Теория языков программирования

Составил ответы на часть №1 по билетам: Потеренко А.Г. 2006г. (гр. ВТ-31)

Содержание

1. Понятия транслятор, интерпретатор, ассемблер, машинный язык.

Транслятор - это программа, которая переводит исходную программу в эквивалентную ей объектную программу. Исходная программа пишется на некотором исходном языке, объектная программа формируется на объектном языке. Выполнение программы самого транслятора происходит во время трансляции.

Если исходный язык является языком высокого уровня, например таким, как ФОРТРАН, С и Паскаль, и если объектный язык - ассемблер или некоторый машинный язык, то транслятор называется ***компилятором***. ***Машинный язык*** иногда называют кодом машины, поэтому и объектная программа иногда называется объектным кодом.

Трансляция исходной программы в объектную происходит во время компиляции, а фактическое выполнение объектной программы происходит во время выполнения готовой программы.

Ассемблер - это программа, которая переводит исходную программу, написанную на автокоде или на языке ассемблера, на язык вычислительной машины. Автокод (ассемблер) очень близок к машинному языку; действительно, большинство автокодных инструкций является точным символическим представлением команд машины. Более того, автокодные инструкции обычно имеют фиксированный формат, что позволяет легко их анализировать. В автокоде, как правило, отсутствуют вложенные инструкции, блоки и т. п.

Интерпретатор для некоторого исходного языка принимает исходную программу, написанную на этом языке, как входную информацию и выполняет ее. Различие между компилятором и интерпретатором состоит в том, что интерпретатор не порождает объектную программу, которая затем должна выполняться, а непосредственно выполняет ее сам. Для того чтобы выяснить, как осуществить выполнение инструкций исходной программы, чистый интерпретатор анализирует ее всякий раз, когда она должна быть выполнена. Конечно же, это не эффективно и используется не очень часто.

При программировании интерпретатор обычно разделяют на две фазы. На первой фазе интерпретатор анализирует всю исходную программу, почти также, как это делает компилятор, и транслирует ее в некоторое внутреннее представление. На второй фазе это внутреннее представление исходной программы интерпретируется или выполняется. Внутреннее представление исходной программы разрабатывается для того, чтобы свести к минимуму время, необходимое для расшифровки или анализа каждой инструкции при ее выполнении.

Как указывалось выше, сам компилятор - это не что иное, как программа, написанная на некотором языке, для которой входной информацией служит исходная программа, а результатом является эквивалентная ей объектная программа. Исторически сложилось так, что компиляторы писались на автокоде вручную. Во многих случаях это был единственный доступный язык. Однако сейчас компиляторы разрабатываются на языках высокого уровня (поскольку при этом уменьшается время, затрачиваемое на программирование и отладку, а также обеспечивается удобочитаемость программы компилятора, когда работа завершена).

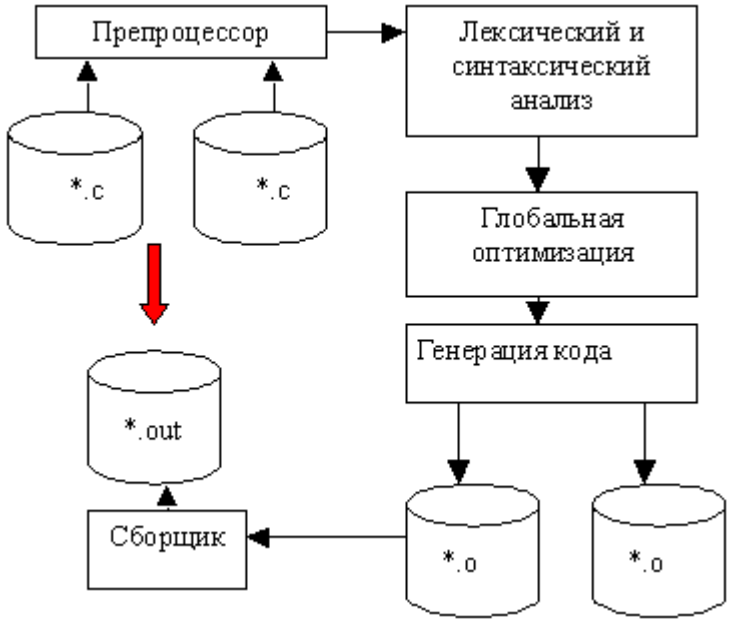
Кроме того, теперь мы имеем много языков, разработанных специально для составления компиляторов. Эти так называемые "компиляторы компиляторов" являются некоторым подмножеством в "системах построения трансляторов" (СПТ).

2. Процесс компиляции, структура компилятора, проходы компилятора.

Процесс компиляции разделяется на несколько этапов:

1. **Препроцессор**.
   Исходная программа обрабатывается путём подстановки имеющихся макросов и заголовочных файлов.
2. **Лексический и синтаксический анализ**.
   Программа преобразовывается в цепочку лексем, а затем во внутреннее представление в виде дерева.
3. **Глобальная оптимизация**.
   Внутреннее представление программы неоднократно преобразовывается с целью сокращения размера и времени исполнения программы.
4. **Генерация кода**.
   Внутреннее представление преобразовывается в блоки команд процессора, которые преобразовываются в ассемблеровский текст или в объектный код.
5. **Ассемблирование**.
   Если генерируется ассемблерный текст, производится его ассемблирование с целью получения объектного кода.
6. **Сборка**.
   Сборщик соединяет несколько объектных файлов в исполняемый файл или библиотеку.

Структура компилятора:



Проходы компилятора:

1. ***На фазе лексического анализа*** (ЛА) входная программа, представляющая собой поток символов, разбивается на лексемы - слова в соответствии с определениями языка. Основным формализмом, лежащим в основе реализации лексических анализаторов, являются конечные автоматы и регулярные выражения. Лексический анализатор может работать в двух основных режимах: либо как подпрограмма, вызываемая синтаксическим анализатором за очередной лексемой, либо как полный проход, результатом которого является файл лексем. В процессе выделения лексем ЛА может как самостоятельно строить таблицы имен и констант, так и выдавать значения для каждой лексемы при очередном обращении к нему. В этом случае таблица имен строится в последующих фазах (например, в процессе синтаксического анализа). На этапе ЛА обнаруживаются некоторые (простейшие) ошибки (недопустимые символы, неправильная запись чисел, идентификаторов и др.).
2. ***Основная задача синтаксического анализа*** - разбор структуры программы. Как правило, под структурой понимается дерево, соответствующее разбору в контекстно-свободной грамматике языка. В настоящее время чаще всего используется либо LL(1)- анализ (и его вариант - рекурсивный спуск), либо LR(1)-анализ и его варианты (LR(0), SLR(1), LALR(1) и другие). Рекурсивный спуск чаще используется при ручном программировании синтаксического анализатора, LR(1) - при использовании систем автоматизации построения синтаксических анализаторов. Результатом синтаксического анализа является синтаксическое дерево со ссылками на таблицу имен. В процессе синтаксического анализа также обнаруживаются ошибки, связанные со структурой программы.

3. На этапе ***контекстного анализа*** выявляются зависимости между частями программы, которые не могут быть описаны контекстно-свободным синтаксисом. Это в основном связи "описание-использование", в частности анализ типов объектов, анализ областей видимости, соответствие параметров, метки и другие. В процессе контекстного анализа строится таблица символов, которую можно рассматривать как таблицу имен, пополненную информацией об описаниях (свойствах) объектов. Основным формализмом, использующимся при контекстном анализе, являются *атрибутные грамматики.* Результатом работы фазы контекстного анализа является атрибутированное дерево программы. Информация об объектах может быть как рассредоточена в самом дереве, так и сосредоточена в отдельных таблицах символов. В процессе контекстного анализа также могут быть обнаружены ошибки, связанные с неправильным использованием объектов. Затем программа может быть переведена во ***внутреннее представление***. Это делается для целей оптимизации и/или удобства генерации кода. Еще одной целью преобразования программы во внутреннее представление является желание иметь *переносимый компилятор*. Тогда только последняя фаза (генерация кода) является машинно-зависимой. В качестве внутреннего представления может использоваться префиксная или постфиксная запись, ориентированный граф, тройки, четверки и другие.
4. Фаз оптимизации может быть несколько. ***Оптимизации*** обычно делят на машинно-зависимые и машинно-независимые, локальные и глобальные. Часть машинно-зависимой оптимизации выполняется на фазе генерации кода. *Глобальная оптимизация* пытается принять во внимание структуру всей программы, локальная - только небольших ее фрагментов. Глобальная оптимизация основывается на глобальном потоковом анализе, который выполняется на графе программы и представляет по существу преобразование этого графа. При этом могут учитываться такие свойства программы, как межпроцедурный анализ, межмодульный анализ, анализ областей жизни переменных и т.д.
5. Наконец, ***генерация кода*** - последняя фаза трансляции. Результатом ее является либо ассемблерный модуль, либо объектный (или загрузочный) модуль. В процессе генерации кода могут выполняться некоторые локальные оптимизации, такие как распределение регистров, выбор длинных или коротких переходов, учет стоимости команд при выборе конкретной последовательности команд. Для генерации кода разработаны различные методы, такие как таблицы решений, сопоставление образцов, включающее динамическое программирование, различные синтаксические методы.

3. Генерация кода.

Задача ***генератора кода*** - построение эквивалентной машинной программы по программе на входном языке. Обычно в качестве входного для генератора кода служит некоторое промежуточное представление программы. В свою очередь, генерация кода состоит из ряда специфических, относительно независимых подзадач: распределение памяти (в частности, распределение регистров), выбор команд, генерация объектного (или загрузочного) модуля.

В какой-то мере схема генератора кода зависит от формы промежуточного представления. Ясно, что генерация кода из дерева отличается от генерации кода из троек, а генерация кода из префиксной записи отличается от генерации кода из ориентированного графа.

1. Модель машины

При изложении алгоритмов генерации кода мы будем следовать некоторой модели машины, в основу которой положена система команд микропроцессора. В системе команд используются следующие способы адресации:

D - значение находится в регистре данных;
A - значение находится в адресном регистре;

Также на этом этапе выбираются команды над этими регистрами.

2. Динамическая организация памяти

Рассмотрим схему реализации магазина периода выполнения для простейшего случая, когда все переменные в магазине (фактические параметры и локальные переменные) имеют известные при трансляции смещения. Магазин служит для хранения локальных переменных (и параметров) и обращения к ним в языках, допускающих рекурсивные определения процедур. Еще одной задачей, которую необходимо решать при трансляции языков с блочной структурой - обеспечение реализации механизмов статической вложенности.

Всего есть две возможные схемы динамической организации памяти: ***схему со статической цепочкой*** и ***с дисплеем в памяти***.

В первом случае все статические контексты связаны в список, который называется статической цепочкой; в каждой записи для процедуры в магазине хранится указатель на запись статически охватывающей процедуры (помимо, конечно, указателя динамической цепочки - указателя на "базу" динамически предыдущей процедуры). Использование той или иной схемы определяется, помимо прочих условий, прежде всего числом адресных регистров.

Рассмотрим теперь организацию магазина с дисплеем. Дисплей - это массив (DISPLAY) фиксированного размера, определяющего ограничение на число статических уровней вложенности процедур. i-й элемент этого массива представляет собой указатель на область активации процедуры i-го статического уровня. Доступ к переменным самой внутренней процедуры осуществляется через регистр BP. При вызове процедуры меньшего статического уровня изменений в дисплее не производится. При вызове локальной процедуры в дисплее отводится элемент для текущей (вызывающей) процедуры. При вызове процедуры того же уровня (i) i-й элемент замещается на указатель области активации текущей процедуры и при выходе восстанавливается. Тем самым DISPLAY[i] всегда указывает на область активации последней вызванной процедуры i-го статического уровня.

3. Назначение адресов

Назначение адресов переменным, параметрам и полям записей происходит при обработке соответствующих объявлений. В однопроходном трансляторе это может производиться вместе с построением основной таблицы символов и соответствующие адреса (или смещения) могут храниться в этой же таблице. В процессе работы генератора кодов поддерживается таблица Table, в которой по этому номеру (входу) содержится следующая информация:

- для типа - его размер;
- для переменной - адрес;
- для поля записи - смещение внутри записи;
- для процедуры - размер локальных;
- для диапазона индексов массива - значение левой границы.

4. Трансляция переменных.

Переменные отражают все многообразие механизмов доступа в языке. Переменная имеет синтезированный атрибут ADDRESS - это запись, описывающая адрес в команде. Этот атрибут сопоставляется всем нетерминалам, представляющим значения. В любом языке есть множество способов трансляции переменных.

5. Трансляция целых выражений

Трансляция выражений различных типов управляется синтаксически благодаря наличию указателя типа перед каждой операцией.

6. Распределение регистров при вычислении арифметических выражений

Одной из важнейших задач при генерации кода является распределение регистров. Рассмотрим хорошо известную технику распределения регистров при трансляции арифметических выражений, называемую ***алгоритмом Сети-Ульмана***.

Пусть система команд машины имеет неограниченное число универсальных регистров, в которых выполняются арифметические команды.

7. Трансляция логических выражений

Логические выражения, включающие логическое умножение, логическое сложение и отрицание, можно вычислять как непосредственно, используя таблицы истинности, так и с помощью условных выражений, пользуясь очевидными правилами:

A & B эквивалентно `if A then B else False`,
A v B эквивалентно `if A then True else B`.

8. Выделение общих подвыражений

Выделение общих подвыражений проводится на линейном участке и основывается на двух положениях.

1. Поскольку на линейном участке переменной может быть несколько присваиваний, то при выделении общих подвыражений необходимо различать вхождения переменных до и после присваивания. Для этого каждая переменная снабжается номером. Вначале номера всех переменных устанавливаются равными `0`. При каждом присваивании переменной ее номер увеличивается на `1`.
2. Выделение общих подвыражений осуществляется при обходе дерева выражения снизу вверх слева направо.

9. Генерация оптимального кода методами синтаксического анализа

Техника генерации кода, рассмотренная выше, основывалась на однозначном соответствии структуры промежуточного представления и описывающей это представление грамматики. Для генерации более качественного кода может быть применен подход, изложенный в настоящей главе.

Этот подход основан на понятии "сопоставления образцов": командам машины сопоставляются некоторые "образцы", вхождения которых ищутся в промежуточном представлении программы, и делается попытка "покрыть" промежуточную программу такими образцами. Если это удается, то по образцам восстанавливается программа уже в кодах.

Синтаксический анализ для Т-грамматик. Обычно код генерируется из некоторого промежуточного языка с довольно жесткой структурой. В частности, для каждой операции известна ее размерность (число операндов). Назовем грамматики, удовлетворяющие этим ограничениям, Т-грамматиками.

Образцы, соответствующие машинным командам, задаются правилами грамматики (вообще говоря, неоднозначной). Генератор кода анализирует входное префиксное выражение и строит одновременно все возможные деревья разбора. После окончания разбора выбирается дерево с наименьшей оценкой. Затем по этому единственному оптимальному дереву генерируется код.

Выбор дерева вывода наименьшей стоимости. Т-грамматики, описывающие системы команд, обычно являются неоднозначными. Чтобы сгенерировать код для некоторой входной цепочки, необходимо выбрать одно из возможных деревьев вывода. Это дерево должно представлять желаемое качество кода, например размер кода и/или время выполнения.

4. Промежуточные представления программы.

В процессе трансляции программы часто используют промежуточное представление (ПП) программы, предназначенное прежде всего для удобства генерации кода и/или проведения различных оптимизаций программы. Сама форма ПП зависит от целей его использования. Наиболее часто используемыми формами ПП является ***ориентированный граф (или, в частности, абстрактное синтаксическое дерево), тройки, четверки, префиксная или постфиксная запись, атрибутированное абстрактное дерево***.

1. Представление в виде ориентированного графа

Простейшей формой промежуточного представления является синтаксическое дерево программы. Более полную информацию о входной программе дает ориентированный ациклический граф (ОАГ), в котором в одну вершину объединены вершины синтаксического дерева, представляющие общие подвыражения.

2. Трехадресный код

Трехадресный код - это последовательность операторов вида `x:= y op z`, где `x`, `y` и `z` - имена, константы или сгенерированные компилятором временные объекты. Здесь `op` - двуместная операция, например операция плавающей или фиксированной арифметики, логическая или побитовая. В правую часть может входить только один знак операции.

Составные выражения должны быть разбиты на подвыражения, при этом могут появиться временные имена (переменные). Смысл термина "трехадресный код" в том, что каждый оператор обычно имеет три адреса: два для операндов и один для результата. Трехадресный код - это линеаризованное представление синтаксического дерева или ОАГ, в котором явные имена соответствуют внутренним вершинам дерева или графа.

В виде трехадресного кода представляются не только двуместные операции, входящие в выражения. В таком же виде представляются операторы управления программы и одноместные операции. В этом случае некоторые из компонент трехадресного кода могут не использоваться.

Трехадресный код - это абстрактная форма промежуточного кода. В реализации трехадресный код может быть представлен записями с полями для операции и операндов. Рассмотрим три реализации трехадресного кода: четверки, тройки и косвенные тройки.

Четверка - это запись с четырьмя полями, которые будем называть `op`, `arg1`, `arg2` и `result`. Поле `op` содержит код операции.

Чтобы избежать внесения новых имен в таблицу символов, на временное значение можно ссылаться, используя позицию вычисляющего его оператора. В этом случае трехадресные операторы могут быть представлены записями только с тремя полями: `op`, `arg1` и `arg2`. Поля `arg1` и `arg2` - это либо указатели на таблицу символов (для имен, определенных программистом, или констант), либо указатели на тройки (для временных значений). Такой способ представления трехадресного кода называют ***тройками***. Тройки соответствуют представлению синтаксического дерева или ОАГ с помощью массива вершин.

Числа в скобках - это указатели на тройки, а имена – это указатели на таблицу символов. На практике информация, необходимая для интерпретации различного типа входов в поля `arg1` и `arg2`, кодируется в поле `op` или дополнительных полях.

Трехадресный код может быть представлен не списком троек, а списком указателей на них. Такая реализация обычно называется ***косвенными тройками***.

При генерации объектного кода каждой переменной, как временной, так и определенной в исходной программе, назначается память периода исполнения, адрес которой обычно хранится в таблице генератора кода. При использовании четверок этот адрес легко получить через эту таблицу.

Более существенно преимущество четверок проявляется в оптимизирующих компиляторах, когда может возникнуть необходимость перемещать операторы. Если перемещается оператор, вычисляющий `x`, не требуется изменений в операторе, использующем `x`. В записи же тройками перемещение оператора, определяющего временное значение, требует изменения всех ссылок на этот оператор в массивах `arg1` и `arg2`. Из-за этого тройки трудно использовать в оптимизирующих компиляторах.

В случае применения косвенных троек оператор может быть перемещен переупорядочиванием списка операторов. При этом не надо менять указатели на `op`, `arg1` и `arg2`. Этим косвенные тройки похожи на четверки. Кроме того, эти два способа требуют примерно одинаковой памяти. Как и в случае простых троек, при использовании косвенных троек выделение памяти для временных значений может быть отложено на этап генерации кода. По сравнению с четверками при использование косвенных троек можно сэкономить память, если одно и то же временное значение используется более одного раза.

3. Линеаризованные представления

В качестве промежуточных представлений весьма распространены линеаризованные представления. Линеаризованное представление позволяет относительно легко хранить промежуточное представление на внешней памяти и обрабатывать его в порядке чтения. Самая распространенная форма линеаризованного представления - это запись дерева либо в порядке его обхода снизу-вверх (постфиксная запись, или обратной польской), либо в порядке обхода его сверху - вниз (префиксная запись, или прямой польской).

Таким образом, ***постфиксная запись*** (обратная польская) – это список вершин дерева, в котором каждая вершина следует непосредственно за своими потомками.

В постфиксной записи вершины синтаксического дерева явно не присутствуют. Они могут быть восстановлены из порядка, в котором следуют вершины и из числа операндов соответствующих операций. Восстановление вершин аналогично вычислению выражения в постфиксной записи с использованием стека. В префиксной записи сначала указывается операция, а затем ее операнды.

4. Организация информации в генераторе кода

Чисто синтаксическое дерево несет только информацию о структуре программы. На самом деле в процессе генерации кода требуется также информация о переменных (например, их адреса), процедурах (также адреса, уровни), метках и т.д. Для представления этой информации возможны различные решения. Наиболее распространены два.

1. Всю эту информацию можно хранить в таблицах генератора кода.

2. Информация хранится в вершинах дерева с соответствующими указателями.

5. Уровень промежуточного представления

Как видно из приведенных примеров, промежуточное представление программы может в различной степени быть близким либо к исходной программе, либо к машине. Например, промежуточное представление может содержать адреса переменных, и тогда оно уже не может быть перенесено на другую машину. С другой стороны, промежуточное представление может содержать раздел описаний программы, и тогда информацию об адресах можно извлечь из обработки описаний. В то же время ясно, что первое более эффективно, чем второе.

Операторы управления в промежуточном представлении могут быть представлены в исходном виде (в виде операторов языка `if`, `for`, `while` и т.д.), а могут содержаться в виде переходов. В первом случае некоторая информация может быть извлечена из самой структуры (например, для оператора `for` - информация о переменной цикла, которую, может быть, разумно хранить на регистре, для оператора `case` - информация о таблице меток и т.д.). Во втором случае представление проще и унифицировано. Некоторые формы промежуточного представления лучше годятся для различного рода оптимизаций (например, косвенные тройки - для перемещения кода), некоторые - хуже (например, префиксная запись для этого плохо подходит).

5. Лексический анализ.

Основная задача лексического анализа - разбить входной текст, состоящий из последовательности одиночных символов, на последовательность слов, или лексем, т.е. выделить эти слова из непрерывной последовательности символов. Все символы входной последовательности с этой точки зрения разделяются на символы, принадлежащие каким-либо лексемам, и символы, разделяющие лексемы (разделители). В некоторых случаях между лексемами может и не быть разделителей. С другой стороны, в некоторых языках лексемы могут содержать незначащие символы. В Си разделительное значение символов-разделителей может блокироваться ('\' в конце строки внутри "...").

Лексический анализатор называется сканером. ***Для работы сканер использует следующую информацию***:

1. Исходный текст программы.
2. Таблица терминальных символов.
3. Таблица литералов.
4. Таблица идентификаторов.

Обычно все лексемы делятся на классы. Примерами таких классов являются числа (целые, восьмеричные, шестнадцатеричные, действительные и т.д.), идентификаторы, строки. Отдельно выделяются ключевые слова и символы пунктуации (иногда их называют символы-ограничители). Как правило, ключевые слова - это некоторое конечное подмножество идентификаторов. В некоторых языках (например, ПЛ/1) смысл лексемы может зависеть от ее контекста и невозможно провести лексический анализ в отрыве от синтаксического.

С точки зрения дальнейших фаз анализа лексический анализатор выдает информацию двух сортов: для синтаксического анализатора, работающего вслед за лексическим, существенна информация о последовательности классов лексем, ограничителей и ключевых слов, а для контекстного анализа, работающего вслед за синтаксическим, важна информация о конкретных значениях отдельных лексем (идентификаторов, чисел и т.д.). Поэтому общая схема работы лексического анализатора такова. Сначала выделяем отдельную лексему (возможно, используя символы разделители). Если выделенная лексема - ограничитель, то он (точнее, некоторый его признак) выдается как результат лексического анализа. Ключевые слова распознаются либо явным выделением непосредственно из текста, либо сначала выделяется идентификатор, а затем делается проверка на принадлежность его множеству ключевых слов. Если да, то выдается признак соответствующего ключевого слова, если нет - выдается признак идентификатора, а сам идентификатор сохраняется отдельно. Если выделенная лексема принадлежит какому-либо из других классов лексем (число, строка и т.д.), то выдается признак класса лексемы, а значение лексемы сохраняется.

Лексический анализатор может работать или как самостоятельная фаза трансляции, или как подпрограмма, работающая по принципу "дай лексему". В первом случае выходом лексического анализатора является файл лексем, во втором лексема выдается при каждом обращении к лексическому анализатору (при этом, как правило, тип лексемы возвращается как значение функции "лексический анализатор", а значение передается через глобальную переменную). С точки зрения формирования значений лексем, принадлежащих классам лексем, лексический анализатор может либо просто выдавать значение каждой лексемы и в этом случае построение таблиц переносится на более поздние фазы, либо он может самостоятельно строить таблицы объектов (идентификаторов, строк, чисел и т.д.). В этом случае в качестве значения лексемы выдается указатель на вход в соответствующую таблицу.

Работа лексического анализатора описывается ***формализмом конечных автоматов***. Однако непосредственное описание конечного автомата неудобно практически. Поэтому для описания лексических анализаторов, как правило, используют либо ***формализм регулярных выражений***, либо ***формализм контекстно-свободных грамматик***, а именно подкласса автоматных, или регулярных, грамматик. Все три формализма (конечных автоматов, регулярных выражений и автоматных грамматик) имеют одинаковую выразительную мощность. По описанию лексического анализатора в виде регулярного выражения или автоматной грамматики строится конечный автомат, распознающий соответствующий язык.

Конечным автоматом является система, которая в каждый момент времени может находится в одном из конечного множества заданных состояний. Каждый шаг (переключение) автомата состоит в том, что при нахождении в определенном состоянии при поступлении на вход одного из множества входных сигналов (воздействий) он переходит в однозначно определенное состояние и вырабатывает определенное выходное воздействие. Легче всего представить себе поведение КА в виде его диаграммы состояний-переходов.

Таким образом, если поведение какого-либо объекта можно описать набором предложений вида: находясь в состоянии A, при получении сигнала S объект переходит в состояние B и при этом выполняет действие D - то такая система будет представлять из себя конечный автомат. На диаграмме состояний-переходов каждому состоянию соответствует кружок, каждому переходу - дуга со стрелкой. Каждое состояние имеет свой “смысл”, заключенный в подписи. Каждый переход осуществляется только при выполнении определенного условия, которое

обозначено подписью под дугой. Кроме того, при выполнении перехода производится действие, при помощи которого автомат обнаруживает свое поведение извне.

У конечного автомата есть еще несколько условий работоспособности:

- автомат имеет некоторое начальное состояние, из которого он начинает работу;
- автомат имеет конечное число состояний;

В каждом состоянии не может быть одновременно справедливыми несколько условий перехода, то есть автомат должен быть ***детерминированным***. Автомат не может перейти одновременно в несколько состояний и не может иметь таких условий перехода.

Специфика программирования КА применительно к лексическому анализу заключается только в действиях, производимых автоматом. Целью ЛА является распознавание и выделение лексемы (слова) - последовательности символов. Поэтому в КА вводится множество заключительных состояний, в которых завершается распознавание. Действия, связанные с распознаванием, практически одинаковы для всех типов лексем (слов), поэтому в КА вводится множество заключительных состояний, по одному на каждый тип лексемы. Они имеют отрицательное значение, и по их обнаружении программа выполняет следующие действия:

- формирует цепочку из накопленных символов - значение лексемы;
- устанавливает тип распознанной лексемы в соответствии с номером конечного состояния КА;
- возвращает КА в исходное (нулевое состояние);
- поскольку обнаружение лексемы иногда происходит в тот момент, когда КА обрабатывает символ, уже к ней не относящийся, то программа должна “вернуть” во входную последовательность заданное число символов для повторного просмотра. Например, любая десятичная константа распознается, когда уже прочитан следующий за ней символ - не цифра. Его то и необходимо вернуть. Для этого в программе определяется массив, в котором для каждого заключительного состояния указано количество возвращаемых символов. Естественно, что его содержимое определяется конкретной лексикой.

6. Классификация языков по Хомскому.

Хомский определил ***четыре основных класса языков в терминах грамматик***, являющихся упорядоченной четверкой (V, T, P, Z), где

1. V - алфавит;
2. $T \subseteq V$ - алфавит терминальных символов;
3. P - конечный набор правил подстановки;
4. Z - начальный символ, принадлежащий множеству V - T.

Язык, порождаемый 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 u может быть тоже последовательностью символов, а правая часть может быть пустой. Если ввести ограничение на правила подстановки, то получится более интересный класс ***языков типа 1***, называемых также контекстно-чувствительными или контекстно-зависимыми языками. В этом случае правила подстановки имеют следующий вид:

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

Термин "контекстно-чувствительная" отражает тот факт, что можно заменить U на u лишь в контексте $x...y$. Дальнейшее ограничение дает класс грамматик, полностью подобный классу, который мы используем; грамматика называется ***контекстно-свободной***, если все ее правила, имеют вид

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

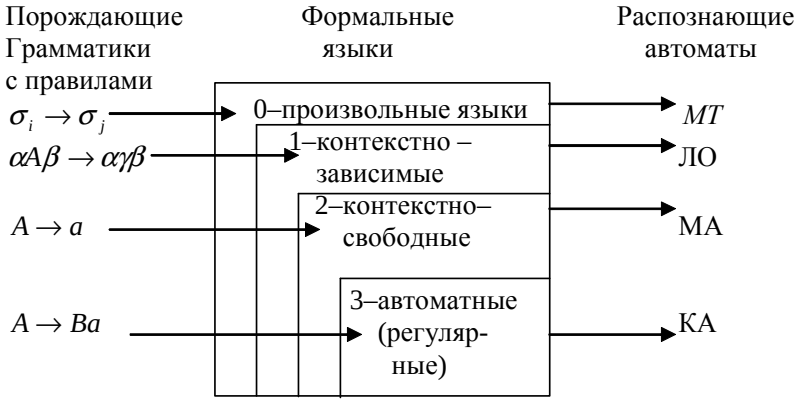
Этот класс назван контекстно-свободным потому, что символ U можно заменить цепочкой u, не обращая внимания на контекст, в котором он встретился. В КС-грамматике может появиться правило вида $U := \Lambda$, где Λ - пустая цепочка. Однако чтобы не усложнять терминологию и доказательства, мы не допускаем таких правил в наших грамматиках. По заданной КС-грамматике G можно сконструировать Λ-свободную (или неукорачивающую) грамматику $G1$ (наш тип), такую, что $L(G1) = L(G) - \{\Lambda\}$. Более того, если G однозначна, то $G1$ также однозначна, поэтому фактически мы не вносим ограничений.

Если мы ограничим правила еще раз, приведя их к виду

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

то получим ***грамматику типа 3***, или регулярную грамматику. Регулярные грамматики играют основную роль как в теории языков, так и в теории автоматов. Множество цепочек, порождаемых регулярной грамматикой, "допускается" машиной, называемой автоматом с конечным числом состояний, и наоборот. Таким образом, мы имеем характеристику этого класса грамматик в терминах автомата. Регулярные языки (те, что порождаются регулярными грамматиками), кроме того, называются регулярными множествами.

Вводя все большие ограничения, мы определили четыре класса грамматик. Таким образом, есть языки с фразовой структурой, которые не являются контекстно-чувствительными, контекстно-чувствительные языки, которые не являются контекстно-свободными, и контекстно-свободные, которые не являются регулярными.



Языки класс «0» (произвольные) не употребляются для практических целей.

Языки класса «1» (контекстно-зависимые); ЛОА, КЗ-грамматики используются для построения программ – переводчиков с естественных языков.

Языки класса «2» (контекстно-свободные); МА, КС-грамматики используются для построения синтаксических анализаторов языков программирования.

Языки класса «3» (автоматные языки); КА, А – грамматики используются для построения текстовых редакторов и отладчиков программ. Например, МЕ, Perl и т.д.

7. Языки грамматики и 4 класса автоматов.

1. Язык для преобразования алгебраических выражений и языки типа 0

Правила этой грамматики можно с тем же успехом назвать правилами вывода для алгебраических преобразований или операторами задачи в пространстве состояний. Вообще, имея задачу в пространстве состояний, можно построить правила, определяющие состояния и операции на них. В обще случае потребуются правила преобразований, как в вышеприведенной таблице. Отсюда заключаем, что:

- любую задачу, допускающую представление в пространстве состояний, можно связать с языком типа 0 и, значит, решить с помощью программы, которая, подобно машине Тьюринга, имеет неограниченный доступ к принципиально бесконечному участку динамической памяти.
- любой язык, порождаемый грамматикой типа 0, рекурсивно перечислим.

2. Линейно ограниченные автоматы и языки типа 1

Линейно ограниченным автоматом (ЛО-автоматом) называют машину Тьюринга, которой запрещено передвигать читающую головку за пределы участка входной ленты, занятого входной цепочкой. Это можно достичь, добавив к входному языку два специальных символа (обычно & и $). Их помещают в начале и в конце входной цепочки, которая после этого превращается в

& s $.

Правила перехода ЛО - автомата подобны правилам для машины Тьюринга за исключением того, что если прочитан символ & или $, то читающая головка передвигается на одну ячейку вправо или влево.

Линейно-ограниченный автомат представляет собой модель программы с неограниченным доступом к конечной области “динамического” участка памяти, причем размер этой области можно определить, исследуя входную цепочку в ходе работы программы перед анализом приемлемости входной цепочки. Фактически можно просто считать, что какая-то часть ленты требуется для ввода как стираемая зона.

Модель ЛО-автомата имеет неочевидное приложение в программировании. Напомним, что автомат называется ***детерминированным***, если для каждого сочетания сигнала на входе и внутреннего состояния существует не более одного правила перехода, определяющего его следующее действие. Автомат называется ***недетерминированным***, если данная комбинация вход — внутреннее состояние служит левой частью более чем одного правила перехода. Когда речь шла о машине Тьюринга, разница между детерминированным и недетерминированным автоматами не рассматривалась, поскольку можно показать, что для каждой недетерминированной машины Тьюринга существует эквивалентная ей детерминированная. Для ЛО-автоматов не известно, так это или нет, поэтому, пока ответ не получен, благоразумный программист должен считать, что если ему нужно написать программу, моделью которой является ЛО-автомат, то он должен считать этот автомат недетерминированным. Для недетерминированного автомата может возникнуть возможность выбора конкретного правила перехода при предъявлении определенной конфигурации памяти. Если выбор сделан неверно, то автомат может впоследствии прийти в состояние останова, не приняв цепочки. В этом случае надо вернуться к моменту выбора и попытаться принять цепочку, избирая другой путь вычислений. Это можно сделать только в том случае, когда сохраняются весьма точные записи предыдущих действий.

Теоретический результат в области ЛО-автоматов таков:

Язык L *воспринимается ЛО-автоматом тогда и только тогда, когда это язык типа* 1.

Этот результат не находит широких приложений, поскольку, как мы показали, ЛО-автоматы моделируют сложные программы. В то же время доказательство того, что областью определения функции f является контекстно-зависимый язык, может оказаться полезным, так как оно выявляет возможности управления памятью и хранения записей, необходимые программе для вычисления f. Чаще используется другой результат.

Все контекстно-зависимые языки рекурсивны.

3. Автомат с магазинной памятью и языки типа 2

Автомат с магазинной памятью (МП-автомат) — это автомат с ограниченным доступом к принципиально бесконечной ленте памяти. МП-автомат может передвигать свою читающую головку только вправо, т.е. он может воспринимать входные символы только в определенном порядке. Далее МП-автомат имеет ленту памяти (магазин), которая может двигаться “вверх” и “вниз”. Если в результате движения ленты памяти символ помещается “над” ее читающей головкой, то этот символ теряется. На языке исследования операций лента памяти работает как очередь, организованная по принципу “последним пришел — первым вышел”. Эта лента предоставляет ограниченный доступ к бесконечной памяти, так как, если j-й символ от

последнего, записанного на ленте, должен быть прочитан, $j - 1$ промежуточных символов должны сначала быть стерты.

МП-автомат начинает работу в состоянии S0, имея цепочку s на входной ленте и пустой магазин. Последующие конфигурации определяются именем прочитанного символа, внутренним состоянием МП-автомата и именем символа над читающей головкой магазина.

Язык воспринимается МП-автоматом тогда и только тогда, когда это язык типа 2.

Это чрезвычайно важный результат для теории вычислений, поскольку наиболее традиционные языки программирования (за редким исключением) являются языками типа 2. Фактически большинство трансляторов сознательно конструировались как МП-автоматы. Техника программирования для управления динамической памятью в МП-автоматах очень хорошо развита.

4. Конечные автоматы и регулярные языки (типа 3)

Конечный автомат — эта машина без динамической памяти. Например, программа на фортране, в которой не используется ленты со стиранием, определяет конечный автомат, так как у нее нет динамической памяти. Подобным образом программы на Алголе и ПЛ/1, которые не определяют динамических структур данных, можно представить как конечные автоматы, так что этот класс программ имеет немалое практическое значение.

Язык L воспринимается конечным автоматом тогда и только тогда, когда L — регулярный язык (типа 3).

Т.о. определены четыре типа языков, от относительно неструктурированного языка типа 0 до сильно структурированного языка типа 3. Каждый из языков более высокого порядка представляет собой собственное подмножество всех языков более низкого порядка. Этой последовательности языков соответствует последовательность все более ограниченных автоматов: машина Тьюринга, ЛО-автоматы, МП-автоматы, конечные автоматы. Эти четыре класса автоматов различаются по степени доступа к принципиально неограниченной ленте памяти. Каждый из них можно рассматривать как основной план для класса программ с аналогичным ограничением на доступ к принципиально бесконечной динамической памяти.

8. Семантический анализ.

Назначение фазы семантического анализа.

Ясно, что текст программы, точно соответствующий грамматике и правильно разобранный на фазе синтаксического анализа, может содержать множество ошибок. Естественно эти ошибки нельзя описать в рамках грамматики, так как они являются контекстными. Например, с точки зрения грамматики фраза `cos(sin((int*)("Argument")));` может быть и правильной, хотя она абсолютно бессмысленна.

Семантические ошибки в основном относятся к:

- неправильным типам аргументов функций и операторов;
- несуществующими переменными, метками, функциями;
- несуществующими полями объектов, функций;
- неправильно сформированными строками;
- неправильным формированием списка аргументов для функций, принимающих переменное количество аргументов;

Решение проблемы семантического анализа является исключительно инженерной технологией, потому как невозможно формализовать практическое большинство операций, относящееся к этой фазе. Укажем лишь некоторые проблемы, с которыми сталкивается программист при реализации этого этапа.

Области видимости и списки идентификаторов.

Для большинства языков, например Си, допускается объявление переменных внутри вложенных блоков. При переопределении имени в таком случае доступ производится к имени, определённому в самом вложенном блоке. Применение специального квалификатора позволяет иметь доступ к глобальному имени, скрытому за объявленным в локальном блоке именем. К этим сложностям в стандарте Си `1999` года добавляются пространства имён `(namespace)`. Таким образом, идентификаторы организованы внутри каждого блока в виде таблиц со ссылками на блоки верхнего уровня. Каждое пространство имён также имеет свою таблицу, их которых обычно активна лишь одна. Доступ к идентификаторам внутри таблиц обычно осуществляется с помощью метода хеширования. В случае обращений к объектам, структурам и массивам дополнительно проверяется семантическая информация, связанная с правильностью обращения к полям класса, методам класса, правильности использования индексации массивов.

Отметим, что генератору кода и глобальному оптимизатору с этапа семантического анализа должна подаваться только корректная информация, так как дополнительные проверки на правильность внутренних структур данных увеличивают в несколько раз время обработки программы.

Преобразование типов.

Довольно тривиальные ошибки могут не отлавливаться программистом, например потеря точности при арифметических сдвигах, или же преобразование типов, при которых могут теряться значащие разряды. В ряде случаев программист осведомлен о последствиях, но возможно он сделал это случайно. Например, типичные случаи таких ошибок:
`if (i=2)` вместо `if (i==2)`

Оптимизации программы.

В ряде случаев на этапе семантического анализа можно сделать простейшие вычисления, например `4*1024` или же убрать из дальнейшего рассмотрения конструкции типа `if (0)`. Зачем это делать, все равно на следующих этапах эти случаи будут оптимизированы? Компиляция даже маленькой программы компилятором, агрессивно выполняющем оптимизации, требует памяти порядка `300`Кбайт, большие программы могут требовать порядка `10-20` Мбайт. Логика выполнения компилятора трудно предсказуема (предсказание переходов не работает примерно в `10%` случаев), кеш-память, тоже в силу принципиальных особенностей обработки данных в компиляторе, не влияет на увеличение производительности так сильно, как в других программах. Так как сложность выполнения оптимизаций обычно пропорциональна квадрату или кубу от величины внутреннего представления программы, лучше явные случаи оптимизации делать на более ранних этапах, чем в общем потоке.

Семантическая оптимизация фактически приписывает формируемым элементам внутреннего представления атрибуты типа, области видимости и т.д., и анализирует совместимость этих атрибутов в рамках арифметических выражений или базовых блоков. Обогащённое такими атрибутами, разобранное представление программы передаётся далее на этапы оптимизации и генерации кода.

9. БНФ – форма и БНФ-грамматика.

На практике применяется форма записи грамматики, традиционно называемая ***нормальными формами Бэкуса-Наура (Бэкуса-Наура форма)*** (БНФ). Терминалы в БНФ записываются как обычные символы алфавита, а нетерминалы - как имена в угловых скобках <>. Например, грамматику целых чисел без знака можно записать в виде:

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

Рассмотрим язык простейших арифметических формул:

<формула>:(<формула>)|<число>|<формула><знак><формула>
<знак>:+|*

Большинство грамматик допускают несколько различных выводов для одной и той же цепочки из языка.

Если в процессе вывода цепочки правила грамматики применяются только к самому левому нетерминалу, говорят, что получен левый вывод цепочки. Аналогично определяется правый вывод.

Ниже представлена ***простая грамматика БНФ***:

```
S ::= {-;FN}|FN
FN::= DL|{DL;.;DL}
DL::= D|D DL
D ::= 0|1|2|3|4|5|6|7|8|9
```

Все символы здесь – сокращения: **S** – начальный символ (start symbol), **FN –** число с дробной частью (fractional number), **DL** – список цифр (digit list), **D** – цифра (digit).

10. Понятие грамматика, словарь, цепочка, язык.

Алфавит - это непустое конечное множество элементов. Назовём элементы алфавита ***символами***. Всякая конечная последовательность символов алфавита A называется ***цепочкой***. Вот несколько цепочек "в алфавите" A={a,b,c}: a,b,c,ab и ааса. Мы также допускаем существование пустой цепочки, т. е. цепочки, не содержащей ни одного символа. Важен порядок символов в цепочке; так цепочка ab не то же самое, что ba, и abca отличается от aabc. ***Длина цепочки*** x (записывается как |x|) равна числу символов в цепочке. Таким образом:

| |=0, |a|=1, |abb|=3

Заглавные буквы M,N,S,T,U, используются как переменные или имена символов алфавита, в то время как строчные буквы u,v,w используются для обозначения цепочек символов. Таким образом, можно написать x=STV, и это означает, что x является цепочкой, состоящей из символов S, T и V именно в таком порядке.

Если x и y - цепочки, то их катенацией xy является цепочка, полученная путем дописывания символов цепочки y вслед за символами цепочки x. Например, если x=XY, y=YZ, то xy=XYYZ и yx=YZXY. Поскольку Λ, не содержащая символов, то в соответствии с правилом катенации для любой цепочки x мы можем записать:

Λx = xΛ = x

Если z = xy - цепочка, то x - ***голова***, а y - ***хвост цепочки*** z. x - правильная голова, если y - не пустая цепочка, z - правильный хвост, если x - не пустая цепочка. Множества цепочек в алфавите обычно обозначаются заглавными буквами A, B . . . Произведение AB двух множеств цепочек A и B определяется как

AB = {xy|x∈A,y∈B}

и читается как "множество цепочек xy такое, что x из A, а y из B.

Мы можем определить степени цепочек. Если x - цепочка, то x^0 - пустая цепочка , $x^1 = x$, $x^2 = xx$, и в общем случае

x^n = x....x (n раз)

Используя это, определим две последние операции в этом разделе - итерацию **A*** множества A и усеченную итерацию **A+** множества A:

$A+ = A^1 \cup A^2 \cup ... \cup A^n \cup ...$
$A* = A^0 \cup A+$

Таким образом, если A={a,b}, то A* включает цепочки Λ,a,b,aa,ab,ba,bb,aaa,aab... Заметим, что A+=AA*=(A*)A.

Примеры:

Пусть z=abb. Тогда |z|=3. Головы z есть Λ,a,ab,abb. Правильные головы z есть Λ,a,ab. Хвосты z - это Λ,b,bb,abb. Правильные хвосты z - это Λ,b,bb.

Пусть x=a, z=abb. Тогда zx=abba, xz=aabb,

$z^0 = Λ$, $z^1 = aab$, $z^2 = aabaab$, $z^3 = aababbabb$,

$|z^0| = 0$; $|z^1| = 3$; $|z^2| = 6$; $|z^3| = 9$;

Пусть S={a,b,c}. Тогда

S+ ={a,b,c,aa,ab,ac,ba,bb,bc,ca,cb,cc,aaa,...}.

S* ={Λ,a,b,c,aa,ab,ac, .}.

Грамматикой G[Z] называется конечное, непустое множество правил; Z - это символ, который должен встретиться в левой части по крайней мере одного правила. Он называется начальным символом. Все символы, которые встречаются в левых и правых частях правил, образуют ***словарь V***.

Если из контекста ясно, какой символ является начальным символом Z, мы будем писать G вместо G[Z].

В заданной грамматике G символы, которые встречаются в левой части правил, называются ***нетерминалами*** или синтаксическими единицами, языка. Они образуют множество нетерминальных символов VN.

Символы, которые не входят в множество VN, называются ***терминальными символами*** (или терминалами). Они образуют множество VT.

Таким образом, V=VN∪VT. Как правило, нетерминалы мы будем заключать в угловые скобки,

чтобы отличить их от терминалов. В грамматике G1 символы 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 - терминальные, а <число>, <чс> и <цифра> - нетерминальные. Мы будем пользоваться надстрочными литерами в том случае, когда нам надо отличить различные вхождения одного и того же нетерминала. Так, можно написать <чс1>::=<чс2><цифра> вместо <чс>::=<чс><цифра>.

Множество правил U::=x, U::=y,..., U::=z с одинаковыми левыми частями будем записывать сокращенно как

U::=x|y|...|z

Например, грамматику G1 можно записать 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** (сокращенно БНФ) или формой Бэкуса - Наура.

Определение языка, соответствующего грамматике

Формальные языки рассматриваются как множества последовательностей "символов" из некоторого алфавита.

Языком над алфавитом A называется любое множество L цепочек в алфавите A, т.е. $L \subseteq A$. В частности, $\varnothing$, $\varnothing^*$ и A^* - языки над любым алфавитом A. Языки будем обозначать большими буквами из средней части латинского алфавита.

Что является ***предложением этого языка***? Для того чтобы ответить на эти вопросы, нам надо определить символы "=>" и "=>+", которыми мы интуитивно пользовались при выводе предложений. Неформально мы пишем v=>w, если можно вывести w из v, заменив нетерминальный символ в v на соответствующую правую часть некоторого правила.

Пусть G - грамматика. Мы говорим, что цепочка v непосредственно порождает цепочку w, и обозначаем это как

v=>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 приводится (редуцируется) к v. Цепочки x и y могут, конечно, быть пустыми. Следовательно, для любого правила U::=u грамматики G имеет место U=>u.

Чтобы задать **грамматику** G=(E,Sym,P), требуется указать:

- множество символов алфавита (или терминальных символов) E. Будем обозначать их строчными символами алфавита и цифрами;
- множество нетерминальных символов Sym (или метасимволов), не пересекающееся с E со специально выделенным начальным символом S. Будем обозначать их прописными буквами;
- множество правил вывода P определяющих правила подстановки для цепочек. Каждое правило состоит из двух цепочек (например, x и y), причем x должна содержать по крайней мере один нетерминал; и означает, что цепочку x в процессе вывода можно заменить на y. Вывод цепочек языка начинается с нетерминала S. Правило грамматики будем записывать в виде x:y. (Также употребляется запись x::= y или x->y)

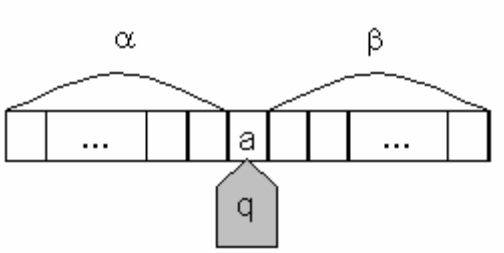
Более строго, определим понятие ***выводимой цепочки***:

- S - выводимая цепочка;
- если xyz - выводимая цепочка и в грамматике имеется правило y:t, то xtz - выводимая цепочка;
- определяемый грамматикой язык состоит из выводимых цепочек, содержащих только терминальные символы.

11. Машина Тьюринга.

Машины Тьюринга представляют собой простой формально определённый рекурсивно полный класс алгоритмов. Каждая машина Тьюринга представляет один алгоритм. Она состоит из

- **ленты** - конечной последовательности **ячеек**, в каждой из которых записано по одному **символу** некоторого конечного алфавита Σ
- **головки** - устройства, которое в каждый момент обозревает некоторую ячейку, может находиться в одном из конечного множества **состояний** Q, читать и писать символ в обозреваемой ячейке, перемещаться вдоль ленты и добавлять при необходимости к её концам новые ячейки
- **программы** - конечного множества **правил** P, управляющих поведением головки.



Машина Тьюринга

Правило имеет вид:

$$(q,a)\rightarrow(q',a',S)$$

где q и q' – состояния, a и a' - символы алфавита, S - **сдвиг** головки, принимающий значение L (сдвиг влево на одну ячейку), R (сдвиг вправо на одну ячейку) или N (отсутствие сдвига). Правило может применяться, когда головка находится в состоянии q и обозревает ячейку с символом a. Если оно применяется, то головка переходит в состояние q', в обозреваемую ячейку записывается символ a', а затем выполняется сдвиг S. Если сдвиг невозможен из-за выхода за границу ленты, то к ней с соответствующей стороны добавляется ячейка с записанным в неё специальным символом $\Lambda\in\Sigma$.

Множество состояний обязательно содержит одно **начальное** состояние q_0 и одно **заключительное** состояние q_f. При этом предполагается, что для каждой пары (q,a), где $q\neq q_f$, в P есть хоть одно правило, а для q_f нет ни одного правила. Таким образом, машина Тьюринга останавливается тогда и только тогда, когда она попадает в заключительное состояние.

Конфигурацией машины Тьюринга называется цепочка вида $\alpha q a\beta$, где $\alpha a\beta$ - содержимое ленты, q - текущее состояние головки, а её позиция указывает обозреваемую ячейку между α и β. Предполагается, что **начальная конфигурация** всегда имеет вид $q_0a\beta$, т.е. головка в этих конфигурациях всегда сдвинута к левому концу ленты.

Формально, машина Тьюринга определяется набором

$$\mathbf{M = (\Sigma, Q, P, q_0, q_f)}$$

где $\Sigma = \{\Lambda, a_1, \ldots, a_n\}$ - конечный ленточный алфавит, $n\geq 1$,
$Q = \{q_0, \ldots, q_f\}$ - конечное множество состояний,
$P\subseteq Q\times\Sigma\times Q\times\Sigma\times S$ где $S = \{L,R,N\}$, - множество правил, записываемых в виде
$(q,a)\rightarrow(q',a',S)$, где (q,a) принимает все значения из $(Q\setminus\{q_f\})\times S$,
q_0, q_f - начальное и заключительное состояния, соответственно
$K = \{\alpha q\beta \mid q\in Q, \alpha\in\Sigma^*, \beta\in\Sigma^+\}$ - множество конфигураций (обозначения Σ^* и Σ^+),
$I = \{q_0\alpha \mid \alpha\in\Sigma^+\}$ - множество начальных конфигураций

12. Формальные грамматики.

Исчисление в виде *формальных систем* предложено американским математиком Хомским (1953 г.) сначала для решения проблем структурной лингвистики, а далее описания формальных языков программирования.

ФГ=<A, V, S, P>

где

$A=\{a, b, c...\}$ – терминальный алфавит

$V=\{A, B, ... S\}$ – вспомогательный алфавит,

S – единственная аксиома, P – правила вида $S \to x$, и $y \to z$, где $x, y, z \in \{\mathcal{A} \cup V\}^*$.

ФГ порождает характерный язык в виде подмножества слов $\mathcal{L} \subset \{A\}^*$. Правила *ФГ* не содержат никаких ограничений на рекурсивность подстановок.

13. Автоматные грамматики.

Конечный алфавит $\mathcal{A}=\{a, b, c,..., x\}$ – терминальный алфавит (малые латинские буквы).

$\mathcal{A}^*=\{a, b, c,..., x\}^*$ – все слова, полученные из букв алфавита, * – операция итерации приписывания (конкатенации).

$\mathcal{L} \subseteq \mathcal{A}^*$ – язык, вообще говоря, состоит из бесконечного числа слов. Возникает проблема описания языка при помощи конечных и конструктивных объектов (например, формул или правил порождения).

Как обычно, рассматриваются две основные задачи.

- **Порождение языка L.** Для порождения слова α служит специальная конструкция – порождающая грамматика *G*, которая представляет собой конечный набор продукций (правил подстановок) для вывода любых слов языка из специального символа *S* (аксиомы) и только их.

Формально – $G=\langle \mathcal{A}, V, P, S \rangle$

A – основной алфавит, *V* – вспомогательный алфавит, $S \in V$ – аксиома

$\sum=(\mathcal{A} \cup V)$ – объединённый алфавит, $\mathcal{A} \cap V=\phi$, *P* – правила (подстановки, продукции): имеет вид $(\sigma_i \to \sigma_j)$, где σ_i, σ_j (в общем случае) $\in (\sum)^*$ (слова в объединённом алфавите)

- **Распознавание слов языка L.** Дано слово ω, определить $\omega \in L$ или $\omega \notin L$. Для решения этой задачи служат распознающие автоматы. Формально автомат определяется – $A=\langle \mathcal{A}, S, P \rangle$, где A – основной алфавит, *S* – алфавит состояний, *P* – правила вида, s_i, $w_i \to S_j$, $w_i \in \mathcal{A}^*$, $s_i, s_j \in S$

Автоматная (регулярная) грамматика – АГ и конечный автомат – КА.

Автоматная грамматика $AГ=\langle \mathcal{A}, V, P, S_0 \rangle$, где A – основной алфавит; *P* – правила вида $v_i \to a_j v_j, v_i, v_j \in V, a_j \in A; s_0$ – аксиома. ***Конечный автомат*** $AK=\langle \mathcal{A}, S, P, s_0, s_k \rangle$, где A – основной алфавит; *S* – алфавит состояний; s_0 – начальное состояние; s_k – конечное состояние; *P* – правила вида $s_i, a_i \to s_j, s_i, s_j \in S, a_i \in \mathcal{A}$.

14. Контекстно-зависимая грамматика, КЗ язык, ЛОА.

КЗГ имеет правила вида:

$\alpha A\beta \to \alpha\gamma\beta$, где A - вспомогательный символ,

$\alpha, \beta, \gamma \in (\mathcal{A} \cup V)^*$ - слова, составленные из букв алфавита А и вспомогательного алфавита V.

Свойства КЗГ:

а) α, β - называется контекстом, который сохраняется в левой и правой части правил подстановок.

б) $\alpha \neq \varnothing$ - не может быть пустым символом, поэтому КЗГ ещё называется неукорачивающей грамматикой.

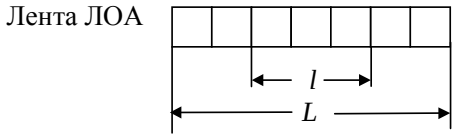
КЗГ применяется в математической лингвистике для анализа естественных языков, там они называются грамматикой непосредственно составляющих (НС - грамматики).

Языки, которые порождаются КЗ-грамматиками, называются КЗ-языками.

Распознающий линейно ограниченный автомат (ЛОА)

Представляет собой машину Тьюринга (МТ) с ограниченной длиной входной ленты, которая называется **линейно-ограниченным автоматом (ЛОА)**.

l - длина слова. L - длина ленты (память) $L = a + bl$,.



где a, b целые числа.

Обычно ЛОА имеют $L = l$. Доказано, что такое ограничение не уменьшает класс КЗ - языков, которые распознаёт ЛОА.

15. Контекстно-свободные грамматики и магазинные автоматы. КС языки.

Существуют языки, которые не могут быть описаны регулярным выражением и соответственно не может быть построен КА, который распознавал бы слова этого языка.

Пример такого языка $L_3 = a^n c b^n$, где $n = 1, 2, 3, \ldots$ Количество a и b в слове L_3 одинаково, например, слово $(aacbb) \in L_3$, а слово $(aaacbb) \notin L_3$.

Контекстно-свободная грамматика (КСГ).

$КСГ = \{A, V, P, S \in V\}$

A – алфавит языка (терминальный алфавит) = $\{a, b, c \ldots\}$; V – вспомогательный алфавит = $\{S, A, B, C, \ldots\}$, где $S \in V$ – начальный символ – аксиома («начало»); $\sum = A \cup V$ – объединённый алфавит P – правила вида $A \to \alpha$, где $A \in V$, $\alpha \in (\Sigma)^*$ – слово из букв объединённого алфавита.

Языки, которые порождаются КС – грамматиками называются ***контекстно-свободными языками***.

Практические примеры применения КС-грамматик.

Первое практическое применение КСГ нашли для описания синтаксиса языков программирования. В частности в 1958 г. для описания языка Алгол была введена спецификация, т.н. нормальная форма Бэкуса – Наура (БНФ).

БНФ – формальная система для описания синтаксиса языков программирования. Впервые была изложена в отчёте Дж. Бэкуса «Algol 60 Report» (редактор П. Наур). БНФ фактически представляет собой форму описания **любой** контекстно-свободной грамматики. Эквивалентность записи БНФ и КСГ впервые отметил В.М. Курочкин в 1960-м году.

Пример. Формальное определение понятия «число со знаком»

а) БНФ – <число со знаком>::=<число>|+<число>|-<число>

Введём алфавиты

A(терминальный) = $\{a, +, -\}, V(вспомогательный) = \{A\}$,

где a – число, A – число со знаком.

б) правила КС-грамматики, порождающие понятие число со знаком.

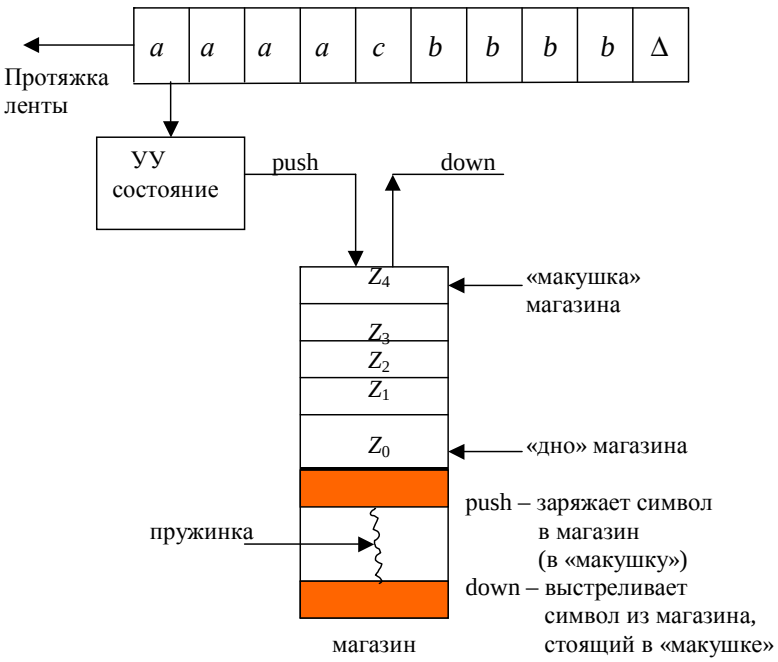
$A \to a$

$A \to +a$

$A \to -a$

Магазинный автомат (МА).

Распознающее устройство для слов КС-языка представляет собой КА сцепленный с магазинной памятью (стеком).



Содержательно работа магазинного автомата описывается следующим образом.

При «заряжании» символа в магазин все символы проваливаются на одну позицию (пружина сжимается»), при «выстреле» выстреливается верхний символ магазина (только один), пружина разжимает и сдвигает всё содержимое магазина на одну позицию вверх. При холостом выстреле – магазин не изменяется.

Определение МА. $MA = \langle \mathcal{A}, V, S, s_0, s_k, z_0 \rangle$

A – алфавит входных символов $\{a, b, c, ...\}, \alpha(слово) \in (\mathcal{A})^*$ на входной ленте, Δ – маркер конца ленты, не входящий в A.

V – алфавит магазина = $\{A, B, C, ..., Z_0\}, Z_0$ – маркер конца магазина, алфавит состояний $S = \{s_0, s_1, ..., s_k\}$.

P – ***правила работы МА 4-х типов***.

1. $S_i, a, A \to S_j, push\ B$.

МА находится в состоянии S_i, считывает входной символ *a*, смотрит на символ *A* «макушки» МА, переводит МА в состояние S_j, «заряжает»» магазин, помещая символ B в «макушку» магазина, и протягивает входную ленту на одну позицию

2. $S_i, a, A \to S_j, down$

всё по аналогии с правилом 1, но удаляет «выстреливает» символ из «макушки» магазина

3. $S_i, a, A \to S_j, blank$

всё по аналогии с правилом 1, не меняет содержимое магазина

4. $S_i, \Delta, Z_0 \to S_k, stop$

входная лента пуста, магазин, МА останавливается

16. Организация таблиц символов компилятора.

В процессе работы компилятор хранит информацию об объектах программы. Как правило, информация о каждом объекте состоит из двух основных элементов: имени объекта и его свойств. Информация об объектах программы должна быть организована таким образом, чтобы поиск ее был по возможности быстрей, а требуемая память по возможности меньше. Кроме того, со стороны языка программирования могут быть дополнительные требования. Имена могут иметь определенную область видимости. Например, поле записи должно быть уникально в пределах структуры (или уровня структуры), но может совпадать с именем объектов вне записи (или другого уровня записи). В то же время имя поля может открываться оператором присоединения, и тогда может возникнуть конфликт имен (или неоднозначность в трактовке имени). Если язык имеет блочную структуру, то необходимо обеспечить такой способ хранения информации, чтобы, во-первых, поддерживать блочный механизм видимости, а во-вторых - эффективно освобождать память по выходе из блока. В некоторых языках (например, Аде) одновременно (в одном блоке) могут быть видимы несколько объектов с одним именем, в других такая ситуация недопустима. Мы рассмотрим некоторые основные способы организации информации в компиляторах: таблицы идентификаторов, таблицы символов, способы реализации блочной структуры.

Таблицы идентификаторов и таблицы символов

Как уже было сказано, информацию об объекте обычно можно разделить на две части: имя (идентификатор) и описание. Удобно эти характеристики объекта хранить по отдельности. Это обусловлено двумя причинами:

1. символьное представление идентификатора может иметь неопределенную длину и быть довольно длинным;
2. как уже было сказано, различные объекты в одной области видимости и/или в разных могут иметь одинаковые имена и незачем занимать память для повторного хранения идентификатора. Таблицу для хранения идентификаторов называют таблицей идентификаторов, а таблицу для хранения свойств объектов - таблицей символов. В таком случае одним из свойств объекта становится его имя и в таблице символов хранится указатель на соответствующий вход в таблицу идентификаторов.

Если длина идентификатора ограничена (или имя идентифицируется по ограниченному числу первых символов идентификатора), то таблица идентификаторов может быть организована в виде простого массива строк фиксированной длины. Некоторые входы могут быть заняты, некоторые - свободны.

Ясно, что, во-первых, размер массива должен быть не меньше числа идентификаторов, которые могут реально появиться в программе (в противном случае возникает переполнение таблицы); во-вторых, как правило, потенциальное число различных идентификаторов существенно больше размера таблицы. Поиск в такой таблице может быть организован методом повторной расстановки.

Второй способ организации таблицы идентификаторов - хранение идентификаторов в сплошном массиве символов. В этом случае идентификатору соответствует номер его первого символа в этом массиве. Идентификатор в массиве заканчивается каким-либо специальным символом (EOS). Второй возможный вариант - в качестве первого символа идентификатора в массив заносится его длина. Для организации поиска в таком массиве создается таблица расстановки.

Таблицы символов и таблицы расстановки

Рассмотрим организацию таблицы символов с помощью таблицы расстановки. Таблица расстановки - это массив указателей на списки указателей на идентификаторы. В каждый такой список входят указатели на идентификаторы, имеющие одно значение функции расстановки.

Вначале таблица расстановки пуста (все элементы имеют значение NIL). При поиске идентификатора id вычисляется функция расстановки H(id) и просматривается линейный список T[H].

Функции расстановки.

Много внимания было уделено тому, какой должна быть функция расстановки. Основные требования к ней очевидны: она должна легко вычисляться и распределять равномерно. Один из возможных подходов заключается в следующем.

1. По символам строки s определяем положительное целое H. Преобразование одиночных символов в целые обычно можно сделать средствами языка реализации. В Паскале для этого служит функция ord, в Си при выполнении арифметических операций символьные значения трактуются как целые.
2. Преобразуем H, вычисленное выше, в номер списка, т.е. целое между 0 и m-1, где m –

размер таблицы расстановки, например, взятием остатка при делении H на m.

Функции расстановки, учитывающие все символы строки, распределяют лучше, чем функции, учитывающие только несколько символов, например в конце или середине строки. Но такие функции требуют больше вычислений.

Таблицы на деревьях

Рассмотрим еще один способ организации таблиц с использованием двоичных деревьев. Будем считать, что на множестве идентификаторов задан некоторый линейный порядок (например, лексикографический), т.е. задано некоторое отношение '<', транзитивное, антисимметричное и антирефлексивное. Каждой вершине двоичного дерева, представляющего таблицу символов, сопоставлен идентификатор. Вершина может иметь нуль, одного (правого или левого) или двух (правого и левого) потомков. Если вершина имеет левого потомка, то идентификатор, сопоставленный левому потомку, меньше идентификатора, сопоставленного самой вершине; если имеет правого потомка, то ее идентификатор больше.

17. Способы задания схем грамматик.

Форма Наура-Бэкуса

При описании синтаксиса конкретных языков программирования приходится вводить большое число нетерминальных символов, и символическая форма записи теряет свою наглядность. В этом случае применяют **форму Наура-Бэкуса (ФНБ)**, которая предполагает использование в качестве нетерминальных символов комбинаций слов естественного языка, заключенных в угловые скобки, а в качестве разделителя - специального знака, состоящего из двух двоеточий и равенства.

Итерационная форма

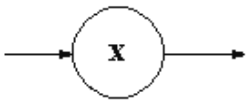
Для получения более компактных описаний синтаксиса применяют итерационную форму описания. Такая форма предполагает введение специальной операции, которая называется итерацией и обозначается парой фигурных скобок со звездочкой. Итерация вида $\{a\}^*$ определяется как множество, включающее **цепочки** всевозможной длины, построенные с использованием символа **a**, и пустую цепочку.

$$\{a\}^* = \{\$, a, aa, aaa, aaaa, ...\}$$

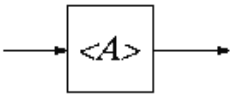
Синтаксические диаграммы

Для того чтобы улучшить зрительное восприятие и облегчить понимание сложных синтаксических описаний, применяют представление правил грамматики в виде синтаксических диаграмм. Правила построения таких диаграмм можно сформулировать в следующем виде:

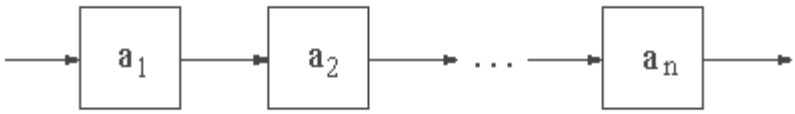
1. Каждому правилу вида $\langle A\rangle \to a_1 | a_2 | \ldots | a_k$ ставится в соответствие диаграмма, структура которой определяется правой частью правила.
2. Каждое появление терминального символа **x** в цепочке a_i изображается на диаграмме дугой, помеченной этим символом **x**, заключенным в кружок.



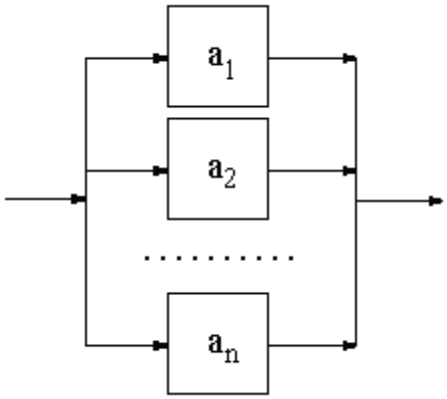
3. Каждому появлению нетерминального символа **<A>** в цепочке a_i ставится в соответствие на диаграмме дуга, помеченная символом, заключённым в квадрат.



4. Порождающее правило, имеющее вид:
   $\langle A\rangle \to a_1 a_2 \ldots a_n$
изображается на диаграмме следующим образом:



5. Порождающее правило, имеющее вид:
   $\langle A\rangle \to a_1 | a_2 | \ldots | a_n$
изображается на диаграмме так:



6. Если порождающее правило задано в виде итерации: $\langle A\rangle \to \{a\}^*$, то ему соответствует диаграмма: