Новосибирский государственный технический университет

Романов Евгений Леонидович

Курс "Основы построения трансляторов"

Методические материалы

2016

По всем вопросам обращайтесь к автору курса Романову Евгению Леонидовичу romanow@vt.cs.nstu.ru или на сайт кафедры вычислительной техники НГТУ http://ermak.cs.nstu.ru .

Все изложенное уложено в http://ermak.cs.nstu.ru/trans/tr.arj архив - 97Kb(Word) или http://ermak.cs.nstu.ru/trans.arj архив - 356Kb(HTML).

© Романов Е.Л. Основы построения трансляторов. (конспект лекций)

1. Введение. Сущность трансляции. Основные термины и определения 3
1. Введение. Сущность трансляции. Основные термины и определения 3
1.1. Фазы трансляции и выполнения программы 3
Препроцессор 3
Трансляция и ее фазы 4
Модульное программирование, компоновка 4
Сущность трансляции. Компиляция и интерпретация 5
Структура транслятора 6
1.2. Связывание. Сравнительная характеристика языков программирования 6
2. Лексический анализ 9
2.1. Сущность лексического анализа 9
Простейший лексический анализатор 10
2.2. Лексический анализатор как конечный автомат 10
Диаграмма состояний и переходов лексического анализатора 12
Лексический анализатор на основе конечного автомата 12
3. Синтаксический анализ 14
3.1. Сущность синтаксического анализа 14
Регулярные грамматики и выражения в лексическом анализе 17
Методы и алгоритмы синтаксического анализа 18
3.2. Нисходящий разбор с возвратами 18
3.3. Рекурсивный спуск 20
3.4. Магазинные автоматы 23
Грамматики класса S (S-грамматики) 23
Грамматики класса Q (Q-грамматики) 26
Грамматики класса LL(1) (LL(1)-грамматики) 27
3.5. Восходящие методы анализа. Свертка-перенос 28
4. Семантический анализ 31
Пример: семантика данных для Си-компилятора 32
Понятие L-value 34
5. Генерация кода. Интерпретация 34
Особенности интерпретации управляющих структур программы 35
Особенности компиляции управляющих структур программы 35
6. Задания к контрольным и лабораторным работам 36
Задания к контрольной работе №1 36
Задания к контрольной работе №2 37
7. Список литературы 38

1. Введение. Сущность трансляции. Основные термины и определения

1.1. Фазы трансляции и выполнения программы

Если языки программирования имеет уже более или менее короткую историю развития, то сама технология подготовки программ, написанных на любом языке программирования, вообще сформировались в начале 60 годов и с тех пор не претерпела существенных изменений. Заложенные тогда принципы оказывают влияние на способы использования стандартных библиотечных функций и разработки больших программ, текст которой содержится в нескольких файлах (модульное программирование).

Подготовка программы начинается с редактирования файла, содержащего текст этой программы, который имеет стандартное расширение для данного языка. Затем выполняется его трансляция, которая включает в себя несколько фаз: препроцессор, лексический, синтаксический, семантический анализ, генерация кода и его оптимизация. В результате трансляции получается объектный модуль -некий "полуфабрикат" готовой программы, который потом участвует в ее сборке. Файл объектного модуля имеет стандартное расширение ".obj". Компоновка (сборка) программы заключается в объединении одного или нескольких объектных модулей программы и объектных модулей, взятых из библиотечных файлов и содержащих стандартные функции и другие полезные вещи. В результате получается исполняемая программа в виде отдельного файла (загрузочный модуль, программный файл) со стандартным расширением -".exe", который затем загружается в память и выполняется.

Препроцессор

Собственно говоря, препроцессор не имеет никакого отношения к языку. Это предварительная фаза трансляции, которая выполняет обработку текста программы, не вдаваясь глубоко в ее содержание. Он производит замену одних частей текста на другие, при этом сама программа так и остается в исходном виде.

ПРЕПРОЦЕССОР -- предварительная фаза трансляции на уровне преобразования исходного текста программы

В языке Си директивы препроцессора оформлены отдельными строками программы, которые начинаются с символа "#". Здесь мы рассмотрим наиболее простые и "популярные".

```
#define        идентификатор  строка_текста
```

Директива обеспечивает замену встречающегося в тексте программы идентификатора на соответствующую строку текста. Наиболее часто она применяется для символического обозначения константы, которая встречается многократно в различных частях программы. Например, размерность массива:

```
#define      SIZE       100
int      A[SIZE];
for (i=0; i<SIZE; i++) {...}
```

В данном примере вместо имени SIZE в текст программы будет подставлена строка, содержащая константу 100. Теперь, если нас не устраивает размерность массива, нам достаточно увеличить это значение в директиве define и повторно оттранслировать программу.

```
#define идентификатор(параметры) строка_с_параметрами
```

Директива отдаленно напоминает определение функции с формальными параметрами, где вместо тела функции используется строка текста. Если препроцессор находит в тексте программы указанный идентификатор со списком фактических параметров в скобках, то он подставляет вместо него соотвествующую строку из директивы define с заменой в строке формальных параметров на фактические. Основное отличие от функции: если функция реализует подобные действия (подстановка параметров, вызов) во время работы программы, то препроцессор -еще до трансляции. Кроме этого, директива define позволяет оформить в таком виде любую часть программы, независимо от того, законченная это конструкция языка или ее фрагмент. В следующем примере стандартный заголовок цикла for представлен в виде директивы define с параметрами:

```
#define      FOR(i,n) for(i=0; i<n; i++)
FOR(k,20) A[k]=0;      // for(k=0; k<20; k++) A[k]=0;
FOR(j,m+2) {...}      // for(j=0; j<m+2;  j++) {...}
```

В таком варианте директива define представляет собой МАКРООПРЕДЕЛЕНИЕ, а замена в тексте программы идентификатора с параметрами на строку - МАКРОПОДСТАНОВКУ.

```
#include <имя_файла>
#include "имя_файла"
```

В текст программы вместо указанной директивы включается текст файла, находящегося в системном или, соответственно, в текущем (явно указанном) каталоге. Наиболее часто в программу включаются тексты заголовочных файлов, содержащие необходимую информацию транслятору о внешних функциях, находящихся в других объектных модулях и библиотеках. Например,

```
#include <stdio.h>
```

включает в программу текст заголовочного файла, содержащего объявления внешних функций из библиотеки стандартного ввода-вывода.

Аналогичные средства в других языках программирования носят название МАКРОПРОЦЕССОР, МАКРОСРЕДСТВА.

Трансляция и ее фазы

Собственно трансляция начинается с лексического анализа программы. ЛЕКСИКА языка программирования -это правила "правописания слов" программы, таких как идентификаторы, константы, служебные слова, комментарии. Лексический анализ разбивает текст программы на указанные элементы. Особенность любой лексики - ее элементы представляют собой регулярные линейные последовательности символов. Например, ИДЕНТИФИКАТОР - это произвольная последовательность букв, цифр и символа "_", начинающаяся с буквы или "_".

СИНТАКСИС языка программирования - это правила составления предложений языка из отдельных слов. Такими предложениями являются операции, операторы, определения функций и переменных. Особенностью синтаксиса является принцип вложенности (рекурсивность) правил построения предложений. Это значит, что элемент синтаксиса языка в своем определении прямо или косвенно в одной из его частей содержит сам себя. Например, в определении оператора цикла телом цикла является оператор, частным случаем которого является все тот же оператор цикла.

СЕМАНТИКА языка программирования - это смысл, который закладывается в каждую конструкцию языка. Семантический анализ -это проверка смысловой правильности конструкции. Например, если мы в выражении используем переменную, то она должна быть определена ранее по тексту программы, а из этого определения может быть получен ее тип. Исходя из типа переменной, можно говорит о допустимости операции с данной переменной.

ГЕНЕРАЦИЯ КОДА - это преобразование элементарных действий, полученных в результате лексического, синтаксического и семантического анализа программы, в некоторое внутреннее представление. Это могут быть коды команд, адреса и содержимое памяти данных, либо текст программы на языке Ассемблера, либо стандартизованный промежуточный код (например, Р-код). В процессе генерации кода производится и его оптимизация.

Модульное программирование, компоновка

Полученный в результате трансляции ОБЪЕКТНЫЙ МОДУЛЬ включает в себя готовые к выполнению коды команд, адреса и содержимое памяти данных. Но это касается только собственных внутренних объектов программы (функций и переменных). Обращение к внешним функциям и переменным, отсутствующим в данном фрагменте программы, не может быть полностью переведено во внутреннее представление и остается в объектном модуле в исходном (текстовом) виде. Но если эти функции и переменные отсутствуют, значит они должны быть каким-то образом получены в других объектных модулях. Самый естественный способ - написать их на том же самом Си и оттранслировать. Это и есть принцип МОДУЛЬНОГО ПРОГРАММИРОВАНИЯ - представление текста программы в виде нескольких файлов, каждый из которых транслируется отдельно. С модульным программированием мы сталкиваемся в двух случаях:

- когда сами пишем модульную программу;

- когда используем стандартные библиотечные функции.

БИБЛИОТЕКА ОБЪЕКТНЫХ МОДУЛЕЙ - это файл (библиотечный файл), содержащий набор объектных модулей и собственный внутренний каталог. Объектные модули библиотеки извлекаются из нее целиком при наличии в них требуемых внешних функций и переменных и используются в процессе компоновки программы.

КОМПОНОВКА - это процесс сборки программы из объектных модулей, в котором производится их объединение в исполняемую программу и связывание вызовов внешних функций и их внутреннего представления (кодов), расположенных в различных объектных модулях. Подробно процесс компоновки применительно к языку Си рассмотрен в 4.6.

В заключение отметим, что источником объектного модуля может быть не только Си-программа, но и программа, написанная на любом другом языке программирования, например, на Ассемблере. Но в этом случае необходимы дополнительные соглашения по поводу "стыковки" вызовов функций и обращений к данным в различных языках.

Сущность трансляции. Компиляция и интерпретация

Под трансляцией в самом широком смысле можно понимать процесс восприятия компьютером программы, написанной на некотором формальном языке. При всем своем различии формальные языки имеют много общего и, в принципе, эквиваленты с точки зрения принципиальной возможности написать одну и ту же программу на одном из них.

КОМПИЛЯЦИЯ - преобразование объектов (данных и операций над ними) с входного языка в объекты на другом языке для всей программы в целом с последующим выполнением полученной программы в виде отдельного шага.

ИНТЕРПРЕТАЦИЯ - анализ отдельного объекта на входном языке с одновременным выполнением (интерпретацией).

Следовательно, компиляция и интерпретация отличаются не характером и методами анализа и преобразования объектов программы, а совмещением фаз обработки этих объектов во времени. То есть при компиляции фазы преобразования и выполнения действий разнесены во времени, но зато каждая из них выполняется над всеми объектами программы одновременно. При интерпретации, наоборот, преобразование и выполнение действий объединены во времени, но для каждого объекта программы.

Если посмотреть на эти различия несколько с другой стороны, то можно заметить, что интерпретатор непосредственно выполняет действия, связанные с определением или преобразованием объектов программы, а компилятор - переводит их на другой (не обязательно машинный язык). Отсюда можно сделать несколько выводов:

- для выполнения программы, написанной на определенном формальном языке после ее компиляции необходим интерпретатор, выполняющий эту программу, но уже записанную на выходном языке компилятора;

- процессор и память любого компьютера (а в широком смысле и вся программная среда, создаваемая операционной системой, является ИНТЕРПРЕТАТОРОМ машинного кода);

- в практике построения трансляторов часто встречается случай, когда программа компилируется со входного языка на некоторый промежуточный уровень (внутренний язык), для которого имеется программный интерпретатор. Многие языковые системы программирования, называемые интерпретаторами, на самом деле имеют фазу компиляции во внутренне представление, на котором производится интерпретация.

Выходной язык компилятора может быть машинным языком для компьютера с другой архитектурой, нежели тот, в котором работает компилятор. Такой компилятор называется КРОСС-КОМПИЛЯТОРОМ, а сама система программирования КРОСС-СИСТЕМОЙ. Такие системы используются для разработки программ для архитектур, не имеющих собственных операционных систем или систем программирования (контроллеры, управляющие микропроцессоры).

Таким образом, граница между компиляцией и интерпретацией в трансляторе может перемещаться от входного языка (тогда мы имеем чистый интерпретатор) до машинного кода (тогда речь идет о чистом компиляторе).

Создание слоя программной интерпретации для некоторого промежуточного языка в практике построения трансляторов обычно встречается при попытке обеспечить совместимость для имеющегося многообразия языков программирования, операционных систем, архитектур и т.д. То есть определяется некоторый внутренний промежуточный язык, достаточно простой, чтобы для него можно было написать интерпретатор для всего

имеющегося многообразия операционных систем или архитектур. Затем пишется одни (или несколько) компиляторов для одного (или нескольких) входных языков на этот промежуточный уровень. Приведем примеры такой стандартизации:

- для обеспечения совместимости и переносимости трансляторов на компьютеры с различной архитектурой или с различными операционными системами был разработан универсальный внутренний язык (P-код). Для каждой такой архитектуры необходимо реализовать свой интерпретатор P-кода. При этом все разнообразие имеющихся компиляторов с языков высокого уровня на P-код может быть использовано без каких-либо изменений.

- язык программирования Java аналогично был разработан для обеспечения переносимости различных приложений в среде Internet.

Структура транслятора

Самое главное в процессе трансляции состоит в том, что он не является линейным, то есть последовательным преобразованием фрагмента программы одного языка на другой. На процесс трансляции одного фрагмента обязательно оказывают влияние другие фрагменты программы. Поэтому трансляция представляет собой несколько последовательных фаз анализа программы, на каждой из которой текст программы разделяется на все более "тонкие" компоненты, а информация о них группируется в некоторое внутреннее представление программы (деревья, таблицы). Затем, или параллельно с этим осуществляется синтез программы уже на выходном языке программирования с использованием информации из внутреннего представления.

Отдельные фазы трансляции могут быть связаны между собой различным образом, через данные в памяти или через файл, что не меняет сущности процесса:

- каждая фаза транслятора получает файл данных от предыдущей фазы, обрабатывает его (линейным или каким-либо другим, например рекурсивным алгоритмом), создает внутренние таблицы данных и по ним формирует выходной файл с данными для следующей фазы;

- фазы трансляции вызывают одна другую в процессе обработки соответствующих языковых единиц. Синтаксический анализ является обычно центральным в такой структуре. То есть основной программой транслятора является синтаксический анализатор, который при анализе структурной единицы языка, называемой предложением (выражение, оператор, определение типа или переменной), вызывает лексический анализатор, для чтения очередной лексической компоненты (идентификатора, константы), а по завершении разбора - семантическую процедуру, процедуры генерации кода или интерпретации. Из этой схемы выпадает только препроцессор, который обычно представляет собой независимую предварительную фазу трансляции.

1.2. Связывание. Сравнительная характеристика языков программирования

Трансляция и последующие действия по подготовке программы к выполнению представляют собой процесс преобразования программы, записанной на некотором формальном языке, в другую формальную систему - архитектуру компьютера, в которой она может быть выполнена (интерпретирована). Для понимания этого процесса, а также отличий, имеющихся в различных языках программирования, в [3] введено понятие СВЯЗЫВАНИЯ, а также ВРЕМЕНИ СВЯЗЫВАНИЯ.

СВЯЗЫВАНИЕ -- процесс установления соответствия между объектами и их свойствами программы на формальном языке программирования (операции, операторы, данные) и объектами архитектуры компьютера (команды, адреса.

ВРЕМЕНЕМ СВЯЗЫВАНИЯ называется соответственно фаза подготовки программы к выполнению (трансляция, компоновка, загрузка), на которой производится это действие. Заметим, что различные характеристики одного и того же объекта (например, переменной) могут связываться с различными элементами архитектуры в разное время, то есть процесс связывания не является одномоментным. Для начала перечислим возможные времена связывания:

- при определении языка;

- при реализации компилятора;

- во время трансляции, включающей в себя:

- препроцессор (макропроцессор)

- лексический, синтаксический и семантический анализ, генерацию кода и его оптимизацию;
- компоновку (связывание);
- во время загрузки программы;
- во время выполнения программы, в том числе:
- при входе в модуль (процедуру, функцию);
- в произвольной точке программы.

В качестве примера рассмотрим простейший фрагмент программы, для которого перечислим более-менее полный перечень времен связывания его различных свойств с элементами архитектуры компьютера:

```
int a,b;  … a+b …
```

1. Тип переменных int - как способ определения целой переменной в машинном слове стандартной длины (представление целого со знаком, дополнительный код), связывается с аналогичной формой представления данных в компьютере при определении языка. Язык Си характерен тем, что базовые типы данных в нем полностью совпадают с соответствующими формами представления данных в компьютере.
2. Конкретная размерность переменной int определяется при реализации соответствующего компилятора.
3. Имя **a** может быть определено в конструкции вида **#define a 0x11FF**. В этом случае имя (псевдо-переменная) связывается со своим значением на первой фазе трансляции - в препроцессоре.
4. Если переменная определяется обычным способом в виде **int a;** то связывание переменной с соответствующим ей типом происходит во время трансляции (на фазе семантического анализа).
5. Если переменная определяется как внешняя (глобальная, вне тела функции), то смысл ее трансляции заключается в распределении под нее памяти в сегменте данных программы, который создается для текущего модуля (файла). Но при этом сама распределенной памяти к конкретной оперативной памяти осуществляется в несколько этапов:

   - при трансляции переменная привязывается к некоторому относительному адресу в сегменте данных объектного модуля (то есть ее размещение фиксируется только относительно начала модуля)

   - при компоновке (связывании) сегменты данных и команд различных объектных модулей объединяются в общий программный файл, представляющий собой образ памяти программы. В нем переменная получает уже относительный адрес от начала всей программы.

   - при загрузке программы в некоторую область памяти (например, в DOS или в режиме эмуляции DOS в WINDOWS) она может размещаться не с самого начала этой области. В этом случае осуществляется привязка адресов переменных, заданных в относительных адресах от начала программного модуля к адресам памяти с учетом перемещения программного модуля (так называемый ПЕРЕМЕЩАЮЩИЙ ЗАГРУЗЧИК, которая имеет место для exe-файлов в DOS).

   - если программа работает не в физической, а в виртуальной памяти, то процесс загрузки может быть несколько иным. Программный модуль условно считается загруженным в некоторое виртуальное адресное пространство (с перемещением или без него как всей программы, так и отдельных ее сегментов). Реальная загрузка программы в память осуществляется уже в процессе работы программы по частям (сегментам, страницам), причем установление соответствия (или связывание) виртуальных и физических адресов осуществляется динамически операционной системой с использованием соответствующих аппаратных средств.
6. Если переменная определяется как автоматическая (локальная внутри тела функции или блока), то она размещается в стеке программы:

   - во время трансляции определяется ее размерность и генерируются команды, которые резервируют под нее память в стеке в момент входа в тело функции (блок). То есть в процессе трансляции переменная связывается только с относительным адресом в стеке программы;

   - связывание локальным переменной с ее адресом в сегменте стека осуществляется при выполнении в момент входа в тело функции (блок). Благодаря такому способу связывания в рекурсивной функции существует столько “экземпляров” локальных переменных, сколько раз функция вызывает сама себя.
7. Тип операции “+” в конкретном выражении **a+b** определяется при трансляции в зависимости от типов операндов. В данном случае генерируется операция целого сложения.
8. С точки зрения времени связывания понятие ИНИЦИАЛИЗАЦИЯ внешних переменных можно определить как связывание переменных с их значениями в процессе трансляции программы (**int a=10;**) С этой точки

зрения обычное присваивание можно рассматривать как связывание переменной с ее значением во время выполнения программы.

С понятием связывания тесно переплетаются понятия СТАТИЧЕСКОГО и ДИНАМИЧЕСКОГО определения данных. Статически определенные данные имеют раннее время связывания - обычно во время трансляции программы, динамические данные - позднее, во время выполнения. Этот принцип легко проиллюстрировать средствами языка Си, в котором все действия, связанные с динамическими данными и поздним связыванием производятся только в явном виде. Возьмем, к примеру, массивы:

- обычное статическое определение массива предполагает его связывание с памятью во время трансляции программы, поэтому тип сохраняемых в нем элементов и их количество должны быть величинами неизменными:

```
int A[100];
```

- динамический массив, размерность которого определяется в процессе работы программы, может быть вычислена в момент создания, но затем не меняется, может быть представлен в Си указателем на динамическую переменную (массив) соответствующей размерности, создаваемую и уничтожаемую внешними функциями. Легко представить себе язык программирования, в котором массивы такого рода определяются в самом трансляторе, в этом случае можно говорить о связывании массива с его размерностью и памятью во время выполнения программы, например, в момент входа в модуль (блок):

```
double *p;
p = malloc(sizeof(double)*n);       // или
p = new double[n];
for (i=0; i<n; i++) p[i] = 5.44;
// ПРИМЕР СИНТАКСИСА ДИНАМИЧЕСКОГО МАССИВА
// n = getnum();
// ReDim double A[n];
```

- виртуальный массив, размерность которого может меняться уже в процессе работы программы, по мере заполнения его данными, в Си также может быть смоделирован с использованием функций перераспределения динамической памяти realloc. При определении таких массивов в самом трансляторе можно говорить о связывании массива с его размерностью и памятью во время выполнения программы, причем многократной, во время заполнения его данными, то есть в любой точке программы:

```
double *p; int n=10;
p = malloc(sizeof(double)*n);
for (i=0; i<10000; i++)
   {
   if (i >= n)            // Если надо,
         { n = n+10;      // увеличить размерность
         p = realloc(p, sizeof(double)*n);
         }
   p[i] = 5.44
   }
// ПРИМЕР СИНТАКСИСА ВИРТУАЛЬНОГО МАССИВА
// ReDim double A[];
// for (i=0; i<10000; i++) A[i]=5.44;
```

- виртуальный массив, в котором может меняться не только размерность, но и типы хранящихся в нем элементов, можно смоделировать в Си с помощью динамического массива указателей на переменные - элементы массива:

```
void **p; int n=10;
p = malloc(sizeof(void*)*n);
int b; double c;
i = 5;
if (i >= n)            // Если надо,
      { n = i+10;      // увеличить размерность
      p = realloc(p, sizeof(void *)*n);
```

```
        }
p[i] = &b;
// ПРИМЕР СИНТАКСИСА ВИРТУАЛЬНОГО МАССИВА
// С ПРОИЗВОЛЬНЫМИ ТИПАМИ ЭЛЕМЕНТОВ
// ReDim  A[];
// int b; double c;
// A[5] = b;
// A[66]= c;
```

Из рассмотренного примера видно, что реализация позднего связывания в языках программирования связана с использованием неявных (скрытых) указателей на структуры данных. Это предположение можно подтвердить и примером связывания для к такого объекта, как функция:

- тело функции является статическим объектом. То есть память под нее выделяется во время трансляции (когда генерируется программный код) в создаваемом объектном модуле. Что же касается вызова функции, то связывание вызова функции с самой функцией может производиться на разных этапах;

- если функция вызывается в том же самом модуле, где она определена, то связывание вызова функции с ее телом (адресом) осуществляется в процессе трансляции;

- если функция определена в другом модуле (частный случай, библиотека), то связывание вызова функции с ее телом (адресом) осуществляется при компоновке. Для этой цели тело функции в объектном модуле сопровождается информацией, именуемой "точкой входа", сам вызов сопровождается "внешней ссылкой", а для корректного формирования вызова Си-компилятору необходимо объявить прототип внешней функции. Для библиотечных функций необходимо подключить заголовочный файл, в котором они содержатся.

В технологии объектно-ориентированного программирования существует термин ПОЛИМОРФНАЯ ФУНКЦИЯ. В Си++ она называется ВИРТУАЛЬНОЙ. На самом деле она представляет собой группу одноименных функций, определенных соответственно для группы родственных (производных классов). При вызове такой функции для объекта обобщающего их класса (базового класса) программа должна идентифицировать, к какому конкретно классу относится текущий объект и выбрать соответствующую ему функцию. С точки зрения понятия связывания это означает, что связывание вызова функции с ее телом может осуществляться в таком случае во время, и только во время работы программы. Действительно, в Си++ механизм виртуальных функций реализуется при помощи массива указателей на функции, который назначения объекту в момент его создания, то есть при выполнении программы.

К позднему (динамическому) связыванию функций относятся и используемые в Windows ДИНАМИЧЕСКИ СВЯЗЫВАЕМЫЕ БИБЛИОТЕКИ (DLL-Dynamic Linking Library) . Фактически в них процесс связывания вызова и тела внешней функции, выполняемый при компоновке, откладывается до момента загрузки программы. В этом случает программный файл содержит несвязанные вызовы внешних функций (внешние ссылки) и перечень используемых библиотек. Загрузка требуемых библиотек и связывание внешних ссылок и точек вход производится в момент загрузки программного файла. Этот способ дополнительно позволяет разделять одну и ту же библиотеку несколькими программами в общем адресном пространстве.

В заключение подчеркнем различие между компиляцией и интерпретацией с точки зрения понятия связывания. Компиляция обычно предусматривает однократное связывание объектов программы с элементами архитектуры при трансляции программы, а интерпретация - многократное связывание при интерпретации соответствующего фрагмента программы.

2. Лексический анализ

2.1. Сущность лексического анализа

Лексический анализ (ЛА) является первой фазой трансляции. Термин "лексика" в обычном языке подразумевает правила составления слов из букв. В языке программирования ЛА соответствует фаза трансляции, в которой из последовательности отдельных литер (символов, букв языка) выделяются слова (лексемы, символы следующей фазы - синтаксического анализа). Типичными словами в языке программирования являются такие компоненты как КОММЕНТАРИИ, ИДЕНТИФИКАТОРЫ, КОНСТАНТЫ, СЛУЖЕБНЫЕ СЛОВА, ЗНАКИ ОПЕРАЦИЙ.

ЛА является наиболее простой и формализованной фазой трансляции. Любой алгоритм ЛА базируется на последовательном просмотре текста, с возвратом и перечитыванием из входной последовательности не более чем одного символа, поэтому программу ЛА иногда называют СКАНЕРОМ. Формальной основой описания процесса ЛА являются конечные автоматы.

Простейший лексический анализатор можно построить, не прибегая ни к каким формальным методам, просто анализируя последовательности символов, образующих лексические единицы. Соответствующий пример приведен в файле hardlex.cpp. Из приведенного ниже фрагмента видно, что программа "зацепляет" первый символ, определяющий начало лексической единицы (букву, цифру и т.д.) а затем просматривает строку до конца элемента лексики (идентификатора, константы). Некоторые лексические единицы при этом имеют еще и собственное значение, которое выходит за рамки алгоритма распознавания. Например, для идентификатора и константы важен не только факт их распознавания, но и их значение, заключенное в цепочке символов. Поэтому анализатор перед началом очередного цикла фиксирует начало распознаваемой последовательности символов, для сохранения ее значения.

```
while (1)
      { fix=i;
        switch(s[i])
            {
case '"':   // Распознавание строковой константы ". . ."
            // с двойными "" внутри
mmm:        i++; while (s[i] !='"') i++;
            i++; if (s[i]=='"') goto mmm;
            lexem(1); break;
case '/':   i++;      // Распознавание / и /*
            if (s[i]!='*')
                  { lexem(14); break; }
            // Распознавание комментария /* ... */
n1:         while (s[i] !='*') i++;
            i++; if (s[i]=='/')
                  { i++; lexem(2); break; }
            goto n1;
case '+':   i++;      // Распознавание += и +
            if (s[i]=='=')
                  { i++; lexem(5); }
            else  lexem(15);
            break;
case '<':   i++;      // Распознавание << и <
            if (s[i]=='<')
                  { i++; lexem(6); }
            else      lexem(16); break;
default:    if (isalpha(s[i]))
            // Распознавание идентификатора
            { i++; while (isalpha(s[i])) i++;
              lexem(11); break; }
            // Распознавание константы
            if  (isdigit(s[i]))
                  { i++; while (isdigit(s[i])) i++;
                  lexem(12); break; }
```

2.2. Лексический анализатор как конечный автомат

Формальной основой ЛА являются конечные автоматы (КА). КА является формальным математическим аппаратом, широко используемым в компьютерной технике. В проектировании аппаратных средств КА используются при разработке управляющей схемы, реализующей заданный алгоритм. Точно так же, в любой программе достаточно сложная логика последовательности действий может быть реализована как напрямую в виде последовательности условных и циклических конструкций, так и в виде программного КА. Для начала дадим неформальное определение КА.

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 А, при получении сигнала S объект переходит в состояние В и при этом выполняет действие D - то такая система будет представлять собой 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.

О том, что конечный автомат может использоваться при анализе последовательностей различных символов в строке, несложно убедиться на простом примере. Пусть требуется написать программу, которая исключает из строки комментарии вида “ /* ... */ ”

Прежде всего, необходимо определить состояния конечного автомата. Таковыми являются состояния программы, возможные при просмотре очередного символа строки:

- состояние 0 - идет обычный текст;

- состояние 1 - обнаружен символ “/”;

- состояние 2 - обнаружено начало комментария “/*”;

- состояние 3 - в комментарии обнаружен символ “*”.

Программа, представляющая собой КА, будет выглядеть следующим образом:

```
void f(char in[],char out[])
{int i, s, j;
for(i=0,s=0,j=0; in[i]!=0; i++)
switch(s)
       {
case 0:      if (in[i])!='/' out[j++] = in[i];
             else s=1;
             break;
case 1:      if (in[i]!='*')
                    {out[j++]=in[i-1]; out[j++]=in[i]; s=0;}
             else s=2;
             break;
case 2:      if (in[i]=='*') s=3;
             break;
case 3:      if (in[i]=='/') s=0;
             break;
       }
}
```

Подробнее рассмотрим характерные особенности этой программы:

- программа представляет собой цикл, в каждом шаге которой анализируется очередной символ из входной строки. Заметим, что только текущий. Программа принципиально не анализирует ни предыдущих, ни последующих символов. Текущий символ играет, таким образом. Роль входного сигнала, по которому автомат переходит из одного состояния в другое.

- программа имеет переменную s, которая и представляет собой текущее состояние КА. В теле основного цикла программы выполняется переключение (switch) по всем возможным значениям этой переменной - состояниям КА.

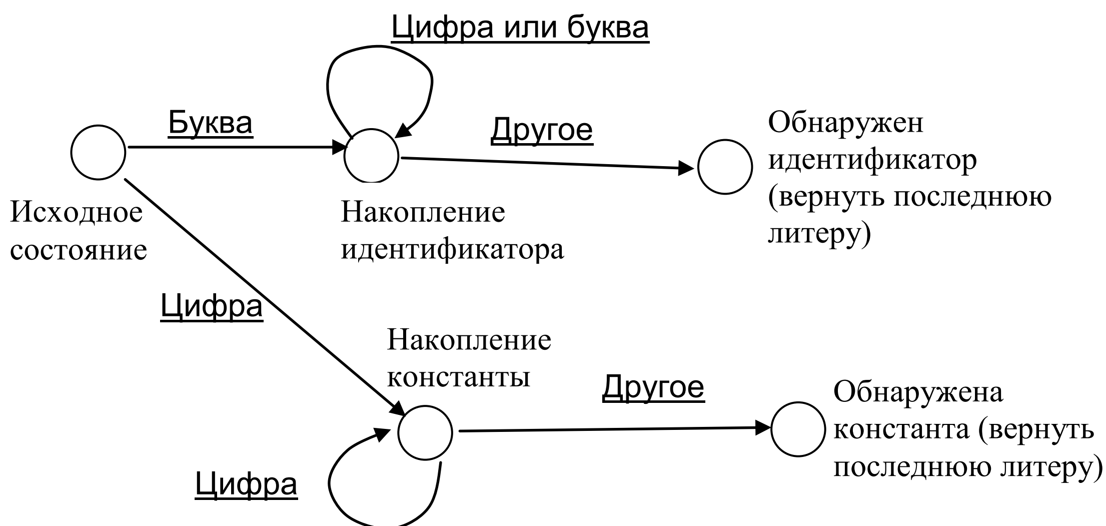
- в каждом состоянии реализуется логика его перехода в другие состояния на основе анализа текущего символа - входного сигнала и вырабатывается соответствующее действие. Например, в состоянии 1, когда программа уже обнаружила символ “/” - возможное начало комментария, она проверяет, не является ли текущий символ “*”. Если это так, автомат переводится в состояние 2 - нахождение

внутри комментария, если нет, то в выходную строку переписывается предыдущий "/" и текущий символы, а автомат возвращается в состояние 0.

Диаграмма состояний и переходов лексического анализатора

Пониманию сущности конечного автомата может в значительной степени помочь ДИАГРАММА СОСТОЯНИЙ-ПЕРЕХОДОВ. Каждому состоянию автомата, которое на диаграмме отображается кружочком, соответствует "поведение" конечного автомата. Например, если автомат распознает константу как последовательность цифр, то он должен иметь состояние, которое можно назвать как "ожидание очередной цифры". Переход автомата из одного состояния в другое отображается направленной дугой. Условие, при котором происходит этот переход, а также действие, которое осуществляет конечный автомат при переходе, подписываются над дугой. В лексическом анализаторе условием перехода является значение очередного символа строки, который анализируется, а неявным действием является продвижение к очередному символу в строке.

Проектирование КА с помощью диаграммы состояний-переходов происходит содержательно. Определяются состояния КА, им присваивается "смысл", а затем определяется, при каких условиях происходит переход из одного в другое. Вот так выглядит диаграмма состояний-переходов для обычной десятичной константы и идентификатора.



Используя диаграмму состояний-переходов, можно напрямую написать программу ЛА, опираясь на приведенный выше пример реализации автомата, обрабатывающего текст:

- основной цикл программы представляет собой последовательный просмотр строки;

- программа имеет переменную состояния, по которой в теле цикла организуется переключение;

- каждой ветке переключателя соответствует одно состояние КА и один узел диаграммы состояний, в ней программируется "поведение" КА для данного состояния - анализируется текущий символ и по нему определяется новое состояние КА и производимое действие;

Лексический анализатор на основе конечного автомата

Диаграмма состояний-переходов является наглядным средством неформального проектирования ЛА. На самом деле существует другой способ проектирования таких программ, основанный на более регулярном и естественном представлении КА, и, в частности, КА для лексического анализа.

В программе определяется переменная - текущее состояние КА (ST);

Все множество входных символов разбивается на непересекающиеся множества - классы. Классы выбираются таким образом, чтобы обеспечивалось однозначное переключение КА, если анализируются не сами символы, а их классы. Например, если ЛА должен различать комментарии, идентификаторы, восьмеричные, десятичные и шестнадцатеричные и строковые константы, то множество символов необходимо разбить на следующие классы:

- цифра 0.

- цифры 1-7.
- цифры 8-9.
- буквы A-F,a-f.
- буква “x”.
- остальные буквы.
- символ “/”.
- символ “*”.
- символ “””.
- остальные символы.

Проще говоря, символы, на которые КА должен обеспечивать различную реакцию, должны быть разнесены в отдельные классы. Естественно, что определением класса символа должна заниматься отдельная функция. Для приведенного примера она имеет вид:

```
int class(char c)
{ switch ( c)
      {
case      '*':      return(7);
case      '"':      return(8);
case      '/':      return(6);
case      '0':      return(0);
case      '8':      return(2);
case      '9':      return(2);
case      'x':      return(3);
default:  if (isdigit(c)) return(1);
          if (isalpha(c))
                {if ((touppeк(c)>='A')&&(toupper(c)<='F'))
                        return(4);
                return(5);
                }
          return(9); }}
```

Далее в программе необходимо определить матрицу переходов. Матрица переходов - это двумерный массив, который для каждой пары “состояние (строка) и класс символа (столбец)” определяет новое состояние, в которое он переводится. Номер этого состояния и находится в матрице. Матрица переходов строится по соответствующей диаграмме состояний-переходов и фактически определяет поведение КА. Принцип заполнения матицы прост: если в состоянии S1 и входном символе L1 на диаграмме имеется дуга (переход) в состояние S2 то элемент массива D[S1][K1] должен быть инициализирован значением S2, где K1=class(L1). В рассматриваемом примере матрица будет выглядеть так:

.

```
int D[11][10]= {
{ 2, 5, 5, 1, 1, 1, 6,-8, 9,-9 },
{ 1, 1, 1, 1, 1, 1,-1,-1,-1,-1 },
{ 5, 4, 5, 3,-4,-4,-4,-4,-4,-4 },
{ 3, 3, 3,-2, 3,-2,-2,-2,-2,-2 },
{ 4, 4,-3,-3,-3,-3,-3,-3,-3,-3 },
{ 5, 5, 5,-4,-4,-4,-4,-4,-4,-4 },
{-5,-5,-5,-5,-5,-5,-5, 7,-5,-5 },
{ 7, 7, 7, 7, 7, 7, 7, 8, 7, 7 },
{ 7, 7, 7, 7, 7, 7,-6, 7, 7, 7 },
{ 9, 9, 9, 9, 9, 9, 9, 9,10, 9 },
{-7,-7,-7,-7,-7,-7,-7,-7, 9,-7 } };
```

Тогда поведение конечного автомата программируется в первом приближении таким простым циклом:

```
for (ST=0,i=0;S[i]!=0;i++) {CL=class(S[i]);ST=D[ST][CL];}
```

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.

```
int      W[]={ 1,1,1,1,1,0,1,0,0 };
if (ST  < 0)
      { int j; ST = -ST-1;       // тип лексемы
      i=i-W[ST];                   // вернуть символы
      printf("%s ",out[ST]);       // вывести цепочку
for (j=FIX; j<i; j++) putchar(S[j]);
puts("");
ST=0; FIX=i; }
```

Для фиксации начала цепочки символов, образующих лексему (слово), в программе вводится переменная FIX, которая при любом переходе в начальное (нулевое) состояние запоминает расположение в строке текущего символа. Заготовка программы ЛА на основе конечного автомата находится в файле lexan.cpp.

3. Синтаксический анализ

3.1. Сущность синтаксического анализа

Сущность синтаксического анализа программы достаточно сложна, чтобы излагать ее "на пальцах". Даже такое средство представления как конечный автомат является недостаточно "мощным" для этого. В первом приближении это можно доказать, ссылаясь на вложенный характер конструкций языка. Элементы лексики представляют собой линейные последовательности, анализ которых и производится конечными автоматами. Синтаксис же предложения не является линейной структурой. Если определения элементов синтаксиса языка (выражения, операторы) являются теми единицами, из которых строится программа, то взаимоотношение этих единиц в конкретной программе может быть представлено в виде дерева. А с деревом тесно связаны такие понятия, как рекурсия и стек. Таким образом, интуитивно становится ясным, что для определения и анализа синтаксиса языка необходим математический аппарат, который допускает рекурсивное определение и порождение своих элементов, а при их анализе используются деревья, рекурсивные функции и работа со стеком.

Формальные грамматики и языки

Формальные грамматики являются математическим аппаратом, который исследует свойства цепочек символов, порожденных заданным набором правил. Определение формальной грамматики (ФГ) включает в себя:

множество терминальных символов **T**;

множество нетерминальных символов **N**;

начальный нетерминальный символ **Z** из множества **N**;

множество порождающих правил вида **A ::= B**, где **B** - цепочка из любых символов грамматики **(N U T)**, **A**-цепочка, содержащая хотя бы один нетерминальный символ.

В дальнейшем мы будем представлять правила грамматики в таком виде:

.

```
E ::= E + T
E ::= E - T          или        E ::= E + T | E – T
```

Где символ **::=** разделяет правую и левую части. В дальнейшем любой символ, который используется для описания самих средств построения предложений языка (правил и т.д.), но не входит в множество символов, из которых они строятся , будем называть МЕТАСИМВОЛОМ. Если два и более правила имеют одинаковую левую часть, то они могут быть объединены, причем их правые части разделяются вертикальной чертой - тоже метасимволом.

ПРЕДЛОЖЕНИЕ ЯЗЫКА -- цепочка терминальных символов.

НЕПОСРЕДСТВЕННЫЙ ВЫВОД -- замена в некоторой цепочки последовательности символов из правой части на последовательность символов из левой, то есть **xAy -> xBy**.

ПОСЛЕДОВАТЕЛЬНОСТЬ НЕПОСРЕДСТВЕННЫХ ВЫВОДОВ основана на том факте, что непосредственный вывод можно применять несколько раз, в том числе и повторно для одних и тех же правил (рекурсивно). Если для цепочки **aaa** существует такая последовательность непосредственных выводов, которая приводит к цепочке **bbb**, то говорят, что цепочка **bbb** выводится из **aaa**.

ПРАВИЛЬНОЕ ПРЕДЛОЖЕНИЕ -- цепочка терминальных символов, выводимая из начального нетерминального символа грамматики **Z**.

РЕКУРСИВНОЕ ПРАВИЛО -- правило, в правой части которого содержится его левая часть, то есть типа **A::=xAy.**

Два правила образуют НЕПРЯМУЮ РЕКУРСИЮ, если они имеют вид:

B ::= xCy,   C ::= sBr.

Сразу же поясним смысл терминов и определений. Цепочка - это последовательность символов, возможно и пустая. Цепочка терминальных символов (предложение) языка обладает тем свойством, что из нее уже не может быть выведена никакая другая цепочка. Это происходит потому, что в левой части любого правила должен быть хотя бы один нетерминальный символ. Отсюда и происходит название ТЕРМИНАЛЬНЫЙ, то есть конечный символ. Таким образом, формальная грамматика своим набором правил определяет правила преобразования цепочек одного вида в другие. Естественно, если ограничить возможные варианты терминальных цепочек только выводимыми из начального символа грамматики **Z**, то мы получим множество правильных предложений. Последовательность применений правил грамматики к начальному символу образуют древовидную структуру, корнем которой является начальный символ **Z**, а концевыми вершинами являются терминальные символы, образующие при обходе дерева слева направо правильное предложение.

Грамматики по ограничениям, накладываемым на правила, образуют несколько классов:

класс 2 - КОНТЕКСТНО-СВОБОДНЫЕ грамматики, имеющие в левой части любого правила единственный нетерминал. Такие грамматики являются основой построения синтаксиса любого языка. Сам нетерминал в левой части обозначает не что иное, как синтаксическую конструкцию, причем возможность ее замены на левую часть - описание этой конструкции, возможно в любой цепочке, где этот нетерминал встречается, то есть в любом контексте. В качестве примера приведем известную грамматику четырех арифметических действий. В дальнейшем по умолчанию все нетерминалы будут

обозначаться большими латинскими буквами, все терминалы - знаками и маленькими латинскими буквами.

.

```
N =  {Z,E,R,F},        T = {+,-,/,*,(,),f}
Z ::= E
E ::= E + R | E - R | R
R ::= R * F | R / F | F
F ::= f | (E)
```

класс 1 - контекстно-зависимые грамматики, ограничение - длина цепочки в правой части правила не превышает длину цепочки в правой. Термин КОНТЕКСТНО-ЗАВИСИМАЯ характеризует частный случай правил в такой грамматике, имеющих вид **xAy ::=xa...ay,** когда замена нетерминала **A** на цепочку **a...a** возможна на только в окружении некоторых символов, то есть в контексте. Этот вид грамматик является достаточно сложным и обычно в синтаксическом анализе не используется.

класс 3 - регулярные грамматики, правила в которых могут иметь в правой части не более одного терминального. а при его наличии - и не более одного нетерминального символов, то есть

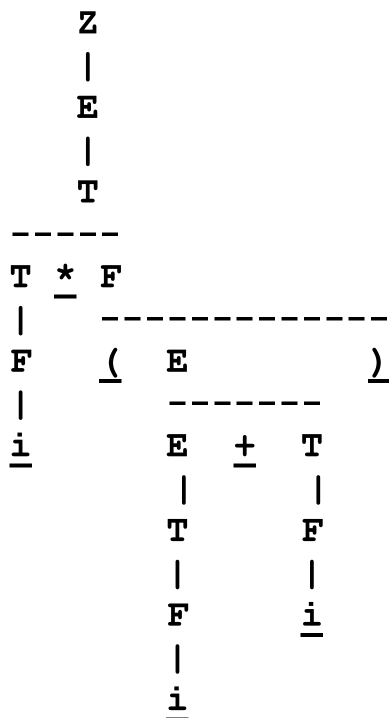
.

```
X ::= aY
X ::= Ya
X ::= a
X ::= ε(пустая цепочка)
```

такие грамматики эквивалентны по своим свойствам конечным автоматам и используются для описания лексики при построении лексических анализаторов.

Теперь следует разобраться, в каких взаимоотношениях находятся формальные грамматики и синтаксический анализ. Синтаксис любого языка программирования определяется контекстно-свободной формальной грамматикой - системой терминальных и нетерминальных символов и множеством правил. Анализируемая программа представляется в такой грамматике предложением языка. Задача синтаксического анализа - определить, является ли это предложение правильным и построить для него последовательность непосредственных выводов из начального символа **Z**, то есть дерево синтаксического разбора. Рассмотрим в качестве примера ту же самую грамматику и цепочку в ней вида **i*(i+i)**

.



Заметим, что последовательность непосредственных выводов при построении дерева однозначно соответствует вложенности данных синтаксических конструкций и обычно соответствует порядку их определения или выполнения (последнее качается уже семантического разбора и генерации кода или интерпретации).

Регулярные грамматики и выражения в лексическом анализе

Регулярные грамматики также, как и конечные автоматы, могут быть использованы для описания лексики. Причем они являются эквивалентными друг другу. Так в правиле вида **U ::=aX** нетерминал в левой части можно интерпретировать как состояние перехода, нетерминал в правой части - как текущее состояние, терминал - как распознаваемый символ. На самом же деле регулярные грамматики являются не слишком удобным средством для непосредственного описания лексики. На практике используется более свободная форма описания, называемая регулярными выражениями. РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ - это язые описания лексики, формально сводимый к регулярным грамматикам. Рассмотрим вкратце его синтаксис.

Простейшие регулярные выражения – [<символ>], где квадратные скобки являются метасимволом. Простейшие регулярные выражения также могут быть заданы в виде [<список символов>]. Пример [+ - * /]. В данном примере разрешается употребление одного символа из этого списка. Еще один вид описания простейших регулярных выражений – с помощью диапазона – [<символ> – <символ>] возможность использования одного символа из этого диапазона. В этом описании квадратные скобки и знак диапазона (дефис) являются метасимволами. Стоит заметить, что программа лексического анализа автоматически распознает диапазон символов и отличает его от знака минус. Диапазон будет интерпретирован правильно, только если начальный и конечный символы являются границами этого диапазона. Т.е. это либо буквы, либо цифры и начальный символ диапазона является меньшим , чем конечный.

Далее приведен список операторов, которые можно использовать при конструировании языка описания.

1. Оператор ?. Порядок использования (регулярное выражение)?. Означает в точности одно или ни одного появления регулярного выражения в тексте. Пример: [+ -]? – либо плюс, либо минус, либо вообще ничего.
2. Оператор *. Порядок использования (регулярное выражение)*. Означает ни одного, одно или несколько появлений регулярного выражения в тексте. Пример: [+ -]? [0-9]*.
3. Оператор +. Порядок использования (регулярное выражение)+. Означает одно или более появлений регулярного выражения в тексте. Пример: [A - Z a - z]+.
4. Оператор ; . Завершение описания класса.
5. Оператор ^. Порядок использования (регулярное выражение)^. Означает в точности любой символ, кроме регулярного выражения. Пример [A]+[Z]^[C]*.
6. Оператор |. Порядок использования (регулярное выражение) | (регулярное выражение). Операция “ИЛИ” над регулярными выражениями.

Оператор “ ”. Порядок использования “<слово>”. <Слово> может состоять из простейших регулярных выражений без использования диапазона. Обычно этот оператор служит для описания ключевых слов языка. В принципе, такую конструкцию можно описать и по-другому с помощью метасимвола квадратных скобок. Для этого необходимо каждый элемент слово описать с помощью простейшего регулярного выражения с использованием метасимвола. Например: “begin”. Равносильная конструкция [b][e][g][i][n]. Использование этого оператора позволяет сделать описание более простым.

С помощью регулярных выражений и вспомогательных операторов, описанных выше можно сконструировать описание любого языка, с использованием букв английского алфавита. Ограничением конструируемых языков является использование квадратных скобок и букв русского алфавита. В заключение приведем пример описания лексики, включающей в себя идентификаторы, десятичные и шестнадцатеричные константы, ключевые слова, знаки операций и операторов.

.

```
Ident = [A-Z a-z _] [A-Z a-z 0-9 _]*;
Const_Dec = [0-9]+;
Const_Hex = [0] [x] [0-9 A-F a-f]+;
Operation = [+-*/];
Operator = [=] ;
Const=Const_Dec|Const_Hex;
Un_oper = "++" | "--";
```

```
  Word = "switch" | "case" | "otherwise" | "main";
ENDOFOPER=[;];
```

Не вдаваясь в подробности работы системы автоматизированного построения лексических анализаторов, отметим, что она производит формальное преобразование лексики в следующей последовательности представлений:

регулярные выражения;

регулярная грамматика;

недетерминированный КА (то есть автомат, имеющий несколько переходов из одного состояния по одному и тому же входному символу, либо имеющий переход по пустому символу);

детерминированный (обычный) КА.

Методы и алгоритмы синтаксического анализа

Синтаксический разбор - построение дерева синтаксического разбора можно производить как сверху вниз - от начального нетерминала к предложению языка, так и снизу вверх - от предложения к начальному символу:

нисходящий синтаксический разбор заключается в поиске замены очередного нетерминального символа в выводимой цепочке на правую часть соответствующего правила. При этом алгоритм может руководствоваться только “незакрытой”, то есть нераспознанной частью предложения. Обычно для этого достаточно одного терминального символа.

при восходящем синтаксическом разборе в предложении (или в промежуточной цепочке) ищется правая часть правила, которое необходимо “свернуть” к нетерминалу левой части. Решение принимается на основе анализа соседних терминальных символов в анализируемой цепочке.

По своей природе алгоритмы синтаксического разбора бывают детерминированные (сразу же строящие правильное дерево) и недетерминированные, то есть допускающие возврат на некоторое число шагов назад.

3.2. Нисходящий разбор с возвратами

В процессе нисходящего разбора происходит замена каждого нетерминала, который встречается в выводимой цепочке, на правую часть правила, в которой он находится в левой части. Очевидно, что самый простой алгоритм построения цепочки вывода заключается в полном переборе всех возможных цепочек. Тогда мы получаем классическую задачу поиска с полным перебором вариантов, которая решается с использованием рекурсивных функций. Соответствующая программная заготовка имеется в файле sindown.cpp. Приведенная в ней грамматика определяет 4 действия арифметики и круглые скобки. Для понимания сущности рекурсивного алгоритма нужно сосредоточиться на выполнении одного шага рекурсии, не пытаясь заглянуть “вглубь”:

грамматика задана в виде массива указателей на строки, содержащие правила. Входная цепочка - также строка

```
char      *GR[]={”E:TM”,”M:E”,”M:+E”,”M:”,”T:FG”,“
G:*T”,”G:/T”,”G:”,”F:a”,”F:(E)”,NULL};
char str[80];
```

- рекурсивная функция при каждом своем вызове получает на вход очередной нетерминал и начало терминальной цепочки, которая должна быть выведена из него. Начало цепочки определяется номером очередного символа во входной строке str.

```
int Step(char sym, int s)
```

- результатом работы текущего, а также всех вложенных (внутренних) вызовов является “покрытие” некоторой левой части цепочки деревом правил, выводимых из данного нетерминала. При этом функция выводит целое число, номер символа в строке, следующего за “покрытой” цепочкой. Если вывод неудачен, то возвращается значение -1.

```
E
T             M
-----    -------
F    G   (пусто)
     |   ----
```

```
   a   /  T
   |   |  ----
   |   |  F  G
   |   |  |  -------
   |   |  a  (пусто)
   |   |  |
 ( a   /  a )
  |_вход   |_____выход
```

рекурсивная функция просматривает набор правил выбирает их них те, которые имеют в левой части входной нетерминал. Для каждого из них она просматривает его правую часть. При этом вводится новая переменная - k -индекс “незакрытой” части цепочки в процессе применения текущего правила

```
int Step(char sym, int s)
{ int i,j,k;
for (i=0; GR[i]!=NULL; i++)
if (GR[i][0]==sym)
{ for (j=2,k=s; GR[i][j]!=0; j++)
```

если текущий символ правила является терминальным, то необходимо сравнить его с текущим “незакрытым” символом во входной строке. Если они одинаковы, то “закрыть его” - перейти к следующему и в правиле, и в цепочке. Если нет, отказаться от этого правил и перейти к следующему.

```
if (isNT(GR[i][j])!=-1)
    {. . .}
   else
   if (GR[i][j]==str[k]) k++;
else break;
```

для нетерминального символа в правиле нужно выполнить тот же самый алгоритм - попробовать произвести из него вывод и закрыть часть цепочки. Это делается рекурсивным вызовом той же функции, но уже для новой “незакрытой” части цепочки и нового нетерминала. Если полученный результат будет свидетельствовать об удачном выводе, то следует в текущем правиле также пропустить “закрытую” данным нетерминалом часть цепочки. Если вывод неудачен - перейти к следующему правилу.

```
if (isNT(GR[i][j])!=-1)
{ int l=Step(GR[i][j],k);
if (l==-1) break;
cout << "*" << GR[i] << "\n"; k=l;
       }
```

обнаружение конца правила при попытке закрыть им часть цепочки свидетельствует об удачном его применении. Тогда функция возвращает индекс первого “незакрытого” символа, оставшегося после применения этого и вложенных в него правил

```
for (i=0; GR[i]!=NULL; i++)
   if (GR[i][0]==sym)
         { for (j=2,k=s; GR[i][j]!=0; j++)
                 { . . . }
         if (GR[i][j]==0)
               {
               cout<<GR[i]<<"__"<<&str[k]<<"\n";  return k;
               }
         } return -1;
```

Естественным ограничением для полного перебора вариантов является исключение прямой или косвенной левосторонней рекурсии. То есть в грамматике принципиально недопустимы правила вида

```
E ::= E + T   или  сочетания правил
E ::= X . . .
X ::= E . . .
```

которые приводят к “зацикливанию” алгоритма. В данном случае такое правило применяется само к себе бесконечное число раз.

3.3. Рекурсивный спуск

Рассмотрим еще один “естественный” алгоритм нисходящего СА. Пусть имеется некоторый нетерминальный символ, из которого выводится предложение языка - цепочка терминальных символов. При нисходящем разборе этот нетерминал на первом шаге необходимо заменить на правую часть одного из правил, в котором он присутствует в левой части. Рассмотрим пример:

```
F ::= i | i[E] | c | *E | &E | i.i | (E)
```

Это классический набор правил для определения базового элемента выражения F как переменной, элемента массива, константы, поля структуры, адресного выражения или объекта, адресуемого через указатель. Предположим, что в процессе нисходящего разбора некоторая часть терминальной цепочки (предложения языка) должна быть выведена из нетерминала F:

.

```
        F
        |
   aaa[jj+46]…        (…i[i+c]…)
```

Для удобства восприятия входная строка приведена и в синтаксических, и в лексических единицах. Тогда единственная проблема состоит в том, чтобы на очередном шаге построения дерева нетерминал F заменить на правую часть одного из перечисленных правил. Человек сделает это интуитивно, выбирая правило F::=i[E] - определение элемента массива

.

```
                    F
                    |
                i [ E   ]
                    |
  aaa[ jj+46 ]… (i[i+c])
```

после чего оставшуюся часть строки jj+46 требуется вывести из нетерминала E. Отсюда следует самый простой неформальный алгоритм синтаксического анализа. Для каждой группы правил с общим нетерминалом в левой части пишется процедура, которая по начальным символам терминальной цепочки в состоянии определить, правую часть какого правила следует применить, затем она параллельно просматривает терминальную цепочку и правую часть правила. Если очередные терминальные символы в цепочке и в правиле совпадают, то они оба пропускаются, если нет - это признак синтаксической ошибки. Если в правиле встречается нетерминальный символ, то необходимо вызвать аналогичную процедуру для обработки группы правил, в которой этот нетерминал встречается в левой части. Для понимания вышесказанного рассмотрим фрагмент программы:

```
char s[80];       // входная цепочка терминалов
int      i;       // текущий терминальный символ
void F()
{
switch(s[i])
      {
case '*': i++; E(); break;
```

```
case '&': i++; E(); break;
case 'c': i++; break;
case '(': i++; E();
            if (s[i]==')') i++; else error(); break;
case 'i': i++;
if (s[i]=='.')
{ i++; if (s[i]=='i') i++;
else error(); }
else if (s[i]=='[')
{ i++; E(); if (s[i]==']') i++;
else error(); }
else i++;
break;
      }}
```

индекс i является указателем на текущий нетерминальный символ во входной строке (цепочке). Каждая процедура, анализируя очередной символ, продвигает этот указатель вперед, если в выбранной правой части правила находится такой же символ, в противном случае информирует о синтаксической ошибке. Например, в правой части правила F::=i[E] после символа "[" и нетерминала E следует символ "]". Последний и проверяется в процедуре анализа после вызова процедуры анализа нетерминала E. Если он действительно присутствует в строке как текущий символ, то он пропускается

```
if (s[i])==']') i++; else error();
```

если в правой части правила имеется нетерминал, то в тот момент, когда в просматриваемом фрагменте строки "наступает очередь" этого нетерминала, вызывается процедура, которая должна производить этот анализ для группы правил с этим нетерминалом в левой части. Например, после анализа цепочки терминалов "i[" выясняется, что речь идет о правиле F::=i[E], тогда для выполнения анализа выражения в скобках необходимо просто вызвать процедуру E().

выбор одной или другой правой части может быть произведен только на основе анализа одного или нескольких текущих терминальных символов. Так выбор одного из правил F::=i | i.i | i[E] производится по первым двум символам (одного здесь не достаточно).

Коль скоро последовательность вызова процедур обработки нетерминальных символов определяется последовательностью применения соответствующих правил при построении дерева, а она обычно является рекурсивной, то в системе процедур имеется скрытая (косвенная) рекурсия. Например, процедура обработки нетерминала F для правила F::=i[E] вызывает процедуру обработки нетерминала E, который в свою очередь может вызвать процедуру обработки нетерминала F. Поэтому данный метод получил название РЕКУРСИВНОГО СПУСКА.

Достоинствами метода является естественность написания процедур синтаксического анализа, последовательность вызова которых в программе совпадает с последовательностью нисходящего вывода. Данный метод исключительно удобен про построении интерпретаторов, когда в процессе синтаксического анализа производится анализ семантики выражения и его интерпретация. Предположим, что мы разрабатываем интепретатор выражений с переменными целого типа с набором перечисленных правил, тогда приведенный фрагмент программы легко преобразовать в интерпретатор.

```
int F()             // результат вычисления F
{
int v,n;            // результат промежуточных вычислений
switch(s[i])
      {
case '(': i++; v=E();
            if (s[i]==')') i++; else error(); break;
case 'c': i++; v = значение_константы_c; break;
case 'i': i++;
if (s[i]=='.')
{ i++; if (s[i]=='i')
            { v=значение_поля_s[i]_переменной_s[i-2]; i++;}
else error(); }
else if (s[i]=='[')
{ i++; n=E(); if (s[i]==']')
```

```
        { v=значение_элемента_n_массива_s[i-2]; i++; }
else error(); }
else { v=значение_переменной_s[i];i++; }
break;
} return v; }
```

Существенным недостатком рекурсивного спуска является трудность анализа правил, правая часть которых начинается с нетерминального символа. В этом случае необходимо преобразование группы правил. Рассмотрим, как это происходит для правил, генерирующих цепочки переменной длины из повторяющихся символов.

T ::= T * F | T / F | F

Эти правила генерируют цепочки вида F * F / F * F / F с переменным количеством нетерминалов F и знаками операций "*,/ " между ними. Очевидно, процедура для нетерминала T должна распознавать циклическую последовательность таких символов и завершать распознавание, когда во входной цепочке ей встретится очередной символ, отличный от "*,/ ".

```
void T()
{ while (1)
      { F();
      if (!(s[i]=='*' || s[i]=='/')) break;
      i++;
      }
}
```

В заключение рассмотрим фрагмент программы синтаксического анализа методом рекурсивного спуска (sindown1.cpp), которая включает процедуры, соответствующие синтаксису арифметических действий, а также лексики десятичных констант и переменных, имя которых состоит из одного символа. В процесс синтаксического анализа включен простой интерпретатор для переменных целого типа, в котором параллельно с анализом производится и выполнение действий, соответствующих операциям.

```
int      VAR[26];      // Массив значений переменных A..Z
char S[100];           // Входная строка
int      i;            // Текущий символ строки
void      error(int n, int i); // Вывод сообщения об ошибке
                               // с номером n и символом i
extern      int      E();
int      F()                   // F ::= a..z | целое | (E)
{ int      x;
if (isdigit(S[i]))
      { x=0;                  // Накопление целой константы
      do      { x=x * 10 + S[i] - '0'; i++; }
while (isdigit(S[i])); return(x); }
else
if (isalpha(S[i]))            // Чтение значения переменной
{ x=VAR[toupper(S[i])-'A']; i++; return(x); }
else
if (S[i]=='(')                // Анализ выражения в скобках
{ i++; x=E();
if (S[i]==')') { i++; return(x); }}
else { error(2,i); return(0); }}
int      T()                  // T ::= T * F | T / F | F
{ int x; x=F();
while (1)
switch      (S[i]) {
case      '*': i++; x=x*F(); break;
case      '/': i++; x=x/F(); break;
default:      return(x); }}

int      E()                 // E ::= E + T | E - T | T
{ int x; x=T();
```

```
while (1)
switch      (S[i]) {
case      '+': i++; x=x+T(); break;
case      '-': i++; x=x-T(); break;
default:       return(x); }}

// Правила объединяют операции сравнения и присваивания
int      P()      // P ::= E=>a…z | E=E | E>E | E<E | E#E
{ int x;       x=E();
switch      (S[i]) {
case '=':   i++;
            if (S[i]=='>')
                { i++;
                if (isalpha(S[i]))
                    { VAR[toupper(S[i])-'A'] = x;
                    return(x); }
                else      { error(4,i); return(0); }
            }
            else return( x== E() );
case '#':   i++; return( x!= E() );
case '<':   i++; return( x < E() );
case '>':   i++; return( x > E() );
case '\0':  return(x);
default:    error(3,i); return(0); }}
```

3.4. Магазинные автоматы

Рассмотренный выше метод рекурсивного спуска является в значительной степени неформальным и, соответственно, не может быть непосредственно применен к заданной грамматике. Все остальные методы СА ориентированы именно на формальный подход к построению транслятора. Имея грамматику определенного вида, можно формально построить анализатор.

Известный нам формализм - конечные автоматы (КА), используемый на этапе лексического анализа, не подходит для синтаксиса. Прежде всего, по той причине, что грамматики дают возможность построения вложенных (рекурсивных) цепочек, анализ которых для КА оказывается непосильным. Интуитивно мы можем догадываться, что если речь идет о вложенности и рекурсии, то обязательно должен присутствовать стек. И действительно, если обычный КА дополнить стеком, то полученная формальная система может служить базой для создания синтаксического анализатора. Такой КА со стеком называется МАГАЗИННЫМ АВТОМАТОМ (МА). Полуформально его можно определить так:

входная строка I, текущий символ в строке (I)

магазин (стек) M, символ в вершине стека (M)

операция записи в стек заданной строки push(M,"…"). Строка помещается в стек, начиная с последнего символа, то есть первый символ оказывается в вершине стека;

операция выталкивания символа из стека pop(M);

операция проверки "стек пуст" M=0;

операция проверки наличия очередного символа в строке "строка пуста" I=0;

операция перехода к следующему символу в строке next(I) ;

S = {Si}, множество состояний КА;

описание поведения МА заключается в том, что для каждой комбинации (Si, (I), (M)) - состояния КА, входного символа и символа в вершине стека задается новое состояние перехода Sj и одна или несколько перечисленных выше операций.

Далее будут рассмотрены несколько видов грамматик, для каждой из которых будет определена методика (алгоритм) построения МА, который будет распознавать предложения этой грамматики. Речь будет идти о нисходящем разборе, грамматики будут рассматриваться в порядке возрастания их сложности.

Грамматики класса S (S-грамматики)

Грамматика строится по очень простому принципу:

правая часть каждого правила должна начинаться с терминального символа;

для двух любых правил, имеющих одинаковые левые части, начальные терминальные символы должны быть различны;

не допускаются правила с пустой правой частью.

Следующая грамматика является S-грамматикой:

.

```
T = {a,b}, N = {S,B}, N0 = S
S ::= aA
S ::= bSb
A ::= aA
A ::= b
```

Данная грамматика порождает цепочки вида **aa…aaab** и **b..baaa..aaabb..b**.

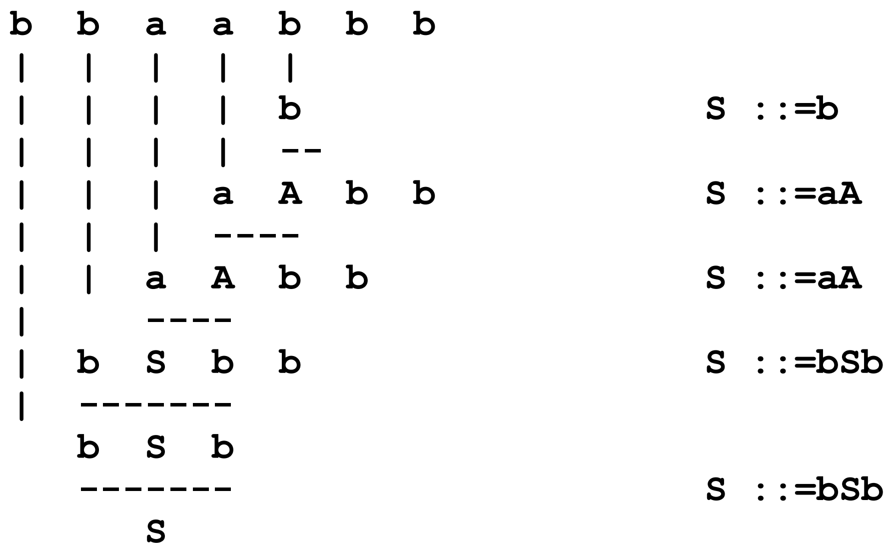
Для начала рассмотрим, как будет происходит нисходящий СА для такой грамматики:

дерево разбора строится сверху вниз;

на каждом шаге в полученной цепочке единственный нетерминальный символ должен быть заменен на правую часть одного из правил, в котором он присутствует в левой части;

принцип замены вытекает из самого определения S-грамматики. Для замены выбирается такое правило, у которого первый терминальный символ совпадает с очередным “незакрытым” символом в входной цепочке. Сказанное поясним примером:

.



Начальный нетерминальный символ грамматики S. Поэтому он и является начальной цепочкой нисходящего вывода. Он может быть заменен на правую часть одного из двух правил **S ::=aA** или **S::=bSb**. Какое будет выбрано, определяется первым символом в распознаваемой цепочке. Поскольку это - **b**, то выбирается второе правило и цепочка примет вид **bSb**. Заметим, что первые символы в распознаваемой и выводимой цепочке совпадают, то есть перекрываются. Тогда второй символ распознаваемой цепочке оказывается “незакрытым” и соответствующим нетерминалу **S** в выводимой цепочке. Для этой пары такая процедура должна быть повторена.

Если использовать МА, то естественным местом для размещения выводимой цепочки является стек (магазин). Тогда правила функционирования МА (пока еще не алгоритм, а перечень действий), можно определить так:

(I)=(M) ==> pop(M), next(I), если текущий символ строки совпадает с символом в вершине стека, то исключить его из стека и продвинуться в строке к следующему (совпадающие символы в начале выводимой и распознаваемой цепочек взаимно уничтожаются);

(M) # (I) && (M) in T =>> Error, если текущий символ строки и символ в вершине стека не совпадают, и при этом символ в стеке является терминальным, то это синтаксическая ошибка

(M) in N, если в вершине стека находится нетерминальный символ, то ищется правило, которое в левой части имеет этот символ, а его правая часть начинается с очередного символа в распознаваемой строке, то есть по нашим обозначениям имеет вид **(M)::=(I)xyz** , тогда в стеке левая часть правила заменяется на правую, то есть по нашим обозначениям выполняются действия **pop(M), push(M,"(I)xyz")**;

распознавание выполнено успешно, когда и стек и строка одновременно становятся пустыми, то есть **I=0 && S=0**, в противном случае - синтаксическая ошибка;

в исходном состоянии в стек записывается начальный нетерминал, **push(N0)**

Перечисленные правила могут быть использованы для представления поведения конкретного КА в виде таблицы. Заметим, что такой МА имеет единственное состояние. Таблица определяет реакцию МА на любую комбинацию символов в стеке и строке. Дополнив множество терминальных символов символом конца строки (цепочки) "#", получим полную картину поведения МА. Для нашего примера она будет иметь вид:

.

(M)\(I)	a	b	#
a	pop(M),next(I)	Error	Error
b	Error	pop(M),next(I)	Error
A	pop(M),push(aA)	pop(M),push(b)	Error
S	pop(M),push(aA)	pop(M),push(bSb)	Error
#	Error	Error	Success

Алгоритм нисходящего разбора для S-грамматик с произвольным набором правил приведен в sgram2.cpp. Здесь рассмотрим его фрагменты.

```
char      s[100]; char      m[100]; int      i,j;

void      push(char *str) // поместить в стек "хвостом вперед"
{ char *q=str;
while (*q !='\0') q++;
q--; while (q >=str) { m[j++] = *q--; }
m[j]='\0'; }

char      *GR[]={ "S:aA", "S:bSb", "A:aA", "A:b", NULL };
int sintax()
{
int      k; char      *q;
i=0; j=1; m[0]='S';             // Исходное состояние МА
while (1)                       // Завершение разбора
{ if ((s[i]=='\0') && (j==0)) break;
      if ((s[i]=='\0') || (j==0))      return 1;
      if ((s[i]==m[j-1]))       // Совпадение символов
            { i++; j--; m[j]='\0'; }
else
            {                   // Поиск правила (M)::=(I)xyz
            for (k=0; GR[k]!=NULL; k++)
                  {
                  q=GR[k];
                  if ((*q==m[j-1]) && (*(q+2)==s[i])) break;
                  }             //Нет правила
            if (GR[k]==NULL) break;
            else                  //Запись в стек "хвостом вперед"
                  { j--; printf("Правило: %s\n",q);
                  push(q+2);}
            }
printf("Стек:%s Строка:%s \n",m, s+i);
return 0;}}
```

Грамматики класса Q (Q-грамматики)

Данная грамматика отличается от S-грамматики наличие дополнительных аннулирующих правил вида:

.

A ::= ε, где ? - пустая строка

Такое правило будем так и называть q-правилом. Приведенную выше S-грамматику можно превратить в Q-грамматику.

```
T = {a,b}, N = {S,B}, N0 = S
S ::= aA
S ::= bSb
A ::= aA
A ::= ε
```

Данная грамматика порождает цепочки вида **aa…aaa** и **b..baaa..aaab..b**. Нетрудно заметить, что аннулирующее правило используется для порождения цепочек, которые могут включать повторяющиеся произвольное число раз фрагменты без явного признака их окончания. В данном примере это имеет отношение к последовательности символов **a..a** , порождаемых двумя последними правилами.

Для Q-грамматик может быть использован тот же метод разбора с применением МА, однако сам принцип необходимо расширить для аннулирующих правил. Заметим, что при замене нетерминального символа правой частью одного из правил мы руководствовались первым терминальным символом правил, которой можно назвать ВЫБИРАЮЩИМ. Тогда для обычного S-правила условием замены является совпадение текущего “незакрытого” символа в распознаваемой цепочке с ВЫБИРАЮЩИМ символом правила. Для Q-правила аналогично можно ввести понятие выбирающего символа. Рассмотрим, на каком основании тот или иной символ можно назвать выбирающим. Пусть в процессе вывода произвольной цепочки из начального нетерминала **S** мы получаем цепочку с последовательностью символов **Ab** в некотором контексте. Если при этом **A** является левой частью аннулирующего правила, то мы имеем все основания его применить. Тогда символ **b**, следующий за **A** в любом другом контексте синтаксического разбора является определяет возможность применения такого правила. Сказанное поясним примером.

```
S >> bSb >> baAb >> ba(A::= ε)b >> baεb >> bab
```

Таким образом, множество выбирающих символов для применения аннулирующего правила **A::=ε** состоит из множества всех символов, которые могут следовать за **A** в любой промежуточной цепочке, выводимой из **S**.

.

```
ВЫБ(A::=ε) = СЛЕД(A)
```

Построить множество символов, следующих за данным нетерминалом можно для простых грамматик и неформально, рассматривая возможные варианты порождения цепочек. В рассматриваемой грамматике приведенный выше вариант является одним из двух случаев. Второй случай предполагает, что в распознавании должен участвовать символ конца цепочки.

```
S# >> aA# >> … >> a…aA# >> a…a(A::=ε)# >> a…a#
```

Однако применить принцип работы МА для S-грамматик к Q-грамматикам можно только в том случае, если множества выбирающих символов для аннулирующих правил и для обычных Q-правил не будут пересекаться. Иначе в работе МА возникает неоднозначность, которая исключает гарантированное распознавание. Таблица выбирающих символов для нашей грамматики будет выглядеть так:

.

Правило   Выбирающие символы

```
S ::= aA    a
S ::= bSb   b
A ::= aA    a
A ::= ε     СЛЕД(A)={b,#}
```

Поскольку аннулирующее правило имеет пустую правую часть, то его применение в алгоритме распознавания в МА сопровождается выталкиванием из стека текущего нетерминального символа. Таблица действий МА с учетом Q-правила будет иметь вид:

.

(M)\(I)	a	b	#
a	pop(M),next(I)	Error	Error
b	Error	pop(M),next(I)	Error
A	pop(M),push(aA)	pop(M)	pop(M)
S	pop(M),push(aA)	pop(M),push(bSb)	Error
#	Error	Error	Success

Алгоритм нисходящего разбора для S-грамматик с произвольным набором правил приведен в qgram1.cpp. Здесь рассмотрим только его отличия от предыдущего. При поиске необходимого правила проверяется, не является ли оно аннулирующим (пустым). В этом случае производится сравнение текущего символа строки (распознаваемой цепочки) с выбирающими символами из строковой константы CH[k]:

```
char      *GR[]={ "S:aA", "S:bSb", "A:aA", "A:", NULL };
char      CH[]={ NULL,NULL,NULL,"b"};
for (k=0; GR[k]!=NULL; k++)         // Поиск правила
      { q=GR[k];if(*q==m[j-1])      // Левая часть ???
            { if (*(q+2)!='\0')     // S-правило
                  { if (*(q+2)==s[i]) break; }
            else                    // Q-правило
                  {                 // Выбирающий символ
                  for (f=CH[k]; *f !='\0'; f++)
                        { if (s[i]==*f) break;
                        if ((s[i]=='\0') && (*f=='*')) break;
                        }
                  if (*f != '\0') break;
                  }}}
```

Грамматики класса LL(1) (LL(1)-грамматики)

Следующий тип формальных грамматик уже может быть использован для практического применения. В дополнение к Q-грамматикам они могут содержать правила, правая часть которых начинается с нетерминального символа. Общий принцип функционирования МА в этом случае остается прежним. Единственное, что здесь необходимо, это определить выбирающие символы, по которым производится замена. Здесь нужно последовать уже опробованному методу. Если из правила **A::=U…**, начинающегося с нетерминала, можно построить цепочку **A** >> **U...** >> **...** >> **x...**, которая начинается с терминального символа **x**, то такой символ можно считать выбирающим, поскольку он дает основания для применения правила **A::=U...**. Таким образом, множество выбирающих символов для правила, начинающегося с нетерминального символа, состоит из множества символов, которые являются первыми во всех возможных цепочках, выводимых из этого правила.

.

ВЫБ(A::=U…) = ПЕРВ(A::=U…)

Естественное требование к выбирающим символам остается прежним: для работоспособности метода необходимо, чтобы множества выбирающих символов для правил с одинаковой левой частью не пересекались. В качестве примера рассмотрим LL(1)-грамматику для четырех действий арифметики и скобок.

.

Правило	Способ выбора	Символы
E ::=TM	ПЕРВ(T)	a,(
M ::=-E	-	-
M ::=+E	+	+
M ::=e	СЛЕД(M)	),#
T ::=FG	ПЕРВ(F)	a,(
G ::=*T	*	*
G ::=/T	/	/
G ::=e	СЛЕД(G)	),+,-,#
F ::=a	A	a
F ::=(E)	(	(

Множества выбирающих символов для правил, начинающихся с нетерминала и аннулирующих правил строятся в такой несложной грамматике путем простого перебора возможных цепочек синтаксического вывода:

.

Правило E ::= TM, способ построения ПЕРВ(T):
1. T >> FG >> a, выбирающий символ a
2. T >> FG >> (E), выбирающий символ (

.

Правило G ::= ε, способ построения СЛЕД(G):
1. E >> TM >> FGM >> FG+E, символ +
2. E >> TM >> FGM >> FG-E, символ -
3. F >> (E) >> ™ >> (FGM) >> (FG), символ)
4. E# >> TM# >> FGM# >> FG#, символ #

Программа нисходящего разбора для LL(1)-грамматик с перечисленными выше правилами имеется в lgram1.cpp. Принципиально она не отличается от программы для Q-грамматик. Имеется массив указателей на строки-правила GR и массив указателей на строки выбирающих символов CH. Для S-правила выбирающие символы отсутствуют (NULL-указатель), поэтому производится сравнение с первым символом правой части правила, для остальных - последовательно проверяются все выбирающие символы в соответствующей строке.

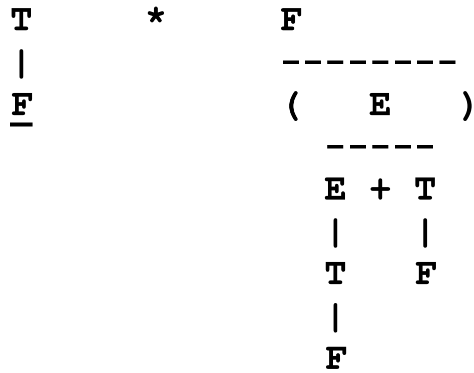
```
char       *GR[]=
{"E:TM","M:-E","M:+E","M:","T:FG","G:*T","G:/T","G:","F:a","F:(E)",NULL };
char       *CH[]=
{ "a(",NULL,NULL,")#","a(",NULL,NULL,"+-)#",NULL,NULL};
```

3.5. Восходящие методы анализа. Свертка-перенос

Восходящие методы синтаксического анализа состоят в том, что в цепочке (промежуточной или терминальной) ищется правая часть очередного правила, которое должно быть заменено своим нетерминалом. Очевидно, что не любая правая часть правила может быть свернута в любое время. Если посмотреть на дерево синтаксического разбора, то нетрудно заметить, что свертку необходимо производить регулярно, путем обхода дерева слева направо.

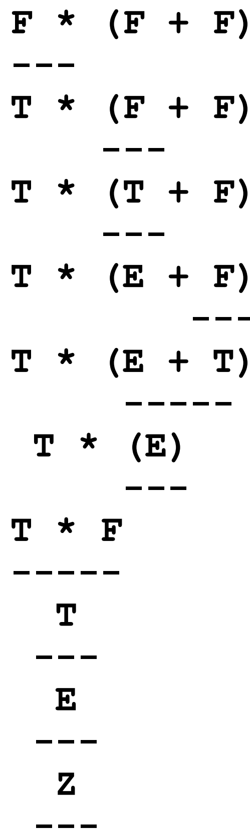
.

```
Z
|
E
|
T
------------
```



В данном дереве необходимо выполнить свертку по самому левому правилу T::=F в цепочке. Правая часть такого правила называется ОСНОВОЙ. Таким образом, основа - это полная правая часть первого правила при просмотре цепочки слева направо. Найденная в цепочке основа должна быть заменена на нетерминал из левой части соответствующего правила. Для более полного понимания принципа приведем последовательность основ цепочек при выполнении такого процесса свертки.

.



Способ нахождения основы цепочки реализован в методе восходящего разбора "свертка-перенос":

В процессе анализа используется магазинный автомат (КА со стеком);

На каждом шаге работы автомата сравнивается пара символов - текущий символ входной цепочки (I) и текущий символ в стеке (M). По результату сравнения возможны два действия:

Перенос терминального символа из строки в стек: PUSH((I), NEXT(I) - в соответствии с принятыми для МА обозначениями.

Свертка правой части правила, которое в данный момент находится в стеке. Возможны два принципиально различных варианта свертки: правило единственное, либо возможен выбор одного из нескольких.

Для формального задания поведения МА необходимо задать таблицу его действий на полный абор сочетаний - входной терминальный символ и любой символ в стеке. В клетке таблицы могут быть указаны следующие действия:

Сообщение о синтаксической ошибке;

Сообщение об успешном завершении разбора;

Перенос символа из входной цепочки в стек;

Вызов процедуры для анализа правой части правила в стеке и замены ее на нетерминал.

Для грамматики арифметических выражений со скобками приведем пример такой таблицы.

.

	+-	*/	(	)	F	#
+-	-	-	П	-	П	-
*/	-	-	П	-	П	-
(	-	-	П	-	П	-
)	C3	C3	C3	C3	C3	C3
F	C1	C1	-	C1	-	C1
T	C2	П	-	C2	-	C2
E	П	-	-	П	-	C4
#	-	-	П	-	П	+

.

 # - начало/конец цепочки
Процедуры свертки правил:
C1 -> T::= T*F | T/F | F
C2 -> E::= E+T | E-T | T
C3 -> F::=(E)
C4 -> POP(M)

Как видно, для простых грамматик построение такой таблицы не выходит за рамки здравого смысла. Для каждой пары символов производится анализ ситуации (контекста цепочки), в которой она может встретиться, на основании чего и заполняется клетка таблицы. Если ситуация свидетельствует, что не вся правая часть правила в стеке, то необходимо сделать перенос, если вся - анализировать возможные правила. Заготовка программы, выполняющей синтаксический анализ для данной грамматики по методу "свертка-перенос" находится в файле lowtohigh.cpp.

Грамматика языка задана массивом указателей на строки - правила. Строками заданы множества терминальных символов грамматики (TS) и всех символов грамматики (NTS). Определены также стек и входная строка.

```
char      *GR[]={"Z:E","E:E+T","E:E-T","E:T","T:T*F","
T:T/F","T:F","F:i","F:c","F:(E)",NULL };
char      TS[]= "+-*/()ic#" ;
char      NTS[]= "+-*/()ic#FTEZ" ;
char      s[100];      int      ns;
char      m[100];      int      sp;
```

Двумерный массив определяет действия МА на каждую пару сочетаний символа в стеке и во входной цепочке: 0 - ошибка, 1-перенос, 2-свертка, 3-успешный разбор.

```
int      TBL[14][10]={
/*            + - * / ( ) i c # err */
/* + */      { 0,0,0,0,1,0,1,1,0,0 },
/* - */      { 0,0,0,0,1,0,1,1,0,0 },
/* * */      { 0,0,0,0,1,0,1,1,0,0 },
```

```
/* / */      { 0,0,0,0,1,0,1,1,0,0 },
/* ( */      { 0,0,0,0,1,0,1,1,0,0 },
/* ) */      { 2,2,2,2,0,2,0,0,2,0 },
/* i */      { 2,2,2,2,0,2,0,0,2,0 },
/* c */      { 2,2,2,2,0,2,0,0,2,0 },
/* # */      { 0,0,0,0,1,0,1,1,0,0 },
/* F */      { 2,2,2,2,0,2,0,0,2,0 },
/* T */      { 2,2,1,1,0,2,0,0,2,0 },
/* E */      { 1,1,0,0,0,1,0,0,2,0 },
/* Z */      { 0,0,0,0,0,0,0,0,3,0 },
/*err*/      { 0,0,0,0,0,0,0,0,0,0 },
      };
```

Вспомогательная функция STS определяет номер символа в строке терминальных символов или всех символов грамматики

```
int       STS(char c,char T[]) {. . .}
```

Функция main буквально повторяет приведенный выше алгоритм разбора. Для каждой пары символов в строке и в вершине стека вызывает действие, определенное для них в таблице.

```
void      main()
{ int      i,j,stop;
printf("Строка:"); gets(s); strcat(s,"#");
ns=0; sp=0; m[0]='#'; m[1]='\0';
nts=strlen(TS); stop=0;
while(!stop)
      { i=TBL[ STS(m[sp],NTS) ][ STS(s[ns],TS) ];
      switch (i) {
case  1:  sp++; m[sp]=s[ns];      m[sp+1]='\0';
          ns++; break;            // Перенос
case  2:  if (TESTGR()==0)       // Свертка
          stop++; break;
case  3:  stop++; break;
default:  stop++; break;
          }}}
```

Функция TESTGR реализует самый общий принцип свертки: она просматривает список правил и ищет первое из них, которое содержит в вершине стека свою правую часть, после чего заменяет его на левую. То есть последовательность расположения правил с одинаковой правой частью в списке может влиять на правильность синтаксического разбора. А именно: если есть два правило с одинаковым окончанием правой части, то более длинное должно находиться в начале списка.

```
int       TESTGR()
{ char *ss,*q; int      n,j;
for (n=0; GR[n] !=NULL; n++)       // Поиск правила
      { for (q=ss=GR[n]+2; *ss !='\0'; ss++);
      ss--;
      for(j=sp; (ss>=q) && (m[j]==*ss); j--,ss--);
      if (q== ss+1)                  // Свертка
            { j++; sp=j; m[sp]=*GR[n];
            m[sp+1]='\0'; return(1); }
      }
return (0);}
```

4. Семантический анализ

Семантический анализ является наименее формализуемой частью процесса трансляции, поскольку касается "смысла" тех конструкций языка, которые анализируются на этапах лексического и синтаксического анализа. Если же подходить к этому процессу с точки зрения "здравого смысла", то транслятор должен построить в процессе трансляции в памяти некоторую структуру данных, в которой зафиксировать все смысловые (семантические) взаимоотношения между объектами программы. Обычно объектами программы являются типы, переменные и функции (процедуры), которые обозначаются именами (идентификаторами). Для них уже на этапе

лексического анализа может быть составлена таблица, которая на этапе семантического анализа дополняется данными о семантике объектов. Отсюда вырисовывается общая схема взаимодействия фаз транслятора:

- Обычно транслятор создает несколько семантических таблиц, по видам используемых в языке объектов, поскольку содержание их может быть в принципе различным. Например, в Си-компиляторе имеет смысл заводить таблицы типов, локальных и глобальных переменных, функций. Таблицы эти заполняются семантическими процедурами при свертывании соответствующих правил. Например, при свертывании определения переменной ее имя должно заноситься в таблицу локальных или глобальных имен;

- на этапе лексического анализа при обнаружении очередного идентификатора его значение (строка символов, имя) либо непосредственно сопровождает распознанную лексему, либо помещается в таблицу со ссылкой на нее в той же лексеме. В обоих случаях значение лексемы (имя) сопровождает ее на этапе синтаксического анализа, где лексема выступает как терминальный символ и передается далее фазе семантического анализа.

- в процессе синтаксического анализа, при "свертывании" правила, в котором в качестве терминалов фигурируют имена, вызывается семантическая процедура, соответствующая этому правилу. Она должна получить ссылки на эти имена из соответствующих лексем;

- семантическая процедура по ссылкам на имена определяет, является ли операция, заданная правилом, допустимой для семантики этих имен. Кроме того, семантическая процедура может создать новые записи в таблицах, изменить их содержание, произвести поиск и т.д.. Поскольку любое правило при свертывании приводится к нетерминалу, стоящему в левой части, а он, в свою очередь, включается в другие, вышележащие правила, то семантическая процедура должна поставить ему в соответствие также некоторую семантику (смысл). Это делается также путем возвращения ссылки на элемент таблицы.

- фаза генерации кода или интерпретации обычно выполняется в виде такой же процедуры, как семантический анализ, в тех же самых условиях - в момент "свертки" правой части правила.

- элементы семантических таблиц в свою очередь могут быть связаны с некоторыми динамическими структурами данных, которые отражают внутреннюю семантику хранимых объектов. Это могут быть списки, массивы, деревья. В целом образуется некоторая семантическая сеть, по структуре которой семантические процедуры и производят анализ тех или иных частей программы.

Пример: семантика данных для Си-компилятора

Рассмотрим, как можно представить семантику переменных и типов данных (ТД) для языка Си. Прежде всего, условимся, что все типы данных, явно или неявно определяемые в программе, будут размещаться в таблице (массиве) TYPES. Элементом этого массива является структура, которая содержит описание ТД (d_type). Компонентами этой структуры являются:

- name - имя ТД. Если этот ТД является базовым, то его имя инициализировано в таблице. Если это ТД определяется в спецификаторе typedef, то оно берется из определения. Кроме того, в контексте часто определяются ТД для переменных, а также абстрактные ТД, которые не имеют имени – для них имя содержит пустую строку;

- size - размерность памяти под ТД в байтах. Каждый ТД в Си имеет фиксированную размерность, которая используется для создания переменных такого типа;

- TYPE – идентификатор текущего ТД. Если ТД является базовым, то он идентифицируется значением BTD. Если это производный ТД, то он обычно представляет собой цепочку (или дерево) вложенных друг в друга ТД, Текущий ТД может быть указателем (PTR), массивом (ARR), структурой (STRU) или объединением (UNI) (функции здесь не рассматриваются).

- child – указатель на описание вложенного (составляющего) ТД. Для всех ТД, кроме структуры и объединения, имеется единственный составляющий ТД, на который ссылается указатель. Для структурированного типа указатель ссылается на массив описателей составляющих ТД;

- dim – количество элементов в составляющем ТД или его описании. Если текущий ТД – массив, то это количество его элементов, а child указывает на единственный описатель вложенного ТД. Если это структура или объединение, то dim определяет количество элементов структуры, а child указывает на массив описателей этих элементов;

- В следующем примере семантическая сеть для различных ТД задана с помощью инициализации, чтобы по ней можно было показать, каким образом определения различных ТД сохраняются в семантических таблицах. Реально же инициализируются только описания базовых ТД, остальные строятся динамически в процессе семантического анализа явных и контекстных определений типов.

```
#define       BTD      0
#define      PTR      1
#define      STRU      2
#define      UNI      3
#define      ARR      4
struct d_type
{
      char      name[20];       // Имя ТД
      int      size;      // Размерность памяти ТД в байтах
      int      dim;            // Количество элементов вложенного ТД
      int      TYPE;      // Идентификатор типа
      d_type *child;      // Вложенный ТД (один или несколько)
};
extern      d_type TYPES[100];

// Определение полей структурированного типа man
d_type XXX[]=
{'name',      20,  20,  ARR, &TYPES[0]},      // char name[20];
{'addr',       2,   0,  PTR, &TYPES[3]},      // char *addr;
{'class',      2,   2,  BTD, &TYPES[1]};      // int class;

d_type TYPES[100] =
// Определение БТД
{"char",     1,      0,      BTD,    NULL},
{"int",      2,      0,      BTD,    NULL},
{'long',     4,      0,      BTD,    NULL},
// Неявное определение ТД или абстрактный ТД char*
{"",         2,      0,      PTR,    &TYPES[0]},
// Явное определение ТД typedef char *PSTR
{"PSTR",     2,      0,      PTR     &TYPES[0]},
// Неявное определение ТД или абстрактный ТД int [20];
{"",        40,      20,     ARR,    &TYPES[1]},
// Определение структурированного типа man
{"man",     24,      3,      STRU,   &XXX};
```

Понятно, что программы, работающие даже с такой семантической сетью, будут достаточно сложны. В качестве примера приведем программу, которая подсчитывает для произвольного ТД его размерность памяти в байтах с учетом всех вложенных в него ТД. Поскольку структурированный ТД предполагает ветвление семантической сети, то такая программа будет в добавок ко всему и рекурсивной.

```
int      GetSize(d_type *p)
{
switch      (p->TYPE)
      {
// Размерность БТД фиксирована
case BTD: return p->size;
// Размерность указателя постоянна
case PTR: return 2;
// Размерность массива – произведение числа элементов
// на размерность вложенного ТД
case ARR: return dim * GetSize(p->child);
// Размерность структуры – сумма размерностей элементов
case STRU:
      int s,i;
      for (s=0,i=0; i<dim; i++)
            s+=GetSize(&(p>child[i]));
      return s;
// Размерность объединения – максимальная размерность элемента
case STRU:
      int s,i,k;
      for (s=0,i=0; i<dim; i++)
            { k=GetSize(&(p->child[i])); if (k>s) s=k; }
      return s;
```

```
}}
```

Содержание семантической таблицы для переменной естественным образом вытекает из ее основных свойств в языке и может включать в себя:

- имя переменной;
- указатель на описание типа в таблице типов;
- смещение (адрес), который получает эта переменная при трансляции в том сегменте данных, где она размещается компилятором;
- указатель на область памяти, где размещаются ее значение – для интерпретатора.

Анализ семантики переменных при таком подходе может выглядеть следующим образом:

- при синтаксическом анализе правил определений и объявлений переменных семантическими процедурами параллельно строится семантическая сеть и заполняется таблица типов, в описание переменной в таблице переменных включается указатель на ее тип;
- при синтаксическом анализе правил построения выражений для заданной переменной семантические процедуры параллельно проверяют соответствие текущей операции текущему типу данных в семантической сети. Результат операции также получает указатель на элемент семантической сети, таким образом он связывается со своим типом данных для следующей операции.

Понятие L-value

К семантическому анализу имеет отношение и понятие l-value, характеризующее некоторый "тонкий" семантический смысл выражения, которое касается способа формирования его значения. Если происходит анализ и свертка правила, соответствующего некоторой операции, то у транслятора существуют два способа формирования ее результата:

- в виде нового объекта-значения. В таком случае транслятором должна быть "заведена" его семантика в виде некоторого внутреннего объекта, создаваемого в процессе работы программы;
- в виде объекта-операнда или его части. В таком случае семантика этого объекта-результата включает в себя неявный указатель (ссылку) на "исходный" объект. Такой результат операции называется l-value, от слова LEFT, что означает, что данное выражение может стоять в левой части операции (оператора) присваивания.

Общая стратегия транслятора должна состоять в том, что он должен сохранять результат в виде l-value до тех пор, пока не встретится операция, в которой он не в состоянии это сделать. Тогда уже он может переходить к значениям - промежуточным объектам. Рассмотрим ряд примеров для Си-компилятора, проиллюстрировав внутреннее представление выражения через l-value средствами того же Си.

.

```
Выражение      Код компилятора      Примечание
B[i]           &B[i]                l-value
B[i].money     &(B[i].name)         l-value
B[i].money+5   x=(*&B[i].money)+5   транслятор поддерживает
                                    признак l-value для
                                    выражения до операции "+"
```

5. Генерация кода. Интерпретация

Последняя фаза трансляции - генерация кода или интерпретация - по своему способу включения в транслятор аналогична семантическому анализу: при выполнении шага синтаксического разбора ("свертке" правила в восходящем или подборе правила в нисходящем разборе) вызывается семантическая процедура, по результатам которой вызывается аналогичная процедура для генерации кода и интерпретации. В принципе эти две процедуры можно объединить.

Использование стека для компиляции выражений при восходящем разборе

Поскольку всех восходящих методах СА используется стек, то любая "свертка" может генерировать команды, рассчитанные на размещение операндов (или их адресов) в аппаратном стеке процессора. Пусть некоторый условный процессор имеет следующий набор команд для работы со стеком:

- PUSH(V) – загрузить значение V в стек;
- V=POP() – извлечь значение из вершины стека;

- V=GET(n) – получить значение n-го элемента относительно вершины стека, не удаляя его оттуда.

Тогда “свертка” общеизвестных правил может сопровождаться генерацией неизменного кода относительно текущего состояния “аппаратного” стека:

.

```
E ::= E + T         a=POP(); a=a+POP(); PUSH(a);
F ::= E[c]          a=POP(); a=a*sizeof(x);
                    a=a+POP(); PUSH(*a);
F ::=c              PUSH(c);
```

Особенности интерпретации управляющих структур программы

Принцип интерпретации заключается в одновременном анализе и выполнении программы. При этом сама интерпретация управляющих структур программы (прежде всего, условных и циклических конструкций) должна выполняться одновременно с их синтаксическим анализом. Отсюда следует такой вывод: текст некоторой части программы просматривается, анализируется и интерпретируется столько раз, сколько раз выполняется эта программная ветвь. Простые способы обеспечить это свойство в процессе нисходящего синтаксического разбора состоят в следующем:

- синтаксический анализ условного оператора нужно выполнять полностью, но включать интерпретацию только для той ветви, которая должна выполняться;
- синтаксический анализ любой циклической конструкции должен предусматривать возврат по тексту к началу тела цикла и повторный его синтаксический разбор на каждом шаге.

Сказанное можно проиллюстрировать примером интерпретатора конструкций if и do…while, использующим метод рекурсивного спуска.

```
// Признак proc=1 выполнение конструкции
void IF(int prc)
{ if (s[i]!='(') { error(); return; }
i++;
int x=W();
if (s[i]!=')') { error(); return; }
// Условное выполнение при x=1
i++; OP(proc & x);
if (s[i]=='e')
// Условное выполнение else при x=0
    { i++; OP(proc & !x); }

}


void DO(int proc)
{                    // Запомнить начало тела цикла
int n = i;
do                   // Повторный синтаксический разбор
      { i=n;       // и интерпретация тела цикла
      OP(proc);
      if (s[i++]!='w') { error(); return; }
      if (s[i++]!='(') { error(); return; }
      } while (W(proc);)
if (s[i++]!=')') { error(); return; }
}
```

Особенности компиляции управляющих структур программы

Основным отличием компиляции управляющих структур программы (прежде всего, операторов) является то, что машинный код имеет обычно последовательный характер (то есть, основан на использовании операции перехода GOTO), а для управляющих структур характерен принцип вложенности. Поэтому с каждой компонентой управляющей структуры может быть связан сколь угодно длинный код, представляющий собой результат компиляции вложенных компонент. При этом возможны следующие решения:

- при восходящем СА необходимо использовать принцип сохранения частей генерируемого кода. То есть с каждым нетерминалом, соответствующим оператору, связывается некоторый временный

файл (или массив), содержащий сгенерированный код этой компоненты. Когда производится "свертка" по некоторому правилу, то сгенерированные коды для нетерминалов правой части правила переписываются в общий код, связанный с нетерминалом левой части. Здесь приведен пример для условного оператора:

.

```
IF::=if (W) OP1 else OP2
         |   |         |__файл OP2
         |   |____________файл OP1
         |________________файл W
```

.

```
IF →   x=(файл W);
IF NOT X GOTO M1;
      (файл OP1); GOTO M2;
M1:    (файл OP2);
M2:
```

- При нисходящем разборе наряду с определенным выше методом сохранения частей управляющего кода возможна также генерация "меток вперед". В этом случае возможна запись генерируемого кода в обычный последовательный файл. Проиллюстрируем, как будет выглядеть метод рекурсивного спуска для условного оператора:

```
void IF()
{
if (s[i]!='(') { error(); return; }
i++;
// Анализ условия и генерация кода
W();
if (s[i]!=')') { error(); return; }
i++;
// Генерация кода для проверки условия
cout << "IF NOT A THEN GOTO M1;"
// Анализ оператора и генерация кода
OP();
if (s[i]!='e')
// генерация кода для if без else
   cout << "M1:";
else
// генерация кода для else
   { i++;
   cout << "GOTO M2; M1:";
   // Анализ оператора и генерация кода
   OP();
   cout << "M2:"
      }
}
```

6. Задания к контрольным и лабораторным работам

Задания к контрольной работе №1

Для заданной лексики:

- написать фрагмент программы неформального лексического анализа, используя программную заготовку hardlex.cpp;

- построить диаграмму состояний-переходов КА лексического анализатора.

- определить классы символов и построить матрицу переходов КА. По возможности проверить работоспособность анализатора, используя программную заготовку lexan.cpp.

Вариант задания содержит 4-5 лексем из следующего списка (по умолчанию используется синтаксис языка Си):

идентификаторы произвольной длины;

десятичные константы;

восьмеричные константы;

шестнадцатеричные константы;

строковые константы. Символ-ограничитель константы “ внутри строки передается в виде последовательности \”;

комментарии вида /*…*/;

операции +,++,-,--,*, / ;

операции <, << ,>, >> ;

операции &, &&, |, || ;

операции =,==,!=,++, +=,+ ;

служебные слова int, if, interrupt;

служебные слова for, float, far;

служебные слова string, struct, step;

служебные слова else, end, exit;

строковые константы. Символ-ограничитель константы “ внутри строки передается в виде последовательности из двух таких символов (пустые строки не допускаются);

комментарии вида //…//;

комментарии вида ((…));

“смайлики” вида “:-)” , “:-(”, “:-)) ”, “:-((” (или другие, по выбору).

Задания к контрольной работе №2

Используя заданный синтаксис построить формальные грамматики, для них разработать представление управляющих данных и привести пример разбора по заданному алгоритму:

- Алгоритм рекурсивного спуска. Разработать грамматику. Написать функции разбора правил, аналогично приведенным в sindown1.cpp. Привести пример дерева разбора.
- Нисходящий разбор с использованием LL(1) грамматики. Разработать грамматику. Создать таблицу выбирающих символов и таблицу действий МА. Привести пример дерева разбора. По возможности проверить грамматику на программной заготовке lgram1.cpp;
- Восходящий метод "свертки-переноса". Разработать грамматику. Создать таблицу действий МА. Привести пример дерева разбора. . По возможности проверить грамматику на программной заготовке lowtohigh.cpp.

Желательно результаты работы проверить на соответствующей программной заготовке (имена приведены в тексте) и в файл текста программы внести изменения, касающиеся представления заданной грамматики. Результат разбора (дерево разбора) также внести в файл в виде комментария.

Вариант задания содержит грамматику для арифметических выражений со скобками (операции +,-,*,/,) и для 4-5 конструкций из следующего списка (по умолчанию используется синтаксис языка Си, в том числе приоритеты и направление выполнения операций):

Логические операции &&, ||, !;

Операции [], () – вызов функции;

Операции над указателями *,&;

Операции ++,--;

Операции сравнения ==, !=,<, >, <=, >=;

© Романов Е.Л. Основы построения трансляторов. (конспект лекций)

Операция присваивания =;

Операция “запятая”;

Поразрядные операции &, |, ~, ^;

Операции над структурой (“стрелочка” и “точка”);

Преобразование “выражение – оператор” (символ “;”);

Блок (последовательность операторов, заключенная в {});

Оператор while;

Оператор do – while;

Оператор for;

Оператор if (с else и без него);

Оператор if с синтаксисом if-else-endif (Бейсик);

Оператор do-loop (Бейсик);

Определение простых переменных и массивов типов int, double;

Заголовок функции с результатом и параметрами типов int, double;

Определение простых переменных и кратных указателей вида int ***p.

7. Список литературы

1. Ф. Льюис, Д. Розенкранц, Р. Стирнз. Теоретические основы проектирования компиляторов. М., Мир, 1979.

2. Р. Хантер. Проектирование и конструирование компиляторов. М., Мир.

3. Т. Пратт. Языки программирования. Разработка и реализация. М., Мир, 1979.

4. Л. Бек. Введение в системное программирование. М., Мир, 1988.

5. Язык Си для профессионалов. И.В.К.Софт. М.: 1991.