МЕТОДЫ ТРАНСЛЯЦИИ

Конспект лекций

Часть 1

Новосибирск
1997

Конспект лекций быть использован студентами специальности 01.02 факультета ПМИ при изучении курса «Программное обеспечение ЭВМ и методы трансляции»

Составила к.т.н. И.А. Полетаева

Рецензент Т.А. Шапошникова.

Новосибирский государственный технический университет, 1997 г.

Введение

Данное учебное пособие предназначено студентам ФПМИ, изучающим курс «Программное обеспечение ЭВМ. Методы трансляции». Часть 1 данного пособия содержит описание методов проектирования сканера и синтаксических анализаторов нисходящего типа. Описание методов восходящего синтаксического анализа содержится во второй части учебного пособия.

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.

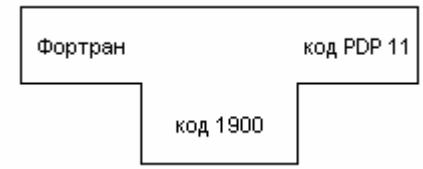
Обычно процессы трансляции и исполнения программы разделены по времени. Сначала вся программа транслируется, а затем исполняется. Трансляторы, работающие в таком режиме, называются трансляторами **компилирующего** типа. Принцип, альтернативный компилированию, реализуется в программах, обычно называемых **интерпретаторами**, которые незамедлительно выполняют текущие предложения языка программирования.

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 и могут применяться в качестве адресов перехода. Ассемблер выполняет довольно простую задачу. 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. Однако это не всегда так.

Для представления компилятора можно воспользоваться Т-образной схемой. Например, компилятор для ПЛ/1, написанный в коде ICL 1900 с целью получения кода 1900, можно изобразить в виде:

Кросс-компилятор выдает код, для какой либо другой машины, а не основной машины. Например, представим компилятор, написанный в коде ЭВМ ICL 1900 для компиляции программ на Фортране в код машины PDP-11. С помощью такого компилятора программы, написанные на Фортране, могут компилироваться на ЭВМ 1900, а пропускаться на ЭВМ PDP-11.



В настоящее время предпринимаются попытки как можно больше обособлять зависимые от машины части компилятора. Это особенно важно в тех случаях, когда компилятор предполагается перемещать с одной машины на другую.

Так, если бы у нас был написанный на Фортране компилятор Паскаля, который должен выработать код, предназначенный для машины А (рис.1), то, чтобы пропустить компилятор на машине А, необходимо, прежде всего, транслировать его в А-код с помощью

компилятора Фортрана, написанного в А-коде и выдающего А-код (рис. 2)

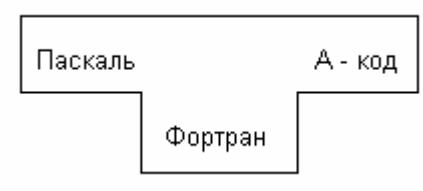


Рис. 1

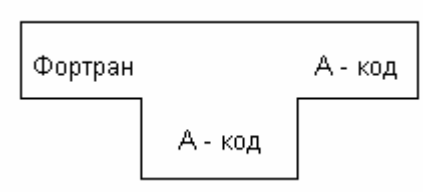


Рис. 2

Из этих двух компиляторов можно получить третий, используя, второй компилятор для компиляции первого (рис. 3)

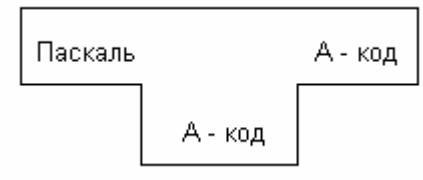


Рис. 3

Это можно показать, соединив вместе две Т-образные схемы в соответствии с правилами, согласно которым ветви среднего звена Т должны показывать те же языки, что и корни, смежных с ними соседних звеньев, а в двух верхних звеньях в правых и левых углах должны быть, записаны одинаковые языки. (рис. 4)

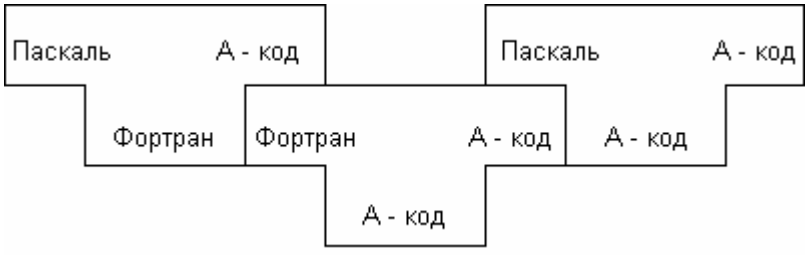


Рис. 4

Можно таким образом получить более сложение Т-образные схемы. Идея Т-образной схемы принадлежит Брэтману [4].

1.2. Основные компоненты трансляции

В состав любого компилятора входят три основных компонента:

- лексический анализатор (блок сканирования)
- синтаксический анализатор
- генератор кода машинных команд

На фазе лексического анализа исходный текст программы разбивается на единицы, называемыми **лексемами**. Такими текстовыми единицами являются слова (например, IF, DO, BEGIN, и др.), имена переменных, константы и знаки операций. Далее эти слова рассматриваются как неделимые образования, а не как группы отдельных символов. После разбиения программы на лексемы следует фаза **синтаксического анализа**, называемая **грамматическим разбором**, на котором проверяется правильность следования операторов. Например, для предложения IF, имеющего вид

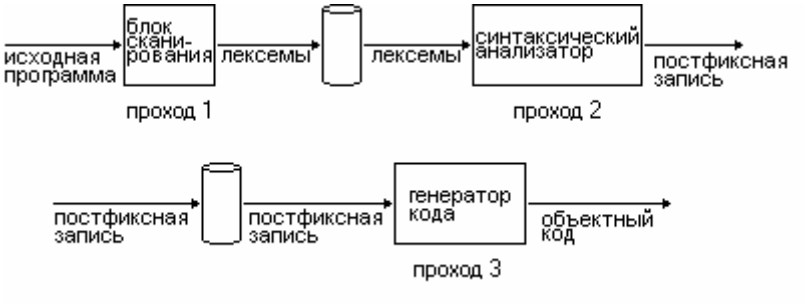
« IF выражение THEN предложение;»

грамматический разбор состоит в том, чтобы убедится, что вслед за лексемой IF следует правильное выражение, за этим выражением следует лексема THEN, за которой следует в свою очередь правильное предложение, оканчивающиеся знаком ‘**;**’.

Последним выполняется процесс **генерирования кода**, который использует результаты синтетического анализа и создает программу на машинном языке, пригодную к выполнению.

Взаимодействие трех компонентов может осуществляться различными способами.

1) компилятор с тремя проходами



2) компилятор с двумя проходами

3) компилятор с одним проходом

В первом варианте блок сканирования считывает исходную программу и представляет ее в виде **файла лексем.** 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 выступает в роли **основной управляющей программы**, вызывая блок сканирования и генератор кода, организованные в виде подпрограмм. Синтаксический анализатор постоянно обращается к блоку сканирования, получая от него лексему за лексемой 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.

- 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 формула D=A+(E+F), может быть утрачена.

- Трудность в решении проблемы перехода по метке.

Во время обработки предложения «GOTO метка» могут возникнуть осложнения, так как «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 лексему за лексемой 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 многие 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 недействительной в каком либо смысле (т.е. принадлежит ли она к 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 представлена 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, такие, как Бейсик 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.

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.

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.

1.3. Некоторые аспекты процесса компиляции.

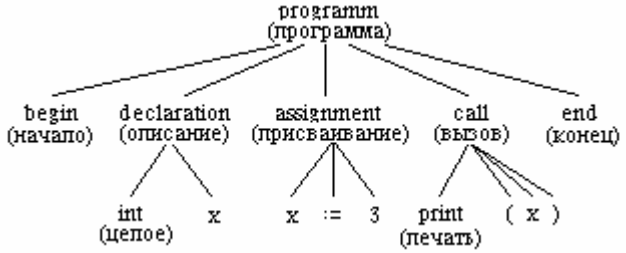
Работа компилятора складывается из двух основных этапов [4]. Сначала он распознает структуру и значение программы, которую он должен компилировать, а затем он выдает эквивалентную программу в машинном коде (или коде сборки). Эти два этапа называются анализ и синтез.

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. Так, 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 идентификаторы (print) или слова языка (begin, end). Конечную вершину с меткой print можно было бы заменить поддеревом:

однако это усложняет структуру. 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.

Для проверки определенных требований языков компилятор строит таблицы. Из объявления int x в таблицу, обычно называемую таблицей символов, вводится элемент, указывающий тип x. Таблицы символов обычно строятся параллельно с синтаксическим анализом. Они требуются во время генерирования кода, а также для проверки действенности программ.

Могут оказаться необходимыми и некоторые другие таблицы. Во время лексического анализа идентификаторы переменной длины обычно заменяются символами фиксированной длины, а для нахождения соответствия между этими символами и первоначальными идентификаторами используются таблицы идентификаторов. В таких языках как АЛГОЛ 68, где комплексные виды могут определяться пользователем, подробная информация о них содержится в таблице видов.

В процессе компиляции, на этапе синтеза, требуется выделить память в объектной машине для записи значений, необходимых программе. Объем памяти, выделенный во время компиляции, обычно называют статическим, а объем памяти, выделяемый во время прогона - динамическим.

Запоминающее устройство, как правило, организуется по принципу магазина и называется стеком.

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.

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. ГРАММАТИКИ И ЯЗЫКИ.

Содержание данного раздела носит вспомогательный характер и позволяет значительно упростить чтение и понимание последующего материала [3].

Дадим несколько определений

Алфавит – это непустое конечное множество элементов. Элементы алфавита называются символами.

Цепочка – всякая конечная последовательность символов алфавита.

Если x и y – цепочки, то их *конкатенацией* (или *катенацией*) xy является цепочка, полученная путем дописывания символов цепочки y вслед за символами цепочки x.

Множества цепочек в алфавитах будем обозначать как A, B, C,... .

Произведение AB двух множеств цепочек A и B определяется как $AB = \{xy \mid x \in A, y \in B\}$.

$\{\Lambda\}$ – множество, состоящее из пустого символа Λ.

Степени цепочки x определяются следующим образом:

$x^0 = \Lambda,\ x^1 = x,\ x^2 = xx,\ x^3 = xxx, \ldots.$

Степени алфавита A:

$A^0 = \{\Lambda\},\ A^1 = A,\ A^n = AA^{n-1},\ (n > 0)$.

Итерация A^* множества A: $A^* = A^0 \cup A^+$

Усеченная итерация A^+ мнжества A:

$A^+ = A^1 \cup A^2 \cup \ldots \cup A^n \cup \ldots$

Продукцией или *правилом* подстановки называется упорядоченная пара (U, x), которая обычно записывается $U ::= x$, где U - символ, x - непустая конечная цепочка символов. U называется *левой частью*, x - *правой частью* продукции.

Далее вместо термина «продукция» будем использовать термин «правило».

Грамматикой $G[x]$ называется конечное, непустое множество правил, z - символ, который должен встретиться в левой части, по крайней мере, одного правила. Он называется *начальным символом*, *аксиомой* или *помеченным символом*.

Все символы, которые встречаются в левых и правых частях правил, образуют *словарь* V, Если из контекста ясно, какой символ является символом z, то вместо $G[z]$ будем писать G.

В заданной грамматике G символы, которые встречаются в левой части называются *нетерминальными* или *синтаксическими*

единицами языка. они образуют множество нетерминальных символов V_N. Символы, которые не входят в V_N, называются *терминальными символами* (или *терминалами*). Они образуют V_T.

$$V = V_N \cup V_T$$

Как правило, нетерминалы будем заключать в угловые скобки <>.

Множество правил $U::=x, U::=y, \ldots, U::=z$ с одинаковыми левыми частями будем записывать $U::=x \mid y \mid \ldots \mid z$.

Пример.

Грамматика **G1 <число>** записывается следующим образом:

<число> ::= <чс>

<чс> ::= <чс><цифра> / <цифра>

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

Эта форма записи называется нормальной формой Бэкуса (НБФ) или формой Бэкуса-Наура.

Пусть G- грамматика. Мы говорим, что цепочка v непосредственно порождает цепочку w, и обозначаем это $v \Rightarrow w$, если для некоторых цепочек x и y можно написать $v = xUy$, $w = xuy$, где $U::=u$ - правило грамматики G. Мы также говорим, что w *непосредственно выводима из* v или что w *непосредственно редуцируется (приводится) к* v

Цепочки x и y могут, конечно, быть пустыми. Следовательно, для любого правила $U::=u$ грамматики G имеет место $U \Rightarrow u$.

Говорят, что v *порождает* w или w *приводится к* v, что записывается как $v \Rightarrow +w$, если существует последовательность непосредственных выводов

$v = u0 \Rightarrow u1 \Rightarrow u2 \Rightarrow \ldots \Rightarrow un = w, n > 0$.

Эта последовательность называется *выводом длины* n. Говорят, что w является словом для v. И, наконец, пишут $v \Rightarrow *w$, если $v \Rightarrow +w$ или $v \Rightarrow w$.

Заметим, что пока в цепочке есть хотя бы один нетерминал, из нее можно вывести новую цепочку.

Терминал (конечный, заключительный) – символ, который не встречается в левой части ни одного из правил.

Пусть $G[z]$ - грамматика. Цепочка x называется *сентенциальной формой*, если x выводится из z, т. е. $z \Rightarrow *x$.

Предложение – это сентенциальная форма, состоящая из терминальных символов.

Язык $L(G[z])$ - это множество предложений.

Хомский впервые в 1956г. описал формальный язык. С тех пор теория формальных языков быстро прогрессировала. Хомский определил четыре основных класса языков в терминах грамматик, являющихся упорядоченной четверкой (V,T,P,z), где V - алфавит, $T \subseteq V$ - алфавит терминальных символов, P - конечный набор правил подстановки, z - начальный символ, $z \in (V-T)$.

Язык, порожденный грамматикой – это множество терминальных цепочек, которые можно вывести из z.

Различие четырех типов грамматик заключается в форме правил подстановки, допустимых в P.

Говорят, что G - это (по Хомскому) грамматика типа 0 или грамматика с фразовой структурой, если правила имеют вид:

$$u ::= v, \text{ где } u \in V^+ \text{ и } v \in V^*$$

То есть левая часть, может быть тоже последовательностью символов, а правая часть может быть пустой. Языки этого класса могут служить моделью естественных языков.

Грамматика типа 1 (контекстно-чувствительные, или контекстно-зависимые языки) имеют правила подстановки вида:

$$xUy ::= xuy, \text{ где } U \in (V-T),\ u \in V^+ \text{ и } x,y \in V^*$$

«Контекстно-чувствительная» отражает тот факт, что U можно заменить на u лишь в контексте $x \ldots y$.

Грамматика называется контекстно-свободной – типа 2 (КС-грамматика), если все ее правила имеют вид:

$$U ::= u, \text{ где } U \in (V-T), \text{ и } u \in V^*$$

Грамматика типа 2 является хорошей моделью для языков программирования.

Регулярная грамматика (тип 3 или автоматная грамматика, А-грамматика.) – это грамматика, правила которой имеют вид:

$$U ::= N \text{ или } U ::= WN, \text{ где } N \in T \text{ а } U,W \in (V-T)$$

Регулярные грамматики играют основную роль, как в теории языков, так и в теории автоматов.

В реальных языках программирования отдельные подмножества можно отнести к третьему классу.

Иерархия языков по Хомскому включающая, т.е. все грамматики типа 3 являются грамматиками типа 2, все грамматики типа 1 являются грамматиками типа **O** и т.д. Иерархия грамматик соответствует иерархии языков. Например, если язык можно генерировать с помощью грамматики типа 2, то его называют языком типа 2. Эта иерархия опять включающая. Включения так же справедливы в том, что существуют языки, которые относятся к типу **i**, но не к типу **(i+1)(0<=i=>2)**.

Иерархия Хомского важна с точки зрения языков программирования. Чем меньше ограничений в грамматике, тем сложнее ограничения, которые можно наложить на генерируемый язык. Таким образом, оказывается, что чем более универсален язык используемой грамматики, тем больше средств типичных языков программирования можно описать.

Однако чем универсальнее грамматика тем сложнее должна быть машина (или программа), которая применяется для распознавания строк соответствующего языка.

С грамматикой типа 3 ассоциируется класс распознавателей, известный как конечный автомат, или машина с конечным числом состояний, между которыми происходит передача управления по мере считывания символов строки, причем строка принимается или нет в зависимости от того, какого состояния машина достигает в итоге. Для языка, генерируемого с помощью КС -грамматики, необходим (вообще) автомат магазинного типа, т.е. конечный автомат плюс стек, а для контекстно-зависимых языков - линейный автомат с ограничениями, т.е. машина Тьюринга с конечным объемом ленты. Наконец, языку типа 0 требуется машина Тьюринга в качестве распознавателя.

Синтаксические деревья помогают понять синтаксис предложения.

Рассмотрим пример:

Грамматика в **G1[<число>]** содержит правила

< число> ::= < чс >

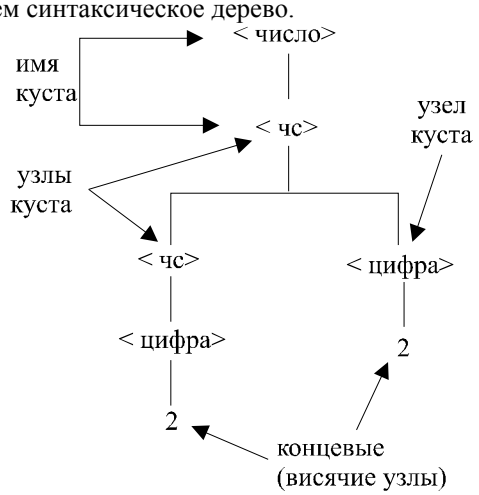
<чс> ::= <чс> <цифра> / <цифра>

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

Приведем пример вывода числа ***22***:

<число> ⇒ <чс> ⇒ <чс> <цифра> ⇒ <цифра><цифра> ⇒ 2<цифра> ⇒ 22

Таким образом, **<число> ⇒ +22** и длина вывода **=5**. Для данного вывода нарисуем синтаксическое дерево.



При чтении слева направо концевые узлы образуют цепочку, вывод которой представлен деревом.

Можно восстановить вывод по синтаксическому дереву при помощи обратного процесса. Самый правый концевой куст указывает непосредственный вывод: **2 <цифра> ⇒ 22**. Этот вывод получаем, отсекая правый концевой куст.

Продолжаем процесс, всегда восстанавливая последний непосредственный вывод, на который указывает концевой куст синтаксического дерева, и затем отсекая этот куст, до достижения корня дерева.

3. ДВЕ СТРАТЕГИИ РАЗБОРА.

Разбор сентенциальной формы означает построение вывода и, возможно, синтаксического дерева для нее. Программу разбора называют также ***распознавателем***, так как она распознает только предложения рассматриваемой грамматики.

Различают два типа алгоритмов разбора: нисходящий (сверху вниз, развертка), и восходящий (снизу-вверх, свертка). Эти термины соответствуют способу построения синтаксических деревьев. При нисходящем разборе дерево строится от корня (начального символа) вниз к концевым узлам. Метод восходящего разбора состоит в том,

что , отправляясь от заданной цепочки, пытаются привести её к начальному символу.

На каждом шаге анализа нисходящий распознаватель формирует цель – найти вывод, начинающийся с некоторого нетерминального символа и порождающий часть входной строки. Распознаватель пытается достичь этой цели путем целенаправленного перебора различных возможностей. Предположим, что входной язык относится к классу 2 по Хомскому, и его грамматика контекстно - свободная.

Основная идея нисходящего анализа состоит в следующем.

Начиная процесс анализа входной строки $s_1 s_2 \dots s_n$ распознавалель исходит из предположения, что эта строка является предложением входного языка. Отсюда вытекает основная цель анализа – найти вывод

$$A \Rightarrow + s_1 s_2 \dots s_n , \qquad (1)$$

где A – начальный символ грамматики.

Если существует вывод (1), то имеется порождающее правило $A \Rightarrow u_1 u_2 \dots u_m , u_i \in V$ и вывод $u_1 u_2 \dots u_m \Rightarrow + s_1 s_2 \dots s_n$. Среди символов u_i могут быть терминальные u_T и нетерминальные u_N

Поскольку выходная строка по определению состоит только из терминальных символов, а левая часть любого правила содержит только u_N для каждого нетерминального символа u_N должен существовать вывод

$$u_N \Rightarrow + s_{N_1} s_{N_2} \dots s_{N_k} , \qquad (2)$$

где $s_{N_1} s_{N_2} \dots s_{N_k}$ – часть входной строки.

Вывод (1) можно найти, если известны выводы (2). Для каждого нетерминального символа u_N, включая и начальный символ A, может быть несколько правил с различными правыми частями. Какое именно правило применять, заранее неизвестно. Поэтому при «неверном» применении правила анализ заходит в «тупик». Для выхода из «тупика» приходится возвращаться к исходной точке и делать новые попытки, что сильно усложняет алгоритм и замедляет процесс трансляции. Процесс завершается, когда найден вывод (1) или когда установлено, что этого вывода не существует (входная строка не является предложением входного языка). Обычно нисходящий распознаватель просматривает символы входной строки

и символы правой части применяемого правила слева направо. Такие распознаватели называются левосторонними.

Пример:

Рассмотрим схему нисходящего анализа предложений языка, порождаемого грамматикой G правила этой грамматики следующие:

<Предложение> ::= <Подлежащие><сказуемое>

<Подлежащие> ::= <имя существительное>|<местоимение>

<имя существительное> ::= КОТ | ПЕС

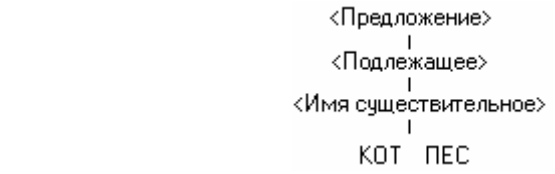
<Местоимение> ::= ОН

<Сказуемое> ::= <глагольная форма>

<Глагольная форма> :: = ИДЕТ | ЛЕЖИТ

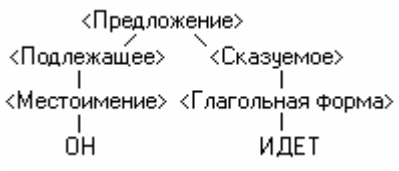
Требуется найти вывод для предложения « ОН ИДЕТ ». В качестве главной цели выбираем символ <Предложение> начальный символ грамматики G.

Первая вспомогательная цель: <Подлежащее> – первый символ правой части для <Предложение>, вторая вспомогательная цель - <Имя существительное> – (правая часть первого правила для <Подлежащее>).



Непосредственная проверка показывает, что вспомогательная цель <Имя существительное> недостижима, так как правые части обоих правил для <Имя существительное> состоят из терминальных символов КОТ и ПЕС, и ни один из них не совпадает с первым символом анализируемой строки.

Поэтому вместо <Имя существительное> выбираем новую вспомогательную цель <Местоимение> - правую часть второго правила для <Подлежащее>



Теперь вспомогательная цель <Подлежащее> достигнута, в результате получена левая ветвь синтаксического дерева. После этого рассматриваем нераспознанную часть синтаксической строки и формируем новую вспомогательную цель <Сказуемое>. Для распознавания <Сказуемое> формируется еще одна вспомогательная цель <Глагольная форма>, которая приводит к достижению главной цели – построению вывода, который порождает анализируемое предложение. В производственных трансляторах широко применяют комбинацию нисходящих и восходящих методов анализа. Например, нисходящий анализ выделяет отдельно крупные синтаксические конструкции (описание, оператор), каждая из которых затем анализируется методами восходящего анализа.

Методы восходящего анализа нашли широкое применение в действующих трансляторах. Общая идея восходящего анализа состоит в следующем. Входная программа рассматривается как строка символов $s_1 s_2 \ldots s_n$. Распознаватель отыскивает часть строки, которую можно привести к нетерминальному символу. Такая часть строки называется *фразой*. Фраза, которая непосредственно приводима к нетерминальному символу называется непосредственно приводимой.

В большинстве восходящих распознавателей отыскивается самая левая непосредственно приводимая фраза, которая называется основной. Основа заменяется нетерминальным символом. Во вновь полученной строке опять отыскивается основа, которая также заменяется нетерминальным символом и т. д.

Процесс продолжается до получения начального символа либо до установления невозможности приведения сроки к начальному символу. Последовательность промежуточных срок, которая заканчивается начальным символом, образуют разбор. Если строка неприводима к начальному символу, то входная программа некорректна.

Пример:

Для строки ОН КОТ в грамматике G фразами являются ОН и КОТ. Основа ОН Приведение к <Местоимение> дает сроку <Местоимение> КОТ с двумя фразами <Местоимение> и КОТ. Далее получаем:
<Местоимение>КОТ, <Подлежащее>КОТ, <Подлежащее><ИС>, <Подлежащее><Подлежащее>

Дальнейшее приведение невозможно, строка синтаксически некорректна.

4. СКАНЕР

4.1. Регулярные выражения и конечные автоматы.

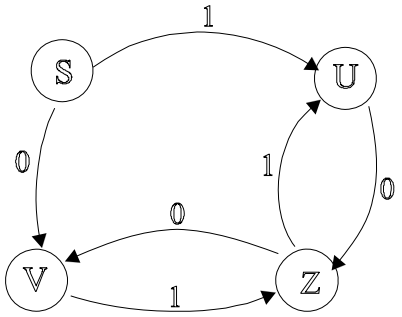
В данном разделе используем результаты без доказательств [3].

Диаграммы состояний.

Рассмотрим регулярную грамматику $G[z]$.

Z :: = U0 / V1
U :: = Z1 / 1
V :: = Z0 / 0
L (G) = { B^n | n >0}, где B= {01, 10}

Легко видеть, что порождаемый ею язык состоит из последовательностей, образуемых парами 01 или 10. Чтобы облегчить распознавание предложений грамматики G, нарисуем диаграмму состояний.



В этой диаграмме каждый нетерминал грамматики G представлен узлом или состоянием; кроме того, есть начальное состояние **S**, предполагается, что G не содержит **S**.

Каждому правилу **Q ::= T** в G соответствует дуга с пометкой **T**, направленная от начального состояния **S** к состоянию **Q**, Каждому правилу **Q ::=RT** соответствует дуга с пометкой **T**, направленная от состояния **R** к состоянию **Q**.

Чтобы распознать или разобрать цепочку x, используем диаграмму состояний следующим образом:

1. Первым текущим состоянием считаем начальное состояние. Начинаем с самой левой литеры в цепочке x повторяем шаг 2 до тех пор, пока не будет достигнут правый конец x.
2. Сканируем следующую литеру сроки x, продвигаемся по дуге помеченной этой литерой, переходя к следующему состоянию.

Если при каком-то повторении шага 2 такой дуги не окажется, то цепочка x не является предложением. Если мы достигаем конца x, то x – предложение тогда и только тогда, когда последнее текущее состояние есть Z.

Последовательность действий соответствует алгоритму восходящего разбора. На каждом шаге (кроме первого) основой является имя текущего состояния, за которым следует входной символ. Символ, к которому приводится основа, будет именем следующего состояния.

Пример:

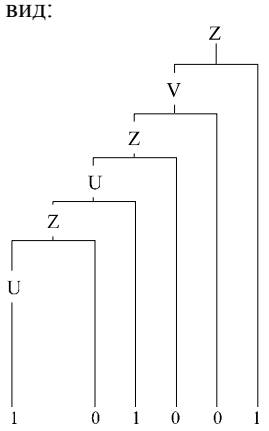
Проведем разбор предложения 101001

шаг	Текущее состояние	Остаток цепочки x
1	S	101001
2	U	01001
3	Z	1001
4	U	001
5	Z	01
6	V	1
7	Z	

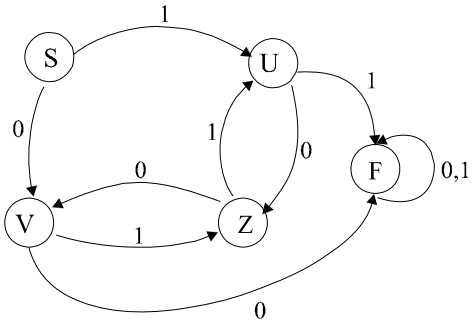
В данном примере разбор выглядит простым благодаря простому характеру правил. Нетерминалы встречаются лишь как первые символы правой части. На первом шаге первый символ предложения всегда приводит к нетерминалу. На каждом последующем шаге первые два символа **UT** сентенциальной формы **UTt** приводят к нетерминалу **V**, при этом используется правило **V ::= UT**. При выполнении этой редукции имя текущего состояния **U**, а имя следующего текущего состояния **V**. Так как каждая правая часть

единственна, то единственным оказывается символ, к которому она приводится.

Синтаксические деревья для предложения регулярных грамматик всегда имеют подобный вид:



Чтобы избавиться от проверки на каждом шаге, есть ли дуга с соответствующей пометкой, можно добавить еще одно состояние **F** (НЕУДАЧА) и добавлять все необходимые дуги от всех состояний к **F**. Добавляется так же дуга, помеченная всеми литерами из **F** в **F**.



4.2. Детерминированный конечный автомат.

Определение: Детерминированный автомат с конечным числом состояний (КА) – это пятерка (K , V_T , M , S , Z).

1) K – алфавит элементов, называемых состояниями;

2) V_T – входной алфавит (литеры, которые могут встретится в цепочке или предложении);

3) M – отображение (или функция) множества $K \times V_T$ в множестве K (если $M(Q,T) = R$, то это значит, что из состояния Q при входной литере T происходит переключение в состояние R);

4) $S \in K$ – начальное состояние;

5) Z – множество заключительных состояний, $Z \subset K$.

Можно формально определить, как работает КА с входной цепочкой t. Определим:

$M(Q, \Lambda) = Q$ при любом состоянии Q. Если на входе пустой символ, состояние не изменится.

$M(Q,Tt) = M(M(Q,T),t)$ для всех $t \in V_T^*$ и $T \in V_T$.

Говорят, что КА допускает цепочку t (цепочка считается допускаемой), если $M(S,t) = P$, где $P \in Z$.

Такие автоматы называются детерминированными, т.к. на каждом шаге входная литера однозначно определяет следующие текущее состояние.

Пример: Рассмотренной ранее диаграмме состояний соответствует КА

M(S,0) = V, M(U,0) = Z, M(F,0) = F, M(S,1) = U,
M(V,0) = F, M(Z,0) = V, M(F,1) = F, M(V,1) = Z,
M(Z,1) = 0, M(V,1) = F.

Если предложение x принадлежит грамматике G, то оно так же допускается КА, соответствующим грамматике G. Для любого КА существует грамматика G, порождающая только те предложения, которые являются цепочками допускаемыми КА.

4.3. Представление в ЭВМ.

КА с состояниями $S_1 S_2 \ldots S_n$ и входными литерами $T_1, T_2, \ldots, T_m$ можно представить матрицей B, состоящей из $n \times m$ элементов. Элемент $B[i, j]$ содержит число k – номер состояния S_k такого, что $M(S_i, T_j) = S_k$. Можно условиться, что состояние S_1 – начальное, а список заключительных состояний представлен вектором. Такая матрица называется матрицей переходов.

Другим способом представления может быть списочная структура. Представление каждого состояния с k дугами, исходящими из него, занимает $2 * k + 2$ слов. Первое слово – имя состояния, второе – значение k, каждая последующая пара слов содержит терминальный символ из входного алфавита и указатель на начало представления состояния, в которое надо перейти по этому символу.

4.4. Недетерминированный КА.

Если грамматика G содержит два правила с одинаковыми правыми частями, то отображение M оказывается неоднозначным. Автомат, построенный по такой диаграмме, называется недетерминированным конечным автоматом и определяется следующим образом:

Недетерминированным КА (НКА) называется пятерка (K, V_T, M, S, Z), где

1. K – алфавит состояний;
2. V_T – входной алфавит;
3. M – отображение $K \times V_T$ в подмножество множества K;
4. $S \subseteq K$ – множество начальных состояний;
5. $Z \subseteq K$ – множество заключительных состояний.

Отличия:

1. Отображение M дает не единственное, а (возможно пустое) множество состояний
2. Может быть несколько начальных состояний.

Как и ранее, $M(Q,\Lambda)=\{Q\}$, $M(Q,Tt)$ есть объединение множеств $M(P,t)$, где $P \in M(Q,T)$.

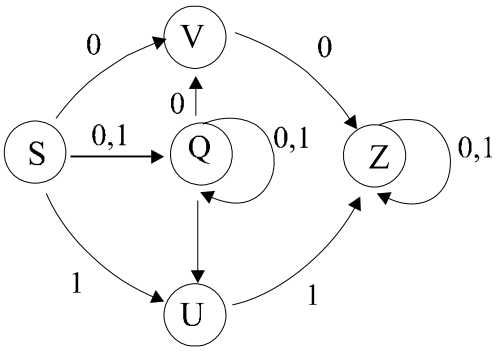
Цепочка t допускается автоматом, если найдется состояние P, такое, что $P \in M(S,t)$ и $P \in Z$.

Пример:

Рассмотрим регулярную грамматику $G[z]$:

Z :: = U1 | V0 | Z0 | Z1
U :: = Q1 | 1
V :: = Q0 | 0
Q ::= Q0 | Q1 | 0 | 1

Диаграмма состояний и ее НКА имеют вид:



NKAF = ({S,Q,V,U,Z}, {0,1}, M, {S}, {Z})

M(S,0)={V,Q}, M(S,1)={U,Q}, M(V,0)={Z}, M(V,1)={ Λ }=$\varnothing$, . . .

Состояние НЕУДАЧА представлено подмножеством $\varnothing$.

4.5. Построение КА из НКА.

Покажем способ построения КА из НКА, при котором как бы параллельно проверяются все возможные пути разбора и отбрасываются тупиковые. Если в НКА имеются, к примеру, выбор из трех состояний x, y, z, то в КА будет одно состояние $[xyz]$, которое представляет все три.

Теорема.

Пусть НКА F =(K ,V_T ,M ,S ,Z) допускает множество цепочек L . Определим КА F' =(K' ,V_T ,M' ,S' ,Z') следующим образом:

1. Алфавит состояний K' состоит из всех подмножеств множества K . Обозначим элемент множества K' через $[s_1, s_2, \ldots, s_i]$, где $s_1, s_2, \ldots, s_i \in K$. Положим, что $\{s_1, s_2\} = \{s_2, s_1\} = [s_1, s_2]$.

2. Множества входных литер V_T для F и F' одни и те же.

3. Отображение M' определим как:

$M'([s_1, s_2, \ldots, s_i], T) = [r_1, r_2, \ldots, r_j]$, где

$M(\{s_1, s_2, \ldots, s_i\}, T) = \{r_1, r_2, \ldots, r_j\}$.

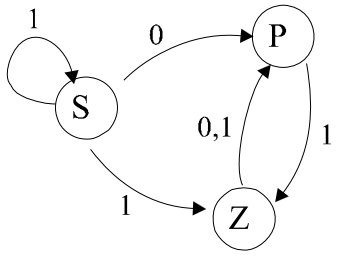
4. Пусть $S = \{s_1, s_2, \ldots, s_n\}$, тогда $S' = [s_1, s_2, \ldots, s_n]$.

5. Пусть $Z = \{s_j, s_k, \ldots, s_l\}$, тогда $Z' = [s_j, s_k, \ldots, s_l]$.

Утверждается, что F и F' допускают одно и тоже множество цепочек.

Пример:

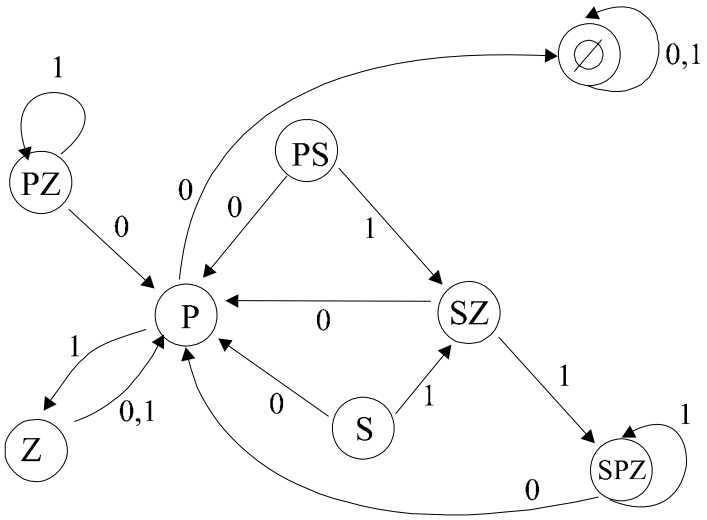
ÍÊÀ



Начальное состояние S .

Заключительное состояние Z .

КА



Начальное состояние S или PS.

Заключительное состояние Z, ZP, SZ, SPZ.

Состояния PS и PZ можно исключить, т.к. нет путей, к ним ведущих. Построенный автомат не является минимальным, возможно построить автомат с меньшим числом состояний.

Для построения КА из НКА воспользуемся следующим приемом.

Строим матрицу переходов для НКА.

	0	**1**
S	**P**	**S,Z**
P		**Z**
Z	**P**	**P**

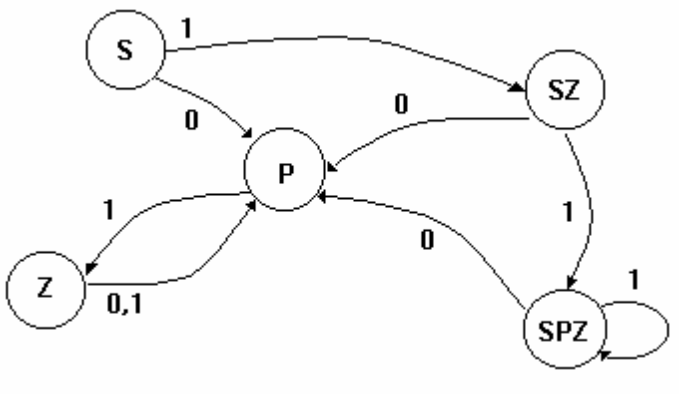
Вводим новое состояние **SZ**

	0	**1**
S	**P**	**SZ**
SZ	**P**	**SZ, P**
P		**Z**
Z	**P**	**P**

Вводим еще одно новое состояние **SPZ**

	0	**1**
S	**P**	**SZ**
SZ	**P**	**SPZ**
SZP SPZ	**P**	**SPZ**
P		**Z**
Z	**P**	**P**

Больше неоднозначных переходов нет, строим граф КА.



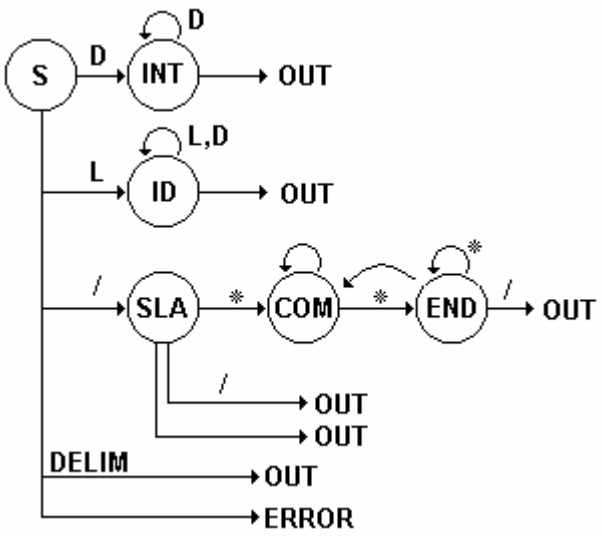
4.6. Программирование сканера.

Покажем, как программируется сканер [3]. Сканер самая простая часть компилятора, иногда называемая лексическим анализатором. Сканер просматривает литеры исходной программы слева направо и строит символы программы - целые числа, идентификаторы, служебные слова. Символы передаются затем на обработку синтаксическому анализатору. На этой стадии может быть исключен комментарий. Сканер также может заносить идентификаторы в таблицу символов и выполнять другую простую работу, которая фактически не требует анализа исходной программы. Он может выполнять большую часть работы по макрогенерации в тех случаях, когда требуется только текстовая подготовка. обычно сканер

передает символы анализатору во внутренней форме. Каждый разделитель (служебное слово, знак операции или знак пунктуации) будет представлен целым числом. Идентификаторы или константы можно представить парой чисел. Первое число, отличное от любого целого числа, используемого для представления разделителя, характеризует сам идентификатор или константу. Второе число является адресом или индексом идентификатора или константы в некоторой таблице. Это позволяет в остальных частях компилятора работать эффективно, оперируя символами фиксированной длины, а не с цепочками литер переменной длины.

4.7. Диаграмма состояний.

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



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 уже его первая литера ил нет. 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.

Для работы сканера потребуются следующие переменные и подпрограммы:

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

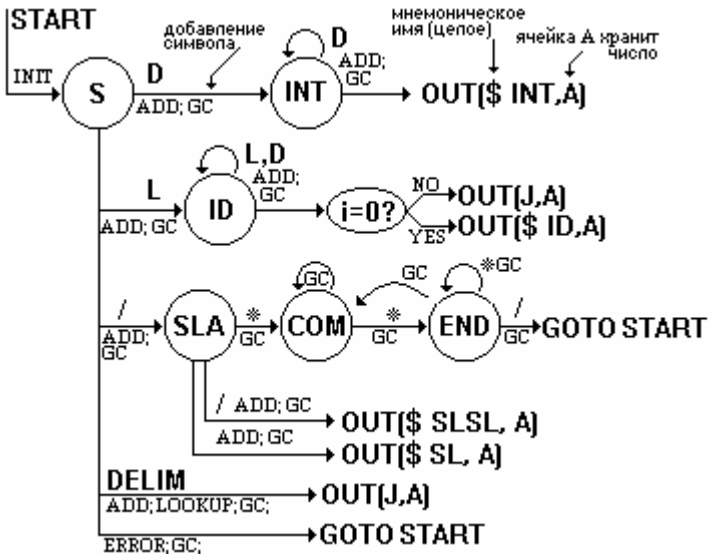
2. INEGER CLASS, где CLASS – содержит целое число, которое характеризует класс литеры находящейся в CHAR. Будем считать, что класс D (цифра)=1, класс L (буква)=2, класс, содержащий литеру / , =3, класс DELIM=4.

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, то 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.

Под 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.

Последим 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), к программе, заканчиваем работу возвратом к программе, которая вызвала сканер, с парой величин

К моменту выхода литера следующего символа находится в CHAR.

Процедура имеет два параметра – внутренне представление получаемого символа, второй цепочка литер, составляющих символ.

5. МЕТОДЫ СИНТАКСИЧЕСКОГО АНАЛИЗА.

5.1. Синтаксический анализ сверху вниз.

Метод рекурсивного спуска – легко реализуемый детерминированный метод разбора сверху вниз [4]. 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 соответствующе 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.

5.2. LL(1)-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 Грейбаха.

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.

Грамматика с порождающими правилами

S → pX
S → qY
X → aXb
X → x
Y → aYd
Y → y

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

S → R
S → T
R → pX
T → qY
X → aXb
X → x
Y → aYd
Y → y

поскольку правые части двух правил не начинаются с терминалов.

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

Пример:

Рассмотрим проблему разбора строки ‘рааахввв’ с помощью S - грамматики. Начав с символа S, попутаемся генерировать строку. Применим левосторонний вывод.

Исходная строка	Вывод
paaaxbbb	S
paaaxbbb	pX
paaaxbbb	paXb
paaaxbbb	paaXbb
paaaxbbb	paaaXbbb
paaaxbbb	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 в 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. LL(1)-грамматика является обобщением S-грамматики. Принцип ее обобщения позволяет строить нисходящие детерминированный анализатор.

LL(1) означает, что строки разбирать 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 → RY

S → TZ

R → paXb

T → qaYd

можно вывести, что порождающее правило **S → RY** желательно применять только при разборе сверху вниз (допуская, что для R нет других порождающих правил), когда предварительно просматриваемым символом является p. Аналогично, порождающее правило **S → RZ** 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 → Ac

P → Bd

A → a

A → aA

B → b

B → bB

а и **в** - символы предшественники **P**.

Определение

Множество символов-предшественников для строки терминалов и /или нетерминалов

$a \in S(x) \Leftrightarrow x \Rightarrow +ay$

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.

Пример:

P → AB

P → BG

A → aA

A → ε

B → c

B → bB

Имеем S(**AB**) = {**a**, **b**, **c**}, поскольку **A** может генерировать пустую строку.

S(**BG**) = {**b**, **c**}

Следовательно, грамматика не будет LL(1). Рассмотрим грамматику с порождающими правилами:

T $\rightarrow$ **AB**
A $\rightarrow$ **PQ**
A $\rightarrow$ **BC**
P $\rightarrow$ **pP**
P $\rightarrow \varepsilon$
Q $\rightarrow$ **qQ**
Q $\rightarrow \varepsilon$
B $\rightarrow$ **bB**
B $\rightarrow$ **e**
C $\rightarrow$ **cC**
C $\rightarrow$ **f**

которая дает

S(**PQ**) = {**p**, **q**}, S(**BC**) = {**b**, **e**}.

Однако, так **PQ** может генерировать пустую сторону, следующим просматриваемым символом при применении порождающего правила **A** $\rightarrow$ **PQ** может быть **b** или **e** (вероятные символа, 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** –- нетерминал, то его направляющими символами будут S(**A**) + (все символы, следующие за **A** , если **A** может генерировать пустую строку).

В более общем случае для заданного варианта x нетерминала **P** (**P** $\rightarrow x$) имеем

DS(**P**, x) = {a | a $\in$ S(x) или ($x \Rightarrow \varepsilon$ и a $\in$ F(**P**))},

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

Так, в приведенном ранее примере направляющие символы –- это символы

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.

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.

Грамматика

A → XYZ
X → PQ
Y → RS
R → TU
P → ε
P → a
Q → aa
Q → ε
S → cc
T → dd
U → ee
Z → ε**.**

После первого прохода массив 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 → XYZ
X → PQ
Y → RS
R → 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 → 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

Далее формируется матрица, показывающая 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 → QR** и **Q → qV** можно заключить, что **Q** есть непосредственный предшественник **P** , а **q** –- непосредственный предшественник **Q**.

В матрице предшественников для каждого нетерминала отводится строка, а для каждого терминала и нетерминала -- столбец. 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 пустой строки. 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.

Пример. Из порождающих правил **P → QR** , **Q → qV** можно заключить, что **q** есть символ (не непосредственный) предшественник **P**. Или же, как вытекает из матрицы непосредственных предшественников, единица в **P**-й строке **Q**-го столбца и единица в **Q**-й строке **q**-го столбца свидетельствуют о том, что если мы хотим сформировать полную матрицу предшественников (а не только непосредственных), нам 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.

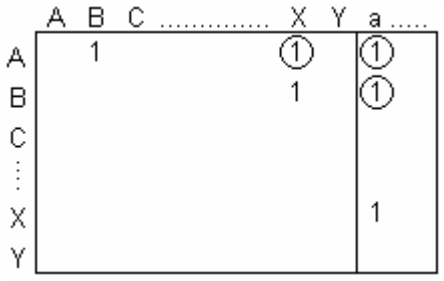
Пример.

Приведенный алгоритм иллюстрируем множеством порождающих правил:

A → BX

B → XY

X → aa



В таблице не непосредственные предшественники заключены в кружки. Этот процесс известен как нахождение транзитивного замыкания матрицы и у него известен аналог в теории графов.

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 из порождающих правил грамматики.

Пример.

Как вытекает из правила **S → ABC**, **B** может следовать за **A**, и элемент (**A**,**B**) в матрице следования имеет значение 1. Аналогичным образом, **C** может следовать за **B**, и элемент (**B,C**) устанавливается на 1. Если **B** генерирует пустой символ, то естественно заключить, что **C** может следовать за **A**, и элемент (**A,C**)= 1. Менее явно правила **P → QR**, **Q → 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, и, следовательно, нарастить матрицу следования.

На основании массива пустых строк, матрицы предшествования и матрицы следования можно проверить признак LL(1). Там, где в левой части более чем одного правила появляется нетерминал, необходимо вычислять направляющие символы различных альтернативных правых частей. Если для каких-либо из этих нетерминалов различные множества направляющих символов не являются непересекающимися, 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 → Aa

A → 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 → BC

B → CD

C → 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 → Aa

A → Bb

B → Cc

C → Dd

C → e

D → 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 правило не выдерживается для всех правил, если есть левый рекурсивный цикл. При выбранном порядке в правиле **D → Az** **A** предшествует **D** в этом упорядочении. Теперь начнем замещать **A**, пользуясь всеми правилами, имеющими **A** в левой части. В результате получим

D → Bbz

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

D → Ccbz

Затем он повторяется еще раз, получается два правила

D → ecbz

D → Ddcbz.

Теперь преобразованная грамматика имеет вид:

S → Aa

A → Bb

B → Cc

C → Dd

C → e

D → ecbz

D → 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 → ecbz

D → Ddcbz

на

D → ecbZ

D → ecbzZ

Z → dcbz

Z → 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

$A \to A\alpha_1, A \to A\alpha_2, \ldots, A \to A\alpha_r$

$B \to \beta_1, B \to \beta_2, \ldots, B \to \beta_s$

то эти правила можно заменить на следующие

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

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, 4].

Пример.

Порождающие правила

S → aSb

S → aSc

S → ε

можно преобразовать путем факторизации в правила

S → aSX

S → ε

X → b

X → 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.

Пример.

Рассмотрим правила

1. **P → Qx**
2. **P → Ry**
3. **Q → sQm**
4. **Q → q**
5. **R → sRn**
6. **R → r**

Оба множества направляющих символов для двух вариантов **P** содержат **s**, и, пытаясь «вынести **s** за скобки», мы замещаем **Q** и **R** в правых частях правил 1 и 2:

Эти правила можно заменить следующими:

P → qx

P → ry

P → sP$_1$

P$_1$ → Qmx

P$_1$ → Rny

Правила для **P$_1$** аналогичны первоначальным правилам для **P** и имеют пересекающиеся множества направляющих символов. Можно преобразовать эти правила так же, как и правила для **P**:

P$_1$ → sQmmx

P$_1$ → qmx

P$_1$ → sRnny

P$_1$ → rny

Факторизуя, получаем

P$_1$ → qmx

P$_1$ → rny

P$_1$ → sP$_2$

P$_2$ → Qmmx

P$_2$ → Rnny

Правило для **P$_2$** аналогично правилам для **P$_1$** и **P**, но длиннее их, и теперь уже очевидно, что этот процесс бесконечный. Все попытки преобразовать грамматику в LL(1)-форму с помощью какого-либо алгоритма при некоторых входах обречены на неудачу (или зацикливание). Чем замысловатее (и сложнее) алгоритм, тем больше случаев он может охватить, однако всегда найдутся какие-либо входы, с которыми он потерпит поражение.

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 на 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, 4].

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

Пример.

Правило $\mathbf{E \to E+T \mid T}$, обозначающее по крайней мере одно вхождение **T**, за которым следует сколь угодно (в том числе и нуль) вхождений **+T**, можно переписать как $\mathbf{E \to E\{+T\}}$. Правило $\mathbf{E \to T \mid E+T \mid E–T}$ можно записать как $\mathbf{E \to T\{ (+ \mid –) T\}}$, и правило

$\mathbf{T \to T*F \mid T / F \mid F}$ может иметь вид $\mathbf{T \to F \{ * F \mid / F \}}$.

Теперь сформулируем два принципа, на основании которых правила языка, включающие прямую левостороннюю рекурсию, преобразуются в правила, использующие итерацию.

Пример 1. Если существуют правила вида

$\mathbf{U \to xy \mid xw \mid ... \mid xz}$, то их надо заменить на $\mathbf{U \to x (y \mid w \mid ... \mid z)}$, где скобки являются метасимволами.

Допустима факторизация в более общем виде, такая, как в арифметических выражениях.

Пример 2. Если в Примере 1 $\mathbf{y=y_1y_2}$ и $\mathbf{w=y_1w_2}$, то можно заменить $\mathbf{U \to x (y \mid w \mid ... \mid z)}$ на $\mathbf{U \to x (y_1(y_2 \mid w_2) \mid ... \mid z)}$.

Исходные правила $\mathbf{U \to x \mid xy}$ преобразуются к виду $\mathbf{U \to x (y \mid \Lambda)}$, где Λ – пустая цепочка. Когда бы не использовалось подобное преобразование, Λ всегда помещается как последняя альтернатива, так как мы принимаем условие, что если цель – Λ , то эта цель всегда сопоставляется.

Помимо того, что факторизация позволяет исключить прямую рекурсию, использование этого приема сокращает размеры грамматики и позволяет проводить разбор более эффективно.

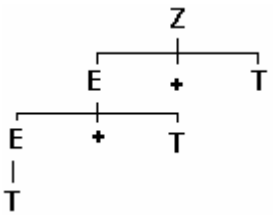
После факторизации в грамматике останется не более одной правой части с прямой левосторонней рекурсией для каждого из нетерминалов. Если такая правая часть есть, делаем следующее:

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

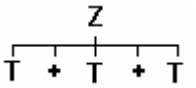
Преобразуем эти правила и виду $\mathbf{U \to (x \mid y \mid \mid z)\{v\}}$.

Пример.

Дерево, использующее рекурсию.
Z → E
E → E+T | T



Пример.
Дерево, использующее итерацию
Z → E
E → T{+T}



Правила **U → Vx** и **V → U y | y** дают вывод **U → +Uyx**.

Избавиться от этого не так просто, но обнаружить такую ситуацию можно.

5.3. LL(1) – таблицы 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 синтаксис анализатора в отношении 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 → begin DECLIST comma STATELIST end

DECLIST → d semi DECLIST

DECLIST → d

STATELIST → s semi STATELIST

STATELIST → 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 → begin DECLIST comma STATELIST end

DECLIST → dX

X → semi DECLIST

X → ε

STATELIST → sY

Y → semi STATELIST

Y → ε

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.

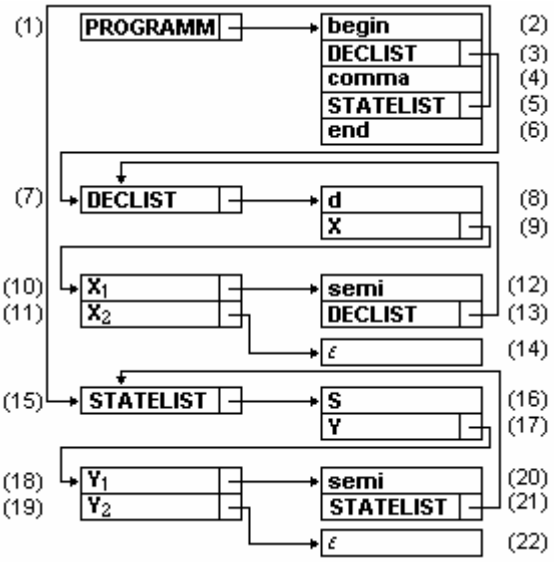


Рис. 1

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

	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.

I	**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 semmi; 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,	6

	перейти к pt[20]	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 эоементов 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 этод 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 неприемлему

символу, а в таблице разбора есть список возможных символов продолжения.

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

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

Список литературы

1. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. – Т.1. Синтаксический анализ. – М.: Мир, 1978.
2. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. – Т.2. Компиляция. – М.: Мир, 1978.
3. Грис Д. Конструирование компиляторов для цифровых вычислительных машин. М.: Мир, 1975.
4. Хантер Р. Проектирование и конструирование компиляторов. – М.: Мир, 1984.
5. Лебедев В.Н. Введение в системное программирование. – М.: Статистика, 1975.

Оглавление