lc = lexEQ; return;}
if(si == '!') {
    nxsi();
    if(si == '=') {nxsi(); lc = lexNE; return;}
} unset();
if(si == '>') {
    nxsi();
    if(si == '=') {nxsi(); lc = lexGE; return;}
} unset();
if(si=='>') {nxsi(); lc=lexGT; return;}
if(si == '<') {
    nxsi();
    if(si == '=') {nxsi(); lc = lexLE; return;}
} unset();
if(si=='<') {nxsi(); lc=lexLT; return;}
lc = lexError; er(0); nxsi();
} while (lc == lexComment || lc == lexSkip || lc == lexIgnore);
}

```

Остановимся на ряде моментов. В сканер введен цикл, который отфильтровывает лексемы, не обрабатываемые распознавателем. Это комментарии, пропуски, игнорируемые символы.

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

```

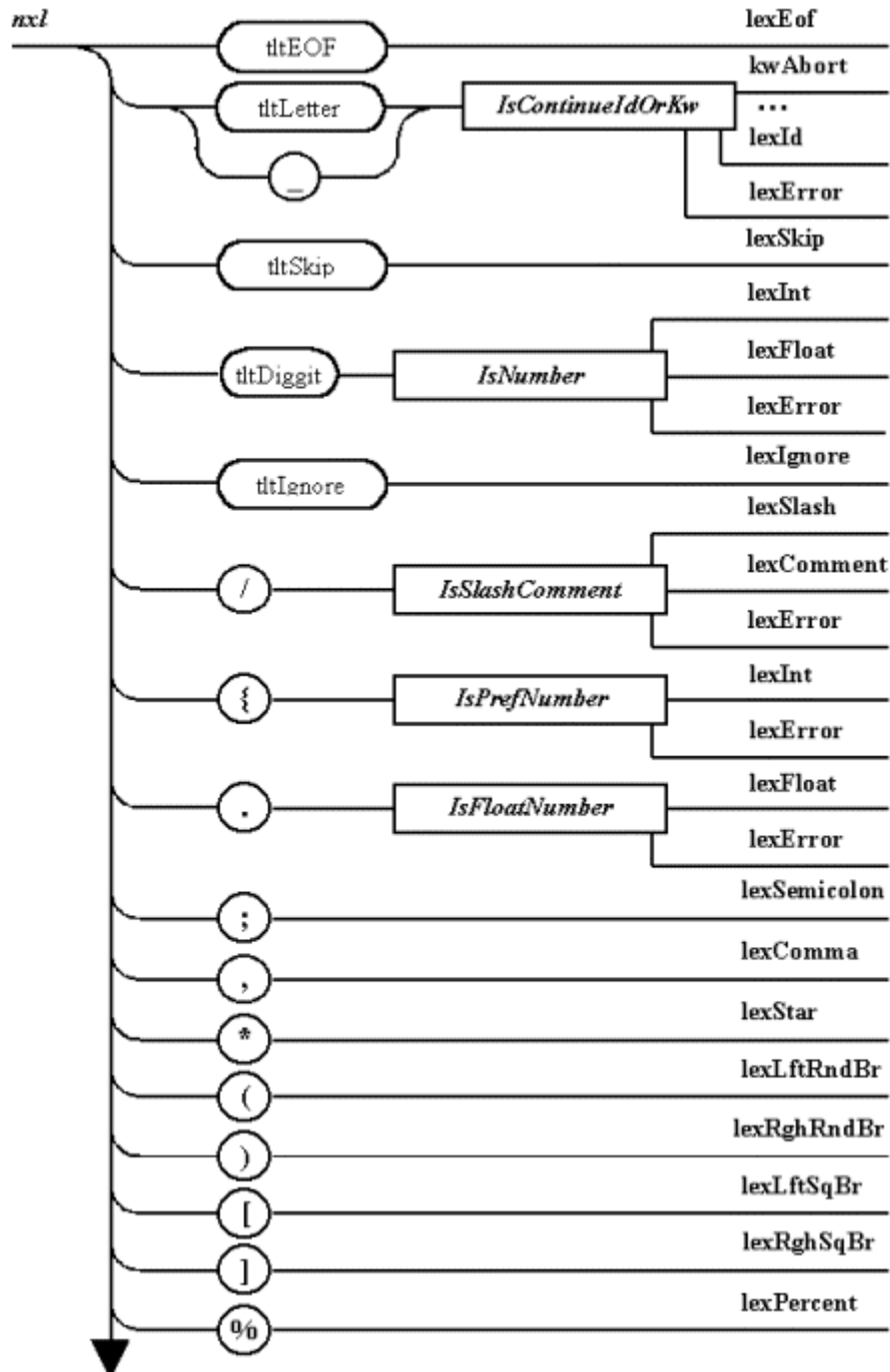
static void unset() {
    fseek(infil, oldpoz, 0);
    nxsi();
    i_lv=-1;
    lv[0]='\0';
    poz = oldpoz;
    line=oldline;
    column=oldcolumn;
}

```

Вот, в общем-то, и все. С реализацией отдельных диаграмм, процедурой обработки ошибок и тестовыми процедурами можно ознакомиться по представленным примерам.

# Прямой лексический анализатор DPL

При разработке прямого лексического анализатора поступают очень просто: все правила, используемые в этой фазе, сваливают в одну кучу, после чего начинается их группировка по группам, начинающихся с одинаковых начальных символов. Эти символы образуют набор альтернатив, анализируемых на первом шаге или, что одно и то же из начального состояния единого и плоского 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 на рис. 5.4.



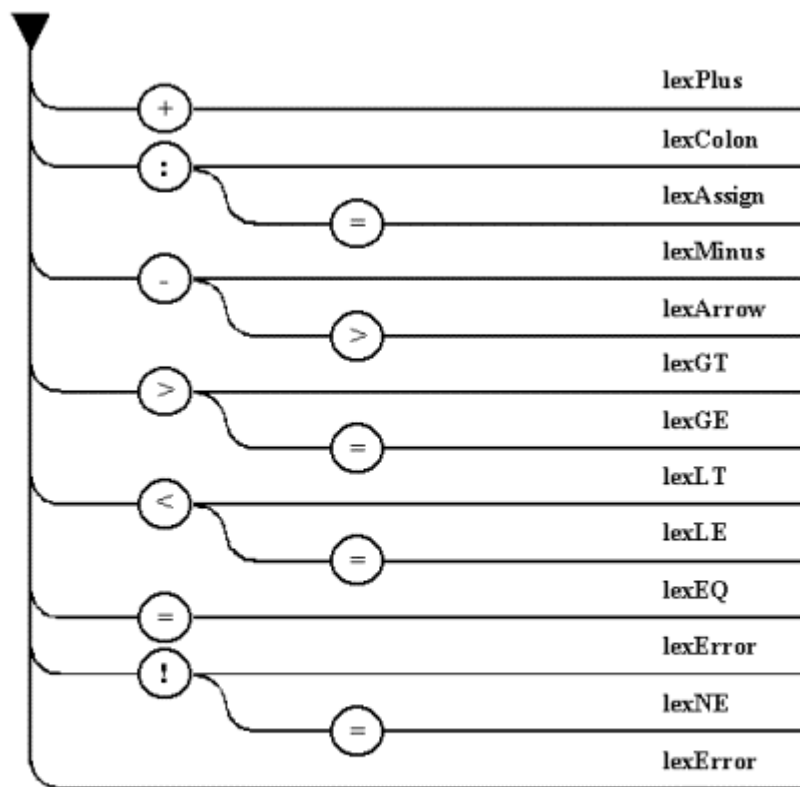


Рис. 6.5. Диаграммы Вирта, используемые для построения прямого лексического анализатора.

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(si == EOF) {lc = lexEof;}
    else if(isSkip(si)) {nxsi(); lc = lexSkip;}
    else if(isLetter(si) || si == '_'){
        lv[++i_lv]=si; nxsi(); id_etc();
    }
    else if(isDigit(si)) {number();}
    else if(isIgnore(si)) {nxsi(); lc = lexIgnore;}
    else if(si == '/') {nxsi(); divcom();}
    else if(si == "\"") {nxsi(); string_const();}
    else if(si == ';') {nxsi(); lc = lexSemicolon;}
    else if(si == ',') {nxsi(); lc = lexComma;}
    else if(si == ':') {
        nxsi();
        if(si == '=') {nxsi(); lc = lexAssign;}
        else lc = lexColon;
    }
}
```

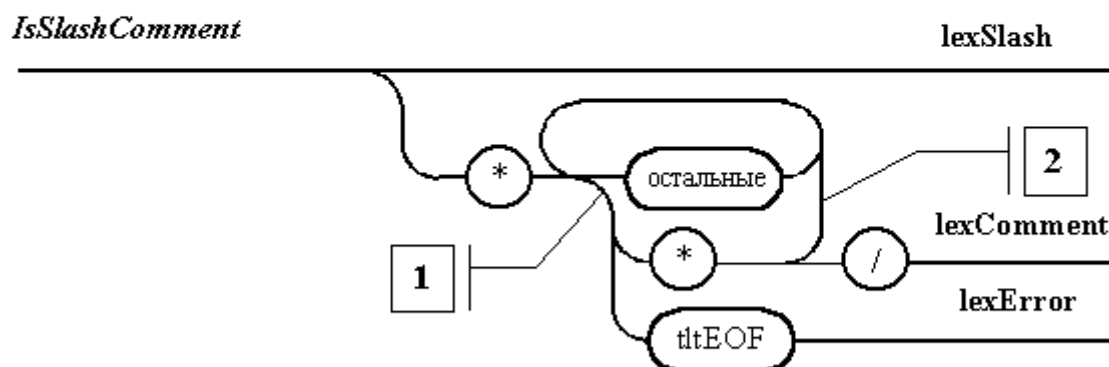
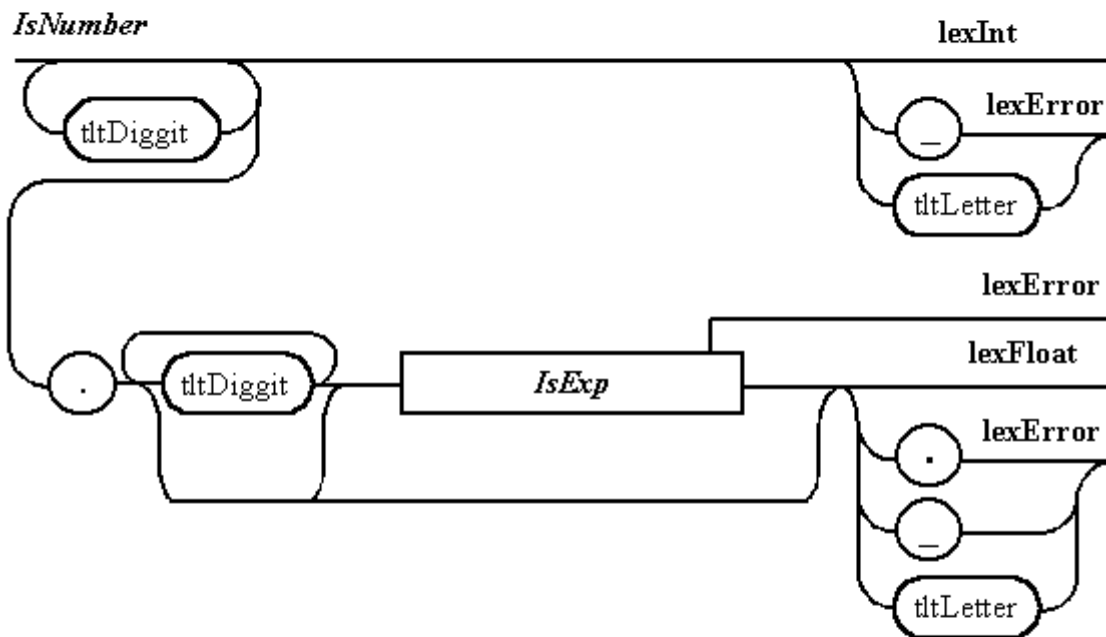
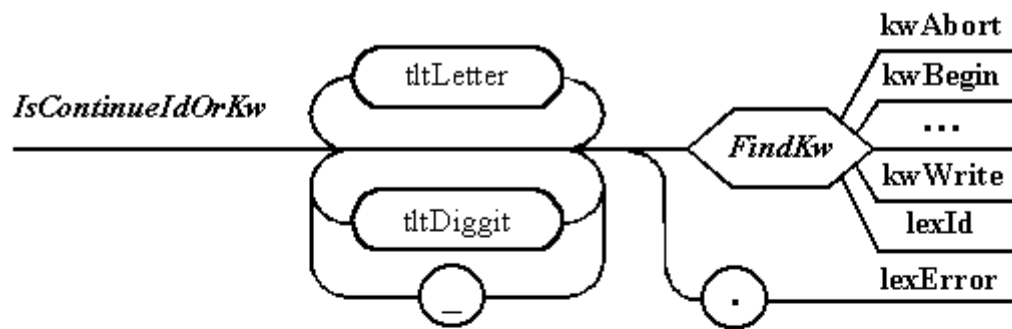
```

else if(si == '(') {nxsi(); lc = lexLftRndBr;}
else if(si == ')') {nxsi(); lc = lexRghRndBr;}
else if(si == '[') {nxsi(); lc = lexLftSqBr;}
else if(si == ']') {nxsi(); lc = lexRghSqBr;}
else if(si == '*') {nxsi(); lc = lexStar;}
else if(si == '%') {nxsi(); lc = lexPercent;}
else if(si == '+') {nxsi(); lc = lexPlus;}
else if(si == '-') {
    nxsi();
    if(si == '>') {nxsi(); lc = lexArrow;}
    else lc = lexMinus;
}
else if(si == '=') {nxsi(); lc = lexEQ;}
else if(si == '!') {
    nxsi();
    if(si == '=') {nxsi(); lc = lexNE;}
    else {lc = lexError; er(1);}
}
else if(si == '>') {
    nxsi();
    if(si == '=') {nxsi(); lc = lexGE;}
    lc = lexGT;
}
else if(si == '<') {
    nxsi();
    if(si == '=') {nxsi(); lc = lexLE;}
    lc = lexLT;
}
else if(si == '{') {nxsi(); prenumber();}
else if(si == '.') {lv[++i_lv]=si; nxsi(); fltnumber2();}
else {lc = lexError; er(0); nxsi();}
} while (lc == lexComment || lc == lexSkip || lc == lexIgnore);
}

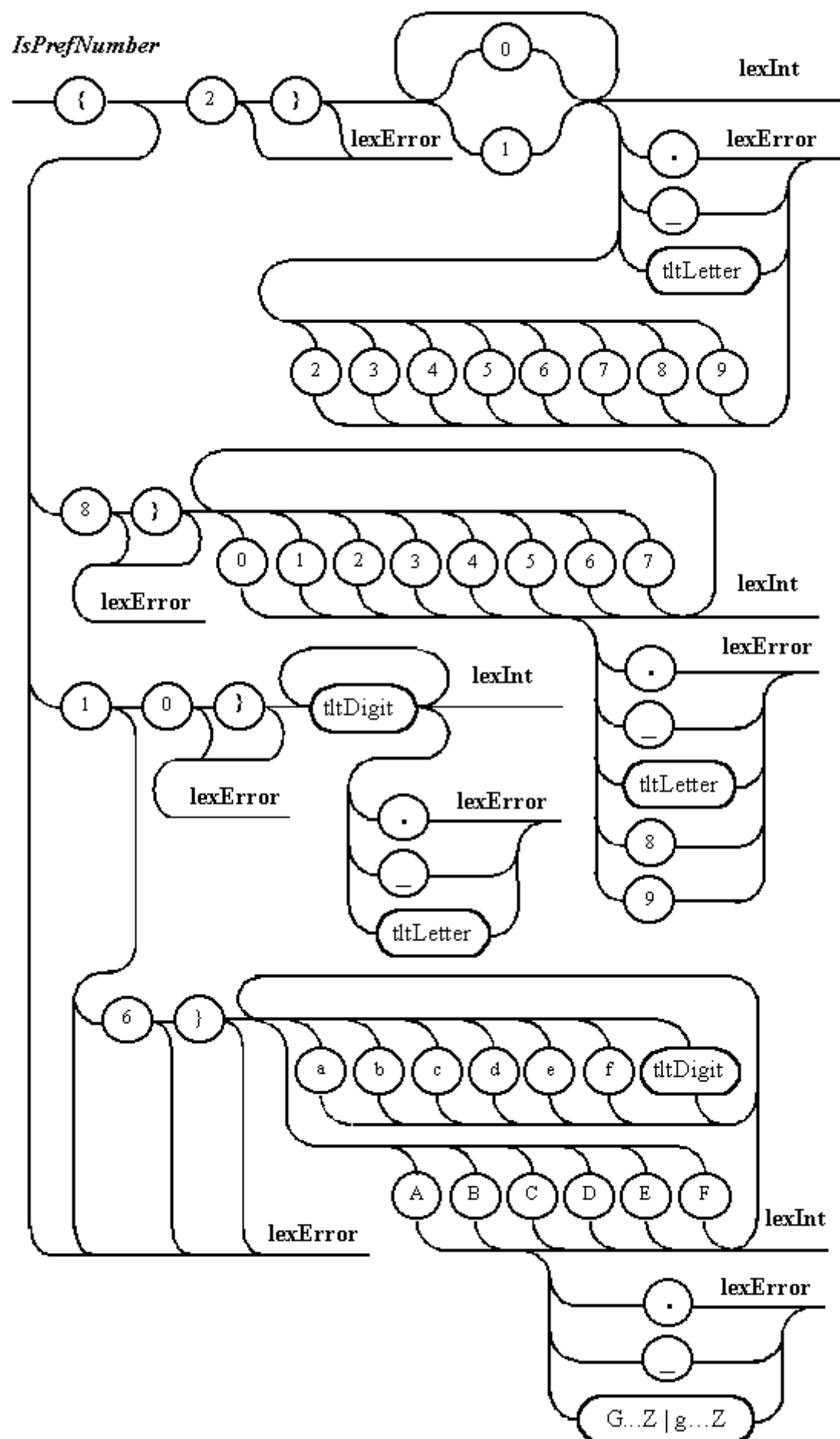
```

Напоминаю, что при взятии для анализа и транслитерации следующего символа, не надо учитывать позицию в файле.

Диаграммы Вирта, описывающие отдельные подавтоматы прямого лексического анализатора, представлены на рис. 5.5. Их программную реализацию проще всего изучить на предлагаемых для загрузки исходных текстах



*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 «/»



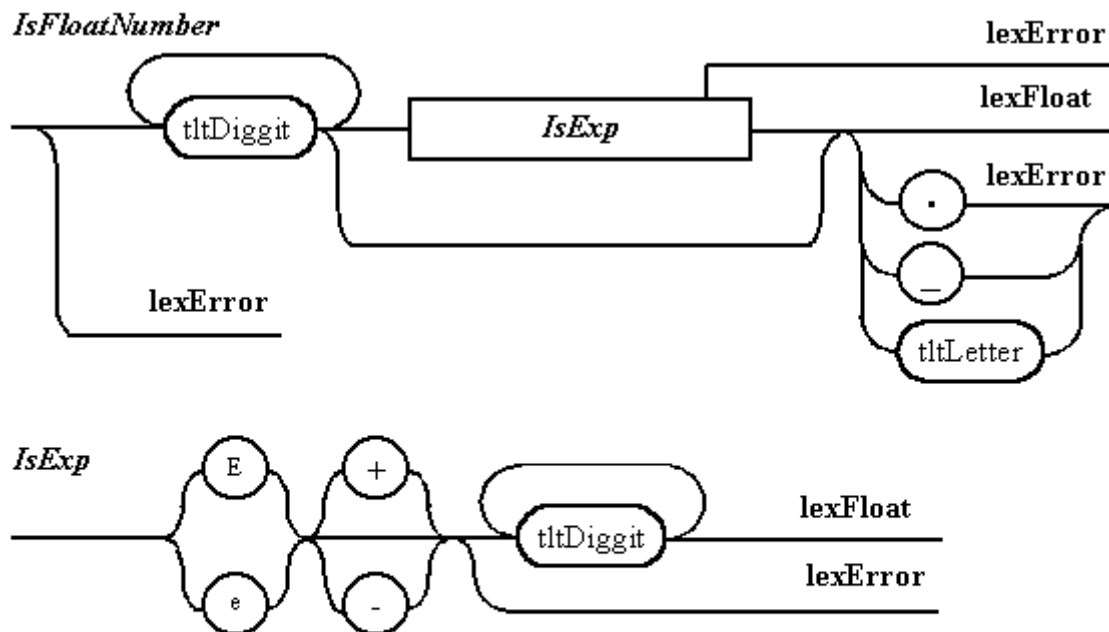


Рис. 5.5. Диаграммы Вирта, описывающие отдельные крупные фрагменты правил прямого лексического анализатора.

## Контрольные вопросы и задания

1. Разработать лексический анализатор в соответствии с вариантом задания, полученным для выполнения лабораторных работ.

# Тема 6. Общие принципы организации синтаксического разбора

## Содержание темы

Назначение синтаксического разбора. Классификация методов синтаксического разбора.

### Назначение синтаксического разбора

Синтаксический разбор (распознавание) является первым 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. Затем проводится добавление новых объектов в объектную модель программы или в промежуточное представление.

Разбор предназначен для доказательства того, что анализируемая входная цепочка, записанная на входной ленте, принадлежит или не принадлежит множеству цепочек порождаемых грамматикой данного языка. Выполнение синтаксического разбора осуществляется распознавателями, являющимися автоматами. Поэтому данный процесс также называется распознаванием входной цепочки. Цель доказательства в том, чтобы ответить на вопрос: принадлежит ли анализируемая цепочка множеству правильных цепочек заданного языка. Ответ "да" дается, если такая принадлежность установлена. В противном случае дается ответ "нет". Получение ответа "нет" связано с понятием отказа. Единственный отказ на любом уровне ведет к общему отказу.

Чтобы получить ответ "да" относительно всей цепочки, надо его получить для каждого правила, обеспечивающего разбор отдельной подцепочки. Так как множество правил образуют иерархическую структуру, возможно с рекурсиями, то процесс получения общего положительного ответа можно интерпретировать как сбор по определенному принципу ответов для листьев, лежащих в основе дерева разбора, что дает положительный ответ для узла, содержащего эти листья. Далее анализируются обработанные узлы, и уже в них полученные ответы складываются в общий ответ нового узла. И так далее до самой вершины. Данный принцип обработки сильно напоминает бюрократическую систему, используемую в организационном управлении любого предприятия. Так поднимается вверх информация, подтверждающая выполнение указания начальника организации. До этого, теми же путями, вниз спускалось и разделялось исходное указание.

### Классификация методов синтаксического разбора

Если попытаться формализовать задачу на уровне элементарного метаязыка, то она будет ставиться следующим образом. Дан язык  $L(G)$  с грамматикой  $G$ , в которой  $S$  - начальный нетерминал. Построить дерево разбора входной цепочки

$$a = a_1 a_2 a_3 \dots a_n.$$

Естественно, что существует огромное количество путей решения данной задачи, и целью разработчика распознавателя является выделение приемлемых вариантов его реализации. Для того, чтобы понять, что, и каким образом, влияет на принципы функционирования распознавателя, а следовательно, и на организацию разбора, рассмотрим некоторые возможные варианты. Общая классификация рассматриваемых вариантов построения распознавателя представлена на рис. 6.1.



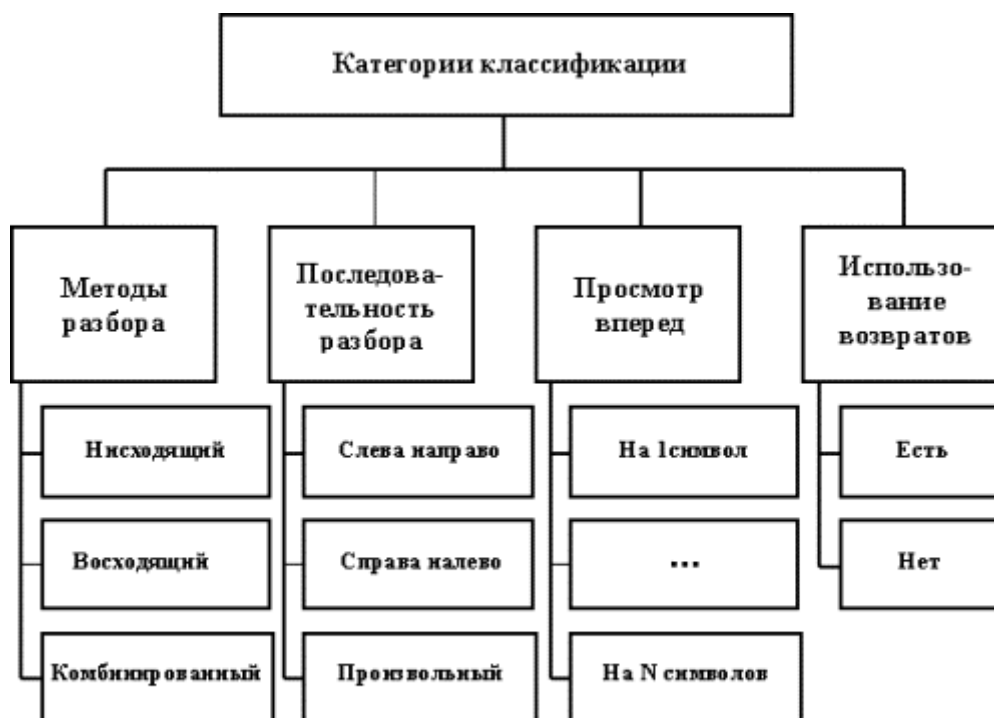


Рис. 6.1. Классификация методов организации синтаксического разбора.

На самом верхнем уровне выделяются:

- методы разбора;
- последовательность разбора;
- использование просмотра вперед;
- использование возвратов.

## Методы разбора

Выделяются два основных метода синтаксического разбора:

- нисходящий разбор;
- восходящий разбор.

Кроме этого можно использовать комбинированный разбор, сочетающий особенности двух предыдущих.

Нисходящие и восходящие подходы широко используются в различных областях человеческой деятельности, особенно в тех из них, которые связаны с анализом и синтезом искусственных систем. В частности, можно отметить методы разработки программного обеспечения сверху вниз (нисходящий) и снизу вверх (восходящий).

**Нисходящий разбор** заключается в построении дерева разбора, начиная от корневой вершины. Разбор заключается в заполнении промежутка между начальным нетерминалом и символами входной цепочки правилами, выводимыми из начального нетерминала. Подстановка основывается на том факторе, что корневая вершина является узлом, состоящим из листьев, являющихся цепочкой терминалов и нетерминалов одного из альтернативных правил, порождаемых начальным нетерминалом.

Подставляемое правило в общем случае выбирается произвольно. Вместо новых нетерминальных вершин осуществляется подстановка выводимых из них правил. Процесс протекает до тех пор, пока не будут установлены все связи дерева, соединяющие корневую вершину и символы входной цепочки, или пока не будут перебраны все возможные комбинации правил. В последнем случае входная цепочка отвергается. Построение дерева разбора подтверждает принадлежность входной цепочки данному языку. При этом, в общем случае, для одной и той же входной цепочки может быть построено несколько деревьев разбора. Это говорит о том, что грамматика данного языка является недетерминированной.

Эти рассуждения иллюстрируются следующим примером. Пусть будет дана грамматика G:

$$G_6 = (\{S\}, \{a, +, *\}, P, S)$$

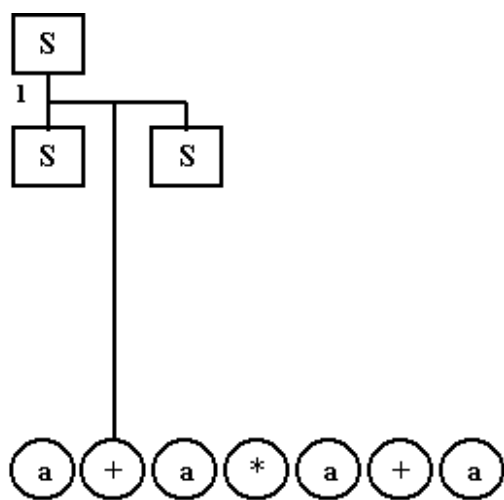
Где P определяется как:

1.  $S \rightarrow a$
2.  $S \rightarrow S + S$
3.  $S \rightarrow S * S$

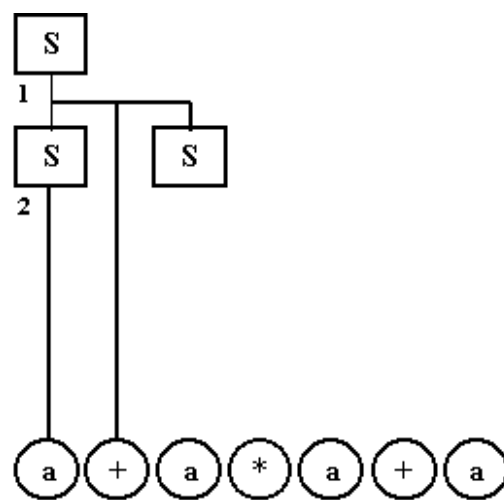
Цепочки, порождаемые данной грамматикой можно интерпретировать как выражения, состоящие из операндов "a", а также операций "+" и "\*". Недетерминированность грамматики позволяет порождать одну и ту же терминальную цепочки с использованием различных выводов. Например, выражение "a+a\*a+a" можно получить следующими способами:

1.  $S \Rightarrow S+S \Rightarrow a+S \Rightarrow a+S*S \Rightarrow a+a*S \Rightarrow a+a*S+S \Rightarrow a+a*a+S \Rightarrow a+a*a+a$
2.  $S \Rightarrow S+S \Rightarrow S+a \Rightarrow S*S+a \Rightarrow S*a+a \Rightarrow S+S*a+a \Rightarrow S+a*a+a \Rightarrow a+a*a+a$  (6.1)
3.  $S \Rightarrow S*S \Rightarrow S+S*S \Rightarrow S+S*S+S \Rightarrow a+S*S+S \Rightarrow a+a*S+S \Rightarrow a+a*S+a \Rightarrow a+a*a+a$

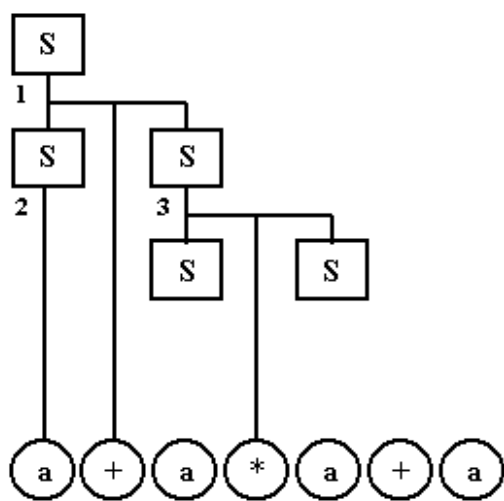
И так далее. В этом пример число вариантов одной и той же произвольной цепочки вывода настолько велико, что не имеет и смысла говорить о практическом применении данной грамматики. Но в данном случае она позволяет показать, каким образом могут порождаться различные деревья при нисходящем разборе. Пошаговое построение различных деревьев показано на рис. [6.2](#), [6.3](#), [6.4](#). Можно отметить, что процесс построения дерева совпадает с последовательностью шагов вывода входной цепочки.



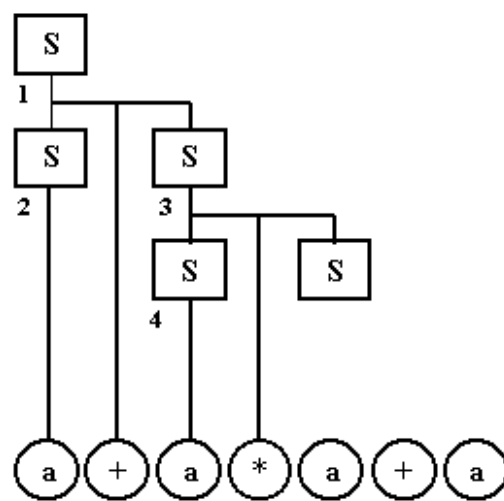
а) III ar 1



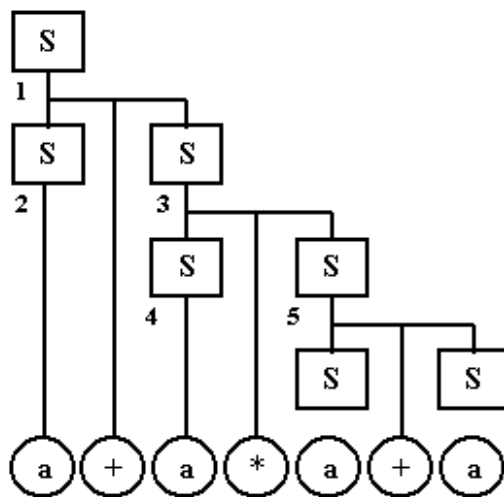
б) III ar 2



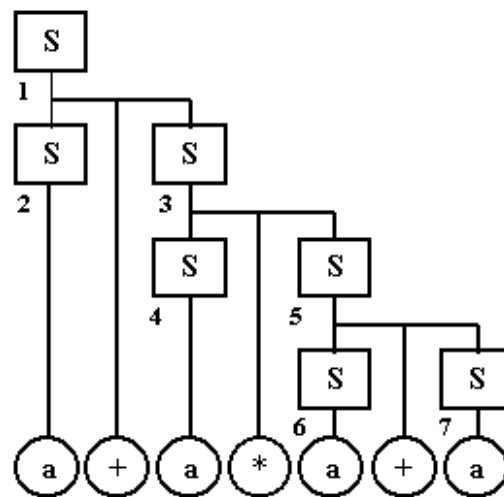
в) III ar 3



г) III ar 4

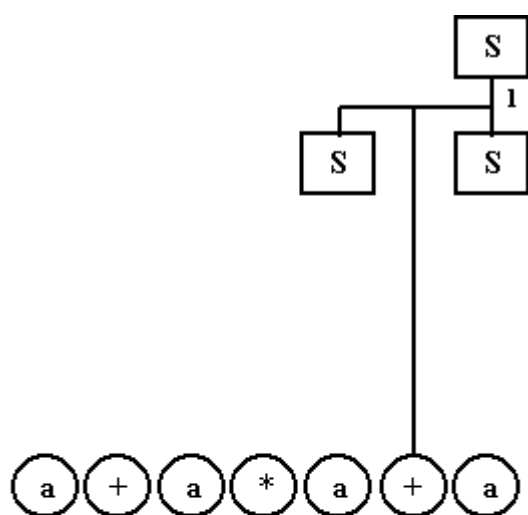


д) III ar 5

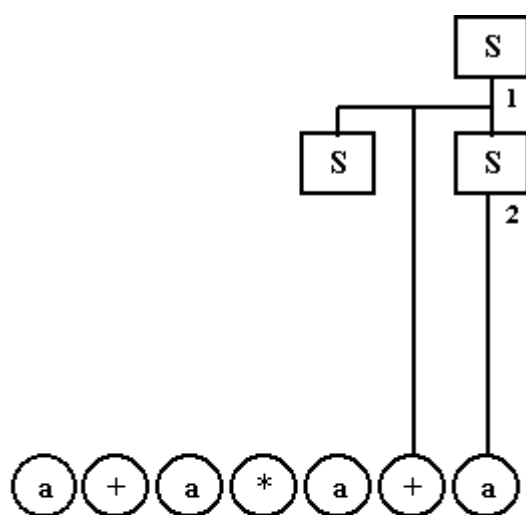


е) Шаги 6-7

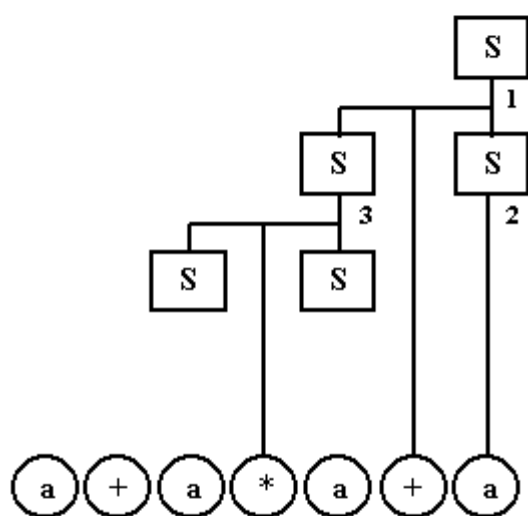
Рис. 6.2. Нисходящий разбор слева направо.



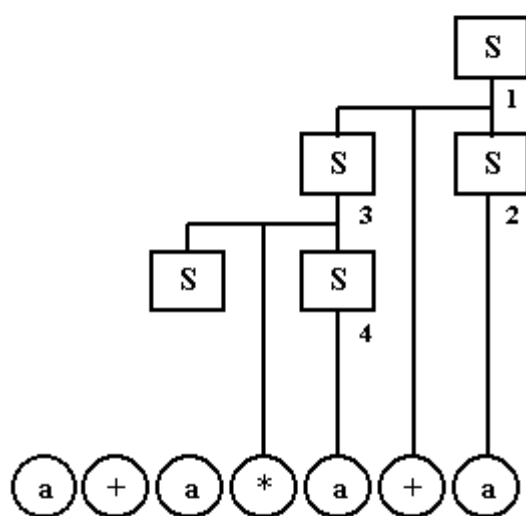
а) Шаг 1



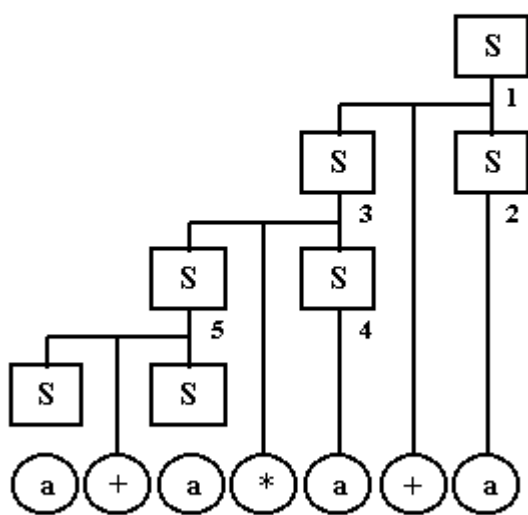
б) Шаг 2



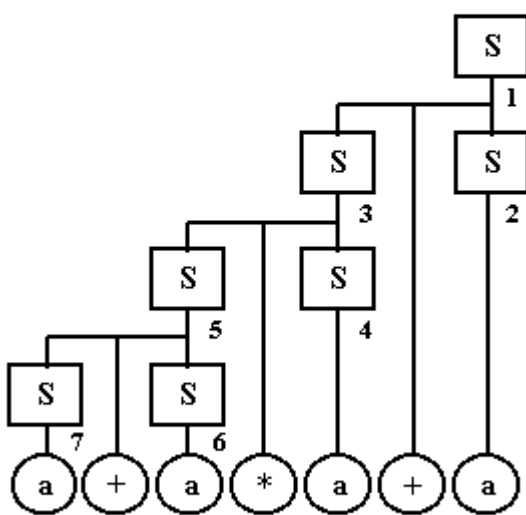
в) Шаг 3



г) Шаг 4



д) Шаг 5



е) Шаги 6-7

Рис. 6.3. Нисходящий разбор справа налево.

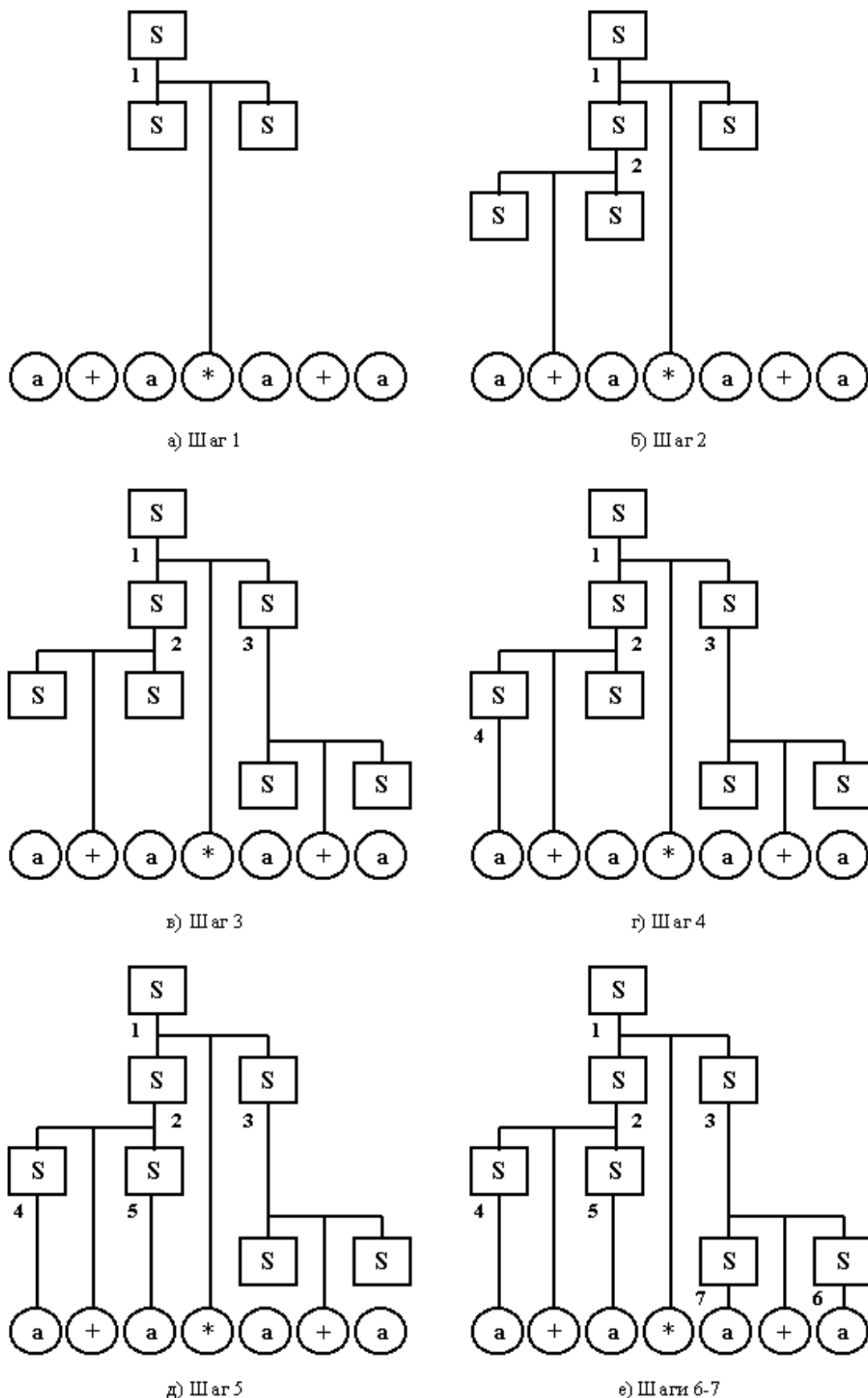


Рис. 6.4. Нисходящий произвольный разбор, с операции умножения.

При **восходящем разборе** дерево начинает строиться от терминальных листьев путем подстановки правил, применимых к входной цепочке, опять таки, в общем случае, в произвольном порядке. На следующем шаге новые узлы полученных поддеревьев используются как листья во вновь применяемых правилах. Процесс построения дерева разбора завершается, когда все символы входной цепочки будут являться листьями дерева, корнем которого окажется начальный нетерминал. Если, в результате полного перебора всех возможных правил, мы не сможем построить требуемое дерево разбора, то рассматриваемая входная цепочка не принадлежит данному языку.

Восходящий разбор также непосредственно связан с любым возможным выводом цепочки из начального нетерминала. Однако, эта связь, по сравнению с нисходящим разбором, реализуется с точностью до "наоборот". На рис. 6.5, 6.6, 6.7 приведены примеры построения деревьев разбора для грамматики  $G_6$  и процессов порождения цепочек, представленных выражениями (6.1). Из рисунков видно, что шаги порождения дерева соответствуют движению по представленным цепочкам вывода справа налево.

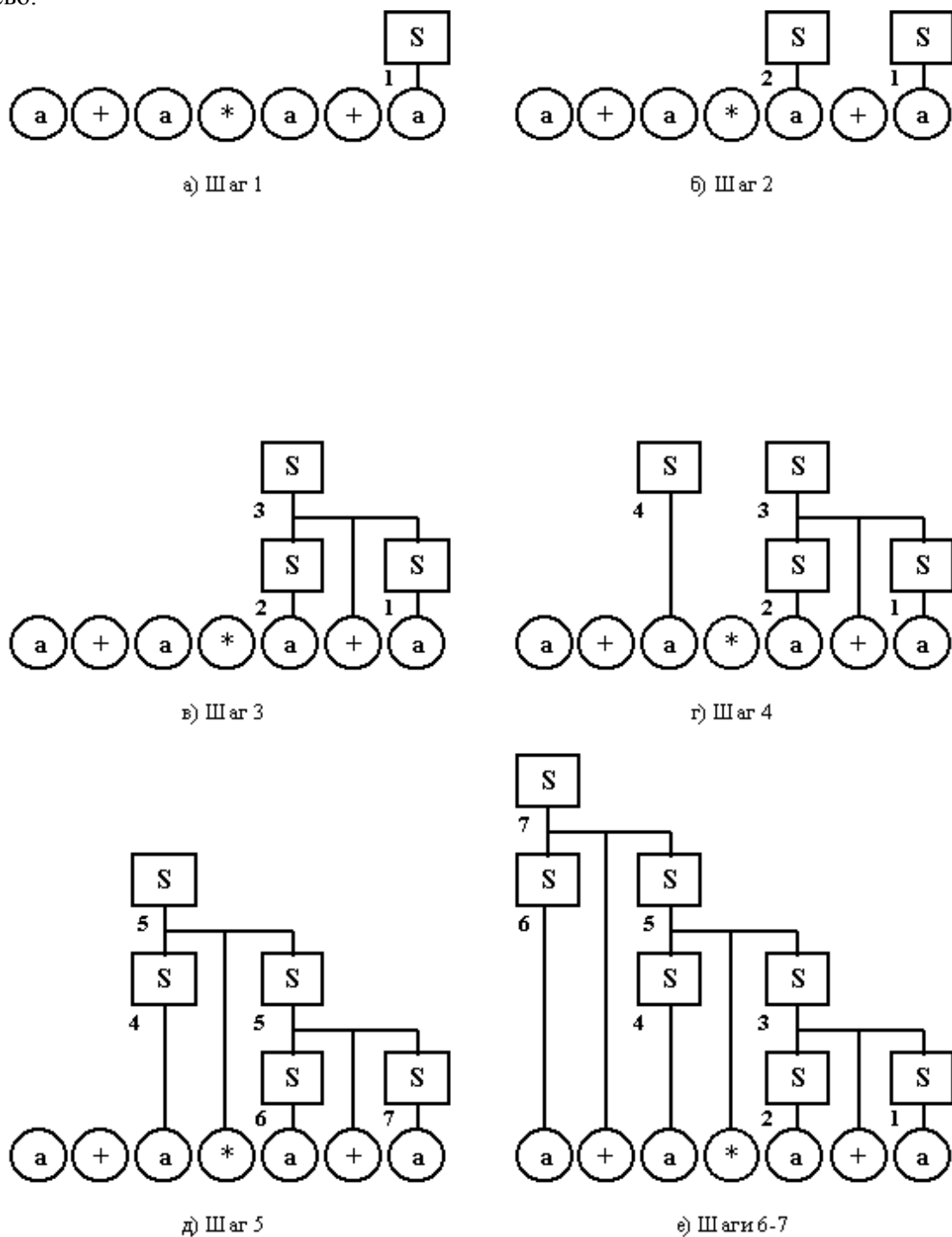
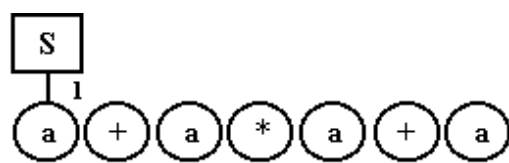
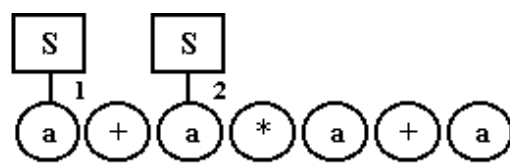


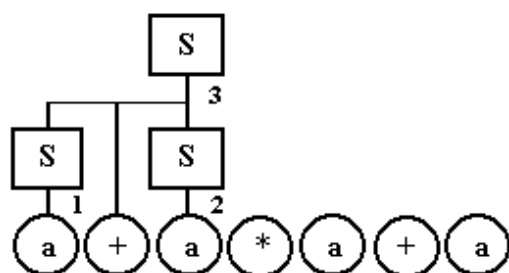
Рис. 6.5. Восходящий разбор слева направо



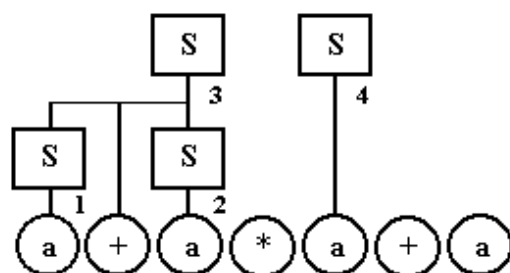
а) Шаг 1



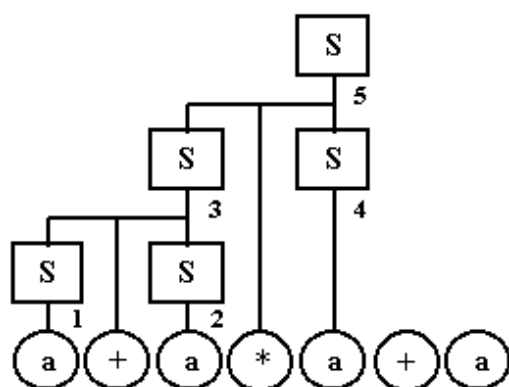
б) Шаг 2



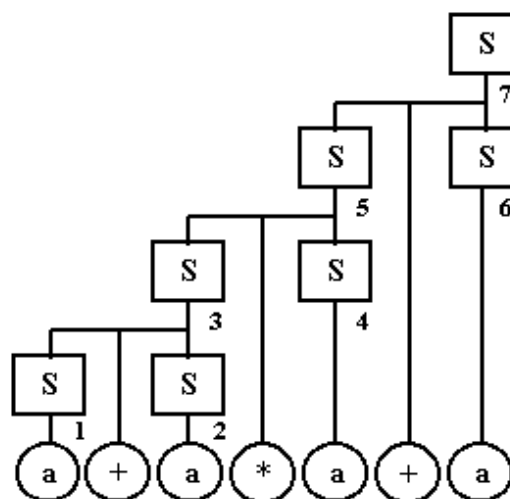
в) Шаг 3



г) Шаг 4



д) Шаг 5



е) Шаги 6-7

Рис. 6.6. Восходящий разбор справа налево.

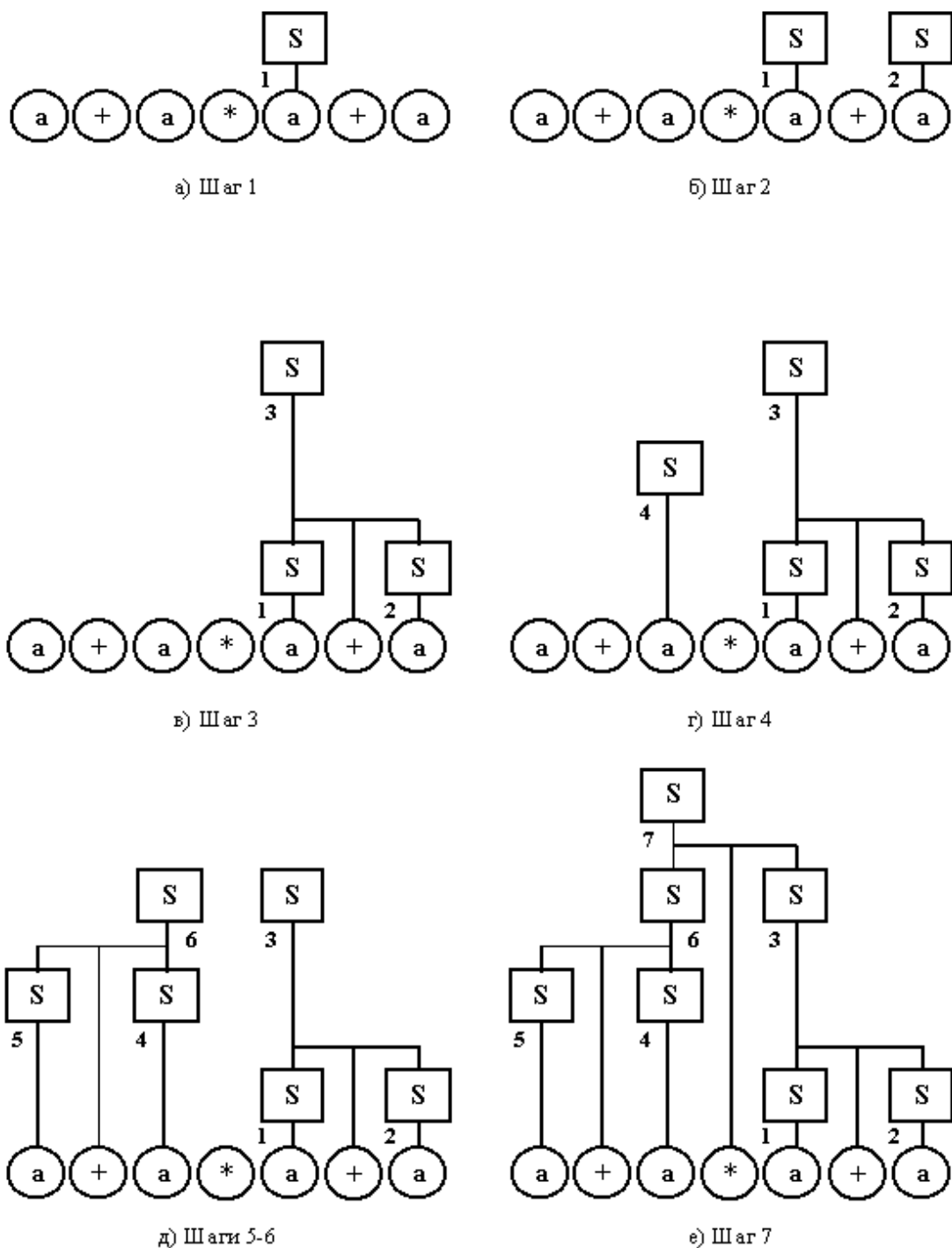


Рис. 6.7. Восходящий произвольный разбор.

Комбинированный разбор может быть реализован тогда, когда процесс распознавания разбивается на два этапа. На одном из них осуществляется нисходящий, а на другом - восходящий разбор. Этапов может быть и больше, а порядок их применения - произвольным. Комбинированным можно считать разбор в любом трансляторе, если фазу лексического анализа принять за первый этап, а синтаксического - за второй. При этом лексический анализатор нельзя считать истинным распознавателем, так как осуществляется формирование только одного слоя ветвей в дереве разбора, расположенного над символами входной цепочки (рис. 6.8).



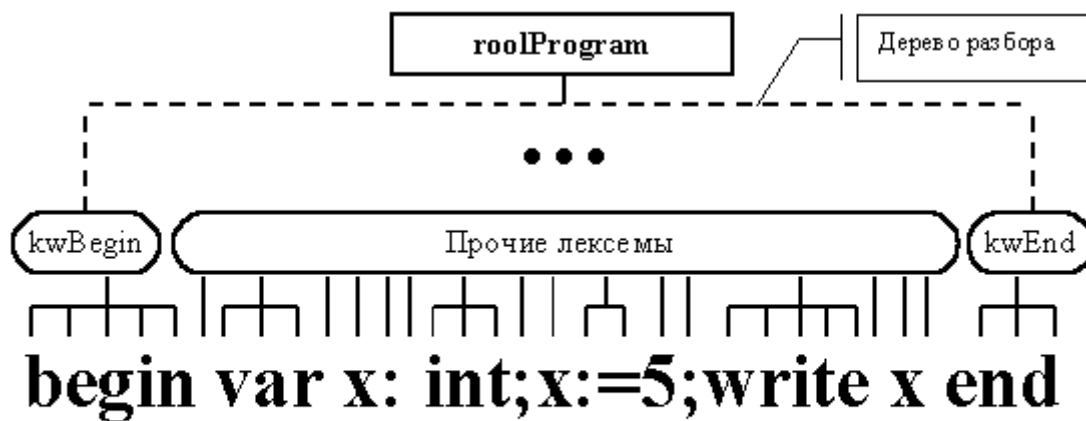


Рис. 6.8. Пример комбинированного разбора, когда в роли распознавателя самых нижних ветвей дерева выступает лексический анализатор.

Прямой лексический анализатор будет являться восходящим распознавателем, который может сочетаться с нисходящим. Непрямой же лексический анализ можно рассматривать как нисходящий разбор некоторой подцепочки (осуществляются проверки версий). Поэтому комбинированный разбор будет при его совместном использовании с восходящим распознавателем. В синтаксическом же анализаторе комбинация различных видов разборов обычно не используются, так как в этом нет особого смысла.

### Последовательность разбора.

Последовательность разбора определяет, каким образом осуществляется формирование фрагмента дерева разбора на каждом шаге подстановки. Эти подстановки могут осуществляться **слева направо, справа налево, произвольно**. Различные последовательности разбора можно рассмотреть на примере грамматики  $G_6$  и выводов одной и той же цепочки, описанных выражениями 6.1. В первом примере представлен вывод слева направо, когда порождение новой цепочки на каждом шаге осуществляется для самого левого нетерминала. Как только из самого левого нетерминала порождается терминальная цепочка, осуществляется переход к нетерминалу, расположенному правее. Второй пример иллюстрирует выполнение вывода справа налево. Третий и четвертый пример иллюстрируют произвольный порядок вывода. Следует отметить, что использование упорядоченного разбора ускоряет его выполнение за счет уменьшения числа перебираемых правил.

Последовательность разбора непосредственно сочетается с методом разбора. Так, при нисходящем разборе слева направо, подстановка правил вместо самых левых нетерминалов ведет к тому, что входная цепочка распознается с ее начала ([рис. 6.2](#)). Нисходящий разбор справа налево ведет к первоначальному подтверждению символов с конца цепочки ([рис. 6.3](#)). Наоборот, восходящий разбор слева направо осуществляет замену на нетерминал символов, расположенных в конце цепочки ([рис. 6.5](#)). Замена начальных символов производится при восходящем разборе справа налево ([рис. 6.6](#)). Произвольный разбор не оговаривает последовательность подстановки правил ([рис. 6.4, 6.7](#)). Это ведет к большому количеству переборов.

Повышение эффективности разбора осуществляется разработкой грамматик, специально поддерживающих согласованные между собой метод и последовательность. Так, например, Грамматики, предназначенные для нисходящего разбора, обычно используются для вывода слева направо. Следовательно, и входная цепочка будет разбираться слева направо. Это позволяет быстрее получать нужные символы, а не ждать конца цепочки и лишь потом осуществлять разбор (можно, конечно, саму цепочку разбирать от конца к началу, но это уже из области симметрии). Грамматики, ориентированные на восходящий разбор, обычно оптимизированы под правый вывод входной цепочки, что позволяет, при синтаксическом разборе, опять-таки осуществлять подстановки нетерминалов слева направо.

### Использование просмотра вперед.

Языки (как и другие системы) бывают различными по сложности. Ряд из них практически невозможно описать с помощью простых грамматик. Поэтому, в грамматиках могут встречаться

альтернативные правила, начинающиеся с одинаковых цепочек символов. Возникающая неоднозначность может быть разрешена путем предварительного просмотра правила на  $n$  символов вперед до той границы, начиная с которой данное правило можно будет отличить от других. В контекстно свободных (КС) грамматиках число, определяющее количество символов, анализируемых перед выбором подстановки (1, 2...), используется для классификации. По этому критерию КС грамматики, задаются следующим образом: КС(1), КС(2),...

Просмотр вперед - это один из возможных вариантов упорядочивания подстановок, обеспечивающий решение проблемы недетерминированности. Наряду с ним используются: преобразование грамматик к детерминированным и анализ с возвратами.

## **Использование возвратов**

Синтаксический разбор с возвратами выполняется аналогично тому, как осуществляется непрямым лексический анализ. Возвраты производятся для альтернативных правил, начинающихся с одинаковых подцепочек. В этом случае появление отказа при разборе правила ведет к восстановлению входной головки в то положение, в котором она была до входа в данное правило. Использование возвратов может выступать в качестве альтернативы просмотру вперед. Приоритет правил, определяющий порядок их обхода, назначается также как и при лексическом анализе и зависит от того, является ли некоторое правило подмножеством другого. Метод универсален и легок для понимания и реализации. Однако, такой подход замедляет разбор и может вести к дополнительным издержкам во время семантического анализа и построения объектной модели.

## **Резюме.**

Приведенные варианты организации синтаксического разбора могут встречаться в разнообразных сочетаниях. Однако, при разработке языка программирования всегда необходимо искать разумный компромисс. Несмотря на разнообразие факторов, влияющих на особенности организации разрабатываемого языка программирования, всегда надо учитывать, что:

- чем сложнее синтаксис языка программирования, тем сложнее его грамматика;
- чем сложнее грамматика, тем больше сложность и универсальность методов разбора;
- универсальные методы разбора ведут к более медленному его выполнению.

Поэтому, от начала процесса разработки языка программирования до его реализации постоянно необходимо обеспечивать баланс между сложностью замысла и простотой реализации. Следует также отметить, что усложнение синтаксиса не всегда обеспечивает более удобное восприятие конструкций языка пользователем. Зачастую можно иметь простой синтаксис и удобный для использования язык, поддерживаемый очень быстрым транслятором.

## **Контрольные вопросы и задания**

1. Назначение синтаксического разбора.
2. Что является результатом синтаксического разбора?
3. Назовите основные 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.

## Тема 7. Применение автоматов с магазинной памятью для нисходящего разбора слева направо

### Содержание темы

Необходимость использования автоматов с магазинной памятью. Организация автомата с магазинной памятью. Общая связь между грамматиками и автоматами с магазинной памятью. Связь между S-грамматикой и автоматом с магазинной памятью. Построение автомата с магазинной памятью по q-грамматике. LL(1) - грамматики Программная реализация нисходящего автомата с магазинной памятью.

### Необходимость использования автоматов с магазинной памятью

Синтаксический разбор осуществляется с применением более сложных грамматик, обеспечивающих иерархическое определение одних правил через другие. Поэтому, для построения распознавателей, мощности конечных автоматов, используемых при лексическом анализе, уже не хватает. Необходим более мощный и универсальный автомат, поддерживающий построение дерева разбора и выстраивающий его как сверху вниз, так и снизу вверх. Из предыдущих тем известно, что конечный автомат можно расширить дополнительной рабочей памятью, чтобы обеспечить распознавание цепочек, порождаемых практически любыми грамматиками. Организация и поведение такого автомата определяется классом грамматик. Как определено в классификации Хомского, контекстно-свободным грамматикам можно поставить в соответствие автомат с магазинной памятью (АМП).

То, что даже простые цепочки проблематично распознать с использованием конечного автомата, можно проиллюстрировать решением следующей элементарной задачи: необходимо построить распознаватель правильной вложенности круглых скобок. Такая вложенность скобок часто встречается в различных языках программирования при построении арифметических выражений. Для решения данной задачи необходимо:

1. определять равенство скобок, то есть количество открывающихся и закрывающихся скобок должно быть равным;
2. следить за правильностью их вложения, то есть, чтобы любой закрывающейся скобке предшествовала соответствующая открывающаяся.

Невозможность использования конечного автомата подтверждается и тем, что грамматику, порождающую рассматриваемые цепочки, нельзя свести к праволинейной (тому виду, который, по классификации Хомского, эквивалентен конечному автомату). Ее также нельзя представить только итеративными диаграммами Вирта, непосредственно соответствующими конечному автомату. Это определяется наличием в грамматике правил с рекурсией в середине. А такая рекурсия не может быть сведена к итерации. Грамматика  $G_{7.1}$  Может быть определена следующим образом:

$$G_{7.1} = (\{A, S\}, \{ (, ) \}, P, S),$$

Где P определяется как:

1.  $S \rightarrow (A) A$
2.  $A \rightarrow S$
3.  $A \rightarrow \epsilon$

Одним из путей решения задачи является добавление счетчика скобок, который, при просмотре цепочки увеличивается на 1, если встретится открывающаяся скобка. Закрывающаяся скобка уменьшает значения счетчика на единицу. Начальное состояние счетчика (перед просмотром цепочки) равно 0. По завершении просмотра цепочки значение счетчика должно быть равным 0. При этом, в ходе просмотра цепочки счетчик не может принимать отрицательные значения. Подход обеспечивает решение поставленной задачи. Однако, наличие счетчика уже определяет автомат с дополнительно рабочей памятью, которому также необходимо наличие арифметического устройства. Поэтому, использование стека вряд ли будет сложнее. А экономить ячейки памяти мы давно уже разучились. Кроме того, в реальной ситуации могут быть более сложные зависимости. Например, может быть несколько видов

скобок со своими правилами вложенности и их взаимным расположением. Значит, необходим универсальный подход, обеспечивающий преодоление различных ситуаций. Таким универсальным подходом и является использование автомата с магазинной памятью.

## Организация автомата с магазинной памятью

АМП в качестве рабочей памяти использует стек, называемый также магазином. Данная память поддерживает только ограниченные операции доступа, в то же время достаточные для решения сложных задач, включая и задачи распознавания цепочек. Автомат с магазинной памятью определяется следующими пятью объектами:

1. Конечным **множеством входных символов**, включающим концевой маркер ( $\dashv$ ).
2. Конечным **множеством магазинных символов**, включающим маркер дна ( $\nabla$ ).
3. Конечным множеством состояний, включающим начальное состояние.
4. **Устройством управления (УУ)**, которое каждой комбинации входного символа, магазинного символа и состояния ставит в соответствие выход или переход. Переход, в отличие от выхода, заключается в выполнении операций над магазином, состоянием и входом. Операции, запрашивающие входной символ после концевого маркера или выталкивания из магазина после маркера дна, а также операция вталкивания маркера дна, исключаются.
5. **Начальным содержимым магазина**, содержащим маркер дна и, возможно пустую, цепочку магазинных символов.

Автомат с магазинной памятью называется **распознавателем**, если у него 2 выхода: "**Допустить**" и "**Отвергнуть**".

### Операции автомата

Динамическое поведение АМП описывается через его операциями над входной цепочкой и стеком, а также переходами из одного состояния в другое. К операциям над стеком относятся:

1. "**Вытолкнуть**" - выталкивает из стека верхний символ (будем также использовать сокращенное обозначение " $\uparrow$ ").
2. "**Втолкнуть A**" - вталкивает в стек магазинный символ A (будем также использовать сокращенное обозначение " $\downarrow A$ ").
3. "**Заменить XYZ**" - используется для сокращения записи, когда необходимо вытолкнуть верхний символ и вместо него втолкнуть несколько других (в данном случае X, Y, Z). Эквивалентна:  
 $\uparrow \downarrow X \downarrow Y \downarrow Z$  (сокращенно обозначим:  $\uparrow XYZ$ ).

Переход АМП из одного состояния в другое указывается явно операцией "**Состояние t**", где **t** - новое состояние автомата (будем сокращать текст данной операции до "**[t]**").

Сдвиг входной головки на один символ вправо относительно входной ленты задается операцией "**Сдвиг**" (сократим до " $\rightarrow$ "). После ее выполнения текущим символом становится следующий символ на входной ленте. Другой операцией над входной головкой является "**Держать**", которая не изменяет положение входной головки до следующего шага (можно просто не писать, если нет сдвига).

**Переход или шаг автомата** - это выполнение операций над стеком и входной головкой, а также изменение состояния. При этом необязательно, чтобы за один шаг происходили все изменения. Возможно: или входная головка останется на месте, или не произойдет операции над стеком, или не изменится состояние.

## Распознаватель скобочных выражений

Рассмотрим одну из возможных реализация АМП, для задачи проверки корректности вложенности круглых скобок. Кратко опишем общий принцип работы автомата. Если входная головка читает "(", то в магазин заталкивается символ **A**. Если входная головка читает ")", то из магазина выталкивается содержащийся там символ. Цепочка отвергается, если:

1. На входе остаются правые скобки, а магазин пуст.
2. Входная цепочка прочитана до конца, а в магазине остаются символы **A**, соответствующие левым скобкам, для которых не нашлось пары.

Определим данный АМП следующим образом:

Множество входных символов: { (, ),  $\neg$  }.

Множество магазинных символов: { **A**,  $\nabla$  }

Множество состояний: **t**, где **t** является также и нач. состоянием автомата, раз оно единственное.

Переходы:

1. (, **A**, **S**  $\Rightarrow$   $\downarrow$  **A**, **S**,  $\rightarrow$
2. (,  $\nabla$ , **S**  $\Rightarrow$   $\downarrow$  **A**, **S**,  $\rightarrow$
3. ), **A**, **S**  $\Rightarrow$   $\uparrow$ , **S**,  $\rightarrow$
4. ),  $\nabla$ , **S**  $\Rightarrow$  Отвергнуть
5.  $\neg$ , **A**, **S**  $\Rightarrow$  Отвергнуть
6.  $\neg$ ,  $\nabla$ , **S**  $\Rightarrow$  Допустить

В начальном состоянии магазин содержит только маркер дна ( $\nabla$ ).

Из представленного описания видно, что поведение автомата имитирует ранее рассмотренный метод распознавания с использованием счетчика. Только вместо счетчика используются "палочки" (раньше с помощью палочек учили считать в школе). Эти палочки могут записываться в стек и стираться из него, отражая разность между прочитанными открывающимися и закрывающимися скобками. Работу данного АМП можно рассмотреть на примере распознавания нескольких цепочек. Пусть, первая цепочка будет иметь следующий вид:

(( ))

Тогда осуществляемые автоматом переходы можно представить в виде следующей последовательности текущих состояний его устройств.

Номер шага	Содержимое стека	Состояние автомата	Остаток входной цепочки	Номер применяемого правила
1	$\nabla$	[t]	(( )) $\neg$	2
2	$\nabla$ <b>A</b>	[t]	( ) $\neg$	1
3	$\nabla$ <b>A</b> <b>A</b>	[t]	) ( ) $\neg$	3
4	$\nabla$ <b>A</b>	[t]	( ) $\neg$	1
5	$\nabla$ <b>A</b> <b>A</b>	[t]	) ) $\neg$	3
6	$\nabla$ <b>A</b>	[t]	) $\neg$	3
7	$\nabla$	[t]	$\neg$	Допустить

В приведенном примере цепочка оказалась распознанной. Следующий пример раскрывает поведение автомата при распознавании цепочки, содержащей большее число правых круглых скобок чем левых:

(( ))

Таблица переходов и состояний в этом случае будет выглядеть следующим образом:

Номер шага	Содержимое стека	Состояние автомата	Остаток входной цепочки	Номер применяемого правила
1	▽	[t]	( ))) →	2
2	▽ A	[t]	))) →	3
3	▽	[t]	)) →	Отвергнуть

Аналогичные таблицы можно представить и для других вариантов входных цепочек.

Устройство управления данного автомата с магазинной памятью можно представить в виде следующей таблицы переходов.

Магазинные символы	Входные символы		
	(	)	→
A	↓A, →	↑, →	Отвергнуть
▽	↓A, →	Отвергнуть	Допустить

Такая таблица является типичной формой представления одного внутреннего состояния для любого АМП. Если у автомата имеется несколько внутренних состояний, то для каждого из них строится такая таблица переходов. В большинстве реальных случаев АМП имеют только одно состояние.

## Общая связь между грамматиками и автоматами с магазинной памятью

Эта связь давно доказана и рассмотрена в специальной литературе для различных классов грамматик и методов разбора. В рамках данной работы нет особой необходимости повторять материал или его выдержки из толстых книг. Можно только сослаться на основные источники информации, позволяющие получить фундаментальную подготовку по формальным грамматикам и методам разбора [Ахо78, Льюис]. Однако, и в этих книгах рассматриваются только такие грамматик и автоматы, которые могут быть эффективно реализованы. Из всего разнообразия грамматик, используемых для построения распознавателей, нас будут интересовать только КС(1) грамматик, ориентированные на нисходящий левый разбор, при котором входная цепочка просматривается слева направо. Использование данных грамматик позволяет создавать достаточно мощные и понятные языки программирования.

Следует пояснить, почему дальнейшее изучение ограничивается только нисходящими методами разбора. Методы, используемые при нисходящем разборе, достаточно универсальны и разнообразны. Применение восходящего разбора позволяет использовать более мощные КС(1) грамматик, в том числе и грамматик с левой рекурсией, которые при нисходящем разборе использовать невозможно. Возникают проблемы преобразования таких грамматик в грамматик с правой рекурсией, ориентированные на нисходящий разбор. Такое преобразование не всегда очевидно. Однако, применение диаграмм Вирта для представления синтаксиса языков программирования позволяет легко заменить все левые рекурсии на итерации и использовать полученные правила для нисходящего разбора. Поэтому, на мой взгляд, при практической разработке трансляторов, не имеет особого смысла использовать восходящий разбор. Забегая вперед, отмечу, что не вижу особого смысла использовать и автоматы с магазинной памятью, так как можно использовать другую модель распознавателя, более близкую по смыслу к диаграммам Вирта. Эта иерархически порождаемые конечные автоматы (пока я считаю, что для построения трансляторов она придумана и введена мною).

Для того, чтобы рассмотреть зависимость, существующую между КС(1) грамматиками и нисходящими распознавателями с магазинной памятью, обратимся к материалу, изложенному в книге Льюиса, Розенкранца и Стринза [Льюис]. Нет смысла выдумывать то, что уже изложено в виде, доступном для понимания.

## Связь между S-грамматикой и автоматом с магазинной памятью

Не все КС-грамматики пригодны для построения нисходящего детерминированного АМП, так как многие из них могут порождать одну и ту же терминальную цепочку различными левыми выводами. Это говорить об их неоднозначности и невозможности использования для детерминированного разбора. Однако, определены и изучены такие классы грамматик, которые поддерживают нисходящий детерминированный разбор. Наиболее простыми из них являются S-грамматики.

КС-грамматика называется **S-грамматикой** (а также раздельной, или простой) тогда и только тогда, когда выполняются два условия:

1. Правая часть каждого правила начинается терминалом  $T_i$ .
2. Если два правила имеют совпадающие левые части, то правые части этих правил должны начинаться различными терминальными символами.

### Пример

Рассмотрим две грамматики, которые задают один и тот же язык.

Грамматика  $G_{7.2}(S)$ , Содержит правила P:

1.  $S \rightarrow aT$
2.  $S \rightarrow TbS$
3.  $T \rightarrow bT$
4.  $T \rightarrow ba$

Это не S-грамматика, так как правило 2 начинается с нетерминала, что нарушает условие 1, а правила 3, 4 имеют совпадающие левые части и начинаются с одинаковых терминальных символов.

S-грамматика  $G_{7.3}(S)$  для этого же языка может использовать для порождения цепочек следующие правила:

1.  $S \rightarrow abR$
2.  $S \rightarrow bRbS$
3.  $R \rightarrow a$
4.  $R \rightarrow bR$

Использование данной грамматики позволяет упростить разбор, используя для этого соответствующий АМП.

### Обобщенный алгоритм построения нисходящего АМП для S - грамматики

Построение нисходящего автомата с магазинной памятью для заданной S-грамматики осуществляется по следующему обобщенному алгоритму.

1. Множество входных символов автомата - это множество терминальных символов грамматики, расширенное концевым маркером.
2. Множество магазинных символов состоит из маркера дна, нетерминальных символов грамматики и терминалов, встречающихся в правых частях правил, за исключением тех, которые занимают только крайнюю левую позицию.
3. В начальном состоянии магазин содержит маркера дна и начальный нетерминал.
4. Устройство управления АМП, имеет одно состояние и описывается управляющей таблицей, строки которой помечены магазинными символами, столбцы - входными символами, а элементы таблицы описываются в ниже следующих пунктах.
5. Каждому правилу грамматики ставится в соответствие элемент (клетка) таблицы. Правило

должно иметь вид:

$$A \rightarrow b\alpha,$$

где  $A$  - нетерминал,  $b$  - терминал,  $\alpha$ - цепочка из терминалов и нетерминалов. Этому правилу будет соответствовать элемент в строке  $A$  и столбце  $b$ :

### Заменить ( $\alpha^r$ ), Сдвиг

где  $\alpha^r$  - цепочка  $\alpha$ , записанная в обратном порядке (для того, чтобы удовлетворить соглашению, что верхний символ находится справа при принятом способе записи стека, когда его дно располагается слева). Если правило имеет вид

$$A \rightarrow b,$$

то вместо

### Заменить $\epsilon$ , Сдвиг

используется

### Вытолкнуть, Сдвиг.

6. Если магазинным символом является терминал  $b$ , то элементом таблицы в строке  $b$  и столбце  $b$  будет

### Вытолкнуть, Сдвиг

7. Элементом таблицы, который находится в строке маркера дна и столбце концевого маркера, является

### Допустить

8. Элементы таблицы, не описанные ни в одном из пунктов 5, 6, 7 заполняются операцией

### Отвергнуть

Используя данное описание, построим автомат для S- грамматики  $G_{8.3}(S)$ :

Магазинные символы	Входные символы		
	a	b	$\neg$
S	$\uparrow Rb, \rightarrow$	$\uparrow SbR, \rightarrow \equiv \equiv \downarrow bR, \rightarrow$	Отвергнуть
R	$\uparrow, \rightarrow$	$\uparrow R, \rightarrow$	Отвергнуть
b	Отвергнуть	$\uparrow, \rightarrow$	Отвергнуть
$\nabla$	Отвергнуть	Отвергнуть	Допустить

Каким образом найдена представленная последовательность шагов построения АМП? Что-то получили с использованием дедукций, что-то - индукции. Остальное высосали из пальца. Если же присмотреться внимательнее к уже предложенным шагам, то можно понять, как они появились. В основе всего лежат особенности S-грамматики. Рассмотрим левый вывод терминальной цепочки **bbababa**. Он будет осуществляться следующим образом:

$$S \Rightarrow bRbS \Rightarrow bbRbS \Rightarrow bbababR \Rightarrow bbababa$$

Процедура построения нисходящего дерева разбора, представленная на рис. 7.1, показывает, что на каждом шаге правило для дерева выбирается однозначно и увеличивает количество распознанных символов на один.



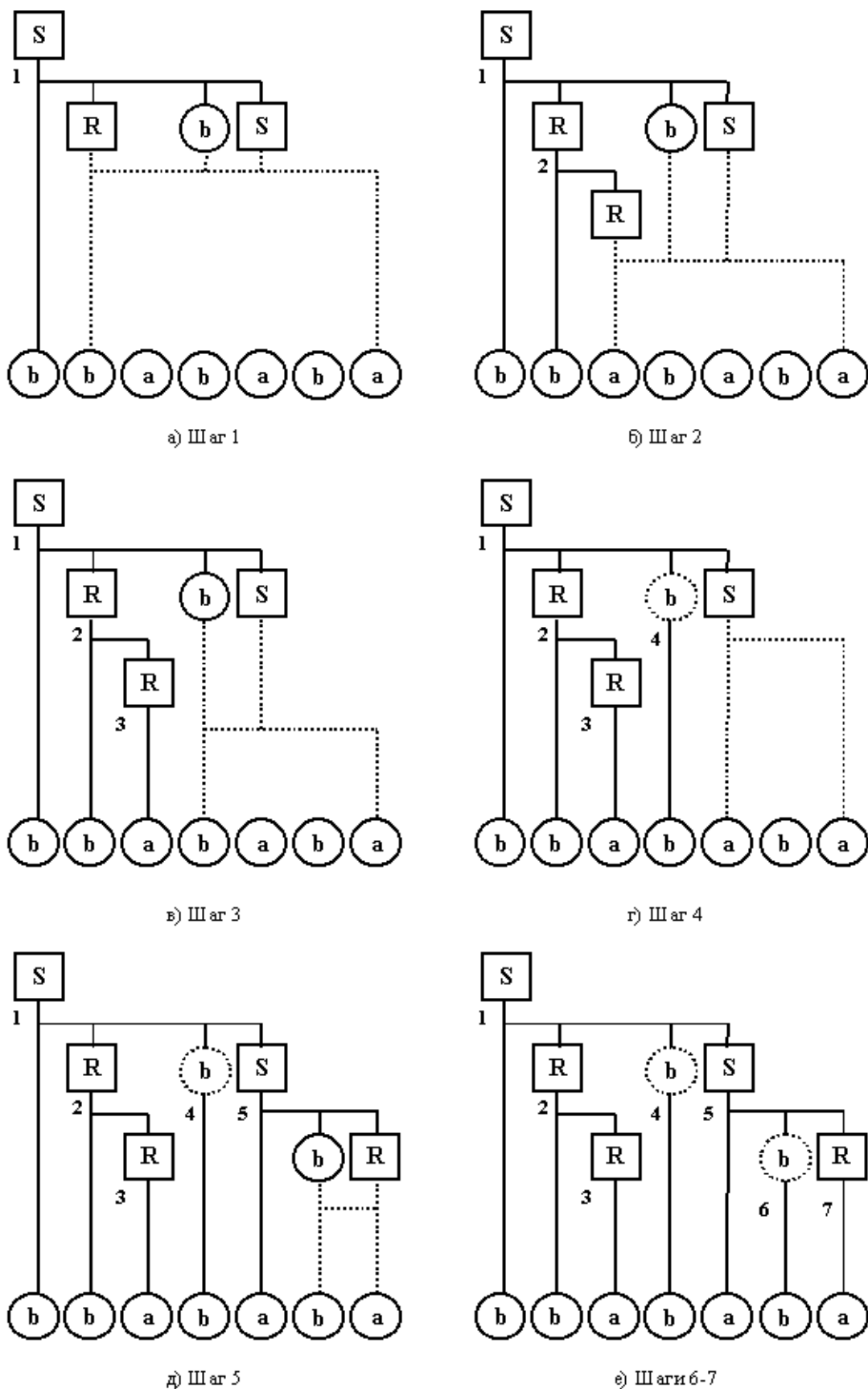


Рис. 7.1. Нисходящий разбор слева направо с использованием S-грамматики.

Этот выбор определяется грамматикой, которая четко устанавливает соответствие между текущим нетерминалом, входным символом, а также правилом, определяющим текущий нетерминал. Правило задано таким образом, что на первом месте в его правой части стоит символ, совпадающий с текущим символом цепочки. Поэтому, не нужны возвраты назад. Если соответствующего правила не будет найдено, произойдет отказ. При нахождении искомого правила, его первый символ нейтрализуется входным символом и отбрасывается (и потому не заносится в стек). Остальные же символы необходимо сохранить, чтобы использовать для формирования новых проверок на последующих шагах. Понятно также, что сохранение должно осуществляться в стеке (нет другой рабочей памяти) и в реверсивном

порядке (так как самый левый символ правила должен разбираться первым, чтобы было соответствие с левым выводом). Отсюда также понятно, что терминалы, расположенные в середине правила, будут заноситься в стек и поэтому должны являться магазинными символами. Ясно также, что когда на вершине стека располагается терминал, он не может являться левой частью правила. Поэтому его необходимо просто вытолкнуть из стека, нейтрализуя эквивалентным входным символом.

## S-грамматика и распознавание вложенности скобок

Обладая методами создания АМП по S-грамматике, вернемся к задаче распознавания вложенности скобок. Раньше соответствующий автомат был построен эвристически. Между тем, грамматика  $G_{7.1}$  не является S-грамматикой. И, к своему удивлению, мы не можем ее преобразовать к S-грамматике даже для этой простой задачи. Попытаемся разобраться с возникшими проблемами.

Нет ничего сложного в построении правила S-грамматики для начального нетерминала, правая часть которого всегда начинается с открывающейся скобки:

$$S \rightarrow (B .$$

Но, при определении  $B$ , возникает ряд проблем. Правая часть  $B$  может начинаться с закрывающейся скобки, за которой не будет следовать ничего:

$$B \rightarrow ) .$$

Но, вместе с тем, вслед за закрывающейся скобкой может вновь следовать произвольный набор вложенных скобок, полностью повторяющий цепочку, порождаемую из начального нетерминала:

$$B \rightarrow )S .$$

Возникает противоречие условию 2, которое в S-грамматике разрешить невозможно. Это противоречие не позволяет осуществить порождение произвольных цепочек. Остается только генерировать разнообразные варианты внутри охватывающих круглых скобок. Для этого правило  $B$ , начинающееся с открывающейся круглой скобки, должно выглядеть следующим образом:

$$B \rightarrow (BB .$$

Данный анализ показывает, что вся проблема возникает из-за того, что предполагаемая цепочка может повториться множество раз или не появиться вообще (обобщая, можно сказать, что цепочка может повториться ноль или большее число раз). А возможность одновременного описания необязательных или повторяющихся цепочек как раз и отсутствует в S-грамматике. Для этих случаев необходимо иметь более мощный механизм порождения цепочек, допускающий пустые цепочки, например, q-грамматику.

## Построение автомата с магазинной памятью по q-грамматике

Контекстно-свободная грамматика называется **q-грамматикой** тогда и только тогда, когда выполняются два условия:

1. Правая часть каждого правила либо представляет собой пустую цепочку  $\epsilon$ , либо начинается с терминального символа.
2. Множество выбора правил с одной и той же левой частью не пересекаются.

Второе условие следует рассмотреть подробнее, так как необходимо ввести ряд определений. Одно из них - множество выбора. Для демонстрации определений будем использовать пример q-грамматики  $G_{7.4}(S)$ :

1.  $S \rightarrow aAS$
2.  $S \rightarrow b$
3.  $A \rightarrow cAS$
4.  $A \rightarrow \epsilon$

О том, что это не S - грамматика говорит правая часть правила 4, которая не начинается с терминального символа. Однако, метод построения НАМП, изложенный для S-грамматики почти подходит и здесь.

**Множество терминалов, следующих за данным нетерминалом  $X$**  (Обозначим через  $СЛЕД(X)$ ) - множество терминальных символов, которые могут следовать непосредственно за  $X$  в какой-либо промежуточной цепочке, выводимой из  $S$ , включая и  $\rightarrow$ .

Это множество символов называется также **множеством следуемых за  $X$  терминалов**. Для приведенной грамматики можно определить следующие множества:

$$СЛЕД(A) = \{a, b\}$$

$$СЛЕД(S) = \{a, b\}$$

Терминалы, следующие за  $A$ , определяются исходя из того, что за  $A$  следует всегда нетерминал  $S$ , правая часть правила для которого начинается либо с  $a$ , либо с  $b$ . Терминалы, следующие за  $S$ , выводятся путем подстановки в правило 1 правила 3, которая показывает, что за  $S$  всегда следует  $S$ , начинающаяся либо с  $a$ , либо с  $b$ :

$$S \Rightarrow aAS \Rightarrow acASS.$$

Данное понятие используется для того, чтобы показать, когда следует применять правило с пустой правой частью  $A \rightarrow \epsilon$ . Если входной символ  $T_i \notin \{a, b\}$ , то есть, является символом  $c$  или  $\rightarrow$ , а на вершине магазина  $A$ , то бесполезно применять  $A \rightarrow \epsilon$ . Если это правило будет применено, то  $A$  удаляется из стека, а так как за  $A$  символы  $c$  или  $\rightarrow$  следовать не могут, то в результате процесс останавливается. Представим, что у нас уже есть автомат, соответствующий грамматике  $G_{7.4}$ , и очень похожий на АМП, распознающий  $S$ -грамматику. Тогда, применительно к цепочке **aacbb**, можно получить несколько решений. В начале рассмотрим стратегию, при которой в первую очередь применяется всегда пустое правило.

Номер шага	Содержимое стека	Остаток входной цепочки	Номер применяемого правила	Комментарии
1	$\nabla S$	aacbb $\rightarrow$	1	
2	$\nabla SA$	acbb $\rightarrow$	4	Иначе, процесс разбора дальше не пойдет
3	$\nabla S$	acbb $\rightarrow$	1	
4	$\nabla SA$	cbb $\rightarrow$	4	$A$ здесь данный шаг ведет к отказу по отношению к правильной цепочке
5	$\nabla S$	cbb $\rightarrow$	Отвергнуть	Придется осуществлять возврат

Правильным было бы применение на шаге 4 правила 3, в соответствии с ранее рассмотренными рекомендациями о месте и времени применения пустого правила.

Номер шага	Содержимое стека	Остаток входной цепочки	Номер применяемого правила	Комментарии
1	$\nabla S$	aacbb $\rightarrow$	1	
2	$\nabla SA$	acbb $\rightarrow$	4	Иначе, процесс разбора дальше не пойдет
3	$\nabla S$	acbb $\rightarrow$	1	

4	▽ SA	cbb→	3	Пойдем в соответствии с предложенными рекомендациями
5	▽ SSA	bb→	4	Иначе, процесс разбора дальше не пойдет
6	▽ SS	bb→	2	
7	▽ S	b→	2	
8	▽	→	Допустить	Цепочка принадлежит языку!

Таким способом решается одна из задач детерминированного разбора, связанная с особенностями применения пустого правила. Однако, при разборе, построенном на основе q-грамматики возможна и такая ситуация, когда:

$$A \rightarrow b\alpha, A \rightarrow \epsilon \text{ и } b \in \text{СЛЕД}(A).$$

В этом случае стоит проблема выбора одного из правил с одинаковыми левыми частями. Для ее решения введем понятие **множества выбора** для данного правила.

Если правило грамматики имеет вид:  $A \rightarrow b\alpha$ , где  $b$  - терминал, а  $\alpha$  - цепочка терминалов и нетерминалов, то определим множество выбора для правила  $A \rightarrow b\alpha$  равным  $b$ . Обозначим данную запись следующим образом:

$$\text{ВЫБОР}(A \rightarrow b\alpha) = b.$$

Если правило имеет вид:  $A \rightarrow \epsilon$ , то определим:

$$\text{ВЫБОР}(A \rightarrow \epsilon) = \text{СЛЕД}(A).$$

Если  $p$  - номер правила, то будем также писать "**ВЫБОР** ( $p$ )" вместо "**ВЫБОР** (правило)"

**ВЫБОР** ( $p$ ) - множество выбора правила  $p$ .

Проиллюстрируем построение множества выбора на примере грамматики  $G_{7.4}$ .

$$\text{ВЫБОР}(1) = \text{ВЫБОР}(S \rightarrow aAS) = \{a\}$$

$$\text{ВЫБОР}(2) = \text{ВЫБОР}(S \rightarrow b) = \{b\}$$

$$\text{ВЫБОР}(3) = \text{ВЫБОР}(A \rightarrow cAS) = \{c\}$$

$$\text{ВЫБОР}(4) = \text{ВЫБОР}(A \rightarrow \epsilon) = \text{СЛЕД}(A) = \{a, b\}$$

В соответствии с определением, рассмотренным в начале, имеем q-грамматику, так как:

$$\text{ВЫБОР}(1) \cup \text{ВЫБОР}(2) = \emptyset,$$

$$\text{ВЫБОР}(3) \cup \text{ВЫБОР}(4) = \emptyset.$$

## Построение нисходящего автомата

Рассмотренные понятия и методы их использования приводят к небольшой корректировке правил порождения АМП по S-грамматике, чтобы получить процедуру создания автомата по q-грамматике. Эта корректировка касается только пункта 5. Можно, конечно, полностью переписать исправленный алгоритм создания автомата, тем более, что использование современных редакторов позволяет осуществить эту процедуру без проблем. К тому же компания Microsoft приучила нас не экономить ресурсы. Но я не буду оригинальным и воспользуюсь авторским текстом [Льюис]. Пятое правило заменяется двумя следующими:

5.1. Правило грамматики применяется всякий раз, когда магазинный символ является его левой частью, а входной символ принадлежит его множеству выбора. Для того, чтобы применить правило вида:

$$A \rightarrow b\alpha,$$

используется переход:

Заменить ( $\alpha^r$ ), Сдвиг

Если правило имеет вид:

$$A \rightarrow b,$$

то используется:

Вытолкнуть, Сдвиг.

Для того, чтобы применить правило:

$$A \rightarrow \epsilon$$

используется переход

Вытолкнуть.

5.2. Если имеется правило с нетерминалом **A** в левой части и элемент, соответствующий магазинному **A** и входному символу **b** не был создан по правилу 5.1., то таким элементом м.б. либо применение пустого правила ( $A \rightarrow \alpha$ ) или операция "Отвергнуть".

### Примеры построения АМП по q-грамматике

Построим автомат с магазинной памятью по q-грамматике, используемой выше для различных иллюстраций. Его таблица переходов будет иметь следующий вид.

Магазинные символы	Входные символы			
	a	b	c	¬
S	↕SA, →	↑, →	Отвергнуть	Отвергнуть
A	↑	↑	↕SA, →	Отвергнуть
▽	Отвергнуть	Отвергнуть	Отвергнуть	Допустить

Следующий пример иллюстрирует использование правила 5.2, когда описанное в нем ячейка таблицы переходов может быть реализована любым из двух вариантов. Я бы предпочел второй, чтобы меньше мучиться. Грамматика  $G_{7.5}(S)$  описывается следующими правилами:

- $S \rightarrow aA$
- $S \rightarrow b$
- $A \rightarrow cSa$
- $A \rightarrow \epsilon$

Множества выбора для каждого из правил будут выглядеть следующим образом:

- ВЫБОР(1) = ВЫБОР ( $S \rightarrow aA$ ) = {a}**  
**ВЫБОР(2) = ВЫБОР ( $S \rightarrow b$ ) = {b}**  
**ВЫБОР(3) = ВЫБОР ( $A \rightarrow cSa$ ) = {c}**  
**ВЫБОР(4) = ВЫБОР ( $A \rightarrow \epsilon$ ) = СЛЕД (A) = {a, ¬}**

Полученные данные использованы для построения таблицы переходов. Ячейка, содержащая альтернативные правила, находится на пересечении строки с магазинным символом **A** и столбце с входным символом **b**.

Магазинные символы	Входные символы			
	a	b	c	¬
S	↕A, →	↑, →	Отвергнуть	Отвергнуть
A	↑	1)↑ или 2)Отвергнуть	↕aS, →	↑
a	↑, →	Отвергнуть	Отвергнуть	Отвергнуть
▽	Отвергнуть	Отвергнуть	Отвергнуть	Допустить

Для того, чтобы проверить, как действуют эти правила, необходимо ввести неправильную цепочку. Возьмем для примера цепочку **ab**. В первом случае используем выталкивание, которое ведет к отвержению цепочки на третьем шаге.

Номер шага	Содержимое	Остаток	Применяемый	Комментарий
------------	------------	---------	-------------	-------------

	стека	входной цепочки	переход	
1	$\nabla S$	$ab \neg$	$\updownarrow A, \rightarrow$	
2	$\nabla A$	$b \neg$	$\uparrow$	Использовано выталкивание
3	$\nabla$	$b \neg$	Отвергнуть	Все равно отвергли

Во втором варианте цепочка отвергается уже на втором шаге.

Номер шага	Содержимое стека	Остаток входной цепочки	Применяемый переход	Комментарий
1	$\nabla S$	$ab \neg$	$\updownarrow A, \rightarrow$	
2	$\nabla A$	$b \neg$	Отвергнуть	На шаг раньше

### Распознавание вложенности скобок и q-грамматика

Теперь у нас достаточно знаний для построения простой грамматики, порождающей вложенные скобки. Данная q-грамматика содержит следующие правила:

1.  $S \rightarrow (B)B$
2.  $B \rightarrow (B)B$
3.  $B \rightarrow \epsilon$

Правила выбора выглядят следующим образом:

**ВЫБОР(1) = ВЫБОР ( $S \rightarrow (B)B$ ) =  $\{\}$**

**ВЫБОР(2) = ВЫБОР ( $B \rightarrow (B)B$ ) =  $\{\}$**

**ВЫБОР(3) = ВЫБОР ( $B \rightarrow \epsilon$ ) = СЛЕД ( $B$ ) =  $\{\}, \neg$**

Они лишний раз подтверждают, что мы построили q-грамматику. Следует специально отметить, что, начиная с q-грамматики, далее постоянно приходится доказывать принадлежность грамматики заданному классу, чтобы затем использовать необходимые процедуры построения автомата. Таблица переходов данного автомата будет выглядеть следующим образом:

Магазинные символы	Входные символы		
	(	)	$\neg$
S	$\updownarrow B)B, \rightarrow$	Отвергнуть	Отвергнуть
B	$\updownarrow B)B, \rightarrow$	$\uparrow$	$\uparrow$
)	Отвергнуть	$\uparrow, \rightarrow$	Отвергнуть
$\nabla$	Отвергнуть	Отвергнуть	Допустить

## LL(1) - грамматики

Понятие множества выбора можно обобщить таким образом, чтобы применять его к КС-грамматикам произвольного вида.

**LL(1)-грамматика** - это такая КС(1)-грамматика, в которой множества выбора правил с одинаковыми левыми частями не пересекаются.

Обобщение понятия множества выбора позволяет использовать правила, начинающиеся с нетерминалов. В качестве примера можно представить LL(1)-грамматику, содержащую следующие правила:

1.  $S \rightarrow AbB$
2.  $S \rightarrow d$
3.  $A \rightarrow Cab$
4.  $A \rightarrow B$
5.  $B \rightarrow cSd$
6.  $B \rightarrow \epsilon$
7.  $C \rightarrow a$
8.  $C \rightarrow ed$

То, что это LL(1)-грамматика, еще надо доказать, так как в противном случае результат разбора может оказаться непредсказуемым. Да и автомат с магазинной памятью мы построить не сможем из-за того, что не имеем дополнительной информации, требуемой для построения таблицы переходов. А она появляется в ходе доказательств.

В книге Льюиса и пр. [Льюис] приводится описание доказательства принадлежности заданной грамматики к LL(1)-грамматике. Оно занимает **одиннадцать** страниц, использует демонстрационные примеры, разреженные матрицы и т.д. Я неоднократно принимался за его чтение, но каждый раз с грустью убеждался, что мой **IQ** (надо как-нибудь, хоть разок, проверить его значение) является недостаточно высоким, чтобы осилить материал. Поэтому были попытки перескочить эти страницы, выбросить нисходящий разбор из головы и обратиться к другой религии, проповедующей восходящий разбор. Но и здесь мне не хватило веры в истинность избранного пути. Поэтому, как это часто бывает в жизни, я нашел одну маленькую религию, которую подогнал под свои воззрения. После этого я смог не только успокоить свою душу, но и нести новый свет в массы. Эта религия зиждется на диаграммах Вирта, источая дальше свои идеи на распознаватели и реализацию. Подробно она будет раскрыта в следующей теме. Ее применение позволило обойти мне непереваримое доказательство и остаться в лоне нисходящего разбора.

## Программная реализация нисходящего автомата с магазинной памятью

Разработку программы, реализующей логику работы нисходящего распознавателя, можно осуществить **на основе** построенной **таблицы переходов** нисходящего автомата с магазинной памятью. Кроме того, учитывая эквивалентность построенной таблицы, и исходной грамматики, а также используя особенности организации современных языков программирования, можно осуществить реализацию **на основе имеющихся правил**. Главное в этом случае, чтобы грамматика соответствовала одному из рассмотренных классов (изучайте пропущенное доказательство). Метод разработки программы на основе синтаксических правил называется **рекурсивным спуском**.

## Разработка программы по таблице переходов АМП

В этом случае нам необходимо построить наборы магазинных и входных символов, смоделировать основные устройства автомата и написать процедуру, реализующую таблицу переходов. Основными модулями являются стек и модуль чтения входной цепочки символов.

Реализация стека должна, по крайней мере поддерживать три основных функции: вталкивание в стек (**Push**), выталкивание из стека (**Pop**) и чтение вершины стека (**Top**). Стек должен контролировать возможное переполнение и попытку выталкивания маркера дна (маркер явно может отсутствовать, поэтому необходимо контролировать попытку выталкивания из пустого стека). Модуль чтения входной цепочки содержит набор функций, обеспечивающих доступ к требуемому входному потоку, а также перемещение по нему слева направо.

Таблица переходов может быть реализована с использованием переключателей (например, операторов `switch` в языках C, C++) или путем интерпретации данных, располагаемых в соответствующей матрице. Второй способ позволяет строить различные распознаватели, не меняя программу, а первый работает намного быстрее.

Чтобы не быть голословными, схематически рассмотрим один из вариантов возможной реализации на примере, распознавателя круглых скобок. При этом, учитывая учебный характер примера, будем придерживаться простейших структур данных и постараемся обойтись без лишних проверок. Принятые при реализации решения кратко описаны в виде комментариев к разрабатываемой программе, исходный текст которой приводится ниже. Программа реализует таблицу переходов с применением переключателей. Для упрощения разработки строка скобок читается с клавиатуры. Исходные тексты программы можно скачать из [архива](#) (вместе с проектом для MS VC++ 6.0).

```
// BrackStack.cpp - автомат с магазинной памятью,  
// распознающий вложенность круглых скобок.  
// Построен на основе таблицы переходов,  
// созданной по следующей q-грамматике:  
// 1. S -> (B)V  
// 2. B -> (B)V  
// 3. B -> empty  
#include <iostream>  
#include <string>  
#include <vector>  
#include <stack>  
using namespace std;  
// Магазинными символами автомата являются:  
// S, B, маркер дна (bottom).  
// Их значение должно быть за пределами видимых символов.  
enum stackSymb {S = 256, B, bottom};  
// Строка с входной цепочкой, имитирующая входную ленту  
string str;  
int i; // Текущее положение входной головки  
stack <int, vector<int> > stck; // Стек для магазинных символов  
// Функция, реализующая чтение символов в буфер из входного потока.  
// Используется для ввода с клавиатуры распознаваемой строки.
```



// Ввод осуществляется до нажатия на клавишу Enter.

// Символ '\n' является конечным маркером входной строки.

```
void GetOneLine(istream &is, string &str) {  
    char ch;  
    str = "";  
    for(;;) {  
        is.get(ch);  
        if(is.fail() || ch == '\n') break;  
        str += ch;  
    }  
    str += '\n'; // Добавляется конечной маркер  
}
```

// Инициализация устройств АМП

```
void Init() {  
    // Инициализация стека  
    while(!stck.empty()) stck.pop();  
    stck.push(bottom);  
    stck.push(S);  
    // Инициализация входной головки  
    i = 0;  
}
```

// Устройство управления АМП, анализирующего вложенность скобок.

// Имитирует таблицу переходов АМП.

```
bool Parser() {  
    // Инициализация устройств АМП  
    Init();  
    int step = 0;  
    // Цикл анализа состояний  
    for(;;) {  
        // Тестовый вывод информации о текущем шаге,  
        // текущем символе, вершине стека  
        cout << "step " << step++ << ": str[" << i << "] = " << str[i]  
            << " Top = " << stck.top() << "\n";  
        // Проверка содержимого на вершине стека  
        switch(stck.top()) {  
            // Анализ первой строки таблицы переходов  
            case S:
```

```

switch(str[i]) {
case '(': // [S, (]
    stck.top() = B; // эквивалентно stck.pop(); stck.push(B);
    stck.push('(');
    stck.push(B);
    i++; // следующий входной символ
    break;
case ')': // [S, )] - ) не может стоять в начале
    cout << "Position " << i << ", "
        << "Error 1: '\)' can not is in begin!\n";
    return false;
case '\n': // [S, концевой маркер]
    cout << "Position " << i << ", "
        << "Error 2: Empty string!\n";
    return false;
default: // Ошибка, такой символ не допустим
    cout << "Position " << i << ", "
        << "Incorrect symbol! " << str[i] << "\n";
    return false;
}
break;

```

// Анализ второй строки таблицы переходов

case B:

```

switch(str[i]) {
case '(': // [B, (]
    stck.top() = B; // эквивалентно stck.pop(); stck.push(B);
    stck.push('(');
    stck.push(B);
    i++; // следующий входной символ
    break;
case ')': // [B, )] - вытолкнуть без сдвига
    stck.pop();
    break;
case '\n': // [B, концевой маркер] - вытолкнуть без сдвига
    stck.pop();
    break;
default: // Ошибка, такой символ не допустим

```

```

    cout << "Position " << i << ", "
        << "Incorrect symbol! " << str[i] << "\n";
    return false;
}
break;
// Анализ третьей строки таблицы переходов
case ')':
    switch(str[i]) {
        case '(': // [], () - в принципе недостижимо
            cout << "Position " << i << ", "
                << "Error 3: I want '\')'\n";
            return false;
        case ')': // [], ]) - вытолкнуть со сдвигом
            stck.pop(); // вытолкнуть
            i++; // сдвиг
            break;
        case '\n': // [], концевой маркер]
            cout << "Position " << i
                << ", Error 4: I want '\')'\n";
            return false;
        default: // Ошибка, такой символ не допустим
            cout << "Position " << i << ", "
                << "Incorrect symbol! " << str[i] << "\n";
            return false;
    }
    break;
// Анализ четвертой строки таблицы переходов
case bottom:
    switch(str[i]) {
        case '(': // [bottom, () - невозможно при пустом магазине
            cout << "Position " << i << ", "
                << "Error 4: Impossible situation [bottom, '\()'\n";
            return false;
        case ')': // [bottom, ]) - лишняя )
            cout << "Position " << i << ", "
                << "Error 5: unnecessary '\()'\n";
            return false;
    }

```

```

    case '\n': // [bottom, концевой маркер] - допустить!
        return true;
    default: // Ошибка, такой символ не допустим
        cout << "Position " << i << ", "
            << "Incorrect symbol! " << str[i] << "\n";
        return false;
    }
    break;
}
}
}

// Главная функция используется для инициализации устройств
// АМП перед каждым новым прогоном и тестирования до тех пор,
// пока не будет прочитана пустая строка.
int main () {
    string strCursor;
    str = "";
    // Цикл распознавания различных входных цепочек
    do {
        // Чтение очередной входной цепочки в буфер
        cout << "Input bracket's expression!: ";
        // Формируем очередную строку скобок для распознавания.
        GetOneLine(cin, str);
        // Здесь начинается разбор принятой строки.
        if(Parser())
            cout << "+++++ OK! +++++\n";
        else
            cout << "--- Fatal error (look upper error message)! ---\n";
        // Вывод разобранной строки и значения позиции входной головки.
        cout << "Line: " << str;
        strCursor = " Pos: " + string(i, '-');
        strCursor += '^';
        cout << strCursor << " i = " << i << "\n\n";
    } while(str != "\n");
    cout << "Goodbye!\n";
    return 1;
}

```

## Разработка программы с использованием метода рекурсивного спуска

Метод рекурсивного спуска учитывает особенности современных языков программирования, в которых реализован рекурсивный вызов процедур. Если обратиться к дереву нисходящего разбора слева направо (рис. 7.1), то можно заметить, что в начале происходит анализ всех поддеревьев, принадлежащих самому левому нетерминалу. Затем, когда самым левым становится другой нетерминал, то анализ происходит для него. При этом используются и полностью раскрываются все нижележащие правила. Это навязывает определенные ассоциации с иерархическим вызовом процедур в программе, следующих друг за другом в охватывающей их процедуре. Поэтому, все нетерминалы можно заменить соответствующими им процедурами или функциями, внутри которых вызовы других процедур - нетерминалов и проверки терминальных символов будут происходить в последовательности, соответствующей их расположению в правилах. Такая возможность подкрепляется и другой ассоциацией. Вызов процедуры или функции реализуется через занесение локальных данных в стек, который поддерживается системными средствами. Заносимые данные определяют состояние обрабатываемого нетерминала, а машинный стек соответствует магазину автомата.

Использование рекурсивного спуска позволяет достаточно быстро и наглядно писать программу распознавателя на основе имеющейся грамматики. Главное, чтобы последняя соответствовала требуемому виду. Исходный текст программы распознавания вложенности скобок, использующей метод рекурсивного спуска, приведен ниже. Несмотря на наличие совпадающих функций, программа приведена полностью, чтобы дать отчетливое представление об ее структуре. А этот пример расположен в другом [архиве](#).

```
// Рекурсивный распознаватель вложенности круглых скобок.
// Построен на основе правил следующей q-грамматики:
// 1. S -> (B)B
// 2. B -> (B)B
// 3. B -> empty
#include <iostream>
#include <string>
#include <vector>
#include <stack>
using namespace std;
// Строка с входной цепочкой, имитирующая входную ленту
string str;
int i; // Текущее положение входной головки
int erFlag; // Флаг, фиксирующий наличие ошибок в середине правила
// Функция, реализующая чтение символов в буфер из входного потока.
// Используется для ввода с клавиатуры распознаваемой строки.
// Ввод осуществляется до нажатия на клавишу Enter.
// Символ '\n' является концевым маркером входной строки.
void GetOneLine(istream &is, string &str) {
    char ch;
    str = "";
    for(;;) {
        is.get(ch);
        if(is.fail() || ch == '\n') break;
        str += ch;
```

```

}
str += '\n'; // Добавляется концевой маркер
}
// Функция, реализующая распознавание нетерминала B.
bool B() {
    if(str[i] == '(') {
        i++;
        if(B()) {
            if(str[i] == ')') {
                i++;
                if(B()) { // Если последний нетерминал корректен,
                    // то корректно и все правило.
                    return true;
                } else { // Ошибка в нетерминале B
                    erFlag++;
                    cout << "Position " << i << ", "
                        << "Error 1: Incorrect last B!\n";
                    return false;
                }
            } else { // Это не ')'
                erFlag++;
                cout << "Position " << i << ", "
                    << "Error 2: I want '\')'\n";
                return false;
            }
        } else {
            erFlag++;
            cout << "Position " << i << ", "
                << "Error 3: Incorrect first B!\n";
            return false;
        }
    } else // Смотрим другую альтернативу:
        return true; // Пустая цепочка допустима
}
// Функция, реализующая распознавание нетерминала S.
bool S() {
    if(str[i] == '(') {
        i++;
        if(B()) {
            if(str[i] == ')') {
                i++;

```

```

if(B()) {
    if(str[i] == '\n') {
        return true; // за последним нетерминалом - конец
    } else { // Там ничего не должно быть
        erFlag++;
        cout << "Position " << i << ", "
            << "Error 4: I want string end after last right bracket!\n";
        return false;
    }
} else {
    erFlag++;
    cout << "Position " << i << ", "
        << "Error 4: Incorrect last B!\n";
    return false;
}
} else {
    erFlag++;
    cout << "Position " << i << ", "
        << "Error 5: I want '\')'\n";
    return false;
}
} else {
    erFlag++;
    cout << "Position " << i << ", "
        << "Error 6: Incorrect first B!\n";
    return false;
}
}
return false; // Первый символ цепочки некорректен
// то, что это ошибка, лучше определить снаружи
}
// Функция запускающая разбор и определяющая корректность его завершения,
// если первый символ не принадлежит цепочки
bool Parser() {
    // Начальная инициализация.
    erFlag = 0;
    i = 0;
    // Процесс пошел!
    if(S())
        return true; // Все прошло нормально
    else {

```

```

if(erFlag)
    cout << "Position " << i << ", "
        << "Error 7: Internal Error!\n";
else
    cout << "Position " << i << ", "
        << "Error 8: Incorrect first symbol of S!\n";
return false; // Есть ошибки
}
}
// Главная функция используется для тестирования до тех пор,
// пока не будет прочитана пустая строка
int main () {
    string strCursor;
    str = "";
    // Цикл распознавания различных входных цепочек
    do {
        // Чтение очередной входной цепочки в буфер
        cout << "Input bracket's expression!: ";
        // Формируем очередную строку скобок для распознавания.
        GetOneLine(cin, str);

        // Здесь начинается разбор принятой строки.
        if(Parser())
            cout << "+++++ OK! +++++\n";
        else
            cout << "----- Fatal error (look upper error message)! -----\n";
        // Вывод разобранной строки и значения позиции входной головки.
        cout << "Line: " << str;
        strCursor = " Pos: " + string(i, '-');
        strCursor += '^';
        cout << strCursor << " i = " << i << "\n\n";
    } while(str != "\n");
    cout << "Goodbye!\n";
    return 1;
}

```

Следует отметить, что две представленные программы отличаются только фрагментами, осуществляющими разбор. Тестовая функция у них совпадает. Трудно сказать, какой из методов лучше. Использование рекурсивного спуска позволяет написать программу быстрее, так как не надо строить автомат. Ее текст может быть и менее ступенчатым, если использовать инверсные условия проверки или другие методы компоновки текста. Лично я не вижу причин заменять рекурсивный спуск автоматом с магазинной памятью, особенно в том случае, если грамматику задавать с использованием диаграмм Вирта. Но в общем (в который раз!) можно сказать, что о вкусах не спорят.



## Тема 8. Использование динамически порождаемых автоматов для нисходящего разбора слева направо

### Содержание темы

Семантический разрыв между формальными грамматиками и автоматами с магазинной памятью. Модель динамически порождаемых конечных автоматов. Графический метаязык для описания динамически порождаемых конечных автоматов. Использование диаграмм Вирта для представления динамически порождаемых конечных автоматов, распознающих КС(1) грамматику.

Примечание. Возможно, что представленный ниже материал для большинства не покажется полезным с практической точки зрения. Но хотелось подвести хоть какое-то обоснование под используемые методы и практические приемы разработки трансляторов. При наличии представления о нескольких десятках языках программирования и более чем о десятке формальных моделей постоянно возникают маниакальные идеи разработать "супер" язык или модель. Для меня эта модель не является ни первой, ни последней. А.Л.

### Семантический разрыв между формальными грамматиками и автоматами с магазинной памятью

Применение автоматов с магазинной памятью для построения нисходящих распознавателей показывает, что одним из препятствий на пути эффективной разработки трансляторов является **семантический разрыв** [Майерс] между формальным описанием исходных языков и методами реализации трансляторов этих языков. **Семантический разрыв** проявляется в том, что объекты и операции, которыми разработчик манипулирует при описании языка, не совпадают с объектами и операциями, которыми разработчику приходится манипулировать при реализации транслятора.

Обобщенно, процесс построения типичного транслятора выглядит следующим образом:

1. разработка синтаксиса (формального описания) языка;
2. разработка распознавателя языка на основе полученного формального описания.

Описание синтаксиса осуществляется с использованием формальной грамматики, которая определяет синтаксическую структуру языка. Построение же распознавателя основывается на автоматной модели и оперирует соответствующими понятиями из теории автоматов, такими, как входной алфавит автомата, множество его состояний, множество магазинных символов и переходов.

Наличие семантического разрыва приводит к необходимости преодолевать ряд проблем в при разработке транслятора. **Первая из них** - *необходимость преобразования модели формальной грамматики в автоматную модель распознавателя*. Между этими моделями существует несколько противоречий:

1. **Противоречие между иерархической структурой исходного описания языка с использованием формальной грамматики и одноуровневой табличной моделью автомата, реализующего синтаксический разбор.** Формальные грамматики оперирует абстрактными понятиями, заложенными в основу языка, такими как *программа*, *тип* или *арифметическое выражение*. Все описание языка можно рассматривать как набор понятий, связанных между собой некоторыми отношениями. Для представления этих понятий в метаязыках используется нетерминальные символы. Одни нетерминалы могут описываться через другие, более мелкие, формируя, таким образом, некоторую структуру взаимосвязей между понятиями языка. Описание одного нетерминала может состоять из нескольких правил, но эти правила всегда рассматриваются вместе. В противоположность этому, автомат с магазинной памятью, построенный на основе формальной грамматики, оперирует понятиями магазинных символов, некоторые из которых могут соответствовать понятиям языка (нетерминалам исходной грамматики). Однако большинство из них представляют собой не имеющие смысла для разработчика компилятора языка символы, появившиеся в результате преобразования модели формальной грамматики к модели автомата с магазинной памятью, которыми, тем не менее,

разработчик вынужден манипулировать. Таким образом, при построении автомата теряется исходная структура правил языка. Кроме того, структурированность описания часто исчезает за счет вынужденного разложения единственного правила, описывающего некоторый нетерминал. Это разложение преобразует грамматику к виду, удобному для построения автомата с магазинной памятью. В результате, в процессе работы с преобразованной моделью, разработчик вынужден постоянно держать в уме исходную структуру понятий языка, что чревато большой вероятностью внесения в проект ошибок.

2. ***Противоречие между высокоуровневыми средствами описания языков и ограничениями на грамматики, используемые для перехода к автоматной модели распознавателя.*** Все современные метаязыки позволяют разработчику использовать относительно высокоуровневые механизмы, такие как альтернативные фрагменты правил, итерации и необязательные конструкции. Вместе с тем, на грамматику, из которой может быть получен искомым автомат с магазинной памятью, накладываются жесткие ограничения, касающиеся непосредственно вида ее правил, и эти ограничения не допускают использование описанных выше механизмов.

Формальное описание языка, в конечном счете, должно быть преобразовано в модель автомата, осуществляющего разбор. Это преобразование основывается на эвристических алгоритмах, каждый из которых ориентирован на конкретный класс грамматик и автоматов. В результате необходимости применения данного шага теряется связь между исходным описанием языка и его реализацией. Если в процессе разработки будут внесены изменения в автоматную модель, то они уже никаким образом не повлияют на исходное представление, которое часто является также и пользовательским описанием.

**Вторая проблема** - это необходимость преобразования грамматики. Для преобразования исходного описания языка в соответствующую автоматную модель, необходимо, чтобы грамматика принадлежала какому-либо конкретному классу. Если это условие не выполняется, то автомат не может быть реализован, и, следовательно, требуется преобразование грамматики. Если выясняется, что грамматика языка требует преобразования, то это может означать что:

1. язык принадлежит требуемому классу, то есть существует грамматика, описывающая его и обладающая необходимыми признаками;
2. язык не принадлежит требуемому классу, и любые попытки преобразования грамматики обречены на неудачу.

Первая ситуация возникает, если реализация языка не представляет трудностей, но при попытке его описания была использована грамматика, не удовлетворяющая необходимым условиям. В этом случае исходная грамматика может быть преобразована в допустимую, хотя это не является идеальным подходом. Дело в том, что в большинстве случаев преобразования приходится выполнять вручную разработчику компилятора, используя при этом эвристические приемы, каждый из которых ориентирован на конкретный класс грамматик. Существуют причины, по которым желательно избегать таких преобразований. Во-первых, трудно выбрать вид преобразования, во-вторых, невозможно гарантировать, что они не изменят создаваемый язык.

Вторая ситуация может возникнуть, когда описываемый язык сложен и содержит конструкции, которые могут представлять затруднения при их разборе. Попытка описания этих конструкций языка на синтаксическом уровне приводит к тому, что изначально описываемый язык не будет удовлетворять требованиям, допускающим его реализацию. Естественно, что любая грамматика этого языка не будет удовлетворять налагаемым условиям, и все попытки получения допустимой грамматики потерпят неудачу.

# Модель динамически порождаемых конечных автоматов

Для сокращения семантического разрыва предлагается модель *динамически порождаемых конечных автоматов*. Описание исходного языка с использованием данной модели, с одной стороны, сохраняет описанные выше и свойственные современным метаязыкам иерархичность и высокий уровень представления, а с другой, не требует преобразования в плоскую (одноуровневую) автоматную модель распознавателя, так как может быть реализована непосредственно в терминах используемых абстракций.

Обычный конечный автомат позволяет описывать только регулярные конструкции языков. Предлагаемая модель расширяет конечный автомат и позволяет описывать любой контекстно-свободный язык.

Динамически порождаемый конечный автомат (ДПК-автомат) - это четверка:

$$S = (A, Z, \delta, u_1),$$

где:

- $A = \{ a_1, \dots, a_s, \dots, a_S \}$  - множество всех состояний автомата (*алфавит состояний*);  $a_s \in (U \cup V)$ , где  $U = \{ u_1, \dots, u_m, \dots, u_M \}$  - множество *основных состояний*,  $V = \{ v_1, \dots, v_n, \dots, v_N \}$  - множество *зарезервированных конечных состояний* автомата,  $U \cap V = \emptyset$ ;
- $Z = \{ z_1, \dots, z_p, \dots, z_P \}$  - множество *входных символов* (*входной алфавит*);  $z_f \in (T \cup G)$ , где  $T = \{ t_1, \dots, t_k, \dots, t_K \}$  - множество *терминальных символов*,  $G = \{ g_1, \dots, g_p, \dots, g_P \}$  - множество *порождающих символов*,  $T \cap G = \emptyset$ ;
- $\delta: U \times Z \rightarrow A$  - *функция переходов*, реализующая отображение  $G$  в  $A$ ; функция  $\delta$  некоторым парам *основное состояние - входной символ*  $(u_m, z_f)$  ставит в соответствие состояние автомата  $a_s = \delta(u_m, z_f)$ ,  $a_s \in A$ ;
- $u_1 \in U$  - *начальное состояние* автомата.

Распознаватель языка  $R = \{ S_i \}_{i=1 \dots T}$  - это множество динамически порождаемых конечных автоматов.

Автомат  $S_1$ , используемый для инициализации процесса распознавания, называется *начальным*.

В исходный момент ДПК-автомат находится в состоянии  $u_0$ . При переходе в одно из зарезервированных конечных состояний  $v_n$  автомат завершает свою работу, возвращая логическое значение, определяемое этим состоянием. Возможны три конечных состояния и соответствующие им значения, которые приведены в таблице 1. Возвращаемое значение используется при динамическом порождении автоматов.

Таблица 1

*Множество зарезервированных конечных состояний*

Обозначение	Название	Возвращаемое значение
$v_1$	допуск	Истина
$v_2$	откат	Ложь
$v_3$	ошибка	Ложь

Входной алфавит  $Z$  ДПК-автомата объединяет два множества. Множество терминальных символов  $T$  - это символы алфавита описываемого языка, поступающие на вход автомата непосредственно из входного потока. Каждому символу  $g_p$  из множества порождающих символов  $G$  соответствует свой

конечный автомат  $S_i$ , входящий в состав распознавателя языка, то есть  $\forall g_p \in G \exists S_i \in R$ .

Порождающий символ  $g_p$  служит условным обозначением тех подцепочек символов во входной цепочке, которые могут быть распознаны ДПК-автоматом  $S_i$ , соответствующим данному порождающему символу.

Переход автомата в основное состояние  $u_m$ , по порождающему символу  $g_i$ , вызывает динамическое порождение экземпляра соответствующего ДПК-автомата  $S_i$ , который производит попытку распознавания текущей подцепочки. Возвращаемые порожденным автоматом логическое значение и результат сравнения текущего входного символа с терминальными символами входной цепочки, определяют срабатывание того или иного перехода в новое состояние. В общем случае возможно одновременное срабатывание нескольких переходов, но тогда полученная модель автомата не будет являться детерминированной. Поэтому в разработанную автоматную модель введен механизм приоритетов срабатывания переходов. В соответствии с этим механизмом для каждого состояния  $u_m$  пары  $(u_m, z_f)$ ,  $f = 1, \dots, F$  упорядочены в порядке проверки срабатывания соответствующих им переходов.

Описание языка представляет собой совокупность динамически порождаемых автоматов, каждый из которых описывает одно из понятий, определяющее синтаксис языка.

## Графический метаязык для описания динамически порождаемых конечных автоматов

Для визуального представления предлагаемой модели описания разработан графический метаязык (А-схемы), отображающий модель ДПК-автомата в виде ориентированного размеченного графа, вершины которого соответствуют состояниям автомата, а связи между вершинами - переходам. В качестве примеров описания грамматик с использованием этого метаязыка на рис. 8.1 приведено описание синтаксиса простейших арифметических выражений.

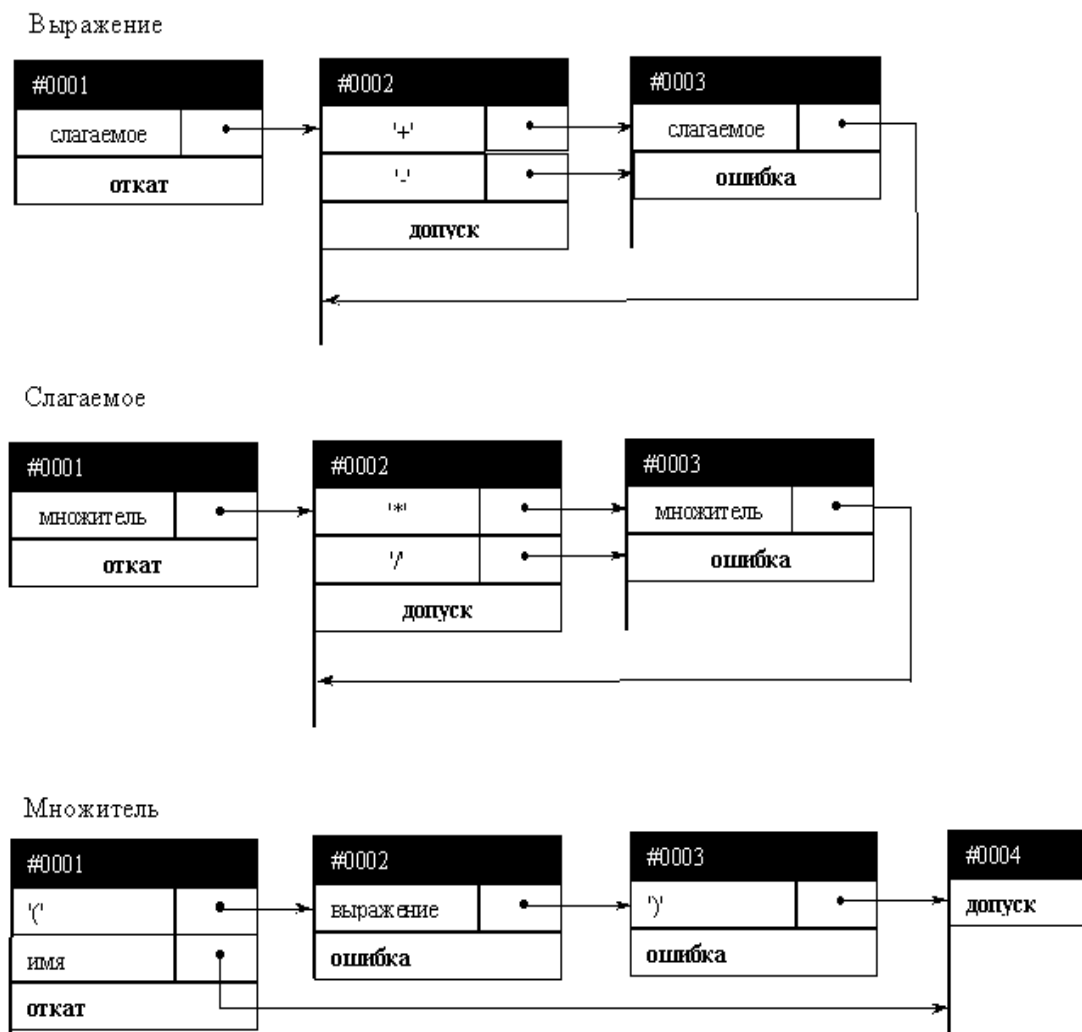


Рис. 8.1. А-схема простого арифметического выражения.

На рис. 8.2 - описание синтаксиса комментариев языка C++.

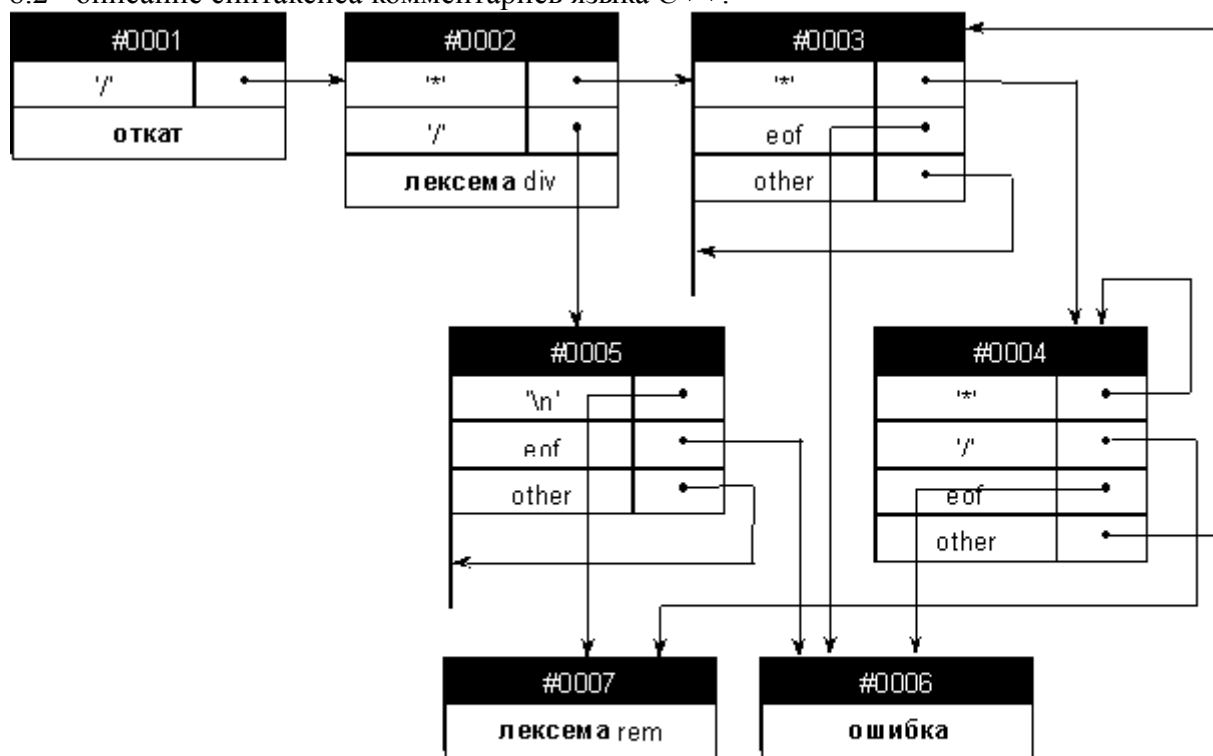


Рис. 8.2. А-схема комментариев языка C++.

Ниже перечислены несколько основных принципов, которые были заложены в основу метаязыка.

1. Каждое состояние автомата  $u_m$  представляется в виде вершины, состоящей из заголовка и последующих нескольких строк. Каждая строка соответствует переходу автомата и может иметь две различные формы представления. В первом случае строка содержит один из символов входного алфавита  $z_f$ , по которому может быть выполнен переход, и начальную точку, из которой проводится соответствующая ветвь перехода. Такая строка соответствует паре *текущее состояние - входной символ*  $(u_m, z_f)$ , которой функция  $\delta$  ставит в соответствие некоторое новое состояние  $a_s$ . Строки, соответствующие парам  $(u_m, z_f)$ , располагаются в порядке убывания приоритета сопоставленных им переходов. Во втором случае строка содержит название одного из зарезервированных конечных состояний  $v_n$ , вызывающих завершение работы автомата. Эта строка является сокращенной формой записи перехода в указанное конечное состояние, если не был выполнен ни один из переходов, сопоставленных предыдущим строкам вершины. Такая строка должна быть последней строкой вершины автомата.
2. Номер  $m$  состояния автомата отображается в заголовке состояния. Можно также задавать имена тем состояниям, которые имеют осмысленную интерпретацию. В этом случае вместо номера состояния в заголовке отображается его имя.
3. Каждое состояние автомата отображается с утолщенной левой границей, которая может быть продолжена за пределы исходной вершины. Эта линия и заголовок состояния предназначены для связывания состояний между собой через ветви переходов. Ветвь перехода берет начало из начальной точки, расположенной в соответствующей строке перехода, и оканчивается на заголовке или левой границе того состояния, в которое требуется перейти. Использование специальным образом выделенных начальных и конечных точек переходов позволяет использовать ветви в виде простых линий без стрелок на концах.
4. С каждой строкой перехода может быть связана некоторая семантика, для доступа к которой используются интерактивные возможности среды разработки, в которую предполагается интегрировать данный метаязык.

Метаязыком, наиболее близким к *A*-схемам, являются диаграммы Вирта и *R*-схемы. Можно легко построить алгоритмы преобразования между ними. Ниже перечислены действия, необходимые для преобразования диаграмм Вирта к *A*-схемам.

1. Диаграмма Вирта, описывающая нетерминал преобразуется в *A*-схему, описывающую соответствующий ДПК-автомат.
2. Вершины диаграммы Вирта соответствуют ветвям перехода между состояниями автомата.
3. Множества исходящих альтернативных связей диаграммы Вирта, образующие точки ветвления, соответствуют состояниям ДПК-автомата. Точки входа и выхода из графа диаграммы Вирта соответствуют начальному и конечному состояниям ДПК-автомата.

Диаграммы Вирта, соответствующие ранее представленным *A*-схемам выражения и комментария, приведены на рис. 8.3 и 8.4 соответственно.

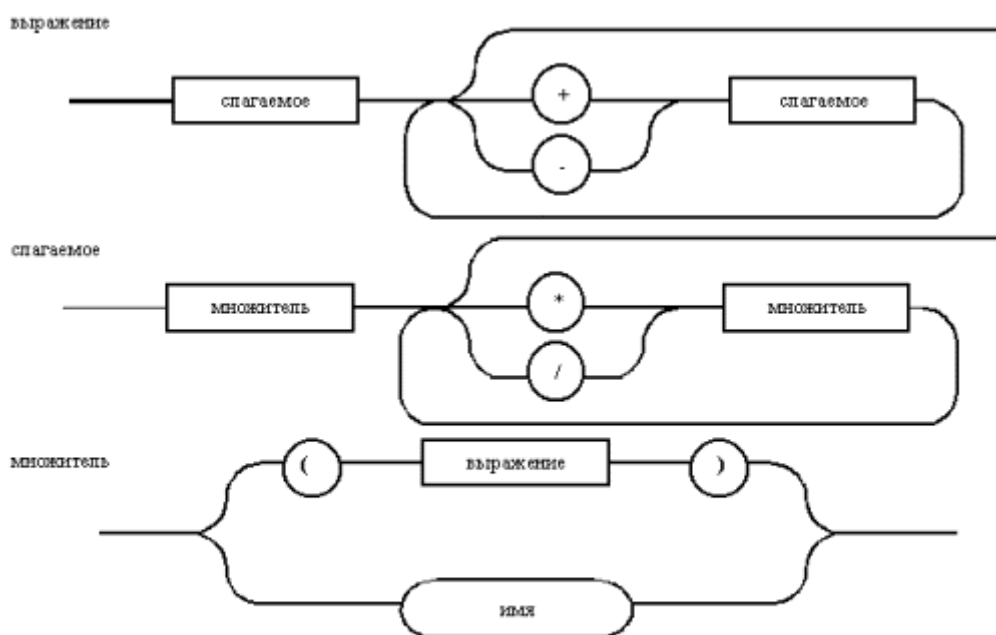


Рис. 8.3. Диаграммы Вирта, описывающие синтаксис простого арифметического выражения.

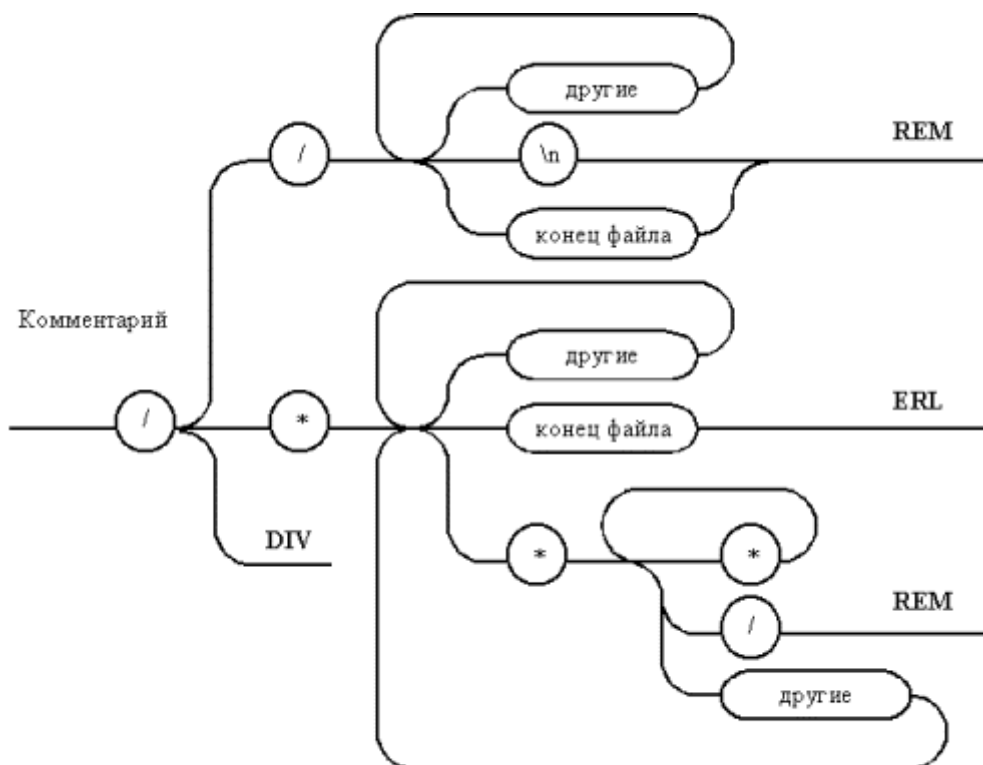


Рис. 8.4. Диаграммы Вирта, описывающие синтаксис комментария C++.

*Примечание. К сожалению (а может, к счастью:) все дальнейшие работы по созданию инструментальной системы на основе этого метаязыка оказались замороженными. Сказалась нехватка времени и других ресурсов. А так хотелось слепить графическую альтернативу yacc'у и lex'у! Что делать?! Не я первый, не я последний. А.Л.*

## **Использование диаграмм Вирта для представления динамически порождаемых конечных автоматов, распознающих КС(1) грамматику**

Из представленных примеров видно, что диаграммы Вирта можно непосредственно использовать не только в качестве метаязыка для представления формальных грамматик, но и как язык представления динамически порождаемых конечных автоматов. При этом, отдельным автоматом будет являться любое из правил уже разработанного синтаксиса языка программирования. Такой подход позволяет использовать диаграммы Вирта для построения нисходящего распознавателя точно так же, как они использовались для построения лексических анализаторов. Основным отличием от лексического анализа является то, что диаграммы распознавателя содержат нетерминалы, которые обрабатываются отдельными, динамически порождаемыми конечными автоматами.

Применение другой автоматной модели позволяет интерпретировать процесс нисходящего распознавания с использованием диаграмм Вирта следующим образом.

1. Каждая диаграмма Вирта задает один динамически порождаемый конечный автомат.
2. Входными символами любого динамически порождаемого конечного автомата являются терминальные символы, поступающие из входной цепочки и нетерминалы, определяющие отдельные правила.
3. Начальным состоянием динамически порождаемого автомата является входная дуга диаграммы Вирта.
4. Конечному состоянию автомата соответствует выходная дуга диаграммы Вирта.
5. Множество состояний любого динамически порождаемого конечного автомата является множеством точек ветвления связей диаграммы Вирта, описывающих различные альтернативы ветвления. При наличии только одной альтернативы состоянию автомата соответствует дуга, выходящая из терминальной или нетерминальной вершины.
6. Процесс распознавания начинается с порождения автомата, задаваемого правилом, описывающим начальный нетерминал.
7. Проход через терминальную вершину соответствует переходу по символу входной цепочки определяемому в этой вершине. При этом осуществляется переход из одного состояния автомата в другое.
8. Проход через нетерминальную вершину соответствует динамическому порождению нового конечного автомата в соответствии с именем нетерминала и продолжением разбора из его начального состояния.
9. Достижение конечного состояния эквивалентно достижению выхода диаграммы Вирта. В этом случае автомат сигнализирует об успешном завершении работы сигналом "допустить", который передается породившему его автомату. Родительский автомат считает в этом случае проведенный разбор нетерминала успешным и переходит в следующее состояние в соответствии с дугой выходящей на диаграмме Вирта из разобранного нетерминала.
10. Если автомат не может перейти из начального состояния в другое состояние, он порождает сигнал "отката" и передает управление своему родительскому автомату. Родительский автомат, получив данный сигнал, пытается осуществить анализ другой существующей альтернативы.
11. При невозможности перехода в другое состояние из состояния, не являющегося начальным, автомат сигнализирует об ошибке выдачей сигнала "отказ". Это ведет к отказу входной цепочки и завершению работы всех автоматов.
12. Цепочка допускается в том случае, если ее допускает автомат, порожденный для распознавания начального нетерминала.

# Используемые источники информации

## Литература

- [Аммерааль99] Аммерааль Леен. STL для программистов на C++. Пер. с англ. - М.: ДМК, 1999 - 240 с., ил.
- [АРНФТС] Шишмарев А.И., Заморин А.П. Англо-русско-немецко-французский толковый словарь по вычислительной технике. М.: Издательство "Русский язык", 1978.
- [Ахо78] Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. - М.: Мир, 1978.
- [Бек] Бек Л. Введение в системное программирование. - М.: Мир, 1988.
- [Браун] Браун П. Макропроцессоры и мобильность программного обеспечения. - М.: Мир, 1977.
- [Буч98] Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд./Пер. с англ. - М.: "Издательство Бином", СПб.: "Невский диалект", 1998 г. - 560 с.
- [Вайнгартен] Вайнгартен Ф. Трансляция языков программирования. - М.: Мир, 1977.
- [Вирт85] Вирт Н. Алгоритмы + структуры данных = программы. - М.: Мир, 1985.
- [Вирт89] Вирт Н. Алгоритмы и структуры данных. - М.: Мир, 1989.
- [Грис84] Грис Д. Наука программирования. М.: Мир, 1984.
- [Дейкстра78] Дейкстра Э. Дисциплина программирования. М.: Мир, 1978.
- [Зиглер] Методы проектирования программных систем./Пер. с англ. - М.: Мир, 1985. - 328 с.
- [Кауфман] Кауфман В. Ш. Языки программирования. Концепции и принципы. - М.: Радио и связь, 1993. - 432 с.
- [Кнут77] Кнут Д. Искусство для программирования для ЭВМ. Т2: Получисленные алгоритмы. - М.: Мир, 1977.
- [Кнут78] Кнут Д. Искусство для программирования для ЭВМ. Т3: Сортировка и поиск. - М.: Мир, 1978.
- [Кэмпбел] Кэмпбел-Келли М. Введение в макросы. - М.: Советское радио, 1978.
- [Легалов96] Легалов А.И., Сиротинина Н.Ю. Организация таблиц имен: методические указания по лабораторной работе для студентов специальностей 220100, 220400, 220600. КГТУ, Красноярск, 1996.
- [Легалов2000] Легалов А.И. Процедурно-параметрическая парадигма программирования. Возможна ли альтернатива объектно-ориентированному стилю? - Красноярск: 2000. Деп. рук. № 622-В00 Деп. в ВИНТИ 13.03.2000. - 43 с.
- [Льюис] Льюис Ф., Розенкранц Д., Стринз Р. Теоретические основы проектирования компиляторов. - М.: Мир, 1979.
- [Майерс] Майерс Г. Архитектура современных ЭВМ: В 2-х книгах. Кн. 1. - М.: Мир, 1985. - 364 с.
- [Маккиман] Маккиман У. Генератор Компиляторов. - М.: Статистика, 1980.
- [Рейоурд] Рейоурд-Смит В.Дж. Теория формальных языков. Вводный курс. - М.: Радио и связь, 1988.
- [Сибуя] Сибуя М., Ямамото Т. Алгоритмы обработки данных. - М.: Мир, 1986.
- [Страуструп99] Страуструп Б. Язык программирования C++, 3-е изд./Пер. с англ. - СПб.; М.: "Невский Диалект" - "Издательство БИНОМ", 1999 г. - 991 с.
- [Фостер] Фостер Дж. Автоматический синтаксический анализ. - М.: Мир, 1975.
- [Фьюэр] Языки программирования Ада, Си, Паскаль. Сравнение и оценка. / Под ред. Фьюэра А.Р., Джехани Н. - М.: Радио и связь, 1989.
- [Хантер] Хантер Р. Проектирование и конструирование компиляторов/ Пер. с англ.: - М.: Финансы и статистика, 1984. - 232 с.