УЗБЕКСКОЕ АГЕНТСТВО СВЯЗИ И ИНФОРМАТИЗАЦИИ

ТАШКЕНТСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Кафедра «Технологии программирования»

МЕТОДИЧЕСКОЕ ПОСОБИЕ ДЛЯ ЛАБОРАТОРНЫХ ЗАНЯТИЙ ПО ДИСЦИПЛИНЕ

"Системное программное обеспечение"

для студентов направлений образования
5521900 – Информатика и информационные технологии
5140900 – Профессиональное образование
5320200 – Информатизация и библиотечность

Toshkent 2008

Авторы: Назиров Ш.А., Кабулов Р.В., Урынбаев С.К. Методическое пособие для лабораторных занятий по дисциплине «СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ»

Данное методическое пособие предназначено для студентов изучающих предмет «Системное программное обеспечение».

Цель пособия – закрепить знания, полученные при изучении теоретической части курсов и получить практические навыки разработки компилятора и лексического анализатора входного языка транслятора.

Методическое пособие соответствует программе курса «Системное программное обеспечение», целью которой является познакомить студента с основным приемами, необходимыми для решения приведенных задач.

№	Темы	Часы
1	Лексический анализ входного языка транслятора.	2
2	Методы построения таблиц идентификаторов	2
3	Лексический анализатор как конечный автомат	2
4	Метод рекурсивного спуска синтаксического анализа	2
5	Нисходящий синтаксический анализ без возвратов	2
6	Восходящий синтаксический анализ без возвратов	2
7	Семантический анализ	2
8	Генерация кода. Интерпретация	2
	Всего	16

Рецензенты:

доцент НУ Уз
кандидат физика – математических наук Хайдаров А.

старший научный сотрудник
Института Математики и
Информационных технологии
кандидат физика – математических наук Нуралиев Ф.Н.

Лабораторная работа №1

Тема: Лексический анализ входного языка транслятора

Цель. Цель работы состоит в составлении программы (сканера), производящей лексический анализ текста, соответствующего заданному алфавиту и грамматике алгоритмического языка.

I.Краткие теоретические сведения

Лексический анализ

Этап лексического анализа текста исходной программы выделяется в самостоятельный этап работы транслятора, как с методической целью, так и с целью сокращения времени компиляции программы. Последнее достигается за счёт того, что исходная программа в виде последовательности символов, преобразуется на этапе лексической обработки к некоторому стандартному виду, что облегчает дальнейший анализ.

Под лексическим анализом понимают процесс предварительной обработки исходной программы, на котором основные лексические единицы программы - лексемы: ключевые слова, идентификаторы, метки, константы приводятся к единому формату и заменяются условными кодами или ссылками на соответствующие таблицы, а комментарии исключаются из текста программы.

Результатом лексического анализа является список лексем-дескрипторов и таблиц. В таблицах хранятся значения выделенных в программе лексем.

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

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

Количество классов лексем в языках программирования может быть различным. Наиболее распространёнными классами являются:

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

Могут вводиться и другие классы. Это обусловлено в первую очередь той ролью, которую играют различные виды слов при написании исходной программы и переводе её в машинную программу. При этом наиболее предпочтительным является разбиение всего множества слов, допускаемых в языке программирования, на такие классы, которые бы не пересекались между собой.

В общем случае все выделяемые классы являются либо конечными (ключевые слова, разделители и др.) - классы фиксированных для данного языка программирования слов, либо бесконечными или очень большими (идентификаторы, константы, метки)- классы переменных для данного языка программирования слов.

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

В ходе лексического анализа значения лексем из бесконечных классов помещаются в таблицы соответствующих классов. Конечность таблиц объясняет ограничения, существующие в языках программирования на длины и соответственно число используемых в программе идентификаторов и констант.

Числовые константы перед помещением их в таблицу могут переводиться из внешнего символьного во внутреннее машинное представление. Содержимое таблиц, в особенности таблицы идентификаторов, в дальнейшем пополняется на этапе семантического анализа исходной программы и используется на этапе генерации объектной программы.

Лексический анализатор (или сканер) - это часть компилятора, которая читает литеры программы на исходном языке и строит из них слова (лексемы) исходного языка. На вход лексического анализатора поступает текст исходной программы, а выходная информация передается для дальнейшей обработки компилятором на этапе синтаксического анализа и разбора.

Рассмотрим основные идеи, которые лежат в основе построения лексического анализатора, и проблемы, возникающие при его разработке.

Первоначально в тексте входной программы сканер выделяет последовательность символов, которая по его предположению должна быть словом в программе, т.е. лексемой. Может выделяться не вся последовательность, а только один символ, который считается началом лексемы. Это сделать просто, если слова в программе отделяются друг от друга специальными разделителями, например, пробелами или запрещено использование служебных слов в качестве переменных, либо классы лексем распознаются по вхождению первых символов лексемы.

Затем, проводится идентификация лексемы. Она заключается в сборке лексемы из символов, начиная с выделенного на предыдущем этапе, и проверки правильности записи лексемы данного класса.

Идентификация лексемы из конечного класса выполняется путём сравнения её с эталонным значением. Основная проблема здесь - минимизация времени поиска эталона. В общем случае может понадобиться полный перебор слов данного класса, особенно, если выделенное слово содержит ошибку. Уменьшить время поиска можно, используя различные

методы ускоренного поиска: упорядоченный список, линейный список, метод расстановки и др.

Для идентификации из очень больших классов используются специальные методы сборки лексем с одновременной проверкой правильности написания. В этих методах применяется формальный математический аппарат- теория регулярных языков и конечных распознавателей.

При успешной идентификации значение лексемы из бесконечного класса помещается в таблицу идентификации лексем данного класса. При этом осуществляют проверку: не хранится ли уже там значение данной лексемы, т.е. необходимо проводить просмотр элементов таблицы. Таблица при этом должна допускать расширение. Опять же для уменьшения времени доступа к элементам таблицы она должна быть специальным образом организована, при этом должны использоваться специальные методы ускоренного поиска элементов.

После проведения успешной идентификации лексемы формируется её образ - дескриптор, он помещается в выходные данные лексического анализатора. В случае неуспешной идентификации формируется сообщение об ошибках в написании слов программы.

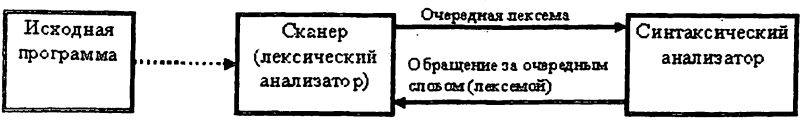
В ходе лексического анализа могут выполняться и другие виды лексического контроля, в частности, проверяться парность скобок и других парных символов, наличие метки у оператора, следующего за GOTO и т.д.

Результаты работы сканера передаются в последствии на вход синтаксического анлизатора. Имеются две возможности их связи: раздельная связь и нераздельная связь.

При раздельной связи выходные данные сканера формируются полностью и затем передаются синтаксическому анализатору. При нераздельной связи, когда синтаксическому анализатору требуется очередной образ лексемы, он вызывает лексический анализатор, который генерирует дескриптор и возвращает управление синтаксическому анализатору.

Второй вариант характерен для однопроходных трансляторов. Таким образом, процесс лексического анализа достаточно прост, но может занимать значительное время трансляции.

Работу синтаксического и лексического анализаторов при нераздельной связи можно изобразить в виде следующей схемы:



Рассмотрим конкретный пример. Пусть нам дана программа на некотором алгоритмическом азыке:

```
PROGRAM PRIMER;
VAR X,Y,Z : REAL;
BEGIN
  X:=5;
  Y:=6;
  Z:=X+Y;
END;
```

Применим следующие коды для типов лексем:
К1- ключевое слово;
К2- разделитель;
К3- идентификатор;
К4- константа.

Лексический анализ можно производить, если нам задан алфавит, список ключевых слов языка и служебных символов.

Пусть всё это имеется. Тогда внутренние таблицы сканера примут следующий вид.

Таблица 1. Ключевые слова

№	Ключевое слово
1.	PROGRAM
2.	BEGIN
3.	END
4.	FOR
5.	REAL
6.	VAR

Таблица 2. Разделители

№	Разделители
1.	;
2.	,
3.	+
4.	-
5.	/
6.	*
7.	:
8.	=
9.	.

Результат работы сканера таблица идентификаторов и таблица констант

Таблица 3. Идентификаторы

№	Идентификаторы
1.	PRIMER
2.	X
3.	Y
4.	Z

Таблица 4. Константы

№	Знач. констант
1.	5
2.	6

На основании составленных таблиц можно записать входной текст через введённые дескрипторы (дескрипторный текст):

(K1, 1) (K3, 1) (K2, 1)
(K1, 6) (K3, 2) (K2, 2) (k3, 3) (K2, 2) (K3, 4) (K2, 7) (K1, 5) (K2,1)
(K1, 2)
(K3, 2) (K2, 7) (K2, 8) (K4, 1) (K2, 1)
(K3, 3) (K2, 7) (K2, 8) (K4, 2) (K2, 1)
(K3, 4) (K2, 7) (K2, 8) (K3, 2) (K2, 3) (K3, 3) (K2, 1)
(K1, 3) (K2, 9).

Простейший лексический анализатор

Таким образом, алгоритм работы простейшего сканера можно описать так:

- просматривается входной поток символов программы на исходном языке до обнаружения очередного символа, ограничивающего лексему;
- для выбранной части входного потока выполняется функция распознавания лексемы;
- при успешном распознавании информация о выделенной лексеме заносится в таблицу лексем, и алгоритм возвращается к первому этапу;
- при неуспешном распознавании выдается сообщение об ошибке, а дальнейшие действия зависят от реализации сканера - либо его выполнение прекращается, либо делается попытка распознать следующую лексему (идет возврат к первому этапу алгоритма).

Работа программы-сканера продолжается до тех пор, пока не будут просмотрены все символы программы на исходном языке из входного потока.

II. Постановка задачи.

1. Написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием и порождает таблицу лексем. Текст на входном языке задается в виде символьного (текстового) файла. Программа должна выдавать сообщения о наличие во входном тексте ошибок, которые могут быть обнаружены на этапе лексического анализа.

Длину идентификаторов и строковых констант считать ограниченной 32 символами. Программа должна допускать наличие комментариев неограниченной длины во входном файле. Форму организации комментариев предлагается выбрать самостоятельно.

2. Написать программу, которая выполняет то же самое, но с помощью метода двоичного поиска в упорядоченной таблице лексем.

III. Варианты

1. Входной язык содержит арифметические выражения, разделенные символом ;(точка с запятой). Арифметические выражения состоят из идентификаторов, десятичных чисел с плавающей точкой (в обычной и логарифмической форме), знаков операций и скобок.

2. Входной язык содержит математические выражения, разделенные символом ;(точка с запятой). Математические выражения состоят из идентификаторов, десятичных чисел с плавающей точкой (в обычной и логарифмической форме), знаков операций, вызовов математических функций и скобок.

3. Входной язык содержит логические выражения, разделенные символом;(точка с запятой). Логические выражения состоят из идентификаторов, знаков операций и скобок.

4. Входной язык содержит упрощенные операторы цикла типа ***while*** *<логическое выражение>* ***do*** *<оператор присваивания>*; Логическое выражение может содержать идентификаторы, знаки операций сравнения, целые десятичные числа без знака, скобки и логические операции ***and*** и ***or***. Оператор присваивания должен состоять из идентификатора, знака присваивания и целой десятичной константы без знака.

5. Входной язык содержит упрощенные операторы цикла типа ***repeat*** *<оператор присваивания>* ***until*** *<логическое выражение>*. Логическое выражение может содержать идентификаторы, знаки операций сравнения, целые десятичные числа без знака, скобки и логические операции ***and*** и ***or***. Оператор присваивания должен состоять из идентификатора, знака присваивания и целой десятичной константы без знака.

6. Входной язык содержит упрощенные условные операторы типа ***if*** *<логическое выражение>* ***then*** *<оператор присваивания>* ***else*** *<оператор присваивания>*; (часть ***else*** в операторе может отсутствовать). Логическое выражение может содержать идентификаторы, знаки операций сравнения,

целые десятичные числа без знака, скобки и логические операции ***and*** и ***not***. Оператор присваивания должен состоять из двух идентификаторов, разделенных знаком присваивания.

7. Входной язык содержит упрощенные операторы цикла типа ***for*** *<оператор присваивания>* ***to*** *<константа >* ***do*** *<оператор присваивания>*;. Оператор присваивания должен состоять из идентификатора, знака присваивания и целой десятичной константы без знака. Константа – целое десятичное число без знака.

8. Входной язык содержит упрощенные операторы цикла типа ***for*** *<оператор присваивания>* ***to*** *<константа >* ***downto*** *<оператор присваивания>*;. Оператор присваивания должен состоять из идентификатора, знака присваивания и целой десятичной константы без знака. Константа – целое десятичное число без знака.

9. Входной язык содержит выражения над строковыми константами, разделенные символом;(точка с запятой). Выражения состоят из идентификаторов, строковых констант, заключенных в двойные кавычки, одиночных символов, заключенных в одинарные кавычки и знаков операции конкатенации +.

10.Входной язык содержит последовательность вызовов процедур, разделенных символом ;(точка с запятой). Вызов процедуры должен состоять из имени процедуры и списка параметров. В качестве параметров могут выступать идентификаторы, целые десятичные числа без знака, шестнадцатеричные числа, десятичные числа с плавающей точкой.

11.Входной язык содержит последовательность вызовов процедур, разделенных символом ;(точка с запятой). Вызов процедуры должен состоять из имени процедуры и списка параметров. В качестве параметров могут выступать идентификаторы, строковые константы, заключенные в двойные кавычки и одиночные символы, заключенные в одинарные кавычки.

12.Входной язык содержит последовательность описаний массивов в соответствии со спецификацией языка Паскаль, разделенных символом ;(точка с запятой). Считать, что массивы могут содержать только элементы скалярных типов ***integer***, ***real***, ***byte***, ***word*** и ***char***.

13.Входной язык содержит последовательность описаний переменных и указателей в соответствии со спецификацией языка Паскаль, разделенных символом;(точка с запятой). Считать, что могут быть использованы только типы ***integer***, ***real***, ***byte***, ***word*** и ***char***.

14.Входной язык содержит последовательность описаний записей (***record***) в соответствии со спецификацией языка Паскаль, разделенных символом ;(точка с запятой). Считать, что записи могут содержать только поля скалярных типов ***integer***, ***real***, ***byte***, ***word***, ***char*** и строки ***string*** с возможным указанием длины строки в квадратных скобках.

Лабораторная работа №2

Тема: Методы построения таблиц идентификаторов

Цель. Цель работы состоит в составлении программы с применением различных методов построения таблиц идентификаторов на примере заданного простейшего входного языка.

I. Краткие теоретические сведения

Организация таблиц символов компилятора

В процессе работы компилятор хранит информацию об объектах программы. Как правило, информация о каждом объекте состоит из двух основных элементов: имени объекта и его свойств.

Информация об объектах программы должна быть организована таким образом, чтобы поиск ее был по возможности быстрей, а требуемая память по возможности меньше. Кроме того, со стороны языка программирования могут быть дополнительные требования.

Имена могут иметь определенную область видимости. Например, поле записи должно быть уникально в пределах структуры (или уровня структуры), но может совпадать с именем объектов вне записи (или другого уровня записи). В то же время имя поля может открываться оператором присоединения, и тогда может возникнуть конфликт имен (или неоднозначность в трактовке имени). Если язык имеет блочную структуру, то необходимо обеспечить такой способ хранения информации, чтобы, во-первых, поддерживать блочный механизм видимости, а во-вторых - эффективно освобождать память по выходе из блока. В некоторых языках (например, Аде) одновременно (в одном блоке) могут быть видимы несколько объектов с одним именем, в других такая ситуация недопустима. Мы рассмотрим некоторые основные способы организации информации в компиляторах: таблицы идентификаторов, таблицы символов, способы реализации блочной структуры.

Таблицы идентификаторов и таблицы символов

Как уже было сказано, информацию об объекте обычно можно разделить на две части: имя (идентификатор) и описание. Удобно эти характеристики объекта хранить по отдельности. Это обусловлено двумя причинами: 1) символьное представление идентификатора может иметь неопределенную длину и быть довольно длинным; 2) как уже было сказано, различные объекты в одной области видимости и/или в разных могут иметь одинаковые имена и незачем занимать память для повторного хранения идентификатора. Таблицу для хранения идентификаторов называют таблицей идентификаторов, а таблицу для хранения свойств объектов - таблицей символов. В таком случае одним из

15. Входной язык содержит последовательность описаний констант в соответствии со спецификацией языка Паскаль. Константы могут быть целыми десятичными числами со знаком, целыми шестнадцатеричными числами, целыми десятичными числами с плавающей точкой, строками или символами.

IV. Содержание отчета

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.
2. Постановка задачи
3. Описание КС-грамматики входного языка в форме Бэкуса-Наура.
4. Описание алгоритма работы сканера для распознавания цепочек (в соответствии с вариантом задания).
5. Текст программы (оформляется после выполнения программы на ЭВМ).
6. Выводы по проделанной работе.

V. Методические указания

1. Простейший лексический анализатор

Простейший лексический анализатор можно построить, не прибегая ни к каким формальным методам, просто анализируя последовательности символов, образующих лексические единицы. 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;
}
```

свойств объекта становится его имя и в таблице символов хранится указатель на соответствующий вход в таблицу идентификаторов. .

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

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

Построение таблиц идентификаторов на основе функций расстановки (хэш - функций)

Функцией расстановки или хэш – функцией называется целочисленная функция h, определенная на множестве идентификаторов и принимающая значения от 0 до N, где N размер таблицы идентификаторов.

При построении таблиц идентификаторов для заданного идентификатора A вычисляется значение функции расстановки k=h(A). Если строка с номером k занята, то идентификатор уже существует в таблице, если свободна, то в эту строку записывается идентификатор.

При поиске для заданного идентификатора A также вычисляется значение функции расстановки k=h(A). Если строка с номером k занята, то идентификатор существует в таблице, если свободна, идентификатор не найден.

На практике функции расстановки могут иметь одно и то же значение для различных идентификаторов. При поиске в такой таблице если элемент таблицы h(A) свободен, то это означает, что идентификатора в таблице нет. Если же занят, то это еще не означает, что идентификатор A в таблицу занесен, поскольку (вообще говоря) много идентификаторов могут иметь одно и то же значение функции расстановки.

Построение таблиц идентификаторов на основе такой функции расстановки может быть организован методом повторной расстановки или рехэширования.

Пусть A - идентификатор. Если элемент таблицы k=h(A) свободен, то по данному адресу записывается идентификатор. Если же элемент занят, вычисляют вторичную функцию расстановки k=h2(k). Если элемент таблицы свободен, то по данному адресу записывается идентификатор, иначе вычисляется k=h3(k), и т.д. Как правило, hi=h2 для i>=2. Аргументом функции h2 является целое в диапазоне [0,N-1].

Поиск идентификатора осуществляется таким же образом. Пусть A - идентификатор. Если элемент таблицы h(A) свободен, то это означает, что идентификатора в таблице нет. Если же занят, то это еще не означает, что

идентификатор A в таблицу занесен, поскольку (вообще говоря) много идентификаторов могут иметь одно и то же значение функции расстановки. Для того чтобы определить, нашли ли мы нужный идентификатор, сравниваем A с элементом таблицы h(A). Если они равны – идентификатор нашли, если нет - надо продолжать поиск дальше. Для этого вычисляют вторичную функцию расстановки h2(A). Вновь возможны четыре варианта: либо элемент таблицы свободен, либо нашли идентификатор, либо таблица вся просмотрена и идентификатора нет, либо надо продолжать поиск. Для продолжения поиска применяют вновь функцию h3(A), и т.д. Как правило, hi=h2 для i>=2.

Аргументом функции h2 является целое в диапазоне [0.N-1] и она может быть устроена по-разному.

1) h2(i)=(i+1) mod N

Берется следующий (циклически) элемент массива. Этот вариант плох тем, что занятые элементы "группируются", образуют последовательные занятые участки и в пределах этого участка поиск становится по-существу линейным.

2) h2(i)=(i+k) mod N, где k и N взаимно просты.

По-существу это предыдущий вариант, но элементы накапливаются не в последовательных элементах, а "разносятся".

3) h2(i)=(a*i+c) mod N - "псевдослучайная последовательность".

Здесь c и N должны быть взаимно просты, b=a-1 кратно p для любого простого p, являющегося делителем N, b кратно 4, если N кратно 4.

Таблицы символов и таблицы расстановки

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

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

Функции расстановки (хэш – функции)

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

1. По символам строки s определяем положительное целое H.

Преобразование одиночных символов в целые обычно можно сделать средствами языка реализации. В Паскале для этого служит функция ord, в Си при выполнении арифметических операций символьные значения трактуются как целые.

2. Преобразуем H, вычисленное выше, в номер списка, т.е. целое между 0 и m-1, где m - размер таблицы расстановки, например, взятием остатка при делении H на m.

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

Простейший способ вычисления H - сложение кодов символов строки. Перед сложением с очередным символом можно умножить старое значение H на константу q. Т.е. полагаем H0=0, Hi=q*Hi-1+ci для 1<=i<=k, k - длина строки. При q=1 получаем простое сложение символов. Вместо сложения можно выполнять сложение ci и q*Hi-1 по модулю 2. Переполнение при выполнении арифметических операций можно игнорировать.

Сравнение различных методов реализации таблиц

Рассмотрим преимущества и недостатки тех или иных методов реализации таблиц с точки зрения техники использования памяти. Если таблица размещается в массиве, то, с одной стороны, отпадает необходимость использования динамической памяти, а с другой - появляется ряд осложнений. Использование динамической памяти, как правило, довольно дорогая операция, поскольку механизмы поддержания работы с динамической памятью довольно сложны. Необходимо поддерживать списки свободной и занятой памяти, выбирать наиболее подходящий кусок памяти при запросе, включать освободившийся кусок в список свободной памяти и, возможно, склеивать куски свободной памяти в списке. С другой стороны, использование массива требует отведения заранее довольно большой памяти, а это означает, что значительная память вообще не будет использоваться. Кроме того, часто приходится заполнять не все элементы массива (например, в таблице идентификаторов или в тех случаях, когда в массиве фактически хранятся записи переменной длины, например, если в таблице символов записи для различных объектов имеют различный состав полей). Обращение к элементам массива может означать использование операции умножения при вычислении индексов, что может замедлить исполнение. Наилучшим, по-видимому, является механизм доступа по указателям и использование факта магазинной организации памяти в компиляторе. Для этого процедура выделения памяти выдает необходимый кусок из подряд идущей памяти, а при выходе из процедуры вся память, связанная с этой процедурой, освобождается простой перестановкой указателя свободной памяти в состояние перед началом обработки процедуры. В чистом виде это не всегда, однако, возможно.

Например, локальный модуль в Модуле-2 может экспортировать некоторые объекты наружу. При этом схему реализации приходится "подгонять" под механизм распределения памяти. В данном случае, например, необходимо экспортированные объекты вынести в среду охватывающего блока и свернуть блок локального модуля.

II. Постановка задачи

1. Написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием и порождает таблицу лексем с помощью таблиц расстановки и повторной расстановки. Текст на входном языке задается в виде символьного (текстового) файла. Программа должна выдавать сообщения о наличие во входном тексте ошибок, которые могут быть обнаружены на этапе лексического анализа.

Длину идентификаторов и строковых констант считать ограниченной 32 символами. Программа должна допускать наличие комментариев неограниченной длины во входном файле. Форму организации комментариев предлагается выбрать самостоятельно.

2. Выполнить то же самое с использованием таблиц символов.

III. Варианты

1. Входной язык содержит арифметические выражения, разделенные символом ;(точка с запятой). Арифметические выражения состоят из идентификаторов, десятичных чисел с плавающей точкой (в обычной и логарифмической форме), знаков операций и скобок.

2. Входной язык содержит математические выражения, разделенные символом ;(точка с запятой). Математические выражения состоят из идентификаторов, десятичных чисел с плавающей точкой (в обычной и логарифмической форме), знаков операций, вызовов математических функций и скобок.

3. Входной язык содержит логические выражения, разделенные символом;(точка с запятой). Логические выражения состоят из идентификаторов, знаков операций и скобок.

4. Входной язык содержит упрощенные операторы цикла типа ***while*** *<логическое выражение>* ***do*** *<оператор присваивания>*; Логическое выражение может содержать идентификаторы, знаки операций сравнения, целые десятичные числа без знака, скобки и логические операции ***and*** и ***or***. Оператор присваивания должен состоять из идентификатора, знака присваивания и целой десятичной константы без знака.

5. Входной язык содержит упрощенные операторы цикла типа ***repeat*** *<оператор присваивания>* ***until*** *<логическое выражение>*. Логическое выражение может содержать идентификаторы, знаки операций сравнения, целые десятичные числа без знака, скобки и логические операции ***and*** и ***or***.

Оператор присваивания должен состоять из идентификатора, знака присваивания и целой десятичной константы без знака.

6. Входной язык содержит упрощенные условные операторы типа ***if*** *<логическое выражение>* ***then*** *<оператор присваивания>* ***else*** *<оператор присваивания>*; (часть ***else*** в операторе может отсутствовать). Логическое выражение может содержать идентификаторы, знаки операций сравнения, целые десятичные числа без знака, скобки и логические операции ***and*** и ***not***. Оператор присваивания должен состоять из двух идентификаторов, разделенных знаком присваивания.

7. Входной язык содержит упрощенные операторы цикла типа ***for*** *<оператор присваивания>* ***to*** *<константа >* ***do*** *<оператор присваивания>*;. Оператор присваивания должен состоять из идентификатора, знака присваивания и целой десятичной константы без знака. Константа – целое десятичное число без знака.

8. Входной язык содержит упрощенные операторы цикла типа ***for*** *<оператор присваивания>* ***to*** *<константа >* ***downto*** *<оператор присваивания>*;. Оператор присваивания должен состоять из идентификатора, знака присваивания и целой десятичной константы без знака. Константа – целое десятичное число без знака.

9. Входной язык содержит выражения над строковыми константами, разделенные символом;(точка с запятой). Выражения состоят из идентификаторов, строковых констант, заключенных в двойные кавычки, одиночных символов, заключенных в одинарные кавычки и знаков операции конкатенации +.

10.Входной язык содержит последовательность вызовов процедур, разделенных символом ;(точка с запятой). Вызов процедуры должен состоять из имени процедуры и списка параметров. В качестве параметров могут выступать идентификаторы, целые десятичные числа без знака, шестнадцатеричные числа, десятичные числа с плавающей точкой.

11.Входной язык содержит последовательность вызовов процедур, разделенных символом **;(точка с запятой). Вызов процедуры должен состоять** из имени процедуры и списка параметров. **В качестве параметров могут выступать идентификаторы, строковые константы, заключенные в двойные** кавычки и одиночные символы, заключенные в одинарные кавычки.

12.Входной язык содержит последовательность описаний массивов в соответствии со спецификацией языка Паскаль. разделенных символом ;(точка с запятой). Считать, что массивы могут содержать только элементы скалярных типов ***integer***, ***real***, ***byte***, ***word*** и ***char***.

13.Входной язык содержит последовательность описаний переменных и указателей в соответствии со спецификацией языка Паскаль, разделенных символом;(точка с запятой). Считать, что могут быть использованы только типы ***integer***, ***real***, ***byte***, ***word*** и ***char***.

14.Входной язык содержит последовательность описаний записей (***record***) в соответствии со спецификацией языка Паскаль, разделенных символом ;(точка с запятой). Считать, что записи могут содержать только поля скалярных типов ***integer***, ***real***, ***byte***, ***word***, ***char*** и строки ***string*** с возможным указанием длины строки в квадратных скобках.

15.Входной язык содержит последовательность описаний констант в соответствии со спецификацией языка Паскаль. Константы могут быть целыми десятичными числами со знаком, целыми шестнадцатеричными числами, целыми десятичными числами с плавающей точкой, строками или символами.

IV. Содержание отчета

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.
2. Постановка задачи
3. Описание КС-грамматики входного языка в форме Бэкуса-Наура.
4. Описание алгоритма работы сканера для распознавания цепочек (в соответствии с вариантом задания).
5. Описание алгоритма построения таблиц идентификаторов (в соответствии с вариантом задания).
6. Текст программы (оформляется после выполнения программы на ЭВМ).
7. Выводы по проделанной работе.

V. Методические указания

1. Использование функций расстановки.

Поиск в таблице можно описать следующим алгоритмом:

```
type Pair=record Yes:boolean;
          Point:integer
     end;
function Search(id):Pair;
var H,H0: integer;
begin H:=h(id); H0:=h;
   loop if T[H]=id then return(true,H)
      elsif T[H]=Empty then return(false,H)
      else H:=h2(H);
      if H=H0 then return(false,NIL);
end  end end end;
```

Алгоритм вырабатывает: (true,P), если нашли требуемый идентификатор, P - указатель на него; (false,NIL), если искомого

идентификатора нет и в таблице нет свободного места; (false.P). если искомого идентификатора нет, но в таблице есть свободный вход P.

Занесение идентификатора в таблицу осуществляется следующим алгоритмом:

```
function Insert(id):integer;
 var P:Pair;
 begin P:=search(id);
    with P do
    if not Yes and Point<>NIL
    then T[Point]:=id;
    end;
    return(Point)
 end; end;
```

2. Использование таблиц расстановки.

Поиск в таблице может быть описан следующей процедурой:

```
type Element= record IdenP:integer;
                Next:pointer to Element;
          end;
Pointer=pointer to Element
function Search(Id):Pointer;
 var P:Pointer;
 begin P:=T[H(Id)];
    loop if P=nil then return(nil)
       elsif IdenTab[P^.IdenP]=Id then return(P)
       else P:=P^.Next
 end  end; end;
```

IdenTab - таблица идентификаторов. Занесение объекта в таблицу может осуществляться следующей процедурой:

```
function Insert(Id):Pointer;
 var P,H:Pointer;
 begin P:=Search(Id);
    if P<>nil then return(P)
    else H:=H(Id); new(P);
       P^.Next:=T[H]; T[H]:=P;
       P^.Idenp:=Include(Id);
    end;
    return(P);
 end;
```

3. Реализация функции расстановки.

Функция Hashpjw, приведенная ниже. вычисляется, начиная с H=0. Для каждого символа с сдвигаем биты H на 4 позиции влево и добавляем с. Если какой-нибудь из четырех старших бит H равен 1. сдвигаем эти 4 бита на 24 разряда вправо. затем складываем по модулю 2 с H и устанавливаем в 0 каждый из четырех старших бит, равных 1.

```
#define PRIME 211
#define EOS '\0'
int Hashpjw(s)
char *s;
{ char *p;
unsigned H=0, g;
for (p=s; *p != EOS; p=p+1)
 {H=(H<<4)+(*p);
  if (g = H & 0xf0000000)
    {H=H^(g>>24);
    H=H^g;
}  }
return H%PRIME;
```

Лабораторная работа № 3

Тема: Лексический анализатор как конечный автомат

Цель. Цель лабораторной работы состоит в составлении программы, производящей лексический анализ текста на основе конечного автомата, соответствующего заданному алфавиту и грамматике алгоритмического языка.

I. Краткие теоретические сведения

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 A, при получении сигнала S объект переходит в состояние B и при этом выполняет действие D - то такая система будет представлять из себя конечный автомат. На диаграмме состояний-переходов каждому состоянию соответствует кружок, каждому переходу - дуга со стрелкой. Каждое состояние имеет свой "смысл", заключенный в подписи. Каждый переход осуществляется только при выполнении определенного условия, которое обозначено подписью под дугой. Кроме того, при выполнении перехода производится действие, при помощи которого автомат обнаруживает свое поведение извне.

У конечного автомата есть еще несколько условий работоспособности:

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

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

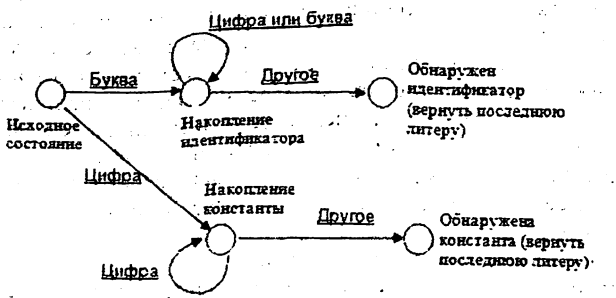
Диаграмма состояний и переходов лексического анализатора

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.



II. Постановка задачи

1. Для заданного варианта лексики:

построить диаграмму состояний и переходов распознающего КА. обратить внимание на количество возвращаемых символов в каждом заключительном состоянии и на логику распознавания префиксов и суффиксов (начала и окончания) лексем;

реализовать и протестировать программу лексического анализатора на основе построенного КА.

2. Реализовать и протестировать программу позволяющей выделить из произвольной строки лексемы заданного типа.

III. Варианты

1. 1111111 | 1222222221 | 123333333321 – количество цифр в середине – любое.

2. (.....) | ((.....)) | (((....)))) – одинарные. двойные или тройные парные скобки. содержащие внутри любую цепочку символов.

3. (+,-)125.666E(+,-)44 – вещественная константа произвольного вида. содержащая целую, дробную части и порядок, а также знаки (любая компонента может быть пропущена).

4. 1111122222333333222111 1 – последовательность из любого количества цифр (цифры 2,3 могут отсутствовать).

5. 0+++0-000+0-- - константы из 0-символов. операции постинкремента и постдекремента, сложения и вычитания (автомат проверяет правильность всей цепочки).

6. +01 | +10 | + | 11010010 – одиночный +, после + лексема может содержать 01 или 10. любое другое сочетание (11 или 00) считается началом другой лексемы (подсказка: возврат 2 символов).

7. aaaaaaabc | aaaaa | bbbbbca | bbbbbb | cccccab | cccccc – повторяющаяся цепочка из 2 и более символов a,b или c. после которых может следовать пара символов (подсказка: возврат 2 символов).

8. "".....\123...\321...\..."..." – строковая константа. содержащая восьмеричные константы и кавычки.

9. Строковые константы 1.....21...22....1 и 2.....12...11.....2, где 1- ". 2-', заключенные в одинарные или двойные апострофы. Если в строке встречается такой символ. то он предваряется апострофом другого вида.

10. 111+22+3--1+2-1 – знаки +/- увеличивают/уменьшают ожидаемое значение очередной цифры на 1.

11. Строка, в которой символы +/- не могут стоять рядом (обязательно должны быть разделены каким-либо другим).

12. Строка цифр 1,2,3 в любой последовательности.

13. ((...)(..)((..)..)) – строка. содержащая вложенные скобки с глубиной вложенности не более 3. . Проверка лексической правильности ВСЕЙ строки.

14. Строка цифр 1,2,3 в любой последовательности. Выделяет лексемы из последовательности НЕУМЕНЬШАЮЩИХСЯ цифр, например 122222233/13/112.

15. Строка символов a,b,c вида a-a-abc-c-cb-b-acaca-a. Одинаковые символы разделяются знаком '-'.

IV. Содержание отчета

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.
2. Постановка задачи
3. Граф конечного автомата для распознавания цепочек (в соответствии с вариантом задания).
4. Описание алгоритма работы сканера.
5. Текст программы (оформляется после выполнения программы на ЭВМ).
6. Выводы по проделанной работе.

V. Методические указания

1. Анализ последовательностей различных символов в строке, на основе конечного автомата.

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.

Лабораторная работа № 4

Тема: Метод рекурсивного спуска синтаксического анализа

Цель. Цель лабораторной работы состоит в составлении программы, производящей синтаксический анализ формальной грамматики на основе метода рекурсивного спуска.

I. Краткие теоретические сведения

Две модели анализаторов

В лексическом анализе были рассмотрены две модели анализаторов: автоматная и на «жесткой» логике:

✓ В анализаторе на «жесткой» логике лексика «сщита» в управляющие конструкции программы (ветвления, , циклы), т.е. непосредственно в **алгоритмическую** часть. В результате анализатор настроен на единственный вариант лексики;

✓ В автоматной модели лексика «сщита» в управляющие таблицы конечного автомата (КА), т.е. по существу является **данными.** Это позволяет настраивать анализатор на различные варианты лексики, но требует предварительной работы по проектированию самого КА и построению его управляющих таблиц, либо наличия языка формального языка описания лексики и транслятора с него.

Наличие двух вариантов иллюстрирует один из фундаментальных тезисов **теории алгоритмов - эквивалентность алгоритма и данных:** уменьшить объем алгоритмической части программы можно за счет «переноса» управляющих конструкций алгоритма в данные и наоборот.

Нисходящий разбор без возвратов

В синтаксическом анализе, аналогично, известен метод **рекурсивного спуска,** основанный на «зашивании» правил грамматики непосредственно в управляющие конструкции распознавателя. Рассмотрим данный метод для грамматик специального вида.

Идеи нисходящего разбора без возвратов:

✓ происходит последовательный просмотр входной строки слева-направо;

✓ очередной символ входной строки является основанием для выбора одной из правых частей правил группы при замене текущего нетерминала;

✓ терминальные символы входной строки и правой части правила «взаимно уничтожаются»:

✓ обнаружение нетерминала в правой части рекурсивно повторяет этот же процесс.

В методе рекурсивного спуска они претерпевают такие изменения:

✓ каждому нетерминалу соответствует отдельная **процедура (функция),** распознающая (выбирающая и «закрывающая») одну из правых частей

правила, имеющего в левой части этот нетерминал (т.е. для каждой группы правил пишется свой распознаватель):

✓ во входной строке имеется указатель (индекс) на текущий «закрываемый символ». Этот символ и является основанием для выбора необходимой правой части правила. Сам выбор «зашит» в распознавателе в виде конструкций **if** или **switch**. Правила выбора базируются на построении **множеств выбирающих символах SEL**;

✓ просмотр выбранной части реализован в тексте процедуры-распознавателя путем сравнения ожидаемого символа правой части и текущего символа входной строки;

✓ если в правой части ожидается терминальный символ и он совпадает с очередным символом входной строки, то символ во входной строке пропускается, а распознаватель переходит к следующему символу правой части;

✓ несовпадение терминального символа правой части и очередного символа входной строки свидетельствует о синтаксической ошибке;

✓ если в правой части встречается нетерминальный символ, то для него необходимо вызвать аналогичную распознающую процедуру (функцию).

Множества выбирающих символов особенно просто определяются для специальных грамматик называемых S грамматикой.

Правило грамматики называются S правилом, если он отвечает следующим условиям:

- правая часть правила должна начинаться с терминального символа;
- для двух любых S правил, имеющих одинаковые левые части, начальные терминальные символы должны быть различны.

Грамматика называется S грамматикой, если все правила являются S правилами. Выбирающее множество для каждого правила в этом случае состоит из одного символа.

Достоинства метода

Метод рекурсивного спуска имеет еще одно достоинство. Он позволяет отойти от канонической формы представления формальных грамматик и проектировать распознаватель частично на содержательном уровне, что делает его более естественным и «читабельным». Это касается, прежде всего, синтаксических конструкций повторения, которые в тексте распознавателя могут быть заменены обычными циклами типа **while** или **for**. Для этого, группу правил, которая реализует такой цикл, нужно реализовать в виде одного распознавателя, записав в нем соответствующий программный цикл. В теле цикла должна быть часть распознавателя, соответствующая повторяющейся конструкции, а условия продолжения и ограничения цикла должны проверяться соответствующими терминалами.

Второе упрощение заключается в частичном отказе от проверки выбирающих символов в некоторых группах правил. Как известно, явные выбирающие символы имеют место для S-правил. Для аннулирующих правил и правил, начинающихся с нетерминала, их необходимо вычислять. В

том случае, если такое правило в группе – единственное, его выбор можно осуществлять по принципу «от противного», т.е. если не встречается ни один из выбирающих символов S-правил. Правда, при этом теряются ветви программы, ответственные за обнаружение ошибок, хотя это поправимо: ошибки могут быть обнаружены правилами более высокого уровня.

Рекурсивный спуск позволяет также решать на содержательном уровне проблемы, связанные с просмотром вперед на более чем один символ.

Достоинствами метода является естественность написания процедур синтаксического анализа, последовательность вызова которых в программе совпадает с последовательностью нисходящего вывода. В целом, грамматика перекладывается в текст программы достаточно просто и естественно. Для выбора той или иной части правил используются выбирающие символы, аннулирующие правила представляют собой «заглушки», не выполняющие никаких содержательных функций.

Нисходящий разбор с возвратами

Принципы нисходящего разбора можно применить и в «первозданном» виде, не используя каких-либо дополнительных свойств и соотношений, выводимых из грамматики. Метод рекурсивного спуска с возвратами:

✓ шаг рекурсивного алгоритма заключается в выводе всех возможных цепочек из очередного нетерминала и выборе первой подходящей, покрывающей начало незакрытой части входного предложения

✓ в качестве результата рекурсивная функция возвращает индекс – части строки, закрытой цепочкой, выведенной из соответствующего нетерминала, либо ошибку, если цепочку вывести не удается;

✓ рекурсивная функция просматривает все множество правил (заданных массивом указателей на строки) и выбирает те, у которых в левой части стоит закрываемый символ:

✓ просматривая правую часть выбранного правила, программа выполняет различные действия в зависимости от того, является ли он терминалом или нетерминалом:

✓ для терминального символа производится сравнение его с очередным символом строки, при совпадении они оба пропускаются, при несовпадении – правая часть считается неподходящей;

✓ для нетерминального символа функция вызывает сама себя рекурсивно по отношению к этому нетерминалу и очередному символу строки. Если вызов неудачен, то вся правая часть является неподходящей, если удачен – то «закрывается» часть входной строки, просмотренная рекурсивным вызовом

✓ если в результате такого просмотра удается дойти до конца правой части правила, то вызов считается успешным и функция возвращает индекс очередного (незакрытого) символа входной строки. При отсутствии хотя бы одного такого правила вызов считается неудачным и функция возвращает ошибку.

Требование однозначности выбора предполагает, что в каждом случае будет найдена только одна подходящая правая часть правила. Отсутствие

таковой представляет тот случай в работе алгоритма, когда вышележащий вызов «идет не по тому пути» и просматривает «не ту» правую часть. Поэтому возвраты с отрицательным результатом вполне обоснованы. Особый вопрос – аннулирующие правила с пустой правой частью – они подходят всегда. Логично предположить, что их применение должно иметь более низкий приоритет, когда другие правила с непустой правой частью дали отрицательный результат. В приведенном примере программы этого можно достигнуть, помещая аннулирующие правила в конец группы правил с одинаковой левой частью.

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

E :: E + T или
E :: X...
X :: E...

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

II. Постановка задачи

1. Для заданного варианта синтаксиса:
- ✓ разработать КС-грамматику языка;
- ✓ разработать множество выбирающих символов;
- ✓ разработать и реализовать программу распознавателя основе метода рекурсивного спуска без возвратов.

2. Выполнить то же самое на основе общего метода рекурсивного спуска с возвратами.

III. Варианты

1. Паскаль – грамматика выражений для арифметических операций, условных выражений, присваивания и скобок.

2. Паскаль – грамматика выражений для сравнения, логических операций, присваивания и скобок.

3. Паскаль – грамматика для определения с помощью **type** новых типов данных и определения, переменных на их основе.

4. Паскаль – грамматика определений простых переменных и указателей.

5. Паскаль – грамматика определения массивов и массивов указателей.

6. Паскаль – грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;** **if** (с else и без него), и блочных скобок.

7. Паскаль – грамматика арифметических выражений со скобками. операторов **<выражение>;** и **<ид>=<выражение>;**. **case** и блочных скобок.

8. Паскаль – грамматика арифметических выражений со скобками. операторов **<выражение>;** и **<ид>=<выражение>;**. **for,** и блочных скобок.

9. Паскаль – грамматика арифметических выражений со скобками. операторов **<выражение>;** и **<ид>=<выражение>;**, **while** и блочных скобок.

10.Паскаль – грамматика арифметических выражений со скобками. операторов **<выражение>;** и **<ид>=<выражение>;**. **repeat-until** и блочных скобок.

11.Паскаль – грамматика структурированного типа **record** и контекстное определение переменных на его основе.

12.Паскаль – грамматика структурированного типа **record** и контекстное определение переменных и массивов на его основе.

13.Паскаль – грамматика определения функции вида **i(,i,i,i...){ E=i;E=i;E; }** Программа представляет собой последовательность определений функций. Выражение может содержать вызов функции вида **i(E,E,E,...).**

14.Паскаль – грамматика контекстного определения переменных и определения функций.

15.Паскаль – грамматика контекстного определения переменных и определения процедур.

IV. Содержание отчета

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.
2. Постановка задачи
3. Описание КС-грамматики входного языка.
4. Описание множества выбирающих символов(для первого задания)
5. Описание алгоритма работы сканера (в соответствии с вариантом задания).
6. Текст программы (оформляется после выполнения программы на ЭВМ).
7. Выводы по проделанной работе.

V. Методические указания

1. Применение метода рекурсивного спуска на содержательном уровне.

Для начала рассмотрим. как работает метод на достаточно простой грамматике, построив для нее формальными методами множества выбирающих символов.

Z :: N# SEL(Z::N#) ={a}
N :: UM SEL(N::UM) = {a}
M :: +UM SEL(M::+UM) = {+}
M :: ε SEL(M::ε) ={#,|}
U :: aSK SEL(U::aSK) = {a}
S ::.aS SEL(S::aS) = {a}
S :: ε SEL(S::ε) ={|,+,|,#}
K :: |N| SEL(K::|N|) = {|}
K :: εSEL(K:: ε) = {#,+,|}

Грамматика продуцирует цепочки вида **aaa|a+a|+aa+aaa|a+a|a||**, состоящие из идентификаторов, построенных из символов **a**, соединенных в цепочки символом '+', возможно с квадратными скобками, в которых могут быть аналогичные цепочки. Текст распознавателя сначала напишем, формально следуя этой грамматике:

В распознаватель для предыдущего примера можно включить два цикла – распознавание последовательности символов идентификатора и распознавание цепочки идентификаторов с возможными скобками, разделенный знаками '+'. Тогда в программе окажется единственный рекурсивно вызываемый распознаватель для нетерминала N, от которого нельзя отказаться, поскольку он определяет рекурсивную вложенность синтаксических конструкций.

```
//------------------------------ RecDown1.cpp ----------------------------------------
---------
// Z :: N#     SEL(Z::N#)={a}
void Z(){
        printf("Z::%s\n",&s[i]);
        if (s[i]!='a') throw "error a(1)";
        N();
        if (s[i]=='#') i++;
        else throw "error #";
        }
// Распознаватель для N с двумя циклами
void N(){
        printf("N::%s\n",&s[i]);
        while(1){                                   // Внешний цикл для повторения
+...+
                if (s[i]!='a') throw "error a(2)";
                while(s[i]=='a') i++;                       // Цикл для идентификатора
                if (s[i]=='['){                             // Необязательные [N]
                        i++;                        // Пропуск терминала [
                        N();
                        if (s[i]!=']') throw "error ]";     // Проверка на терминал ]
```

```
                i++;                        // Пропуск терминала [
                }
            if (s[i]=='+') { i++; continue; }       // Продолжение списка -
внешний цикл
            if (s[i]==']') { return; }              // Окончание списка - выход
            if (s[i]=='#') { return; }
            throw "error N";
            }}
void main(){
    try{
    Z(); if (s[i]==0) puts("success");
    } catch(char *ss){ puts(ss); puts(&s[i]); }}
```

Приведенный пример – это крайний случай. Возможны и промежуточные варианты, когда распознаватель частично следует структуре правил исходной грамматики, а частично распознает синтаксис «содержательно».

2. Частичный отказ от проверки выбирающих символов в некоторых группах правил.

Сказанное поясним на примере традиционной грамматики арифметических действий со скобками.

```
Z :: E#      SEL(Z::E#)={a,c,(}
E :: TM      SEL(E::TM)= {a,c,(}
M :: -TM
M :: +TM
M :: ε       SEL(M::ε)= {#,),|}
T :: FG      SEL(T::FG)= {a,c,(}
G :: *FG
G :: /FG
G :: ε SEL(G::ε)= ={+,-,#,),|}
F :: aX
F :: c
F :: (E)
X :: ε
X :: |E|
```

Аннулирующие правила для завершения цепочек операций сложения/вычитания и умножения/деления имеют выбирающими символами большинство терминальных символов. Достаточно сказать, что для группы правил умножения/деления с общим нетерминалом **G** символами, сигнализирующими об ошибке (не входящими в множества выбирающих) являются **a,c,(,|**. Для группы правил сложения/вычитания к ним добавляются * и /. На самом деле появление этих символов в таком

контексте соответствует ошибкам вида **a+a[]()**. При отсутствии контроля на уровне операций сложения и умножения ошибки такого рода будут обнаружены правилами более высокого уровня, например, **Z :: E#** или **F :: (E)**.

Программа, реализованная методом рекурсивного спуска, будет содержать распознающие процедуры для нетерминалов **E** и **T**, содержащие явные циклы для цепочек арифметических операций, в которых продолжение цикла обусловлено обнаружением терминала соответствующей операции (например, сложения или вычитания), а окончание цикла – при **любом другом**.

```
extern void F(),G(),T(),E(),Z();
void E() {                          // Цикл распознавания цепочки T+T+...+T
      T();
      while (if (s[i]=='+' || s[i]=='-')      // Проверка только на символ
продолжения
            { i++; T(); }
      }
void T() {                          // Цикл распознавания цепочки F*F*...*F
      F();
      while (if (s[i]=='*' || s[i]=='/') // Проверка только на символ
продолжения
            { i++; F(); }
      }
void F(){
      switch(s[i]){
case 'c':     i++; break;
case '(':     i++; E();             // Рекурсивный вызов анализатора для (E)
            if (s[i]==')') i++;
            else throw "error )";
            break;
case 'i':     i++;
            if (s[i]=='[')
                  { i++; E();  // Рекурсивный вызов анализатора для a[E]
                  if (s[i]==']') i++;
                  else throw "error )"; }
            break;
      }}
void Z(){
      E(); if (s[i]!='#') throe "error #";
      else i++;
      }
```

3. Просмотр вперед на более чем один символ.

Рассмотрим, например, такое расширение грамматики арифметических выражений оператором присваивания:

O :: a=E; | E; | a[E]=E;

В данной грамматике все правила имеют один и тот же выбирающий символ **a**. Т.е. по первому символу нельзя определить, является ли входная строка просто выражением, либо она содержит присваивание. В рекурсивном же спуске это можно сделать, если предусмотреть возврат к началу строки, если в начале ее не будет обнаружено присваивания.

```
void O() {
      int k=i;                              // Запомнить начало строки
      if (s[i]=='a'){
            i++;                    // пропуск a
            if (s[i]=='['){               // необязательный [E]
                  i++;
                  E();
                  if (s[i]!=']') throw "error ]";
                  i++;
                  }
            if (s[i]=='=')                // Для присваиваний – a=E или a[E]=E
                  { i++; E(); } // анализ выражения после =
            else                    // для обычного выражения – возврат в начало
                  { i=k; E(); }           // и повторный анализ строки
            if (s[i]!=';') throw "error ;";     // проверка на ограничитель - ;
            i++;
            }}
```

Лабораторная работа № 5

Тема: Нисходящий синтаксический анализ без возвратов

Цель. Изучение основных понятий теории регулярных грамматик, ознакомление с назначением и принципами работы лексических анализаторов (сканеров), получение практических навыков построения сканера на примере заданного простейшего входного языка.

I. Краткие теоретические сведения

Отношения между символами в формальных грамматиках

Формальные методы синтаксического анализа используют определения множеств символов **FIRST, LAST, FOLLOW**, вводимые для нетерминалов.

Множество **FIRST** нетерминального символа **U** или **FIST(U)** множество символов (терминальных и нетерминальных), с которых может начинаться цепочка, выводимая из **U**. Содержательно, для построения этого множества следует просмотреть все возможные выводы, производимые из **U**, само собой, ограничивая возможные зацикливания в подстановках. Формальный алгоритм естественным образом рекурсивен: для заданного нетерминала берутся все правила, в которых он находится в левой части. Если правая часть правила начинается с терминала, то он просто добавляется в множество, если с нетерминала (правило вида U::=S...), то для него алгоритм вызывается рекурсивно, после чего полученный результат (множество **FIST(S)**) добавляется в исходное множество (Заметим, что попутно составляются и запоминаются множества для всех "транзитных" нетерминалов). Единственный нюанс заключается в возможном наличии аннулирующих правил с пустой правой частью для них вместо множества **FISRT** строится множество **FOLLOW** для нетерминала левой части (см.ниже).

Множество **LAST** является строится аналогично **FIRST**, но только включает последние символы, завершающие все возможные цепочки, выводимые из заданного нетерминала.

Множество **FOLLOW** связано с использованием в формальной грамматике аннулирующих правил, которые заменяют нетерминал левой части на пустую цепочку, завершая, тем самым, цикл повторения, либо исключая необязательные элементы. В этом случае принципиально важным является ответ на вопрос, в каком окружении (контексте) может находиться такой нетерминал, точнее, какой символ находится "вслед за ним". Множество **FOLLOW(U)** содержит терминальные символы, которые могут встречаться "вслед за..." нетерминалом **U** во всех промежуточных цепочках, выводимых в данной ФГ. Формальный алгоритм перебора таких цепочек естественным образом рекурсивен. Просматриваются правые части всех правил и определяются все включения нетерминала **U**. Для каждого из них возможны варианты:

– если нетерминал последний в правой части (правило вида X::=....U), то множество символов, "следующих за U" включает в себя множество символов, "следующих за X", то есть за нетерминалом левой части. рекурсивно вызывается функция для построения множества **FOLLOW(X)** и ее результат добавляется к текущему множеству для **U**;

– если вслед за нетерминалом в правой части следует терминал, то он просто добавляется в множество;

– если вслед за нетерминалом в правой части следует терминал, то для него определяется множество **FIRST**, которое добавляется к текущему множеству для **U**.

С большой долей успеха можно неформально определять множество **FOLLOW**, учитывая, что повторяющиеся последовательности, завершаемые аннулирующими нетерминалами, как правило, включены в определенные контексты (окружения терминальных символов), которые являются естественными (контекстными) ограничителями этих цепочек. Рассмотрим это на примере формальной грамматики арифметических выражений.

O ::= E;
E ::= TM
M ::= e | +TM | -TM
T ::= FG
G ::= e | *FG | /FG

F ::= a | (E)

Нетерминал M в левой части аннулирующего правила используется для завершения цепочки операций сложения и вычитания, выводимой из E. Само же выражение встречается всего в двух контекстах: как выражение в скобках и как выражение, ограниченное символом "точка с запятой". Эти два символа и составляют множество **FOLLOW.**

E > T+T-TM > T+T-T (в контексте "E;" и "(E)")
FOLLOW(M)={),; }

Несколько сложнее обстоит дело с нетерминалом G, ограничивающем цепочку операций умножения и деления. Поскольку он используется для завершения цепочек, выводимых из T, а этот нетерминал встречается в уже рассмотренной выше цепочке в контексте операций сложения/вычитания, то операции +,- являются контекстными ограничителями для M, наряду с уже известными:

E > T+T-T > T+F*FG-T
E > T+T-TM > T+T-FG (в контексте "E;" и "(E)")
FOLLOW(G)={),;,+,- }

Магазинные автоматы

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. Если обычный КА дополнить стеком то есть магазинной памятью, то полученная формальная система может служить базой для создания синтаксического анализатора. Такой КА со стеком называется МАГАЗИННЫМ АВТОМАТОМ (МА).

Стек реализует память работающий по принципу "последним прибыл первым обслужен" (LIFO).

Основные операции связанные со стеком:

- операция записи в стек заданной строки **push** ("..."). Строка помещается в стек, начиная с последнего символа, то есть первый символ оказывается в вершине стека;
- операция выталкивания символа из стека **pop()**;
- операция проверки "стек пуст" **empty()**.

Дополнительные операции:

- операция проверки наличия очередного символа в строке "строка пуста";
- операция перехода к следующему символу в строке **next ()** ;
- операция сообщения об ошибке **error();**
- операция сообщения об успешном выполнении анализа **success()**.

Для МАГАЗИННОГО АВТОМАТА необходимо определить:

- входная строка I, текущий символ в строке (I)
- магазин (стек) M, символ в вершине стека (M)
- S = {Si}, множество состояний КА;

Описание поведения МА заключается в том, что для каждой комбинации состояния КА, входного символа и символа в вершине стека - (Si, (I), (M)) 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.

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:

– в исходном состоянии в стек записывается начальный нетерминал,

– 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);

– если текущий символ строки и символ в вершине стека не совпадают, и при этом символ в стеке является терминальным, то это синтаксическая ошибка

– 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, тогда в стеке левая часть правила заменяется на правую;

– распознавание выполнено успешно, когда и стек и строка одновременно становятся пустыми. в противном случае - синтаксическая ошибка;

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(),next()	error()	error()
b	error()	pop(),next()	error()
A	pop(),push(aA)	pop(),push(b)	error()
S	pop(),push(aA)	pop(),push(bSb)	error()
#	error()	error()	success( )

Грамматики класса Q (Q-грамматики)

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

A ::= e, где e - 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 ::= e

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.

Построить множество символов, следующих за данным нетерминалом, можно для простых грамматик и неформально, рассматривая возможные варианты порождения цепочек.

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 ::= e **FOLLOW(A)={b,#}**

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

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

Грамматики класса LL(1)

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 теминального 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, что было определено выше как множество **FIRST**

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	**FIRST(T)**	**a,(**
M ::=-E	**-**	**-**
M ::=+E	**+**	**+**
M ::=e	**FOLLOW(M)**	**),#**
T ::=FG	**FIRST(F)**	**a,(**
G ::=*T	*****	*****
G ::=/T	**/**	**/**
G ::=e	**FOLLOW(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 FIRST(T):

1. T > FG > a, выбирающий символ a
2. T > FG > (E), выбирающий символ (

Правило G ::= y, способ построения FOLLOW(G):

1. E > TM > FGM > FG+E, символ +
2. E > TM > FGM > FG-E, символ -
3. F > (E) > TM > (FGM) > (FG), символ)
4. E# > TM# > FGM# > FG#, символ #

II. Постановка задачи

1. Для заданного варианта синтаксиса:
   разработать LL(1)-грамматику;
   Разработать таблицу переходов на основе выбирающих множеств;

Разработать и реализовать программу распознавателя LL(1) на основе метода рекурсивного спуска;

при помощи программы-распознавателя LL(1) проверить работоспособность грамматики. Формальная проверка заключается в том, что множества выбирающих символов должны быть непересекающимися. Содержательная проверка состоит в подтверждении того, что данная грамматика **действительно реализует требуемый синтаксис** (вложенность, приоритеты различных синтаксических единиц обеспечивается данной грамматикой);

2. Выполнить то же самое на основе магазинного автомата.

III. Варианты

1. Си-грамматика выражений для арифметических операций, условных выражений, присваивания и скобок.

2. Си-грамматика выражений для сравнения, логических операций, присваивания и скобок.

3. Си-грамматика выражений для поразрядных операций, присваивания и скобок.

4. Си-грамматика определений простых переменных указателей и ссылок с инициализаторами.

5. Си-грамматика определения массивов и массивов указателей с инициализаторами.

6. Си-грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;** **if** (с else и без него), и блочных скобок.

7. Си-грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **switch** (с else и без него) и блочных скобок.

8. Си-грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **for,** и блочных скобок.

9. Си-грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **while** и блочных скобок.

10.Си-грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **do-whille** и блочных скобок.

11.Определение структурированного типа **struct** и контекстное определение переменных на его основе: **struct a{...} a,a[c]**.

12.Определение структурированного типа **struct** и контекстное определение переменных на его основе: **struct a{...} a,a[c],*a,**a**.

13.Си-грамматика определения функции вида **i(,i,i,i…){ E=i;E=i;E; }** Программа представляет собой последовательность определений функций. Выражение может содержать вызов функции вида **i(E,E,E,…).**

14.Си-грамматика контекстного определения переменных и определения функций с использованием передачи по ссылке и по значению.

15.Си-грамматика контекстного определения переменных и определения функций – процедур(тип **void**) с использованием передачи по значению.

IV. Содержание отчета

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.
2. Постановка задачи
3. Описание КС-грамматики входного языка и множеств выбирающих символов.
4. Таблица отношений предшествования
5. Описание алгоритма работы сканера (в соответствии с вариантом задания).
6. Текст программы (оформляется после выполнения программы на ЭВМ).
7. Выводы по проделанной работе.

V. Методические указания

1. Метод рекурсивного спуска для LL(1) грамматики

Для начала рассмотрим, как работает метод рекурсивного спуска на достаточно простой грамматике, построив для нее формальными методами множества выбирающих символов.

Z :: N# SEL(Z::N#) = FIRST(N)=FIRST(U)={a}
N :: UM SEL(N::UM) = FIRST(U)={a}
M :: +UM SEL(M::+UM) = {+}
M :: ε SEL(M::ε) = FOLLOW(M)=FOLLOW(N)={#,]}
U :: aSK SEL(U::aSK) = {a}
S :: aS SEL(S::aS) = {a}
S :: ε SEL(S::ε) = FOLLOW(S)= {],+,|,#}
K :: [N] SEL(K::[N]) = {[}
K :: ε SEL(K:: ε) = FOLLOW(K)=FOLLOW(U)={#,+,|}

Грамматика продуцирует цепочки вида **aaa[a+a]+aa+aaa[a+a[a]]**, состоящие из идентификаторов, построенных из символов **a**, соединенных в цепочки символом '+', возможно с квадратными скобками, в которых могут быть аналогичные цепочки. Текст распознавателя сначала напишем, формально следуя этой грамматике:

```
//---------------------------------- RecDown1.cpp ----------------------------------
---------------
char s[100]="aaa[aa+aa[a]]+aa#";
int i=0;
extern void N().M(),U(),S(),K();
// Z :: N#      SEL(Z::N#)={a}
void Z(){
 printf("Z::%s\n",&s[i]);
 if (s[i]!='a') throw "error a(1)";
 N();
 if (s[i]=='#') i++;          // Проверка и пропуск символа в правой части
правила
 else throw "error #";
 }
//N :: UM      SEL(N::UM)={a}
void N(){
 printf("N::%s\n",&s[i]);
 if (s[i]!='a') throw("error a(2)");
 U(); M();
 }
//M :: +UM  SEL(M::+UM) = {+}
//M :: eSEL(M::e) ={#,]}
void M(){
 printf("M::%s\n",&s[i]);
 switch(s[i]){
 case '#':
 case ']':      return;          // Аннулирующее правило
 case '+': i++;          // пропуск символа +
        U(); M(); break;
 default:      throw "error [,a";
}}
//U :: aSK SEL(U::aSK)   = {a}
void U(){
 printf("U::%s\n",&s[i]);
 if (s[i]!='a') throw "error a(3)";
 i++;              // пропуск символа a
 S(); K();
 }
//S :: aS  SEL(S::aS)      = {a}
//S :: e   SEL(S::e)  ={[,+,],#}
void S(){
 printf("S::%s\n",&s[i]);
 if (s[i]=='a') { i++; S(); }
 else return;          // аннулирующее правило S::e для всех остальных
 }
```

```
//K :: [N]      SEL(K::[N]) = {[}
//K :: e        SEL(K:: e) = {#,+,]}
void K(){
 printf("K::%s\n",&s[i]):
 switch(s[i]){
 case 'a': throw "error a(4)";
 case '[':      i++;          // пропуск символа [
        N():
        if (s[i]==']') i++;
        else throw "error 4";
        break;                // ошибка - ожидается символ ]
 default:      return:              // аннулирующее правило по ],+,#
 }}
void main(){
 try{
 Z(): if (s[i]==0) puts("success");
 } catch(char *ss){ puts(ss); puts(&s[i]); } }
```

2. Применение магазинного автомата для частного случая S грамматики

Рассмотрим применение стека на достаточно простой S грамматике

S : aA
S : bSb
A : aA
A : b

Текст распознавателя сначала напишем, формально следуя этой грамматике:

```
//--------------------------------- SGram1.cpp ---------------------------------------
----------

#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <dos.h>

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

void    push(str)
char    *str;
{
char    *q;
```

```
q=str;
while (*q !='\0') q++;
q--;
while (q >=str)
 {
 m[j++] = *q--;
 }
m[j]='\0';
}

void main()
{
printf("Строка:");
gets(s);
i=0;
j=1;
m[0]='S';

while (1)
 {
clrscr();
gotoxy(1,10);
 if ((s[i]=='\0') && (j==0))
        {
        printf("Разбор закончен\n");
        break;
        }
 if ((s[i]=='\0') || (j==0))
        {
        printf("Ошибка разбора\n");
        break;
        }
 if ((s[i]==m[j-1]))
        {
        printf("Уничтожение %c\n",s[i]);
        i++;
        j--;
        m[j]='\0';
        }
 else
        {
        switch(m[j-1])
                {
 case  'a':
 case  'b':
```

```
                printf("Ошибка \n");
                break;
  case  'A':    j--;
                if (s[i]=='a')
                      {
                      printf("Правило A:aA\n");
                      push("aA");
                      }
                else
                      {
                      printf("Правило A:b\n");
                      push("b");
                      }
                break;
  case  'S':    j--;
                if (s[i]=='a')
                      {
                      printf("Правило S:aA\n");
                      push("aA");
                      }
                else
                      {
                      printf("Правило S:bSb\n");
                      push("bSb");
                      }
                break;
                }
      }
printf("Строка:%s\nСтек  :%s\n".s+i.m);
sleep(2);     } }
```

Лабораторная работа №6

Тема: Восходящий синтаксический анализ без возвратов

Цель. Цель работы состоит в составлении программы, производящей синтаксический анализ формальной грамматики на основе методов восходящего разбора.

I. Краткие теоретические сведения

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. Т.е. синтаксическое дерево строится снизу-вверх: в текущем множества «незакрытых» вершин ищется подмножество потомков и над ними «надстраивается» вершина-предок. При этом обход вершин и аналогичный просмотр цепочки символов происходит слева-направо. Первая полная правая часть правила называется **основой.**

T = {a,*,+,#} N = {Z,U,B,A}
Z :: U# // 1
U :: U+B // 2
U :: B // 3
B :: *B // 4
B :: A // 5
A :: Aa // 6
A :: a // 7

Приведенная грамматика продуцирует цепочки вида ****aa+aa+*aaa.** Из правил грамматики видно, что первая замена производится по правилу **A::a**, иначе в цепочке просто нет необходимых нетерминалов для подстановки. Затем в нетерминалу **A** справа «присоединяются» терминалы **a** из входной строки, а уже затем – символы * слева.

Теперь нужно обсудить, как будет выглядеть распознаватель и каковы будут принципы его работы. Во-первых, по аналогии с нисходящим разбором можно предположить, что для обнаружения **основы** достаточно пары символов – последнего символа основы и следующего символа строки (в нашем примере это сочетание **aa**). Т.е. для каждой пары символов грамматики однозначно можно сформулировать утверждение, является ли эта пара концом основы или нет. Опять-таки это связано с глубиной просмотра вперед входной строки – она равна 1.

Во-вторых, распознавателю необходим стек, и для него требуется определить функциональное назначение. Поскольку просмотр строки в поисках основы требует сохранения пройденных символов, резонно это

делать в стеке. Тогда замена правой части правила (**основы**) на левую будет также производиться в вершине стека. Сам стек будет хранить «недосвернутую» просмотренную часть цепочки, для которой еще не накоплена основа.

Теперь можно сформулировать основные принципы восходящего разбора с использованием магазинного автомата (МА), именуемого также методом «**свертка-перенос**»:

Первоначально в стек помещается первый символ входной строки, а второй становится текущим;

МА выполняет два основных действия: **перенос** (сдвиг - **shift**) очередного символа из входной строки в стек (с переходом к следующему), **PUSH((I), NEXT(I)** в соответствии с принятыми для МА обозначениями;

Поиск правила, правая часть которого хранится в стеке и замена ее на левую – **свертка (reduce);**

Решение, какое из действий – перенос или свертка выполняется на данном шаге, принимается на основе анализа пары символов – символа в вершине стека и очередного символа входной строки. Свертка соответствует наличию в стеке **основы**, при ее отсутствии выполняется перенос. Управляющими данными МА является таблица, содержащая для каждой пары символов грамматики указание на выполняемое действие (свертка, перенос или недопустимое сочетание -ошибка) и сами правила грамматики.

Положительным результатом работы МА будет наличие начального нетерминала грамматики в стеке при пустой входной строке.

Как следует из описания, алгоритм не строит синтаксическое дерево, а производит его обход «снизу-вверх» и «слева-направо». В соответствии с этим, грамматика, допускающая распознавание подобным методом, называется **LR(1)** – (**left** –просмотр входной строки слева направо, **right**-правая подстановка – замена правой части на левую, восходящий разбор, **1**-глубина просмотра входной строки для принятия решения).

Для формального задания поведения МА необходимо задать таблицу его действий на полный набор 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 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. На самом деле, таблицу можно упростить еще больше. При выполнении свертки группу правил с одинаковой правой частью (C1,C2.C3.C4) можно однозначно определить по сочетанию последнего (левого) символа правой части и текущего символа входной строки. Поэтому в таблице достаточно оставить только признак свертки.

Понятно, что содержательное заполнение управляющей таблицы можно выполнить только для простых грамматик. Следующим шагом является разработка формальных методов заполнения таких таблиц. которые основаны на анализе свойств и соотношений между символами, имеющим место в формальной грамматике.

Проблемы восходящего разбора. Конфликты свертка-свертка, свертка-перенос

Восходящий разбор противоположен нисходящему не только по формальным моментам. но и по тому, что проблемы в каждом из них возникают там, где противоположный метод реализуется безболезненно. Основное достоинство нисходящего разбора состоит в «программировании» будущего (ожидаемого) синтаксиса выбирающими элементами языка, в частности, ключевыми словами. Поэтому **глобальный контекст** (окружение) каждой анализируемой конструкции оказывается заранее известным. Это означает, что синтаксически правильные сочетания одних и тех же символов в разных контекстах могут быть заданы по-разному. Например, в одной конструкции допустимы сочетания вида **a+a+a,** а в другой – **a++a++a.** Существенным недостатком нисходящего разбора является невозможность распознавания вариантов синтаксиса, имеющих синтаксически общее «начало» но различные варианты продолжения. Здесь можно процитировать пример совмещения синтаксиса оператора присваивания и обычного выражения:

O :: a=E; |E; | a|E|=E;

В методе рекурсивного спуска данная проблема решается неформальным методом путем повторного перечитывания и анализа части входной строки, соответствующей части арифметического выражения при отсутствии знака '=' после идентификатора с возможными индексными скобками.

В методе «свертка-перенос» проблема «общего начала», наоборот, отсутствует. При наличии единого завершающего синтаксиса несколько синтаксически различных правых частей могут быть свернуты к одному и тому же нетерминалу. То есть свертка указанных выше правил производится в рамках стандартного алгоритма восходящего разбора. Наоборот, возникает проблема **глобального контекста**. Одно и то же сочетание символов в разных контекстах может быть как самостоятельной конструкцией, так и составной частью конструкции более высокого уровня. Соответственно, в разном контексте оно будет требовать либо свертки – в первом случае, либо переноса – во втором.

Перейдем теперь к более конкретному обсуждению отмеченных особенностей. Восходящий разбор влияет на «внешний вид» грамматики. Во-первых, здесь отсутствуют аннулирующие правила, поэтому повторения задаются в виде явной рекурсии, а нулевое число повторений выносится в отдельное правило. В качестве примера рассмотрим грамматику арифметических выражений, включающую в себя вызов функций с произвольным (в том числе и пустым) списком параметров.

E :: E-T | E+T | T
T :: T*F | T/F | F
F :: a() | a(E)| a(LE) | (E) | a | a[E]
L :: LE, | E,

В приводимом уже примере пустая последовательность фактических параметров в вызове функции задана отдельным правилом, отдельное правило вводится для единственного параметра, а также группа правил для списка.

F ::= a | a()| a(E) | a(LE)
L ::= E,| LE,

Следствием неоднозначности восходящего разбора являются конфликты, которые возникают, если пара соседних символов находится более чем в одном отношении, даже если эти отношения идентичны (свертка-свертка).

Допустимый конфликт **«свертка-свертка»** происходит в том случае, если правые части различных правил заканчиваются одним и тем же символом (или цепочкой символов) и приводятся к одному и тому же нетерминалу (т.е. встречаются в одной группе правил). Поскольку в момент свертки распознаватель производит поиск правила, то при последовательном их переборе необходимо размещать более длинные правила раньше более коротких. В нашем примере к таким конфликтам приводит символ **')'**, приводящий к свертке выражений нескольких видов, и символ **T** в группе правил описания цепочки операций сложения/вычитания:

E :: E-T
E :: E+T
E :: T **более короткое правило – в конце**
...
F :: a() **несколько сверток по символу)**
F :: a(E)
F:a(LE)
F:(E)
F:a
L:LE,
L:E,

Неразрешимый конфликт «свертка-свертка» происходит между правилами, имеющими одинаковую правую часть и различные нетерминалы в левой части.

Конфликт «свертка-перенос» является следствием невозможности определения глобального контекста, в котором находится просматриваемый фрагмент цепочки, то есть на уровне выбора между этими двумя действиями мы не может заглянуть «глубже в стек» и посмотреть, в окружении каких символов находится текущая пара. Устранить такой конфликт можно, если зафиксировать такой контекст «раньше», путем введения дополнительных нетерминалов, которые «будут помнить», что началась та или иная синтаксическая конструкция.

O :: for(a=E;ECE;E)O | E; | if(ECE)O | if(ECE)OeO
E :: E+T | E-T | T
C :: < | > | = | <> | <= | >=
...
F::=a | c | (E)

В данной грамматике – несколько конфликтов, причем все они касаются выражений – цепочек **E;** или **(E)**. Заголовок оператора цикла и оператор – как выражение, ограниченное «точкой с запятой» имеют одинаковый контекст (окружение), но в первом случае необходим перенос, т.к. оператор цикла еще не закончился, а во втором – свертка к оператору. Разрешить конфликт можно, «зацепив» заголовок цикла и построив цепочку правил до достижения конца оператора цикла. Аналогично обстоит дело с условным оператором:

O::=WO | E; | XO | XeO
D::=for(a=E;
U::=DECE;
W::=UE)
X::=if(ECE)

Если посмотреть на проблему с формальной стороны, то наличие конфликта говорит о существовании в грамматике минимум двух правил со следующим свойствами:

в одном правиле эти символы находятся в одной правой части, т.е. **X::...ab...** или имеется правило **X::...aB...**, причем **b∈FIRST(B)**, т.е. такие сочетания требуют переноса:

в другой правой части правила имеется сочетание **X::...Ab...**, причем **a∈LAST*(A)** или **X::...AB...**, причем **a∈LAST*(A)** и **b∈FIRST(B)**, т.е. требуется свертка.

Исключение таких конфликтов представляет собой довольно трудную задачу. В общем случае требуется вводить дополнительные нетерминалы и правила для них, чтобы «разбить» правые части правил, в которых имеется «конфликтующий» перенос, т.е. заменить **X::...aB...** на **Y::...a** и **X::YB...**

Формальные методы восходящего разбора. Простое предшествование

Для формального метода построения управляющей таблицы можно использовать простые соотношения, которые устанавливаются между символами ФГ в соответствии с имеющейся вложенностью ее правил. Для начала посмотрим, какие отношения между символами ФГ требуется установить при восходящем разборе, представляющем собой построение дерева «снизу-вверх» и «слева-направо».

Очевидно, что определяющим фактором для процесса свертки в произвольном дереве является **вложенность** поддеревьев, к которым относятся символы границы стека и входной строки (**c** и **d**). Можно сказать, что символ **c** находится «глубже» в дереве, чем символ **d**, если под глубиной понимать длину пути до общего нетерминала (вершины), из которой они оба выводятся. На первый взгляд может показаться, что соотношение «глубины» для различных пар символов невозможно установить однозначно, ведь речь идет о всех возможных деревьях, которые могут быть построены в ФГ. Но это не так. Во-первых, взаимное расположение символов в правильных предложениях более или менее строго предопределено, а во-вторых, в рассматриваемом отношении участвуют только граничные символы дерева. Они, как известно, определяются множествами **FIRST** и **LAST** для нетерминала, который является вершиной поддерева.

Построение отношения предшествования

Отношение «глубины» официально называется отношением предшествования. Затрагивает оно не все пары символов (вершин) в синтаксических деревьях, а только граничные вершины поддеревьев. Образно говоря, это отношения типа «старший сын младшего брата». Для их построения необходимо рассматривать структуру порождаемой грамматикой деревьев. Понятно, что это можно сделать и простым перебором, но в общем виде можно обойтись анализом правых частей правил формальной грамматики и с использованием множеств **FIRST** и **LAST** встречающихся нетерминальных символов.

Напомним, что множество **FIRST(U)** нетерминального символа **U** – это терминальные символы, которые могут стоять первыми в цепочках, выводимых из **U**. Аналогичным образом определяется множество $\mathbf{LAST^+(U)}$, которое включает не только терминалы, но и промежуточные нетерминалы, на которые может оканчиваться цепочка, выводимая из **U**. При отсутствии аннулирующих правил (правил с пустыми правыми частями) в ФГ, используемых в восходящих методах разбора, эти множества строятся довольно легко. На синтаксическом дереве множества **FIRST** и $\mathbf{LAST^+}$ выглядят как левая нижняя граница и правая границы множества всех деревьев с корневой вершиной **U**.

Для приведенной выше грамматики эти множества выглядят следующим образом:

FIRST(E)=FIRST(T)=FIRST(F)={(,a,c }
LAST+(F)={),a,c }
LAST+(T)={),a,c,F }
LAST+(E)={),a,c,F,T }

Итак, построение отношения предшествования начинается с перечисления все пар соседних символов правых частой правил, которые и определяют все варианты отношений смежности для границ возможных выстраиваемых на них деревьях. (Сразу отметим, что правила с одним символом в правой части являются для этой цели непродуктивными). Далее, в каждой паре возможны следующие комбинации терминальных/нетерминальных символов и продуцируемые из них элементы отношений:

1. Все пары соседних символов находятся в отношении «равенства» = или одинаковой глубины. При этом для метода «свертка-перенос» нетерминальный символ не может являться символом входной строки, поэтому пары с нетерминалом справа в построении отношения предшествования для такого метода не участвуют.

E::αUmβ → U=m

Например,

F:: a[E] → a = E, E =]

2. Для пары нетерминал-терминал правая граница поддерева, выстраиваемая на основе нетерминала (множество **LAST⁺**) находится «глубже» правого терминала, т.е.

E::αUmβ → LAST⁺(U) > m

3. Аналогичное обратное отношение выстраивается для пары терминал-нетерминал: левая нижняя граница поддерева, выстраиваемого на нетерминале «глубже» левого терминала

E::αmUβ → m < FIRST(U)

4. Наиболее сложное, но и самое «продуктивное» соотношение – два рядом стоящих нетерминала, которые производят сразу два отношения:

E::αNUβ → LAST⁺(N) > FIRST(U), N < FIRST(U)

Что «в переводе на русский язык означает»: правая граница левого поддерева «глубже» левой нижней границы правого смежного поддерева, но при этом корневая вершина левого поддерева «выше» той же самой левой нижней границы правого смежного поддерева.

Пример построения отношения предшествования

В заключение, рассмотрим построение отношения предшествования для простой грамматики.

Z::U#	**(1)**
U::U+E	**(2)**
U::E	**(3)**
E::AB[E]	**(4)**
E::AB	**(5)**

E::B (6)
A::*A (7)
A::* (8)
B::Ba (9)
B::a (10)

FIRST(U) = FIRST(E) = FIRST(A) □ FIRST(B) = {a,*}
FIRST(A) = {*}
FIRST(B) = {a}
$LAST^+(U) = E + LAST^+(E) = \{E,B,a,|\}$
$LAST^+(E) = \{|\} + B + LAST^+(B) = \{B,a,|\}$
$LAST^+(B) = \{a\}$
$LAST^+(A) = \{*\}$

Z::U# → U = #
Z::U# → $LAST^+(U)$ > # → {E,B,a,|} > #

U::U+E → $LAST^+(U)$ > + → {E,B,a,|} > +
U::U+E → + < FIRST(E) → + < {a,*}
U::U+E → U = +
E::AB|E| → | < FIRST(E) → | < {a,*}
E::AB|E| → E = |
E::AB|E| → $LAST^+(E)$ > | → {B,a,|} > |
E::AB|E| → $LAST^+(B)$ > | → a > |
E::AB|E| → B = |
E::AB → $LAST^+(A)$ > FIRST(B) → * > a
E::AB → A < FIRST(B) → A < a
A::*A → * < FIRST(A) → * < *
B::Ba → B = a
B::Ba → $LAST^+(B)$ > a → a > a

	*	a	|	|	+	#
*	$<_7$	$>_5$					
A		$>_9$	$>_4$	$>_4$	$>_2$	$>_1$
|	$<_4$	$<_4$				
|				$>_4$	$>_2$	$>_1$
+	$<_2$	$<_2$				
U					$=_2$	$=_1$
E				$=_4$	$>_2$	$>_1$
A		$<_5$				
B		$=_9$	$=_4$	$>_4$	$>_2$	$>_1$

II. Постановка задачи

1. Для заданного варианта синтаксиса:
разработать LR(1)-грамматику;
разработать таблицу действий для магазинного автомата;
Разработать программу распознавателя LR(1) на основе магазинного автомата;
. при помощи программы-распознавателя LR(1) проверить работоспособность грамматики. Формальная проверка заключается в том, что множества выбирающих символов должны быть непересекающимися. Содержательная проверка состоит в подтверждении того, что данная грамматика **действительно реализует требуемый синтаксис** (вложенность, приоритеты различных синтаксических единиц обеспечивается данной грамматикой);
оформить отчет, в котором привести описание грамматики (группы правил, их назначение, множества FIRST и LAST* нетерминалов левой части), пример распознавания синтаксической ошибки (состояние распознавателя в момент ее обнаружения и возможное на основе его анализа сообщение об ошибке)., привести пример синтаксического дерева для правильного предложения.

2. Для заданного варианта синтаксиса:
разработать LR(1)-грамматику;
разработать таблицу отношений предшествования:
Разработать программу распознавателя LR(1) на основе магазинного автомата;
. при помощи программы-распознавателя LR(1) проверить работоспособность грамматики. Формальная проверка заключается в том, что множества выбирающих символов должны быть непересекающимися. Содержательная проверка состоит в подтверждении того, что данная грамматика **действительно реализует требуемый синтаксис** (вложенность, приоритеты различных синтаксических единиц обеспечивается данной грамматикой);
оформить отчет, в котором привести описание грамматики (группы правил, их назначение, множества FIRST и LAST* нетерминалов левой части), таблицу отношения предшествования, пример распознавания синтаксической ошибки (состояние распознавателя в момент ее обнаружения и возможное на основе его анализа сообщение об ошибке). Построить все элементы отношения предшествования, выводимые из двух заданных правил (с отметкой в таблице), привести пример синтаксического дерева для правильного предложения.

III. Варианты

1. Си-грамматика выражений для арифметических операций, условных выражений, присваивания и скобок.

2. Си-грамматика выражений для сравнения, логических операций, присваивания и скобок.

3. Си-грамматика выражений для поразрядных операций, присваивания и скобок.

4. Си-грамматика определений простых переменных указателей и ссылок с инициализаторами.

5. Си-грамматика определения массивов и массивов указателей с инициализаторами.

6. Си-грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;** **if** (с else и без него), и блочных скобок.

7. Си-грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **switch** (с else и без него) и блочных скобок.

8. Си-грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **for**, и блочных скобок.

9. Си-грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **while** и блочных скобок.

10. Си-грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **do-whille** и блочных скобок.

11. Определение структурированного типа **struct** и контекстное определение переменных на его основе: **struct a{...} a,a[c]**.

12. Определение структурированного типа **struct** и контекстное определение переменных на его основе: **struct a{...} a,a[c],*a,**a**.

13. Си-грамматика определения функции вида **i(,i,i,i...){ E=i;E=i;E; }** Программа представляет собой последовательность определений функций. Выражение может содержать вызов функции вида **i(E,E,E,...)**.

14. Си-грамматика контекстного определения переменных и определения функций с использованием передачи по ссылке и значению.

15. Си-грамматика контекстного определения переменных и определения функций – процедур(тип **void**) с использованием передачи по значению.

IV. Содержание отчета

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.
2. Постановка задачи
3. Описание КС-грамматики входного языка.
4. Таблица отношений предшествования(для задания 2).
5. Таблица действий магазинного автомата.
6. Описание алгоритма работы сканера.
7. Текст программы (оформляется после выполнения программы на ЭВМ).
8. Выводы по проделанной работе.

V. Методические указания

1. Синтаксический анализатор арифметических формул:

Рассматривается грамматика:

S : a
S : S*S | S+S

Матрица предшествования:

	a	+	*	#
a	-	>	>	>
+	<	>	<	>
*	<	>	>	>
#	<	<	<	+

Построенный частичный вывод x будем хранить в стеке:

```
  /* Коды лексем и метасимволов */
void yylex()
{
switch(c)
{
case 'a': return(0):
case '+': return(1):
case '*': return(2):
case '#': return(3):
default: printf("%s: ".c); printf ("%s\n"."ошибочный символ"). exit(1):
}
/* матрица предшествования */
static chat MM[][] = (" ->> >","<><>". "<>>>"."<<<+"):

main() {
 int c, d, b:
  stinit(): stpush(0):
```

```
    do {
      c = yylex(); b=sttop():
      if (MM[b][c]== '+') success();
      if (MM[b][c]== '-') error():
    while ( MM[b][c]== '>') {
       do {
         d = stpop();
       } while ( MM[b][c]!= '<');
      }
      stpush(c);
    } while(c!=0);
}
/* Реализация стека терминалов */
#define MAXSTK  100
static char stack[MAXSTK], stptr;
void stinit()  { stptr = 0; }
void stpush(char e) {
  if ( stptr== MAXSTK) error();
  stack[++stptr] = e;
}
char sttop() { if (stptr=0) error(): return(stack[stptr]): }
char stpop() { if (stptr=0) error(); return(stack[stptr--]); }
void error() { printf ("%s\n", "отказ"); exit(1); }
void success () { printf ("%s\n", "успешно"); exit(1); }
```

Лабораторная работа №7

Тема: Семантический анализ

Цель. Цель лабораторной работы является изучения семантики и создания программы, производящей семантический анализ формальной грамматики.

I. Краткие теоретические сведения

Особенности семантики и семантического анализа

Следующий шаг анализа текста программы – семантический, существенно отличается от двух предыдущих – лексического и синтаксического. И дело не столько в том, что фаза семантического анализа реализуется не формальными, а **содержательными** методами (т.е. на данный момент нет универсальных математических моделей и формальных средств описания «смысла» программы). Лексический и синтаксический анализ имеют дело со **структурными,** т.е. внешними, текстовыми конструкциями языка. Семантика же, ориентированная на содержательную интерпретацию, имеет дело с внутренним представлением «смысла» объектов, описанных в программе. Для любого, имеющего опыт практического программирования, ясно, что формальные конструкции языка дают описание свойств и действий над **внутренними объектами**, с которыми имеет дело программа. Для начала перечислим все, что их касается и лежит на поверхности:

большинство объектов являются **именованными.** Имя объекта позволяет его идентифицировать, существуют различные области действия имен, соглашения об именах, различные умолчания и т.п.. Все это относится к семантике;

виды, сложность и набор характеристик объектов различаются в разных языках программирования и сильно зависят от области приложения языка (в этом смысле семантика языков программирования более разнообразна, нежели синтаксис и лексика). Например, классический Си, ориентированный на максимальное приближение к архитектуре компьютера, работает с такими объектами, как **типы данных, переменные, функции**. Все они имеют различные свойства и характеристики. Например, переменная характеризуется именем, типом данных, размерностью, областью действия, временем жизни, текущим значением;

объекты связаны между собой (ссылаются друг на друга). В том же Си переменная ссылается на описание того типа данных, к которому она относится, далее производный тип данных ссылается на базовый и т.п.. Можно сказать, что семантика программы во внутреннем представлении выглядит как система взаимосвязанных объектов;

внутреннее представление семантики программы не совсем удачно называется **семантическими таблицами.** На самом деле

структура данных, соответствующая представлению семантики, может быть любой. Термин **«таблицы»** говорит о том, что имеются множества объектов различных типов, для каждого из которых заведена отдельная таблица, но нельзя забывать, что элементы различных таблиц связаны между собой.

Семантика программы – внутренняя модель системы именованных объектов, с которыми работает программа, с описанием их свойств и характеристик.

Теперь, когда у нас есть представление о синтаксической фазе, можно оценить ее центральную роль в организации процесса трансляции. Только на уровне синтаксиса текст программы представляет собой **единое структурное целое** – любое предложение языка сводится к единственному начальному нетерминалу **Z**. Лексические единицы, как известно, вообще независимы друг от друга. Семантика программы тоже не обладает структурной целостностью и представлена фрагментарно, но при этом связана с синтаксисом следующим образом:

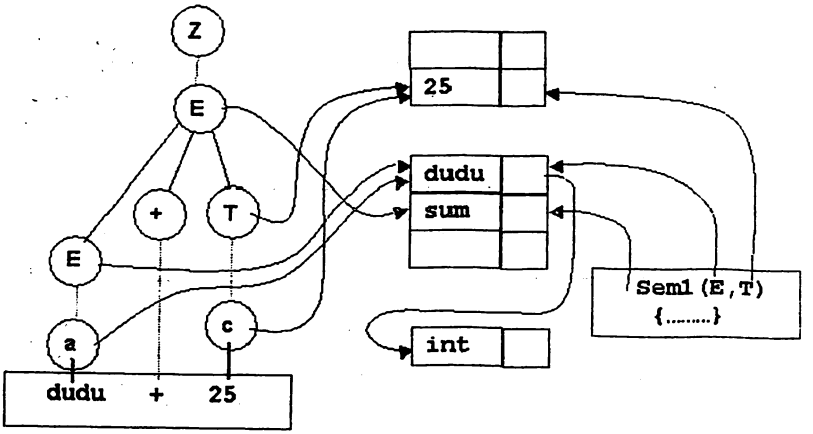
один и тот же семантический объект (например, переменная) может встречаться в различных, синтаксически несвязанных частях программы;

синтаксические конструкции описаний, определений и объявлений являются **источником семантики** объектов программы, они «заявляют» о существовании объектов и задают их свойства;

синтаксические конструкции, связанные с действиями, выполняемыми над объектами, являются **потребителями семантики,** их интерпретация, корректность, «смысл» зависят от семантических свойств объекта. Забегая вперед, можно заметить, что заключительная фаза трансляции (генерация кода, интерпретация) может рассматриваться как особые семантические действия, производимые над объектами;

первичным источником семантики является лексический анализ. Некоторые из лексем (например, идентификаторы и константы) наряду с **классом лексемы (обозначение выходной единицы лексического анализа),** т.е. **символом (обозначение той же единицы на входе синтаксического анализатора) имеют значение. Значением лексемы является сама** распознанная цепочка литер, она и представляет **семантическую составляющую лексемы,** которая попадает в семантические таблицы;

Лексемы, или то же самое, что терминальные символы входной строки (в терминах синтаксического анализа), ссылаются в семантические таблицах на свою семантику. В дальнейшем каждый промежуточный нетерминал также ссылается на собственную семантику. При этом любое правило преобразует семантику терминалов и нетерминалов правой части в семантику нетерминала левой части при помощи назначенной правилу **семантической процедуры;**



семантическая процедура, получая ссылки на семантику терминальных и нетерминальных символов правой части, формирует семантику результата и размещает ее в семантических таблицах, связывая ее через ссылку с нетерминалом левой части. Таким образом, семантическая составляющая транслятора тоже является фрагментарной (набор семантических процедур, соответствующих правилам грамматики) и объединяется в единое целое только в рамках синтаксического дерева.

Неформализуемость семантической фазы состоит в том, что семантическая модель транслируемой программы (семантические таблицы и связи между ними) создается компонентами транслятора (семантическими процедурами), которые программируется обычными средствами, т.е. как обычные компьютерные программы, без привлечения промежуточных формальных средств описания: грамматик, регулярных выражений и т.п.. Отсюда и уникальность семантики языка.

Замечание. Исторически сложилось, что в описание семантической фазы трансляции часто попадают разделы, связанные со структурами данных и алгоритмами их обработки, например, такие как хэширование (размещение и поиск вычислением адреса), двоичный поиск в таблицах. На самом деле они не имеют никакого отношения к основной идее: множества объектов внутреннего представления программы могут быть реализованы и в виде списков, деревьев и т.п..

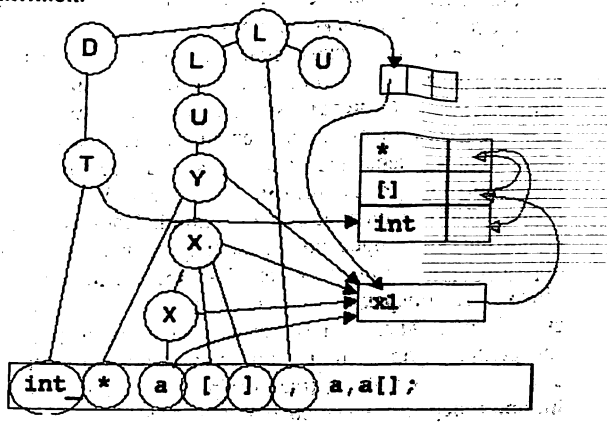
Семантические таблицы для Си-подобного компилятора

Попробуем в первом приближении представить себе, как может выглядеть внутреннее описание семантики объектов «классического» Си. В нем имеет место одноуровневая система объектов различных видов: **типов данных, переменных, функций.** Каждому такому виду объектов должна соответствовать своя **семантическая таблица (таблица имен).** На самом деле речь идет не об одной, а о нескольких таблицах.

Поскольку имеется одна глобальная область, а у каждой функции – своя локальная область переменных, то транслятор должен иметь множество таблиц переменных – по одной на каждую область действия. Очевидно, каждая функция должна ссылать на свою таблицу переменных. Кроме того, при трансляции любой функции имеют место **контекст локальных** и **глобальных переменных:** анализ имен переменных производится в соответствии с глобальной таблицей имен и таблицей имен текущей транслируемой функции. Очевидно, что семантическая процедура, которая связана с синтаксисом заголовка функции, должна создавать такую таблицу и устанавливать ее в качестве локального контекста трансляции. Кстати, формальные параметры функции также являются составной частью такого контекста, их семантика также должна быть внесена в эту таблицу.

Различные элементы синтаксиса языка программирования по-разному взаимодействуют с семантикой.

```
D :: T L;
T :: int
T :: char
L :: U | L,U
U :: Y
Y :: X | *Y
X :: a | X[]
```



При синтаксическом анализе описаний, т.е. при обработке правил, составляющих определение или объявление переменных, типов данных, заголовков функции, семантические процедуры **заполняют** описанные выше таблицы данными о семантике объектов программы, устанавливают взаимные ссылки.

Проследим, как можно сформировать требуемые структуру синтаксических таблиц, используя восходящий метод «свертка-перенос». Приведенная грамматика обеспечивает порядок построения синтаксического дерева, соответствующий приоритетам операций * и [] в описании. Все остальное выполняется семантическими процедурами в такой последовательности:

с терминалом **a**, соответствующим идентификатору, связана первичная семантика – значение **x1**. Оно становится записью в таблице переменных, первоначально ссылка на описание типа данных отсутствует **(NULL);**

последовательность сверток передает указатель на запись для **x1** в таблице переменных. Некоторые правила предназначены исключительно для установления приоритетов (например, **Y::X** определяет приоритет [] над *). Они не меняют семантики передаваемого объекта. Другие правила, выполняющие свертку операций * и [], должны менять тип переменной. Для этого они добавляют новую запись в таблицу типов и помещают указатель нее в конец цепочки типов, начинающейся с записи **x1**. То есть цепочка типов формируется по принципу очереди: **переменная x1 – массив - указатель**;

когда начинается свертка по правилам, соответствующим списку описателей, для нетерминала **L** заводится массив указателей на записи в таблице переменных, каждый новый нетерминал **U** добавляет к нему еще один указатель;

нетерминал **T** содержит ссылку на общий базовый тип данных, предназначенный для всего списка. При окончательной свертке указатель на него добавляется в конец **всех** сформированных цепочек типов данных.

При синтаксическом анализе выражений, в которых заданы действия, выполняемые над объектами программы, семантические таблицы используются как **источник данных.** Например, при выполнении операций свертки, соответствующих операциям над переменной **x1**, будет происходить аналогичная передача указателя на текущую семантику выражения, связанного с нетерминалом. При этом будет проверяться семантическая правильность выполняемых операций. Например, выражение **x1[i]++** будет семантически корректно, поскольку свертка выражения **x1[i]** аналогичным образом пройдет по цепочке **x1 – массив** до записи – указатель **int***, а для этого ТД операция ++ соответствует в Си++ операциям адресной арифметики. Аналогично, при выполнении сверток действия транслятора по генерации кода или интерпретации будут использовать данные семантических таблиц (например, размерность указуемой переменной типа **int** при генерации кода для 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 временного безымянного 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	Компилируемый код	Примечание
B[i]	&B[i]	l-value
B[i].hight	&B[i].hight	l-value
B[i].hight+5	X=*&B[i].hight X=X+5	Признак l-value поддерживается до операции +

Семантический анализ в нисходящих синтаксических анализаторах

Поскольку семантическая составляющая передается «вверх по дереву», а нисходящий анализ строит его наоборот, сверху вниз, то включение семантической компоненты в синтаксический анализ потребует «хитрых» решений. Рассмотрим некоторые из них:

при программировании распознавателя методом рекурсивного спуска «обратный ход» семантики соответствует передаче результата рекурсивной функции, каковой является каждый распознаватель. Результатом работы каждого распознавателя является ссылка (указатель, индекс) на описание семантики распознаваемого нетерминала в семантических таблицах. Текущий распознаватель, получая в таком виде семантику нетерминалов правой части правила, формирует из них собственную результирующую семантику (возможно, создавая новые записи в семантических таблицах) и возвращает ее в качестве собственного результата;

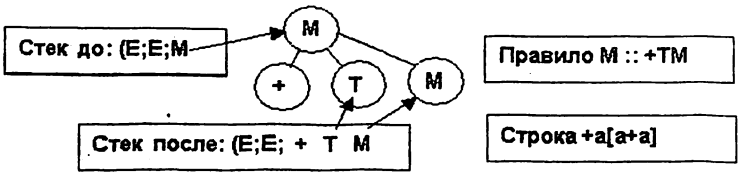
при использовании автоматного распознавателя (магазинного автомата), который не строит дерево, а обходит его сверху вниз, можно в качестве промежуточного уровня построить структуру данных, соответствующую синтаксическому дереву. После чего дерево в терминальные вершины помещаются ссылки на первичную семантику, выполнятся рекурсивный обход дерева, в каждой нетерминальной вершине вызывается семантическая процедура, соответствующая нетерминалу этой вершины. Способ формирования семантики «снизу вверх» аналогичен применяемому в рекурсивном спуске.

Построение синтаксического дерева в явном виде в магазинном автомате нуждается в пояснении. Для этого:

каждый нетерминал, содержащийся в стеке, имеет ссылку (указатель) на свою вершину – корень недостроенного поддерева;

при замене в стеке левой части правила на правую для нетерминалов (и терминалов тоже) правой части создаются вершины дерева, ссылки на них помещаются в нетерминалы стека и вершину, связанную с нетерминалом левой части;

таким образом, нисходящему применению любого правила соответствует достраивание поддерева, определяемого этим правилом.



II. Постановка задачи

1. Дополнить синтаксический анализатор на основе метода рекурсивного спуска, семантической составляющей, обеспечивающей внутреннюю структуру данных, соответствующую анализируемой семантике (семантических таблиц), позволяющей произвести проверку типов и распределение памяти.

2. Дополнить синтаксический анализатор на основе метода рекурсивного спуска, семантической составляющей, обеспечивающей форматирование исходного текста программы согласно уровню вложенности операторов.

III. Варианты

1. Си-грамматика выражений для арифметических операций, условных выражений, присваивания и скобок.

2. Си-грамматика выражений для сравнения, логических операций, присваивания и скобок.

3. Си-грамматика выражений для поразрядных операций, присваивания и скобок.

4. Си-грамматика определений простых переменных указателей и ссылок с инициализаторами.

5. Си-грамматика определения массивов и массивов указателей с инициализаторами.

6. Си-грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;** **if** (с else и без него), и блочных скобок.

7. Си-грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **switch** (с else и без него) и блочных скобок.

8. Си-грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **for,** и блочных скобок.

9. Си-грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **while** и блочных скобок.

10. Си-грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **do-whille** и блочных скобок.

11. Определение структурированного типа **struct** и контекстное определение переменных на его основе: **struct a{...} a,a[c].**

12. Определение структурированного типа **struct** и контекстное определение переменных на его основе: **struct a{...} a,a[c],*a,**a.**

13. Си-грамматика определения функции вида **i(,i,i,i...){ E=i;E=i;E; }** Программа представляет собой последовательность определений функций. Выражение может содержать вызов функции вида **i(E,E,E,...).**

14. Си-грамматика контекстного определения переменных и определения функций с использованием передачи по ссылке и значению.

15. Си-грамматика контекстного определения переменных и определения функций – процедур(тип **void**) с использованием передачи по значению.

IV. Содержание отчета

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.
2. Постановка задачи
3. Описание КС-грамматики.
4. Описание семантических таблиц анализа.
5. Описание алгоритма работы сканера.
6. Текст программы (оформляется после выполнения программы на ЭВМ).
7. Выводы по проделанной работе.

V. Методические указания

1. Семантика переменных и типов данных (ТД) для «классического» 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 описателе typedef, то имеющийся в определении идентификатор становится именем ТД (это и составляет семантику описателя typedef: имя переменной, содержащейся в контекстном определении, следующим за typedef, и является именем вводимого ТД). 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 типов struct и union 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.

```
typedef char *PSTR;
struct man {
char name[20];
char *addr;
int hight;
} A[10];
int B[20],C;
PSTR pp;
#define  BTD    0
#define  PTR    1
#define  STRU    2
```

```
#define  UNI3
#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 *parent;    // Составляющий ТД (один или несколько)
};
extern   d_type TYPES[100];
// Определение полей структурированного типа man
d_type XXX[]={
{"name",   20, 20, ARR, &TYPES[4] },          // char name[20]
{"addr",    4,  1, PTR, &TYPES[3] },   // char *addr;
{"hight",   4,  1, BTD, &TYPES[1] }};          // int class;
// Основная таблица типов данных
d_type TYPES[100] ={
// Определение БТД
{"char",   1,   0,   BTD,  NULL},      // [0]
{"int",    4,   0,   BTD,  NULL},           // [1]
{"long",   8,   0,   BTD,  NULL},           // [2]
// Неявное определение ТД или абстрактный ТД char*
{"",       4,   1,   PTR,  &TYPES[0]}, // [3]
// Неявное определение ТД или абстрактный ТД char[]
{"",       4,   1,   PTR,  &TYPES[0]}; // [4]
// Явное определение ТД typedef char *PSTR
{"PSTR",   4,   1,   PTR   &TYPES[0]}, // [5]
// Неявное определение ТД или абстрактный ТД int [20];
{"",       80,  20,  ARR,  &TYPES[1]}, // [6]
// Определение структурированного типа man
{"man",    28,  3,  STRU,  &XXX}.           // [7]
// Определение ТД – массив структур man[10]
{"",       280, 1,  ARR,   &TYPES[7]}};     // [8]
```

Структура семантической таблицы для переменных естественным образом вытекает из ее основных свойств в языке и должно содержать:

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

```
struct var {
char name[20];          // имя переменной
d_type *ptype;          // указатель на описатель ТД переменной
int    offset;          // смещение (адрес) в сегменте данных
```

```
void    *addr;              // адрес переменной в памяти (для
интепретатора)
};
```

Каждая функция в программе должна иметь собственный контекст – семантическую таблицу локальных переменных и формальных параметров. Кроме того, существует семантическая таблица глобальных переменных.

```
var GLOBAL[100]={
  {"A",      &TYPES[8], 0,     NULL},
  {"B",      &TYPES[6], 280, NULL},
  {"C",      &TYPES[1], 360, NULL},
  {"pp",     &TYPES[5], 364, NULL},
  {NULL}                          // конец таблицы
};
```

2. Программа, 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.

```
//---------- подсчет размерности производного ТД ----------
int GetSize(d_type *p){
switch    (p->TYPE)
  {     // Размерность БТД фиксирована
case BTD:  return p->size;
// Размерность указателя постоянна
case PTR:  return 4;
// Размерность массива – произведение числа элементов
// на размерность вложенного ТД
case ARR:  return p->dim * GetSize(p->child);
// Размерность структуры – сумма размерностей элементов
case STRU:   int s,i;
for (s=0,i=0; i<p->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<p->dim; i++)
{ k=GetSize(&(p->child[i])); if (k>s) s=k; }
return s;
}}
```

Лабораторная работа №8

Тема: Генерация кода. Интерпретация

Цель. Целью лабораторной работы является изучение генерации объектного кода и создании программы для генератора кода и интерпретатора.

I. Краткие теоретические сведения

Синтез выходного кода. Генерация кода. Интерпретация

Заключительная фаза трансляции - генерация кода или интерпретация - по своему способу включения в транслятор аналогична семантическому анализу. При выполнении шага синтаксического разбора («свертке» правила в восходящем разборе или подборе правила в нисходящем разборе) вызывается семантическая процедура, по результатам которой выполнения которой вызывается аналогичная процедура для генерации кода или 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, который будет работать **идентично** стеку распознавателя. 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 каждому правилу, в котором встречается бинарная или унарная операция, будет соответствовать неизменный код, извлекающий операнды из стека и возвращающий туда же результат выполнения операции.

Правило	Код стековой машины
E :: E + T	POP ax; POP bx; ADD ax,b; PUSH ax;
F :: a[E]	POP ax; MUL ax, size(int); ADD ax,#offset(a); MOV ax,[ax]; PUSH ax;
F :: c	MOV ax,#c; PUSH ax;

Выполняемая последовательность «сверток» правых частей правил приведет к появлению необходимой последовательности групп операций над стеком. В принципе, можно «сэкономить» операции над стеком. если

последний операнд держать не в вершине стека, а в некотором регистре, например ax, который следует интерпретировать как вершину стека. В приведенном примере показаны только операции свертки, «серая» часть строки находится в стеке.

Строка	Правило	Код стековой машины
(x1+12)*y2+6#		
(a+c)*a+c#	F :: a	PUSH ax; MOV ax,x1
(F+c)*a+c#	T :: F	---
(T+c)*a+c#	E :: T	---
(E+c)*a+c#	F :: c	PUSH ax; MOV ax,#12
(E+F)*a+c#	T :: F	---
(E+T)*a+c#	E :: E+T	POP bx; ADD ax,bx
(E)*a+c#	F :: (E)	---
F*a+c#	T :: F	---
T*a+c#	F :: a	PUSH ax; MOV ax,y2
T*F+c#	T :: T*F	POP bx; MUL ax,bx
T+c#	E :: T	---
E+c#	F :: c	PUSH ax; MOV ax,#6
E+F#	T :: F	---
E+T#	E :: E+T	POP bx; ADD ax,bx
E#	Z :: E#	---
Z		

Особенности интерпретации управляющих структур программы

Принцип интерпретации заключается в одновременном анализе и выполнении программы. На самом деле интерпретация в чистом виде встречается крайне редко, в основном в простых трансляторах. В сложных системах, как правило, имеется предварительная компиляция в промежуточный код.

Тем не менее, трансляторы, использующие непосредственную интерпретацию исходного текста, тоже встречаются. Для них хорошо подходит термин движок. Программа-интерпретатор просматривает исходный текст, запоминая позиции текущего положения элементов синтаксиса в файле (вплоть до использования функций позиционирования по текстовому файлу **fseek, ftell**), если по условиям интерпретации к этим конструкциям придется возвращаться (например, начало тела цикла). В данном случае происходит полное совмещение двух действий: текущая транслируемая управляющая структура программы одновременно является и текущей исполняемой (интерпретируемой).

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.

Особенности компиляции управляющих структур программы

Основной особенностью компиляции управляющих структур программы является то, что система команд процессора, работающего с линейной адресуемой памятью, имеет набор управляющих примитивов, более близкий к языкам неструктурированного типа:

команды обработки данных;

команды проверки условий;

команды условного и безусловного перехода по результат проверки условий.

Данный набор управляющих конструкций наиболее близок к блок-схемам, которые базируются на трех сходных примитивах – действие, условие, переход. Поэтому основная задача компилятора для языков, имеющих структурированные управляющие конструкции (последовательность (блок), ветвление и цикл в сочетании с принципом вложенности), состоит в «линеаризации» структурированного кода – размещении его в линейной памяти и генерации команд условного перехода.

Если результатом трансляции является программный код, то в операторах перехода должны быть сгенерированы численные адреса, для вычисления которых необходимо знать размерности сгенерированных блоков кодов. Речь не обязательно идет о физическом адресном пространстве. Например, при трансляции исходного текста программы на Си++ в объектный модуль, адресация всех данных и команд (для управляющих конструкций – адресов переходов) строится в адресном пространстве этого модуля, относительно его начала. Аналогичные вещи происходят при трансляции класса на языке Java в промежуточный байт-код.

Более простым вариантом является промежуточная трансляция на язык Ассемблера. Напомним, что основной задачей его является трансляция во внутреннее двоичное представление машинных команд и данных, обозначенных символическими именами (операции, метки областей памяти – переменных), а также распределение памяти под полученное двоичное представление. В таком случае задача транслятора состоит в генерации символьных меток, помечающих части управляющих конструкций. Рассмотрим возможные решения:

при восходящем СА необходимо использовать принцип сохранения частей генерируемого кода. То есть с каждым нетерминалом,

соответствующим оператору, связывается некоторый временный файл, 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, а для программного кода самой управляющей конструкции генерируются уникальные метки (можно использовать глобальный счетчик меток для всего генерируемого кода). Здесь приведен пример для условного оператора:

```
IF::=if (W) OP1 else OP2
        | |     |__файл OP2
        | |_____________файл OP1
        |_______________файл W

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

при нисходящем разборе с использованием метода рекурсивного спуска можно обойтись обычным последовательным файлом, в который процедуры распознавателя будут записывать генерируемый выходной код. При этом необходима генерация «меток вперед» в каждой управляющей конструкции, прежде чем будут вызваны распознаватели для вложенных в нее компонент.

II. Постановка задачи

1. Построить компилятор позволяющий произвести генерацию кода.
2. Построить интерпретатор для заданной грамматики.

III. Варианты

1. Паскаль – грамматика выражений для арифметических операций, условных выражений, присваивания и скобок. **Код:** префиксная польская запись.

2. Паскаль – грамматика выражений для арифметических операций, условных выражений, присваивания и скобок. **Код:** постфиксная польская запись.

3. Паскаль – грамматика выражений для сравнения, логических операций, присваивания и скобок. **Код:** префиксная польская запись.

4. Паскаль – грамматика выражений для сравнения, логических операций, присваивания и скобок. **Код:** постфиксная польская запись.

5. Паскаль – грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;** **if** (с else и без него), и блочных скобок. **Код:** тройки.

6. Паскаль – грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;** **if** (с else и без него), и блочных скобок. **Код:** косвенные тройки.

7. Паскаль – грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **case** и блочных скобок. **Код:** косвенные тройки.

8. Паскаль – грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **case** и блочных скобок. **Код:** четверки.

9. Паскаль – грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **for,** и блочных скобок. **Код:** четверки.

10. Паскаль – грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **for,** и блочных скобок. **Код:** косвенные тройки.

11. Паскаль – грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **for - downto,** и блочных скобок. **Код:** четверки.

12. Паскаль – грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **while** и блочных скобок. **Код:** тройки.

13. Паскаль – грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **while** и блочных скобок. **Код:** четверки.

14. Паскаль – грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **repeat-until** и блочных скобок. **Код:** косвенные тройки.

15. Паскаль – грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;**, **repeat-until** и блочных скобок. **Код:** четверки.

IV. Содержание отчета

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.
2. Постановка задачи
3. Описание КС-грамматики входного языка.

4. Описание алгоритма генератора кода (в соответствии с вариантом задания).

5. Описание алгоритма работы интерпретатора.(для второго задания).

6. Текст программы (оформляется после выполнения программы на ЭВМ).

7. Выводы по проделанной работе.

V. Методические указания

1. Интерпретатора конструкций if и do...while, использующий 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){
int n = i;                    // Запомнить начало тела цикла
            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; }}
```

2. Генерация «меток вперед» в каждой управляющей конструкции, прежде чем будут вызваны распознаватели для вложенных в нее компонент.

```
int Mnum=0;          // Номер генерируемой метки
void IF() {
int mm=Mnum;              // Зарезервировать две метки
Mnum=Mnum+2;
if (s[i]!='(') { error(); return; }
i++;
W();                  // Анализ условия и генерация кода проверки условия
if (s[i]!=')') { error(); return; }
```

```
    i++;
    fprintf( fd,"TST AX\n");          // Генерация кода для оператора IF
    fprintf( fd,"JEQ M%d\n",mm); // переход на метку для else
    OP();                             // Анализ оператора и генерация кода для ветки
then
    if (s[i]!='e')                    // генерация кода для if без else
    fprintf(fd, "M%d: ",mm);
    else                              // генерация кода для else
    i++;
    fprintf(fd, "JMP M%d\n ",mm+1);
    fprintf(fd, "M%d: ",mm);
    OP();          // Анализ оператора и генерация кода для ветки else
    fprintf(fd, "M%d: ",mm+1);
    }}
```

Литература

Основная литература

1. Молчанов А.Ю. Системное программное обеспечение: Учебник для вузов. –СПб: Питер, 2003.-396 с.
2. Афанасьев А.Н. Формальные языки и грамматики.: Учебная школа: УлГТУ, 1997. – 84 с.
3. Вирт Н. Алгоритмы и структуры данных – М.; МИР, 1989. – 360 с.
4. Гордеев А.З., Молчанов А.Д. Системное программное обеспечение. – Спб-Петер, 2002. -734с.
5. Дворогин А.И. Основы трансляции. Учебное пособие. – Волгоград. ВолГТУ,1999.80г.
6. Карпов С.Ю. Теория автоматов. Учебные пособия для вузов. –СПб: Питер, 2003.-201с.
7. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции -.: Мир, 1979.-487с.

Дополнительная литература

1. Кнут Д. Искусство программирования для ЭВМ. Т. 1. Основные алгоритмы. - М.: Мир, 1976.
2. Грисс Д. Конструирование компиляторов для цифровых вычислительных машин. –М. Мир, 1975-544с.
3. Курочкин В.М. Алгоритм распределения регистров для выражений за один обход дерева вывода//2 Всес. конф "Автоматизация производства ППП и трансляторов". 1983. С.104-105.
4. Надежин Д.Ю., В.А.Серебряков, В.М.Ходукин. Промежуточный язык Лидер (предварительное сообщение) // Обработка символьной информации. - М., 1987. С. 50-63.
5. Aho A., Sethi R., Ullman J. Compilers: principles, techniques and tools. N.Y.: Addison-Wesley, 1986.
6. Бек Л. Введение в системное программирование.М.: Мир, 1988.-448с.
7. Компаниец Р.И. и др. Системное программирование. Основы построения трансляторов.- СПб.: КОРОНА принт, 2000 г. -256 с.
8. Aho A.U.,Ganapathi M.,Tjiang S.W. Code generation using tree matching and dynamic programming // ACM Trans.Program. Languages and Systems.1989. V.11.N 4.
9. Fraser C.W., Hanson D.R. A Retargetable compiler for ANSI C. // SIGPLAN Notices. 1991. V 26.
10. A. Bezdushny, V. Serebriakov. The use of the parsing method for optimal code generation and common subexpression elimination // Techn. et Sci. Inform. 1993. V.12. N.1. P.69-92.
11. Harrison M.A. Introduction to formal language theory. Reading, Mass.: Addison-Wesley, 1978.

ОГЛАВЛЕНИЕ

Методическое пособие для практических занятий по дисциплине «Системное программное обеспечение»

Разработка рассмотрена на заседании кафедры «ТП»
и рекомендована к печати
(протокол № от 2008 года)

Авторы — Зав. каф. Назиров Ш.А.
Доц.каф. Кабулов Р.В.
Ст.пр. каф. Уринбаев С.К.

Ответственный редактор — Проректор по учебной работе ТУИТ, д.т.н., проф. Каримов М.М.

Корректор — Павлова С.И.

Бумага офсетная. Заказ № 150
Тираж. 50
Отпечатано в типографии ТУИТ
Ташкент 700084. ул.А.Тимура - 108