Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

И. И. Пилецкий, В. В. Шиманский

МЕТОДЫ ТРАНСЛЯЦИИ

Методическое пособие
для студентов специальности 1-31 03 04 «Информатика»
всех форм обучения

Минск БГУИР 2012

УДК 004.415.3(076)
ББК 32.973.26-018.2я73
П32

Р е ц е н з е н т:
заведующий кафедрой многопроцессорных систем и сетей
Белорусского государственного университета,
кандидат технических наук Л. Ф. Зимянин

Пилецкий, И. И.

П32 Методы трансляции : метод. пособие для студ. спец. 1-31 03 04 «Информатика» всех форм обуч. / И. И. Пилецкий, В. В. Шиманский. – Минск : БГУИР, 2012. – 88 с.: ил.

ISBN 978-985-488-679-4.

В методическом пособии рассматривается описание методов, техник и технологий формального определения языков программирования и построения различных анализаторов исходных программ, используемых при разработке трансляторов. Пособие содержит методы и технологии построения семантических анализаторов, использующих различные эвристические техники анализа семантики.

Рекомендовано студентам, магистрантам и преподавателям для теоретического и практического освоения курса «Методы трансляции».

УДК 004.415.3(076)
ББК 32.973.26-018.2я73

ISBN 978-985-488-679-4

ОГЛАВЛЕНИЕ

1. ОБЩАЯ МОДЕЛЬ КОМПИЛЯТОРА 5

1.1. Основные задачи компиляторов 5
1.2. Интерпретатор 5
1.3. Компилятор 6
1.4. Объектная программа 7
1.5. Трансляция в ассемблер 8
1.6. Кросс-транслятор 9
1.7. Виртуальная машина 10
1.8. Компиляция «на лету» 10
1.9. Фазы компиляции 11

2. РБНФ, БНФ, СИНТАКСИЧЕСКИЕ ДИАГРАММЫ 18

2.1. Форма Бэкуса–Наура 18
2.2. Расширенная форма Бэкуса–Наура 18
2.3. Графическое представление 19

3. ФОРМАЛЬНОЕ ОПРЕДЕЛЕНИЕ СИНТАКСИСА ЯЗЫКОВ. КЛАССИФИКАЦИЯ ЯЗЫКОВ ПО ХОМСКОМУ 21

3.1. Задача определения языка 21
3.2. Определение грамматики 22
3.3. Некоторые свойства грамматик 23
3.4. Синтаксические деревья. Неоднозначность 24
3.5. Иерархия Хомского 26

4. ЛЕКСИЧЕСКИЙ АНАЛИЗ 30

4.1. Регулярные выражения и КА 32
4.2. Диаграмма состояний 33
4.3. Детерминированный КА 35

5. НЕДЕТЕРМИНИРОВАННЫЙ КОНЕЧНЫЙ АВТОМАТ 37

6. НИСХОДЯЩИЕ И ВОСХОДЯЩИЕ МЕТОДЫ СИНТАКСИЧЕСКОГО АНАЛИЗА 44

6.1. Синтаксический анализ 44
6.2. Классы синтаксических анализаторов 44
6.3. Грамматический разбор: общие понятия 45
6.4. Магазинные автоматы 47
6.5. Алгоритм разбора сверху-вниз 49
6.6. Перебор вариантов 52
6.7. Априорные критерии для разбора сверху вниз 53
6.8. Преобразование к нормальной форме LL(1) 54
6.9. Матрица предшественников 54
6.10. Метод рекурсивного спуска 55

7. ГРАММАТИЧЕСКИЙ РАЗБОР СНИЗУ ВВЕРХ 61

7.1. Восходящие анализаторы 61
7.2. Основа для методов разбора снизу вверх 64
7.3. МА и LR(k)-анализатор для методов разбора снизу вверх 65
7.4. Грамматики с операторным предшествованием 66
7.5. Грамматика с простым предшествованием 68
7.6. LR-таблицы разбора и алгоритм анализа цепочек 70

8. ПРЕДСТАВЛЕНИЕ ГРАММАТИКИ В ПАМЯТИ 75

9. ПОЛЬСКАЯ ЗАПИСЬ 77

9.1. Алгоритм перевода в постфиксную запись 78
9.2. Алгоритм перевода, проверки и вычисления. 79

10. СЕМАНТИКА 82

10.1. Идентификация 82
10.2. Контроль типов 83
10.3. Эквивалентность типов 84
10.4. Таблицы для контроля семантики 85

Литература 88

1. Общая модель компилятора

1.1. Основные задачи компиляторов

Компьютеры сами по себе способны выполнять только очень ограниченный набор операций, называемых машинными кодами. Во времена, когда появились первые компьютеры, программы писались в машинных кодах, представляющих собой последовательности двоичных чисел, однозначно воспринимаемых компьютером. В конце 50-х гг. прошлого века появились первые языки программирования, такие как язык ассемблера и Фортран. Для того, чтобы компьютер мог понять программу, написанную на некотором языке программирования, необходим переводчик (транслятор) такой программы в машинные коды. Отметим, что, если оператор языка ассемблера отображается при трансляции чаще всего в одну машинную инструкцию, предложения языков более высокого уровня отображаются обычно в несколько машинных инструкций.

Трансляторы бывают двух типов: **компиляторы (compiler) и интерпретаторы (interpreter)**. Процесс компиляции состоит из двух частей: **анализа (analysis) и синтеза (synthesis).** Анализирующая часть компилятора разбивает исходную программу на составляющие ее элементы (конструкции языка) и создает промежуточное представление исходной программы. Синтезирующая часть из промежуточного представления создает новую программу, которую компьютер в состоянии понять. Такая программа называется объектной программой. Объектная программа может в дальнейшем выполняться без перетрансляции. В качестве промежуточного представления обычно используются деревья, в частности, так называемые деревья разбора. Под деревом разбора понимается дерево, каждый узел которого соответствует некоторой операции, а сыновья этого узла – операндам.

1.2. Интерпретатор

В отличие от компилятора, интерпретатор не создает никакой новой программы, а просто выполняет каждое предложение языка программирования. Можно сказать, что результатом работы интерпретатора является «число».

Обычно интерпретатор, так же, как и компилятор, анализирует программу на входном языке, создает промежуточное представление, а затем выполняет операции, содержащиеся в тексте этой программы. Например, интерпретатор может построить дерево разбора, а затем выполнить операции, которыми помечены узлы этого дерева (рис. 1.1).

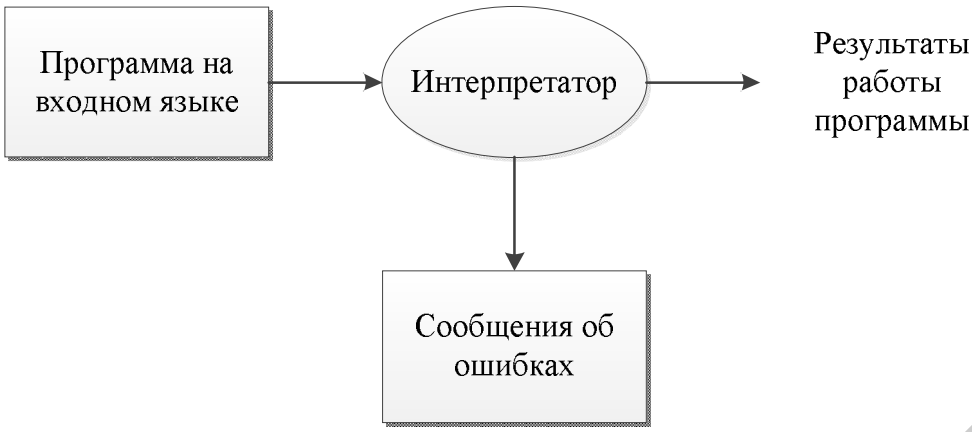


Рис. 1.1

В том случае, если исходный язык достаточно прост (например, если это язык ассемблера или Бэйсик), то никакое промежуточное представление не нужно, и тогда интерпретатор – это простой цикл. Он выбирает очередную инструкцию языка из входного потока, анализирует и выполняет ее. Затем выбирается следующая инструкция. Этот процесс продолжается до тех пор, пока не будут выполнены все инструкции, либо пока не встретится инструкция, означающая окончание процесса интерпретации (рис. 1.2).

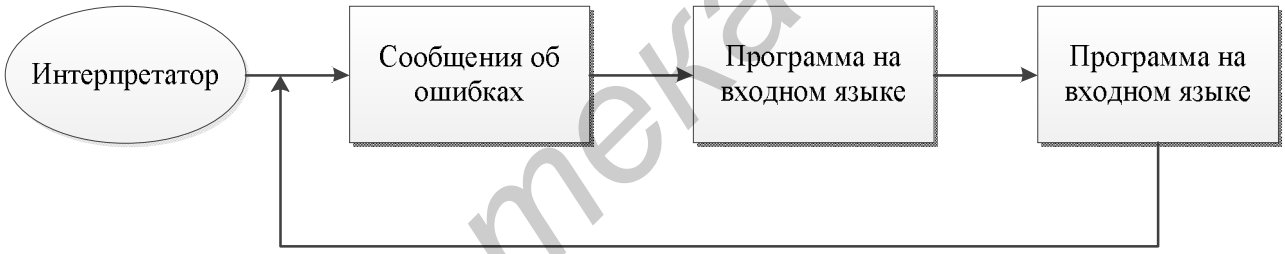


Рис. 1.2

1.3. Компилятор

Компилятор переводит программы с одного языка на другой. Входом компилятора служит цепочка символов, составляющая исходную программу на языке программирования L_1. Выход компилятора (объектная программа) также представляет собой цепочку символов, но принадлежащую другому языку L_2, например, языку некоторого компьютера. При этом сам компилятор написан на языке L_3, возможно, отличающемся от первых двух. Будем называть язык L_1 исходным языком, язык L_2 – целевым языком, а язык L_3 – языком реализации. Таким образом, можно говорить о компиляторе как об отображении множества L_1 в множество L_2, т. е. K_{L3}: $L_1 \rightarrow L_2$ (рис. 1.3).

Отметим, что далеко не всегда исходные программы корректны с точки зрения исходного языка. Более того, некорректные программы подаются на вход компилятору значительно чаще, чем корректные – таков уж современный процесс

разработки программ. Поэтому крайне важной частью процесса трансляции является точная диагностика ошибок, допущенных во входной программе.

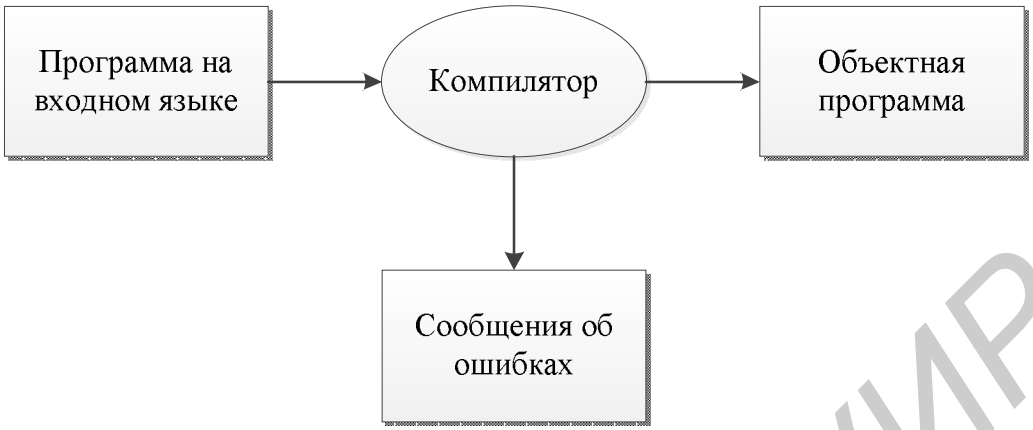


Рис. 1.3

Существует огромное количество различных языков программирования (>2500), начиная с таких традиционных языков программирования, как Фортран и Паскаль, и кончая современными объектно-ориентированными языками, такими, как C# и Java. Практически каждый язык программирования имеет какие-то свои особенности с точки зрения создателя транслятора.

1.4. Объектная программа

Объектная программа может быть:

- последовательностью абсолютных машинных команд;
- последовательностью перемещаемых машинных команд;
- программой на языке ассемблера;
- программой на некотором другом языке.

Наиболее эффективным методом с точки зрения скорости компиляции является отображение исходной программы в *абсолютную программу* на машинном языке, пригодную для непосредственного исполнения. Такой тип компиляции больше всего подходит для небольших программ, не использующих независимо компилируемых подпрограмм.

Однако для более сложных программ необходимо создавать объектные программы в форме последовательности *перемещаемых машинных команд*.

Перемещаемая машинная команда представляет собой команду, в которой адресация ячеек памяти производится относительно некоторого подвижного начала. Объектную программу называют также перемещаемым объектным сегментом. Этот сегмент может быть связан с другими сегментами, такими, как *независимо компилируемые подпрограммы пользователя, подпрограммы ввода−вывода и библиотечные функции*.

Такое связывание (создание единого перемещаемого объектного сегмента из набора различных сегментов) осуществляется программой, которая называется *редактором связей*. Далее единый перемещаемый объектный

сегмент или модуль загрузки размещается в памяти программой, называемой загрузчиком, которая превращает перемещаемые адреса в абсолютные. После этого программа готова к выполнению (рис. 1.4).

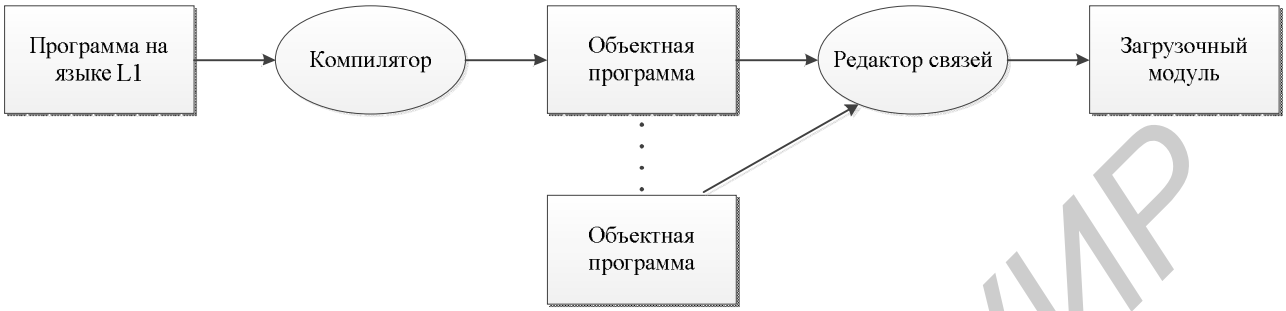


Рис. 1.4

1.5. Трансляция в ассемблер

Трансляция программы в ассемблер несколько упрощает конструирование компилятора. Однако такой подход удлиняет технологическую цепочку выполнения программы, так как объектная программа, порожденная компилятором, должна быть впоследствии обработана ассемблером, а также редактором связей и загрузчиком (рис. 1.5).

У трансляции в ассемблер есть несколько ощутимых преимуществ:

1) уровень ассемблера все-таки выше, чем у машинного кода, поэтому при трансляции в ассемблер программисту не приходится возиться с некоторыми утомительными техническими деталями, например, ассемблер берет на себя разрешение операторов безусловного перехода на еще неопределенные метки (переходы вперед), распределение памяти под данные, расчет длин переходов и т. п.;

2) использование ассемблера позволяет отследить целый ряд ошибок, которые могут возникнуть при генерации кода (например, неправильная мнемоника команды); при генерации в машинные коды «отловить» такие ошибки значительно труднее;

3) порождаемый текст на ассемблере значительно «читабельней», чем машинный код; это может помочь при отладке компилятора.

При генерации кода для платформы .NET MSIL представляет собой высокоуровневый ассемблер, максимально абстрагированный от конкретных целевых платформ.

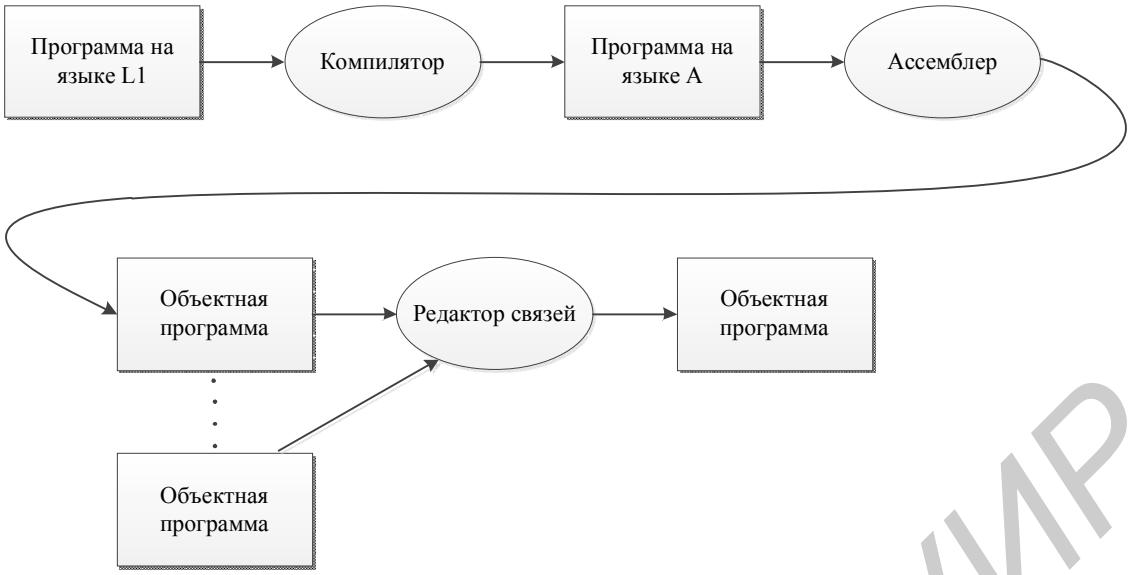


Рис. 1.5

1.6. Кросс-транслятор

Кросс-транслятор:

1) компилятор, который работает на одной платформе и создает код для другой платформы;

2) переносимый компилятор как вариант создания программы на языке более высокого уровня.

Пусть имеются два компьютера: компьютер M с языком ассемблера A и компьютер M_1 с языком ассемблера A_1. Кроме того, предположим, что имеется компилятор K_{A1}: $P \rightarrow A_1$, а сам компьютер M по каким-то причинам недоступен либо пока еще не существует компилятор K_A: $P \rightarrow A$. Нас интересует компилятор K_A: $L \rightarrow A$. В такой ситуации мы можем использовать M_1 в качестве *инструментальной* машины и написать компилятор K_P: $L \rightarrow A$, который принято называть *кросс-транслятором* (*cross-compiler*). Как только машина M станет доступной, мы сможем перенести K_P на M и «раскрутить» его с помощью K_A. Понятно, что это решение достаточно трудоемко, поскольку могут возникнуть проблемы при переносе, например, из-за различий операционных систем.

Под переносимой (*portable*) программой понимается программа, которая может без перетрансляции выполняться на нескольких (по меньшей мере на двух) платформах. В связи с этим возникает вопрос о переносимости объектных программ, создаваемых компилятором. Компиляторы, созданные по методикам, рассмотренным выше, порождают *непереносимые* (*non-portable*) объектные программы. Поскольку компилятор, в конечном итоге, является программой, то мы можем говорить и о переносимых компиляторах. Одним из способов получения переносимых объектных программ является *генерация объектной программы* на языке более высокого уровня, чем язык ассемблера. Такие компиляторы иногда называют ***конвертерами*** **(*converter*)**.

1.7. Виртуальная машина

Другой способ получения *переносимой* (*portable*) объектной программы связан с использованием *виртуальных машин* (*virtual machine*). При таком подходе исходный язык транслируется в коды некоторой специально разработанной машины, которую никто не собирается реализовывать «в железе». Затем для каждой целевой платформы пишется интерпретатор виртуальной машины.

Понятно, что архитектура виртуальной машины должна быть разработана таким образом, чтобы конструкции исходного языка удобно отображались в систему команд и сама система команд не была слишком сложной (рис. 1.6). При выполнении этих условий можно достаточно быстро написать интерпретатор виртуальной машины.

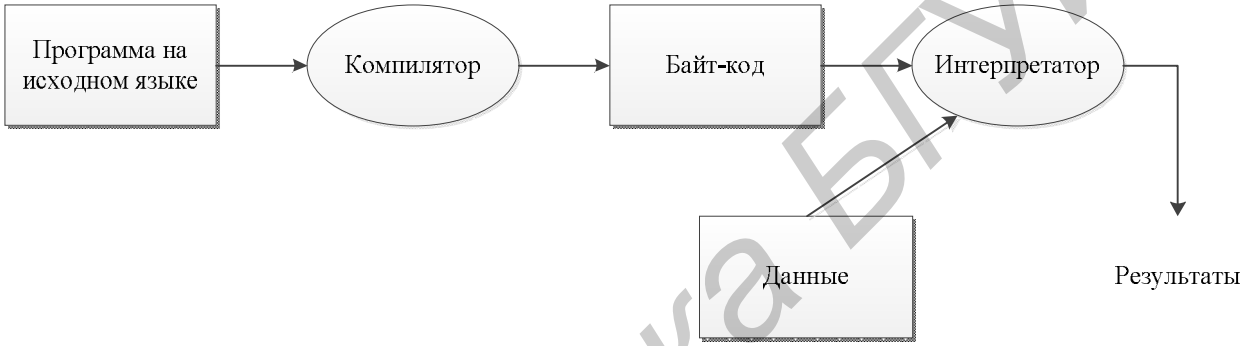


Рис. 1.6

Одна из первых широко известных виртуальных машин была разработана в 70-х гг. Н. Виртом при написании компилятора Pascal-P. Этот компилятор генерировал специальный код, названный P-кодом и представлявший собой последовательность инструкций гипотетической стековой машины. Сегодня идея виртуальных машин приобрела широкую известность благодаря языку Java, компиляторы которого обычно генерируют так называемый *байт-код*, т. е. объектный код, который представляет собой последовательность команд виртуальной Java-машины.

1.8. Компиляция «на лету»

Основная недостаток использования виртуальных машин заключается в том, что обычно время выполнения интерпретируемой программы значительно больше, чем время работы программы, оттранслированной в «родной» машинный код. Для того чтобы увеличить скорость работы приложений, была разработана технология *компиляции «на лету»* (*Just-In-Time compiling*; иногда такой подход называют также динамической компиляцией). Идея заключается в том, что JIT-компилятор генерирует машинный код прямо в оперативной памяти, не сохраняя его. Это приводит к значительному увеличению скорости выполнения приложения, именно так и устроена платформа .NET (рис. 1.7).

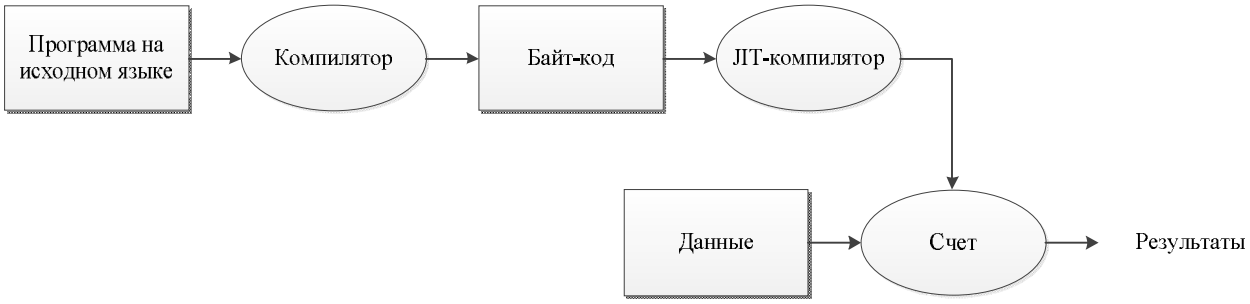


Рис. 1.7

Часто JIT-компилятор используется вместе с интерпретатором виртуальной машины. Организовывается это следующим образом. Вначале сгенерированный байт-код поступает на вход интерпретатора виртуальной машины. Одновременно с интерпретатором работает программа, которая вычисляет время интерпретации определенного фрагмента байт-кода, например процедуры. Если оказывается, что время интерпретации некоторого фрагмента кода достаточно большое, то вызывается JIT-компилятор, который транслирует его в «родные» машинные коды. Когда при выполнении приложения произойдет повторное обращение к этому «куску» кода, то он уже не будет интерпретироваться, вместо этого будет выполняться сгенерированный фрагмент машинного кода.

Использование связки «компилятор + интерпретатор + JIT-компилятор» позволяет заметно повысить скорость выполнения исходной программы, причем переносимость кода, создаваемого компилятором, естественно, сохраняется.

1.9. Фазы компиляции

Процесс создания компилятора можно свести к решению нескольких задач, которые принято называть фазами компиляции (compilation phases). Обычно компилятор состоит из следующих фаз:

- лексический анализ;
- синтаксический анализ;
- видозависимый анализ;
- оптимизация;
- генерация кода.

Продемонстрируем преобразования, которым подвергается исходная программа на фазах компиляции, на небольшом примере:

*position = initial + rate * 60,*

index=start +finish - midl.

Здесь все переменные вещественные.

1.9.1. Лексический анализ

Входом компилятора служит программа на исходном языке программирования. С точки зрения компилятора это просто последовательность символов. Задача первой фазы компиляции, *лексического анализатора* (*lexical analysis*), заключается в разборе входной цепочки и выделении некоторых более «крупных» единиц, *лексем*, которые более удобны для последующего разбора (рис. 1.8). Примерами лексем являются основные ключевые слова, идентификаторы, константные значения (числа, строки, логические) и т. п.

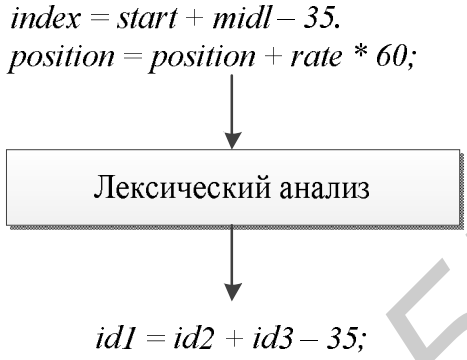


Рис. 1.8

На этапе лексического анализа обычно также выполняются такие действия, как удаление комментариев и обработка директив условной компиляции.

Для отображения некоторых лексем достаточно всего одного числа (это может быть, например, номер ключевого слова согласно внутренней нумерации компилятора), в то время как для записи других лексем может потребоваться пара, состоящая из номера лексического класса и ссылки на таблицу внешних представлений. Хорошая модель лексического анализатора – конечный преобразователь.

1.9.2. Синтаксический анализ

Синтаксический анализатор (*syntax analyzer, parser*) получает на вход результат работы лексического анализатора и разбирает его в соответствии с некоторой грамматикой. Эта грамматика аналогична грамматике, используемой при описании входного языка. Однако грамматика входного языка обычно не уточняет, какие конструкции следует считать лексемами (рис. 1.9).

Синтаксический анализ является одной из наиболее формализованных и хорошо изученных фаз компиляции. Различные методы построения синтаксических анализаторов будут рассмотрены далее.

После синтаксического анализа можно считать, что исходная программа преобразована в некоторое промежуточное представление. Одной из форм промежуточного представления является дерево разбора. В дереве разбора программы внутренние узлы соответствуют операциям, а листья представляют операнды.

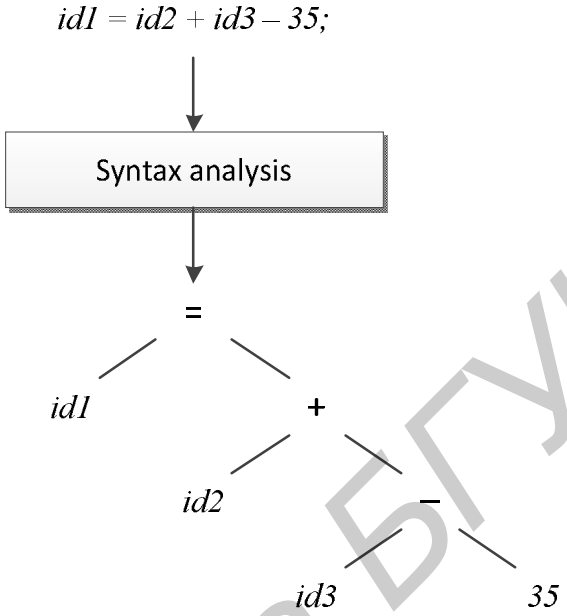


Рис. 1.9

1.9.3. Видозависимый анализ

Видозависимый анализ (*type checking*), иногда также называемый *семантическим анализом* (*semantic analysis*), обычно заключается в проверке правильности типов данных, используемых в программе (рис. 1.10). Кроме того, на этом этапе компилятор должен также проверить, соблюдаются ли определенные контекстные условия входного языка. В современных языках программирования одним из примеров контекстных условий может служить обязательность описания переменных: для каждого использующего вхождения идентификатора должно существовать единственное определяющее вхождение. Другой пример контекстного условия: число и атрибуты фактических параметров вызова процедуры должны быть согласованы с определением этой процедуры.

Такие контекстные условия не всегда могут быть проверены во время синтаксического анализа и потому обычно выделяются в отдельную фазу.

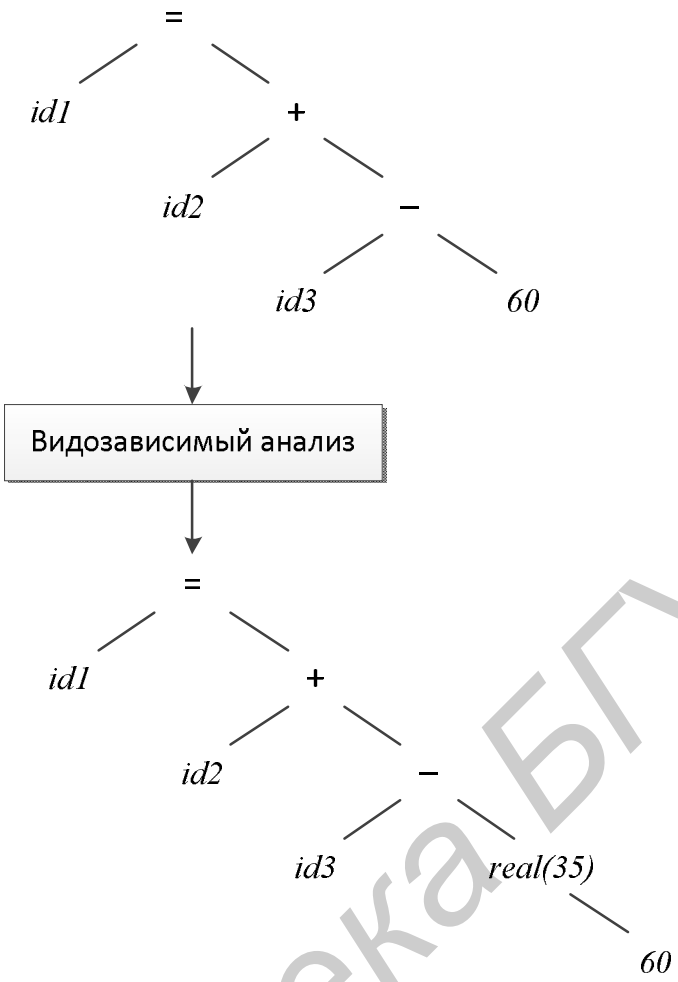


Рис. 1.10

1.9.4. Оптимизация кода

Основная цель *фазы оптимизации* (*code optimization*) заключается в преобразовании промежуточного представления программы в целях повышения эффективности результирующей объектной программы (рис. 1.11). Отметим, что существует различные критерии эффективности, например, скорость исполнения или объем памяти, требуемый программой. Очевидно, что все преобразования, осуществляемые на фазе оптимизации, должны приводить к программе, эквивалентной исходной.

Некоторые оптимизации тривиальны, другие требуют достаточно сложного анализа программы. Наиболее распространенными методами оптимизации являются:

– константные вычисления;
– уменьшение силы операций;
– выделение общих подвыражений;
– чистка циклов и т. д.

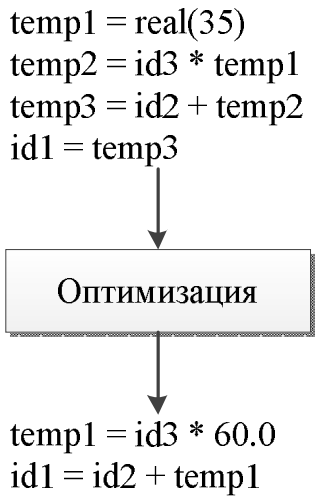


Рис. 1.11

1.9.5. Генерация кода

В результате всех предыдущих действий по оптимизированной версии промежуточного представления генерируется объектная программа. Эту задачу решает фаза *генерации кода* (*code generator*)(рис. 1.12).

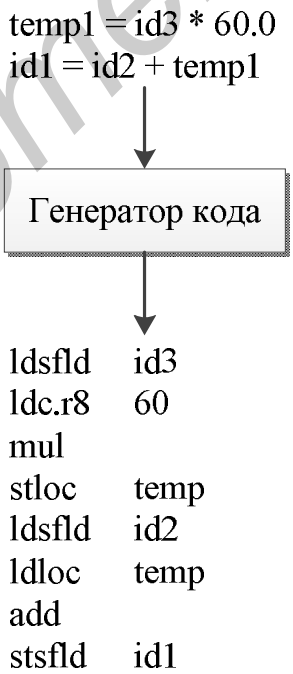


Рис. 1.12

Помимо собственно генерации кода, на этом этапе необходимо решить множество сопутствующих проблем, например:

– **распределение памяти**, т. е. *отображение имен исходной программы в адреса памяти*;

– **распределение регистров**, т. е. *определение для каждой точки программы множества переменных, которые должны быть размещены в регистрах*;

– выбор такой последовательности записи значений в регистры, которая избавила бы от необходимости частой выгрузки значений из регистров, а затем повторной загрузки.

Практически все эти задачи решаются окружением времени исполнения среды .NET и поэтому остаются за пределами данного курса.

1.9.6. Просмотры

Под просмотром (или проходом) компилятора понимается процесс обработки **всего**, возможно, уже преобразованного, текста исходной программы.

Одна или несколько фаз компиляции могут выполняться на одном просмотре. Например, лексический анализ и синтаксический анализ часто выполняются на одном просмотре, т. е. синтаксический анализатор может обращаться к лексическому анализатору за очередной лексемой лишь по мере необходимости. С другой стороны, некоторые оптимизации могут выполняться на нескольких просмотрах.

Передача информации между просмотрами происходит в терминах так называемых промежуточных языков. Таким образом, если компилятор состоит из N просмотров, то должно быть определено $N - 1$ промежуточных языков. Каков этот промежуточный язык, зависит от разработчиков компилятора. Обычно программа на промежуточном языке представляет собой синтаксическое дерево и, возможно, какие-то внутренние таблицы компилятора.

Желательно осуществлять относительно мало просмотров, т. к. чтение программы на одном промежуточном языке и запись ее на другом промежуточном языке могут занимать довольно много времени. С другой стороны, объединяя несколько фаз в один просмотр, следует помнить о том, что иногда невозможно выполнить некоторую фазу, не получив информацию из предыдущих фаз. Например, C# позволяет использовать имя метода до того, как он был описан. Понятно, что мы не можем выполнить видозависимый анализ до тех пор, пока мы не будем знать имена и типы всех методов объектов. Таким образом, эти задачи должны решаться на разных просмотрах (на рис. 1.13 – один просмотр, а точнее, полтора).

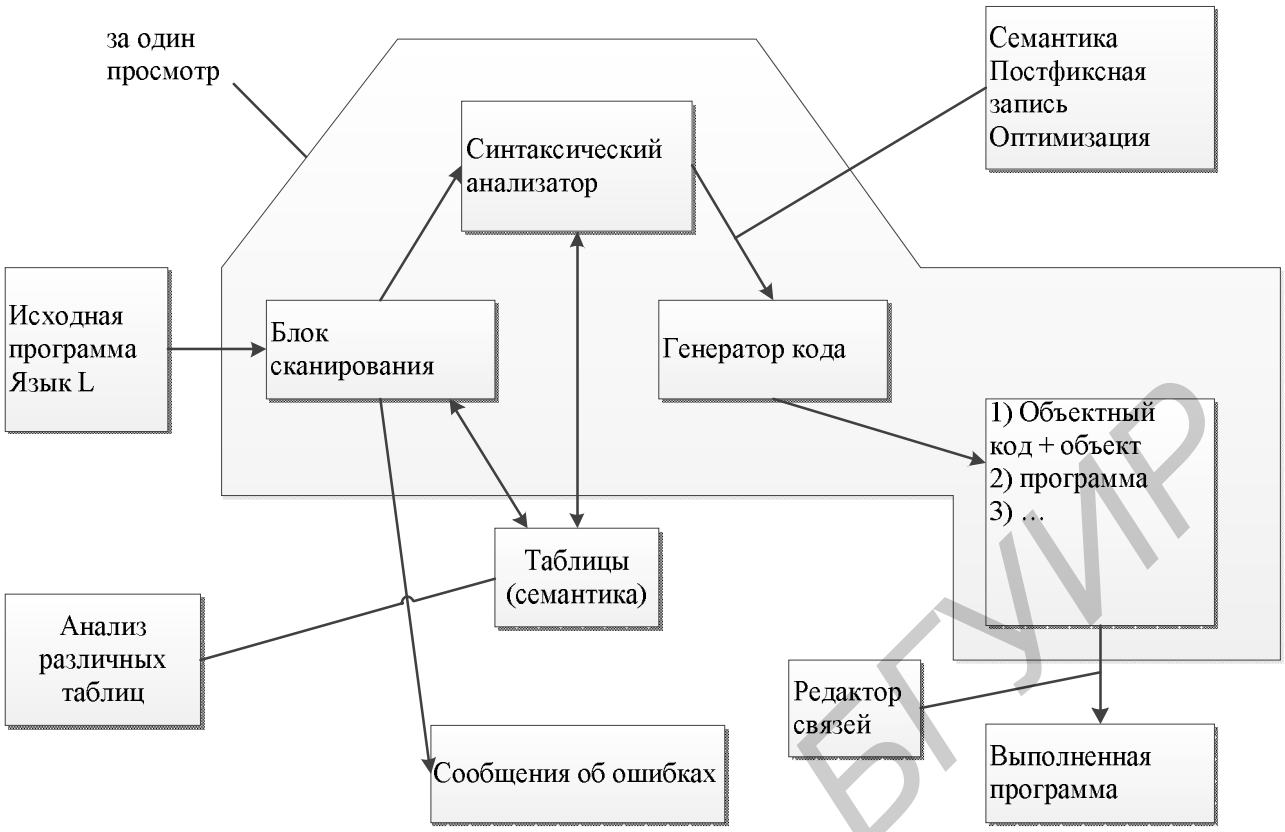


Рис. 1.13

Решение задачи:
- сколько фаз (возможно и более 80 фаз компиляции ПЛ/1);
- какие лексемы: if, +, /*, -<имя>, тип;
- на основании чего проверять синтаксис => метаязык (для описания языка);
- как описать семантику: нет формализма, но правила известны;
- каков генератор кода на целевую машину.

Пример:

```
x := y + z
if A > B
THEN switch = 0
ELSE switch = 1
IF A > 0 THEN N = 0
ELSE N > 1;
GOTO met1;
```

Как определить язык:

1) перечислить все цепочки, но $L_{цеп} \rightarrow \infty$;

2) определить правила, которые описывают язык с помощью порожденных правил.

2. РБНФ, БНФ, синтаксические диаграммы

2.1. Форма Бэкуса−Наура

Перейдем теперь к рассмотрению наиболее распространенных способов задания синтаксиса современных языков программирования. Естественно, наиболее часто используется некоторый вид формальных грамматик. Однако с помощью формальной грамматики определяется только контекстно-независимая составляющая языка. Поэтому для реальных языков программирования нельзя в общем случае сказать, что полученная с помощью такой грамматики цепочка терминалов является семантически правильной программой, т. к. правильная программа должна удовлетворять еще контекстным условиям.

Один из наиболее распространенных способов описания синтаксиса языка – это *форма Бэкуса─Наура* (J. W. Backus, P. Naur). Этот способ был разработан для описания Алгола-60, однако в дальнейшем он стал использоваться для многих других языков. При записи грамматики в форме Бэкуса─Наура используются два типа объектов:

─ основные символы (или терминальные символы, в частности, ключевые слова);

─ металингвистические переменные (или нетерминальные символы), значениями которых являются цепочки основных символов описываемого языка. Металингвистические переменные обозначаются словами (русскими или английскими), заключенными в угловые скобки (<...>);

─ металингвистические связки (::=, |)

Пример:

<целое> ::= <целое без знака> | + <целое без знака> | - <целое без знака>
<целое без знака> ::= <цифра> | <целое без знака> <цифра>
<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Форма Бэкуса─Наура не позволяет задавать контекстные условия. При использовании формы Бэкуса─Наура контекстные условия (т. е. семантика) задаются в словесной форме.

2.2. Расширенная форма Бэкуса─Наура

При определении синтаксиса языков Паскаль и Modula-2 Н. Вирт использовал *расширенную форму Бэкуса─Наура* (EBNF):

– нетерминалы записываются как отдельные слова;

– терминалы записываются в кавычках, например, «BEGIN»;

– вертикальная черта "|", как и прежде, используется для определения альтернатив;

– круглые скобки "( )" используются для группировки;

– квадратные скобки "[ ]" используются для определения возможного вхождения символа или группы символов;

– фигурные скобки "{ }" используются для определения возможного повторения символа или группы символов;

– символ равенства "=" используется вместо символа ::=;

– символ точка "." используется для обозначения конца правила;

– комментарии заключаются между символами (* … *);

– ε эквивалентно [ ].

Пример:

Integer = Sign UnsignedInteger.
UnsignedInteger = digit {digit}.
Sign = ["+"|"–"].
digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".

В 1981 г. Британский институт стандартов (British Standards Institute) опубликовал стандарт EBNF. BSI-стандарт получился более наглядным, чем расширенная форма Бэкуса─Наура, предложенная Н. Виртом:

─ элементы правил разделяются запятыми;

─ правила заканчиваются точкой с запятой;

─ пробелы не являются значащими.

Пример:

Constant Declaration = "CONST",
Constant Identifier , "=", Constant Expression, ";",
{Constant Identifier, "=", Constant Expression, ";"};
Constant Identifier = identifier;
Constant Expression = Expression;

2.3. Графическое представление

Совершенно иной способ представления синтаксиса – это *графическое представление*. Такое представление известно как синтаксические диаграммы (syntax diagrams) или синтаксические схемы (syntax charts). Они использовались для определения синтаксиса языков Паскаль, Modula-2. Они имеют форму блок-схем (flow diagram) (рис. 2.1).

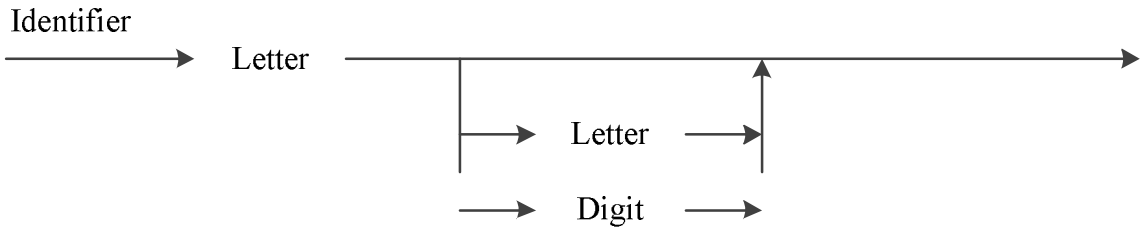


Рис. 2.1

Часто вместо таких синтаксических диаграмм используются другие, в которых терминалы записываются в кружочках, а нетерминалы – в прямоугольниках. Такие синтаксические диаграммы действительно похожи на блок-схемы (рис. 2.2).

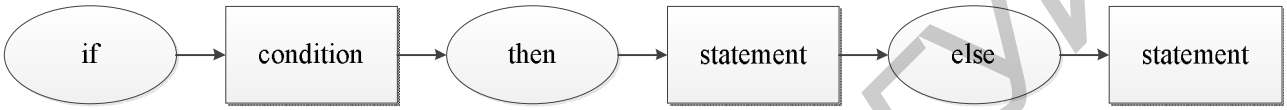


Рис. 2.2

Итак: метаязыки => грамматики => образуют метаязыки => порождающие правила.

Язык можно задать следующими способами:

1) перечислить все цепочки;
2) через программу-распознаватель;
3) с помощью грамматики (см. G1, G2 и G3).

Примеры:

G_1

<число> ::= <последовательность-десятичных-цифр>
<последовательность-десятичных-цифр> ::= <последовательность-десятичных-цифр><цифра>|<цифра>
<цифра> ::= 0|1|2|3|4|5|6|7|8|9

Могут быть различные грамматики для задания одних и тех же предложений.

Язык состоит из всевозможных вещественных чисел, например: 327, 0110, -12.

G_2

<число> ::= [<знак>]< последовательность-десятичных-цифр >
<последовательность-десятичных-цифр > ::= < последовательность-десятич-
ных-цифр > <цифра> | <цифра>
<цифра> ::= 0|1|2|3|4|5|6|7|8|9
<знак> ::= -|+

G_3

<S> ::= <A><B>
<A> ::= X|Y
<B> ::= Z|V
Язык: XZ, XV, YZ, YV.

3. Формальное определение синтаксиса языков. Классификация языков по Хомскому

3.1. Задача определения языка

Одна из первых задач, возникающих в процессе компиляции, – это определение рассматриваемого языка программирования. При рассмотрении языков, состоящих из конечного множества цепочек, проще всего явным образом перечислить все допустимые входные цепочки. Но что делать с языками, не вводящими никаких ограничений на длину входной цепочки? Для потенциально бесконечных языков потребуется ввести какой-то конструктивный способ описания, который позволит задать правила, описывающие порождаемый ими язык. Такое описание должно удовлетворять некоторым свойствам:

- само описание должно иметь конечную длину;
- для данного описания языка должен существовать алгоритм, который мог бы проверить принадлежность некоторой входной цепочки языку.

Существует целый ряд математических формализмов, в той или иной степени удобных для задания языков – вообще, этап анализа входной программы наиболее разработан и лучше всего поддержан математическими теориями.

Наиболее распространенным механизмом, во-первых, являются грамматики, которые задают все подходящие цепочки языка с помощью некоторых порождающих правил. Очевидное достоинство грамматик заключается в том, что существует множество систем, которые по заданной грамматике генерируют программу, проверяющую соответствие входной цепочки определяемому языку. Более того, полезную работу синтаксического анализатора (например, построение дерева разбора) можно проводить параллельно с самим распознаванием языка.

Другая часто используемая идея заключается в том, что *создается некоторый обобщенный алгоритм, проверяющий за конечное число шагов принадлежность данной цепочки языку. Такой алгоритм либо останавливается после конечного числа шагов и говорит* «*да*»*, либо останавливается и говорит* «*нет*». Теоретически нас могло бы устроить и зацикливание алгоритма на неподходящих входных цепочках, но на практике такой способ не совсем удобен.

3.2. Определение грамматики

Грамматики представляют собой наиболее распространенный класс описаний языков. При описании грамматики необходимо начать с определения алфавита языка, который задается как набор допустимых *терминальных* символов.

Кроме того, необходимо определить набор *правил вывода* вида $\alpha \rightarrow \beta$, с помощью которых строятся все цепочки языка. В левой и правой части этих правил могут встречаться специальные *нетерминальные* символы; в процессе вывода нетерминальные символы заменяются с помощью соответствующих правил до полной замены на соотвествующие терминалы. Наконец, грамматика должна включать в себя *начальный символ*, или аксиому, с которой начинается получение любого предложения языка.

Введем обозначения:
T – множество терминальных символов, строчные буквы;
N – множество нетерминальных символов, прописные буквы;
P – множество правил языка, синтаксических, порождающие правила вида (α, β), $\alpha \rightarrow \beta$, $\alpha \in V^+$, $\beta \in V^*$.

Определим также понятие выводимости:
если $\alpha\beta\gamma$ – цепочка, состоящая из символов языка G, а $\beta \rightarrow \delta$ – правило языка G, то $\alpha\beta\gamma \Rightarrow_G \alpha\delta\gamma$ ($\alpha\delta\gamma$ непосредственно выводима из $\alpha\beta\gamma$ *в G*). Рефлексивное и транзитивное замыкание этого отношения обозначим как $\alpha \Rightarrow^*_G \beta$ (цепочка β выводима из α; имя грамматики можно опускать для краткости $\alpha \Rightarrow^* \beta$).

Здесь:
S – начальный символ $\in$ N,
V – словарь языка $T \cup N$, или алфавит языка,
V^* – множество цепочек + пустая,
V^+ – без пустой,
L = {множество-всех предложений-заданных-G}.

Тогда грамматика G определяется как четверка: G = (T, N, P, S).

S обязательно должен быть в левой части некоторого правила P.

Предложения языка – это, во-первых, цепочка символов, выводимая из S, и, во-вторых, состоит только из терминальных символов.

Итак, для данной грамматики G = (T, N, P, S) язык L(G) есть множество цепочек T^+, выводимых из S (т. е. подмножество всех терминальных цепочек, выводимых из S, т. е. в T^+ нет пустых символов в множестве терминальных цепочек).

$L(G) = \{W \mid S \Rightarrow {}^*W,\ W \in T^+\}$ (*W выводима из S)

Пример:

Грамматикой, порождающей язык $\{0^n1^n \mid n\geq 0\}$, является G_0: $G_0=(\{0,1\},\{S\}, P, S)$, где $P = \{S\to 0S1,\ S\to\varepsilon\}$.

Пример:

Рассмотрим грамматику, порождающую язык $\{a^mb^n \mid m, n \geq 0\}$.
Такая грамматика имеет вид $G_1= (\{a,b\},\ \{S,A,\ B\},\ P,\ S)$,
где набор правил определяется следующим образом: $P = \{S\to AB,\ A\to aA,\ A\to\varepsilon,\ B\to bB,\ B\to\varepsilon\}$.

3.3. Некоторые свойства грамматик

Отметим некоторые свойства грамматик, которые нам придется учитывать в дальнейшем при построении грамматик реальных языков программирования.

Во-первых, *различные грамматики могут порождать один и тот же язык* (такие грамматики называются *эквивалентными*). Например, приведенная выше грамматика G_1 эквивалентна следующей грамматике $G_2 = (\{a,b\},\ \{S,Y\},\ P,\ S)$, где правила из P определены следующим образом: $P = \{\ S \to aS,\ S \to a,\ S \to b,\ S \to bY,\ Y \to b,\ Y \to bY,\ S \to \varepsilon\}$.

Несмотря на эквивалентность определяемых языков, одна грамматика может быть значительно удобнее другой с точки зрения ее использования в компиляторе. Поэтому в дальнейшем мы будем рассматривать некоторые приемы преобразования грамматик с целью улучшения удобства работы с языком.

Во-вторых, необходимо отметить, что определение грамматик не накладывает никаких ограничений на количество нетерминалов в левой части правил и приведенные выше примеры не должны создавать обманчивого впечатления, что все грамматики содержат один и только один нетерминал в левой части каждого правила. В качестве иллюстрации приведем следующий пример: $G_3 = (\{a,b,c\},\ \{S,B,C\},\ P,\ S)$, где P содержит следующие правила: $P = \{S \to aSBC,\ S \to abC,\ CB \to BC,\ bB \to bb,\ bC \to bc,\ cC \to cc\}$. Это абсолютно законная грамматика, порождающая язык $\{a^nb^nc^n,\ n\geq 1\}$.

В качестве другого примера приведем еще одну грамматику, эквивалентную грамматике G_1: $G_4 = (\{0,1\},\ \{A,S\},\ P,\ S)$, где $P = \{S \to 0A1,\ 0A \to$

00A1, S → ε}. В этом варианте левая часть одного из правил содержит пару из терминального и нетерминального символа.

В-третьих, приведем здесь соглашение, которое нам пригодится впоследствии: для обозначения *n* правил вида $\alpha \rightarrow \beta_1$, $\alpha \rightarrow \beta_2$, ..., $\alpha \rightarrow \beta_n$ мы будем использовать следующую запись: $\alpha \rightarrow \beta_1 | \beta_2 | ... | \beta_n$. В правилах такого вида вертикальная черта читается как «или».

3.4. Синтаксические деревья. Неоднозначность

Рассмотрим построение синтаксических деревьев для разбора цепочек языка, а на их основе, в свою очередь, свойства порождающих грамматик языка.

G_1 [число]
<число> ::= [<знак>] <последовательность десятичных чисел>
< последовательность десятичных чисел > ::= < последовательность десятичных чисел > <цифра> | <цифра>
<цифра> ::= 0|1|2|3|4|5|6|7|8|9
<знак> ::= + | -

G_1 [число], предложение – 15, его вывод.
<число> => < последовательность десятичных чисел > => < последовательность десятичных чисел ><цифра> => <цифра><цифра> => 15

Задача:
1) какие есть цепочки (перечисление цепочек);
2) распознавание цепочки (синтаксический анализ; принадлежит ли цепочка 125 языку $L(G_1)$ или нет).

Рассмотрим грамматику, которая определяет любые двоичные цифры:
G_{B1} = ({S}, {0, 1}, P, S).
{S} – не терминал
{0, 1} – терминальные символы
P – правила
S – начальный символ

P = { S ::= 0S
S ::= 1S
S ::= 0
S ::= 1 }

Вывод цепочки: 0101 S => 0S => 01S => 010S => 0101
Дерево вывода (рис. 3.1):

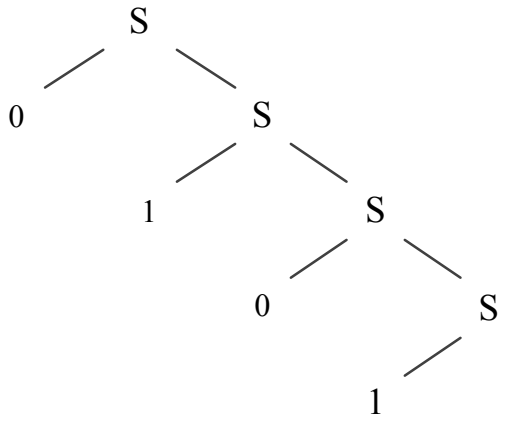


Рис. 3.1

Данная грамматика – правосторонняя, что бывает удобным для построения некоторого класса распознавателей. Вывод один – дерево разбора также одно.

G_{B2} = ({S, T}, {0, 1}, P, S)

P = { S ::= <T> <S> | 0 | 1
T ::= 0 | 1 }

Цепочка 01 (! 2 вывода) S_1 => TS => 0S => 01 (вывод правосторонний)
S_2 => TS => T1 => 01 (вывод левосторонний)

Дерево вывода:

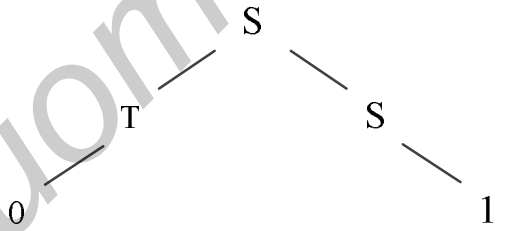


Рис. 3.2

Данная грамматика – левосторонняя, имеет два вывода и одно дерево разбора (рис. 3.2).

G_{B3} = ({S}, {0, 1}, P, S)

P = {S ::= SS | 0 | 1}

Цепочка 011 S_1 => SS => 0S => 0SS => 011
S_2 => SS => SSS => 0SS => 01S => 011

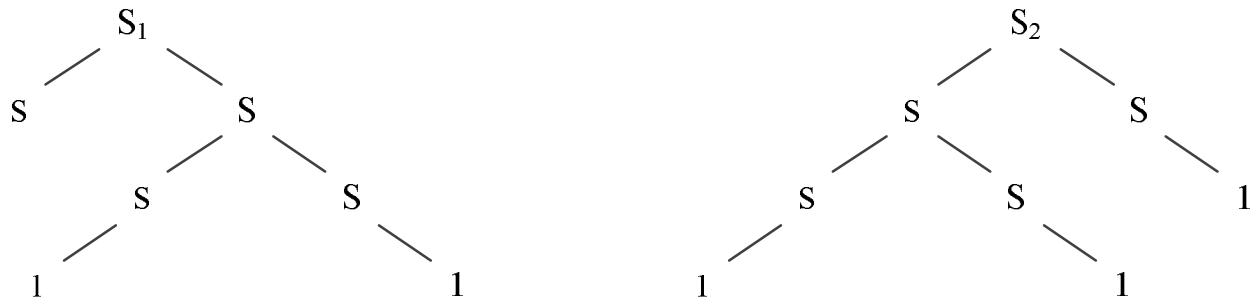


Рис. 3.3

Данная грамматика – левосторонняя и неоднозначная, имеет два вывода и два дерева разбора (рис. 3.3). Как делать разбор в распознавателе? Такие грамматики должны быть преобразованы к однозначным. Пример грамматики задания выражений (рис. 3.4).

G [выраж.] = (N, T, P, S)

<выражение> ::= <терм> | <выражение> + <терм>
<терм> ::= <первичный> | <терм> * <первичный>
<первичный> ::= x | y | z

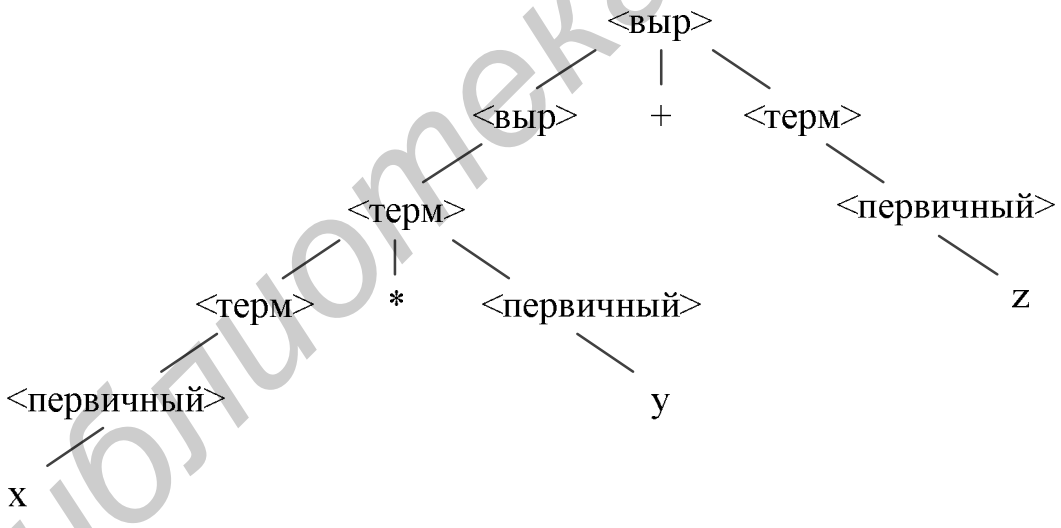


Рис. 3.4

3.5. Иерархия Хомского

Итак, мы убедились, что существует множество различных видов грамматик и, предположительно, некоторые из них могут оказаться удобнее для наших целей. Поэтому мы введем классификацию грамматик согласно их внешнему виду (эта классификация известна как *иерархия Хомского*, по фамилии автора – Ноэма Хомского).

Грамматика G называется:

– *выровненной вправо* (*праволинейной*), если любое правило из P имеет вид $A \to xB$ или $A \to x$, где A, B – нетерминалы, а x – терминал (возможно, пустой);

– *контекстно-свободной* (*бесконтекстной*), если любое правило из P имеет вид $A \to \alpha$, где A – нетерминал, α – нетерминал или терминал;

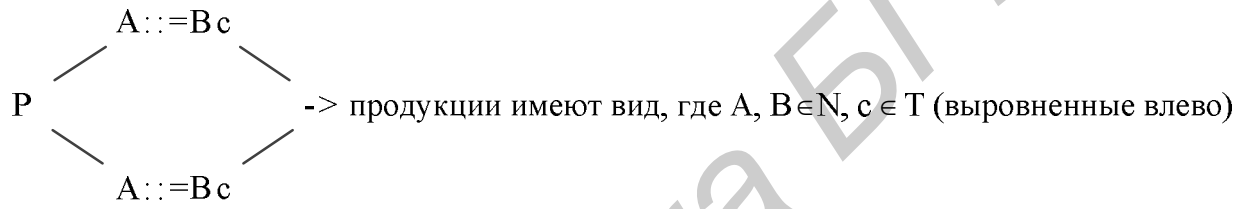
– *контекстно-зависимой* (*неукорачивающей*), если все правила из P имеет вид $\alpha \to \beta$, где $|\alpha| \le |\beta|$;

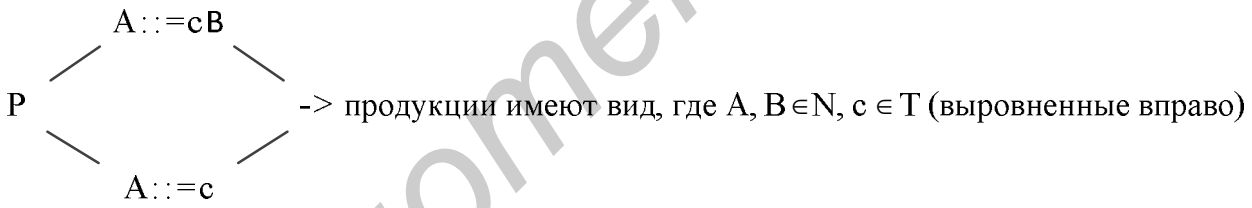
– *общего вида* (*без ограничений*), если грамматика не удовлетворяет ни одному из указанных выше ограничений.

Классификация по Хомскому

Типы грамматик по сложности распознавателей.

1. Регулярные грамматики (тип 3, автоматные):





Пример:

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

2. Контекстно-свободные грамматики (тип 2) (бесконтекстные) (используются для большинства языков программирования).
Продукция P имеет вид:

$$A ::= W,\ W \in (N \cup T),\ A \in N$$

Пример:
<формула> ::= <терм> {<плюс-минус><формула>} (повторять 0 –> n раз)
<терм> ::= <элемент> {<умножить-разделить><элемент>}
<элемент> ::= (<формула>)<число>
<плюс-минус>::=+/-
< умножить-разделить>::=*/ /

Пример (цепочки языка): 3+5*2, (12+2)*3

3. Контекстно-зависимая грамматика (тип 1, неукорачивающиеся)

$P \rightarrow A::=B$, где $A \in N^+$, $B \in (N \cup T)^*$, $|A| \leq |B|$
$P \rightarrow \alpha A \beta ::= \alpha B \beta$

Пример:
S ::= aSBC | abC
CB ::= BC
bB ::= bb
cC ::= cc
bC ::= bc
цепочки: a..ab..bc..c, $a^n b^n c^n$
$a^2b^2c^2$ = aabbcc → aabbcC → aabbCC → aabBCC → aabCBC → aSBC → S

4. Грамматика без ограничений (с фразовой структурой, тип 0)

$P \rightarrow A ::= B$, $A \in (N \cup T)^+$, $B \in (N \cup T)^*$,
где + – без пустого символа,
* – с пустым символом.

Иногда приведенные выше классы нумеруют от трех до нуля и называют каждый класс «грамматикой типа *n*», например, грамматика общего вида называется грамматикой типа 0. Мы будем избегать этого обозначения, т. к. оно не проясняет суть вопроса.

Очевидно, что эта классификация – включающая, т. е. все контекстно-свободные грамматики являются и контекстно-зависимыми, все контекстно-зависимые грамматики являются грамматиками общего вида и т. д. Кроме того, можно показать, что существуют языки, принадлежащие к типу *i*, но не к типу *i*+1. Например, язык G_3 является контекстно-зависимым, но не контекстно-свободным, т. е. не существует контекстно-свободной грамматики, порождающей этот язык. С другой стороны, некоторые нерегулярные грамматики могут порождать регулярные языки (например, грамматика G_1 – нерегулярная, но порождаемый ею язык регулярен, т. к. эквивалентная G_1 грамматика G_2 регулярна). Наконец, отметим, что определение контекстно-зависимой грамматики запрещает использование правил вида $A \rightarrow \varepsilon$. Это сделано для того, чтобы алгоритм, определяющий принадлежность цепочки языку, не мог зациклиться.

3.6. Распознаватели для различных классов грамматик

Под *распознавателем* будем понимать обобщенный алгоритм, позволяющий определить некоторое множество (в нашем случае – язык) и использующий в своей работе следующие компоненты: *входную ленту, управляющее устройство с конечной памятью и дополнительную рабочую память*. Обычно считается, что управляющее устройство может только читать информацию, записанную на входной ленте (чтение производится с помощью входной головки, указывающей на текущий символ) и продвигаться по ней вперед и, возможно, назад. Распознаватель также может изменять состояние памяти, которая может быть организована как конечный линейный список ячеек или как стек (в русской литературе называемый также магазином). В качестве примеров распознавателей можно назвать машину Тьюринга, конечные и магазинные автоматы, которые должны быть известны студентам из предыдущих курсов.

Язык определяется путем задания некоторого множества допустимых заключительных состояний распознавателя: если цепочка, поданная на входную ленту, позволяет распознавателю выполнить последовательность шагов и остановиться в заключительном состоянии, то цепочка принадлежит языку.

Оказывается, каждому классу грамматик из иерархии Хомского соответствует класс распознавателей, определяющий тот же класс языков:

– язык L праволинейный тогда и только тогда, когда он определяется (односторонним детерминированным) конечным автоматом;

– язык L контекстно-свободный тогда и только тогда, когда он определяется (односторонним недетерминированным) автоматом с магазинной памятью;

– язык L контекстно-зависимый тогда и только тогда, когда он определяется (двусторонним недетерминированным) автоматом с магазинной памятью;

– язык L рекурсивно перечислимый тогда и только тогда (т. и т.т.), когда он определяется машиной Тьюринга (этими понятиями мы оперировать не будем; формально они определяются в других курсах).

Распознаватель:
Некоторый алгоритм, позволяющий определить язык (L) и использующий:
1) входную ленту;
2) управляющее устройство с конечной памятью;
3) дополнитель рабочей памяти (машина Тьюринга, конечные автоматы, магазинный автомат).

Если распознаватель останавливается в заключительном состоянии, то цепочка принадлежит языку (см. табл. 3.1).

Таблица 3.1

Языки по Хомскому	Распознаватель
Т.3 Регулярный L тогда и только тогда →	Конечный автомат (односторонний детерминированный)
Т.2 КС (язык КС тогда и только тогда) →	Автомат с магазинной памятью (односторонний недетерминированный)
Т.1 КЗ (язык КЗ тогда и только тогда) →	Автомат с магазинной памятью (двусторонний недетерминированный)
Т.0 Язык L рекурсивно перечислимый тогда и только тогда →	Когда определяется машиной Тьюринга

4. Лексический анализ

Исходное текстовое представление программы не очень удобно для работы компилятора, поэтому во время анализа программа прежде всего разбивается на последовательность строк, или, как принято говорить, *лексем* (*lexeme*). Множество лексем разбивается на непересекающиеся подмножества (лексические классы). Лексемы попадают в один лексический класс, если они неразличимы с точки зрения синтаксического анализатора. Например, во время синтаксического анализа все идентификаторы можно считать одинаковыми.

Размеры лексических классов различны. Например, лексический класс идентификаторов, вообще говоря, бесконечен. В большинстве языков программирования имеются следующие лексические классы: ключевые слова, идентификаторы, строковые литералы, числовые константы. Каждое подмножество сопоставляется с некоторым числом, называемым *идентификатором лексического класса* или просто *лексическим классом*.

От одного языка к другому варьируются правила использования символов языка, в частности, пробелов. В некоторых языках, таких как Алгол-68 или в ранних версиях языка Фортран, пробелы являются значащими только в строковых литералах. Рассмотрим популярный пример, иллюстрирующий потенциальную сложность распознавания лексем в Фортране. В операторе DO 5 I = 1,25 мы не можем определить, что DO не является ключевым словом до тех пор, пока не встретим десятичную точку.

С другой стороны, в операторе DO 5 I = 1,25 мы имеем семь лексем: ключевое слово DO, метку 5, идентификатор I, оператор =, константу 1, запятую и константу 25. Причем, до тех пор пока не встретим запятую, мы не можем быть уверены в том, что DO – это ключевое слово. Чтобы как-то разрешить эту ситуацию, Фортран 77 позволяет использовать необязательную запятую между меткой и индексом DO оператора. Использование такой запятой позволяет сделать оператор DO понятнее и более читабельным.

В большинстве современных языков программирования ключевые слова являются зарезервированными, т. е. их смысл предопределен и не может быть

изменен пользователем. Если ключевые слова не являются зарезервированными, то лексический анализатор должен уметь различать ключевые слова и определенные пользователем идентификаторы. Естественно, что это сильно затрудняет лексический анализ; например, в PL/1 вполне легален следующий оператор:

IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;

При разборе такого оператора необходимо постоянно переключаться с режима «THEN, ELSE как ключевые слова» на трактовку «THEN, ELSE как идентификаторы» и обратно.

Для хранения идентификаторов и констант используются: таблица представлений, таблица имен с пулом идентификаторов или что-то другое. В таблице представлений хранится по одному экземпляру всех *внешних представлений* идентификаторов (и, возможно, также всех констант). Затем идентификаторы заменяются на ссылку в таблицу – этот процесс называется *свертыванием*.

Одна из простейших форм организации таблицы – это массив указателей на строки. Однако при таком подходе замедляются два основных процесса, связанных с таблицей представлений: поиск идентификатора в таблице и добавление нового элемента. При этом поиск идентификатора в таблице является, наверное, самой массовой задачей в процессе компиляции, т. к. выполняется для каждого использующего вхождения идентификатора. Желательно добиться максимального быстродействия для этой операции.

Поэтому более распространена другая форма организации таблицы представлений – в виде набора хэшированных списков. Для этого выбирается некоторая *хэш-функция* (в русскоязычных изданиях иногда также называемая функцией расстановки), выдающая по данному идентификатору некоторое число от 0 до H–1, где H – константа, называемая длиной оглавления. Затем все идентификаторы с одинаковым хэш-значением связываются в список. Таким образом, для того, чтобы проверить, встречался ли уже новый идентификатор в программе или нет, достаточно сравнить его только с идентификаторами из таблицы представлений, имеющими одинаковое хэш-значение. Агрегированная сущность «Таблица имен» приводится в разделе описания семантики.

Замечено, что большинство использований идентификатора находится недалеко от места его описания, поэтому рекомендуется добавлять новые идентификаторы в начало хэш-списка, а не в конец. Это повышает скорость поиска, а также упрощает поддержку реализации стандартных правил видимости в языках с блочной структурой. Например, перед входом в блок можно запоминать текущее состояние хэш-таблицы, а затем при поиске идентификатора внутри данного блока считать активным идентификатором первую переменную с данным именем, встреченную в хэш-таблице. Затем при выходе из блока необходимо восстанавливать предыдущее состояние хэш-таблицы.

Нетрудно заметить, что большинство распознаваемых лексем носят четко заданную структуру и потому возникает естественное желание применить к задаче лексического анализа теорию языков, т. е. описать с помощью какого-либо формализма характер цепочек, принимаемых на вход, а затем автоматически сгенерировать по этому описанию лексический анализатор. Так, ниже в табл. 4.1 выделены классы лексем и формализм, описывающий их.

Таблица 4.1

Классы лексем	**Формализм G (типа 3) ⇔ Регулярные выражения**
Идентификаторы	<идентификатор> ::= <буква> | <идентификатор> <буква> | <идентификатор> <цифра>
Служебные слова	BEGIN, …
Целые числа	<целое> ::= <цифра> | <целое> <цифра>
Вещественные числа	<вещественное число> ::= [+ | -] [<целое>].<целое> [E[+ | -]<целое>]
Разделитель однолитерный +, -, (,)	<разделитель> ::= + | - | (|) | / |
Разделитель двулитерный //, /*, **	<разделитель> ::= + | - | (|) | / |

Здесь все правила имеют следующий вид:
U ::= T T ∈ {терминалам}
U ::= VT V ∈ {нетерминалам}.

Реализация:
Грамматика типа 3 (Р. Г.) = Регулярное выражение → диаграмма → программа
↓
КА – конечный автомат → программа

4.1. Регулярные выражения и КА

Регулярное выражение – это:
1) A (алфавит терминалов – или пусто);
2) P, Q – регулярные выражения (состоят из элементов A), то =>
3) PQ – регулярное выражение (Q следует за P),
   P|Q (регулярное выражение) P или Q,
   P^* (нуль или более экземпляров P).

Примеры:

1) A – {a, b}
   (aab | ab)* => aababaab
                  ababab
2) A – {<буквы, цифры>}
   <идент> ::= <буква> | <идент> <буква> | <идент><цифра>
   идентификатор => б(б|ц)*

3) G [<вещ-число>] = (по грамматике типа 3 строится регулярное выражение)
   = [+ | -] [<целое>].<целое> [E[+ | -]<целое>] =
   = [+ | -] цифра*.цифра цифра* [E[+ | -] цифра цифра*]

В примерах 2 и 3 регулярные грамматики и регулярные выражения определяют один и тот же язык. По имеющемуся регулярному выражению легко написать лексический анализатор вручную, т. к. алгоритм разбора присутствует прямо в описании регулярного выражения.

4.2. Диаграмма состояний

Построение и распознавание цепочек:

G[Z] = Z ::= U0 | V1
       U ::= Z1 | 1 => L(G) = { цепочки начинаются 01, 10}.
       V ::= Z0 | 0

Построение диаграммы состояний:
1) выберем S – начальное состояние (нетерминал не принадлежит N{Z, U, V});
2) любой нетерминал G = состояние: S, Z, U, V – это вершины графа;
3) любому правилу типа Q ::= T => дуга (с пометкой T от начального состояния к Q: S → Q) U ::= 1, V ::=0 – это дуги графа из начального состояния S;
4) любому правилу типа Q ::= RT => дуга (с пометкой T от R к Q: R -> Q)
Z ::= U0, Z ::= V1, U ::= Z1, V ::= Z0 – это дуги графа для других состояний (рис. 4.1).

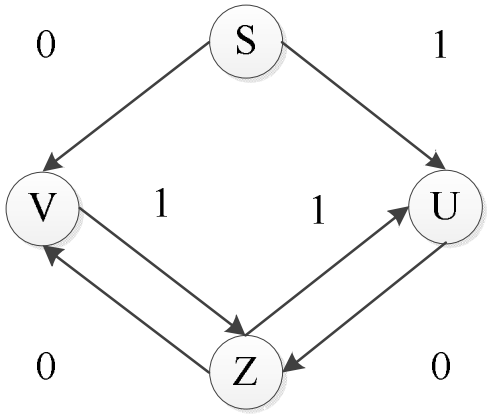


Рис. 4.1

Распознавание цепочки X:

1) первое начальное состояние S
   $X = x_1x_2x_3x_4x_5$ (начать и до конца);
2) сканировать следующую литеру X и по дуге с пометкой X_i к следующему состоянию. Если нет состояния, то ошибка.

Конец цепочки: если x_i – конец и состояние Z, то цепочка $X \Rightarrow \in L(G)$
Разбор и синтаксическое дерево для цепочки X = 101001 (см. табл.4.2 и рис. 4.2):

Синтаксическое дерево разбора легко построить по диаграмме состояний (см. рис. 4.2).

Таблица 4.2

Шаг	Состояние в диаграмме	Остаток цепочки X
1	S	101001
2	U	01001
3	Z	1001
4	U	001
5	Z	01
6	V	1
7	Z	

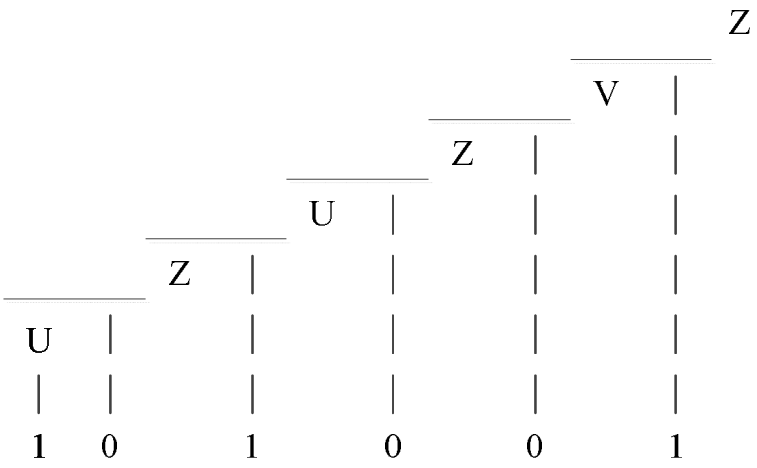


Рис. 4.2

Для обработки ошибок лучше не искать все состояния перехода, а добавить состояние F и все необходимые дуги из других состояний (рис. 4.3).

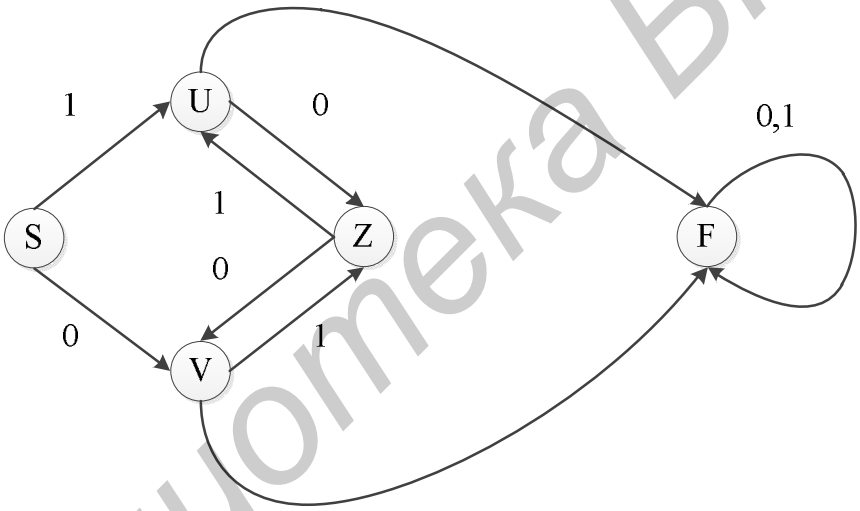


Рис. 4.3

4.3. Детерминированный КА

Конечный автомат – это один из самых простых механизмов, используемых при работе с языками. У этого распознавателя есть только фиксированный набор ячеек памяти, а управляющее устройство может только сдвигаться вправо по входной ленте и изменять состояния памяти. Основная часть конечного автомата – это *функция перехода*, определяющая возможные переходы для любого текущего состояния и любого входного символа. Подразумевается, что допускается возможность перехода сразу в несколько

различных состояний автомата, т. е. управляющее устройство распознавателя может быть и недетерминированным. Недетерминированность понимают следующим образом: если возможно сразу несколько переходов из данного состояния, то создается несколько копий автомата, по одному на каждое новое состояние. Цепочка считается принадлежащей языку, если хотя бы одна из последовательностей шагов завершается в заключительном состоянии.

В принципе, для определения последующих действий конечного автомата достаточно знать текущее состояние управляющего устройства плюс последовательность все еще необработанных символов на входной ленте. Этот набор данных называется конфигурацией автомата.

Детерминированный КА – это следующая пятерка (Грис):
КА = (K, VT, M, S, Z),
где
K – алфавит состояний множества,
VT – входной алфавит множества,
M – отображение $K \times VT \rightarrow K$, т. е. M(Q, T) = R, Q ::= RT,
S – начальное состояние, $S \in K$,
Z – множество заключительных состояний, $\neq \varnothing$.

Отображения определяются как:

1) M(Q, Λ) = Q, для любого Q (сост.);
2) M(Q, Tt) = M(M(Q, T), t) = M(P`, t), где t – входная цепочка; P` = M(Q, T) в состоянии Q при входной цепочке Tt, применяем отображение M → P`= = M(Q, T); $t \in VT^+$; $T \in VT$;
3) t допускается, если M(S, t) = P; $P \in \{Z\}$; $P = \{p_1, p_2, \ldots, p_n\}$ существует последовательность состояний из S →Z.

Два КА: $КА_1 = (K_1, VT_1, M_1, S_1, Z_1)$ и $КА_2 = (K_2, VT_2, M_2, S_2, Z_2)$ **эквивалентны**, если они распознают один и тот же язык над алфавитом VT.

$$VT_2 = VT_2$$

Утверждение: для любого L типа 3 (рекурсивный) существует КА такой, что L = G[S] ≈ L(KA).

Пример:

1 2 3 4 5
КА = ({S, Z, U, V, F}, {0, 1}, M, S, {Z})
M(S, 0) = V
M(S, 1) = U
M(U, 0) = Z
M(U, 1) = F
M(V, 0) = F
M(V, 1) = Z
M(Z, 0) = V
M(Z, 1) = U
M(F, 0) = F
M(F, 1) = F

Реализация 1:

Шаг 1. Строим таблицу состояний автомата (табл. 4.3):

Таблица 4.3

	Состояния	**S1**	**…**	**…**	**…**	**Sn**
Вх. литеры		S	Z	U	V	F
T1	0	V4	V4	Z2	F5	F5
…	1	U3	U3	F5	Z2	F5
Tm						

A(i, j) = K – номер состояния Sk,
M(Si, Tj) = Sk,
S1 – начальное состояние.

Шаг 2. Если нужно, строим диаграмму состояний.

Шаг 3. Пишем программу, где в зависимости от входного символа выполняем переход в следующее состояние. Если цепочка прочитана и произошел останов в конечном состоянии Z, то цепочка правильная, иначе – ошибка.

Реализация 2:

```
q = q0;
c = GetChar();
while (c! = eof)
{
   q = move(q,c);
   c = GetChar();
}
if (q is in StopA)
   return “Y”;
else return “N”;
```

5. Недетерминированный конечный автомат

Существуют грамматики, у которых имеется несколько порождающих правил с одинаковыми правыми частями. Такие грамматики порождают несколько переходов из одного и того же состояния. Так, для G[X] – существуют правила V ::= UT, W ::= UT => в диаграмме две дуги с T (рис. 5.1).

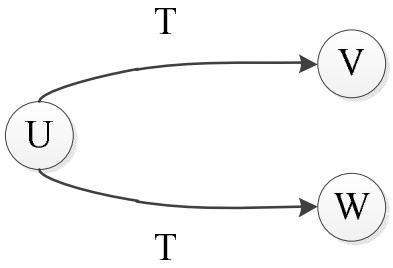


Рис. 5.1

Отображение M(Q, T) неоднозначно.
Определим НКА как следующую пятерку:
НКА = (K, VT, M, S, Z),
где K – алфавит состояний, множество;
VT – входной алфавит, множество;
M – отображение $K \times VT \rightarrow K$:
1) не единственное, возможно пустое M(Q, T)→R1
|-----→R2
|-----→ Λ
2) несколько начальных состояний S;
S – начальное состояние, $S \subseteq K$,
Z – заключительное состояние, $Z \subseteq K$.

Расширим отображение M до $K \times VT^*$, VT^* допускается пусто.
1) M(Q, Λ) = {Q}
2) M(Q, Tt) = M({P1, P2, …, Pn}, t); $t \in VT^+$, $T \in VT$.
Переходим в состояние M(Pi, T), а затем переходим к M(P, t)
t – допускается, если
3) M(S, t) = P, где $P \in \{Z\}$, т.е. $P \in \{M(S, t)\}, \{Z\}$.

Имеет место следующая **теорема**: если L = L(M) для некоторого недетерминированного конечного автомата M, то существует L = L(M') для некоторого детерминированного автомата M'.

Эта теорема доказывается конструктивным образом, т. е. путем указания общего алгоритма построения детерминированного автомата M', определяющего тот же язык, что и M. Пусть M = (Q, Σ, δ, q_0, F); тогда мы определим M' = (Q', Σ, δ', q_0', F').

Пример:

G[Z] = два смежных 0 или 1
Z ::= U1 | V0 | Z0 | Z1
U ::= Q1 | 1
V ::= Q0 | 0
Q ::= Q0 | Q1 | 0 | 1

Диаграмма для G[Z] (рис. 5.2):

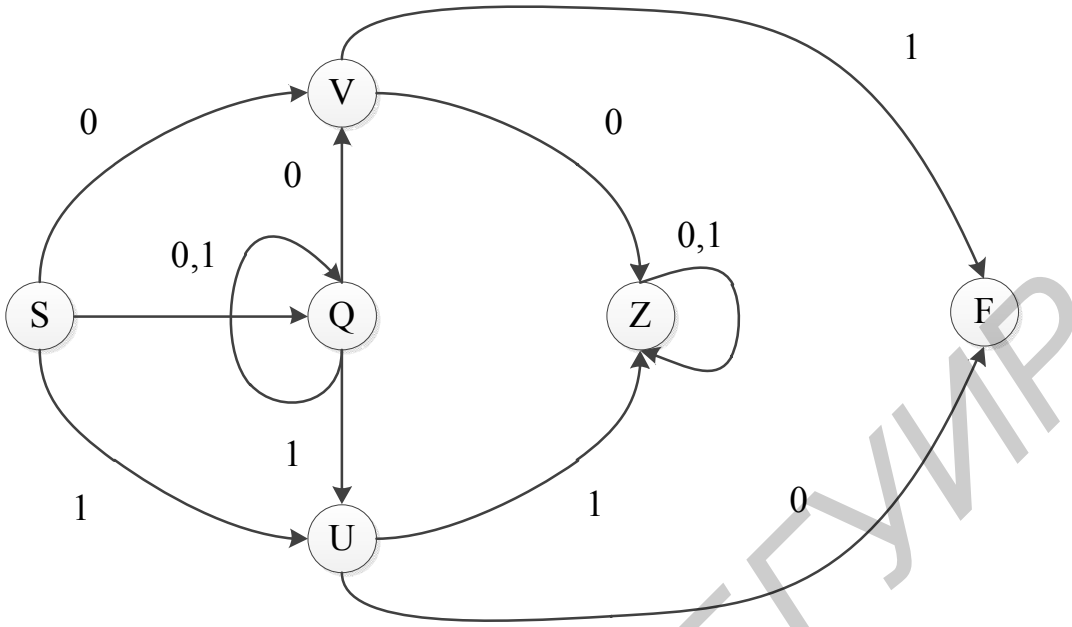


Рис. 5.2

НКА для G[Z]:
НКА = ({S, Q, V, U, Z}, {0, 1}, M, {S}, {Z})
M(S, 0) = {V, Q}
M(S, 1) = {U, Q}
M(V, 0) = {Z}
M(V, 1) = {F/∅}
M(U, 0) = {F/∅}
M(U, 1) = {Z}
M(Q, 0) = {V, Q}
M(Q, 1) = {U, Q}
M(Z, 0) = {Z}
M(Z, 1) = {Z}

Разбор цепочки t = 01001 (табл. 5.1 и рис. 5.3):

Таблица 5.1

Шаг	Текущее состояние	Остаток цепочки	Возможный приемник	Выбор
1	S	01001	V, Q	Q
2	Q	1001	U, Q	Q
3	Q	001	V, Q	V
4	V	01	Z	Z
5	Z	1	Z	Z

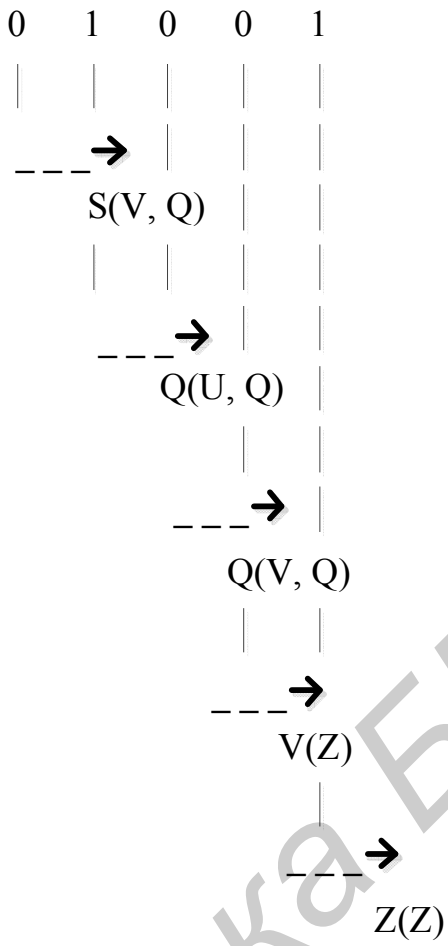


Рис. 5.3

Управляющая таблица разбора НКА (табл. 5.2):

Таблица 5.2

	S	Q	V	U	Z	F
0	V, Q	V, Q	Z	F	Z	F
1	U, Q	U, Q	F	Z	Z	F

Преобразование НКА в КА

Теорема:
Если L = L(НКА) для некоторого НКА, то существует КА, такой, что L = L(КА).
Доказательство: конструктивное; показать, что есть!

Пусть есть следующий НКА:
НКА = ({A, B}, {0, 1}, M, {A}, {B})

Таблица состояний НКА (табл. 5.3).

Таблица 5.3

НКА	A	B
0	∅	{B}
1	{A, B}	{B}

Диаграмма состояний НКА (рис. 5.4):

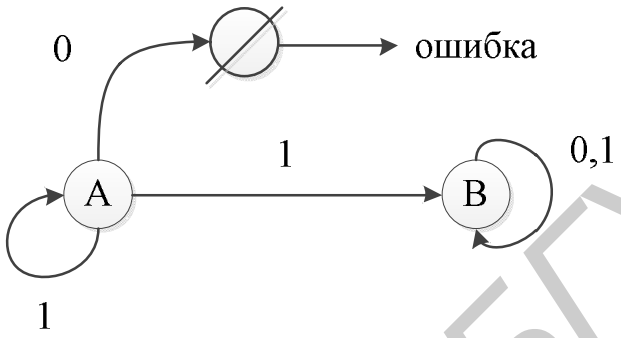


Рис. 5.4

Функции перехода:

M(A,1) = {A,B} A ::= 1A

M(B,0) = B → A ::= 1B + A::=1 (начало работы), B::=1, B::=0 (окончание работ)

M(B,1) = B B ::= 0B

B ::= 1B

Грамматика для НКА:

G1 A::=1A|1B|1

B::=0B|1B|1|0

Построим КА для данного НКА:

Существует (есть) >>KA = ({A`,B`,C`},{0,1},M,{A`},{B`}) (K_1 все подмножества из K множества).

A` - {B}

B` - {A,B}

C` - ∅

Все состояния новые.

Таблица состояний для КА (табл. 5.4).

Таблица 5.4

КА	A`	B`	C`
0	C`	B`	C`
1	B`	B`	C`

Диаграмма состояний для КА (рис. 5.5):

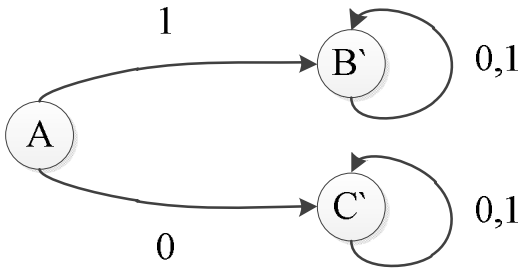


Рис. 5.5

Грамматика G2 = G1.

G2 A`::=1B`|0C`|1 (начало)

B`::=0B`|1B`|0|1 (доопределить, иначе не закончим)

C`::=0C`|1C` (конец работы)

Множество конечных состояний можно определить, как $\{B', C'\}$.

Преобразование НКА в КА.

НКА=(K,VT,M,S,Z)=>KA=(K`, VT`, M`, S`, Z`)

1. Алфавит K^1 состоит из всех подмножеств множества К.

   K=$\{S_1, S_2, \ldots, S_i\}$, элементы K`=$[S_e, \ldots, S_m]$, …, [] (конкретные состояния, упорядоченные).

2. VT(НКА) = VT(КА).
3. M`($[S_1, S_2, \ldots, S_i]$, T) = $[R_1, R_2, \ldots, R_i]$, где M($\{S_1, S_2, \ldots, S_i\}$, T) = $\{R_1, R_2, \ldots, R_j\}_i$.
4. S = $\{S_1, \ldots, S_n\}$ => S` = $[S_1, \ldots, S_n]$.
5. Z = $\{ S_1, \ldots, S_k\}$ => Z` = $[S_j, \ldots, S_k]$.

Утверждение 1:

Существует преобразование, которое строит КА из НКА.

НКА = ({S, P, Z}, {0,1}, {M(S, 0)={P}}, {S, P}, {Z}) =>

M(S, 1)={S, Z}

M(P, 0)=∅

M(P, 1)={Z}

M(Z, 0)={P}

M(Z, 1)={P}

=> КА = ({S, P, Z, SP, SZ, PZ, SPZ, ∅},{0, 1},M^1,[SP],{[7],...}}

M` = смотрите диаграмму состояний;

[SP] – начальное состояние;

{[Z], [SZ], [PZ], [SPZ]} – конечное состояние.

Состояния S, PZ – недостижимы (рис. 5.6).

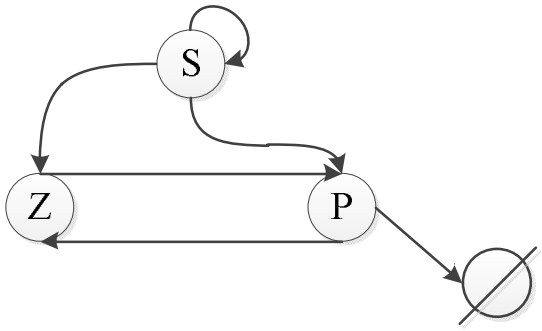


Рис. 5.6

НКА (табл. 5.5).

Таблица 5.5

НКА	S	P	Z
0	P	∅	P
1	S, Z	Z	P

КА (табл. 5.6).

Таблица 5.6

КА	S	P	Z	SP	SZ	PZ	SPZ	∅
0	P	∅	P	P	P	∅	P	∅
1	SZ	Z	P	SZ	SZP	Z	SPZ	∅

Утверждение 2:
Существует преобразование, которое строит канонический КА из неканонического. Преобразование (теорема Гриса) показано на рис. 5.6.

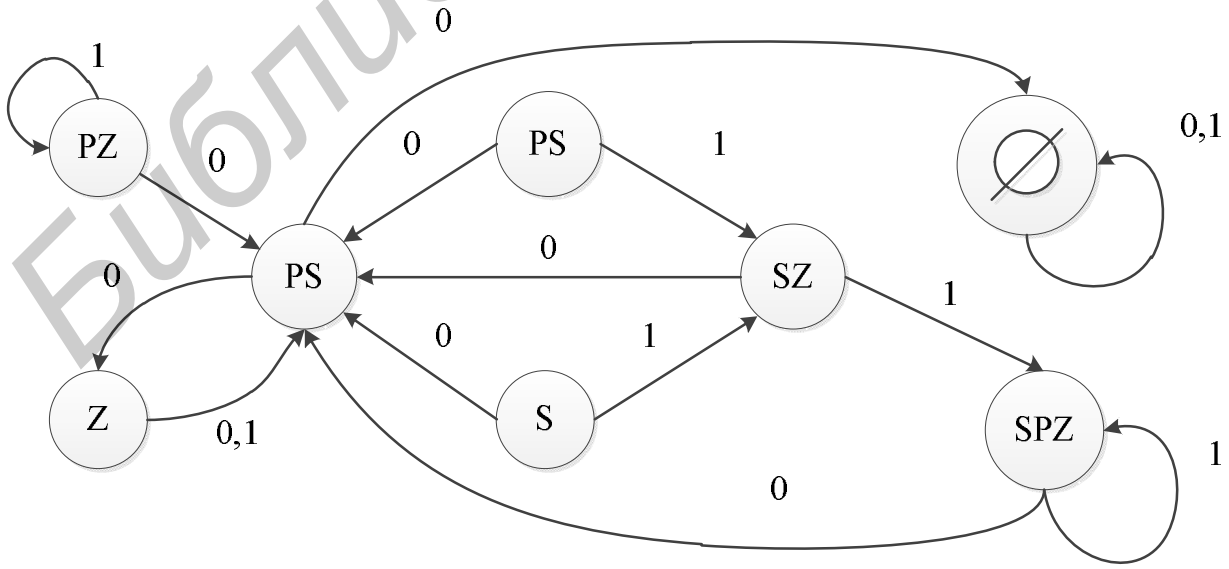


Рис. 5.6

6. Нисходящие и восходящие методы синтаксического анализа

6.1. Синтаксический анализ

Синтаксический анализ – это процесс, который определяет, принадлежит ли некоторая последовательность лексем языку, порождаемому грамматикой. В принципе, по любой грамматике можно построить синтаксический анализатор, но грамматики, используемые на практике, имеют специальную форму. Например, известно, что для любой контекстно-свободной грамматики может быть построен анализатор, сложность которого не превышает $O(n^3)$ для входной строки длины n, но в большинстве случаев по заданному языку программирования мы можем построить такую грамматику, которая позволит сконструировать и более быстрый анализатор. Анализаторы реально используемых языков обычно имеют линейную сложность; это достигается, например, за счет просмотра исходной программы слева направо с заглядыванием вперед на один терминальный символ (лексический класс).

Вход синтаксического анализатора – последовательность лексем и таблица, например, таблица внешних представлений, которые являются выходом лексического анализатора.

Обычно, однако, фаза лексического анализа не выделяется как отдельный просмотр. В этом случае лексическим анализатором управляет синтаксический анализатор, а именно, каждый раз, когда синтаксический анализатор решает, что ему необходима очередная лексема исходной программы, он вызывает процедуру, реализующую лексический анализатор.

Выход синтаксического анализатора – дерево разбора и таблицы, например, таблица идентификаторов и таблица типов, которые являются входом для следующего просмотра компилятора (например, это может быть просмотр, осуществляющий контроль типов).

Отметим, что совсем не обязательно, чтобы фазы лексичекого и синтаксического анализа выделялись в отдельные просмотры. Обычно эти фазы взаимодействуют друг с другом на одном просмотре. Основной фазой такого просмотра считается фаза синтаксического анализа, при этом синтаксический анализатор обращается к лексическому анализатору каждый раз, когда у него появляется потребность в очередном терминальном символе.

6.2. Классы синтаксических анализаторов

Большинство известных методов анализа принадлежат одному из двух классов, один из которых объединяет *нисходящие* (*top-down*) алгоритмы, а другой – *восходящие* (*bottom-up*) алгоритмы. Происхождение этих терминов связано с тем, каким образом строятся узлы синтаксического дерева: либо от корня (аксиомы грамматики) к листьям (терминальным символам), либо от листьев к корню.

Нисходящие анализаторы строят вывод, начиная от аксиомы грамматики и заканчивая цепочкой терминальных символов. С нисходящими анализаторами связаны так называемые LL-грамматики, которые обладают следующими свойствами:

– они могут быть проанализированы без возвратов;

– первая буква L означает, что мы просматриваем входную цепочку слева направо (*left-to-right scan*);

– вторая буква L означает, что строится левый вывод цепочки (*leftmost derivation*).

Популярность нисходящих анализаторов связана с тем, что эффективный нисходящий анализатор достаточно легко может быть построен вручную, например, методом рекурсивного спуска. Кроме того, LL-грамматики легко обобщаются: грамматики, не являющиеся LL-грамматиками, обычно могут быть проанализированы методом рекурсивного спуска с возвратами.

С другой стороны, восходящие анализаторы могут анализировать бóльшее количество грамматик, чем нисходящие, и поэтому именно для таких методов существуют программы, которые умеют автоматически строить анализаторы. С восходящими анализаторами связаны LR-грамматики. В этом обозначении буква L по-прежнему означает, что входная цепочка просматривается слева направо (*left-to-right scan*), а буква R означает, что строится правый вывод цепочки (*rightmost derivation)*. С помощью LR-грамматик можно определить большинство использующихся в настоящее время языков программирования.

6.3. Грамматический разбор: общие понятия

Контекстно-свободные грамматики – это такие грамматики, у которых подстановки для конкретных классов нетерминалов не зависят:

1) от окружающего контекста;
3) от последствий применявшихся операций, т. е.

$A::=W$, $A \in N$, $W \in (N \cup T)$ – выводима из S (никакие «соседи» в последовательности влияния друг на друга не оказывают).

Пример:

$G_x[S]$ S::=T|S+T
T::=<идент>|T×<идент>
<идент>::=a|b|c|…|z
=>выражения a×b + c×d + e

Наша задача – заполнить дерево синтаксического разбора, т.е. выполнить синтаксический анализ программы. Заполнение дерева разбора может выполняться как снизу вверх, так и сверху вниз (рис. 6.1).

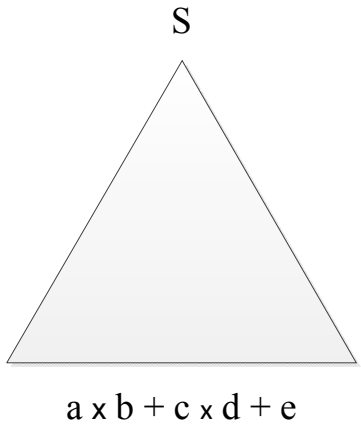


Рис. 6.1

Восходящие анализаторы соответствуют LR-грамматике:
– цепочка просматривается L (left-to-right),
– строится правый вывод цепочки.

Задача: Допустима ли цепочка a×b + c×d + e или нет (рис. 6.2).

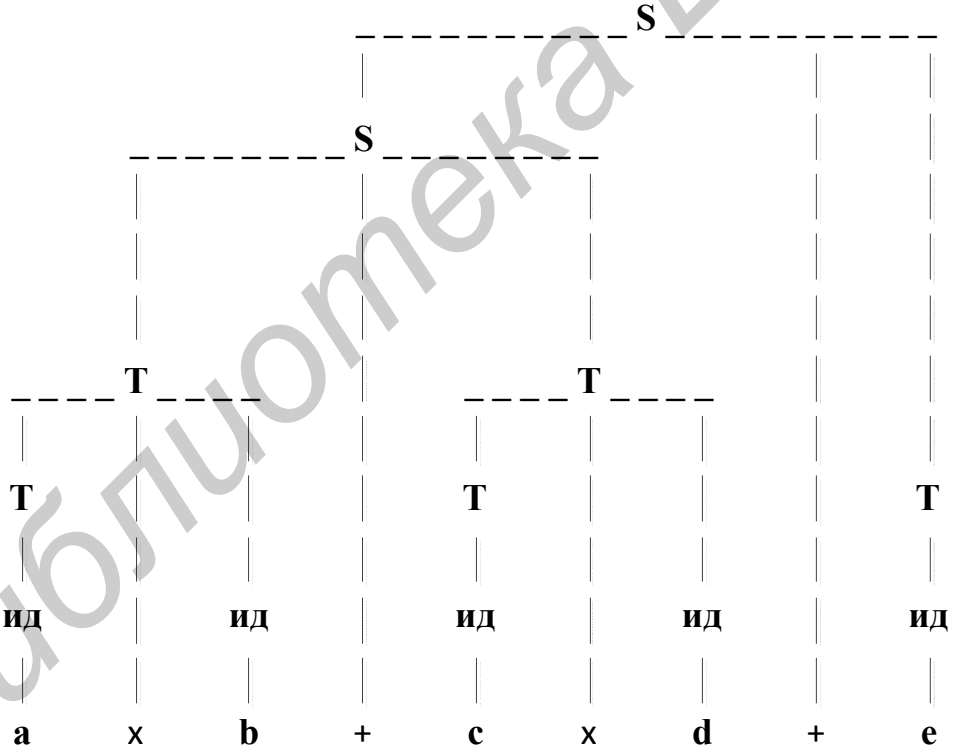


Рис. 6.2

Нужны некие (без возвратов):
1) априорные тесты (например, S ::= T | S + T | S – T|T*<идентификатор>);
2) первое слово: IF, DO.

Пример разбора сверху вниз = ааааа (рис. 6.3):
G_2[S] S ::= a | aSa.

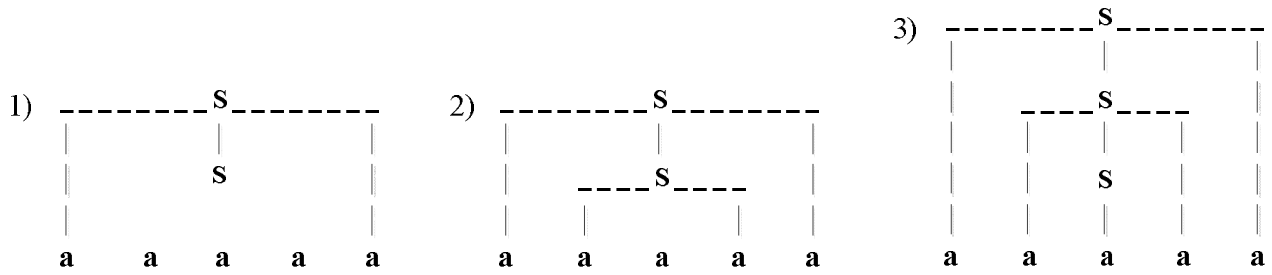


Рис. 6.3

То есть осталось одно слово => S ::= a, больше => S ::=aSa

Соответствуют LL-грамматики Нисходящие анализаторы
- анализ без возвратов;
- L – left-to-right scan;
- L – строится левый вывод цепочки (leftmost derivation).

6.4. Магазинные автоматы

6.4.1. МА для LL(K)-анализаторов

LL(K)-анализатор (рис. 6.4):
- цепочка символов следует слева направо;
- левый вывод цепочки (от аксиомы к цепочке).

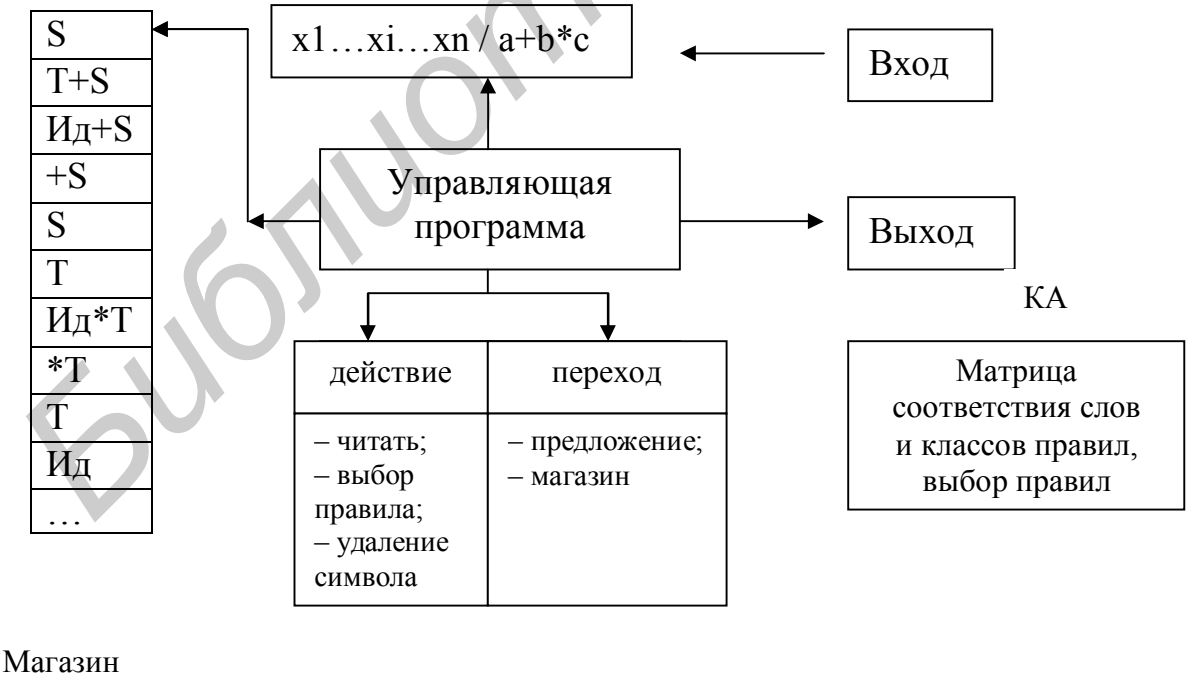


Рис. 6.4

Алгоритм анализа:

1. Начинаем с S, помещаем символ в магазин.
2. Когда первый символ в магазине, проверяем:
   - «слово» → читать предложение (соответствие установлено);
   - «правило» → выбор правила в грамматике (в магазин);
   - «stop» → цепочка пуста, в магазине первоначальный символ S.

Пример работы автомата:

S::=T|T+S
T::= ид|ид*T

Разбор цепочки a+b*c.

$$S \xrightarrow{a+} T+S \xrightarrow{} ид+S \xrightarrow{} S \xrightarrow{} T \xrightarrow{b*} ид*T \xrightarrow{e} T \xrightarrow{x\text{-пусто}} ид \xrightarrow{} \dots$$

6.4.2. МА для LR(R)-анализаторов

Магазинный автомат широко используется для создания LR(K)-анализаторов:

- цепочка символов слева направо;
- правый вывод;
- k символов для принятия решения.

Метод «перенос – свертка» (рис. 6.5)**:**

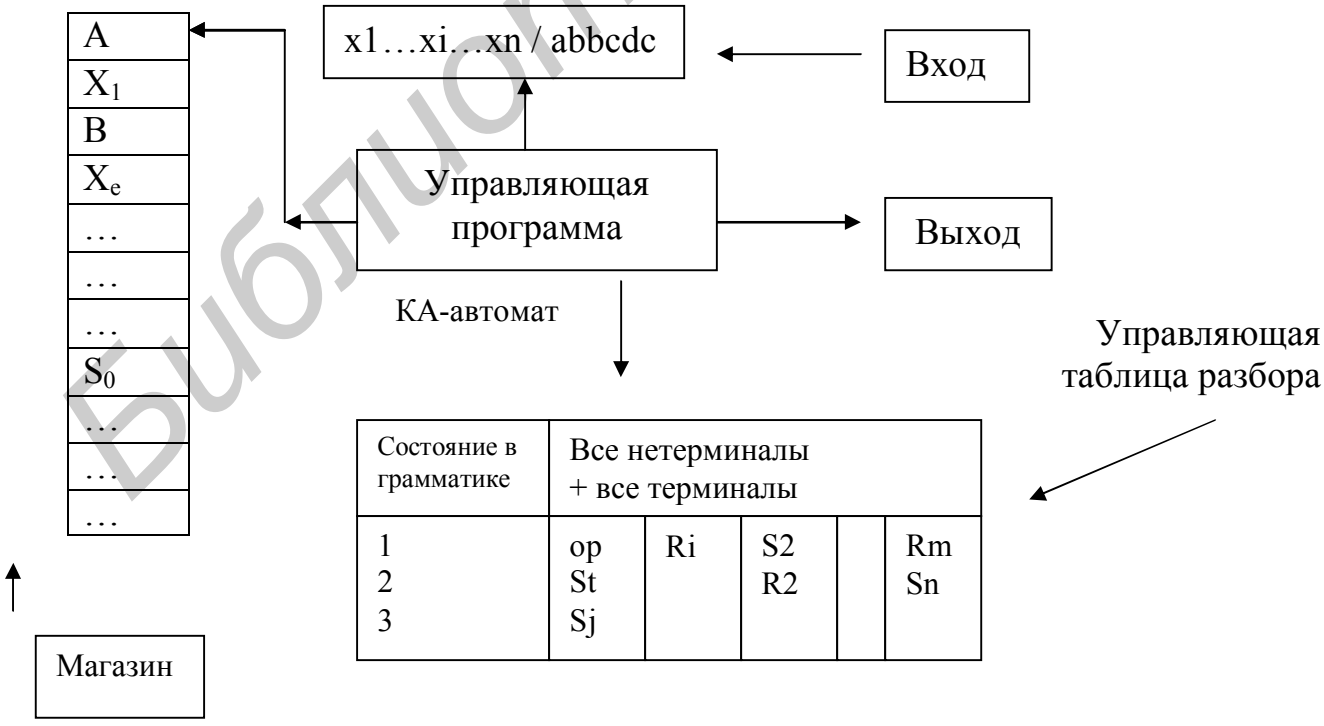


Рис. 6.5

Sj – перенос в магазин символа цепочки.
Ri – свертка по правилу i = символов.

Алгоритм анализа:

1) читать символы xj в магазин, пока не будет символа-свертки правой части правила («перенос»);

2) извлечь *i* символов из магазина («свертка») и нетерминал поместить в магазин. Свертка может быть многократной. Потом вернуться к шагу 1 или остановиться (если аксиома).

Пример:
S::=aABe
A::=Abc|b
B::=d

A=b A=Abc B=d
abbcde → aAbcde → aAde → aABe → S
(обратный порядок от цепочки к аксиоме).

6.5. Алгоритм разбора сверху вниз

Суть алгоритма:

1) построим дерево разбора или разбор сверху вниз:
   а) сверху вниз;
   б) слева направо;
2) начинаем с первоначального символа и символа предложения, последовательно производим подстановку классов (правил), пытаясь добиться соответствия по всему предложению;
3) алгоритм разбора:
   а) начинаем с S ≡ Варианты;
   б) если первый символ *слово = предложение;*
   в) если класс (правило), то ≡ Выбор.

Пример разбора цепочки a+b*c для грамматики (табл. 6.1):

S ::= T | T+ S
T ::= ид | ид × T

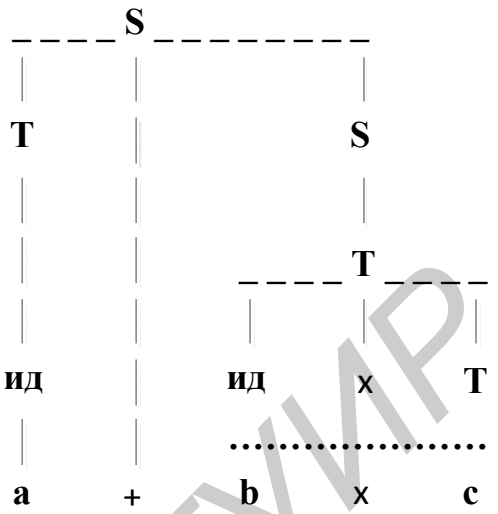
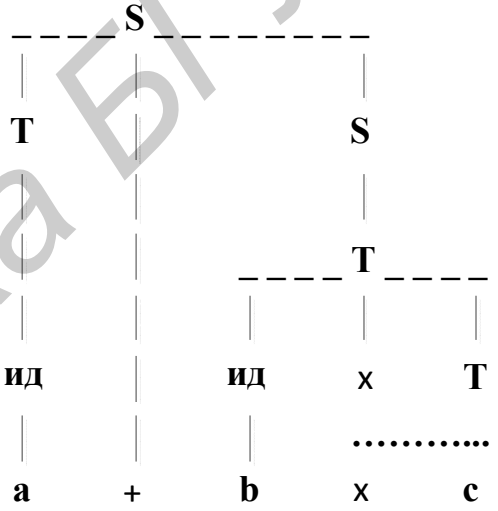
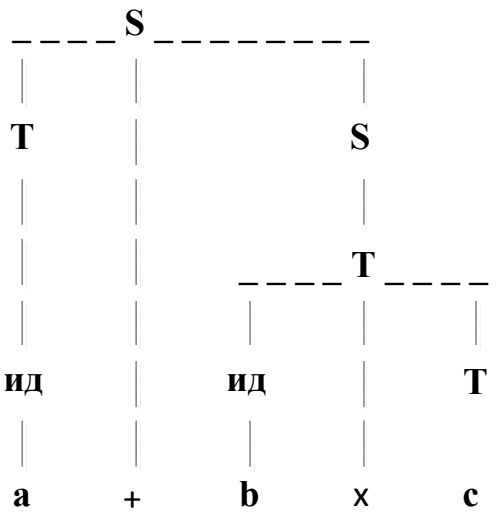
Таблица 6.1

Шаги	**Предложение**	**Правило в памяти; магазин**	**Дерево**
1	2	3	4
1	a + b×c	S Правило	S …….. a + b×c !Замена слева направо и в правилах

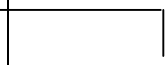
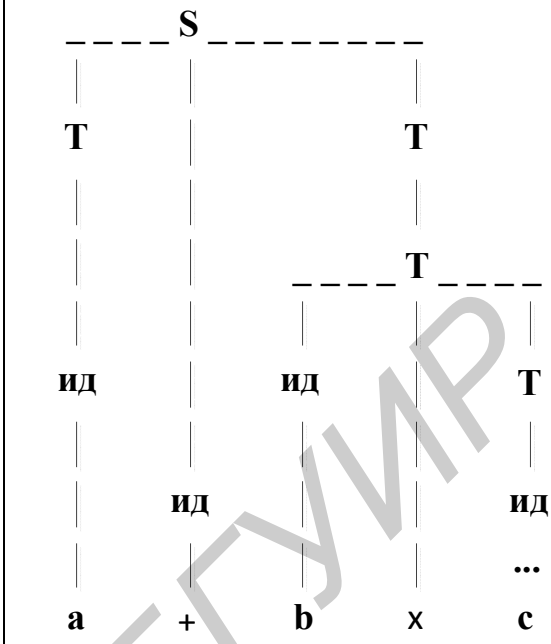
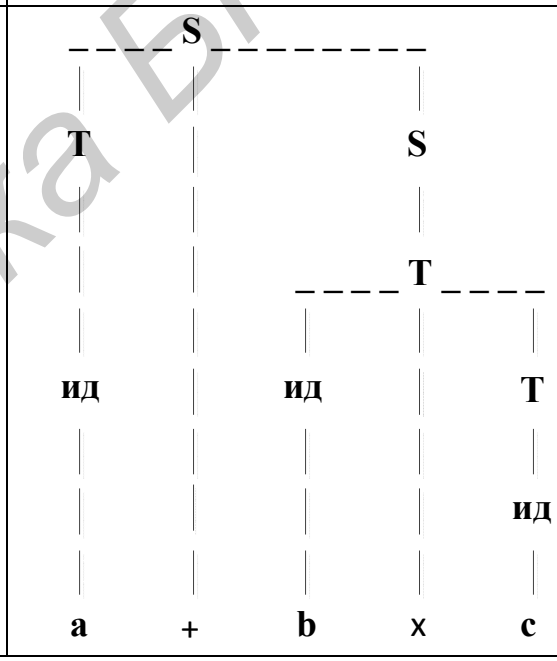
Продолжение табл. 6.1

1	2	3	4
2	a + b×c	T + S Правило	**T** **T** + **S** …… a + b×c
3	a + b×c ← Ид + S		**T** **T** + **S** **ид**
4	+ b×c ← + S	Слово	**T** **T** + **S** **ид** …… a + b×c
5	b×c	S Правило	**T** **T** + **S** **ид** …… a + b x c
6	b×c	T Правило	**S** **T** + **S** **ид** **T** ….. a + b x c

Продолжение табл. 6.1

1	2	3	4
7	b×c	ид × T Слово	
8	×c	×T Слово	
9	C	T Правило	

Окончание табл. 6.1

1	2	3	4
10			
11			

6.6. Перебор вариантов

Алгоритм перебора вариантов:

1) сопоставляется просмотренная часть предложения с остатком предложения → остаток структуры (правил);
2) определяется направляющий символ правила.

Пример:

G[S] S ::= T | T + S
T ::= ид | ид × T, разобрать предложение a + b*c (табл. 6.2).

Таблица 6.2

Цепочка	Магазин (по КА см. п. 6.4.1, делается выбор)
a + b×c	S
a + b×c	1. T 2. T + S
a + b×c	Ид ид+S ид ×T ид ×T+S
+ b×c	– +S – направляющий символ правила + ×T ×T + S
b×c	S
b×c	T T+S
b×c	Ид ид ×T ид + S ид ×T+S
×c	– ×T – направляющий символ правила * +S ×T+S
c	T T+S
c	Ид ид ×T ид + S ид ×T+S
–	– ×T +S ×T+S

Оба остатка пусты, т. е. :

1) разбор закончен;
2) предложение $\in$ G[S].

6.7. Априорные критерии для разбора сверху вниз

Для грамматики G[S]

S ::= T | S+T

T ::= ид | T × ид.

При разборе возможно зацикливание (рекурсия).

Для такой G[S] возможно зацикливание сверху вниз.

$$(S),\ \begin{cases} T \\ S+T' \end{cases} \begin{cases} \text{ид} \\ T \times \text{ид} \\ T+T \\ S+T+T \end{cases} \Rightarrow \text{рекурсия слева } T \to T\alpha$$

То есть предыдущий метод «перебор вариантов» не будет работать (прямой рекурсии не должно быть).

6.8. Преобразование к нормальной форме LL(1)

Все правила начинаются с фраз (слов).

В грамматике G[S1] есть рекурсия.

$$\left.\begin{array}{l} S ::= T \mid T+S \\ T ::= \text{ид} \mid \text{ид} \times T \end{array}\right| \Rightarrow \left.\begin{array}{l} S ::= \text{ид} \mid \text{ид} \times T \mid \text{ид} + S \mid \text{ид} \times T + S \\ T ::= \text{ид} \mid \text{ид} \times T \end{array}\right| \Rightarrow \text{нет рекурсии слева.}$$

Алгоритм для перебора вариантов

1. Правая часть начинается с символа (терминала) и определяет правило.
2. Если в левой части > 1 одинакового правила, то справа должен быть разный нетерминал, который определяет правило.
3. Заглядываем вперед и по символу в правиле (* или +) и символу в цепочке определяем правило.

6.9. Матрица предшественников

Если *b* – текущее слово, то замена на правила начинается с *b*<...>, где *b* – направляющее правило => матрица соответствия (слов и классов (правил)), предписывающая какие правила могут начинаться и с каких слов.

Алгоритм матрицы предшественников

1. Определяется матрица со строками для каждого нетерминала и столбцами для каждого нетерминала и терминала.
2. Указываются слова и классы (правила), с которых могут начинаться правила (табл. 6.3).

S ::= a | Tb
T ::= c | Td

Таблица 6.3

Слово	S	T	a	b	c	d	Правило
S	0	1	1	0	0	0	=> S начинается с T или a
T	0	1	0	0	1	0	=> T начинается с T или c

3. Для любой строки добавить OR (или) строки, которые представлены для данного правила (для S OR T => S, т.к. S ::= Tb (табл. 6.4).

Таблица 6.4

Слово	S	T	a	b	c	d	Правило
S	0	1	1	0	1	0	=> замыкание (свертка)
T	0	1	0	0	1	0	

6.10. Метод рекурсивного спуска

Метод рекурсивного спуска – один из наиболее быстрых методов:
а) не имеет универсального анализатора;
б) работает прямо по правилам.

Суть метода:

Представление каждого нетерминального символа (S, T, …) в виде процедуры (без параметров), распознающей в тексте цепочки языка L(S), L(T), L(X), … Тело процедуры строится на основе правил. ➔ А так как характер правил рекурсивный ➔ + сверху вниз ➔ рекурсивный спуск.

Пример:

Пусть есть правило КС – нормализованные грамматики, сверху вниз:
$P ::= t_1T_1 \mid \ldots \mid t_nT_n$

Тогда процедура P имеет следующий вид:

```
Procedure P
… // общие переменные
        call Read_Lex(lex, n_lex)
        select(n_lex):
                when (‘t1’) call T1 …;
                when (‘t2’) call T2 …;
                …
                when (‘tn’) call Tn …;
                otherwise call Error1()
        end;
…
end;
```

Синтаксис и семантика языка X1:

1. <метка>:<оператор>
2. <идентификатор>:=<выражение>
3. DCL<имя типа><тип>
4. <тип>:=fixed-bin | char(n) (но тогда нужны операции под строками)

5. <выражение>-<см. ниже>

{ <выражение>:=<терм>{<on +/-><терм>}
<терм>:=<множ.>{ <on +/- ><множ.>}
<on +/->:= +/-
<множ.>:= <идентификатор>/(<выражение>)
<on +/->:= */ ÷ }

6. DO <итерация> … END
7. Procedure <имя>[<параметры>] … END
8. Goto <метка>
9. Call <имя процедуры> [<аргументы>]
10. IF <выражение-отношение> THEN <оператор> [ELSE <оператор>]
11. Select

Ниже представлен проект программ на псевдоязыке синтаксического анализа методом рекурсивного спуска языка X1.

```
DCL (lex1, lex2) CHAR(20) VAR, (n_lex1, n_lex2) FIXED B IN (15);
    Procedure Statement;
    …
    do while (not eof(input));
        call Read_lex(lex1)(n_lex1);
        select(n_lex1);      /* все операторы */
            when (<идентификатор >)
                call Read_lex(lex2) (n_lex2);
                select(n_lex2):
                    when(':') /* здесь знак, правильно № лексемы*/
                        call Label(lex1, lex2);
                    when(':=')
                        call Move_Stmt(lex1, lex2)
                    otherwise call Error1(n_err, lex1, lex2, n_line);
                end; /* метка, := */
            when (<кл.-слово>)
                select(n_lex1)
                    when ('Procedure') call Proc_Stmt(lex1);
                    when ('do') call Do_Stmt(lex1);
                    when ('GoTo') call Proc_Goto(lex1);
                    when ('call') call Call_Stmt(lex1);
                    when ('if' OR 'else') call If_Stmt(lex1);
/*согласовать, проверить вложенность */
                    when ('else') call else_Stmt(lex1);
                    otherwise call Error1(n_err, lex1, lex2, n_line);
                end;
            otherwise call Error1(n_err, lex1, lex2, n_line);
        end; /* select */
```

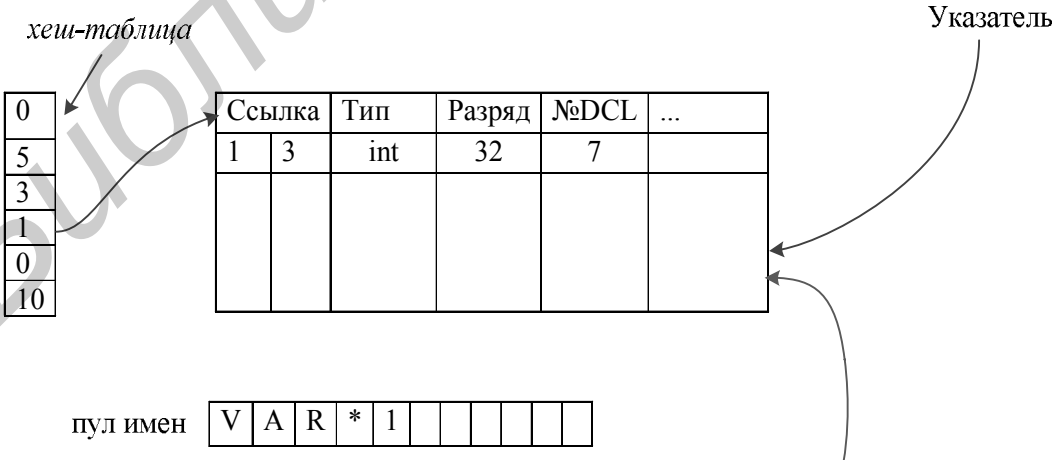
```
        end; /* do */
Procedure Label(Lex1, Lex2);
/* проверить описание метки по таблице меток */
/* проверить правильное определение метки */
/* имя-метки(контекст) где описана (№ опер.GOTO */
/* не описанные, неиспользуемые, неправильные переходы */
        IF <есть ошибки>
        THEN
                DO;
                        Обработать ошибки;
                        RETURN;
                END;
        ELSE
                DO;
                        Занести метку и ее контекст в таблицу меток;
                END;
        RETURN;
END;

Procedure move_stmt(Lex1, Lex2);
/*Проверить описание идентификатора*/
/* идентификация: определяющее использующее*/
/* по таблице имен, таблице типов */
/* таблице идентификаторов*/
/* Задача: Связать использующее=определяющее*/
```

1. Таблица типов

|имя|тип|разряд|

2. Таблица имен



3. Таблица идентификаторов

хеш	имя	класс лексемы	ссылка

```
CALL VARIABLE(TEXT);
        IF <есть ошибки>
        THEN
                DO;
                        Обработать ошибки;
                        RETURN;
                END;
        CALL EXPRES;
        /* выдать обратную польскую запись и определить тип выражения*/
        /* проверить тип переменной и тип выражения*/
        CALL VARIABLE(TEXT);
        /* контроль типов:    { статический,
                               { динамический */
        /* эквивалентность:   { структурная (от базы)
                               { поименная  */
END;

Procedure EXPRES
/* нет обработки конца выражения (;) */
        CALL TERM;      /* умножение, деление */
        DO WHILE (Lex1 != '+', != '-')
                CALL        Read_Lex(Lex1);
                SELECT (Lex1);
                        WHEN ('+' OR '-')
                                CALL Read_Lex(Lex1);
                                CALL TERM; /* */ */
                        OTHERWISE
                                CALL ERROR1(N_ERR);
                                RETURN;
                END;
        END;
END;
```

Типы:

1. Примитивные(базовые) – integer, char, float, boolean, …
2. Конструкторы типов – array, struct, pointer, record, procedure

Примеры (Паскаль):

```
var Mart: array[1..10,1..20] of Integer;
type Addr = record index: integer;
struct: array [1..20] of char end;
var p:^massiv;
```

```
        function fx(a1, a1: char): ^ integer;
        /*представление типов: в виде деревьев или последовательности битов из
        базовых через конструкторы*/

Procedure TERM;
        …
        CALL Multip;
        DO WHILE (Lex1 != ‘*’ OR Lex1 != ‘/’);
                CALL Read_Lex(Lex1);
                SELECT (Lex1)
                        WHEN (‘*’ OR ‘/’)
                                CALL Read_Lex(Lex1);
                                CALL Multip;
                        OTHERWISE
                                CALL ERROR1(N_7);
                                RETURN;
                END /* select;
        END /* do;
        …
END;
Procedure Multip;
        IF Lex1 = <идентификатор>
        THEN DO
                CALL VARIABLE(Lex1);
                RETURN;
                END;
        ELSE DO
                IF Lex1 = ‘(’
                THEN DO
                                CALL Read_Lex(Lex1);
                                CALL EXPRES;
                                CALL Read_Lex(Lex1);
                                IF Lex1 = ‘)’
                                THEN
                                        RETURN;
                                ELSE DO
                                        CALL ERROR1(N_8);
                                        RETURN;
                                        END;
                        END;
                END;
END;
```

```
Procedure  IF_stmt;
        …
        CALL EXPRES_EQV;
        CALL        Read_Lex(Lex1);
        IF Lex1 != ‘THEN’
        THEN DO
                CALL ERROR1(N_9);
                RETURN;
                END;
```

Важно – возможны варианты:

1. Либо RETURN, признак обработки IF (возможен ELSE), проверка по стеку вложенности управляющих операторов (см. п. 10).

Proc	Function	Do	Do While	If	Select
.	-	-	-	-	-
.	-	-	-	then	-
.	-	-	-	else	-
End	End	End	End	Endif	End

2. Либо CALL Statement, но возможна сильная рекурсия.

```
END;

Procedure EXPRES_EQV;
        …
        DO WHILE (Lex1 != ‘THEN’)
                CALL Read_Lex(Lex1);
                CALL EXPRES;
                CALL Read_Lex(Lex1);
                IF Lex1 != <,>,>=,<=,<>
                THEN DO
                        CALL ERROR1(N_10,<пропуск>);
                        END;
                ELSE DO
                        CALL Read_Lex(Lex1);
                        CALL EXPRES;
                        END;
        END;
END;
```

Procedure Proc_Stmt(Lex1)

1. Дерево статической вложенности процедур (для анализа правильности вызовов).
2. Стек статической вложенности процедур (для определения области действия имен).
3. Имя процедуры – тип = таблица процедур.
4. Читать параметры.

END;

Procedure Proc_Goto(Lex1)

1. Таблица меток

Есть ли там ? – занести метку;

– занести № Goto;

2. Проверить контекст метки.

END;

Таблица процедур (табл. 6.5).

Таблица 6.5

Имя ссылки	N1	N2	№ по порядку	№ уровня вложенности	Параметры

7. Грамматический разбор снизу вверх

7.1. Восходящие анализаторы

Восходящий анализатор (bottom-up parsing) предназначен для построения дерева разбора, начиная с листьев и двигаясь вверх к корню дерева разбора. Этот процесс можно представить как «свертку» исходной строки *w* к аксиоме грамматики. Каждый шаг свертки заключается в сопоставлении некоторой подстроки *w* и правой части какого-то правила грамматики и замене этой подстроки на нетерминал, являющийся левой частью правила. Если на каждом шаге подстрока выбирается правильно, то в результате получим правый вывод строки *w*.

Пример:

Рассмотрим грамматику

$S \rightarrow aABe$

$A \rightarrow Abc$

$A \rightarrow b$

$B \rightarrow d$

Цепочка *abbcde* может быть свернута в аксиому следующим образом: *abbcde, aAbcde, aAde, aABe, S.*

Фактически эта последовательность представляет собой правый вывод этой цепочки, рассматриваемый справа налево:

S→aABe→aAde→aAbcde→abbcde.

5 4 3 2 1

Таким образом:

1) исходный объект – все предложения.
2) общая задача – свести исходный объект с помощью правил подстановок в обратном порядке к первоначальному символу.

Пример:

G(S)

S ::= T|T+S;

T ::= <идентификатор>|<идентификатор>*T;

Цепочка: a + b * c

Разбор (рис. 7.1 – 7.4):

1. T ::= ид (да)
   T ::= ид * T (нет)

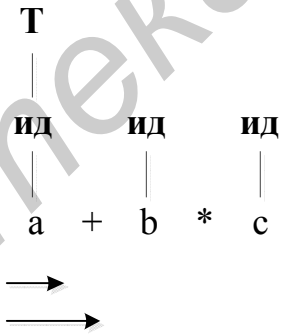


Рис. 7.1

2. S ::= T (нет)
   S ::= T + S (да)

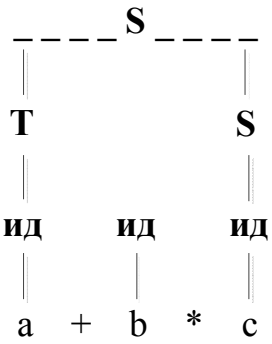


Рис. 7.2

3. T ::= ид (нет)
   T ::= ид +T (да)

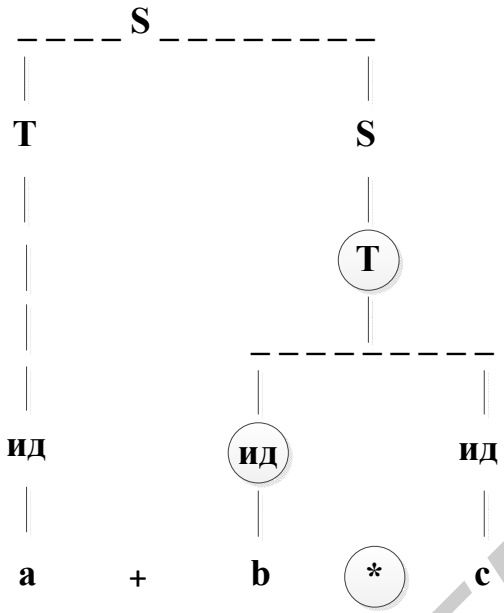


Рис. 7.3

4. T ::= ид (да)
   T ::= ид*T (нет)

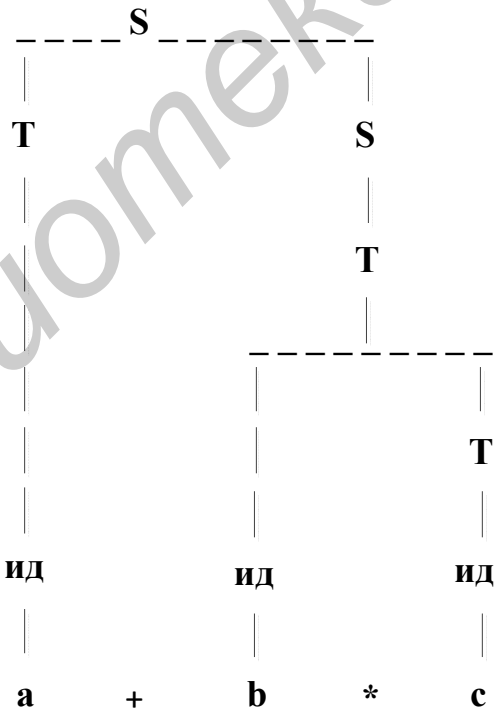


Рис. 7.4

7.2. Основа для методов разбора снизу вверх

Задача (уточнение).

Дано:

1. Грамматика G_x (S).
2. Предложение αβγ.

Получить:

Редуцировать αβγ S $\xrightarrow[G_x]{}$ на основе G_x.

Как всегда, предполагаем, что правила из G_x определяются однозначно.

Алгоритм:

1. Найти основу V в αβγ.
2. Найти правило U ::= v…
3. Привести в v в αβγ к ...U... в αβγ(замена) и т. д.

Введение, общее для предшествия:

– операторного;

– простого.

1. Предложение $\alpha(G_x)$ в состоянии разбора (свертки) $İ_1$, $İ_2$ … $İ_n$; W_j, W_{i+1} … $W_ı$, где $İ_k$ – терминал или нетерминал (слово/класс), W_h – терминал(слово).

2. Если правило свертки определяется по интервалу $İ_1$ … $İ_n$; W_j, … W_{j+t-1}, то язык LR(t). Для нас обычно достаточно LR(1).

3. Если правило подстановки определяется по (m, s) элементам $İ_1$, $İ_2$ … $İ_n$; W_j, W_{i+1} … $W_ı$, то язык LR(m,s) , у нас будет LR(1,1).

|------m--------| |----------s----------|

Основа определяется на стыке $İ_n$ и W_j.

Два метода разбора:

– операторное предшествование;

– простое предшествование, отношение между элементами грамматики.

Любое промежуточное состояние разбора имеет вид

$S_1, S_2, \dots, S_{j-1}, S_j, S_{j+1}, S_{j+2}, \dots, S_i, S_{i+1}$

$<\ <\ <\ <\ =\ =\ =\ =\ >$

7.3. МА и LR(k)-анализатор для методов разбора снизу вверх

LR(k)-анализ означает, что

– входная цепочка обрабатывается слева направо (left-to-right parse);

– выполняется правый вывод (rightmost derivation);

– не более *k* символов цепочки (k-token lookahead) используется для принятия решения.

При LR(k)-анализе применяется *метод «перенос-свертка» (shift-reduce)*. Этот метод использует магазинный автомат. Суть метода сводится к следующему. Символы входной цепочки переносятся в магазин до тех пор, пока на вершине магазина не накопится цепочка, совпадающая с правой частью какого-нибудь из правил (операция «перенос», «*shift*»). Далее все символы этой цепочки извлекаются из магазина и на их место помещается нетерминал, находящийся в левой части этого правила (операция «свертка», «*reduce*»). Входная цепочка допускается автоматом, если после переноса в автомат последнего символа входной цепочки и выполнения операции свертки в магазине окажется только аксиома грамматики.

Анализатор состоит из входной цепочки, выхода, магазина, управляющей программы и таблицы, которая имеет две части (действие и переход). Схема такого анализатора выглядит следующим образом (рис. 7.5):

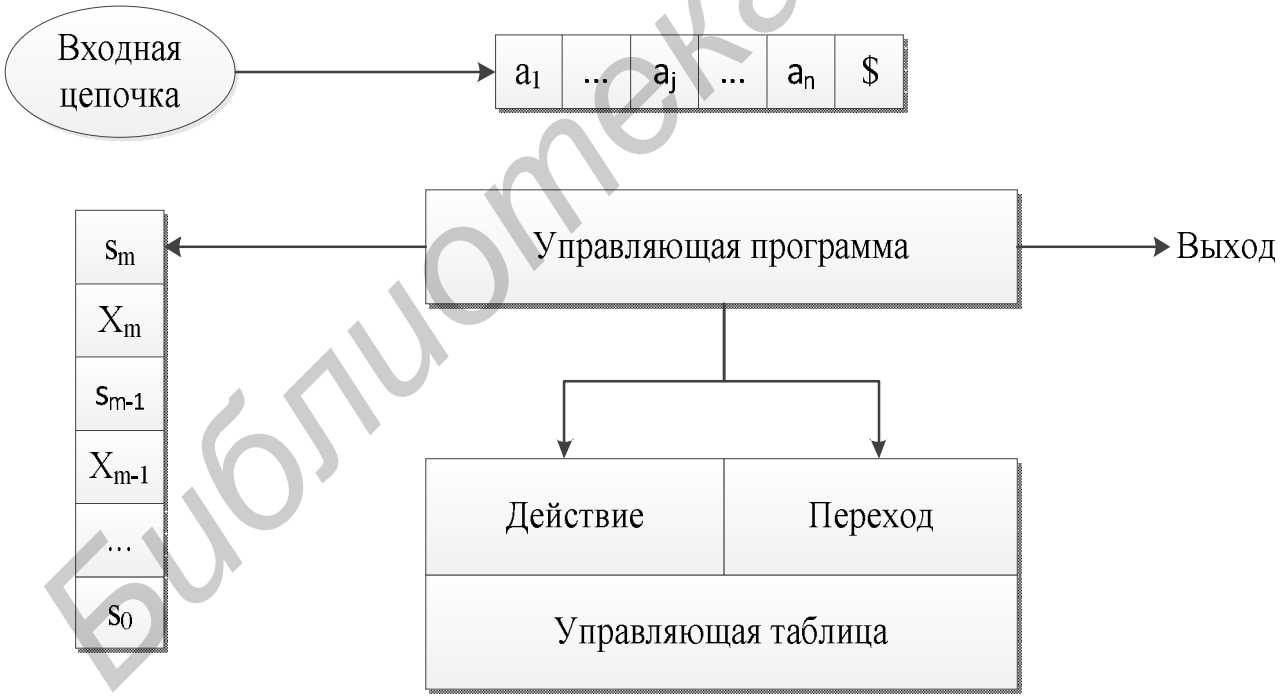


Рис. 7.5

7.3.1. Управляющая программа анализатора

Управляющая программа одинакова для всех LR-анализаторов, а таблица изменяется от одного анализатора к другому. Программа анализатора читает последовательно символы входной цепочки. Программа использует магазин для запоминания строки вида $s_0X_1s_1X_2\ldots X_ms_m$, где s_m – вершина магазина. Каждый X_i – символ грамматики, а s_i – символ, называемый состоянием. Каждое состояние суммирует информацию, содержащуюся в стеке перед ним. Комбинация символа состояния на вершине магазина и текущего входного символа используется для индексирования управляющей таблицы и определения операции переноса–свертки. При реализации грамматические символы не обязательно располагаются в магазине; однако мы будем использовать их при обсуждении для лучшего понимания поведения LR-анализатора.

Программа, управляющая LR-анализатором, ведет себя следующим образом. Рассматривается пара: s_m – текущее состояние на вершине магазина, a_i – текущий входной символ; после этого вычисляется action $[s_m, a_i]$, которое может иметь одно из четырех значений:

1) shift *s*, где *s* – состояние;
2) свертка по правилу $\alpha \rightarrow \beta$;
3) допуск (*accept*);
4) ошибка.

Функция **goto** получает состояние и символ грамматики и выдает состояние. Функция goto, строящаяся по грамматике G, есть функция переходов детерминированного магазинного автомата, который распознает язык, порождаемый грамматикой G.

7.4. Грамматики с операторным предшествованием

Операторной грамматикой $G_{оп}$ (S) называется грамматика, у которой:

– нет правил вида A := αBCβ, где α, β – любые последовательности, возможно пустые;

–определены отношения между терминалами (словами) грамматики.

Отношения между терминалами (p, q):

1. p = q, т. е. терминалы входят в одно правило, выводятся одновременно, имеют одинаковые значения предшествования.
   A := αpqβ, A := αpBqβ, где α, β – любые последовательности, возможно пустые.

2. p > q; p выводится раньше q.
   A := αBqβ и есть вывод B := γp или B := γpC.

3. p < q, p выводится позже q.
   A := αpBβ и есть вывод B := qγ или B := Cqγ.

Пример:

Грамматика:
S := T | S + T

T := P | T * P

P := ид | (S)

Цепочка: a + b * (c + d)

Цепочка с предшествованием
(см. табл. 7.1):
a + b * (c + d)
> < = > < < = > < = >

Управляющая таблица
(матрица предшествования)

Таблица 7.1

	+	*	(	)	ид
+	>	<	<	>	<
*	>	>	<	>	<
(	<	<	<	=	<
)	>	>	=	>	
ид	>	>		>	

Примеры подвыражений:
(a), a+(в+(a)), +...+,+(,+...),+ид,*(,)+,)*,(...),)),*...,ид+, ид*, ид)

Алгоритм разбора:

Шаг 1. Формируется магазинный список последовательно из İ слов и правил (классов).

I => $İ_1$, $İ_2$, … , $İ_n$ – просмотрено; W_j, W_{j+1}, W_h – остаток предложения.

Шаг 2. Если последнее слово p в İ, т. е. p = Wj или p < Wj => İ = İ +Wj, то – в магазин.

Если (=, <, >) не определено, то – ошибка.

Шаг 3. Если p > Wj, в İ выбирается максимальное количество слов, удовлетворяющих: İa < İb = Ic = … = p – основа.

Для всех слов есть некоторая подстановка в грамматике, поэтому выполнить свертку по правилу U::= İb Ic…P.

Свертка:

Шаг 1. İ от İa до конца магазина.

Шаг 2. Wj не удаляется из последовательности.

Шаг 3. Переход к шагу 1.

Шаг 4. Если предложение закончилось и магазин İ – пуст (т. е. только S), то все хорошо, иначе – ошибка.

В процессе свертки из рассмотрения исключаются классы (правила), учитываются отношения между словами => операторная грамматика.

Пример:

S := T | S + T
T := P | T * P
P := ид | (S)

Состояние магазина для разбора цепочки a + b*(c + d) (чтение (ч.), свертка (с.)):
ч.a +
с. S +
ч. S + b*
с. S + T*
ч. S + T * (c +
с. S + T * (S +
ч. S + T * (S + d)
с. S + T * (S + T)
с. S + T * (S)
с. S + T * P
с. S + T

7.5. Грамматика с простым предшествованием

Устанавливаются отношения предшествования между элементами грамматики P и S_1, терминальными и нетерминальными (как в подразд. 7.4). Выводятся одновременно:

1. $P = S_1$, если есть правило U := … PS_1 … .

Предшествования:

2. $P > S_1$, P – часть основы, а S – нет, т. е. P – предшествует S_1.
   Например, U:=...P, а так как предложение разбора ...P S_1..., следует вывод: S_1 позже P.
3. $P < S_1$, U := S …, а P – нет, т. е. S предшествует.

Другие отношения не определены.

Пример:

Gx(Z) Z := bMb
M := L|a
L:= Ma

Цепочки:
bab, b(aa)b, b((aa)a)b.

Управляющая таблица (матрица предшествования) – все элементы грамматики и их отношения (табл. 7.2):

Таблица 7.2

	Z	**b**	**M**	**L**	**a**	**(**	**)**
Z							
b			=		<	<	
M		=			=		
L		>			>		
a		<			>		=
(			<	=	<	<	
)		>			>		

Разбор цепочки b(aa)b (табл. 7.3):

Таблица 7.3

Шаг		Основа	Приведение	Вывод
1	b (a a) b < < > = >	a	M	b(aa)b=>b(Ma)b
2	b (a a) b < < > = >	Ma)	L	b(Ma)b=>b(Lb
3	b (a a) b < < > = >	(L	M	b(Lb=>bMb
4	b (a a) b < < > = >	bMb	Z	bMb=>Z

7.5.1. Алгоритм разбора

Шаг 1. Установить отношения в управляющей таблице (матрице предшествования).

1 – Si < Sj

2 – Si = Sj

3 – Si > Sj

Шаг 2. Подготовить магазин и индексы S(1) = #, i = 1 (магазин), k = 1 (№ символа).

Шаг 3. Сканировать след символ ➔ R, k = k + 1.

Шаг 4. Если S(i) не > R (т. е. <, =), то i = i + 1, S(i) = R – к шагу 2.

Шаг 5. S(i) > R, вся основа в стеке ➔ найти начало основы в стеке

S(j) … S(i) > R

S(j – 1) < S(j)

Шаг 6. Найти среди правил грамматики правую часть S(j-1)…S(j). Если есть правило – к шагу 9.

Шаг 7. Иначе – к шагу 8.

Шаг 8. Если S(i) = Z, R = #, i = 1 => все хорошо – конец, иначе – ошибка.

Шаг 9. Выполнить свертку в стеке S(j) = Q, где Q := U, i = j, далее к шагу 4.

Пример:

Z := bMb

M := L|a

L := Ma

b(aa)b (табл. 7.4)

Таблица 7.4

Шаг	S(1), S(2)…	Отношение	Предложение
1	2	3	4
1	#	<	b(aa)b
2	#b	<	(aa)b
3	#b(	<	aa)b
4	#b(a	>	a)b
5	#b(M	=	a)b
6	#b(Ma	=	)b
7	#b(Ma)	>	b
8	#b(L	>	b

Окончание табл. 7.4

1	2	3	4
9	#bM	=	b
	#bMb	>	#
	#Z	?	#

7.6. LR-таблицы разбора и алгоритм анализа цепочек

Основные элементы LR-таблиц разбора и алгоритм анализа цепочек приведены ниже.

Грамматика G[s]: (*все правила должны быть пронумерованы*)
а) правила

1) S::=real IDLIST;
2) IDLIST::=IDLIST, ID;
3) IDLIST::=ID;
4) IDLIST::=A|B|C|D…

б) стеки (или магазин)

1) стек символов из грамматик;
2) стек номеров состояний (1, 2,…, 7) из таблицы разбора;

в) таблица разбора (управляющая таблица) представлена в табл. 7.5.
В магазинном автомате – управляющая таблица, главный элемент.
Построение:
Любой терминал, любой нетерминал и пустой символ $\perp$ – столбцы таблицы.
Состояние, в котором находится анализатор: S_i, R_j;

Таблица 7.5

Состояние в грамматике	**S**	**IDLIST**	**ID**	**Real**	**,**	**A, B, C, D**	**$\perp$ – знак окончания строк**
	1	**2**	**3**	**4**	**5**	**6**	**7**
1	HALT			S2			
2		S5	S4			S3	
3					R4		R4
4					R3		R3
5					S6		R1
6			S7			S3	
7					R2		R2

г) элементы таблицы разбора и алгоритм работы анализатора.

1. Элементы сдвига Si (S7):
   – поместить в стек символов символ столбца;
   – поместить в стек состояний состояние i (их 7);

– перейти к состоянию i (их 7);

– если входной символ – терминал, то принять его.

2. Элементы приведения Rj (в нашем случае правила грамматики R1,…,R4):

– выполнить приведения с помощью правил j (допустив, что n есть число элементов в правой части правила);

– удалить n элементов из стека символов и стека состояний;

– перейти к состоянию, указанному в верхней части стека состояний;

– нетерминал (в левой части правила 4 (S, ID, IDLIST)) => следующий входной символ следует считать следующим входным символом.

3. Элементы ошибок = пусто (здесь, в данной ситуации).

4. Элементы остановки – конец разбора.

Пример:

Разобрать цепочку символов – real A, B, C ⊥ (табл. 7.6).

Чтение символов переключается в:

1) входную цепочку;
2) по приведенному правилу – в магазин.

Таблица 7.6

Цепочка символов	**Входной символ**	**Стек символов (магазин)**	**Стек состояния**	**Таблица разбора**
real A, B, C ⊥ ↑	real	пусто	1	(1, real) => S2
real A, B, C ⊥ ↑	A	real	2 1	(2, A) => S3
real A , B, C ⊥ ↑	ID(,)	A real	3 2 1	(3, ,) => R4 (переключение: – приведение – по правилу 4 =>)
real A , B, C ⊥ ↑	ID	real	2 1	(2, ID) => S4 (вход ID – состояние = 2) Переключение на цепочку
real A , B, C ⊥ ↑	,	ID real	4 2 1	(4, ,) => R3 (переключение на грамматику, приведение)
real A , B, C ⊥ ↑	IDLIST	real	2 1	(2, IDLIST) => S5 (переключение)
real A , B, C ⊥ ↑	,	IDLIST real	5 2 1	(5, ,) => S6
real A, B, C ⊥ ↑	B	, IDLIST real	6 5 2 1	(6, B) => S3

Продолжение табл. 7.6

Цепочка символов	Входной символ	Стек символов (магазин)	Стек состояния	Таблица разбора
real A, B , C ⊥ ↑	ID (,)	B , IDLIST real	3 6 5 2 1	(3, ,) => R4
real A, B , C ⊥ ↑	ID	, IDLIST real	6 5 2 1	(6, ID) => S7
real A, B , C ⊥ ↑	,	ID , IDLIST real	7 6 5 2 1	(7, ,) => R2 в правой части правила 3 символа
real A, B , C ⊥ ↑	IDLIST	real	2 1	(2, IDLIST) => S5
real A, B , C ⊥ ↑	,	IDLIST real	5 2 1	(5, ,) => S6
real A, B, C ⊥ ↑	C	, IDLIST real	6 5 2 1	(6, C) => S3
real A, B, C ⊥ ↑	⊥	C , IDLIST real	3 6 5 2 1	(3, ⊥) => R4
real A, B, C ⊥ ↑	ID	, IDLIST real	6 5 2 1	(6, ID) => S7
real A, B, C ⊥ ↑	⊥	ID , IDLIST real	7 6 5 2 1	(7, ⊥) => R2

Окончание табл. 7.6

Цепочка символов	Входной символ	Стек символов (магазин)	Стек состояния	Таблица разбора
real A, B, C ⊥ ↑	IDLIST	real	2 1	(2, IDLIST) => S5
real A, B, C ⊥ ↑	⊥	IDLIST real	5 2 1	(5, ⊥) => R1
real A, B, C ⊥ ↑	S		1	(1, S) => HALT

7.6.1. Построение LR-таблиц разбора

Основные понятия:

1) разбор (распознавании правил разбора) – конфигурация разбора.

Конфигурация в правилах (позиция в правилах) => показывает продвижение в разборе правила;

2) состояние в таблице разбора ≈ соответствует конфигурации «минус» неразличимые конфигурации, без неразличимых конфигураций;

3) строим замыкания, начиная с (1, 0), до тех пор пока все конфигурации не попадут в какое-то состояние (переходы по всей грамматике).

Рассмотрим построение LR-таблиц разбора на конкретном примере грамматики:

Шаг 1. Определяем состояния, конфигурацию разбора в грамматике.

(1) S ::= real IDLIST
(1,0) (1,1) (1,2)
(2) IDLIST ::= IDLIST , ID
(2,0) (2,1) (2,2) (2,3)
(3) IDLIST ::= ID
(3,0) (3,1)
(4) ID ::= A | B | C | D
(4,0) (4,1)

Шаг 2. Выполняем преобразование грамматики и определяем состояние в разборе. Переход из одного состояния в другое возможен вводом символа терминал | нетерминал (табл. 7.7).

Таблица 7.7

Состояние (конфигурация) разбора	База из грамматики	Замыкание (множество конфигураций)	Преобразованная грамматика и ее конфигурация
1	(1,0)	{(1,0)}	(1) $S::=_1real_2IDLIST_5$
2	(1,1)	{(1,1),(2,0),(3,0),(4,0)}	(2) $IDLIST::=_2IDLIST_{5,6}ID_7$
3	(4,1)	{(4,1)}	(3) $IDLIST::=_2ID_4$
4	(3,1)	{(3,1)}	(4) $ID::=_{(2,6)}A\|B\|C\|D_3$
5	{(1,2)(2,1)}	{(1,2),(2,1)}	
6	(2,2)	{(2,2),(4,0)}	
7	(2,3)	{(2,3)}	

Таким образом, для одной базы соответствует > 1 конфигурации (1,2)(2,1).

Число состояний в анализаторе равно числу множеств неразличных конфигураций в грамматике (табл. 7.8):

Таблица 7.8

Состояние	S	IDLIST	ID	real	,	ABCD	⊥
1				S2			
2		S5	S4			S3	
3							
4							
5					S6		
6			S7			S3	
7							

Шаг 3. Строим LR-таблицу разбора. Заполняем элементами сдвига (S).

Правило (заполнения):

– из состояния $_1$real→2;

– из 2 или IDLIST… и т. д.

Получено состояние без приведения. Затем выполняется процесс дополнения элементами свертки (R)

В целом задача зависит от контекста и предварительно просматриваемого символа:

1) приведение возможно => окончание правил состояний 3, 4, 5, 7;

2) можно элементы Ri внести во все столбцы, но раннего анализа не будет, и для состояния 5 уже есть S6 (=> предварительно просмотреть символ(ы));

3) из правил (1)(2) => это только ⊥, а приведение только ⊥ => для состояния S→R1→ ⊥ (табл. 7.9)

Таблица 7.9

Состояние	S	IDLIST	ID	real	,	ABCD	⊥	R_i
1	HALT			S2				
2		S5	S4			S3		
3					R4		R4	←R4
4					R3		R3	←R3
5					S6		R1	←R1/(S6)
6			S7			S3		
7					R2		R2	←R2

8. Представление грамматики в памяти

Замечания для LL(1).

1. Нет прямой левосторонней рекурсии, косвенная может быть:
   T ::= Tα, но U ::= Vα
   V ::= Uβ | x => U ::= Uαβ.
2. Правила вида
   U ::= xy|xz|xα преобразуются к виду U ::= x(y|z|α).
3. Для правил вида
   U ::= x|y|…|z (множество направляющих символов порождающих правил непересекающееся) все множество выводимых символов должно быть различным.

x => *Au, y => *Bv, то $A \neq B$ (направляющие символы различны).

Методы представления грамматик в памяти рассмотрим на примере грамматики.

S ::= a | bTTd
T ::= bc | a

1. Скобочная запись грамматики. Правила в виде скобок, терминалы в правилах:

((a), (b, ((b, c), (a)), ((b, c), (a)), d))
  S T T

2. Список: узлы – правила, лист – терминал (рис. 8.1).

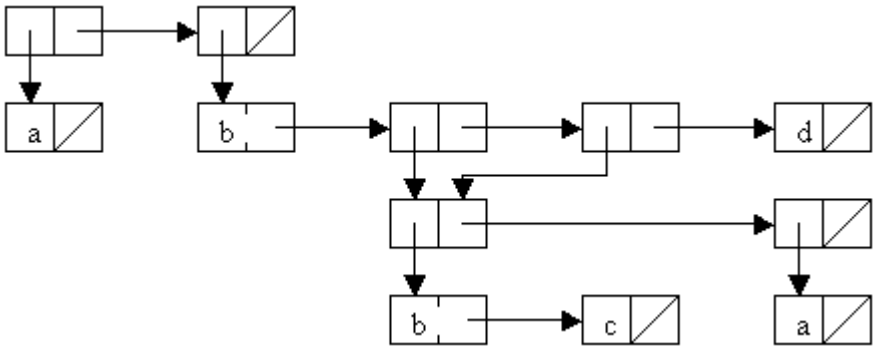


Рис. 8.1

3. Метод Гриса.

Все элементы грамматик представляются в виде дерева.

Рассмотрим пример для грамматики типа «оператор присваивания» (рис. 8.2).

G[ОП] <оператор-присваивания> ::= <переменная> ::= <выражение>
<переменная> ::= <идентификатор>
<выражение> ::= <терм> {<оп +-><терм>} ! итерация
<терм> ::= <множит> {<оп */><множит>}
<оп +-> ::= <+|->
<множит> ::= <идентификатор> | <выражение>
<оп*/> ::= * | /

Пример 1

<условный-оператор> ::= IF <отношение> THEN <оператор> ELSE <оператор>
<отношение> ::= [(] <выражение-отношение> [)] [{!|&} [(] <выражение-отношение> [)]]…
<выражение-отношение> ::= <выражение> <оператор-отношение> <выражение>
<оператор-отношение> ::= < | > | >= | <= | = | <>

Пример 2

Лог.-выр. ::= [¬] лог.-множ. [![¬] лог.-множ.]
лог.-множ. ::= лог.-терм. [& лог.-терм.]
лог.-терм. ::= лог.-конст. | лог.-терм. | [(] отношение [)]
отношение ::= выражение-оператор |отношение-выражение.

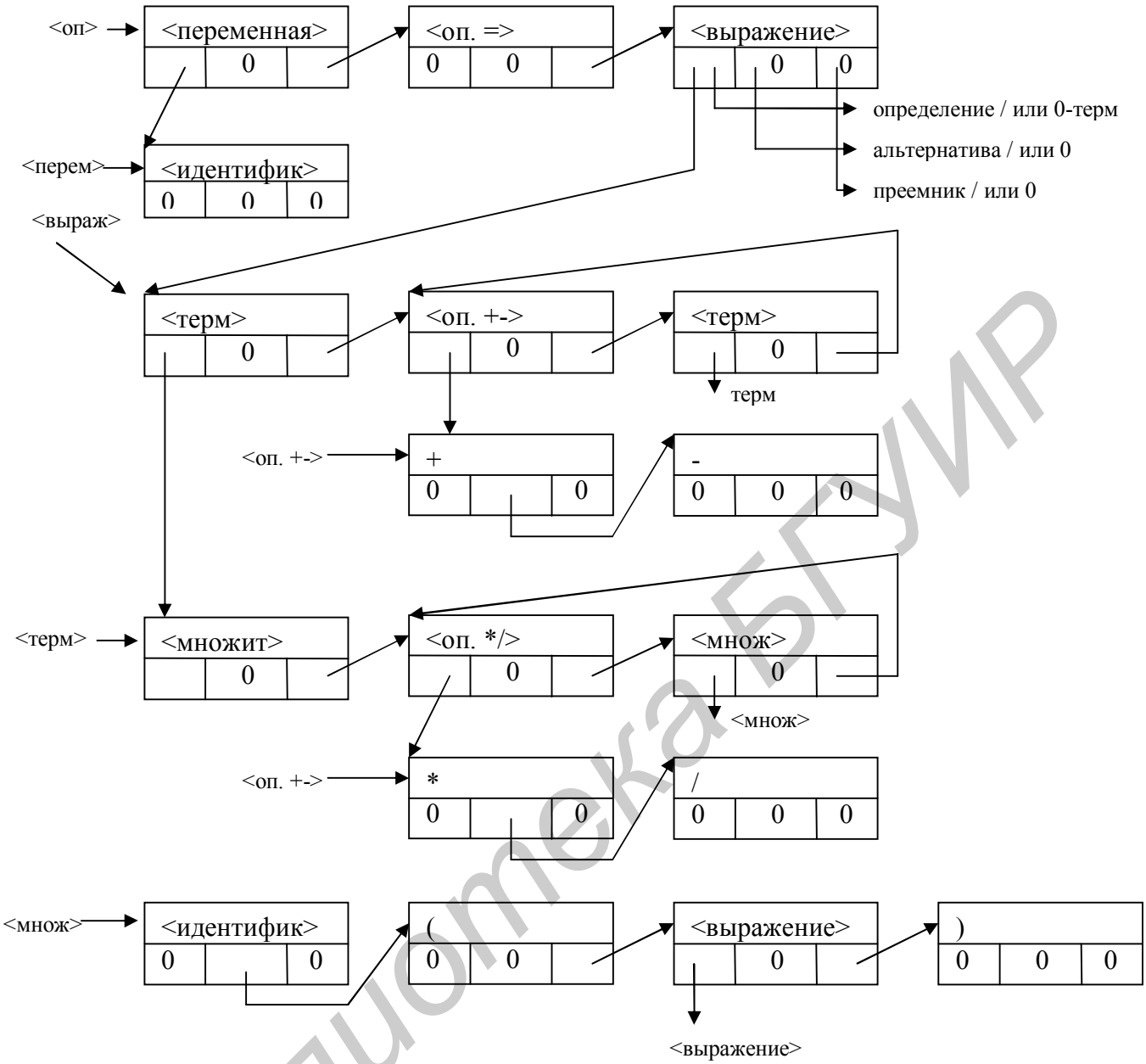


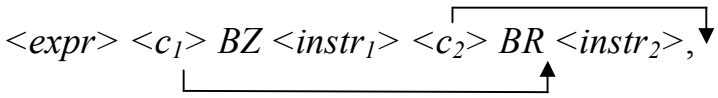
Рис. 8.2

9. Польская запись

Польская запись была предложена польским математиком Лукашевичем. В этой форме записи все операторы непосредственно предшествуют операндам. Так, обычное выражение $(a+b)*(c–d)$ в польской записи может быть представлено как $*+ab–cd$.

Такую форму записи называют также префиксной. Аналогичным образом вводится *обратная*, или *постфиксная польская запись*, в которой все операторы выписываются после операндов. Так, вышеприведенный пример, в

обратной польской записи будет записан следующим образом: *ab+cd–**. Для представления унарных операций в польской записи можно воспользоваться эквивалентными выражениями, использующими бинарные операции, как в следующем примере: *–b → 0 – b*, а можно ввести новый знак операции, например, *@b*. Польская запись может быть распространена не только на арифметические выражения, но и на прочие конструкции языка. Например, оператор *a := a + b;* может быть записан в польской записи как *:=a+ab*, *ab+a:=*, а условный оператор **if** <expr> **then** <instr$_1$> **else** <instr$_2$> может быть записан как следующая последовательность операторов:



где c_1 указывает на первую инструкцию <instr$_2$>, а c_2 – на первую инструкцию, следующую за <instr$_2$>, BR – безусловный переход на адрес <c_2>, а BZ – переход на <c_1> при условии равенства нулю выражения <expr>.

Пользуясь такой терминологией, мы можем называть традиционную форму записи выражений *инфиксной*, т. к. в ней знаки операций расположены между операндами. Понятно, что любое выражение может быть переведено из инфиксной формы в польскую запись и наоборот. Польская запись замечательна тем, что при ее использовании исчезает потребность в приоритетах операций – каждая операция выполняется в порядке появления в исходной цепочке (хотя очевидно, что приоритет операций необходимо учитывать при преобразованиях из инфиксной формы).

Польская запись (особенно обратная) очень хорошо накладывается на стековую модель: каждый встреченный операнд загружается в стек, а операции производятся только на вершине стека: каждая операция снимает необходимое количество операндов с вершины стека и кладет на стек свой результат.

9.1. Алгоритм перевода в постфиксную запись

1. Инфиксная запись a+b.
2. Префиксная +ab.
3. Постфиксная ab+ (рис. 9.1).

(x – y) + z → xy – z +
x – (y + z) → xyz + –
x*(y + z)*w → xyz + * w *
(a + b)*(c + d) → ab + cd + *
a + b*c + d → abc* + d +
(a + b)*c + d → ab + c*d +

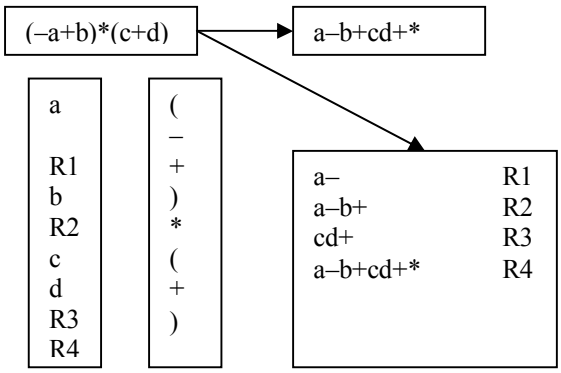


Рис. 9.1

Здесь
B1 (a + b)*(c + d) – входная строка,
B2 a – b + cd + * – выходная строка,
CO + – стек операций + стек операндов,
Rx – промежуточный результат.

Алгоритм перевода инфиксного выражения в постфиксную запись:

Пусть Lск = 0.
1. Читать лексему,
   а) если идентификатор, то записать его в B2;
   б) если знак операции, то в стек;
   в) если скобка (, то пропустить, отметить, что была левая Lск = Lск + 1;
   г) если скобка), то Lск = Lск – 1 (признак конца подвыражения).
2. Конец подвыражения (см. синтаксис, грамматику, признаки, операции * /,) записать знак операции из вершины стека → O1.
3. Конец выражения, освободить стек операций → O2 => выход.

9.2. Алгоритм перевода, проверки и вычисления

B1 – входная строка (–a + b) + (c + d).
B2 – выходная строка a – b + cd + *.
Rx – промежуточный результат и его тип (табл. 9.1).

Таблица 9.1

O1 (стек операндов)			**Тип**	**O2 (стек операций)**		
A	R1	R2	0111	-	+	*
B			0101			

Пусть Lск = 0.

Шаг 1. Читать лексему. Если конец выражения – Шаг 6, иначе – Шаг 2.
Шаг 2. Если операнд, то – в стек О1, О2 => Шаг 1.
Шаг 3. Если «(», то Lск = Lск + 1 => Шаг 1.
Шаг 4. Если «)», проверить уровень вложенности скобок, Lск = Lск – 1 => Шаг 1.
Шаг 5. Если знак операции – проверить стеки О1, О2:

а) по таблице;
б) по старшинству операций (приоритет);
в) по грамматике (конец подвыражения).

Возможно, выполнить предыдущую операцию. Если:

а) ДА, проверить типы операндов, выполнить операцию и результат поместить на место 1-го операнда, операцию → В2:

– стек операндов не пуст:
  - одноместная операция (раньше операнда);
  - двуместная операция (несколько подряд).
– скорректировать стеки, текущую операцию в стек О2;

б) НЕТ, текущую операцию → стек О2, перейти к шагу 1.

Шаг 6. Проверить стеки – выполнить, если нужно операции, + → В2:

– О2 – пуст, Lск = 0 \
→ все хорошо
– О1 – один результат, + его тип /
– иначе – ошибка (табл. 9.2, 9.3).

Стек операндов О1:

Таблица 9.2

Тип	**Стек операндов**		
01011	a	R1	R2
01101	b	C	R3
	d		

Стек операций О2:

Таблица 9.3

Тип	**Стек операций**	
-	+	*
+		

Приоритет операций (табл. 9.4)
B1 = (–a + b) + (c + d)
B2 = a – b + cd + *

Для разбора выражений необходима рекурсивная универсальная программа, которая учитывает приоритеты операций (табл. 9.4).

Таблица 9.4

Тип операций	Обозначение	Приоритет
Унарный	┐ NOT ~ –	6
Двухместный	*/	5
Двухместный	+-	4
Отношения	>, <, =, <>	3
	>=, <=	2
Логические	&, |	1

1) выражения – лексема;
2) сдвиг-свертка – лексема.

Пример разбора 1:

B1: вход ((a+b)-c)*d
B2: выход ab+c-d*
R1c-d*
R2d*
R3

a + b > c – d ab + cd ➔
ab +
R1
R1cd –
R1R2 >
+ >

Пример разбора 2:

Выражения отношения, логические и одноместные операции:

~a+~b>c*d&l= ¬m
B2
a~b~+cd*>lm¬ =&

B1 ~a+~b>c*~d&l= ¬m

O1 a
R1b
R1R2
R3c ?
O2 ~
+ ~
> * ~
& = ¬
B2 a~b~+cd~*>lm¬=&

10. Семантика

Фаза контроля типов решает и проверяет, удовлетворяет ли программа контекстным условиям. Главной составляющей контекстных условий является «правильное использование» программой типов данных, предоставляемых входным языком, т. е. корректность выражений, встречающихся в программе, с точки зрения использования типов. Данная задача включает нахождение объявления в программе каждого используемого идентификатора, и проверку корректности его появления в использующем контексте.

10.1. Идентификация

Идентификация идентификаторов – одна из задач, решение которой необходимо для проверки правильности использования типов.

Понятно, что невозможно убедиться в правильности использования типов в какой-нибудь конструкции до тех пор, пока не определим типы всех ее составных частей. Например, для того, чтобы выяснить правильность оператора присваивания, мы должны знать типы его получателя (левой части) и источника (правой части). Для того, чтобы выяснить, каков тип идентификатора, являющегося, например, получателем присваивания, надо понять, каким образом этот идентификатор был объявлен в программе.

Каждое вхождение идентификатора в программу является либо определяющим, либо использующим. Под определяющим вхождением идентификатора понимается его вхождение в описание, например, int i. Все остальные вхождения являются использующими, например, i = 5 или i+13.

Цель идентификации идентификаторов – определить тип использующего вхождения идентификатора. Эта задача может быть полностью или частично решена в фазе синтаксического анализа. Все зависит от того, может ли использующее вхождение идентификатора встретиться в программе до определяющего вхождения, или нет. Если все определяющие вхождения идентификаторов должны быть расположены текстуально перед использующими вхождениями, то можно выполнить идентификацию на фазе синтаксического анализа. Если же нет, то в фазе синтаксического анализа мы можем обработать определяющие вхождения идентификаторов и только на следующем просмотре текста программы выполнить собственно идентификацию.

Вне зависимости от того, на каком просмотре будет выполняться идентификация идентификаторов, при обработке определяющего вхождения идентификатора необходимо запомнить информацию о типе этого идентификатора. Это можно сделать несколькими путями:

– создать узел в синтаксическом дереве для конструкции «описание идентификатора» и запоминать информацию о типе идентификатора в этом узле;

– создать *таблицу идентификаторов* (IdTab) и в ней запоминать информацию о типе идентификатора. Почему нам может потребоваться новая таблица? Понятно, что если транслируемая программа имеет блочную структуру, и/или язык допускает создание и использование *перегруженных* идентификаторов (overloading), то в *таблице представлений* (таблица представлений сопоставляет некоторому используемому в компиляторе обозначению идентификатора его представление в программе) информацию о типе идентификатора хранить нельзя, поскольку в этой таблице каждая лексема встречается только один раз. Таким образом, нам потребуется новая таблица для хранения информации об определяющих вхождениях идентификаторов.

В п. 10.4.2 приведена таблица имен, которая служит для сбора информации о типах идентификаторов и контроля за правильностью их использования.

10.2. Контроль типов

Если контроль типов осуществляется во время трансляции программы, то мы говорим о *статическом контроле типов*, в противном случае, т. е. если контроль типов производится во время исполнения объектной программы, мы говорим о *динамическом контроле типов*. В принципе, контроль типов всегда может выполняться динамически, если в объектном коде вместе со значением будет размещаться и тип этого значения. Понятно, что динамический контроль типов приводит к увеличению размера и времени исполнения объектной программы и уменьшению ее надежности. Язык программирования называется *языком со статическим контролем типов* или *строго типизированным языком* (strongly typed language), если тип любого выражения может быть определен во время трансляции, т. е. если можно гарантировать, что объектная программа выполняется без типовых ошибок. К числу строго типизированных языков относится, например, Паскаль. Однако даже для такого языка как Паскаль, некоторые проверки могут быть выполнены только динамически.

Например:

```
table: array [0..255] of char;
i: integer;
```

Компилятор не может гарантировать, что при исполнении конструкции table[i] значение i действительно будет не меньше нуля и не больше 255. В некоторых ситуациях осуществить такую проверку может помочь техника, подобная *анализу потока данных (data flow analysis)*, но далеко не всегда. Понятно, что на самом деле этот пример демонстрирует ситуацию, общую для большинства языков программирования, т. е. здесь речь идет о контроле индексов вырезки. Конечно, почти всегда такая проверка выполняется динамически. Для контроля типов используется сущность таблицы имен (идентификаторов) см. п. 10.4.2.

10.3. Эквивалентность типов

Необходимой частью контроля типов является проверка *эквивалентности типов (equivalence of types)*. Крайне необходимо, чтобы компилятор выполнял проверку эквивалентности типов быстро.

Структурная эквивалентность типов (Structural equivalence of types). Два типа называются эквивалентными если они являются одинаковыми примитивными типами, либо они были сконструированы с применением одного и того же конструктора к структурно эквивалентным типам. Иными словами, два типа структурно эквивалентны тогда и только тогда, когда они идентичны.

В некоторых языках типам можно давать имена, которые иногда называют индикантами типа. Рассмотрим пример программы на языке Паскаль:

```
type link = ^cell;
var next: link;
    last: link;
    p: ^cell;
    q, r: ^cell;
```

Возникает вопрос, одинаковые ли типы имеют описанные переменные? К сожалению, ответ зависит от реализации, поскольку в определении языка Pascal не определено понятие «идентичные типы». В принципе здесь возможны две ситуации. Одна из них свзана со структурной эквивалентностью типов. С этой точки зрения все объявленные переменные имеют одинаковый тип.

Второй подход связан с понятием *эквивалентности имен* (name equivalence). В этом случае каждое имя типа рассматривается как уникальный тип, таким образом, два имени типов эквивалентны, если они идентичны. При таком подходе переменные p, q, r имеют одинаковый тип, а переменные «p» и «next» – нет. Обе эти концепции используются в различных языках программирования. Например, Алгол 68 поддерживает структурную эквивалентность.

Проблемы, возникающие в Паскале, связаны с тем, что многие реализации связывают с каждым определяемым идентификатором неявное имя типа. Таким образом, приведенные объявления некоторыми реализациями могут трактоваться следующим образом.

```
type link = ^cell;
    np = ^cell;
    npg = ^cell;
var next: link;
    last: link;
    p: np;
q, r: npq
```

10.4. Таблицы для контроля семантики

Различные таблицы и сущности, построенные на их основании, являются основными элементами транслятора для сбора информации о программе и для контроля семантики. В данном подразделе рассмотрены: таблица имен, таблица меток, таблица процедур, структура вложенности процедур, граф вызовов процедур, стек управляющих операторов, стек и дерево вложенности процедур, таблица ключевых слов и их структура.

10.4.1. Таблица имен

Содержимое:

– имя или идентификатор, его атрибуты, номера операторов (где описаны, где используются, где перевычисляются);

– список внешних переменных, попроцедурные списки;

– агрегаты (массивы[размерность], структуры[вложенность].

Таблица имен служит для контроля типов имен (идентификаторов) при трансляции.

Структура (рис. 10.1).

Структуры или записи могут храниться в пуле имен в виде шаблона: STR1(E1, E2, STR2(E3,E4)).

При обработке описания вся информация о имени и его атрибутах (типах) попадает в таблицу имен. При обработке вхождения определяются все атрибуты имени и выполняется контроль типов. Предполагается, что описание имени должно предшествовать его использованию.

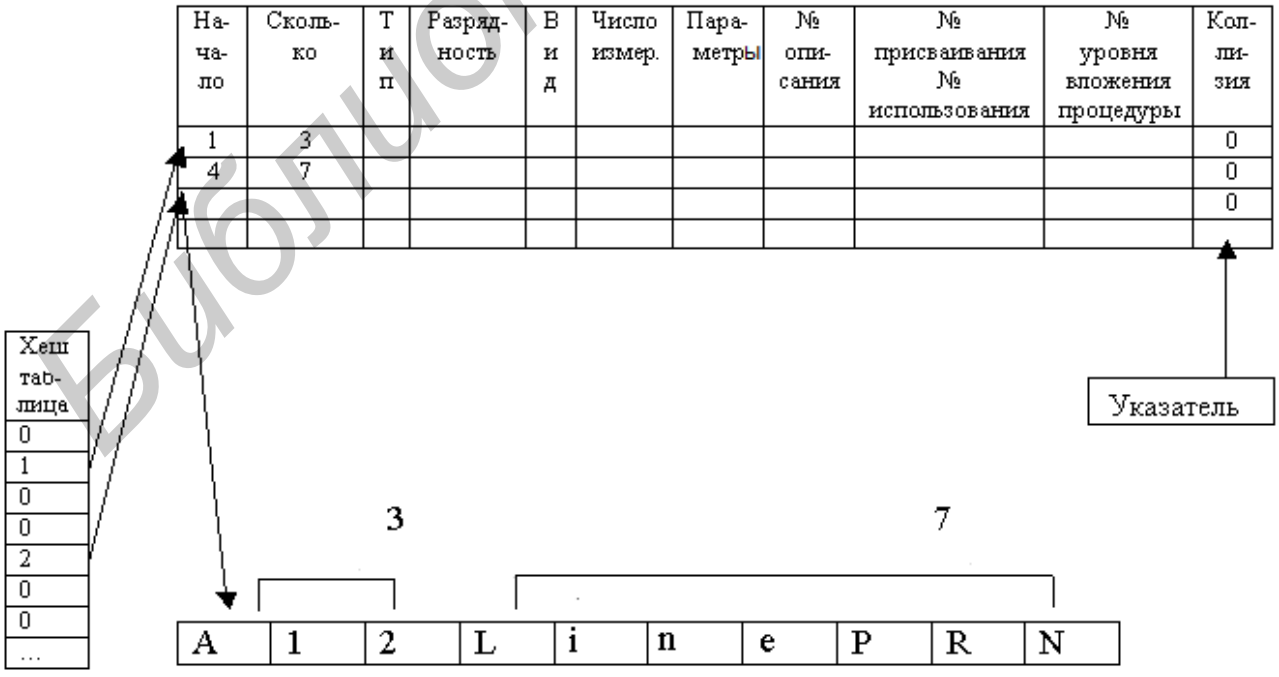


Рис. 10.1

10.4.2. Таблица меток

Таблица меток используется для сбора информации о контексте меток и контроля передач управления в данный контекст. Она позволяет определить неописанные метки, неиспользуемые метки, неправильные переходы.
Содержимое:

– имя метки;
– где она описана;
– контекст;
– операторы перехода.

Информация для полного контроля семантики передач управления может быть собрана только к концу трансляции, поэтому требуется дополнительный просмотр таблиц.

10.4.3. Таблицы контроля вложенности процедур и передач управления

Дерево статической вложенных процедур служит для анализа правильности вызова.

Граф-вызов содержит записи о вызовах других процедур из данной процедуры.

Стек статической вложенности предназначен для определения области действия имен (рис. 10.2). Состояния стека: P, P(Q), P(Q(A)), P(Q(B)), P(R).

10.4.4. Таблица процедур и дерево вложенности процедур

Дерево вложенности процедур используется для проверки области действия имен, служит для контроля передач управления и проверки соответствия типов «аргументы–параметры».

Дерево вложенности процедур должно содержать информацию, подобную следующей:

P(1,100) [Q(2,50) [C(3, 20), B(25, 40)]], R(60,90)]

и должно быть связано с таблицей процедур.

Таблица процедур содержит всю информацию об описании процедуры и ее контекста расположения в программе: какая процедура, внешняя или внутренняя; ее тип (если функция), количество и типы параметров.

Структура таблицы процедур приведена ниже (табл. 10.1).

Таблица 10.1

Ссылка на имя процедуры	Начальный оператор	Конечный оператор	Номер п/п процедуры	Номер уровня вложенности	Ссылка на параметр 1	Ссылка на параметр 2	Тип процедуры	Тип результата

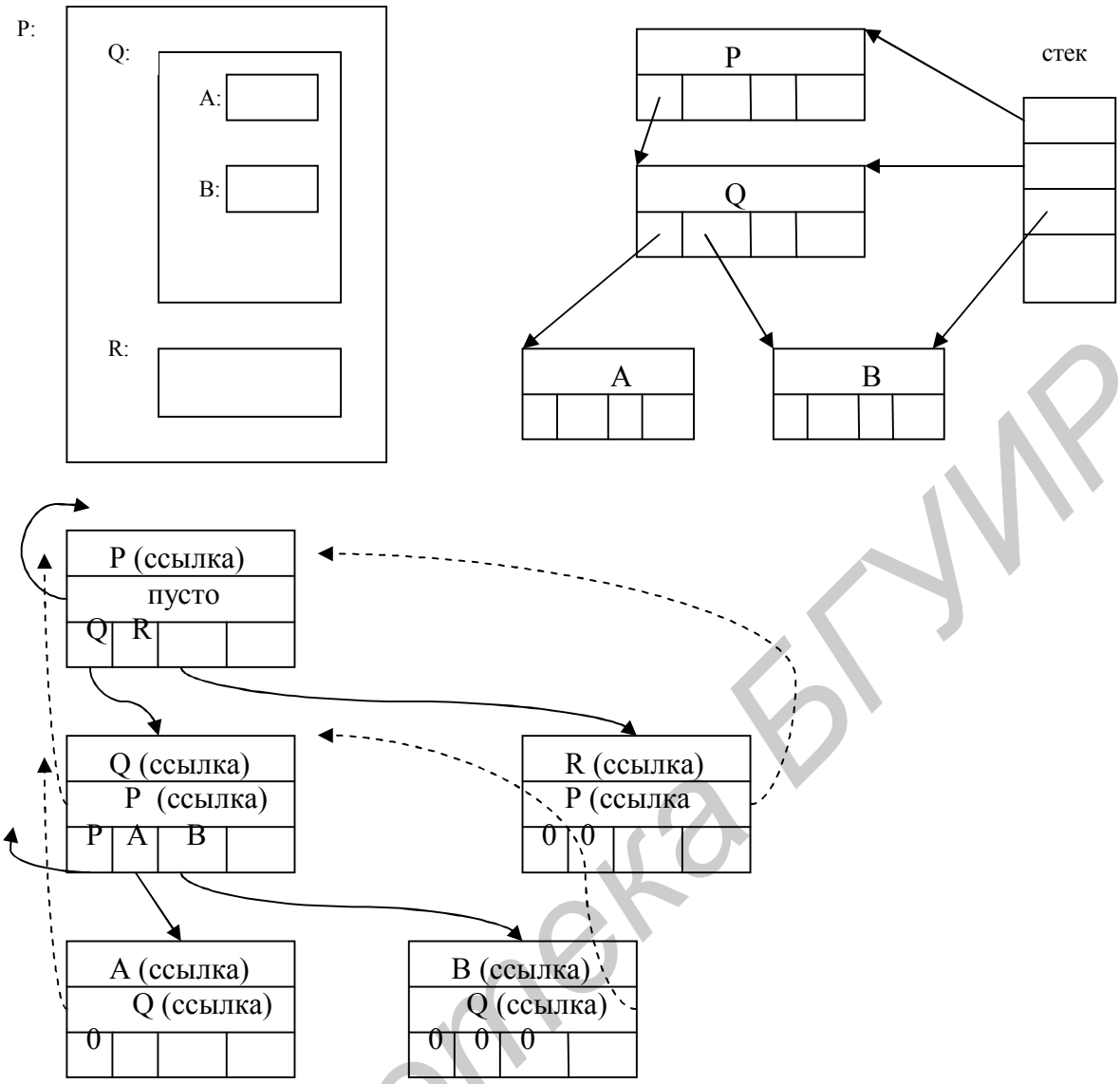


Рис. 10.2

10.4.5. Стек вложенности управляющих операторов

Стек управляющих операторов отражает полную картину вложенности структурных операторов, таких как:

Procedure	Function	Do	Do While	Select	If
End	End	End	End	End	EndIf ,

и служит для контроля структуры программы и передач управления.

Литература

1. Кнут, Д. Семантика контекстно-свободных языков // Семантика языков программирования / Д. Кнут. – М. : Мир, 1980. – С. 137–161.

2. Ахо, А. В. Теория синтаксического анализа, перевода и компиляции. Синтаксический анализ: Т.1, Т.2 / А. В. Ахо, Д. Д. Ульман. – М.: Мир, 1978.

3. Льюис, Ф. Теоретические основы проектирования компиляторов / Ф. Льюис, Д. Розенкранц, Р. Стирнз. – М. : Мир, 1979.

4. Ахо, А. В. Компиляторы: принципы, технологии и инструменты / А. В. Ахо, Р. Сети, Д. Д. Ульман. – М. : Издательский дом «Вильямс», 2001.

5. Грис, Д. Конструирование компиляторов для цифровых вычислительных машин / Д. Грис. – М. : Мир, 1975.

6. Хантер, Р. Проектирование и конструирование компиляторов / Р. Хантер. – М. : Финансы и статистика, 1984.

7. Зелковиц, М. Принципы разработки программного обеспечения / М. Зелковиц, А. Шоу, Дж. Гэннон. – М. : Мир, 1982.

8. Aho, A. Principles of compiler design / A. Aho, J. Ullman. – Addison-Wesley, Reading, MA, 1986.

Св. план 2011, поз. 55

Учебное издание

Пилецкий Иван Иванович
Шиманский Валерий Владимирович

МЕТОДЫ ТРАНСЛЯЦИИ

Методическое пособие
для студентов специальности 1-31 03 04 «Информатика»
всех форм обучения

Редактор Г. С. Корбут
Корректор Е. Н. Батурчик

Подписано в печать	Формат 60x84 1/16.	Бумага офсетная.
Гарнитура «Таймс».	Отпечатано на ризографе.	Усл. печ. л.
Уч.-изд. л. 4,5.	Тираж 120 экз.	Заказ 874.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0494371 от 16.03.2009. ЛП №02330/0494175 от 03.04.2009.
220013, Минск, П. Бровки, 6