МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

И.Л. Еланцева, И.А.Полетаева

Языки программирования и методы трансляции. Раздел «Методы трансляции»

Утверждено Редакционно-издательским советом университета
в качестве учебного пособия

Новосибирск
2007

Рецензенты:

канд. техн. наук, ст. преподаватель
кафедры автоматики НГТУ Е.Л.Веретельникова

д-р техн. наук, проф. кафедры
прикладной математики НГТУ Д.В. Лисицин

Еланцева И.Л., Полетаева И.А.

Языки программирования и методы трансляции. Раздел «Методы трансляции». Конспект лекций. – Новосибирск: Изд-во НГТУ, 2007. – 113 с.

Данный конспект лекций предназначен студентам ФПМИ, изучающим курс «Языки программирования и методы трансляции». Конспект лекций содержит описание методов проектирования сканера, синтаксических анализаторов нисходящего и восходящего типов, методов генерирования кодов.

Введение

Данный конспект лекций предназначен студентам ФПМИ, изучающим курс «Языки программирования и методы трансляции». Конспект лекций содержит описание методов проектирования сканера, синтаксических анализаторов нисходящего и восходящего типов, методов генерирования кодов.

1. ТРАНСЛЯТОРЫ

1.1. НАЗНАЧЕНИЕ, КЛАССИФИКАЦИЯ

Любую программу, которая переводит произвольный текст на некотором входном языке в текст на другом языке, называют транслятором. В частности, исходным текстом может быть **входная программа**. Транслятор переводит ее в выходную или объектную программу. Программа, полученная после обработки транслятором, либо непосредственно исполняется на машине, либо подвергается обработке другим транслятором.

Обычно процессы трансляции и исполнения программы разделены по времени. Сначала вся программа транслируется, а затем исполняется. Трансляторы, работающие в таком режиме, называются трансляторами **компилирующего** типа. Принцип, альтернативный компилированию, реализуется в программах, обычно называемых **интерпретаторами.** Интерпретатор – это программа, которая в качестве входа допускает программу на входном языке и по мере распознавания конструкций входного языка реализует их, выдавая на выходе результаты вычислений, предписанные исходной программой. Транслятор, использующий в качестве входного языка язык, близкий к машинному (автокод или ассемблер), традиционно называют **ассемблером**.

Назначение компилятора заключается в том, чтобы из исходного кода выработать выполнимый. В некоторых случаях компилятор выдает только код строки (ассемблера) для определенной машины, а окончательная выработка машинного кода осуществляется другой программой, называемой ассемблером.

Код ассемблера аналогичен машинному коду, но в нем для команд и адресов используется мнемоника, метки, которые могут применяться в качестве адресов перехода. Компилятор должен транслировать сложные конструкции языка высокого уровня в относительно простой машинный код или код сборки соответствующей машины.

Поэтому в работу компилятора вовлекаются два языка и, кроме них, язык, на котором написан сам компилятор. В простейших случаях это машинный код той ЭВМ, на которой он будет выполняться.

1.2. ОСНОВНЫЕ КОМПОНЕНТЫ ТРАНСЛЯЦИИ

Процесс трансляции можно представить структурной схемой, изображенной на рисунке 1. Из перечисленных на рисунке 1 фаз компиляции обязательными являются лексический, синтаксический анализ и генерация кода.

В состав любого компилятора входят три основных компонента:

– лексический анализатор (блок сканирования);

– синтаксический анализатор;

– генератор кода.

На фазе лексического анализа исходный текст программы разбивается на единицы языка. **Единица языка** – константа, идентификатор, ключевое слово, разделитель, знак операции. **Лексема** – неделимая единица языка и ее атрибуты. Множество атрибутов определяется типом единицы языка и может включать: имя, значение, тип, параметры (для функций). Лексемы сохраняются в соответствующих таблицах лексем.

Одной из важнейших функций компилятора является запись используемых в исходной программе идентификаторов и сбор информации о различных атрибутах каждого идентификатора. Эти атрибуты предоставляют сведения об отведенной идентификатору памяти, его типе, области видимости (в какой части программы он может применяться). При использовании имен процедур атрибуты говорят о количестве и типе их аргументов, методе передачи каждого аргумента (например, по ссылке) и типе возвращаемого значения, если таковое имеется. Вся

эта информация составляет лексему.

Рис.1. Фазы компиляции

Таблица символов представляет собой структуру данных, содержащую записи о каждом идентификаторе с полями для его атрибутов. Данная структура позволяет найти информацию о любом идентификаторе и внести необходимые изменения.

Когда идентификатор считан из исходной программы, требуется определить, не появлялся ли этот идентификатор ранее. Если лексическим анализатором в исходной программе обнаружен новый идентификатор, он записывается в таблицу символов.

После того, как лексема сохранена или найдена в таблице символов, сохраняем токен (тип лексемы и адрес в соответствующей таблице), связанный с этой лексемой, в файле токенов. Однако атрибуты идентификатора обычно не могут быть определены в процессе лексического анализа. В процессе остальных фаз информация об идентификаторе вносится в таблицу символов и используется различными способами. Например, при семантическом анализе и генерации промежуточного кода необходимо знать типы идентификаторов, чтобы

гарантировать их корректное использование в исходной программе и сгенерировать правильные операции по работе с ними. Обычно генератор кода вносит в таблицу символов и использует детальную информацию о памяти, назначенной идентификаторам. Для упрощения изложения в дальнейшем будем употреблять термин «лексема», имея ввиду неделимую единицу языка с атрибутами или без них в зависимости от контекста.

После разбиения программы на лексемы следует фаза **синтаксического анализа**, называемая **грамматическим разбором**, на котором проверяется правильность следования операторов. Например, для предложения *IF*, имеющего вид « *IF* выражение *THEN* предложение ; » , грамматический разбор состоит в том, чтобы убедиться, что вслед за лексемой *IF* следует правильное выражение, за этим выражением следует лексема *THEN*, за которой следует в свою очередь правильное предложение, оканчивающееся знаком «**;**».

Последним выполняется процесс **генерирования кода**, который использует результаты синтаксического анализа и создает программу, пригодную к выполнению.

Взаимодействие трех основных компонентов может осуществляться различными способами:

1) компилятор с тремя проходами (рисунок 2);
2) компилятор с двумя проходами (рисунок 3);
3) компилятор с одним проходом (рисунок 4).

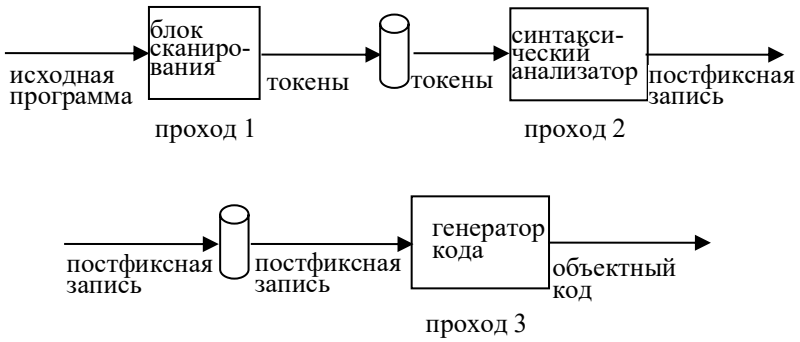


Рис.2. Компилятор с тремя проходами

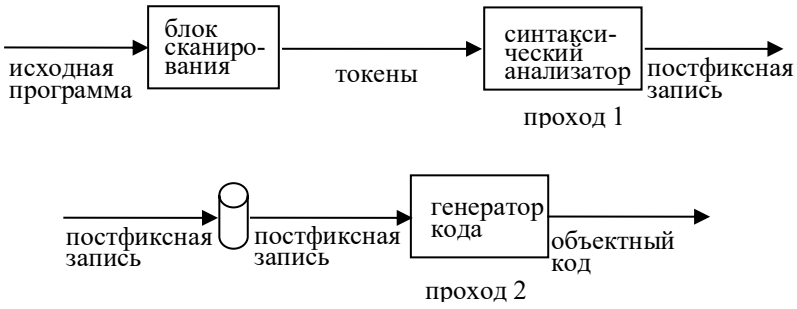


Рис.3. Компилятор с двумя проходами

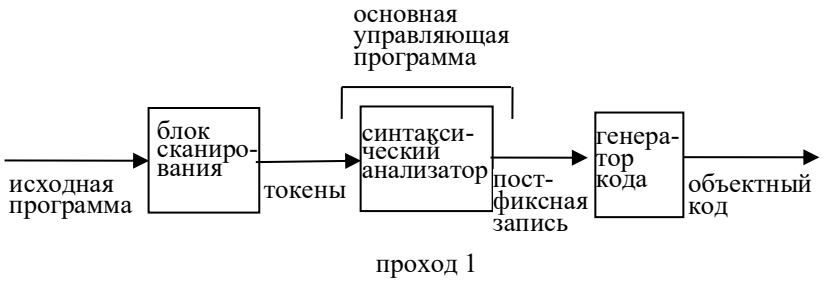


Рис.4. Компилятор с одним проходом

В первом варианте блок сканирования считывает исходную программу и представляет ее в виде **файла токенов.** Синтаксический анализатор читает этот файл и выдает новое представление программы в постфиксной форме. Этот файл считывается генератором кода, который создает объектный код программы. Компилятор такого вида называется трехпроходным, так как программа прочитывается трижды. Такая организация не способна придать компилятору высокую скорость выполнения.

Преимущества: относительная независимость каждой фазы компилирования и гибкость компилятора.

Высокая скорость компилирования может быть достигнута при использовании компилятора с однопроходной структурой. В этом случае синтаксический анализатор, выступая в роли **основной управляющей программы**, вызывает блок сканирования и генератор кода, организованные в

виде подпрограмм. Синтаксический анализатор постоянно обращается к блоку сканирования, получая от него токен за токеном до тех пор, пока не построит новый элемент постфиксной записи, после чего он обращается к генератору кода, который создает объектный код для этого фрагмента программы.

Такая программа отличается эффективностью, так как программа просматривается однажды, и в выполнении не участвуют операции обращения к файлам. Однако, такой организации свойственны недостатки.

1) Неоптимальность создаваемой объектной программы.

Например, если встретится текст

$A = (B + C)$;

$D = (B + C) + (E + F)$; ,

компилятор мог бы построить более эффективный объектный код,

трансформировав программу следующим образом

$A = (B + C)$;

$D = A + (E + F)$; .

Однако в однопроходном компиляторе часть нужной информации к тому времени, когда встретится выражение $D = A + (E + F)$; может быть утрачена.

2) Трудность в решении проблемы перехода по метке.

Во время обработки предложения «GOTO метка» могут возникнуть осложнения, если «метка» еще не встречалась в программе.

Поскольку однопроходной компилятор должен полностью размещаться в памяти, его реализация сопровождается повышенными требованиями к ресурсу памяти.

Структура двухпроходного компилятора занимает промежуточное положение между рассмотренными вариантами. В этом случае синтаксический анализатор, вызывая блок сканирования, получает токен за токеном и строит файл постфиксной записи программы.

Преимущества: небольшое время выполнения, легко разрешить проблему перехода по метке и задачу оптимизации программы.

Данные схемы не исчерпывают всего многообразия организационных

структур компилирующих программ.

В настоящее время некоторые модели ЭВМ сохраняют архитектуру, свойственную первым образцам ЭВМ, в то же время центральные процессоры целого ряда других моделей строятся на основе принципов микропрограммного управления. В результате появилась возможность создавать пользователем микропрограммы с новыми машинными командами. Это обстоятельство требует от компиляторов гибкости и умения приспосабливаться к различным наборам команд.

Компилирование программы включает **анализ** – определение предусмотренного результата действия программы и последующий **синтез –** генерирование эквивалентной программы в машинном коде.

В процессе анализа компилятор должен выяснить, является ли выходная программа действительной в каком-либо смысле (т.е. принадлежит ли она языку, для которого написан данный компилятор), и если она окажется недействительной – выдать соответствующее сообщение программисту. Этот аспект компиляции называется **обнаружением ошибок**.

Интерпретаторы имеют много общего с компилятором. Интерпретатор, также как и компилятор, вначале просматривает исходную программу и выделяет в ней лексемы. Для этого используются блоки сканирования и анализаторы, аналогичные тем, которые входят в состав компилирующих программ. Однако генератор кода вместо построения объектного кода, способного выполнять те или иные операции, сам производит соответствующие действия.

Так как реализация интерпретатора часто отличается сравнительной простой, нередко прибегают к интерпретации вместо компилирования, если скорость выполнения программы не имеет большого значения. Однако из этого не следует, что построить интерпретатор довольно просто. Зачастую в целях сокращения времени обработки программы, интерпретатору приходится тем или иным способом анализировать, прежде чем выполнять.

Интерпретатор может иметь структуру, аналогичную структуре 3- или 2-проходного компилятора. Это означает, что программа может быть переведена в

некоторую форму внутреннего представления, которое затем участвует в процессе выполнения. Это либо постфиксная запись, получаемая при использовании синтаксического анализатора, язык низкого уровня, ориентированный на реальную ЭВМ или гипотетическую машину, которую имитирует интерпретатор, или какая-либо другая форма записи программы.

Применение интерпретатора вместо компилятора имеет следующие преимущества:

1. Передавать сообщения об ошибках пользователю часто бывает легче в терминах оригинальной программы.

2. Версия программы на компилируемом языке нередко оказывается компактнее, чем машинный код, выданный компилятором.

3. Изменение части программы не требует перекомпиляции всей программы. Диалоговые языки часто реализуются посредством интерпретатора. Промежуточным языком обычно служит некая форма обратной польской записи (ОПЗ). Основной недостаток интерпретатора в малой скорости, так как операторы промежуточного кода должны транслироваться всякий раз, когда они выполняются. Временные задержки зависят от конструкции промежуточного языка и не так уж велики.

Применяется метод смешанного кода, в котором наиболее часто выполняемая часть интерпретируется. При этом экономится память, поскольку часть программы, которая должна интерпретироваться, будет по всей вероятности, намного компактнее, чем скомпилированный код.

Интерпретаторы используются также и при эмуляции программ – исполнении на технологической машине программ, составленных для другой (объектной) машины. Данный вариант, в частности, используется при отладке на универсальной ЭВМ программ, которые будут выполняться на специализированной ЭВМ.

Методы трансляции, используемые в компиляторах можно разделить на две группы: *прямые* и *синтаксические*.

Прямые методы трансляции ориентированы, на конкретные входные

языки. Это преимущественно эвристические методы, в которых на основе некоторой общей идеи для каждой конструкции входного языка подбирается индивидуальный алгоритм трансляции. Этапы синтаксического и семантического анализов здесь обычно четко не разделены [6].

Синтаксические методы трансляции отличаются более или менее четко выраженным разделением этапов синтаксического и семантического анализов. Синтаксические методы основаны на теории формальных грамматик. Каждый из этих методов ориентирован не на конкретный входной язык, а на некоторый класс входных языков, точнее, на определенный способ описания синтаксиса входных языков. Поэтому эти методы называются синтаксически ориентированными [1–5, 7].

1.3. НЕКОТОРЫЕ АСПЕКТЫ ПРОЦЕССА КОМПИЛЯЦИИ

Работа компилятора складывается из двух основных этапов [4]. Сначала он распознает структуру и значение программы, которую он должен компилировать, а затем он выдает эквивалентную программу в машинном коде (или коде сборки). Эти два этапа называются анализ и синтез.

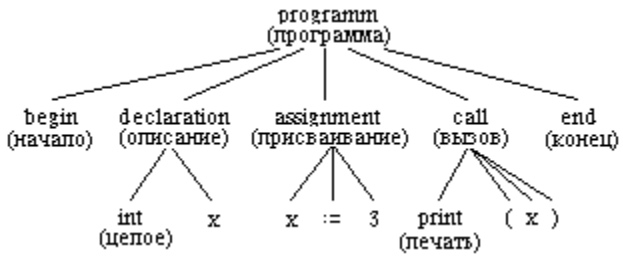
По идее анализ должен проводиться перед синтезом, но на практике они выполняются почти параллельно. Определив исходный язык, мы тем самым задаем значение его каждой допускаемой конструкции. После того, как анализатор распознает все конструкции в программе, он может установить, каким должен быть результат действия этой программы. Затем синтезатор вырабатывает соответствующий объектный код.

Структуру программы удобно представлять в виде дерева.

Пример. Для программы

```
begin
  int x;
  x:=3;
  print(x)
end
```

схема в виде дерева, поясняющая ее структуру, имеет вид:



Допустим, что конечными вершинами дерева разбора являются не отдельные литеры, а единицы языка. В начальной фазе работы компилятор объединяет литеры в так называемые лексемы. Эта фаза процесса компиляции называется лексическим анализом. Остальная часть анализа, т.е. построение дерева, называется синтаксическим анализом или разбором.

Перевод программы с одного языка на другой, в общем случае, состоит в изменении алфавита, лексики и синтаксиса языка программы с сохранением её семантики. Синтаксис языка представляет собой описание правильных предложений. Описание смысла предложений, т.е. значений слов и их внутренних связей, составляет семантику языка. В дополнение заметим, что конкретная программа несет в себе некоторое воздействие на транслятор – прагматизм. В совокупности синтаксис, семантика и прагматизм языка образуют семиотику языка. Процесс трансляции исходной программы в объектную обычно разбивается на несколько независимых подпроцессов (фаз трансляции), которые реализуются соответствующими блоками транслятора. Удобно считать основными фазами трансляции лексический анализ, синтаксический анализ, семантический анализ и синтез объектной программы. Тем не менее, во многих реальных компиляторах эти фазы разбиваются на несколько подфаз, могут также быть и другие фазы (например, оптимизация объектного кода) [3, 7].

Входная программа прежде всего подвергается лексической обработке. Одновременно с переводом исходной программы на внутренний язык (токены) на этапе лексического анализа проводится лексический контроль – выявление в программе недопустимых слов.

Синтаксический анализатор воспринимает выход лексического анализатора

(токены) и переводит последовательность образов лексем (токенов) в форму промежуточной программы. Промежуточная программа является, по существу, представлением синтаксического дерева программы [3]. Последнее отражает структуру исходной программы, т.е. порядок и связи между ее операторами. В ходе построения синтаксического дерева выполняется *синтаксический контроль* – выявление синтаксических ошибок в программе.

Фактическим выходом синтаксического анализатора может быть последовательность команд, необходимых для того, чтобы строить промежуточную программу, обращаться к таблицам справочника, выдавать, когда это требуется, диагностические сообщения [7].

Синтез объектной программы начинается, как правило, с распределения и выделения памяти для основных программных объектов. Затем производится исследование каждого предложения исходной программы и генерируются семантически эквивалентные предложения объектного языка. В качестве входной информации здесь используется синтаксическое дерево программы и выходные таблицы лексического анализатора – таблица идентификаторов, таблица констант и другие. Анализ дерева позволяет выявить последовательность генерируемых команд объектной программы, а по таблице идентификаторов определяются типы команд, которые допустимы для значений операндов в генерируемых командах (например, какие требуется породить команды: с фиксированной или плавающей точкой и т.д.). Непосредственно генерации объектной программы часто предшествует *семантический анализ,* который включает различные виды семантической обработки. Один из видов – проверка семантических соглашений в программе. Примеры таких соглашений: единственность описания каждого идентификатора в программе, определение переменной производится до ее использования и т.д. Семантический анализ может выполняться и на более поздних фазах трансляции, например, на фазе оптимизации программы, которая тоже может включаться в транслятор. Цель оптимизации – сокращение временных ресурсов или ресурсов оперативной памяти, требуемых для выполнения объектной программы.

Введение промежуточного языка на этапе анализа связанно с попыткой разделить зависимые и независимые от машины аспекты генерирования кода. В качестве промежуточного языка можно использовать четверки. Во многих компиляторах нет явного разделения этих двух аспектов генерирования кода. Но если компилятор предполагается сделать переносимым, то желательно как можно полнее разделить его зависимые и независимые от машины части.

Таковы основные аспекты процесса трансляции с языков высокого уровня. Подробнее организация различных фаз трансляции и связанные с ними практические способы их математического описания будут рассмотрены далее.

1.4. ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРА

Два компилятора, реализующие один и тот же язык на одной и той же машине, могут отличаться, если разработчики преследовали различные цели при их построении. Этими целями могут быть:

1) получение эффективного объектного кода;
2) разработка и получение объектных программ;
3) минимизация времени компиляции;
4) разработка компилятора минимального размера;
5) создание компилятора, обладающего широкими возможностями обнаружения и исправления ошибок;
6) обеспечение надежности компилятора.

Эти цели в некоторой степени противоречивы. Компилятор, который должен выдавать эффективный код будет медленней. Для осуществления некоторых стандартных методов оптимизации, таких, как устранение избыточности кода, исключение последовательностей кода из циклов и т. д. может потребоваться очень много времени. Компилятор, в котором выполняются сложные программы оптимизации, является также большим по объему. Размер компилятора может влиять на время, необходимое ему для компиляции программы. В процессе работы часто невозможно оптимизировать время и объем памяти одновременно. Разработчик заранее должен решить, что он предпочитает

оптимизировать, отдавая себе отчет, к чему это может привести. Кроме того, у разработчика может возникнуть желание потратить минимальное время на написание компилятора, что само по себе является препятствием к включению некоторых сложных методов оптимизации.

Программисты ожидают от компилятора вспомогательного сообщения об ошибке. Они также предполагают, что компилятор, обнаружив ошибку, будет продолжать анализировать программу. Компиляторы существенно отличаются по своим возможностям обнаружения и исправления ошибок. Проблема исправления ошибок (особенно синтаксических) не проста и полностью пока не решена. Компилятор, выдающий полезные сообщения и элегантно выходящий из ситуации ошибки, имеет, как правило, больший размер, что вполне компенсируется повышением эффективности работы программы.

Обеспечение надежности должно быть первостепенной задачей при создании любого компилятора. Компиляторы часто представляют собой очень большие программы, а современное состояние науки и техники в этой области не позволяет дать формальное подтверждение правильности компилятора. Наибольшее значение здесь имеет хороший общий проект. Если различные фазы процесса сделать относительно различимыми и каждую фазу построить как можно проще, то вероятность создания надежного компилятора возрастает. Надежность компилятора повышается и в том случае, если он базируется на ясном и однозначном формальном определении языка и если используются такие автоматические средства, как генератор синтаксических анализаторов.

Цели проектирования компилятора часто зависят от той среды, в которой он должен использоваться. Если он предназначается для студентов, эффективность кода будет иметь меньшее значение, чем скорость компиляции и возможность обнаружения ошибок. В производственной среде - наоборот.

В построении компиляторов большую роль играет решение о числе проходов, которые ему придется выполнить. Создание многопроходного компилятора связано с проектированием промежуточных языков для версий исходного текста, существующих между проходами. Строятся также таблицы

какого-либо прохода, которые могут понадобиться в дальнейшем.

2. ЛЕКСИЧЕСКИЙ АНАЛИЗ

Этап лексической обработки текста исходной программы выделяется в отдельный этап работы компилятора, как с методическими целями, так и с целью сокращения общего времени компиляции программы. Последнее достигается за счет того, что исходная программа, представленная на входе компилятора в виде непрерывной последовательности символов, на этапе лексической обработки преобразуется к некоторому стандартному виду, что облегчает дальнейший анализ. При этом используются специализированные алгоритмы преобразования, теория и практика построения которых проработана достаточно глубоко [2].

2.1. ЗАДАЧИ И ФУНКЦИОНИРОВАНИЕ БЛОКА ЛЕКСИЧЕСКОГО АНАЛИЗА

Под *лексическим анализом* будем понимать процесс предварительной обработки исходной программы, на котором основные лексические единицы программы – ключевые (служебные) слова, идентификаторы, метки, константы приводятся к единому формату и заменяются условными кодами или ссылками на соответствующие таблицы (токены), а комментарии исключаются из текста программы [3].

Выходами лексического анализа являются поток токенов (образов лексем) и таблицы, в которых хранятся значения выделенных в программе лексем.

Определение. *Токен* – это пара вида (<тип лексемы>, <указатель>),
где <тип лексемы> – это, как правило, числовой код класса лексемы, который означает, что лексема принадлежит одному из конечного множества классов слов, выделенных в языке программирования;

<указатель> – это может быть либо начальный адрес области основной памяти**,** в которой хранится адрес этой лексемы, либо число, адресующее элемент таблицы, в которой хранится значение этой лексемы.

Количество классов лексем (т.е. различных видов слов) в языках программирования может быть различным. Наиболее распространенными классами являются:

- идентификаторы;
- служебные (ключевые) слова;
- разделители;
- константы.

Могут вводиться и другие классы. Это обусловлено в первую очередь той ролью, которую играют различные виды слов при написании исходной программы и, соответственно, при переводе ее в машинную программу. При этом наиболее предпочтительным является разбиение всего множества слов, допускаемого в языке программирования, на такие классы, которые бы не пересекались между собой. В этом случае лексический анализ можно выполнить более эффективно. В общем случае все выделяемые классы являются либо **конечными** (ключевые слова, разделители и др.) – классы фиксированных для данного языка программирования слов, либо **бесконечными** или очень большими (идентификаторы, константы, метки) – классы переменных для данного языка программирования слов.

С этих позиций коды образов лексем (токены) из конечных классов всегда одни и те же в различных программах для данного компилятора. Коды же образов лексем из бесконечных классов различны для разных программ и формируются каждый раз на этапе лексического анализа.

В ходе лексического анализа значения лексем из бесконечных классов помещаются в таблицы соответствующих классов. Конечность таблиц объясняет ограничения, существующие в языках программирования на длины (и соответственно число) используемых в программе идентификаторов и констант. Необходимо отметить, что числовые константы перед помещением их в таблицу могут переводиться из внешнего символьного во внутреннее машинное представление. Содержимое таблиц, в особенности таблицы идентификаторов, в дальнейшем пополняется на этапе семантического анализа

исходной программы и используется на этапе генерации объектной программы.

Рассмотрим основные идеи, которые лежат в основе построения лексического анализатора, проблемы, возникающие при его работе, и подходы к их устранению [3].

Первоначально в тексте исходной программы лексический анализатор выделяет последовательность символов, которая по его предположению должна быть словом в программе, т.е. лексемой. Может выделяться не вся последовательность, а только один символ, который считается началом лексемы. Это наиболее ответственная часть работы лексического анализатора. Более просто она реализуется для тех языков программирования, в которых слова в программе отделяются друг от друга специальными разделителями (например, пробелами), либо запрещено использование служебных слов в качестве переменных, либо классы лексем опознаются по вхождению первого (последнего) символа лексемы.

После этого (или параллельно с этим) проводится идентификация лексемы. Она заключается в сборке лексемы из символов, начиная с выделенного на предыдущем этапе, и проверки правильности записи лексемы данного класса.

Идентификация лексемы из конечного класса выполняется путем сравнения ее с эталонным значением. Основная проблема здесь – минимизация времени поиска эталона. В общем случае может понадобиться полный перебор слов данного класса, в особенности для случая, когда выделенное для опознания слово содержит ошибку. Уменьшить время поиска можно, используя различные методы ускоренного поиска: метод линейного списка, метод упорядоченного списка, метод расстановки и другие [8].

Для идентификации лексем из бесконечных (очень больших) классов используются специальные методы сборки лексем с одновременной проверкой правильности написания лексемы. При построении этих алгоритмов широко применяется формальный математический аппарат – теория регулярных языков, грамматик и конечных распознавателей [2, 3].

При успешной идентификации значение лексемы из бесконечного класса

помещается в таблицу идентификаторов лексем данного класса. При этом необходимо предварительно проверить: не хранится ли там уже значение данной лексемы, т.е. необходимо проводить просмотр элементов таблицы. Если ее там нет, то значение помещается в таблицу. При этом таблица должна допускать расширение. Опять же для уменьшения времени доступа к элементам таблицы она должна быть специальным образом организована, при этом должны использоваться специальные методы ускоренного поиска элементов.

После проведения успешной идентификации лексемы формируется ее образ – токен, он помещается в выходной поток лексического анализатора. В случае неуспешной идентификации формируются сообщения об ошибках в написании слов программы.

В ходе лексического анализа могут выполняться и другие виды лексического контроля, в частности, проверяется парность скобок и других парных символов.

Выходной поток с лексического анализатора в дальнейшем поступает на вход синтаксического анализатора. Имеется две возможности их связи:

- раздельная связь, при которой выход лексического анализатора формируется полностью и затем передается синтаксическому анализатору;
- нераздельная связь, когда синтаксическому анализатору требуется очередной токен, он вызывает лексический анализатор, который генерирует требуемый токен и возвращает управление синтаксическому анализатору.

Второй вариант характерен для однопроходных трансляторов. Таким образом, процесс лексического анализа может быть достаточно простым, но в смысле времени компиляции оказывается довольно долгим. Как указывается в [4], больше половины времени, затрачиваемого компилятором на компиляцию, приходится на этап лексического анализа. В следующем примере для приведенной программы показан возможный вариант содержимого таблиц и выхода лексического анализатора.

Пример. Вход лексического анализатора:

```
Program Primer;
Var  X, Y, Z : Real;
Begin
X := 5;
Y := 5;
Z := X + Y;
End;
```

Внутренние таблицы лексического анализатора:

Таблица ключевых слов

№ п/п	Ключевое слово
1	*Program*
2	*Begin*
3	*End*
4	*For*
5	*Real*
6	*Var*
…	. . .

Таблица разделителей

№ п/п	Разделитель
1	;
2	,
3	+
4	–
5	/
6	*
7	:
8	=
9	.
…	. . .

Выход лексического анализатора:

Таблица идентификаторов

№ п/п	Идентификатор
1	*Primer*
2	*X*
3	*Y*

Таблица констант

№	Знач. константы
1	5

4	*Z*

Пусть в качестве кодов типов лексем используются числа:

10 – ключевое слово; 20 – разделитель; 30 – идентификатор; 40 – константа.

Файл токенов будет иметь вид:

(10,1) (30,1) (20,1)

(10, 6) (30, 2) (20, 2) (30, 3) (20, 2) (30, 4) (20, 7) (10, 5) (20, 1)

(10, 2)

(30, 2) (20, 7) (20, 8) (40, 1) (20,1)

(30, 3) (20, 7) (20, 8) (40, 1) (20,1)

(30, 4) (20, 7) (20, 8) (30, 2) (20, 3) (30, 3) (20, 1)

(10, 3) (20, 1).

2.2. ПРОГРАММИРОВАНИЕ СКАНЕРА

Покажем, как программируется сканер [9, 10]. Сканер самая простая часть компилятора, иногда называемая лексическим анализатором.

Разработка лексического анализатора выполняется достаточно просто, если при этом воспользоваться хорошо разработанным математическим аппаратом. Таким аппаратом является теория регулярных языков и конечных автоматов. В рамках этой теории классы однотипных лексем (идентификаторы, константы и т. д.) рассматриваются как формальные языки (язык идентификаторов, язык констант и т.д.), множество предложений которых описывается с помощью соответствующей порождающей грамматики. При этом языки эти настолько просты, что они порождаются простейшей из грамматик – регулярной грамматикой (грамматика класса 3 в иерархии Хомского). Построенная регулярная грамматика является источником, по которому в дальнейшем конструируется вычислительное устройство, реализующее функцию распознавания предложений языка, порождаемого данной грамматикой. Для регулярных языков таким устройством является конечный автомат. Основные понятия теории регулярных языков и конечных автоматов и ее практическое использование при разработке

сканера рассмотрены в [2, 3] .

Начнем реализацию сканера с того, что нарисуем диаграмму состояний, показывающую, как ведется разбор символа (рисунок 5).

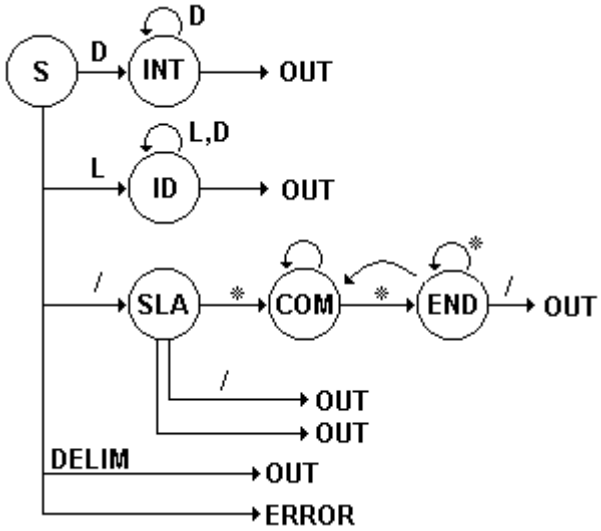


Рис.5. Диаграмма состояний автомата для разбора символов

Метка D используется вместо любой из меток 0,1,2....., 9, т.е. D представляет класс цифр (для упрощения диаграммы)

Метка L представляет класс букв *A, B,.....Z*.

DELIM представляет класс разделителей состоящих из одной литеры +, -, *, (,). Заметим, что литера / не принадлежит к классу разделителей, т.к. обрабатывается особым образом.

Непомеченные дуги будут выбраны, если сканируемая (обозреваемая) литера не совпадает ни с одной из литер, которыми помечены другие дуги.

Дуги, ведущие в OUT говорят от том, что обнаружен конец символа и необходимо покинуть сканер. Одна из проблем, которая возникает при переходе в OUT, состоит в том, что в этот момент не всегда сканируется литера, следующая за распознанным символом. Так, литера выбрана при переходе в OUT из INT, но она еще не выбрана, если только что распознан разделитель (DELIM). Когда потребуется следующий символ, необходимо знать, сканировалась ли уже его первая литера. Это можно сделать, используя, например переключатель. В дальнейшем будем считать, что перед выходом из сканера следующая литера всегда выбрана.

Замечание. Мы составили детерминированную диаграмму, т.е. мы сами определили, что начинается с литеры / : SL (/), SLSL (//) или комментарий, поскольку всегда идем в общее для них состояние SLA.

Теперь перейдем к рассмотрению диаграммы состояний с семантическими процедурами (рисунок 6) [5].

Для работы сканера потребуются следующие переменные и подпрограммы:

1. CHARACTER CHAR, где CHAR – глобальная переменная, значение которой всегда будет сканируемая литера исходной программы.

2. INEGER CLASS, где CLASS – содержит целое число, которое характеризует класс литеры находящейся в CHAR. Будем считать, что класс D (цифра)=1, класс L (буква)=2, класс, содержащий литеру / , =3, класс DELIM=4.

3.STRING A – ячейка, которая будет содержать цепочку (строку) литер, составляющих символ.

4. GETCHAR – процедура, задача которой состоит в том, чтобы выбрать следующую литеру исходной программы и поместить ее в CHAR, а класс литеры в CLASS. Кроме того, GETCHAR, когда это необходимо, будет читать и печать следующую строку исходной программы и выполнять другие подобные мелочи.

Гораздо предпочтительнее использовать для таких целей отдельную процедуру, чем повторять группу команд в тех местах, где необходимо сканировать литеру.

5. GETNOBLANK. Эта программа проверяет содержимое CHAR и, если там пробел, повторно вызывает GETCHAR до тех пор, пока в CHAR не окажется литера, отличная от пробела.

GC – вызов процедуры GETCHAR.

Выражение OUT(C,D) означает возврат к программе, которая вызвала сканер, с двумя величинами C и D в качестве результата.

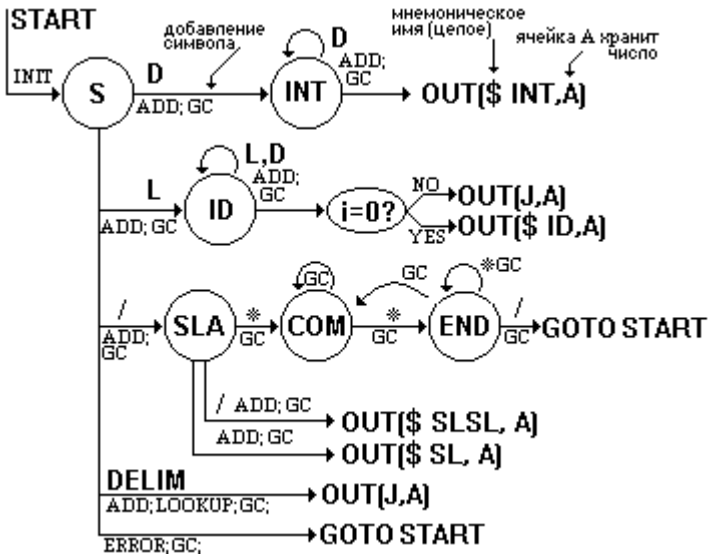


Рис.6. Диаграммы состояний с семантическими процедурами

Над первой дугой, ведущей к начальному состоянию S, записана команда INIT, которая указывает на необходимость выполнения последовательных операций и начальных установок (инициализация).

В данном случае выполняются команды:

GETNOBLANK;

A:= ' ';

Тем самым в CHAR заносится литера отличная от пробела, и производится начальная установка ячейки *A*, в которой будет хранится символ. Команда ADD означает, что литера CHAR добавляется к символу в A.

Команда LOOCUP осуществляет поиск символа набранного в A, по таблице служебных слов и разделителей. Если символ является служебным словом или разделителем, то соответствующий индекс заносится в глобальную переменную J, в противном случае туда записывается 0.

При обнаружении комментария или ошибки мы не выходим из сканера, а начинаем сканировать следующий символ.

Пример. Проследим за разбором символа // . Начнем с инициализации (INIT). Затем переходим в состояние S. Проверяем CHAR, находим, что там /, и по соответствующей дуге переходим в состояние SLA. В момент перехода

добавляем / в A (команда ADD) и вызываем GETCHAR, чтобы сканировать следующую литеру в CHAR, и выбираем дугу, помеченную /. В момент перехода по дуге добавляем литеру из CHAR к цепочке в A и сканируем следующую литеру (GC), заканчиваем работу возвратом к программе, которая вызвала сканер, с парой величин ($SL SL, A).

К моменту выхода литера следующего символа находится в CHAR.

Процедура имеет два параметра: первый параметр – внутреннее представление получаемого символа, второй – цепочка литер, составляющих символ.

После выделения единицы языка производится формирование соответствующего токена и запись нового идентификатора либо константы в соответствующие таблицы.

3.СИНТАКСИЧЕСКИЙ АНАЛИЗ

3.1. НИСХОДЯЩИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ

3.1.1. Общая характеристика и проблемы нисходящего анализа

При нисходящем синтаксическом анализе (анализе сверху вниз) вывод для заданной входной цепочки строят, исходя из начального символа грамматики (аксиомы), т.е. построение синтаксического дерева разбора начинается с корня и разворачивается к листьям [3]. Для этого анализируется несколько первых символов входной цепочки и решается, какое правило для аксиомы можно применить, а затем пытаются распознать элементы правой части этого правила, определяя их как подцели. Поиск подцели приводит к поиску вспомогательных подцелей и т.д. до тех пор, пока не приводит к сравнению символов анализируемой цепочки с некоторыми определенного типа терминальными символами. Они и дают ответ: достигнута ли подцель.

Отличительная черта методов нисходящего разбора заключается в том, что состояние алгоритма (текущая цель) используется как вспомогательная информация для принятия решения. Опознание подцелей производится слева направо и

когда они опознаны, то опознана и сама цель.

В общем случае, в ходе нисходящего анализа, возникают следующие проблемы:

1. Первая проблема возникает, когда цель определяется с использованием правила с левой рекурсией.

Если A – цель, и первое же правило для A имеет вид $A \rightarrow A\ \xi^*$, то в соответствии с данным методом будет установлена подцель A. В свою очередь, вспомогательная подцель A потребуется для введенной подцели A и т.д. Таким образом, леворекурсивные правила при нисходящем разборе надо исключать из грамматики.

2. Предположим, что при нисходящем анализе необходимо заменить самый левый нетерминал, определяемый правилом вида

$A \rightarrow \xi_1^* | \xi_2^* | \ldots | \xi_n^*$

Как узнать, какой цепочкой ξ_i^*, $i = 1,..., n$ необходимо заменить нетерминал A? Общего метода нет. Есть рекомендации:

- упорядочить правила так, чтобы наиболее вероятные альтернативы испытывались первыми. Обычно первой испытывают самую длинную альтернативу. Однако, если входная строка содержит ошибку, все равно придется перебрать все альтернативы;
- заглянуть вперед на 1-2 символа. Это позволяет более точно выбрать альтернативу;
- учитывать использованные в данных обстоятельствах альтернативы. Если альтернативы имеют общие префиксы и их неприемлемость фиксируется в пределах этих префиксов, то можно пропускать целую группу альтернатив с соответствующими префиксами;
- ограничить количество возвратов по глубине.

3. Проблема локализации ошибок. Компилятор должен локализовать место ошибки, помимо указания об ее наличии. Ошибка обнаруживается, когда все подходящие правила исчерпаны. Для локализации ошибки все альтернативы

подходят в равной степени, поэтому в грамматику необходимо вставить правила, реагирующие на ошибки (они описывают неправильные конструкции) и при срабатывании такого правила выдавать сигнал об ошибке или пытаться исправить ее.

Существует целый класс КС-грамматик, которые допускают детерминированный разбор сверху вниз, т.е. разбор без возвратов. Помимо высокой скорости разбора детерминированные методы имеют преимущество в компиляторах, где разбор идет параллельно с построением объектной программы. В таких компиляторах в случае возврата приходится отменять выполняемые действия по построению объектной программы, что весьма дорого обходится. Хотя подкласс КС-языков, разбираемых детерминировано, уже всего класса КС-языков, но оказалось, что большинство языков программирования можно разбирать детерминировано.

3.1.2. Метод рекурсивного спуска

Метод рекурсивного спуска – легко реализуемый детерминированный метод разбора сверху вниз [1, 3]. С его помощью на основании соответствующей грамматики можно написать синтаксический анализатор.

Синтаксический анализатор содержит по одной рекурсивной процедуре для каждого нетерминала *U*. Каждая такая процедура осуществляет разбор фраз, выводимых из *U*. Процедуре сообщается, с какого места данной программы следует начинать поиск фразы, выводимой из *U*. Следовательно , такая процедура целенаправленная или прогнозирующая. Предполагаем, что сможем найти такую фразу. Процедура ищет эту фразу, сравнивая программу в указанном месте с правыми частями правил для *U* и вызывая по мере необходимости другие процедуры для распознавателя промежуточных целей.

Для разбора предложения языка может потребоваться много рекурсивных вызовов процедур, составляющих нетерминалы в грамматике. Если изменить представление грамматики, рекурсию можно заменить итерацией. Замена рекурсии итерацией, возможно делает анализатор более эффективным, а также (не

бесспорно)более удобочитаемым.

Преимущества написания рекурсивного нисходящего анализатора очевидны. Основное из них – это скорость написания анализатора на основании соответствующей грамматики. Другое преимущество заключается в соответствии между грамматикой и анализатором, благодаря которому увеличивается вероятность того, что анализатор окажется правильным, или, по крайней мере, того, что ошибки будут носить простой характер.

Недостатки метода, хотя и менее очевидны, не менее реальны. Из-за большого числа вызовов процедур во время синтаксического анализа анализатор становится относительно медленным. Кроме того, он может быть довольно большим по сравнению с анализаторами, основанными на табличных методах разбора (методы предшествования, *LL*- и *LR*-методы). Метод рекурсивного спуска способствует включению в анализатор действий по генерированию кода, это неизбежно приводит к смешиванию различных фаз компиляции. Последнее снижает надежность компиляторов или усложняет обращение с ними и как бы привносит в анализатор зависимость от машины.

3.1.3. LL-метод синтаксического анализа

LL(1)-грамматика – это грамматика такого типа, на основании которой можно получить детерминированный синтаксический анализатор. Прежде чем определить *LL*(1)-грамматику, введем понятие *S*- грамматики.

Определение. *S - грамматика* представляет собой грамматику, в которой:

1. Правая часть каждого порождающего правила начинается с терминала;

2. В тех случаях, когда в левой части более чем одного порождающего правила появляется нетерминал, соответствующие правые части начинаются с разных терминалов.

Первое условие аналогично утверждению, что грамматика находится в нормальной форме Грейбах.

Второе условие помогает написать детерминированный нисходящий анализатор, так как при выводе предложения языка всегда можно сделать выбор между альтернативными порождающими правилами для самого левого нетерминала в сентенциальной форме, предварительно исследовав следующий символ.

Пример. Грамматика с порождающими правилами

$S \to pX \qquad S \to qY$

$X \to aXb \qquad X \to x$

$Y \to aYd \qquad Y \to y$

представляет собой *S*-грамматику, тогда как следующая грамматика, которая генерирует тот же язык, не является ею:

$S \to R \qquad S \to T$

$R \to pX \qquad T \to qY$

$X \to aXb \qquad X \to x$

$Y \to aYd \qquad Y \to y,$

поскольку правые части двух правил не начинаются с терминалов.

В некоторых случаях грамматику, которая не является *S*-грамматикой, можно преобразовать в нее, не затрагивая при этом генерируемый язык.

Пример. Рассмотрим разбор строки «*paaaxbbb*» с помощью *S*-грамматики. Начав с символа *S*, попытаемся генерировать строку. Применим левосторонний вывод. В исходной строке жирным шрифтом отмечен символ, обрабатываемый в данный момент.

Исходная строка	Вывод
p*aaaxbbb*	*S*
p*aaaxbbb*	*pX*
*p****a****aaxbbb*	*paXb*
*pa****a****axbbb*	*paaXbb*
*paa****a****xbbb*	*paaaXbbb*
*paaa****x****bbb*	*paaaxbbb*

При выводе начальные терминалы в сентенциальной форме сверяются с символами в исходной строке. Там, где допускается замена самых левых нетерминалов в сентенциальной форме с помощью более чем одного порождающего правила всегда можно выбрать соответствующее порождающее правило, исследовав следующий символ во входной строке. Это связано с тем, что поскольку мы имеем дело с *S*-грамматикой, правые части альтернативных порождающих правил будут начинаться с различных терминалов. Таким образом, всегда можно написать детерминированный анализатор, осуществляющий разбор сверху вниз для языка, генерируемого *S*-грамматикой.

S-грамматики, хотя и допускают детерминированный нисходящий разбор, описывают достаточно узкий класс языков. Естественным расширением этого класса было бы – разрешить, чтобы первый символ в правой части правила подстановки грамматики мог бы быть нетерминалом, но при этом грамматика всё же допускала детерминированный разбор.

LL(1)-грамматика является обобщением *S*-грамматики. Принцип ее обобщения позволяет строить нисходящий детерминированный анализатор. Обозначение «*LL*(1)» означает, что строки разбираются слева направо (первая *L*) и используются самые левые выводы (вторая *L*), а цифра 1 – что варианты порождающих правил выбираются с помощью одного предварительного просмотренного символа.

Пример.

На основании порождающих правил

$S \to RY$ $\qquad$ $S \to TZ$

$R \to paXb$ $\qquad$ $T \to qaYd$

можно вывести, что порождающее правило $S \to RY$ следует применять при разборе сверху вниз (если, для R нет других порождающих правил), только когда предварительно просматриваемый символ – p. Аналогично, порождающее правило $S \to TZ$ рекомендуется в тех случаях, когда таким символом является q.

Определение. *Символом-предшественником* называется символ $a \in S(A) \Leftrightarrow$

$A \Rightarrow +ax$, где A – нетерминальный символ, x – строка терминалов и/или нетерминалов; $S(A)$ – множество символов предшественников A.

Пример: В грамматике с порождающими правилами

$P \to Ac$ $\qquad$ $P \to Bd$

$A \to a$ $\qquad$ $A \to aA$

$B \to b$ $\qquad$ $B \to bB$

a и b - символы предшественники P.

Определение. Множество *символов-предшественников* $S(x)$ для строки терминалов и/или нетерминалов определяется как $S(x) = \{a \mid x \Rightarrow +ay\}$. Здесь x и y есть строки терминалов и/или нетерминалов (y может быть и пустой строкой).

Необходимым условием для того, чтобы грамматика обладала признаками $LL(1)$, является непересекаемость множеств символов – предшественников для альтернативных правых сторон порождающих правил.

Следует проявлять осторожность в тех случаях, когда нетерминал в начале правой части может генерировать пустую строку (далее пустую строку будем обозначать ε).

Пример: Для грамматики

$P \to AB$ $\qquad$ $P \to BG$

$A \to aA$ $\qquad$ $A \to \varepsilon$

$B \to c$ $\qquad$ $B \to bB$

имеем $S(AB) = \{a, b, c\}$, поскольку A может генерировать пустую строку, и $S(BG) = \{b, c\}$. Следовательно, грамматика не будет $LL(1)$.

Рассмотрим грамматику с порождающими правилами:

$T \to AB$

$A \to PQ$ $\qquad$ $A \to BC$

$P \to pP$ $\qquad$ $P \to \varepsilon$

$Q \to qQ$ $\qquad$ $Q \to \varepsilon$

$B \to bB$ $\qquad$ $B \to e$

$C \to cC$ $C \to f,$

которая дает $S(PQ) = \{p, q\}$, $S(BC) = \{b, e\}$.

Однако так как PQ может генерировать пустую сторону, следующим просматриваемым символом при применении порождающего правила $A \to PQ$ может быть b или e (вероятные символы, следующие за A), и одного следующего просматриваемого символа недостаточно, чтобы различить две альтернативные правые части для A. Последнее связано с тем, что b и e являются также символами предшественниками для BC.

Определение. Если A может генерировать пустую строку, то его *направляющими символами* будут

$DS(A) = S(A) + F(A)$, где $F(A)$ – все символы, следующие за A.

В более общем случае, когда существует правило вывода $P \to x$, имеем

$DS(P, x) = \{a \mid a \in S(x)$ или $a \in F(P)$, если $x \to \varepsilon\}$,

где $F(P)$ есть множество символов, которые могут следовать за P.

Так, в приведенном ранее примере множества направляющих символов для альтернативных правил

$DS(A, PQ) = \{p, q, b, e\}$,

$DS(A, BC) = \{b, e\}$.

Поскольку указанные множества пересекаются, данная грамматика не может служить основой для детерминированного нисходящего анализатора, использующего один предварительно просматриваемый символ для различения альтернативных правых частей.

Определение. Грамматику называют *LL(1)-грамматикой*, если для каждого нетерминала, появляющегося в левой части более одного порождающего правила, множества направляющих символов, соответствующих правым частям альтернативных порождающих правил, непересекающиеся.

Все $LL(1)$-грамматики можно разбирать детерминированно сверху вниз.

Существует алгоритм, который позволяет выяснить, представляет ли собой заданная грамматика *LL*(1)-грамматику. Этот алгоритм состоит из следующих шагов:

Шаг 1. Формирование массива пустых строк.

Шаг 2. Формирование матрицы предшественников.

Шаг 3. Формирование матрицы следования.

Рассмотрим подробно содержание каждого шага.

Шаг 1. Формирование массива пустых строк.

Прежде всего нужно установить, какие нетерминалы могут генерировать пустую строку. Для этого создадим одномерный массив, где каждому нетерминалу соответствует один элемент. Любой элемент массива может принимать одно из трех значений: *YES, NO* или *UNDECIDED*. Просматриваем грамматику столько раз, сколько требуется для того, чтобы каждый элемент принял значение *YES* или *NO*.

При первом просмотре исключаются все правила, содержащие терминалы. Если это приводит к исключению всех правил для какого-либо нетерминала, соответствующему элементу массива присваиваем значение *NO*.

Затем для каждого порождающего правила с ε в правой части этому элементу массива, который соответствует нетерминалу в левой части, присваиваем значение *YES*, и все порождающие правила для этого нетерминала исключаются из грамматики.

Если требуются дополнительные просмотры (т.е. значения некоторых элементов массива имеют все еще значение *UNDECIDED*), выполняются следующие действия:

1. Каждое порождающее правило, имеющее такой символ в правой части, который не может генерировать пустую строку (о чем свидетельствуют значения соответствующего массива), исключается из грамматики. В том случае, когда для нетерминала в левой части исключенного правила не существует других порождающих правил, значение элемента массива, соответствующего этому

нетерминалу, устанавливается *NO*.

2. Каждый нетерминал в правой части порождающего правила, который может генерировать пустую строку, стирается из правила. В том случае, когда правая часть правила становится пустой, элементу массива, соответствующему нетерминалу в левой части, присваивается значение *YES*, и все порождающие правила для этого нетерминала исключаются из грамматики.

Этот процесс продолжается до тех пор, пока за полный просмотр грамматики не изменится ни одно из значений элементов массива. Если допустить, что вначале грамматика была «чистой» (т. е. все нетерминалы могли генерировать конечные или пустые строки), то теперь все элементы массива будут установлены на значения *YES* или *NO*.

Пример. Рассмотрим грамматику

$A \to XYZ$

$X \to PQ$

$Y \to RS$

$R \to TU$

$P \to \varepsilon$ $P \to a$

$Q \to aa$ $Q \to \varepsilon$

$S \to cc$

$T \to dd$

$U \to ee$

$Z \to \varepsilon$.

После первого прохода массив пустых строк имеет вид

A	X	Y	R	P	Q	S	T	U	Z
U	U	U	U	Y	Y	N	N	N	Y

U – *UNDECIDED* (нерешенный), Y – *YES* (да), N – *NO* (нет).

Грамматика сведется к следующей

$A \to XYZ$

$X \to PQ$

$Y \to RS$

$R \to TU.$

После второго прохода массив и грамматика имеют вид

A	X	Y	R	P	Q	S	T	U	Z
U	Y	N	N	Y	Y	N	N	N	Y

$A \to XYZ$

Третий проход завершается заполнением массива

A	X	Y	R	P	Q	S	T	U	Z
N	Y	N	N	Y	Y	N	N	N	Y

На этом выполнение первого шага завершено.■

Шаг 2. Формирование матрицы предшественников.

В матрице предшественников для каждого нетерминала отводится строка, а для каждого терминала и нетерминала – столбец.

Сначала выявляем всех *непосредственных* предшественников каждого нетерминала. Этот термин используется для обозначения тех символов, которые из одного порождающего правила уже видны как предшественники. Например, на основании правил $P \to QR$ и $Q \to qV$ можно заключить, что Q есть непосредственный предшественник P , а q – непосредственный предшественник Q.

Если нетерминал A, например, имеет в качестве непосредственных предшественников B и C, в A-ю строку в B-м и C-м столбцах помещаются единицы.

Там, где правая часть правила начинается с нетерминала, необходимо проверить, может ли данный нетерминал генерировать пустую строку, для чего используется массив пустых строк. Если такая генерация возможна, символ, следующий за нетерминалом (при их наличии), является непосредственным предшественником нетерминала в левой части правила и т. д. Как только

непосредственные предшественники будут введены в матрицу, мы сможем делать следующие заключения.

Из порождающих правил $P \to QR$, $Q \to qV$ можно заключить, что q есть символ (не непосредственный) предшественник P**,** нужно поместить единицу в P -ю строку q-го столбца. В обобщенном варианте, когда в (i, j)-ой позиции и в (j, k)-ой позиции стоят единицы, нам следует поставить единицу и в (i, k)-ую позицию. Процесс следует повторять до тех пор, пока не будет таких случаев, когда в (i,j)-й позиции и в (j,k)-й позиции появляются единицы, а в (i,k)-й позиции – нет.

Пример. Рассмотрим множество порождающих правил:

$A \to BX$

$B \to XY$

$X \to aa$

Окончательный вид матрицы предшественников показан на рисунке 7. В матрице не непосредственные предшественники заключены в кружки.

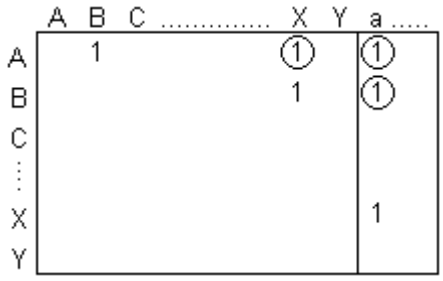


Рис. 7. Матрица символов-предшественников.

Описанный процесс формирования матрицы предшественников известен как нахождение транзитивного замыкания матрицы и у него известен аналог в теории графов.■

Шаг 3. Формирование матрицы следования.

Если бы не проблема нетерминалов, генерирующих пустую строку, для *LL*(1)-проверки потребовалось бы только изучение матрицы предшествования. Однако при вычислении направляющих символов иногда приходится

рассматривать символы, которые по праву могут следовать за определенным нетерминалом. Поэтому строится матрица следования.

Пример. Из правила $S \to ABC$ вытекает, что B может следовать за A, и элемент (A,B) в матрице следования должен иметь значение 1. Аналогичным образом, C может следовать за B, и элементу (B,C) устанавливается значение 1. Если B генерирует пустой символ, то естественно заключить, что C может следовать за A, и элемент $(A,C)= 1$.

Менее явно правила $P \to QR$, $Q \to VU$ позволяют сделать вывод о том, что R следует за U или, в более общем виде (на основании второго правила), что «любой символ, следующий за Q, следует и за U». Заметим, что если, например, B следует за A и b есть предшественник B , то b следует за A. Таким образом, матрицу предшествования можно использовать для того, чтобы вывести больше символов-следователей. ■

На основании массива пустых строк, матрицы предшествования и матрицы следования можно проверить, удовлетворяет ли грамматика признаку $LL(1)$. Там, где в левой части более чем одного правила появляется нетерминал, необходимо вычислять направляющие символы различных альтернативных правых частей. Если хотя бы для одного нетерминала множества направляющих символов для альтернативных правил пересекаются, грамматика *не* $LL(1)$. В противном случае, она будет $LL(1)$.

Ранее был показан алгоритм для определения признака $LL(1)$-грамматики. В этой связи возникает два вопроса:

1. Все ли языки обладают $LL(1)$-грамматикой?

2. Если нет, то существует ли алгоритм для определения свойства LL(1)-языка (т. е. можно ли его генерировать с помощью LL(1)-грамматики или нет)?

Ответ на первый вопрос отрицательный, ответ на второй вопрос тоже отрицательный. О проблеме говорят, что она неразрешима. Из истории известно, что такого алгоритма не существует (по крайней мере, алгоритма, который с гарантией сработал бы в любом случае). Все попытки получить подобный

алгоритм могут привести к бесконечному зацикливанию в определенных ситуациях.

Насколько важен этот результат на практике? Оказывается, что «очевидной» грамматикой для большинства языков программирования является не *LL*(1)-грамматика. Однако обычно очень большое число КС средств языка программирования можно представить с помощью *LL*(1)-грамматики. Проблема заключается в том, чтобы, имея грамматику, которая не обладает признаком *LL*(1), найти эквивалентную ей *LL*(1)-грамматику. Так как алгоритм, позволяющий определить существует ли такая *LL*(1)- грамматика или нет, отсутствует, значит отсутствует и алгоритм, который мог бы определить существуют ли соответствующие преобразования, и выполнить их. Многие разработчики компиляторов, обладающие достаточным опытом, не испытывают затруднений в преобразовании грамматик вручную с целью устранения их свойств, отличных от *LL*(1). Тем не менее, с ручным преобразованием связаны серьезные опасения. Основное из них заключается в том, что человек, выполняющий это преобразование, может случайно изменить язык, генерируемый данной грамматикой. Поэтому по мере возможности следует избегать преобразований вручную для того, чтобы создать надежный компилятор.

Однако отсутствие общего решения проблемы вовсе не означает невозможности ее решения для частных случаев. Прежде чем остановиться на этом вопросе, посмотрим, что требуется для преобразования грамматики в *LL*(1)-форму.

1. Устранение левой рекурсии.

Грамматика, содержащая левую рекурсию, не является *LL*(1)-грамматикой.

Рассмотрим правила

$A \to Aa$ $\qquad\qquad$ $A \to a$

(левая рекурсия в A)

Здесь a есть символ-предшественник для обоих вариантов нетерминала A.

Аналогично грамматика, содержащая левый рекурсивный цикл, не может быть *LL*(1)-грамматикой. Например:

$A \to BC$

$B \to CD$

$C \to AE$

Можно показать, что грамматику, содержащую левый рекурсивный цикл, нетрудно преобразовать в грамматику, содержащую только прямую левую рекурсию. Далее, за счет введения дополнительных нетерминалов, левую рекурсию можно исключить полностью (в действительности она заменяется правой рекурсией, которая не представляет проблемы в отношении *LL*(1)-свойства).

Пример. Рассмотрим грамматику

$S \to Aa$

$A \to Bb$

$B \to Cc$

$C \to Dd$

$C \to e$

$D \to Az,$

которая имеет левый рекурсивный цикл, вовлекающий *A, B, C, D*. Чтобы заменить этот цикл на прямую левую рекурсию, можно действовать следующим образом.

Упорядочим нетерминалы, а именно: *S, A, B, C, D*. Рассмотрим все правила вида $X_i \to X_j\gamma$, где X_i и X_j – нетерминалы, а γ – строка терминальных и нетерминальных символов. В отношении правил, для которых $j \geq i$ никакие действия не производятся. Однако это условие не выдерживается для всех правил, если есть левый рекурсивный цикл. При выбранном порядке в правиле $D \to Az$ A предшествует D в этом упорядочении. Теперь начнем замещать A, пользуясь всеми правилами, имеющими A в левой части. В результате получим

$D \to Bbz$

Поскольку *B* предшествует *D* в упорядочении, процесс повторяется, получаем правило

$D \to Ccbz$

Затем он повторяется еще раз, получается два правила

$D \to ecbz \qquad D \to Ddcbz.$

Теперь преобразованная грамматика имеет вид:

$S \to Aa$

$A \to Bb$

$B \to Cc$

$C \to Dd \qquad C \to e$

$D \to ecbz \qquad D \to Ddcbz$

Все порождающие правила имеют требуемый вид, а левый рекурсивный цикл заменен прямой левой рекурсией.

Чтобы исключить прямую левую рекурсию, введем новый нетерминальный символ Z и заменим правила

$D \to ecbz \qquad D \to Ddcbz$

на

$D \to ecbZ \qquad D \to ecbzZ$

$Z \to dcbz \qquad Z \to dcbzZ$

Заметим, что до и после преобразования D генерирует регулярное выражение $(ecbz)(dcbz)^*$.

Обобщая, можно показать, что если нетерминал A появляется в левых частях $r+s$ порождающих правил, r из которых используют прямую левую рекурсию, а s – нет, то есть

$A \to A\alpha_1, \quad A \to A\alpha_2, \quad ..., \quad A \to A\alpha_r$

$B \to \beta_1, \quad B \to \beta_2, \quad ..., \quad B \to \beta_s$

то эти правила можно заменить на следующие

$$\left.\begin{array}{l} A \to \beta_i \\ A \to \beta_i Z \end{array}\right\} \quad 1 \le i \le s \;, \quad \left.\begin{array}{l} Z \to \alpha_i \\ Z \to \alpha_i Z \end{array}\right\} \quad 1 \le i \le r$$

Неформальное доказательство заключается в том, что до и после преобразования A генерирует регулярное выражение

$$(\beta_1 \,|\, \beta_2 \,|\; \ldots \;|\, \beta_s)(\alpha_1 \,|\, \alpha_2 \,|\; \ldots \;|\, \alpha_r)^*$$

Левую рекурсию всегда можно исключить из грамматики и нетрудно

написать программу для общего случая

Во многих ситуациях грамматики, не обладающие признаком $LL(1)$, можно преобразовать в $LL(1)$-грамматику с помощью процесса **факторизации** [1, 5].

Пример. Порождающие правила

$S \to aSb$

$S \to aSc$

$S \to \varepsilon$

можно преобразовать путем факторизации в правила

$S \to aSX$

$S \to \varepsilon$

$X \to b$

$X \to c$

В результате получена $LL(1)$-грамматика.

Процесс факторизации нельзя автоматизировать, распространив его на общий случай. Иначе это противоречило бы неразрешимости проблемы определения наличия признака $LL(1)$ у языка [9].

Что же делать разработчику, если его преобразователь грамматики не может выдать $LL(1)$-грамматику? В этом случае он должен попытаться определить на основании выхода преобразователя, какая часть грамматики не поддается преобразованию, и либо преобразовать эту часть вручную в $LL(1)$-форму, либо переписать ее так, чтобы преобразователь грамматики смог ее преобразовать. Это возможно при условии, что используется $LL(1)$-язык. Если данное условие не выполняется, то синтаксический анализатор $LL(1)$ нельзя получить, не изменив реализуемый язык. Выход преобразователя может указать, почему язык не является $LL(1)$-языком.

Рассмотрим еще несколько способов преобразования грамматики [1, 3, 5].

1. Лучший способ избавиться от прямой левосторонней рекурсии – использовать фигурные и круглые скобки в качестве метасимволов.

Пример.

Правило $E \to E+T \mid T$, обозначающее по крайней мере одно вхождение T, за

которым следует сколь угодно (в том числе и нуль) вхождений $+T$, можно переписать как $E \to T\{+T\}$.

Правило $E \to T \mid E+T \mid E–T$ можно записать как $E \to T\{ (+ \mid –) T\}$, и правило $T \to T*F \mid T / F \mid F$ может иметь вид $T \to F \{ * F \mid / F \}$.

Помимо того, что факторизация позволяет исключить прямую рекурсию, использование этого приема сокращает размеры грамматики и позволяет проводить разбор более эффективно. После факторизации в грамматике останется не более одной правой части с прямой левосторонней рекурсией для каждого из нетерминалов. Если такая правая часть есть, делаем следующее:

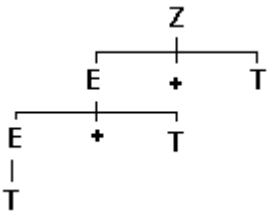
Пример 3. Пусть $U \to x \mid y \mid ... \mid z \mid Uv$ – правила, у которых осталась леворекурсивная правая часть. Эти правила означают, что членом синтаксического класса U являются x, y или z, за которыми либо ничего не следует, либо следует сколько-то v.

Преобразуем эти правила к виду $U \to (x \mid y \mid \mid z)\{v\}$.

Пример. Дерево, использующее рекурсию.

$Z \to E$

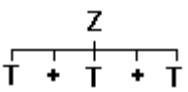
$E \to E+T \mid T$



Пример. Дерево, использующее итерацию

$Z \to E$

$E \to T\{+T\}$



3.1.4. LL(1) – таблица разбора

Найдя *LL*(1)-грамматику для языка, можно перейти к следующему этапу – применению найденной грамматики для приведения в действие в компиляторе фазы разбора [4]. Этот этап аналогичен рекурсивному спуску, только здесь исключаются многочисленные вызовы процедур, благодаря представлению грамматики в табличном виде (таблицы разбора) и использованию не зависящего от исходного языка модуля компилятора для проведения по таблице во время чтения исходного текста. Модуль компилятора, который будем называть драйвером, всегда указывает на то место в синтаксисе, которое соответствует текущему входному символу. Драйверу требуется стек для запоминания адресов возврата всякий раз, когда он входит в новое порождающее правило, соответствующее какому-либо нетерминалу. Представление синтаксиса должно быть таким, чтобы обеспечить эффективный разбор в смысле скорости работы.

В процессе разбора осуществляется последовательность шагов:

1. Проверка предварительно просматриваемого символа с тем, чтобы выяснить, не является ли он направляющим для какой-либо конкретной правой части порождающего правила. Если этот символ не направляющий и имеется альтернативная правая часть правила, то она проверяется на следующем этапе. В особом случае, когда правая часть начинается с терминала, множество направляющих символов состоит только из одного этого терминала.

2. Проверка терминала, появляющегося в правой части порождающего правила.

3. Проверка нетерминала. Она заключается в проверке нахождения предварительного просматриваемого символа в одном из множеств направляющих символов для данного нетерминала, помещении в стек адреса возврата и переходу к первому правилу, относящемуся к этому нетерминалу. Если нетерминал появляется в конце правой части правила, то нет необходимости помещать в стек адреса возврата.

Таким образом, в таблицу разбора включаются по одному элементу на каждое правило грамматики и на каждый экземпляр терминала или нетерминала

правой части правила. Кроме того, в таблице будут находиться элементы на каждую реализацию пустой строки в правой части правила (по одному на каждую реализацию).

Драйвер содержит цикл процедуры, тело которой обрабатывает элемент таблицы разбора и определяет следующий элемент для обработки. Поле перехода обычно дает следующий элемент обработки, если значение поля возврата не окажется истиной. В последнем случае адрес следующего элемента берется из стека (что соответствует концу правила). Если же предварительно просматриваемый символ отсутствует в списке терминалов и значение поля ошибки окажется ложью, нужно обрабатывать следующий элемент таблицы с тем же предварительно просматриваемым символом (способ обращения с альтернативными правыми частями правил).

Рассмотрим грамматику со следующими порождающими правилами:

$PROGRAMM \rightarrow begin\ DECLIST\ comma\ STATELIST\ end$

$DECLIST \rightarrow d\ semi\ DECLIST$

$DECLIST \rightarrow d$

$STATELIST \rightarrow s\ semi\ STATELIST$

$STATELIST \rightarrow s$

Заданная грамматика не является $LL(1)$-грамматикой, альтернативные правила имеют правые части, начинающиеся одинаковыми терминалами: d и s.

Видоизменим грамматику: введем нетерминалы X и Y.

$PROGRAMM \rightarrow begin\ DECLIST\ comma\ STATELIST\ end$

$DECLIST \rightarrow dX$

$X \rightarrow semi\ DECLIST$

$X \rightarrow \varepsilon$

$STATELIST \rightarrow sY$

$Y \rightarrow semi\ STATELIST$

$Y \rightarrow \varepsilon$

С целью проверки признака $LL(1)$ этой грамматики образуем различные таблицы:

Массив пустых строк

PROGRAMM	*DECLIST*	*STATELIST*	*X*	*Y*
N	*N*	*N*	*Y*	*Y*

Полная матрица предшествования

	PROGRAMM	*DECLIST*	*STATELIST*	*X*	*Y*	*begin*	*d*	*s*	*comma*	*semi*	*end*
PROGRAMM						1					
DECLIST							1				
STATELIST								1			
X										1	
Y										1	

Матрица следования

	PROGRAMM	*DECLIST*	*STATELIST*	*X*	*Y*	*begin*	*d*	*s*	*comma*	*semi*	*end*
PROGRAMM											
DECLIST									1		
STATELIST											1
X									1		
Y											1

Первый вариант для *X* имеет множество направляющих символов {*semi*}. Второй вариант – пустая строка, поэтому нужно выяснить, что следует за *X*.

Единственным символом-следователем служит *comma*, так что множеством направляющих символов будет {*comma*}. Множества направляющих символов являются непересекающимися, следовательно, нарушения условия *LL*(1) в отношении порождающих правил для нетерминала *X* нет. Аналогичным образом можем показать, что множествами направляющих символов для *Y* будут {*semi*} и {*end*}.

Поэтому имеем дело с *LL*(1)-грамматикой.

Представим грамматику в виде схемы, показанной на рисунке 8.

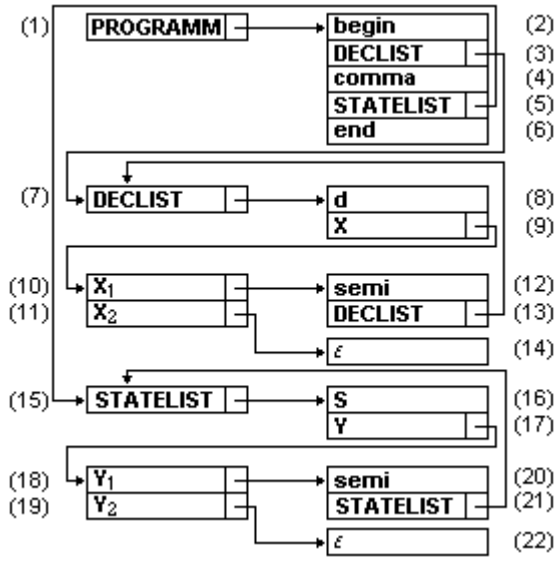


Рис. 8. Схема разбора грамматики.

В скобках слева и справа на рис.8 указаны номера соответствующих элементов таблицы разбора. На основании этой схемы можно построить таблицу разбора (таблица 1).Ноль появляется в поле перехода, когда оно не имеет отношения к делу.

Таблица 1. Таблица разбора грамматики

Номер элемента	*Terminals*	*jump*	*accept*	*stack*	*return*	*error*
1	*{begin}*	2	*false*	*false*	*false*	*true*
2	*{begin}*	3	*true*	*false*	*false*	*true*

3	*{d}*	7	*false*	*true*	*false*	*true*
4	*{comma}*	5	*true*	*false*	*false*	*true*
5	*{s}*	15	*false*	*true*	*false*	*true*
6	*{end}*	0	*true*	*false*	*true*	*true*
7	*{d}*	8	*false*	*false*	*false*	*true*
8	*{d}*	9	*true*	*false*	*false*	*true*
9	*{semi, comma}*	10	*false*	*false*	*false*	*true*
10	*{semi}*	12	*false*	*false*	*false*	*false*
11	*{comma}*	14	*false*	*false*	*false*	*true*
12	*{semi}*	13	*true*	*false*	*false*	*true*
13	*{d}*	7	*false*	*false*	*false*	*true*
14	*{comma}*	0	*false*	*false*	*true*	*true*
15	*{s}*	16	*false*	*false*	*false*	*true*
16	*{s}*	17	*true*	*false*	*false*	*true*
17	*{semi, end}*	18	*false*	*false*	*false*	*true*
18	*{semi}*	20	*false*	*false*	*false*	*false*
19	*{end}*	22	*false*	*false*	*false*	*true*
20	*{semi}*	21	*true*	*false*	*false*	*true*
21	*{s}*	15	*false*	*false*	*false*	*true*
22	*{end}*	0	*false*	*false*	*true*	*true*

Рассмотрим предложение (со следующим символом "⊥")

begin d semi d comma s semi s end ⊥

Пример разбора оформим в виде таблицы 2.

Таблица 2. Пример разбора предложения.

i - номер элемента	Действие	Стек разбора

1	*begin* считывается и проверяется; перейти к *pt*[2]	 0
2	*begin* проверяется и принимается; перейти к *pt*[3]	 0
3	*d* считывается и проверяется; 4 помещается в стек, перейти к *pt*[7]	4 0
7	*d* проверяется, перейти к *pt*[8]	4 0
8	*d* проверяется и принимается; перейти к *pt*[9]	4 0
9	*semi* считывается и проверяется; перейти к *pt*[10]	4 0
10	*semi* проверяется, перейти к *pt*[12]	4 0
12	*semi* проверяется и принимается; перейти к *pt*[13]	4 0
13	*d* считывается и проверяется; перейти к *pt*[7]	4 0
7	*d* проверяется, перейти к *pt*[8]	4 0
8	*d* проверяется и принимается; перейти к *pt*[9]	4 0
9	*comma* считывается и проверяется; перейти к *pt*[10]	4 0
10	*comma* не совпадает с *semi*; ошибка = ложь; перейти к *pt*[11]	4 0
11	*comma* проверяется, перейти к *pt*[14];	4 0
14	*comma* проверяется, возврат= истина,	

	4 извлекаем из стека, перейти к *pt*[4];	0
4	*comma* проверяется и принимается, перейти к *pt*[5];	 0
5	*s* считывается и проверяется, 6 помещается в стек, перейти к *pt*[15]	6 0
15	*s* проверяется, перейти к *pt*[16]	6 0
16	*s* проверяется и принимается; перейти к *pt*[17]	6 0
17	*semi* считывается и проверяется; перейти к *pt*[18]	6 0
18	*semi* проверяется, перейти к *pt*[20]	6 0
20	*semi* проверяется и принимается; перейти к *pt*[21]	6 0
21	*s* считывается и проверяется; перейти к *pt*[15]	6 0
15	*s* проверяется, перейти к *pt*[16]	6 0
16	*s* проверяется и принимается; перейти к *pt*[17]	6 0
17	*end* считывается и проверяется; перейти к *pt*[18]	6 0
18	*end* не совпадает с *semi*, ошибка=ложь, перейти к *pt*[19]	6 0
19	*end* проверяется, перейти к *pt*[22]	6 0
22	*end* проверяется, возврат=истина, 6 извлекаем из стека, перейти к *pt*[6]	 0

6	*end* проверяется и принимается, возврат=истина, 0 извлекаем из стека, i:=0	
0	разбор заканчивается	

При разборе некоторые терминалы проверяются несколько раз. Такой неоднократной проверки можно избежать, если разработчики согласны отложить обнаружение некоторых синтаксических ошибок на более поздний этап. Можно также сократить число элементов в таблице разбора.

При наличии *LL*(1)-грамматики можно написать программу для получения соответствующей таблицы разбора. Используемый при этом алгоритм имеет много общего с проверкой признака *LL*(1) и их часто объединяют.

LL(1)-метод разбора имеет ряд преимуществ:

1. Никогда не требуется возврат, поскольку этот метод детерминированный.

2. Время разбора (приблизительно) пропорционально длине программы.

3. Имеются хорошие диагностические характеристики и существует возможность исправления ошибок, так как синтаксические ошибки распознаются по первому неприемлемому символу, а в таблице разбора есть список возможных символов продолжения.

4. Таблицы разбора меньше, чем соответствующие таблицы в других методах разбора.

5. *LL*(1)-разбор применим к широкому классу языков – всех языков, имеющих *LL*(1)-грамматики. Следует заметить, что в большинстве случаев грамматику для языка программирования приходится преобразовывать к *LL*(1)-грамматике.

3.2. ВОСХОДЯЩИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ

3.2.1. Общие принципы синтаксического анализа снизу вверх

Метод восходящего синтаксического анализа (снизу вверх) состоит в том,

что, отталкиваясь от заданной цепочки, пытаются привести ее к начальному символу (аксиоме грамматики), т.е. восходя от листьев, построить дерево синтаксического анализа [2, 3, 4].

Рассмотрим общие принципы работы детерминированного восходящего синтаксического анализатора. Для заданной входной цепочки восходящий МП-автомат моделирует ее правый вывод в обратном порядке, т.е. правила грамматики применяются справа налево, и манипулирует своими магазинными и текущими входными символами с помощью операций переноса и свертки.

Операция переноса — это операция, которая вталкивает в магазин некоторый символ, соответствующий текущему входному символу, и сдвигает вход, т.е. переносит с входа в магазин текущий символ.

Операция свертки для некоторого правила p с r символами в правой части – это операция, которая выбирается только тогда, когда r верхних символов магазина представляют правую часть правила p. Операция свертки выталкивает из магазина верхние r символов и вталкивает туда символ, соответствующий левой части правила p.

Рассмотрение восходящих методов синтаксического анализа начнем с наиболее простых методов, основанных на предшествовании [4, 5].

3.2.2. Методы, основанные на предшествовании

Рассмотрим три табличных метода синтаксического анализа, основанные на предшествовании: метод (простого) предшествования, метод расширенного предшествования и метод операторного предшествования.

Определение. Пусть $G[Z]$ – грамматика, $w=xuy$ – сентенциальная форма. Тогда u называется *фразой сентенциальной формы* w для нетерминального символа U, если $Z \Rightarrow^* xUy$ и $U \Rightarrow^+ u$; и называется *простой фразой*, если $Z \Rightarrow^* xUy$ и $U \Rightarrow u$.

При использовании восходящего метода в текущей сентенциальной форме повторяется поиск основы (самой левой простой фразы u), которая в соответствии с правилом $U::=u$ приводится к нетерминалу U. При применении любого из

методов разбора возникает вопрос, как найти основу и к какому нетерминалу ее приводить? Этот вопрос будем решать для определенного класса грамматик, называемых грамматиками, основанными на предшествовании [2, 5].

Обобщая ранние работы по синтаксическому анализу, Вирт и Вебер предложили метод предшествования, основанный на том факте, что между любыми двумя соседними символами s_i и s_{i+1} приводимой строки может существовать лишь три отношения:

1) $s_i \dot{<} s_{i+1}$, если символ s_{i+1} самый левый символ некоторой основы: $s_i \underbrace{s_{i+1}\ldots\ldots}_{основа}$

2) $s_i \dot{>} s_{i+1}$, если s_i – самый правый символ основы: $\underbrace{\ldots\ldots s_i}_{основа} s_{i+1}$

3) $s_i \doteq s_{i+1}$, если символы s_i и s_{i+1} принадлежат одной основе: $\underbrace{\ldots\ldots s_i s_{i+1}\ldots\ldots}_{основа}$

Отношения $\dot{<}$, $\dot{>}$, $\doteq$ называют *отношениями предшествования*.

Отношения предшествования зависят от порядка, в котором стоят символы: из $s_i \dot{<} s_j$ отнюдь не следует $s_j \dot{>} s_i$, а из $s_i \doteq s_j$ вовсе не вытекает $s_j \doteq s_i$.

Если для каждой упорядоченной пары символов грамматики существует не более чем одно отношение предшествования, то на каждом шаге синтаксического анализа можно легко выделить основу. Основой является самая левая группа символов приводимой строки $s_i \ldots\ldots s_j$, связанная отношениями предшествования вида $s_{i-1} \dot{<} s_i \doteq s_{i+1} \doteq \ldots \doteq s_j \dot{>} s_{j+1}$.

Определение. Пусть *U, C, D* – нетерминальные символы; *x, y, w, z* – любые строки, возможно пустые. Грамматикой *простого предшествования* называют такую грамматику типа 2 по Хомскому, в которой:

1) для каждой упорядоченной пары терминальных и нетерминальных символов выполняется не более чем одно из трех отношений предшествования, определяемых следующим образом:

А. $s_i \doteq s_j$, если и только если существует правило $U \to x s_i s_j y$.

Б. $s_i \doteq s_j$, если и только если существует правило $U \to xs_iDy$ и вывод $D \Rightarrow^* s_jz$.

В. $s_i \dot{>} s_j$, если и только если существует правило $U \to xCs_jy$ и вывод $C \Rightarrow^* zs_i$ или правило $U \to xCDy$ и выводы $C \Rightarrow^* zs_i$, $D \Rightarrow^* s_jw$.

2) разные порождающие правила имеют различные правые части.

Первое условие обеспечивает единственность отношения предшествования для каждой упорядоченной пары соседних символов в приводимой строке. Если между какими-либо двумя символами нет отношения предшествования, то это означает, что они не могут находиться рядом ни в одном элементе разбора синтаксически правильной сентенциальной формы.

Второе условие обеспечивает однозначность редукции.

Основную идею алгоритма разбора, основанного на отношениях предшествования, можно описать так. Символы входной строки поочередно переписываются в стек до тех пор, пока между символом в вершине стека (обозначим его s_l) и очередным символом входной строки (обозначим его s_i) не появится отношение $\dot{>}$, т. е. окажется, что $s_l \dot{>} s_i$. Тогда стек просматривается в направлении от вершины к началу до тех пор, пока между двумя очередными символами s_{k-1} и s_k не появится отношение $\dot{<}$, т.е. $s_{k-1} \dot{<} s_k$.

Часть стека от символа s_k до символа в вершине s_l – основа. Теперь среди порождающих правил отыскивается правило $U \to s_k \dots s_l$ и если нужно, вызывается семантическая программа, которая обрабатывает строку $s_k \dots s_l$, переводя ее на промежуточный или выходной язык. После этого в стеке строка $s_k \dots s_l$ заменяется символом U, и описанный выше процесс заполнения стека символами входной строки продолжается (рис.9).

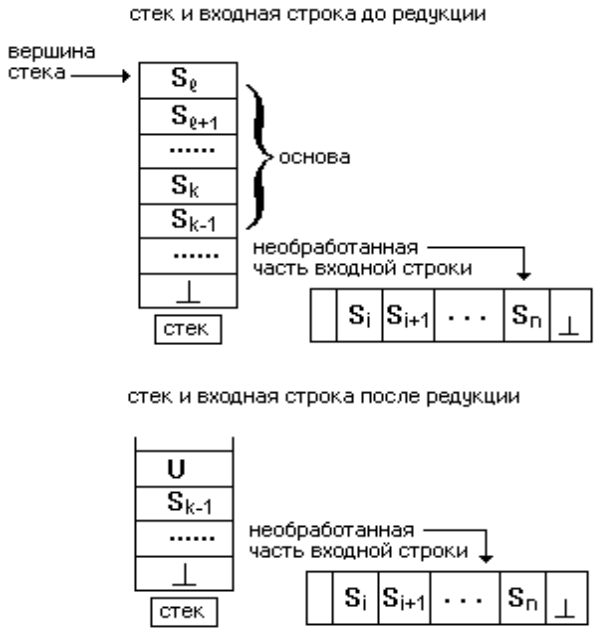


Рис.9. Схема распознавателя, основанного на простом предшествовании.

Если на некотором шаге процесса обнаружится, что между соседними символами s_p и s_q не существует отношения предшествования, то это свидетельствует об ошибке типа "символы s_p и s_q не могут находиться рядом". Соответствующая информация заносится в файл ошибок.

Описанный алгоритм прост и обеспечивает полный синтаксический контроль транслируемой программы.

Отношения предшествования удобно записывать в виде матрицы предшествования, представляющей собой таблицу с двумя входами. Входами в таблицу является предшествующий и последующий символы приводимой строки, а в ее клетках записываются отношения предшествования.

Пример. Задана грамматика $G_1[Z]$:

Z ::= bMb

M ::= (L | a

L ::= Ma)

Матрица предшествования имеет вид:

	Z	*b*	*M*	*L*	*a*	*(*	*)*
Z							
b			$\doteq$		$\dot{<}$	$\dot{<}$	
M		$\doteq$			$\doteq$		
L		$\dot{>}$			$\dot{>}$		
a		$\dot{>}$			$\dot{>}$		$\doteq$
(			$\dot{<}$	$\doteq$	$\dot{<}$	$\dot{<}$	
)		$\dot{>}$			$\dot{>}$		

Пустой элемент матрицы свидетельствует о том, что между двумя символами отношение предшествования не определено.

Отношение предшествования можно найти непосредственно, пользуясь ранее приведенными определениями. Однако можно ввести дополнительные определения, которые позволят сделать процесс построения матрицы предшествования более формализованным [9].

Практическое применение метода предшествования затрудняется неоднозначностью отношений предшествования, которая часто встречается в грамматиках реальных языков программирования. Общего алгоритма приведения грамматики к грамматике предшествования нет, однако существуют алгоритмы, позволяющие в большинстве случаев устранить конфликты предшествования. Недостаток этих алгоритмов состоит в том, что они обычно не гарантируют сохранение единственности правых частей порождающих правил модифицированной грамматики.

Кроме того, после модификации грамматика, как правило, сильно расширяется и становится непригодной для использования программистом. По этим причинам метод предшествования не нашел применения в чистом виде, на практике иногда используют **расширенное предшествование**, сущность которого состоит в использовании контекста для устранения неоднозначности отношений предшествования.

Основная идея **расширенного предшествования** состоит в следующем. Пусть между символами s_j и s_{j+1} существуют отношения $s_j \dot{<} s_{j+1}$ и $s_j \doteq s_{j+1}$, и требуется определить, какое именно отношение нужно взять в контексте

$$\ldots s_j\ s_{j+1}\ s_{j+2} \ldots .$$

Если справедливо первое отношение $\dot{<}$ и грамматика однозначна, то имеется порождающее правило, правая часть которого начинается символами $s_{j+1}s_{j+2}$. Если такого правила нет, то нужно взять второе отношение.

Аналогичные рассуждения можно привести и в других случаях неоднозначности.

В соответствии с этой общей идеей для устранения неоднозначности отношений предшествования между парами символов используются *тройки символов*.

Пример.

$$\ldots s_{k-1} \doteq\dot{<} s_k \doteq s_{k+1} \doteq \ldots . \doteq s_{l-1} \doteq s_l \doteq\dot{>} s_i \ \ldots$$

$$\lfloor \text{ предполагаемая основа } \rfloor$$

На рисунке показан случай, когда на обоих концах предполагаемой основы имеются конфликты предшествования, причем символы $\ldots s_{k-1} \ldots s_l$ находятся в стеке распознавателя, а s_i – очередной символ входной строки. Для выделения правого конца основы заранее заготавливают тройки $s_1 s_2 s_3$ такие, что $s_1 s_2$ – символы, которыми заканчивается правая часть какого-либо порождающего правила, а s_3 – символ, образующий с s_2 конфликтную пару $s_2 \dot{>} \doteq s_3$ или $s_2 \dot{<}\dot{>} s_3$. Если $s_{l-1}s_l s_i = s_1 s_2 s_3$, то справедливо отношение $\dot{>}$, иначе справедливо отношение $\doteq$ или $\dot{<}$.

Для выделения левого конца основы заготавливают тройки $s_1' s_2' s_3'$, такие, что $s_2' s_3'$ – символы, которыми начинается правая часть какого-либо порождающего правила, а s_1' – символ, образующий с s_2' конфликтную пару $s_1' \doteq\dot{<} s_2'$. Если $s_{k-1}s_k s_{k+1} = s_1' s_2' s_3'$, то справедливо отношение $\dot{<}$, иначе справедливо отношение $\doteq$.

Этот метод применим, если использование троек устраняет неоднозначность отношений предшествования. В противном случае нужно

привлечь еще более далекий контекст.

Если конфликты предшествования становятся нормой, то для записи элемента матрицы предшествования уже недостаточно двух бит. Здесь нужно либо записывать каждый элемент матрицы предшествования тремя битами, либо использовать две матрицы с двухбитовыми элементами: одну – для выделения правого конца основы, другую – для левого. В особых случаях нужно заготовить также две таблицы троек:

– таблицу правых троек для устранения неоднозначности на правом конце предполагаемой основы;

– таблицу левых троек для устранения неоднозначности на левом конце.

Если N – число нетерминальных символов грамматики, а T – число терминальных символов, то объем матрицы предшествования для левого конца основы равен $2(N+T)(N+T)$, а для правого конца – $2(N+T)T$, поскольку входная строка состоит только из терминальных символов. Следовательно, общий объем двух двухбитовых матриц равен $2(N+T)(N+2T)$ бит, а объем одной трехбитовой матрицы $3(N+T)(N+T)$.

Отсюда можно заключить, что при $N > T$ выгоднее иметь две двухбитовые матрицы, а при $N < T$ – одну трехбитовую.

Пример. Составим матрицу предшествования и таблицы правых и левых троек для грамматики G_2[П], определенной правилами:

П→⊥В⊥

В→Т

В→В+Т

Т→М

Т→Т*М

М→и

М→(В)

Матрицу предшествования можно построить описанным ранее способом:

s_j / s_i	**В**	**Т**	**М**	⊥	+	*	**и**	**(**	**)**
В				$\doteq$	$\doteq$				$\doteq$
Т				$\dot{>}$	$\dot{>}$	$\doteq$			$\dot{>}$
М				$\dot{>}$	$\dot{>}$	$\dot{>}$			$\dot{>}$
⊥	$\doteq$ $\dot{<}$	$\dot{<}$	$\dot{<}$				$\dot{<}$	$\dot{<}$	
+		$\doteq$ $\dot{<}$	$\dot{<}$				$\dot{<}$	$\dot{<}$	
*			$\doteq$				$\dot{<}$	$\dot{<}$	
и				$\dot{>}$	$\dot{>}$	$\dot{>}$			$\dot{>}$
(	$\doteq$ $\dot{<}$	$\dot{<}$	$\dot{<}$						
)				$\dot{>}$	$\dot{>}$	$\dot{>}$			$\dot{>}$

Дальнейшее решение будем проводить в следующем порядке.

1. Составить таблицу правых троек, для этого необходимо:

а) просмотреть матрицу предшествования и найти пары символов s_2 s_3, для которых существует отношение $\dot{>}$, причем отношения предшествования неоднозначны;

б) для каждой пары, выделенной в пункте а), просмотреть столбец s_2 матрицы предшествования и найти все символы s_1, связанные с символом s_2 отношением $\doteq$;

в) просмотреть порождающие правила. Если существует правило, правая часть которого заканчивается парой s_1s_2 , то тройка $s_1s_2s_3$ должна быть включена в таблицу правых троек.

Однако если одновременно существует правило, в правой части которого символы s_1, s_2 и s_3 стоят рядом, то применение троек не устраняет неоднозначности. Для такой грамматики метод расширенного предшествования неприменим.

В рассматриваемом примере матрица предшествования не содержит

конфликтов предшествования с отношением $\dot{>}$, поэтому таблицу правых троек составлять не нужно.

2. Составить таблицу левых троек, для этого необходимо:

а) просмотреть матрицу предшествования и найти пары символов $s'_1 s'_2$, для которых существует отношение $s'_1 \dot{<} s'_2$, причем отношения предшествования неоднозначны;

б) для каждой пары, выделяемой в пункте а) просмотреть строку s'_2 и найти все символы s'_3, связанные с символом s'_2 отношением $s'_2 \doteq s'_3$;

в) просмотреть порождающие правила. Если существует правило, правая часть которого начинается символами $s'_2 s'_3$, то тройка $s'_1 s'_2 s'_3$ должна быть включена в таблицу левых троек.

Однако, если одновременно существует правило, в правой части которого символы $s'_1 s'_2 s'_3$ стоят рядом, то применение троек в этом случае не устраняет неоднозначность, и метод расширенного предшествования в этом случае неприменим.

В данном примере в пункте а) выявляются три конфликта предшествования:

$\perp \doteq \dot{<}$ **B , +** $\doteq \dot{<}$ **T , (** $\doteq \dot{<}$ **B .**

В соответствии с пунктом б) находим:

B $\doteq \perp$ **, B** $\doteq$ **+ , B** $\doteq$ **)**

а также

T $\doteq$ ***.**

Наконец, в соответствии с пунктом в) обнаруживаем порождающие правила, начинающиеся символами **B+** и **T***, поэтому получаем следующую таблицу левых троек:

Номер тройки	s'_1	s'_2	s'_3
1	$\perp$	**B**	+

2	**(**	**B**	+
3	+	**T**	*

Задача решена.

Пример. Используя распознаватель расширенного предшествования, матрицу предшествования из предыдущего примера и таблицу левых троек, привести к начальному символу строку

⊥*a*+*b***c*⊥

Решение показано в таблице 3.

Таблица 3. Пример разбора предложения методом расширенного предшествования.

Разбор	Примечание
⊥<·**a**·>+**b*****c**⊥	Буква воспринимается как идентификатор – **и**
⊥<·**M**·>+**b*****c**⊥	
⊥<·**T**·>+**b*****c**⊥	
⊥≐<·**B**≐+<·**b**·>***c**⊥	
⊥≐<·**B**≐+<·**M**·>***c**⊥	
⊥≐<·**B**≐+≐<·**T**≐*<·**c**·>⊥	
⊥≐<·**B**≐+≐<·**T**≐*≐**M**·>⊥	Существует тройка +**T***
⊥<·**B**≐+≐**T**·>⊥	1. Нужно принять ≐**T** 2. Существует тройка ⊥**B**+
⊥≐<·**B**≐⊥	Концевой символ прекращает анализ, поскольку существует правило **П**→⊥**B**⊥ – разбор получен.

Метод расширенного предшествования позволяет обойтись без существенных изменений исходной грамматики. Недостатком является большой

объем постоянных таблиц.

Возможен вариант метода расширенного предшествования, в котором определяется только правый конец основы. Для этого используется одна двухбитовая матрица размером $(N+T)T$ и дополнительная таблица правых троек. После появления отношения $\dot{>}$ между верхним символом стека и очередным символом входной строки для выделения основы правые части всех порождающих правил непосредственно сравниваются с верхними символами стека. Сначала сравниваются более длинные правые части. Этот вариант метода расширенного предшествования позволяет обходиться таблицами меньшего объема, однако возрастает время поиска в таблице порождающих правил.

В компиляторах часто применяют ранний вариант метода предшествования – **метод операторного предшествования**. Этот метод применим только к *грамматикам с операторным предшествованием*.

Пусть a и b – терминальные символы; U, C, D – нетерминальные символы; x, y, z – любые строки, в том числе и пустые.

Определение. *Грамматикой с операторным предшествованием* называют грамматику класса 2 по Хомскому, в которой:

1. Правые части порождающих правил не содержат рядом стоящих нетерминальных символов, то есть в грамматике нет правил вида $U \to xCDy$. Грамматику, обладающую таким свойством называют операторной грамматикой.

2. Для каждой упорядоченной пары терминальных символов a и b выявляется не более чем одно отношение предшествования, определенное так:

А. $a \doteq b$, если и только если существует правило $U \to xaby$ или

правило $U \to xaCby$.

Б. $a \dot{<} b$, если и только если существует правило $U \to xaCy$ и вывод

$C \Rightarrow^* bz$ или $C \Rightarrow^* Dbz$.

В. $a \dot{>} b$, если и только если существует правило $U \to xCby$ и вывод

$C \Rightarrow^* za$ или $C \Rightarrow^* zaD$.

3. Различные порождающие правила имеют разные правые части.

Первое из приведенных требований позволяет ограничиться установлением отношений предшествования только для терминальных символов. Это значительно сокращает размер матрицы предшествования и повышает эффективность алгоритма разбора по сравнению с методом предшествования.

В то же время класс грамматик, для которых существует операторное предшествование или которые можно привести к грамматике с операторным предшествованием, достаточно широк. В частности, грамматики реальных языков программирования, по крайней мере, их части, почти всегда можно привести к грамматике с операторным предшествованием.

Второе и третье требования, как и аналогичные требования к грамматике предшествования, обеспечивают "беступиковость" алгоритма разбора. Перечисленные требования не обеспечивают однозначности алгоритма разбора. Однако возникающие неоднозначности без особого труда устраняются семантическими подпрограммами.

Алгоритм разбора использует только отношения терминальных символов и отыскивает для редукции не основу, а так называемую *первичную фразу* – фразу, не содержащую никакой другой фразы, и содержащую по крайней мере один терминальный символ.

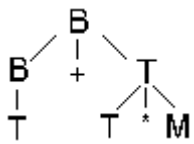
Пример. Задана грамматика $G_3[B]$:

B→T

B→B+T

T→T*M

Сентенциальная форма **⊥T+T*M⊥** имеет дерево разбора



Первичная фраза здесь **T*M**, в то время как основой является самый левый символ **T**.

Из свойств грамматики с операторным предшествованием следует, что, если x – первичная фраза, то либо существует порождающее правило $U \to x$, либо существует порождающее правило $U \to x'$ и вывод $x' \Rightarrow^* x$, в котором применяются только порождающие правила вида $U_i \to U_j$, где U_i и U_j – нетерминальные символы.

Алгоритм разбора отыскивает самую левую первичную фразу и состоит в следующем:

1. Символы входной строки поочередно переписываются в стек до тех пор, пока между верхним терминальным символом стека t_l и очередным терминальным символом входной строки t_i не появится отношение $\dot{>}$ (именно $t_l \dot{>} t_i$).

2. Тогда стек просматривается в направлении от вершины к началу до тех пор, пока между двумя терминальными символами стека t_{k-1} и t_k не появится отношение $\dot{<}$ (именно $t_{k-1} \dot{<} t_k$).

3. Часть стека от символа t_{k-1} исключительно до вершины стека включительно есть первичная фраза (см. рис.10).

4. Среди порождающих правил отыскивается правило $U \to x$, правая часть которого x есть либо найденная первичная фраза, либо отличается от нее только нетерминальными символами. Именно в этом отличии заключена возможность неоднозначного разбора, отмеченная ранее.

Если нужно, вызывается семантическая подпрограмма, соответствующая правилу $U \to x$, которое обрабатывает строку x , переводя ее на выходной язык. Семантическая подпрограмма учитывает как терминальные, так и нетерминальные символы первичной фразы, поэтому в ней можно предусмотреть возможность устранения неоднозначности разбора.

Рис. 10. Схема распознавателя операторного предшествования.

5. Затем в стеке строка x заменяется символом U и описанный процесс повторяется. На рис.10 буквами S показаны нетерминальные символы, а t – терминальные символы. Если на некотором шаге процесса синтаксического анализа обнаружится, что между двумя последовательными терминальными символами t_p и t_q (которые могут быть разделены нетерминальным символом) не существует отношения предшествования, то это свидетельствует об ошибке типа: "контрукции **…** $t_p\ t_q$**...** не существует".

Описанный алгоритм реализуется в трансляторе почти так же, как общий алгоритм предшествования, но матрица предшествования получается гораздо меньших размеров.

Пример. Найдем матрицу предшествования для грамматики G_4[П]:

П→⊥В⊥

В→Т

В→В+Т

Т→М

Т→Т*М

М→и

М→(В)

Нетерминальные символы: **П**, **В**, **Т**, **М**.

Терминальные символы: ⊥, +, *, **и**, **(**, **)**.

На основании рассмотренных выше отношений операторного предшествования получим матрицу операторного предшествования следующего вида:

t_i \ t_j	**(**	**и**	*****	+	**)**	⊥
)			$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{>}$
и			$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{>}$
*****	$\dot{<}$	$\dot{<}$	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{>}$
+	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\dot{>}$	$\dot{>}$	$\dot{>}$
(	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\doteq$	
⊥	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\dot{<}$		$\doteq$

Замечание. Отношение операторного предшествования можно найти непосредственно, пользуясь ранее приведенными определениями. Однако можно ввести дополнительные определения, которые позволят сделать процесс построения матрицы операторного предшествования более формализованным [9].

Распознаватель для грамматики с операторным предшествованием редуцирует самую левую первичную фразу. Отыскивая эту фразу и выполняя редукцию, распознаватель не обращает внимания на нетерминальные символы приводимой первичной фразы. Нетерминальные символы учитываются только в семантических подпрограммах. Поэтому порождающие правила, в правых частях которых используются нетерминальные символы, вообще не используются распознавателем, и их можно не включать в таблицу порождающих правил.

В правых частях оставшихся порождающих правил все различные

нетерминальные символы можно заменить одним символом, например символом *N*, обозначающим произвольный нетерминальный символ.

Практически распознаватель для грамматики с операторным предшествованием без помощи семантической подпрограммы не может найти полный разбор всех строк. Следовательно, этот распознаватель без привлечения семантических подпрограмм не способен выполнить полный синтаксический контроль. Это делает метод операторного предшествования менее надежным, чем метод предшествования. Однако неполнота разбора имеет определенные преимущества: сокращается объем таблицы порождающих правил и число шагов трансляции.

Пример. Используя распознаватель для грамматики с операторным предшествованием G_4[П], таблицу порождающих правил и семантических подпрограмм, а также матрицу предшествования, перевести в обратную польскую (постфиксную) запись выражение

⊥*a*+*b***c*⊥

Постфиксная запись является внутренней формой исходной программы.

Таблица порождающих правил и семантических подпрограмм может быть представлена в виде:

Номер правила	Порождающее правило	Семантическая подпрограмма
1	**П→⊥N⊥**	
2	**B→N+N**	Занести символ "+" в выходную строку.
3	**T→N*N**	Занести символ "*" в выходную строку.
4	**M→и**	Занести идентификатор **и** в выходную строку.

Перевод в ОПЗ методом операторного предшествования представим в виде следующей таблицы:

Запись в выходную строку	Стек распозна-вателя	Отношение предшество-вания	Входная строка	Номер правила	Семантическая подпрограмма
	⊥	$\dot{<}$	*a*+*b***c*⊥		
	⊥$\dot{<}$*a*	$\dot{>}$	+*b***c*⊥	4	"*a*" в выходную строку
a	⊥**M**	$\dot{<}$	+*b***c*⊥		
	⊥**M**+	$\dot{<}$	*b***c*⊥		
	⊥ **M**+$\dot{<}$*b*	$\dot{>}$	**c*⊥	4	"*b*" в выходную строку
b	⊥ **M**+**M**	$\dot{<}$	**c*⊥		
	⊥ **M**+**M***	$\dot{<}$	*c*⊥		
	⊥ **M**+**M***$\dot{<}$*c*	$\dot{>}$	⊥	4	"*c*" в выходную строку
c	⊥ **M**+$\dot{<}$**M*****M**	$\dot{>}$	⊥	3	"*" в выходную строку
*	⊥$\dot{<}$**M**+**T**	$\dot{>}$	⊥	2	"+" в выходную строку
+	⊥**B**	$\doteq$	⊥		
	⊥$\doteq$ **B**$\doteq$⊥			1	
	П				

В соответствии с алгоритмом распознавателя для грамматики с операторным предшествованием символы входной строки переписываются в стек распознавателя до тех пор, пока между верхним терминальным символом в стеке и очередным терминальным символом во входной строке не появится отношение $\dot{>}$. В этот момент в стеке выделяется первичная фраза, заключенная между отношениями $\dot{<}$ и $\dot{>}$, по таблице порождающих правил отыскивается правило,

правая часть которого совпадает с первичной фразой с точностью до нетерминальных символов, затем выполняется редукция и, если нужно, семантическая подпрограмма.

Требуемая обратная польская запись исходного выражения получится, если прочитать запись в выходной строке сверху вниз: *abc**+.

По числу шагов метод операторного предшествования эффективнее метода простого предшествования.

Надо отметить, что грамматики с операторным предшествованием в общем случае не являются подмножеством грамматик простого предшествования.

Сравнительная оценка методов, основанных на предшествовании. Основное достоинство методов, основанных на предшествовании, – простота алгоритмов распознавателя. Другое важное преимущество – наличие общих алгоритмов, позволяющих:

1) проверять, является ли грамматика грамматикой предшествования (или расширенного предшествования, или операторного предшествования);
2) отыскивать отношения предшествования (операторного предшествования);
3) построить таблицу правых и левых троек для метода расширенного предшествования.

Существование таких алгоритмов дает возможность создать конструктор, автоматически составляющий распознаватель для любой грамматики предшествования (или расширенного предшествования, или операторного предшествования).

Однако при практическом построении или эксплуатации транслятора большое значение имеет не только простота алгоритма распознавателя, но также общий объем распознавателя, его быстродействие и простота приведения грамматики реальных языков к грамматике со стандартными свойствами и возможность учета в распознавателе неформализованных особенностей языка.

Большие неприятности при применении методов, основанных на предшествовании, причиняет неоднозначность отношения предшествования,

поскольку надежных стандартных методов приведения грамматик к грамматике предшествования пока нет. В этих ситуациях несомненное преимущество имеет метод расширенного предшествования.

Следует упомянуть о неудобствах, возникающих из-за существования в реальных грамматиках порождающих правил с одинаковыми правыми частями. Для устранения таких неоднозначностей приходится усложнять блок лексического анализа.

Методы предшествования и операторного предшествования можно рассматривать как крайние случаи: в одном методе отношение предшествования устанавливается между всеми символами грамматики, в другом – между терминальными символами. Разработан также "промежуточный" метод, в котором отношения предшествования устанавливаются между всеми терминальными символами и частью нетерминальных. Это увеличивает гибкость метода, позволяя создавать анализатор, использующий минимальное количество отношений предшествования, необходимое для эффективного анализа.

3.2.3. LR – метод синтаксического анализа

Синтаксический анализатор, работающий по принципу "снизу вверх", выполняет действия двух типов:

1) сдвиг, во время которого считывается и помещается в стек символ. Это соответствует продвижению на один пункт вдоль какого-либо правила грамматики;
2) приведение, во время которого множество элементов в верхней части стека замещается каким-либо нетерминалом грамматики с помощью одного из порождающих правил этой грамматики.

Методы разбора снизу вверх почти всегда детерминированные, и перед анализатором стоит единственная задача – знать в конкретной ситуации, какое действие (сдвиг или приведение) выполнить. Если возможны два разных приведения. анализатор должен знать, какое из них выполнять и выполнять ли их вообще.

Детерминированный метод разбора снизу вверх предлагает некоторый критерий для выбора решения в случае возникновения подобных конфликтов. Решение можно принять на основании одного или более предварительно просмотренных символов (как в *LL*-разборе) или на основании информации, имеющейся в стеке разбора.

Определение. *LR(k)-грамматикой* называется грамматика, при использовании которой в качестве основы для анализатора снизу вверх все конфликты типа сдвиг/приведение и приведение/приведение можно разрешать на основании уже прочитанного текста и фиксированного числа предварительно просмотренных символов (максимум k). Буква L в LR показывает, что строки читаются слева направо, а R – что получается правосторонний разбор.

Правила, используемые в правостороннем разборе, который обычно ассоциируется с методами разбора снизу вверх, перечисляются в обратном порядке при продвижении от начального символа предложения вправо.

На практике k всегда принимает значение 0 или 1. Рассмотрение множеств даже двух предварительно просматриваемых символов при разрешении конфликтов сделало бы метод крайне громоздким. Более того, можно показать, что $LR(k)$-язык является также $LR(1)$-языком (то есть может генерироваться $LR(1)$-грамматикой, и даже $LR(0)$-языком, если допустить, что за каждым предложением следует знак окончания. Поэтому, хотя и существуют $LR(2)$-грамматики, но $LR(2)$-языков не существует. Этот вариант не похож на вариант с *LL*, где, увеличивая значение k, всегда можно (хотя и не всегда практически оправданно) разбирать большее число языков.

Все $LL(1)$-грамматики и языки обладают признаком $LR(1)$. Это объясняется тем, что решение о применении конкретного правила во время $LL(1)$-разбора базируется на замене нетерминала в сентенциальной форме и одном предварительно просматриваемом символе, в то время как для решения о выполнении приведения в случае $LR(1)$ требуется поместить в стек всю правую часть правила и считать следующий предварительно просматриваемый символ.

Таким образом, в случае $LR(1)$ имеется по меньшей мере столько же информации, на которой можно основывать решение о применении конкретного правила, сколько ее имеется в случае $LL(1)$.

Существуют также грамматики и языки, которые обладают признаком $LR(1)$, но не обладают признаком $LL(1)$. Все языки, которые можно разобрать детерминированно, имеют $LR(0)$-грамматику. Однако многие типичные свойства грамматик языков программирования относятся не к $LR(0)$-свойствам, а скорее к $LR(1)$-свойствам, так что рассматривая $LR(1)$-грамматики, можно избежать затруднений, связанных с преобразованием грамматик, которые требуются почти всегда при использовании LL-метода.

Это положительное качество LR-разбора, особенно важно, если вспомнить о неразрешимости проблемы преобразования грамматики к $LL(1)$-типу.

В отношении LR-грамматики справедливо следующее:

1. Можно решить, обладает ли грамматика признаком $LR(k)$ для заданного k. В частности, можно решить, является ли грамматика $LR(0)$ или $LR(1)$-грамматикой.
2. Нельзя решить, существует ли такое значение k, для которого заданная грамматика будет $LR(k)$- грамматикой. Этот результат не имеет особого практического значения с точки зрения разбора, так как обычно $LR(k)$-грамматики не представляют интереса, если $k > 1$.

Процесс формирования таблицы разбора рассмотрим на примере грамматики

(1) $S \rightarrow .\ real\ IDLIST$

(2) $IDLIST \rightarrow IDLIST,\ ID$

(3) $IDLIST \rightarrow ID$

(4) $ID \rightarrow A|B|C|D$

Чтобы построить таблицу разбора, необходимо сначала выделить все состояния анализатора. Для этого введем определение конфигурации и замыкания.

Определение. *Конфигурация* – это пара чисел, в которой первое число соответствует номеру правила грамматики, второе – обозначает позицию в этом правиле.

В рассматриваемом примере точка соответствует конфигурации (1,0), т. е. правилу (1) позиции 0; конфигурация (1,1) соответствовала бы точке, появляющейся сразу после *real* в правиле (1), а (2,0) – точке, появляющейся перед *IDLIST* в правой части правила (2). Конфигурации будут использоваться для представления продвижения в разборе. Так, конфигурация (2,2) покажет нам, что правая часть правила (2) распознана по запятую включительно. На любом этапе разбора может быть частично распознано некоторое число правых частей правил.

Определение. *Состояние анализатора* в таблице разбора примерно соответствует конфигурациям в грамматике с той лишь разницей, что конфигурации, которые неразличимы для анализатора, представляются одним и тем же состоянием.

Из заданного состояния, не соответствующего концу правила, можно перейти в другое состояние, введя терминальный или нетерминальный символ. Это состояние называют *преемником* первоначального состояния.

Определение. Новая конфигурация, которая получается при операции образования преемника, называется *базовой*.

Определение. Если за базовой конфигурацией следует нетерминал, то все конфигурации, соответствующие помещению точки слева от каждой правой части правила для данного нетерминала (и т. д. рекурсивно), образуют *замыкание* этой базовой конфигурации.

Например, если (1,0) соответствует состоянию 1, а (1,1) – состоянию 2, то в вышеприведенной грамматике (2,0), (3,0) и (4,0) будут также соответствовать состоянию 2. Мы говорим, что множество конфигураций {(1,1), (2,0), (3,0), (4,0)} образуют замыкание (1,1).

Начинаем с конфигурации (1,0) и последовательно выполняем операции замыкания и образования преемника до тех пор, пока все конфигурации не окажутся включенными в какие-либо состояния. Там, где ряд конфигураций

содержится в одном замыкании, каждая из них будет соответствовать одному и тому же состоянию.

Так, в предыдущей грамматике четко видны семь состояний, которые можно описать следующим образом (табл.4):

Таблица 4. Таблица состояний

Состояние	База	Замыкание
1	(1,0)	{(1,0)}
2	(1,1)	{(1,1),(2,0),(3,0),(4,0)}
3	(4,1)	{(4,1)}
4	(3,1)	{(3,1)}
5	{(2,1),(1,2)}	{(2,1),(1,2)}
6	(2,2)	{(2,2),(4,0)}
7	(2,3)	{(2,3)}

Эти состояния расположены в грамматике следующим образом:

(1) $S \rightarrow_1 real_2\ IDLIST_5$

(2) $IDLIST \rightarrow_2 IDLIST_5\ ,_6\ ID_7$

(3) $IDLIST \rightarrow_2 ID_4$

(4) $ID \rightarrow_{(2,6)} A|B|C|D_3$

Заметим, что конфигурация может соответствовать более чем одному состоянию, и в базе может быть более одной конфигурации, если преемники двух конфигураций в одном и том же замыкании неразличимы. Например, в вышеприведенном примере за конфигурациями (1,1) и (2,0) следует *IDLIST* , что делает (1,2) и (2,1) неразличимыми, пока не осуществится операция замыкания (которая в нашем случае не дает дополнительных конфигураций). Число состояний в анализаторе соответствует числу множеств неразличимых

конфигураций в грамматике.

Таблица разбора представляет собой прямоугольную матрицу (таблицу). Она состоит из столбцов для каждого терминала и нетерминала грамматики и для знака окончания и строк, соответствующих каждому состоянию, в котором может находится анализатор. Каждое состояние соответствует той позиции в порождающем правиле, которую достиг анализатор.

Не зависящая от входного текста часть анализатора или драйвер использует два стека: стек символов и стек состояний. Таблица разбора включает элементы четырех типов.

1. Элементы сдвига. Эти элементы имеют вид *S*7 и означают:
   - поместить в стек символов символ, соответствующий столбцу;
   - поместить в стек состояний 7 и перейти к состоянию 7;
   - если входной символ терминал, принять его.
2. Элементы приведения. Они имеют вид *R*4 и означают:
   - выполнить приведение с помощью правила (4), то есть допустив, что *n* есть число символов в правой части правила (4), удалить *n* символов из стека символов и *n* символов из стека состояний;
   - перейти к состоянию, указанному в верхней части стека состояний;
   - нетерминал в левой части правила (4) нужно считать следующим входным символом.
3. Элементы ошибок. Эти элементы являются пробелами в таблице и соответствуют синтаксическим ошибкам.
4. Элемент(ы) остановки. Ими завершается разбор.

Действия анализатора со сдвигом аналогичны операциям получения преемника. Поэтому действия со сдвигом в таблице разбора вносятся на основании информации о размещении состояний в грамматике (табл. 5). Позиция элементов в таблице вытекает непосредственно из вышеприведенной грамматики. Например, правило (2) означает «из состояния 2 при чтении *IDLIST* перейти в состояние 5», «из состояния 5 при чтении "," перейти в состояние 6» и т. д. Задача

внесения в таблицу действий приведения довольно сложна. Осуществление приведения может в целом зависеть от текущего левого контекста и предварительного просматриваемого символа. Однако единственные состояния, в которых приведения возможны – это состояния соответствующие окончаниям правил (в нашем примере состояния 3, 4, 5, 7).

Таблица 5. Таблица действий со сдвигом

Состояние	*S*	*IDLIST*	*ID*	*real*	*","*	*"A","B", "C","D"*	"⊥"
1				*S*2			
2		*S*5	*S*4			*S*3	
3							
4							
5					*S*6		
6			*S7*			*S*3	
7							

В особом случае *LR*(0)-грамматики предварительно просматриваемый символ не имеет к этому отношения, и действия по приведению могут помещаться в каждый столбец таблицы в любом состоянии, соответствующем окончанию правила. Если бы данная грамматика была *LR*(0)-грамматикой можно было поместить *R*4 во все столбцы состояния 3, *R*3 – во все столбцы состояния 4, *R*1 – во все столбцы состояния 5 и *R*2 – во все столбцы состояния 7. Однако в состоянии 5 в одном столбце уже имеется элемент сдвига. Нельзя помещать элемент приведения в ту же графо-клетку, и возникает конфликт сдвиг/приведение. Алгоритм построения *LR*(0) не срабатывает. Проблема конфликта возникает лишь в связи с состоянием 5, поэтому фактически все остальные элементы приведения можно внести и получить табл. 6.

Таблица 6. Таблица действий со сдвигом и приведениями

Состояние	*S*	*IDLIST*	*ID*	*real*	*","*	*"A","B", "C","D",*	"⊥"
1				*S*2			
2		*S*5	*S*4			*S*3	

3	*R*4	*R*4	*R*4	*R*4	*R*4	*R*4	*R*4
4	*R*3	*R*3	*R*3	*R*3	*R*3	*R*3	*R*3
5					*S*6		
6			*S*7			*S*3	
7	*R*2	*R*2	*R*2	*R*2	*R*2	*R*2	*R*2

Состояние 5 называется неадекватным. Попытаемся разрешить эту проблему, задавая предварительно просматриваемые символы, которые показали бы приведение в состоянии 5, а не сдвиг. Из правил (1) и (2) видно, что такими символами могут быть только "⊥" и "**,**". Приведение возможно лишь в том случае, если символом окажется "⊥" , в то время как анализатор осуществит сдвиг в состояние 6, если следующим символом будет "**,**" . Поэтому мы вносим *R*1 в пятую строку столбца, соответствующую "⊥" . Это не вызывает никакого конфликта, и неадекватность состояния снимается. В ситуациях, когда все неадекватности можно разрешить таким образом, грамматику считают простой *LR*(1)- или *SLR*(1)-грамматикой. Аналогичным образом, рассмотрев предварительно символы, можно исключить из таблицы несколько действий приведения в состояниях 3, 4, 7. Если этого не сделать, то некоторые синтаксические ошибки будут обнаружены на более поздних шагах анализа (но не позже в смысле считанного исходного текста).

Добавление к грамматике дополнительного правила

$S' \rightarrow_1 S\ HALT\ \perp$

приводит к включению элемента *HALT* (элемент останова разбора) в таблицу разбора. Исключение лишних элементов приведения и включение элемента *HALT* дает табл.7.

Таблица 7. Окончательная таблица разбора

Состояние	*S*	*IDLIST*	*ID*	*real*	"**,**"	*"A","B", "C","D"*	"⊥"
1	*HALT*			*S*2			
2		*S*5	*S*4			*S*3	
3					*R*4		*R*4

4					*R*3		*R*3
5					*S*6		*R*1
6			*S*7			*S*3	
7					*R*2		*R*2

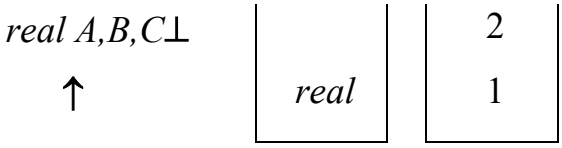
Пример. Рассмотрим, как с помощью вышеописанной таблицы разбора (таблица 7) происходит разбор строки

real A,B,C⊥.

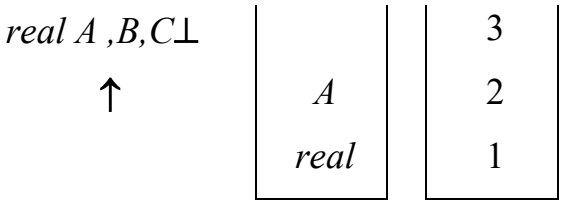
Покажем содержимое стека символов и стека состояний на каждом этапе.

Начнем с состояния 1, которое отображается в стеке:

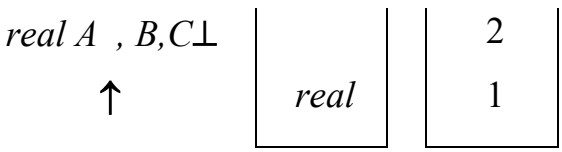
Входной символ *real* – из элемента таблицы (1, *real*) , сдвиг в состояние 2:



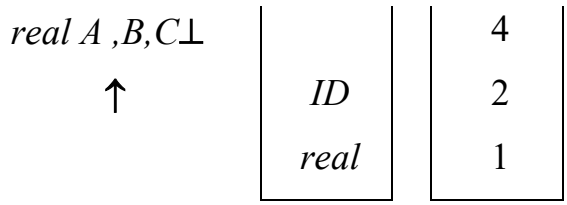
Входной символ *A* – сдвиг в состояние 3:



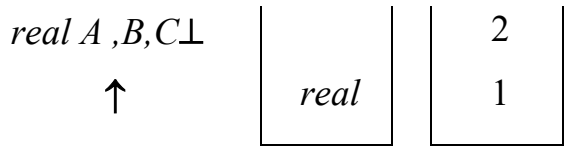
Входной символ "**,**" – приведение посредством правила (4)



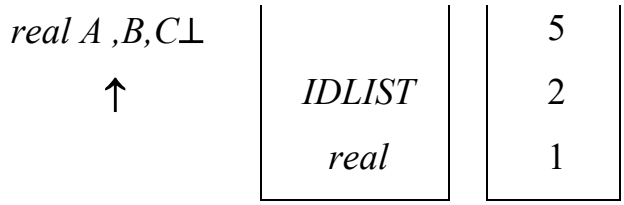
Входной символ *ID* – сдвиг в состояние 4



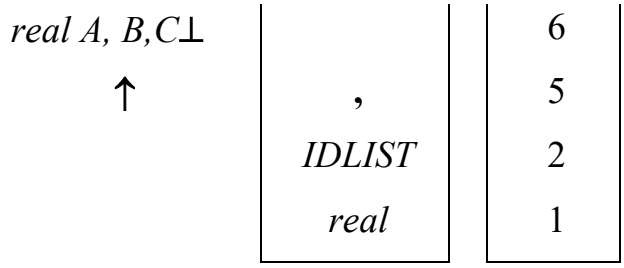
Входной символ "**,**" – приведение посредством правила (3)



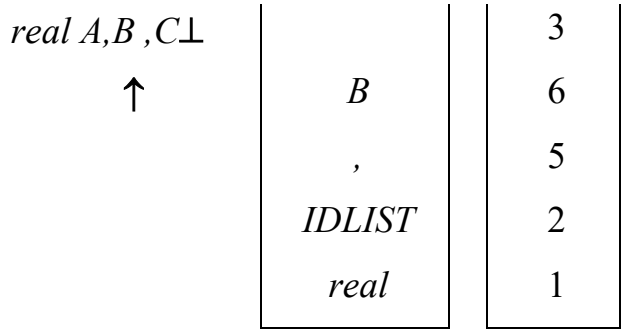
Входной символ *IDLIST* – сдвиг в состояние (5)



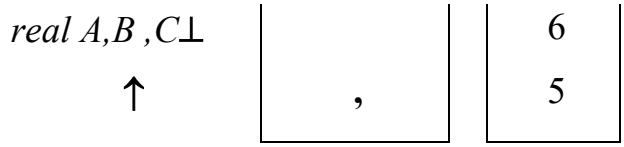
Входной символ "**,**" – сдвиг в состояние (6)



Входной символ *B* – сдвиг в состояние (3)



Входной символ "**,**" – приведение посредством правила (4)



IDLIST	2
real	1

Входной символ *ID* – сдвиг в состояние (7)

real A,B ,C⊥		7
↑	*ID*	6
	,	5
	IDLIST	2
	real	1

Входной символ "**,**" – приведение посредством правила (2)

real A,B ,C⊥		2
↑	*real*	1

Входной символ *IDLIST* – сдвиг в состояние (5)

real A,B ,C⊥		5
↑	*IDLIST*	2
	real	1

Входной символ "**,**" – сдвиг в состояние (6)

real A,B, C⊥		6
↑	,	5
	IDLIST	2
	real	1

Входной символ *C* – сдвиг в состояние (3)

real A,B,C⊥		3

↑	*C*	6
	,	5
	IDLIST	2
	real	1

Входной символ "⊥" – приведение посредством правила (4)

real A,B,C⊥		6
↑	,	5
	IDLIST	2
	real	1

Входной символ *ID* – сдвиг в состояние (7)

real A,B,C⊥		7
↑	*ID*	6
	,	5
	IDLIST	2
	real	1

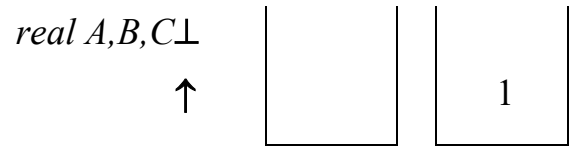
Входной символ "⊥" – приведение посредством правила (2)

real A,B,C⊥		2
↑	*real*	1

Входной символ *IDLIST* – сдвиг в состояние (5)

real A,B,C⊥		5
↑	*IDLIST*	2
	real	1

Входной символ "⊥" – приведение посредством правила (1)



Входной символ *S* – поэтому *HALT* (ОСТАНОВ).

Разбор успешно завершен. Заметим, что после сдвига входным символом всегда является следующий символ, а после приведения – символ, к которому только что привело действие.

Различие между *LR*(1)- и *SLR*(1)-грамматиками заключается в том, что при определении предварительно просмотренных символов в алгоритме построения *SLR*(1) никакого внимания не уделяется левому контексту, тогда как в более общем случае левый контекст учитывается и даже играет решающую роль при решении вопроса о том, можно ли считать заданный символ действительно символом-следователем. Однако для удаления неадекватности в наиболее универсальных *LR*(1)-грамматиках иногда возникает необходимость ввести в таблицу разбора гораздо большее число состояний. Это осуществляется за счет переопределения конфигурации путем включения в нее множества символов-следователей [2, 4].

Как видно из иерархии грамматик (рис.11) *SLR*(1)-грамматики включают грамматики со слабым предшествованием и с простым предшествованием, и также *LL*(1)-грамматики [2].

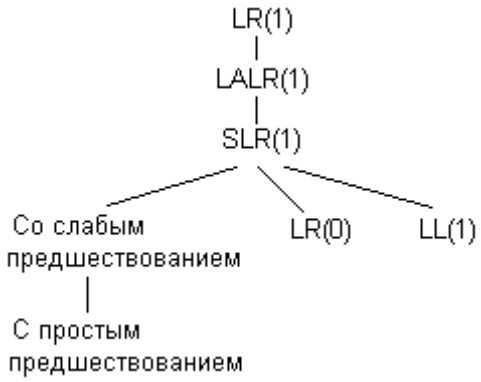


Рис.11. Иерархия грамматик.

Как *LL*-, так и *LR*-методы разбора имеют много достоинств. Оба они детерминированные и могут обнаруживать синтаксические ошибки, как правило, на самом раннем этапе. *LR*-методы обладают тем преимуществом, что они применимы к более широкому классу грамматик и языков, и преобразование грамматики в них обычно не требуется.

Однако на практике в тех случаях, когда требовались преобразования вручную, чтобы привести грамматику к *LL*(1)-форме, эту грамматику все же требовалось преобразовывать, прежде чем она смогла бы послужить основой для *LR*-анализатора. Таким образом, теоретическое преимущество *LR*-анализа перед *LL*-анализаторами (то есть их большая универсальность) оказывается не таким уж значительным на практике. При наличии хорошего преобразователя грамматик сам факт необходимости преобразовать грамматику не вызывает в действительности у разработчика компилятора никаких затруднений, и в большинстве случаев ему даже не приходится рассматривать эту преобразованную грамматику. Существуют серьезные возражения против преобразования вручную: во-первых, трудно выбрать вид преобразования, во-вторых, нет гарантий, что преобразование не изменит язык.

В отношении размеров таблиц и времени разбора следует отметить следующее: по размеру таблиц *LL*-метод и *LR*-метод практически эквивалентны. *LL*-метод несколько быстрее, чем *LR*-метод.

Кроме переборных алгоритмов, выполняющих синтаксический анализ методом "грубой силы" и имеющих экспоненциальную сложность, существуют более эффективные общие (универсальные) алгоритмы, применимые ко всем КС-грамматикам. Среди КС-грамматик существуют двусмысленные грамматики, т. е. такие, когда одна и та же цепочка терминальных символов может быть порождена несколькими разными способами, приводящими к различным структурам (деревьям вывода) исходной цепочки. Именно эта неоднозначность синтаксической структуры предложений является причиной двусмысленности в КС-языках. Универсальные алгоритмы синтаксического анализа должны для любой данной контекстно-свободной грамматики *G* и входной цепочки α

восстанавливать все возможные выводы цепочки α в *G*. Именно этим объясняется сложность универсальных алгоритмов.

Универсальные алгоритмы синтаксического анализа также делятся на два класса: нисходящие и восходящие. Два очень элегантных общих алгоритма синтаксического анализа КС-языков: нисходящий – алгоритм Эрли, и восходящий – алгоритм Кока–Янгера–Кассами рассмотрены в [2].

4. РАСПРЕДЕЛЕНИЕ ПАМЯТИ

После выявления структуры программы транслятор должен выделить место в памяти машины для значений переменных, используемых в программе, и поместить соответствующие адреса в таблицу идентификаторов. Реализуется это в фазе распределения памяти [3]. В современных трансляторах применяются различные схемы распределения памяти, которые в общем случае тесно связаны как с конкретной логической структурой данных языка программирования, так и с возможностями представления этих структур в памяти конкретной машины. Для построения механизма распределения памяти транслятором необходимо предварительно решить вопросы связанные с:

- представлением структур данных языка программирования в адресном пространстве машины (обычно в линейном адресном пространстве);
- организацией передачи данных между процедурами (блоками) программы;
- стратегией выделения, освобождения и учета памяти во время выполнения программы.

Уяснение всех этих вопросов возможно лишь при детальном описании исходного языка программирования. У нас нет такой возможности, поэтому остановимся лишь на общих подходах их решения.

Будем полагать, что язык программирования имеет структуру вложенных блоков и процедур, объединенных в рамках основной программы, которая тоже является процедурой, но без параметров. В языке допускается использование

переменных различных типов: простые переменные, массивы, структуры и т.д. При этом в описании массива внутри блока (процедуры) в качестве индексов можно использовать переменные.

Эффективность обработки данных во время выполнения программы существенно зависит от выбора машинного представления данных языкового уровня.

Типы данных исходной программы в процессе трансляции должны быть отображены на типы данных машины. Представление значений типа данного в памяти машины должно обеспечивать эффективное выполнение операций, применяемых при обработке данных этого типа. Поэтому для простых типов данных (целые и вещественные числа, литеры и т.д.) применяется, как правило, прямое аппаратное представление, для таких представлений операции над типами имеют прямые аналоги машинных команд. Для сложных типов (массивы, структуры, записи и т.д.) помимо памяти под элементы этих типов транслятор создает вспомогательные структуры – дескрипторы (информационные векторы), указатели, с помощью которых и обеспечивается эффективный доступ к значениям элементов сложного типа. С основными способами машинного представления структурных типов данных языков программирования можно познакомится в [1, 3].

Программа на выходе транслятора в общем случае состоит из двух областей: области команд и области данных. В области команд размещаются команды объектной программы, размер этой области вычисляется во время трансляции и равен сумме длин команд объектного хода. Физическая память для области команд выделяется обычно во время загрузки программы загрузчиком (редактором связей, компановщиком).

С областью данных дела обстоят несколько по-другому. Так как в программе есть данные с ограниченным временем существования, например, локальные и промежуточные переменные вложенных блоков и процедур, то нет смысла закреплять за этими переменными ячейки памяти в области данных на все время выполнения программы. И, наоборот, за глобальными и

промежуточными переменными основной программы необходимо закрепить память на все время выполнения программы. В языке программирования могут быть также средства, которые позволяют программисту самому выделять и освобождать память для переменных. С этих позиций всю область данных объектной программы целесообразно разделить на следующие области:

- *статическую область* — эта область имеет постоянное число ячеек памяти, которые закрепляются за элементами данных, имеющих статус глобальных и не локальных переменных, на все время выполнения программы;
- *автоматическую область* — размер этой области изменяется в ходе выполнения программы, изменяется и закрепление ячеек памяти за элементами данных для локальных переменных, однако структура и логика изменения области для этих переменных однозначно определяется структурой вложенности блоков и процедур, которая известна в период трансляции;
- *динамическую область* – управлением выделения и освобождения памяти для данных занимается программист, логика изменения динамической области транслятору не известна.

В языках программирования область действия переменной в процедуре (блоке) определяется в контексте описания процедуры, а не в контексте ее вызова. Это обеспечивает возможность управления памятью в автоматической области с использованием информации о вложенности блоков и процедур, получаемой в ходе синтаксического анализа.

Основная идея этого управления состоит в следующем. Автоматическая область памяти рассматривается как стек, на который в период трансляции накладывается логическая структура, отражающая вложенность блоков и процедур. При этом для внутренних переменных блока отводится участок стека – *рамка* и переменные в блоке адресуются в виде пары (<№ рамки>, <смещение>), где номер рамки соответствует номеру блока (процедуры) при статической их вложенности, смещение – адресу переменной относительно

начала рамки. Во время генерации машинных команд эта пара хранится в таблице идентификаторов и переводится в действительный адрес переменной, т.е. в пару (<регистр адреса базы>, <смещение>), где <регистр адреса базы> — это обычно индексный регистр, в который при входе в блок во время выполнения программы помещается адрес первой ячейки памяти, выделенной данной рамке. А так как динамика вызовов процедур в ходе выполнения программы не может нарушить структуру статической их вложенности, то адрес начала текущей рамки можно всегда определить, если известен адрес начала и размер объемлющей рамки. Поэтому в период выполнения программы необходимо хранить информацию о статической вложенности процедур (блоков). Реализуется это с помощью малого стека – *дисплея,* при этом в стеке находятся адреса начала рамок блоков в последовательности их вызовов. И при входе в блок (процедуру) адрес начала рамки активного блока помещается в вершину дисплея, а при выходе — выталкивается из него.

В объектной программе дисплей может быть реализован различными способами: для него может быть выделен отдельный стек, но его можно хранить и непосредственно в связанной с ним области данных процедуры (блока).

Помимо областей для хранения значений локальных и промежуточных переменных транслятор должен выделить в области данных процедуры ячейки для:

- области сохранения состояния вычислительного процесса процедуры (здесь запоминаются содержимое регистров и адрес возврата в вызывающую область);
- области для приема значений фактических параметров (или их адресов).

Область данных процедуры (блока) имеет фиксированную длину, определяемую во время трансляции. Порядок выделения ячеек для простых переменных, информационных векторов и промежуточных переменных не имеет значения. Самый простой способ – это распределять память, следуя порядку описаний в блоке (процедуре).

Если в языке программирования нельзя использовать динамические массивы, то размер области данных процедуры известен во время трансляции и

функция управления памятью во время выполнения программы сводится к управлению дисплеем. Если же динамические массивы допустимы в языке, то ситуация несколько усложняется. В этом случае размер рамки стека для хранения элементов массива во время трансляции не известен. Решается эта проблема достаточно просто: выделяют статическую память для процедуры (блока) в начале каждой рамки, а во время выполнения – динамическую память для элементов массива в конце рамки.

Таким образом, область данных процедуры логически состоит из двух частей: статической и динамической. В статической части память распределяется во время трансляции и включает следующие основные области:

- область сохранения состояния вычислительного процесса;
- область для локально описанных простых переменных и для информационных векторов локальных структур данных;
- область для промежуточных переменных;
- область для хранения адресов фактических параметров процедуры.

В динамической части области данных процедуры выделяется память для хранения элементов динамических массивов. Для этого при входе в процедуру во время выполнения программы вызывается подпрограмма размещения массива, которая вычисляет всю необходимую информацию и заносит ее в информационный вектор, а затем занимает в стеке нужное число ячеек для элементов массива и заносит вычисленный адрес текущей рамки на вершину дисплея.

Память для динамических структур данных, определяемых программистом, выделяется обычно в пределах единого сегмента памяти, называемого кучей. *Куча* – это блок памяти, части которого выделяются и освобождаются некоторым способом, не подчиняющимся какой-либо структуре. В этом случае возникают проблемы, связанные с выделением, утилизацией и повторным использованием памяти из кучи. Единого метода управления кучей не существует. На практике используются частные приемы, такие как счетчики ссылок, сборка мусора и другие [1, 3, 4, 7].

После выяснения структуры (и значения) программы необходимо выделить место в памяти для значений переменных и т.п. и поместить соответствующие адреса, где это необходимо, в таблицу символов. Фаза распределения памяти почти не зависит от языка и машины и практически одинакова для подавляющего большинства языков, имеющих блочную структуру и реализуемых на многих типичных ЭВМ. Распределение памяти, по существу, заключается в отображении значений, появляющихся в программе, на запоминающем устройстве машины. Если допустить, что мы реализуем типичный язык с блочной структурой, а машина имеет линейное запоминающее устройство, то наиболее подходящим устройством, на котором мы будем базировать распределение памяти, представляется стек или память магазинного типа. Рассмотрим классическую структуру **стека времени прогона** для локального распределения памяти и покажем, как можно произвести глобальное распределение памяти в отдельной области, называемой «кучей».

Почти каждой программе требуется какой-либо объем памяти для хранения значений переменных и промежуточных значений выражений. Например, если идентификатор описывается как

int x

т. е. *x* может принимать значения (по одному) типа целого, то компилятору придется выделить память для *x.* Иными словами, компилятор должен выделить достаточно места, чтобы записать любое целое число в некотором диапазоне, причем ширина этого диапазона зависит собственно от машины (обычно она определяется целым числом слов или байтов).

Память нужна также для промежуточных результатов и констант. Например, при вычислении выражения *a + c*d* сначала вычисляется *c*d,* причем значение запоминается в машине, а затем выполняется сложение. Память, используемая для хранения промежуточных результатов, называется *рабочей.* Рабочая память может быть статической или динамической.

В каждом компиляторе предусмотрена схема распределения памяти,

которая до некоторой степени зависит от компилируемого языка. Если память, выделенная для значений идентификаторов, никогда не высвобождается, то подходящей структурой для неё является одномерный массив. Если считается, что массив имеет левую и правую стороны, память можно выделять слева направо. При этом применяется указатель, показывающий первый свободный элемент в массиве.

В языке, имеющем блочную структуру, память обычно высвобождается при выходе из блока, которому она выделена. В этом случае оптимальным решением было бы разрешить указателю отодвигаться обратно влево при высвобождении памяти. Такой механизм распределения эквивалентен стеку времени прогона или памяти магазинного типа, хотя принято показывать стек заполняющимся снизу вверх. (Некоторые авторы предпочитают стек, заполняющийся сверху вниз.)

Часть стека, соответствующую определенному блоку, называют *рамкой* стека. Считается, что *указатель стека* показывает на его первый свободный элемент.

Кроме указателя стека, требуется также указатель на дно текущей рамки (*указатель рамки*). При входе в блок этот указатель устанавливается равным текущему значению указателя стека. При выходе из блока сначала указатель стека устанавливается равным значению, соответствующему включающему блоку. Указатель рамки включающего блока может храниться в нижней части текущей рамки стека, образуя часть статической цепи или (как мы будем считать) массива, который называется дисплеем. Его можно использовать для хранения во время прогона указателей на начало рамок стека, соответствующих всем текущим блокам. Это несколько упрощает перенастройку указателя рамки при выходе из блока. Для возвращения дисплея в исходное состояние при выходе из вызова процедуры можно применять оба эти метода. В любом случае во время прогона здесь выполняется настройка указателей при входе в блок и выходе из него.

Если бы вся память была статической, адреса времени прогона могли

бы распределяться во время компиляции, и значения элементов дисплея также были бы известны во время компиляции.

Рассмотрим следующий отрезок программы:

begin int n; read(n);
int numbers [1: n] ;
real p;
begin real x ,y; ...end ....end

Место для *numbers* должно выделяться в первой рамке стека, а для *x* и *y* – в рамке над ней. Но во время компиляции неизвестно, где должна начинаться вторая рамка, так как не известен размер чисел. Одно из решений в этой ситуации – иметь два стека: один для статической памяти, распределяемой в процессе компиляции, а другой для динамической памяти, распределяемой в процессе прогона. Однако этого обычно не делают, возможно, из-за тех проблем, которые возникают в связи с наличием более чем одного увеличивающегося и уменьшающегося стека во время прогона. Другое решение заключается в том, чтобы при компиляции выделять статическую память в каждом блоке в начале каждой рамки, а при прогоне – динамическую память над статической в каждой рамке. Это значит, что когда происходит компиляция, мы все еще не знаем, где начинаются рамки (кроме первой), но можем распределять статические адреса *относительно начала определенной рамки.* При прогоне точный размер рамок, соответствующих включающим блокам, известен, так что при входе в блок нужный элемент дисплея всегда можно установить так, чтобы он указывал на начало новой рамки .

Предположим, что массив занимает только динамическую память. Однако некоторая информация о массиве обычно известна во время компиляции, например его размерность (а следовательно, и число границ – две на каждую размерность), и при выборке определенного элемента массива она может потребоваться. Во многих языках сами границы могут быть не известны при компиляции, но почти наверняка мы знаем их число, и для значений этих границ можно выделить статическую память. Тогда мы вправе считать, что

массив состоит из статической и динамической частей. Статическая часть массива может размещаться в статической части рамки, а динамическая – в динамической. Кроме информации о границах, в статической части может храниться указатель на сами элементы массива.

Теперь рассмотрим, как распределяется рабочая память при использовании стека. Во многих языках все идентификаторы должны описываться в блоке, прежде чем можно будет вычислять какие-либо выражения. Отсюда следует, что рабочую память можно выделять в конкретной рамке стека над памятью, предусмотренной для значений, соответствующих идентификаторам (обычно называемой стеком идентификаторов). Конкретнее статическую рабочую память можно выделять в вершине статического стека идентификаторов, а динамическую рабочую память – в вершине динамического стека идентификаторов. Если описатели идентификаторов могут появляться после вычисления выражений, стеки идентификаторов и рабочий считают двумя разными стеками во время фазы распределения памяти, хотя для последующих проходов их можно объединять.

В процессе компиляции статический стек идентификаторов растет по .мере объявления идентификаторов. Однако статический рабочий стек может не только увеличиваться в размере, но и уменьшаться. Возьмем, например,

$$x = a + b*c$$

При вычислении выражения ($a + b*c$) потребуется рабочая память, чтобы записать значение $b*c$ перед выполнением сложения. Ту же самую рабочую память можно использовать для хранения результата сложения. Однако после осуществления присвоения этот объем памяти можно освободить, так как он уже не нужен.

Динамическая рабочая память должна распределяться во время прогона, статическая же может распределяться во время компиляции. Объем статической рабочей памяти, который должен выделяться каждой рамке, определяется не рабочей памятью, требуемой в конце каждого блока (обычно она является нулем), а *максимальной* рабочей памятью, требуемой в любой точке внутри

блока. Для статической рабочей памяти эту величину можно установить в процессе компиляции, если в фазе распределения памяти мы ассоциируем с рабочей стековой областью текущей рамки два указателя, причем один из них указывает на текущую границу статической рабочей памяти, а другой – на максимальный размер, до которого она выросла при работе с текущим блоком. Именно значение этого второго указателя при выходе из блока и дает объем статического рабочего стека, включаемый в соответствующую рамку.

Если в реализуемом языке возможна параллельная обработка, т. е. одновременно могут быть активными несколько процессов или процессы могут сливаться каким-либо произвольным способом, стек становится неприемлемым как модель распределения памяти, поскольку объем памяти уже необязательно будет высвобождаться в порядке, обратном тому, в каком он выделяется. (Принцип «первым вошел – первым вышел» уже не применим.) Но можно воспользоваться подходящим обобщением стека, а именно позволить ему иметь ветви, как у дерева, чтобы каждая ветвь соответствовала одному из параллельных процессов [1, 4].

До сих пор мы. рассматривали только локальную память, удобную для системы распределения памяти, базирующейся на стеке. В большинстве языков программирования обычная блочная структура обеспечивает высвобождение памяти в порядке, обратном тому, в каком она распределялась. Однако в программах со списками и другими структурами данных, содержащими указатели, часто необходимо сохранять память за пределами того блока, в котором она выделялась. Память для любого элемента списка выделяется глобально (т. е. на время действия всей программы).

Хотя обычно термин «глобальная память» подразумевает, что память выделяется на время действия всей программы, попытка вновь использовать тот объем памяти, который уже становится недоступным программе, представляется вполне разумной. Даже в тех случаях, когда подобная стратегия

несколько противоречит духу глобального распределения памяти, ее целесообразно одобрить, если она не изменяет значения ни одной из программ, т. е. процесс восстановления памяти происходит без участия программиста. Поскольку не существует метода обращения к какому-либо конкретному участку памяти, программист не имеет возможности выяснить, перераспределяется ли эта память для других целей.

Глобальная память не может ориентироваться на стек, поскольку его распределение и перераспределение не соответствуют принципу «последним вошел – первым вышел». Обычно для глобальной памяти выделяется специальный участок памяти, называемый иногда «кучей». Компилятор может выделять память и из стека, и из кучи, и в данном случае уместно сделать так, чтобы эти два участка «росли» навстречу друг другу с противоположных сторон запоминающего устройства. Это значит, что память можно выделять из любого участка до тех пор, пока они не встретятся. Такой метод позволяет лучше использовать имеющийся объем памяти, чем при произвольном ее делении на два участка, ограничивающем и стек, и кучу.

Размер стека увеличивается и уменьшается упорядоченно по мере входа и выхода из блоков. Размер же кучи может только увеличиваться, если не считать тех «дыр», которые могут появляться за счет освобождения отдельных участков памяти. Существуют две разные концепции регулирования кучи. Одна из них основана на так называемых *счетчиках ссылок,* а другая – на *сборке мусора* [1, 4].

Счетчики ссылок

При использовании счетчиков ссылок память восстанавливается сразу после того, как она оказывается недоступной для программы. Куча рассматривается как, последовательность ячеек, в каждой из которых содержится невидимое для программиста поле (счетчик ссылок), где ведется счет числу других ячеек или значений в стеке, указывающих на эту ячейку. Счетчики ссылок обновляются во время выполнения программы, и когда значение конкретного счетчика становится нулем, соответствующий объем

памяти можно восстанавливать.

Сборка мусора

Иной метод высвобождения памяти для ее повторного использования известен под названием *сборки мусора.* Этот метод высвобождает память не тогда, когда она становится недоступной, а только тогда, когда программе требуется память в виде кучи (или, возможно, в виде стека, если эти две области памяти «растут» навстречу друг другу), но ее нет в наличии. Таким образом, у программ с умеренной потребностью в памяти необходимость в ее высвобождении может не возникать. Тем же программам, которым может не хватить объема памяти в виде кучи, придется приостановить свои действия и затребовать недоступный объем памяти, а затем уже продолжать свою работу.

5. ГЕНЕРАЦИЯ КОДА

Анализ и синтез в компиляции, хотя их часто удобно рассматривать как отдельные процессы, во многих случаях происходят параллельно. Это совершенно очевидно для однопроходного компилятора, но и в многопроходных компиляторах генерирование объектного или какого-либо промежуточного кода большей частью осуществляется одновременно с синтаксическим анализом. Последнее не должно вызывать удивления: ведь как только синтаксический анализатор распознает, например, присваивание, он, естественно, выдаст код для присваивания в данной точке.

В качестве примера включения действия в грамматику для генерирования кода рассмотрим проблему разложения арифметических выражений на четверки. Выражения определяются грамматикой со следующими правилами:

$S \rightarrow EXP$

$EXP \rightarrow TERM$ $\quad EXP \rightarrow EXP+TERM$

$TERM \rightarrow FACT$ $\quad TERM \rightarrow TERM\ *FACT$

$FACT \rightarrow\ -FACT$ $\quad FACT \rightarrow ID$ $\quad FACT \rightarrow (EXP)$

$ID \rightarrow a \mid b \mid c \mid d \mid e\ ,$

где S — начальный символ. Примеры выражений:

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

Все идентификаторы содержат одну букву: это исключает необходимость выполнять лексический анализ.

Грамматика для четверок имеет следующие правила:

$QUAD \rightarrow OPERAND\ OP1\ OPERAND = INT$

$QUAD \rightarrow OP2\ \ OPERAND = INT$

$OPERAND \rightarrow INT$

$OPERAND \rightarrow ID$

$INT \rightarrow DIGIT$

$INT \rightarrow DIGIT\ \ INT$

$DIGIT \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$ID \rightarrow a \mid b \mid c \mid d \mid e$

$OP1 \rightarrow + \mid *$

$OP2 \rightarrow -$

Примеры четверок: $-a = 4$, $a + b = 7$, $6 + 3 = 11$.

Выражению $(- a + b) * (c + d)$

будет соответствовать такая последовательность четверок:

$$- a = 1$$

$$1 + b = 2$$

$$c + d = 3$$

$$2 * 3 = 4$$

Целые числа с левой стороны от знаков равенства обозначают номера четверок. Из сформированных четверок нетрудно генерировать машинный код, а многие компиляторы на основании четверок осуществляют трансляцию в промежуточный код. Покажем, как включать в грамматику выражений действия для генерирования соответствующих четверок. Действия заключаются в угловые скобки <, > и обозначаются как *A1, A2...* . В данном случае требуются четыре различных действия. Алгоритм пользуется стеком, а номера четверок

обозначаются с помощью целочисленной переменной. Перечислим эти действия:

A1 – поместить элемент в стек;

A2 – взять три элемента из стека, напечатать их с последующим знаком «=» и номером следующей размещаемой четверки и поместить полученное целое число в стек;

A3 – взять два элемента из стека, напечатать их с последующим знаком «=» и номером следующей размещаемой четверки и поместить полученное целое число в стек;

A4 – взять один элемент из стека.

Грамматика с учетом этих добавленных действий примет вид

$S \rightarrow EXP <A4>$

$EXP \rightarrow TERM$

$EXP \rightarrow EXP + <A1> TERM <A2>$

$TERM \rightarrow FACT$

$TERM \rightarrow TERM * <A1> FACT <A2>$

$FACT \rightarrow -<A1>FACT <A3>$

$FACT \rightarrow ID <A1>$

$FACT \rightarrow (EXP)$

$ID \rightarrow a \mid b \mid c \mid d \mid e.$

Действие *A1* используется для помещения в стек всех идентификаторов и операторов, а действия *A2* и *A3* – для получения бинарных и унарных четверок соответственно.

В качестве примера проследим за преобразованием в четверки выражения

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

Действие *A1* выполняется после распознавания каждого идентификатора и оператора, действие *A2* – после второго операнда каждого знака бинарной операции, действие *A3* – после первого (и единственного) операнда каждого знака унарной операции. Действие же *A4* выполняется только один раз после

считывания всего выражения (таблица 8).

Таблица 8. Действия по преобразованию выражения (-a + b) * (c + d) в четверки.

Последняя считанная литера	Действие	Выход
(- (минус) *a*	– *A1,* поместить в стек «-» *A1,* поместить в стек *a* *A3,* удалить из стека 2 элемента, поместить в стек «1»	- a = 1
+ *b*	*A1,* поместить в стек «+» *A1,* поместить в стек *b* *A2,* удалить из стека 3 элемента, поместить в стек «2»	1 + *b* = 2
) * (*c* + *d*	– *A1,* поместить в стек «*» – *A1,* поместить в стек **c** *A1,* поместить в стек «+» *A1,* поместить в стек *d* *A2,* удалить из стека 3 элемента, поместить в стек «3»	*c* + *d* = 3
)	*A2,* удалить из стека 3 элемента, поместить в стек «4»	2*3 = 4
	A4, удалить из стека 1 элемент	

Следует заметить, что:

1. в рассмотренном примере нам ни разу не пришлось сравнивать приоритеты двух знаков операций, поскольку эта информация содержалась в грамматике;
2. этот метод нетрудно распространить на языки с множеством различных приоритетов знаков операций;
3. приведенная грамматика содержит левую рекурсию.

Как уже отмечалось, процесс компиляции можно разбить на две стадии – анализ и синтез. Проанализировав программу и поместив в таблицы информацию, требуемую для генерации кода, компилятор должен переходить к построению соответствующей программы в машинном коде.

Код генерируется при обходе дерева, построенного анализатором [3]. Обычно генерация кода осуществляется параллельно с построением дерева, но может выполняться и как отдельный проход. Если выполняются два прохода, то представление полного дерева разбора необходимо передать из одного прохода в другой.

Будем считать, что фактически для получения машинного кода требуются два отдельных прохода:

1) генерация не зависящего от машины промежуточного кода (или объектного языка);

2) генерация машинного кода (или кода сборки) для конкретной машины.

Во многих компиляторах оба эти процесса выполняются за один проход.

Промежуточные коды (или объектные языки) можно проектировать на различных уровнях. Так, иногда промежуточный код получают, просто разбивая сложные структуры исходного языка на более удобные для обращения элементы. Однако можно в качестве промежуточного кода (в этом случае его чаще называют объектным языком) использовать какой-либо обобщенный машинный код, который затем транслируется в код реальной машины. Получение промежуточного кода возможно до или после распределения памяти. Если это происходит до распределения памяти, то операндами могут служить идентификаторы программы (или их представления после лексического анализа – токенны) и присваиваемые компилятором идентификаторы, причем в последнем варианте используются адреса времени прогона.

Одним из видов промежуточного кода являются четверки. Например, выражение $(-a+b)*(c+d)$ можно представить как четверки следующим образом:

$-a=1$ $\quad 1+b=2$ $\quad c+d=3$ $\quad 2*3=4$

Здесь целые числа соответствуют идентификаторам, присваиваемым компилятором. Четверки можно считать промежуточным кодом высокого уровня. Такой код часто называют трехадресным – два адреса для операндов

(кроме тех случаев, когда имеют место унарные операции) и один для результата. Другой вариант кода – тройки (двухадресный код). Каждая тройка состоит из двух адресов операндов и знака операции. Если сам операнд является тройкой, то используется ее позиция, что исключает необходимость иметь в каждой тройке адрес результата.

Пример. Выражение $a+b+c*d$ можно представить в виде четверок:

$a + b = 1$ $c * d = 2$ $1 + 2 = 3$

и в виде троек:

$a + b$ $c * d$ $1 + 2$

Тройки компактнее четверок, но если в компиляторе есть фаза оптимизации, которая пересылает операторы промежуточного кода, их применение затруднительно. Наилучшее решение этой проблемы – косвенные тройки, т. е. операнд, ссылающийся на ранее вычисленную тройку, должен указывать на элемент таблицы указателей на тройки, а не на саму эту тройку.

Как тройки, так и четверки можно распространить не только на выражения, но и на другие конструкции языка. Например, присваивание

$$a = b$$

в виде четверки представляется : $a = b = 1$,

а в виде тройки: $a = b$

Аналогично условное предложение *if a then b else c fi* можно считать выражением с тремя операндами, которому требуются четыре адреса как четверке и три – как тройке.

Не менее популярны в качестве промежуточного кода префиксная и постфиксная нотации. Будем рассуждать в терминах промежуточного языка (или объектного), состоящего из команд вида

тип-команды параметры

Тип-команды может быть, например, вызовом стандартного обозначения операции, тогда параметрами могут быть имя знака операции, адреса операндов и адрес результата. Например,

STANDOP II+, A, B, C

Здесь *II+* обозначает сложение двух целых чисел, а *A, B, C* служат во время прогона адресами двух операндов и результата. Для того чтобы в промежуточном коде можно было воспользоваться адресами во время прогона, распределение памяти к этому времени должно быть уже закончено. При распределении памяти необходимо знать, какой объем памяти занимает целое, вещественное и другие значения на той машине, для которой выдается объектный код. Это означает, что промежуточный код не является в строгом смысле слова интерфейсом между не зависящей и зависящей от машины частями компилятора. Тем не менее если речь идет о переводе фронтальной части компилятора (т. е. части, транслирующей исходный код в промежуточный) с одной машины на другую, то единственное, что здесь может потребоваться, – это изменение нескольких констант.

Промежуточный код пишется на относительно низком уровне. Обычно выдвигается условие, чтобы промежуточный код отражал структуру реализуемого языка.

Промежуточный код напоминает префиксную нотацию в том смысле, что знак операции всегда предшествует своим операндам. Но он имеет менее общий характер, так как сами операнды не могут быть префиксными выражениями. При получении промежуточного кода для хранения адресов операндов до тех пор, пока не будет напечатан знак операции, используется стек. Поскольку знак операции можно установить (во многих языках) лишь после того, как станут известны его операнды, стек служит также для хранения каждого знака операции на то время, пока не определены оба операнда.

Адрес на время прогона обычно соотносится со стеком, и каждый такой адрес можно представить тройкой вида

(тип-адреса, номер блока, смещение).

Тип-адреса может быть прямым или косвенным (т. е. адрес может содержать значение или указатель на значение) и ссылаться на рабочий стек или стек идентификаторов. Он может быть также литералом или константой.

Номер блока позволяет найти номер уровня блока в таблице блоков, что обеспечивает доступ к конкретной рамке стека через дисплей. В случае литерала или константы номер блока не используется. Смещение (для адреса стека) показывает смещение значения конкретной рамки по отношению к началу стека идентификаторов или рабочего стека. Если тип-адреса представляет собой литерал, то смещение выражается самим значением, а если тип-адреса – константа, то смещение нужно найти в таблице констант по заданному им адресу. В том случае, когда в каждой рамке стека рабочий стек помещается сразу же над стеком идентификаторов, смещения адресов рабочего стека по отношению к началу рамки можно рассчитывать, как только станет известным размер стека идентификаторов для конкретной рамки (т. е. во время прохода, *следующего* за проходом, при котором происходит распределение памяти).

Адреса во время прогона для идентификаторов определяются в процессе распределения памяти и хранятся в таблице символов вместе с информацией о типе и т. п.

Кроме рассмотренных, существуют и другие команды промежуточного кода:

SETLABEL Li для установки метки и

ASSIGN type, add1, add2 для присваивания.

Тип необходим как параметр, чтобы определить размер значения, переписываемого из *add1* в *add2.* Может потребоваться просмотр типа (вида) при трансляции этой команды в фактический код машины, если значения будут содержать динамические части, поэтому во время генерации машинного кода нужна таблица видов.

Рассмотрим структуры данных, которые требуются при генерации кода (т. е. генерации промежуточного кода), и их употребление. Как упоминалось выше, для хранения адресов операндов необходим стек значений. В этом стеке, который называют *нижним стеком,* можно хранить также и другую информацию. Например, значение может быть связано со своими характеристиками:

а) адресом времени прогона;

б) типом;

в) областью действия,

помимо той информации, которая имеет значение для диагностики. Это статическая информация, так как (по крайней мере, для большинства языков) ее можно получить во время компиляции. Так, при компиляции может быть известно если не фактическое значение, то во всяком случае адрес целого числа.

При трансляции *A + B* первыми помещаются в нижний стек статические свойства *A*. Любой элемент нижнего стека можно представить в виде структуры, имеющей поле для каждой из своих статических характеристик. В случае идентификаторов статические характеристики находятся из таблицы символов. Затем в стек знаков операции помещается знак операции +, и в нижний стек добавляются статические характеристики *B*. Знак операции берется из стека знаков операций, а его два операнда – из нижнего стека. Типы операндов используются для идентификации знака операции, после чего генерируется код. И наконец, в нижний стек помещаются статические характеристики результата.

Этот процесс можно распространить и на более сложные выражения, например, на те, которые генерируются грамматикой с правилами

EXP → TERM	*EXP → EXP + TERM*	*EXP → EXP - TERM*
TERM → FACT	*TERM → TERM * FACT*	*TERM → TERM / FACT*
FACT → constant	*FACT → identifier*	*FACT→ (EXP)*

После чтения идентификатора или константы, знака операции и второго операнда необходимо выполнить следующие действия:

A1. После чтения идентификатора или константы (т.е. листа синтаксического дерева) поместить в нижний стек соответствующие статические характеристики.

A2. После чтения оператора поместить символ операции в стек знаков операций.

A3. После чтения правого операнда (который может быть выражением) извлечь из стеков знак операции и его два операнда, генерировать соответствующий код, так как знак операции идентифицирован, и поместить в стек статические характеристики результата. Тип результата становится известным во время идентификации знака операции, например сложение двух целых чисел всегда дает целое число.

При включении в грамматику этих действий она примет следующий вид:

EXP → TERM

EXP → EXP + <A2> TERM <A3>

EXP → EXP - <A2> TERM <A3>

TERM → FACT

*TERM → TERM * <A2> FACT <A3>*

TERM → TERM / <A2> FACT <A3>

FACT → identifier <A1>

FACT → constant <A1>

FACT→ (EXP)

Нижний стек частично используется для передачи информации о типе вверх по синтаксическому дереву. Рассмотрим синтаксическое дерево, соответствующее выражению $a * b + x * y$ (рис. 12)

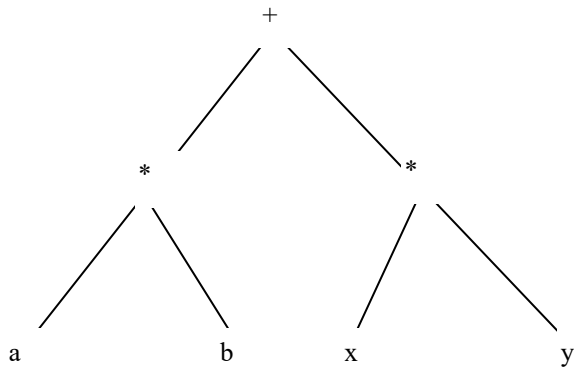


Рис. 12 . Синтаксическое дерево для выражения $a * b + x * y$.

Если значения a и b имеют тип целого, а x и y – тип вещественного значения, компилятор может заключить, воспользовавшись информацией

нижнего стека, что «+» в вершине дерева представляет сложение целого и вещественного значений. Мы можем переписать выражение, расставив действия *A1, A2* и *A3* в том порядке, в каком они будут вызываться при трансляции этого выражения:

*a<A1> * <A2> b <A1 > <A3> + <A2> x <A1> * <A2> y <A1> <A3> <A3>*

Действие *A3* соответствует применению знака операции. Из изложенного выше вытекает, что каждый вызов *A3* соответствует тому месту, где появился бы знак операции в постфиксной форме. Стек знаков операций, по существу, служит для формирования постфиксной нотации. Поэтому последовательность действий при трансляции данного выражения должна быть следующей:

A1. Поместить статические характеристики *a* в нижний стек.

A2. Поместить знак «*» в стек знаков операций.

A1. Поместить статические характеристики *b* в нижний стек.

A3. Извлечь статические характеристики *a* и *b* из нижнего стека и знак «*» из стека знаков операций, генерировать код для умножения двух целых чисел, поместить статические характеристики результата в нижний стек; тип результата – целый.

A2. Поместить знак «+» в стек знаков операций.

A1. Поместить статические характеристики *x* в нижний стек.

A2. Поместить знак «*» в стек знаков операций.

A1. Поместить статические характеристики *y* в нижний стек.

A3. Извлечь статические характеристики *x* и *y* из нижнего стека и знак «*» из стека знаков операций, генерировать код для умножения двух вещественных чисел, поместить статические характеристики результата в нижний стек; тип результата – вещественный.

A3. Извлечь два верхних элемента из нижнего стека и знак «+» из стека знаков операции, генерировать код для сложения целого и вещественного значений, поместить статические характеристики результата в нижний стек; тип результата — вещественный.

Действия *A1, A2, A3* и вышеприведенную грамматику легко расширить, что

позволит использовать

а) большее число уровней приоритета для знаков операций;

б) унарные знаки операций.

Другие случаи употребления нижнего стека рассматриваются далее.

Нижний стек обеспечивает передачу информации вверх по синтаксическому дереву. Для передачи же информации вниз по дереву применяется так называемый *верхний стек.* Значение в него помещается всякий раз, когда во время генерации кода происходит вход в такую конструкцию, как присваивание или описание идентификатора. При выходе из этой конструкции значение из стека удаляется. Следовательно, генератор кода может заключить, например, что компилируемое выражение находится справа от знака присваивания; эта информация способствует оптимизации.

Еще одной структурой данных, которая требуется во время генерации кода, является таблица блоков [4].

Таким образом, во время генерации кода используются следующие основные структуры данных: нижний стек, верхний стек, стек знаков операций, таблица блоков и, кроме того, таблица видов и таблица символов из предыдущих проходов.

Покажем, как генерируется код для некоторых конструктов, типичных для языков программирования высокого уровня.

1. Присваивания

Присваивание имеет вид *destination = source*

Смысл его состоит в том, что значение, соответствующее *источнику,* присваивается значению, которое является адресом (или именем), заданным *получателем.* Например, в

$p = x + y$

значение *«x + y»* присваивается *p.*

Допустим, что статические характеристики источника и получателя уже находятся в вершине нижнего стека. Опишем действия, выполняемые во время

компиляции для осуществления присваивания. Прежде всего из нижнего стека удаляются два верхних элемента, после чего происходит следующее:

1. Проверяется непротиворечивость типов получателя и источника. Так как получатель представляет собой адрес, источник должен давать что-нибудь приемлемое для присваивания этому адресу. В зависимости от реализуемого языка типы *получателя* и *источника* можно определенным образом изменять до выполнения присваивания. Напри мер, если тип источника – целое число, то его можно сначала преобразовать в вещественное, а затем присвоить адресу, имеющему тип вещественного числа.

2. Там, где это необходимо, проверяются правила области действия. Например, в некоторых языках программирования источник не может иметь меньшую область действия, чем получатель. Однако в процессе компиляции нельзя обнаружить все нарушения правил области действия, и в некоторых случаях для проверки этой области приходится создавать код во время прогона.

3. Генерируется код для присваивания, имеющий форму

ASSIGN type, S, D

где *S* — адрес источника*, а D* — адрес получателя.

4. Если язык ориентирован на выражения (т.е. само присваивание имеет значение), статические характеристики этого значения помещаются в нижний стек.

2. Условные зависимости

Почти все языки содержат условное выражение или оператор, аналогичный следующему:

if B then C else D fi

При генерации кода для такой условной зависимости во время компиляции выполняются три действия. Грамматика с включенными действиями:

CONDITIONAL → if B <A1> then C <A2> else D <A3> fi

Действия *A1, A2, A3* означают *(next* – значение номера следующей метки, присваиваемое компилятором):

A1. Проверить тип *B,* применяя любые необходимые преобразования (приведения) типа для получения логического значения. Выдать код для перехода к *L<next>,* если *B* есть «ложь»:

JUMPF L<next>,<address of B>

Поместить в стек значение *next* (обычно для этого служит стек знаков операций). Увеличить *next* на 1. (Угловые скобки (<,>), в которые заключаются *«next»* и *«address of B»,* используются для обозначения значений этих величин, и их не следует путать со скобками, в которые заключаются действия в порождающих правилах грамматики.)

A2. Генерировать код для перехода через ветвь else (т. е. перехода к концу условной зависимости)

GOTO L<next>

Удалить из стека номер метки (помещенный в стек действием *A1),* назвать *i*, генерировать код для размещения метки

SETLABEL L<i>

Поместить в стек значение *next.* Увеличить *next* на 1.

A3. Удалить из стека номер метки (*j*). Генерировать код для размещения метки

SETLABEL L<j>

Если условная зависимость сама является выражением (а не оператором), компилятор должен знать, где хранить его значение, независимо от того, какая часть вычисляется – *then* или *else***.** Это можно сделать, специфицируя адрес, который указывает на данное значение, или. пересылая значение, заданное частью *then* либо частью *else***,** по указанному адресу.

Аналогично можно обращаться с вложенными условными выражениями или

операторами.

3. Описания идентификаторов

Допустим, что типы всех идентификаторов полностью выяснены в предыдущем проходе и помещены в таблицу символов. Адреса распределяются во время прохода, генерирующего код.

Рассмотрим описание *type x*

Перечислим действия, выполняемые во время компиляции:

1. В таблице символов производится поиск записи, соответствующей *x*.

2. Текущее значение указателя стека идентификаторов дает адрес, который нужно выделить для *x*. Этот адрес

(idstack, current block number, idstack pointer)

включается в таблицу символов, а указатель стека идентификаторов увеличивается на статический размер значения, соответствующего *x*.

3. Если *x* имеет динамическую часть, например в случае массива, то генерируется код для размещения динамической памяти во время прогона.

4. Циклы

Рассмотрим следующий простой цикл:

for i to 10 do something od

Для генерации кода требуются четыре действия, которые размещаются следующим образом:

for i <A1> to 10 <A2> do <A3> something <A4> od

Эти действия таковы:

A1. Выделить память для управляющей переменной *i*. Поместить сначала в эту память 1

*MOVE «*1*», address (controlled variable)*

A2. Генерировать код для записи в память значения верхнего предела

рабочего стека

MOVE address (ulimit), (wostack, current block number, wostack pointer)

(wostack pointer – указатель рабочего стека). Увеличить указатель рабочего стека и уменьшить указатель нижнего стека, где хранились статические характеристики верхнего предела.

A3. Поместить метку

SETLABEL L<next>

Увеличить *next* на 1.

Выдать код для сравнения управляющей переменной с верхним пределом и перейти к *L<next>,* если управляющая переменная больше верхнего предела:

JUMPG L<next>, *address (controlled variable), address (ulimit)*

Поместить в стек значение *next.* Поместить в стек значение *next* - 1. Увеличить *next* на 1.

A4. Генерировать код для увеличения управляющей переменной

PLUS address (controlled variable), 1

Удалить из стека номер (*i*). Генерировать код для перехода к *L < i >*

GOTO L<i>

Удалить из стека номер метки (*j*). Поместить метку в конец цикла

SETLABEL <j>

Цикл *for i to 10 do something od*

генерирует код следующего вида:

MOVE «1», *address (controlled variable)*

MOVE address (ulimit), wostack pointer

SETLABEL L1

JUMPG L2, address (controlled variable), address (ulimit)

(something)

PLUS address (controlled variable), 1

GOTO L1

SETLABEL L2

Действие *A4* можно видоизменить, если приращение управляющей переменной будет не стандартным, равным 1, а иным, например в

for j by 5 to 10 do

Для этого, возможно, придется вычислять выражение и хранить его значение в рабочем стеке, чтобы использовать как приращение.

Аналогично в

for i from m + n to 20 do

начальное значение управляющей переменной может быть выражением, значение которого хранилось в рабочем стеке. Можно также включить случай с отрицательным приращением управляющей переменной. Если в цикле содержится часть *while*, как, например, в

for i to 10 while a<b do

действие *A3* следует видоизменить, чтобы при принятии решения о выходе из цикла учитывались значения как части *while*, так и управляющей переменной, причем любая из этих проверок достаточна для завершения цикла.

6. СИСТЕМЫ ПОСТРОЕНИЯ ТРАНСЛЯТОРОВ

Инструментальным средством автоматизации процесса разработки транслятора с языка программирования является система построения трансляторов (СПТ), содержащая набор программ для реализации примитивов (элементарных конструкций), входящих в большинство трансляторов [3, 11, 12, 13].

Чаще всего СПТ используются для автоматизации разработки компиляторов. СПТ, разработанная для построения компиляторов, представляет собой программу, которая компилирует другие компиляторы.

В настоящее время можно выделить следущие классы СПТ: компиляторы компиляторов, синтаксически ориентированные символьные процессоры (трансляторы), расширяющиеся языки с соответствующими расширяющимися трансляторами [11].

Компиляторы компиляторов предназначены для разработчиков компиляторов и поэтому содержат примитивы, которые облегчают оптимизацию программы, распределение памяти и пр.

Синтаксически ориентированные трансляторы являются символьными процессорами более широкого применения и используются в таких задачах, как упрощение алгебраических выражений, символьное дифференцирование и пр. Символьные процессоры используются и для построения компиляторов, но результатом работы такого процессора обычно является текст на автокоде, который необходимо еще раз транслировать. Как правило, их применение оправдано в тех случаях, когда структура входной информации несет основную содержательную нагрузку и когда эта структура может быть описана в терминах, близких к формам Бэкуса.

Расширяющийся язык позволяет определить новые типы данных и новые операторы в терминах существующих конструкций. Это означает, что программист может расширить и изменить язык, тем самым приспособить его к своим нуждам. Существуют языки, которые допускают ограниченные расширения. Истинно расширяющиеся языки позволяют добавлять новые конструкции или операторы с почти произвольным синтаксисом. Такие расширения связаны с идеей макрокоманд в языках высокого уровня.

Некоторые СПТ являются комбинированными, что позволяет их использовать, например, и как компилятор с расширяющимся языком, и как компилятор компиляторов. Пример реализации СПТ в виде расширяющегося языка приведен в [11].

В последнее время все большее распространение приобретают так называемые атрибутные СПТ, входной метаязык которых основывается на атрибутных грамматиках, предложенных Д.Кнутом [1–3, 12]. В атрибутной грамматике, называемой также атрибутной схемой, синтаксис входного языка описывается КС-грамматикой, а семантика языковых конструкций – множеством их атрибутов и набором семантических функций, позволяющих вычислять значения одних атрибутов по известным значениям других. Примеры построения атрибутных СПТ приведены в [3, 12].

ЗАКЛЮЧЕНИЕ

В конспекте лекций рассмотрены алгоритмы построения трансляторов традиционной или классической структуры, базирующиеся на табличных методах синтаксического анализа (*LL*-метод, *LR*-метод и методы, основанные на предшествовании). Несомненным достоинством классических трансляторов является то, что они сравнительно эффективны в том смысле, что всегда может быть достигнут любой желаемый баланс между временем трансляции и качеством рабочей программы.

Рассмотрены вопросы генерации кода, распределения памяти и некоторые другие, которые приходится решать при разработке трансляторов. Подробное изложение данных проблем можно найти в [1–7].

СПИСОК ЛИТЕРАТУРЫ

1. *Ахо А., Сети Р., Ульман Дж.* Компиляторы. Принципы, технологии, инструменты. – М.: Изд-во «Вильямс», 2001.
2. *Карпов Ю.Г.* Теория и технология программирования. Основы построения трансляторов. – С-Пб.: «БХВ-Петербург», 2005.
3. *Компаниец Р.И., Маньков Е.В., Филатов Н.Е.* Системное программирвоание. Основы построения трансляторов. – СПб.: КОРОНАпринт, 2000.
4. *Хантер Р.* Проектирование и конструирование компиляторов. – М.: Финансы и статистика, 1984.
5. *Грис Д.* Конструирование компиляторов для цифровых вычислительных машин. – М.: Мир,1975.
6. *Лебедев В.Н.* Введение в системное программирование. – М.:Статистика, 1975.
7. *Ахо А., Ульман Дж*. Теория синтаксического анализа, перевода и компиляции. Том 1,2. – М., Мир, 1978.
8. *Кнут Д.* Искусство программирования для ЭВМ. Т.3. Сортировка и поиск. – М.: «Мир», 1978.
9. *Полетаева И.А.* Методы трансляции. – Новосибирск,: Изд-во НГТУ, 1998.
10. Языки программирования и методы трансляции. Методические указания. – Новосибирск, Изд-во НГТУ, 2005.
11. *Сивохин А.В.* Автоматизация построения трансляторов и синтез программ. – Пенза, 1987.
12. *Касьянов В.Н., Поттосин И.В.* Автоматизация построения трансляторов. – Новосибирск, 1983.
13. *Парамарчук Г.О., Чижик В.А., Волковысский В.Л.* Теория синтаксического анализа, перевода и компиляции. – Рязань, РРТИ, 1984.

СОДЕРЖАНИЕ