ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»
УЧЕБНО-НАУЧНО-ИССЛЕДОВАТЕЛЬСКИЙ ИНСТИТУТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Кафедра «Информационные системы»

А.П. Гордиенко, Н.И. Салина

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Методические указания
по выполнению лабораторных работ

Дисциплина – «Теория языков программирования и методы трансляции»

Специальности: 230105 «Программное обеспечение вычислительной техники и автоматизированных систем»
080801 «Прикладная информатика (в экономике)
080800.62 «Прикладная информатика»
230201 «Информационные системы и технологии»
230100.62 «Информатика и вычислительная техника»

Печатается по решению редакционно-издательского совета ОрелГТУ

Орел 2009

Авторы: канд. техн. наук, доц. каф. ИС А.П. Гордиенко
ст. преп. каф. ИС Н.И. Салина

Рецензент: канд. техн. наук, доц. каф. ИС А.В. Чижов

Методические указания содержат все необходимые теоретические сведения и практические примеры по дисциплине «Теория языков программирования и методы трансляции» для подготовки и выполнения лабораторных работ.

Предназначены для студентов очной формы обучения, обучающимся по направлению 080800.62 «Прикладная информатика» и специальностям 230105 «Программное обеспечение вычислительной техники и автоматизированных систем», 230100.62 «Информатика и вычислительная техника», 080801 «Прикладная информатика (в экономике)», 230201 «Информационные системы и технологии», изучающим дисциплину «Теория языков программирования и методы трансляции»

Редактор Г.А. Константинова
Технический редактор Г.А. Константинова

Орловский государственный технический университет
Лицензия ИД 00670 от 5.01.2000

Подписано к печати 08.06.2009 г. Формат 60×84 1\16
Печать офсетная. Усл. печ. л. 5,9. Тираж 50 экз.
Заказ №

Отпечатано с готового оригинал-макета
на полиграфической базе ОрелГТУ,
302020, г. Орел, Наугорское шоссе, 29.

СОДЕРЖАНИЕ

1 ЦЕЛЬ ПРАКТИКУМА

Целью проведения лабораторных и практических работ является:

– закрепление теоретических знаний в области построения лексических анализаторов;

– получение начальных навыков описания языков;

– приобретение опыта построения лексических анализаторов с использованием конечных автоматов;

– получение навыков реализации трансляторов.

2 ПОДГОТОВКА К ВЫПОЛНЕНИЮ РАБОТЫ

Каждая лабораторная и практическая работа рассчитана на то, что студент в ходе лекционного курса ознакомился с соответствующим теоретическим материалом. Основной же целью выполнения лабораторной работы является приобретение практических навыков в области построения трансляторов.

При подготовке к лабораторной и практической работе студент должен повторить лекционный материал, относящийся к изучаемому вопросу, а также ознакомиться с материалом, приведенным в соответствующем разделе данного методического указания. При этом необходимо обращать особое внимание на разобранные примеры и фрагменты программ.

Выполнение большинства работ основывается на материале, освоенном в предыдущих работах. При этом возможен вариант, когда лабораторная работа использует результаты выполнения задания предыдущей работы. В других случаях для понимания излагаемого материала и выполнения работы требуются знания основных положений предыдущих работ без обязательного выполнения задания.

В таблице 1 для каждой лабораторной приведена следующая информация. В столбце 3 проставлены номера лабораторных, ознакомление с материалом которых необходимо для выполнения данной работы. В столбце 4 приведены номера лабораторных, задания которых должны быть обязательно выполнены для выполнения текущей работы. В столбцах 5 и 6 приведено количество часов самостоятельной и аудиторной работы студента над лабораторной работой.

Таблица 1 – Сводная информация о лабораторных работах

Название лабораторной работы	Изучить материал лабораторной работы (№)	Выполнить задание лабораторной работы (№)	Самостоятельная работа (часы)	Аудиторная работа (часы)
1	2	3	4	5
Использование конечного автомата для построения лексического анализатора	1		6	4
Построение лексического анализатора с использованием генератора лексических анализаторов Lex	2		2	4
Использование метода рекурсивного спуска для построения синтаксического анализатора	3	2	4	4
Таблично управляемый синтаксический разбор сверху вниз	4		6	4
Синтаксический разбор снизу вверх	5		6	4
Генерация кода и реализация машины	6	3	6	4
Автоматизированное построение трансляторов с использованием генератора Yacc	7		2	4

Готовность студента к работе определяется преподавателем путем проведения собеседования. Основным материалом для собеседования являются контрольные вопросы, приведенные в разделе, посвященном выполнению конкретной лабораторной работы.

Для выполнения лабораторных работ студент должен иметь минимальные навыки программирования в Delphi.

Защита лабораторной работы производится после написания студентом программы и оформления отчета.

Структура разделов, посвященных выполнению лабораторных работ:

1) Цель работы. В этом разделе определяется, какие навыки должен приобрести студент в ходе выполнения лабораторной работы.

2) Теоретические положения. В одном или нескольких пунктах излагается теоретический материал, снабженный подробными примерами. На основе теоретического материала и примеров базируется задание к лабораторной работе.

3) Фрагменты программ. Содержимое этих пунктов демонстрирует реализацию ранее изложенного материала. Эти фрагменты могут быть использованы студентом при выполнении задания.

4) Контрольные вопросы.

5) Задание на лабораторную работу. В общем случае задание включает описание фрагмента распознаваемого языка и реализацию какого-либо алгоритма. Задание к лабораторной работе чаще всего состоит из двух частей. Первая выполняется студентом при подготовке к лабораторной работе и проверяется преподавателем при допуске к лабораторной работе. Вторая часть выполняется непосредственно во время аудиторного занятия.

6) Содержание отчета о выполнении лабораторной работы.

7) Защита лабораторной работы. Защита производится после написания студентом программы и оформления отчета. В общем, при защите лабораторной работы студент должен продемонстрировать работу программы и объяснить какие теоретические положения она иллюстрирует.

3 Лабораторная работа №1 «Использование конечного автомата для построения лексического анализатора»

3.1 Цель работы

Целью проведения лабораторной работы является:

– закрепление теоретических знаний в области построения лексических анализаторов;

– получение начальных навыков описания языков в виде регулярных выражений;

– приобретение опыта построения лексических анализаторов с использованием конечных автоматов;

– получение навыков реализации конечных автоматов.

3.2 Построение лексического анализатора

Основная задача лексического анализа – разбить входной текст, состоящий из последовательности одиночных символов, на последо-

вательность слов (лексем). Все лексемы делятся на классы. Примерами таких классов являются числа, идентификаторы, строки. Отдельно выделяют ключевые слова. Как правило, ключевые слова – это некоторое ограниченное подмножество идентификаторов.

С точки зрения дальнейших этапов работы компилятора, лексический анализатор выдает информацию двух видов: класс лексемы и значение лексемы. Ключевые слова распознаются либо явным выделением из входной последовательности, либо сначала выделяется идентификатор, а затем делается проверка на принадлежность его множеству ключевых слов.

Лексический анализатор может работать или как самостоятельная фаза трансляции, или как подпрограмма, работающая по принципу “дай лексему”. В первом случае выходом лексического анализатора является файл лексем, во втором лексема выдается при каждом обращении к лексическому анализатору.

Работа лексического анализатора описывается и реализуется формализмом конечных автоматов. Однако непосредственное описание конечного автомата неудобно. Поэтому для описания лексических анализаторов применяют другие подходы, например, формализм регулярных выражений.

Будем создавать лексический анализатор по следующей схеме:

– описание распознаваемого языка в форме регулярных выражений;

– преобразование регулярных выражений в недетерминированный конечный автомат;

– преобразование недетерминированного конечного автомата к детерминированному;

– минимизация конечного автомата.

Рассмотрим построение лексического анализатора на примере распознавания буквы азбуки Морзе. Буква состоит из последовательности точек и тире, после каждого такого символа должен следовать пробел. Буква заканчивается пробелом. Буква может быть пустой, то есть состоять из одного символа пробела.

3.2.1 Описание распознаваемого языка в форме регулярного выражения

Определим, что является регулярным выражением:

1) Пустая строка ε является регулярным выражением.

2) Если элемент a принадлежит алфавиту описываемого языка, то a – регулярное выражение.

3) Если r и s регулярные выражения, обозначающие языки L(r) и L(s), то регулярными являются следующие выражения:

а) r | s – объединение языков, то есть утверждается, что элемент должен принадлежать или языку L(r), или языку L(s);

б) r s – конкатенация языков, то есть утверждается, что за элементом языка L(r) следует элемент языка L(s);

в) r* - замыкание Клини языка, то есть ноль или более символов языка L(r);

г) (r) – регулярное выражение в скобках.

Кроме того, для описания языков используются операция r+, означающая один или более символов языка r, и операция r?, означающая ноль или один символ языка r.

Опишем букву азбуки Морзе в форме регулярного выражения. Сначала рассмотрим базовый случай. Элементы точка, тире и пробел принадлежат алфавиту распознаваемого языка, следовательно, существуют регулярные выражения “точка”, “тире” и “пробел” (по правилу 2). За каждой точкой или тире следует пробел, следовательно, существуют регулярные выражения “точка пробел” и “тире пробел” (по правилу 3б). В букве может встречаться либо точка с пробелом, либо тире с пробелом, следовательно, имеет место регулярное выражение “точка пробел | тире пробел” (по правилу 3а). Такая последовательность может встретиться в букве множество раз, а может и не встретиться вообще, то есть регулярное выражение имеет вид “(точка пробел | тире пробел)*” (по правилам 3г и 3в). Буква должна заканчиваться пробелом, то есть регулярное выражение для распознавания буквы азбуки Морзе выглядит следующим образом:

(точка пробел | тире пробел)* пробел

3.2.2 Преобразование регулярного выражения в недетерминированный конечный автомат

Конечный автомат это пятерка $(S, \Sigma, \delta, S_0, F)$.

S - конечное множество состояний.

Σ - конечное множество допустимых входных сигналов.

δ - функция переходов. Она отражает множество Sx(Σ∪{ε}) в множество состояний недетерминированного конечного автомата. Для детерминированного автомата функция переходов отражает множество SxΣ во множество состояний автомата. Другими словами, в зависимости от состояния и входного символа, δ определяет новое состояние автомата.

S_0 - начальное состояние конечного автомата, $S_0 \in S$.

F - множество конечных состояний автомата, $F \in S$.

Работа конечного автомата представляет собой последовательность шагов. Шаг определяется состоянием автомата и входным символом. Сам шаг состоит в изменении состояния автомата и считывании следующего символа входной последовательности.

Существуют следующие правила преобразования регулярных выражений в конечный автомат.

1) Регулярное выражение “ε” преобразуется в автомат из двух состояний и ε -перехода между ними (рисунок 1).

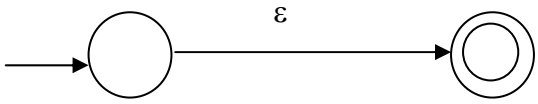


Рисунок 1 – Автомат для ε-перехода

2) Регулярное выражение из одного символа “a” преобразуется в конечный автомат из двух состояний и перехода между ними по входному сигналу a (рисунок 2).

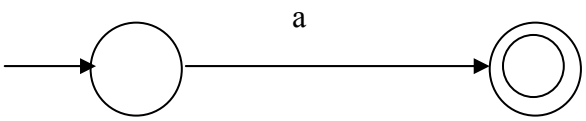


Рисунок 2 – Автомат для перехода по символу a

3) Пусть есть регулярное выражение rs и уже построены конечные автоматы для выражения r и выражения s. Тогда два автомата соединяются последовательно. На рисунке 3 представлены исходные автоматы для языков r и s. На рисунке 4 автомат для распознавания конкатенации этих языков.

Рисунок 3 – Исходные автоматы

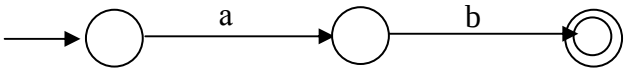


Рисунок 4 – Автомат для конкатенации языков

4) Пусть есть регулярное выражение r | s и уже построены конечные автоматы для выражения r и выражения s (рисунок 3). Тогда в результирующем автомате должна быть альтернатива выполнения одного из двух автоматов. То есть автомат для выражения r | s при автоматах для r и s с рисунка 3 имеет вид, представленный на рисунке 5.

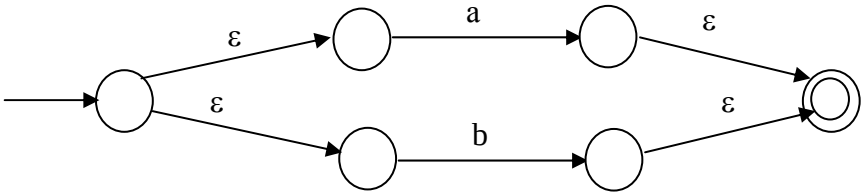


Рисунок 5 – Автомат для объединения языков

5) Пусть есть регулярное выражение r* при построенном конечном автомате r. В этом случае вводятся два новых состояния для возможности обхода автомата выражения r, а также вводится ε-переход между конечным и начальным состояниями для возможности многократного повторения автомата r. Если для регулярного выражения r построен автомат аналогичный рисунку 3, то регулярному выражению r* соответствует конечный автомат, представленный на рисунке 6.

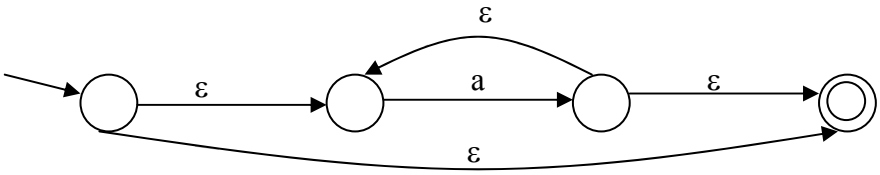


Рисунок 6 – Автомат для замыкания Клини языка

Построим конечный автомат для регулярного выражения
(точка пробел | тире пробел)* пробел

Сначала построим конечные автоматы для регулярных выражений “точка”, “тире” и “пробел” по правилу 2. Эти автоматы представлены на рисунке 7.

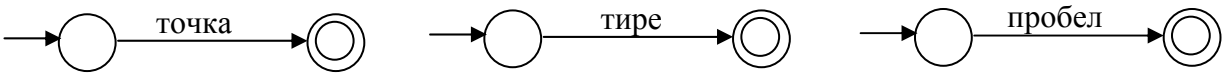


Рисунок 7

Для регулярных выражений “точка пробел” и “тире пробел” строим конечные автоматы по правилу 3 (рисунок 8).

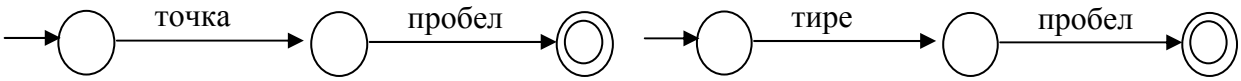


Рисунок 8

Регулярное выражение “точка пробел | тире пробел” преобразуем по правилу 4. Соответствующий автомат изображен на рисунке 9.

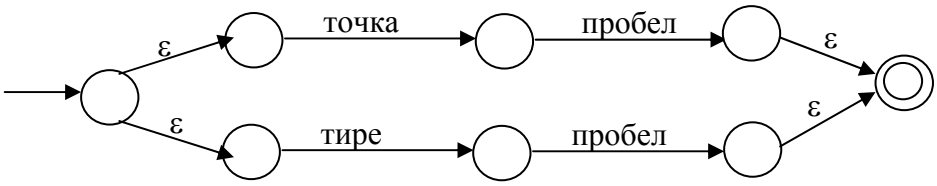


Рисунок 9

Теперь строим конечный автомат для регулярного выражения (точка пробел | тире пробел)*, используя правило 5 (рисунок 10).

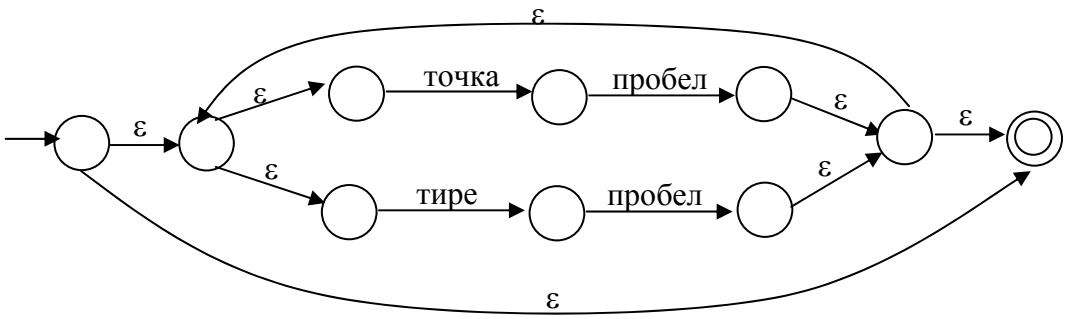


Рисунок 10

Применяя правило 3 для построения конечного автомата для полного регулярного выражения, получим следующий автомат для распознавания буквы азбуки Морзе, представленный на рисунке 11.

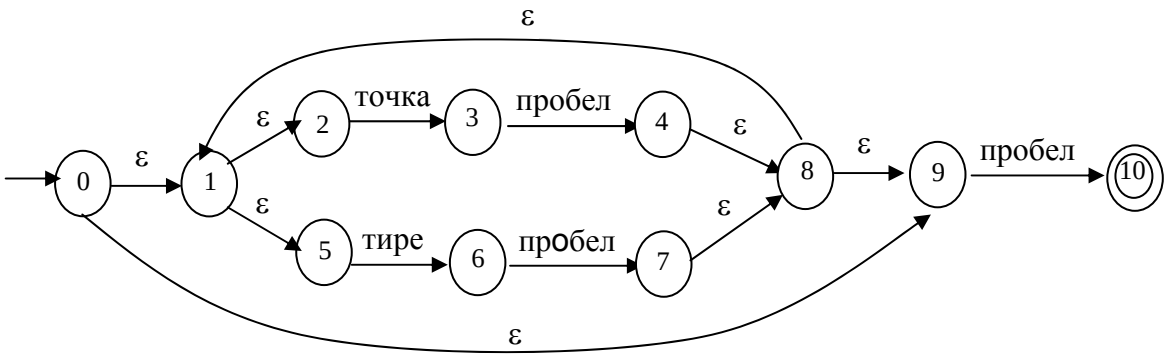


Рисунок 11 – Недетерминированный конечный автомат

3.2.3 Преобразование недетерминированного конечного автомата к детерминированному

Работа конечного автомата заключается в нахождении пути от начального состояния в конечное, по которому можно прочитать лексему. Лексема принимается, если такой путь найден. Недетерминированный конечный автомат включает ε-переходы и переходы из одного состояния по одному и тому же входному символу в разные состояния. Поэтому при работе недетерминированного автомата возможны откаты в процессе поиска пути, вследствие этого недетерминированный автомат работает медленно. Для каждого недетерминированного конечного автомата можно построить детерминированный конечный автомат.

Рассмотрим алгоритм преобразования. Входом алгоритма является недетерминированный автомат N, выходом – детерминированный автомат D, состоящий из множества состояний Dstates и множества переходов Dtrans.

Введем обозначения.

S - состояние из N;

T - множество состояний из N. При работе алгоритма такие множества становятся состояниями D.

Пусть реализованы следующие функции.

εclosure(S). Эта функция строит ε-замыкание состояния S, то есть множество состояний, в которые можно перейти из S по ε-переходам, это множество включает также само состояние S;

εclosure(T). Функция строит ε-замыкание множества состояний T, то есть множество состояний, достижимых из множества состояний T через ε-переходы. Это множество включает и состояния, принадлежащие T.

Move(T, a). Функция строит множество состояний, в которые можно перейти из T по входному символу a.

Алгоритм состоит в следующем.

Dstates:=εclose(S_0);

Пока в Dstates хотя бы одно множество T непомеченно выполнить

begin

пометить множество T

Для каждого входного символа a выполнить

begin

U:=εclosure(Move(T, a));

Если U ∉ Dstates

то Dstates:=Dstates+U;

Dtrans[T, a]:=U;

end;

end;

Рассмотрим преобразование недетерминированного автомата с рисунка 11 к детерминированному.

Определим εclosure(S_0). Через ε-переходы из нулевого стартового состояния можно попасть в состояния 1,2,5,9. Таким образом, в Dstates заносим множество {0,1, 2, 5, 9}.

На первой итерации цикла помечаем множество T={0,1, 2, 5, 9}.

Для входного символа "точка" U={3}, так как по этому символу из состояний множества {0,1, 2, 5, 9} можно перейти только в третье состояние, а из него ε-переходов нет. Dstates={{0,1, 2, 5, 9}', {3}}. Помещаем множество {3} в таблицу переходов в качестве нового состояния при текущем состоянии {0,1, 2, 5, 9} и входном символе “точка”, то есть Dtrans[{0,1, 2, 5, 9}, точка]:={3}. Для входного сигнала "тире" U={6}. Для символа "пробел" U={10}. После первой итерации внешнего цикла конечный автомат имеет вид Dstates={{0,1, 2, 5, 9}', {3}, {6}, {10}}, Dtrans =

	точка	тире	пробел
{0,1, 2, 5, 9}	{3}	{6}	{10}

На второй итерации рассматриваем T={3}, так как это первое непомеченное состояние в Dstate. Из этого множества возможен пе-

реход только по символу "пробел" в состояние 4. Из состояния 4 существуют ε-переходы в состояния 8, 9, 1, 2, 5. После выполнения второй итерации автомат имеет вид. Dstate={{0,1, 2, 5, 9}', {3}', {6}, {10}, {4, 8, 9, 1, 2, 5}}, Dtrans=

	точка	тире	пробел
{0,1, 2, 5, 9}	{3}	{6}	{10}
{3}	-	-	{4, 8, 9, 1, 2, 5}

Рассмотрение множества {6} добавит в автомат множество {7, 8, 9, 1, 2, 5} и соответствующий переход. Из множества {10} нет никаких переходов, так как оно конечное, поэтому четвертая итерация цикла ничего к графу не добавит. После этого конечный автомат будет иметь вид

Dstates={{0,1, 2, 5, 9}', {3}', {6}', {10}', {4, 8, 9, 1, 2, 5}, {7, 8, 9, 1, 2, 5}}

Dtrans=

	Точка	тире	пробел
{0,1, 2, 5, 9}	{3}	{6}	{10}
{3}	-	-	{4, 8, 9, 1, 2, 5}
{6}	-	-	{7, 8, 9, 1, 2, 5}
{10}	-	-	

Из множества {4, 8, 9, 1, 2, 5} по символу "пробел" можно перейти в множество {10}, по символу "тире" – в множество {6}, а по символу "точка" – в состояние {3}. Таким образом, рассмотрение множества {4, 8, 9, 1, 2, 5} не приведет к добавлению к автомату новых состояний, но добавит соответствующие переходы. Аналогичные рассуждения можно провести для множества {7, 8, 9, 1, 2, 5}. После рассмотрения этих двух состояний в Dstates не останется непомеченных множеств, следовательно, алгоритм завершит работу. При этом Dtrans имеет вид

	Точка	тире	пробел
{0,1, 2, 5, 9}	{3}	{6}	{10}

{3}	-	-	{4, 8, 9, 1, 2, 5}
{6}	-	-	{7, 8, 9, 1, 2, 5}
{10}	-	-	
{4, 8, 9, 1, 2, 5}	{3}	{6}	{10}
{7, 8, 9, 1, 2, 5}	{3}	{6}	{10}

Таким образом, конечный автомат будет иметь вид, представленный на рисунке 12.

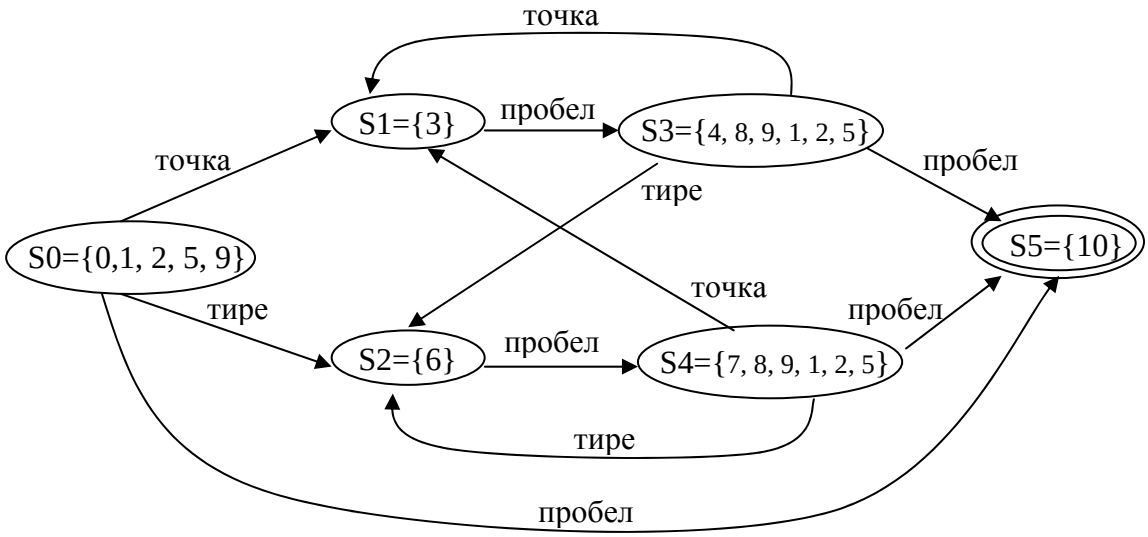


Рисунок 12 – Детерминированный конечный автомат

3.2.4 Минимизация конечного автомата

При минимизации конечного автомата все состояния делятся на две группы: финальные и нефинальные. Затем выполняется следующая процедура. Рассматриваем группу {S1, S2, …, Sk} и некоторый входной символ a. Если по этому входному символу есть переходы в разные группы, то исходную группу разбивают на части. Принадлежность Si к той или иной части зависит от того, в какую группу существует переход из состояния Si по входному символу a. Эта процедура продолжается до тех пор, пока не останется ни одной группы, которую можно было бы разбить по какому-либо входному сигналу.

Минимизируем конечный автомат, приведенный на рисунке 12. Разбиваем все состояния на две группы. В первую войдет конечное состояние S5, а во вторую все неконечные состояния, то есть группа имеет вид {S0, S1, S2, S3, S4}.

Первую группу разделить невозможно, так как она состоит из единственного состояния. Рассмотрим входной сигнал “точка” и вторую группу. По этому входному символу есть переходы только из состояний S0, S3, S4 в состояние S1, то есть для всех состояний группы по входному символу “точка” осуществляется переход в одну и ту же группу. Следовательно, входной сигнал “точка” не разбивает группу. Аналогичные рассуждения можно провести и для входного сигнала “тире”. Теперь рассмотрим входной символ “пробел”. По этому сигналу из состояний S1 и S2 осуществляется переход в состояния той же группы, а из состояний S0, S3, S4 – в состояние S5, то есть в другую группу. Таким образом, группу {S0, S1, S2, S3, S4} делим на две {S0, S3, S4} и {S1, S2}. Разбить получившиеся группы нельзя ни по какому входному сигналу. Таким образом, минимизированный конечный автомат будет иметь три состояния. Рассмотрим переходы между состояниями. Как видно по рисунку 12, из множества состояний {S0, S3, S4} существует переход в множество состояний {S1, S2} по символам “точка” и “тире”, а по символу “пробел” - переход в множество {S5}. Из множества {S1, S2} можно выполнить переход в множество состояний {S0, S3, S4} по символу “пробел”. Из состояния S5 переходов нет. Минимизированный конечный автомат для распознавания буквы азбуки Морзе представлен на рисунке 13.

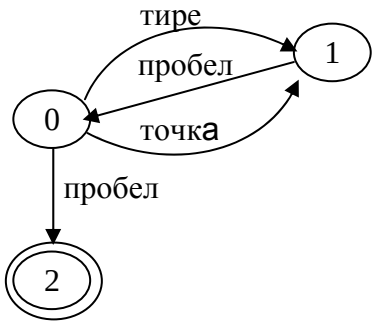


Рисунок 1. – Минимизированный конечный автомат

3.2.5 Конечный автомат для распознавания нескольких лексем

Распознаваемые языки включают более, чем одну лексему. Рассмотрим на примере, как должен быть построен конечный автомат для такого языка.

Пусть язык включает два класса лексем.

Первый описывается регулярным выражением (цифра)+пробел. Для распознавания этой лексемы построен конечный автомат, приведенный на рисунке14.

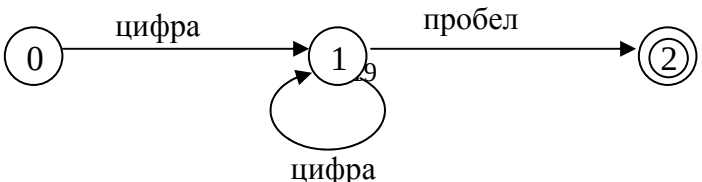


Рисунок 14 – Автомат для распознавания лексемы (цифра)+пробел

Второй класс лексем описывается выражением буква (цифра)*пробел и распознается автоматом, представленным на рисунке 15.

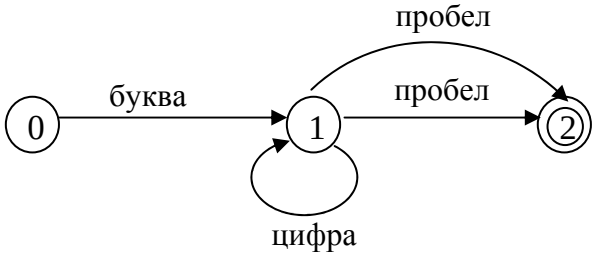


Рисунок 15 – Автомат для распознавания лексемы буква(цифра)*пробел

Результирующий автомат для распознавания двух лексем представлен на рисунке 16. В таком автомате получается два конечных состояния. В зависимости от того, в каком финальном состоянии оказался автомат определяется класс распознанной лексемы.

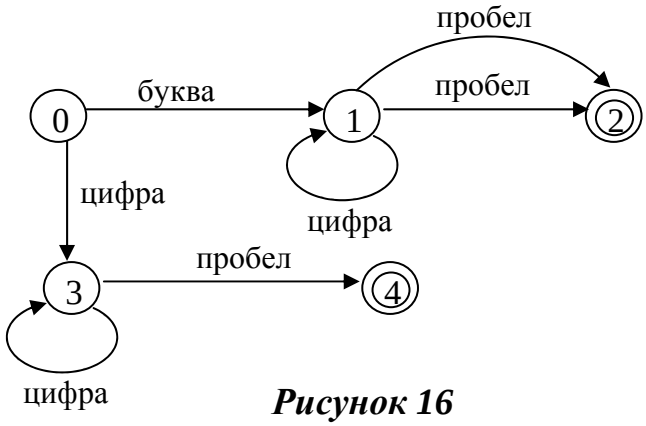


Рисунок 16

3.2.6 Распознавание ключевых слов

Как уже упоминалось, ключевые слова могут распознаваться явным образом, то есть для них строится собственный конечный автомат. Рассмотрим, как распознать язык, состоящий либо из идентификатора, заканчивающегося пробелом, либо из служебного слова if, за которым также следует пробел. Для этого в качестве входных символов необходимо отдельно выделить символ “i”, символ “f”, все символы без “i” и “f”, а также символ “пробел”. Детерминированный автомат для такого языка представлен на рисунке 17.

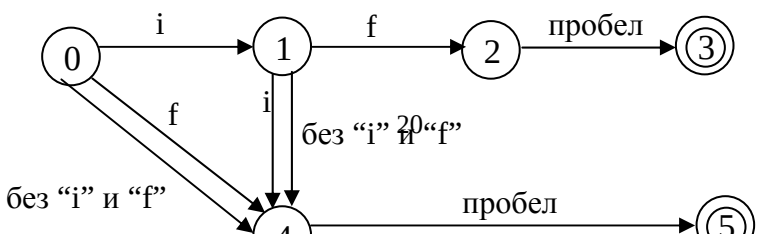


Рисунок 17

В приведенном автомате финальное состояние 3 соответствует распознанной лексеме класса “служебное слово”, а финальное состояние 5 – лексеме класса “идентификатор”.

Этот же язык может распознаваться с использованием недетерминированного автомата. Тогда из состояния 0 в состояние 4 можно попасть еще и по входному символу “i”. Кроме того, будет отсутствовать переход из состояния 2 в состояние 4. Работать этот автомат будет так: если пришел сигнал “i”, то автомат перейдет в состояние 1. Если в состоянии 1 или 2 на входе автомата появится неожидаемый сигнал, то произойдет откат до состояния 0 и работа автомата продолжится переходом в состояние 4.

Второй вариант распознавания ключевых слов не подразумевает построения специальных автоматов для ключевых слов. В этом случае анализатор должен иметь таблицу, содержащую ключевые слова языка. После распознавания идентификатора лексический анализатор сравнивает значение лексемы с элементами таблицы ключевых слов. Если было найдено совпадение, то лексема относится к классу “ключевое слово”, иначе – к классу “идентификатор”.

3.3 Реализация лексического анализатора в виде конечного автомата

Рассмотрим реализацию в виде конечного автомата лексического анализатора, построенного ранее.

Определим тип “входной сигнал”. Этот тип должен включать нижеприведенные элементы.

– “Точка” (dot), “тире” (dash) и “пробел” (space) для отражения входных сигналов конечного автомата.

– “Другой символ” (other). Этот элемент необходим для отражения всех символов, не входящих в алфавит азбуки Морзе, но появляющихся на входе конечного автомата.

– “Конец последовательности” (end). Этот элемент необходим, так как автомат должен знать, когда заканчивается последова-

тельность, которую нужно распознать как букву или как ошибочную последовательность.

```
type
input_signal = (dot, dash, space, other, end)
```

Определяем возможные состояния конечного автомата. Согласно графу, это будут состояния S0, S1, S2, а также специальное состояние S_error, которое будет соответствовать ошибке, возникшей при разборе.

```
type
state = (S0, S1, S2, S_error)
```

Исходя из рисунка 13, определим функцию переходов конечного автомата. В ошибочное состояние автомат переводит любой неожидаемый сигнал. Сигнал “Другой символ” переводит автомат в ошибочное состояние из любого состояния автомата.

```
const
next_state:  array [dot..other, S0..S1] of state =
((S1, S_error),
 (S1, S_error),
 (S2, S0),
 (S_error, S_error));
```

В рассматриваемом примере только один класс лексем, но обычно автомат распознает лексемы нескольких классов, поэтому введем тип “класс лексемы”. В данной реализации этот тип будет включать только одно значение - “буква”.

```
type    lexeme_class=(letter)
```

Реализуем работу конечного автомата, считая, что глобальная переменная Entry : string содержит входную последовательность. Переменные cur_state и cur_input необходимы для хранения текущего состояния автомата и входного символа. Глобальная переменная lex_val будет содержать значение распознаваемой лексемы. В начале работы автомата эта переменная инициализируется пустой строкой, а формирование значение лексемы будет производиться при распознавании символа входной последовательности. Приведенная ниже

функция возвращает класс распознанной лексемы. Работа функции заключается в переходе по состояниям конечного автомата до тех пор, пока не будет достигнуто конечное состояние или входная последовательность не будет разобрана полностью. Если автомат перешел в ошибочное состояние, то в системе возникает исключительная ситуация и пользователь получает сообщение об ошибке. Такое же сообщение пользователь получит, если просмотрена вся входная последовательность, а автомат не находится в конечном состоянии.

```
function state_machine : lexeme_class;
var cur_state:state; cur_input:input_signal;
begin
  lex_val:='';  cur_state:=s0; cur_input:=recognize;
  while (cur_state<>S2) and (cur_input<>endf) do
  begin
    cur_state:=next_state[cur_input, cur_state];
    if cur_state=S_error
    then raise exception.create(‘Лексическая ошибка');
    if cur_state<> s2 then cur_input:=recognize;
  end;
  if (cur_state <> S2) and (cur_input=endf)
  then raise exception.create('Лексическая ошибка')
  else result:=letter;
end;
```

При реализации конечного автомата была использована функция recognize, которая возвращает следующий символ входной последовательности. Эта функция необходима, так как последовательность имеет строковый тип, а для входных символов автомата задан свой тип input_signal. Кроме того, функция формирует значение лексемы.

```
function recognize:input_signal;
begin
if entry=''
then result:=endf
else
begin
  case entry[1] of
  '.': result:=dot;
```

```
    '-': result:=dash;
    ' ': result:=space;
    else result:=other
    end;
    lex_val:=lex_val+entry[1];
    entry:=copy(entry, 2,length(entry));
  end;
  end;
```

Теперь рассмотрим текст основной программы. Она будет состоять в вызове подпрограммы лексического анализатора до тех пор, пока не будет разобрана вся входная последовательность. В зависимости от класса распознанной лексемы, основная программа выдает пользователю сообщение, включающее класс и значение лексемы. Будем считать, что входная последовательность вводится пользователем через объект класса TEdit.

```
  begin
    entry:=edit1.text;
    while (entry<>'') do
    begin
      case state_machine of
      letter:showmessage('Буква'+lex_val);
      end;
    end;
  end;
```

3.4 Контрольные вопросы

1) В чем состоит основная задача лексического анализа?

2) Какую информацию должен выдавать лексический анализатор?

3) Что такое ключевые слова и как они выделяются из входной последовательности?

4) Что такое регулярное выражение?

5) Каким образом описывается конечный автомат?

6) Чем определяется и в чем состоит шаг работы конечного автомата?

7) Каким образом можно преобразовать к недетерминированному конечному автомату регулярные выражения r+ и r?

3.5 Задание

Создать лексический анализатор, выделяющий из входной последовательности следующие лексемы:

– "идентификатор" - последовательность букв и цифр, начинающаяся с буквы;

– "число" - последовательность цифр, целая часть от дробной отделяется точкой или запятой, причем дробная часть может отсутствовать;

– "присвоение" - строка ":=";

– "знак операции" - символы + или – ;

– ключевые слова begin и end.

Лексемы отделяются друг от друга пробелом или начинаются с новой строки.

Для лексемы “число” построить недетерминированный автомат, затем его детерминировать и минимизировать. Для остальных лексем эти действия можно невыполнять.

Входная последовательность должна вводиться с помощью объекта класса tMemo. Результат работы алгоритма также отражается с помощью объекта класса tMemo, в который помещаются строки вида <тип лексемы>,<текст лексемы>. Если автомат находится в ошибочном состоянии, то разбор далее не производится и пользователю выдается соответствующее сообщение.

Пример

входная последовательность

3434334 eree wew3

3434.4 + :=

Результат

число 3434334

идентификатор eree

идентификатор wew3

число 3434.4

знак +

присвоение :=

При выполнении лабораторной работы можно использовать допущение, что входная последовательность не превышает 255 символов.

Написание регулярных выражений и построение автоматов выполняется студентом при подготовке к лабораторной работе. Реализация конечного автомата производится в ходе аудиторного занятия.

3.6 Содержание отчета

Отчет должен содержать:

– Регулярные выражения, описывающие распознаваемые лексемы.

– Недетерминированный, детерминированный и минимизированный автоматы для распознавания лексемы “число”.

– Конечный автомат для распознавания всех лексем, перечисленных в задании.

3.7 Защита лабораторной работы

При защите лабораторной работы студент демонстрирует работу программы и объясняет, как задан и как работает построенный конечный автомат, то есть каким образом автомат меняет состояния для различных видов входной последовательности.

Кроме того, студент должен пояснить, исходя из каких положений задания были написаны регулярные выражения, а также наиболее важные моменты построения детерминированного конечного автомата.

4 Лабораторная работа №2 «Построение лексического анализатора с использованием генератора лексических анализаторов Lex»

4.1 Цель работы

Целью выполнения лабораторной работы является:

– закрепление навыков описания лексем с использованием регулярных выражений;

– знакомство с генератором лексических анализаторов Lex.

4.2 Общие сведения о генераторе лексических анализаторов Lex

Одной из наиболее распространенных систем автоматизации разработки лексических анализаторов является система Lex. Работа генератора Lex заключается в построении конечного автомата на основе регулярных выражений, заданных пользователем.

Исходный текст для Lex создается в текстовом редакторе, например, в блокноте и сохраняется в файле с расширением l. Программа lex.exe, в качестве параметра которой выступает исходный файл с расширением l, строит лексический анализатор на Pascal(то есть файла с расширением pas). Lex создает анализатор на основе кода, содержащегося в файле Yylex.cod, поэтому этот файл должен находиться в том же каталоге, что и файл lex.exe. Если исходный текст содержит ошибки, то при попытке построить лексический анализатор создастся файл ошибок с расширением lst.

Полученный в результате работы Lex модуль присоединяется к проекту Delphi, который будет содержать основной текст программы. Для корректной работы с проектом он должен подключать библиотеку lexlib.pas.

4.3 Lex-программа

Текст Lex-программы может включать регулярные выражения, операторы и функции Pascal, а также переменные и функции, объявленные и реализованные в библиотеке lexlib.pas.

4.3.1 Регулярные выражения

Lex позволяет использовать регулярные выражения, приведенные в таблице 2.

Таблица 2 – Регулярные выражения, используемые в Lex

Название	Обозначение	Описание
1	2	3
Одиночный символ	a	На входе конечного автомата должна появиться только символ “a”

Продолжение таблицы 2

1	2	3
Строка символов	“abc”	На входе конечного автомата должен появиться последовательность символов “a”, “b”, “c”
Символ из строки	[abc]	Первым символом входной последовательности является один из символов “a”, “b” или “c”
Символ из диапазона	[a-z]	Корректен любой символ, имеющий код больший или равный кода символа “a”, но меньший или равный кода символа “z”.
Повторение регулярного выражения 0 и более раз	r*	
Повторение регулярного выражения 1 и более раз	r+	
Повторение регулярного выражения 0 или один раз	r?	
Конкатенация	r1r2	Во входной последовательности за r1 должно следовать r2
Объединение	r1|r2	Во входной последовательности должно встречаться r1 или r2
Все остальное	.	Если входной символ не соответствует ни одному из регулярных выражений, то он соответствует выражению “все остальное”
Следующий символ воспринять не как специальный, а как обычный	\	Применяется для обработки символов *,+,.,-.
Новая строка	\n	

4.3.2 Разделы Lex-программы

Исходный текст программы Lex может состоять из трех разделов, разделенных %%.

раздел определений
%%

раздел правил
%%
Пользовательский раздел

Раздел определений содержит определения регулярных выражений. Каждое выражение имеет вид <имя> <подстановка>. Указанное имя может использоваться далее, при работе вместо него будет отрабатывать регулярное выражение, являющееся подстановкой.

Радел правил Lex-программы имеет вид
p1 {действие 1}
p2 {действие 2}
…
pn {действие n}

Каждое pi – регулярное выражение, а каждое i-ое действие – фрагмент программы, описывающий, какое действие должен сделать лексический анализатор, когда образец pi сопоставляется лексеме. Действие включает код на Pascal, включая функции, определенные в Lexlib.pas.

Пользовательский раздел содержит вспомогательные процедуры, используемые в действиях.

Lex воспринимает последовательность символов, как переносимую без изменения в Pascal, если она начинается не с начала строки. То есть, если есть необходимость получить в Pascal определения переменных, типов и т.п., то их можно указать в тексте исходного файла не сначала строки.

4.3.3 Функции и переменные, используемые при работе с Lex-анализатором.

Lex преобразует текст исходного файла в функцию yylex : integer. Вызов этой функции приводит к запуску лексического анализатора, обрабатывающего остаток входной последовательности. Значение, возвращаемое функцией, соответствует классу распознанной лексемы. Функция yylex возвращает значение в том случае, если действие, соответствующее регулярному выражению, включает передачу управления основной программе. Передать управление можно, используя функции returni(<значение : integer>) и returnc(<значение : char>). Эти функции реализованы в lexlib.pas. Обе они прекращают лексический разбор и передают управление основной программе, но первая возвращает значение, указанное в качестве

параметра, а вторая код символа, указанного в качестве параметра. Значения, возвращаемые функциями returni и returnc, должны быть расценены главной программой как класс распознанной лексемы. В качестве параметра функции return(i) может выступать константа. Использование констант в качестве возвращаемого параметра позволяет сделать код основной программы удобочитаемым. Обычно константы начинают задавать с 257, так как все предыдущие значения могут быть использованы при передаче кодов символов. Если входная последовательность пуста, то функция yylex всегда выдает значение 0.

Все строки, которые должны быть включены в конечный модуль до функции yylex, описываются в разделе определений. Строки, которые должны быть включены в конечный модуль после функции yylex, описываются в пользовательском разделе.

Перед вызовом функции yylex для новой входной последовательности необходимо вызвать функцию yyclear, которая приводит конечный автомат лексического анализатора в начальное состояние. Кроме того, нужно вызвать функцию yyMemoInit. В качестве параметров эта функция должна получить четыре объекта класса TMemo. Первый объект предназначен для ввода пользователем входной последовательности. Второй выводит фрагменты входной последовательности, которые не были распознаны в качестве лексемы какого-либо класса. Третий – для вывода сообщений об ошибках. Четвертый – для формирования отчета о работе анализатора. Последний объект будет использоваться в синтаксическом анализаторе, выполненном в генераторе Yacc. Для указания строки во входном Memo, с которой должен начаться разбор, используется переменная yylineno:integer.

Кроме класса лексемы, лексический анализатор должен возвратить значение лексемы. Для этой цели используется переменная yytext:string[255]. Заметим, что при работе со строковыми типами в Lex необходимо четко задавать их длину. Например, нельзя объявить константу типа string, но можно типа string[20].

4.4 Работа анализатора, построенного с помощью генератора Lex

Лексический анализатор, сгенерированный Lex, взаимодействует с основной программой следующим образом. При вызове его основной программой лексический анализатор посимвольно читает остаток входной последовательности, пока не находит самый длинный префикс, который может быть сопоставлен одному из регулярных

выражений pi. Если несколько регулярных выражений соответствуют входной последовательности, то выбирается первый найденный в Lex-программе. Затем выполняется действие i. Как правило, действие i возвращает управление основной программе. Если это не так, то продолжается разбор до тех пор, пока очередное действие не вернет управление основной программе.

Характерной особенностью конечного автомата, построенного генератором Lex, является отсутствие конечных состояний. Для каждой лексемы входного файла от стартового состояния автомата строится ветка. В ходе работы автомат читает символы из буфера входной последовательности и перемещается между состояниями. В том случае, если символ, считанный из буфера входной последовательности, не ожидается автоматом в текущем состоянии, то разбор завершается и выполняется действие, соответствующее лексеме, распознанной до прихода последнего символа. Этот символ возвращается в буфер входной последовательности, то есть он будет считан первым при дальнейшей работе анализатора. Автомат переводится в стартовое состояние. Если действие, соответствующее распознанной лексеме, передало управление основной программе, то автомат начнет вновь работать после вызова функции yylex из основной программы. Если передачи управления не произошло, то автомат начинает работать снова сразу после выполнения действия.

4.5 Пример построения лексического анализатора с использованием Lex.

Рассмотрим Lex-программу, распознающую целые числа и идентификаторы. Целое число представляет собой последовательность символов, состоящую из одной и более цифр. Идентификатор – это последовательность букв и цифр, начинающаяся с буквы. Кроме того, лексический анализатор должен распознавать символ “+”.

В разделе определений формируем строки, необходимые для компиляции полноценного модуля лексического анализатора. Для этого указываем заголовок модуля, декларируем функцию yylex в интерфейсном разделе для того, чтобы эта функция была доступна из других модулей.

Лексический анализатор будет возвращать лексемы трех классов. Для классов “идентификатор” и “число” зададим константы. При получении на входе символа “+” Lex будет возвращать код этого символа. Определим еще одну константу, которую лексический анализатор будет возвращать, если на входе получен какой-либо символ,

не соответствующий ни одной лексеме. Анализатор, построенный Lex, использует функции и переменные, реализованные в Lexlib.pas, поэтому необходимо подключить соответствующий модуль. Кроме того, в разделе определений опишем две подстановки “цифра” и “буква”. “Цифра” – это любой символ от “0” до “9”. “Буква” – это любой строчный или прописной символ от “a” до “z”. Заметим, что все строки, которые должны быть перенесены в лексический анализатор без изменений, начинаются не с начала строки.

```
  unit Simple;
  interface
  const num=257; id=258; lex_error=313;
  function yylex : Integer;
  implementation
  uses lexlib;
D [0-9]
L [a-zA-Z]
%%
```

Раздел правил Lex-программы будет состоять из пяти правил. Первые три правила соответствуют трем классам лексем, действия, выполняемые при совпадении входной последовательности с этими правилами, состоят в передаче управления основной программе. Четвертое правило необходимо для игнорирования пробелов во входной последовательности, то есть при поступлении пробела в качестве входного символа лексический анализатор не выполняет никаких действий, а продолжает читать входную последовательность со следующего символа. Пятое правило обеспечивает передачу основной программе информации об ошибке при появлении непредусмотренного лексемами символа.

```
{D}+              returni(num);
{L}({L}|{D})+     returni(id);
\+                returnc(‘+’);
" "               ;
.                 returni(lex_error);
%%
```

Пользовательский раздел программы используем для корректного завершения модуля лексического анализатора.

```
 end.
```

Текст основной программы приведен ниже. Объект Memo3 класса TMemo выступает в качестве служебного, а Memo2 предназначен для отражения результатов лексического разбора.

```
var i:integer;
begin
yyclear;
yymemoinit(memo1, memo3, memo3, memo3);
yylineno:=0;
repeat
  i:=yylex;
  case i of
    num: memo2.lines.add('число '+yytext);
    id: memo2.lines.add('идентификатор '+yytext);
    ord('+'): memo2.lines.add('знак '+yytext);
    lex_error: raise exception.Create('Лексическая ошибка')
  end;
until i=0;
end;
```

4.6 Контрольные вопросы

1) Какие разделы выделяются в Lex-программе и каково их назначение?

2) При помощи чего лексический анализатор передает управление основной программе?

3) Каково назначение переменной yytext?

4) Каким образом лексический анализатор, сгенерированный Lex, взаимодействует с основной программой?

5) Каковы особенности конечного автомата, построенного генератором лексических анализаторов Lex?

4.7 Задание

Написать лексический анализатор, распознающий лексемы:

– “Идентификатор” – последовательность букв и цифр, начинающаяся с буквы, в последовательности могут встречаться как строчные, так и заглавные буквы.

– “Целое число” – последовательность цифр.

– “Десятичное” – последовательность цифр, в которой целая часть отделяется от дробной запятой или точкой.

– “Присвоение” – :=

- “Знак операции” – +, -, *, /
- “Служебное слово var”
- “Служебное слово if”
- “Служебное слово then”
- “Служебное слово else”
- “Служебное слово begin”
- “Служебное слово end”
- “Логическая операция” - or или and
- “Знак сравнения” - <>=
- “Разделитель команд” - ;
- “Операция округления” - round
- “Двоеточие” - :
- “Открытая скобка” - (
- “Закрытая скобка” -)

При распознавании все пробелы должны пропускаться. Если во входной последовательности присутствует посторонний символ, то разбор должен прекращаться и появляться сообщение об ошибке.

4.8 Защита лабораторной работы

В ходе защиты лабораторной работы студент поясняет работу созданного лексического анализатора, а именно, созданного входного Lex-файла и текста основной программы. Работа анализатора демонстрируется на примере входных последовательностей различного вида.

5 Лабораторная работа №3. «Использование метода рекурсивного спуска для построения синтаксического анализатора»

5.1 Цель работы

Целью выполнения лабораторной работы является:

- получение навыков описания языка с использованием контекстно-свободных грамматик;
- изучение метода рекурсивного спуска.

5.2 Общие сведения

Не все языки можно разбирать с помощью регулярных выражений, например, нельзя записать регулярное выражение для степенной функции. Для работы с такими языками используются контекстно-свободные грамматики.

В самом общем виде контекстно-свободная грамматика описывает язык как множество строк, полученных применением конечного множества продукций. Формально контекстно-свободная грамматика это четверка <V, ∑, P, S>. Рассмотрим составляющие контекстно-свободной грамматики. V – это конечное множество всех грамматических символов. ∑ - множество терминальных символов, причем ∑ является подмножеством V. Терминальные символы в грамматике будем обозначать строчными буквами. Множество нетерминальных символов N=V-∑. Нетерминальные символы в грамматике будем обозначать заглавными буквами. S – стартовый нетерминал. P – множество правил вывода. P⊆NxV*, то есть продукция – это последовательность, начинающаяся с нетерминального символа (левая часть правила), за которым следует замыкание Клини терминальных и нетерминальных символов (правая часть правила).

Рассмотрим пример грамматики, описывающей арифметические действия. (рис. 18)

1. GOAL→ EXPR
2. EXPR→EXPR+TERM
3. EXPR→ EXPR-TERM
4. EXPR→ TERM
5. TERM→ ERM*FACTOR
6. TERM→ ERM/FACTOR
7. TERM→ FACTOR
8. FACTOR→ num
9. FACTOR→ id

Рисунок 18 – Грамматика языка арифметических действий

В данном примере стартовым является нетерминал GOAL, так как он не встречается в правой части ни одной из продукций. ∑={+, -, *, /, num, id}; N={GOAL, EXPR, TERM, FACTOR}. Любая из продукций состоит из нетерминала в левой части и последовательности терминалов и нетерминалов в правой. Например, правило 2 утверждает, что EXPR есть последовательность из нетерминала EXPR, терминала + и нетеминала TERM.

Будем говорить, что αγβ выводимо за один шаг из αAβ (αγβ=>αAβ), если в грамматике существует правило вывода A→γ, а α и β - произвольные строки из V. Если u1=>u2=>…=>un, то можно

утверждать, что u1 выводится из un за 0 или более шагов (u1=>*un). Если un нельзя вывести из u1 за 0 шагов, то говорят, что un выводимо из u1 за 1 или более шагов(u1=>+un).

Различают два вида вывода: левый и правый. Если на каждом шаге заменяют самый левый нетерминальный символ, то вывод называется левым, если самый правый, то вывод – правый.

Если дана грамматика G со стартовым символом S, то используя отношение =>+, можно определить язык L(G), порожденный грамматикой G. Строки такого языка могут содержать только терминальные символы из G. Строка терминалов w принадлежит L(G) тогда и только тогда, когда w выводимо из S за 1 или более шагов S=>+w.

Таким образом, приведенная ранее грамматика описывает язык арифметических операций. При этом строки этого языка могут содержать только терминалы +, -, *, /, num, id. Пусть есть строка “id*id+num”. Рассмотрим, принадлежит ли эта строка языку арифметических операций. Для этого проверим, можно ли вывести входную строку из стартового символа. В скобках после знака => будем указывать номер продукции, по которой осуществляется вывод.

GOAL=>(1) EXPR=>(2) EXPR+TERM=>(4) TERM+TERM=>(5)
TERM*FACTOR+TERM=>(7) FACTOR*FACTOR+TERM=>(9)
id*FACTOR+TERM=>(9) id*id+TERM=>(7)
id*id+FACTOR=>(8) id*id+num

Получили входную строку, следовательно, строка “id*id+num” принадлежит языку. Заметим, что был использован левый вывод.

Вывод можно представить в виде дерева. Дерево является деревом вывода грамматики, если выполнены следующие условия

- корень дерева помечен стартовым символом
- каждый лист помечен терминалом или ε
- каждая внутренняя вершина помечена нетерминалом
- если N – нетерминал, которым помечена внутренняя вершина и X1, X2, …, Xn – метки ее прямых потомков в указанном порядке, то правило N→X1 X2 … Xn существует в грамматике.

Входная последовательность соответствует языку, определяемому грамматикой, если листья дерева вывода соответствуют этой последовательности.

Для рассмотренного ранее примера дерево вывода представлено на рисунке 19.

Рисунок 19

5.3 Рекурсивный спуск

Существует два метода синтаксического разбора: сверху вниз и снизу вверх. В данной лабораторной работе будем рассматривать первый метод. При разборе сверху вниз начинают разбор со стартового нетерминала и, применяя правила, заменяют нетерминалы левой части правила на последовательности грамматических символов правой. Входная последовательность принимается, если она выведена из стартового символа.

Процедура рекурсивного разбора сверху вниз состоит из следующих шагов:

– Для узла дерева, помеченного как нетерминал A, выбирают одну из продукций вида A→α. После этого строим от A ветви, соответствующие α.

– Если в процессе применения продукций получено обрамление, несоответствующее входной последовательности, то производится откат.

– Находим следующий узел, помеченный нетерминалом, для подстановки правила.

При таком подходе может возникнуть проблема бесконечного цикла. В грамматике для арифметических операций применение второго правила приведет к зацикливанию процедуры разбора. Подобные грамматики называются леворекурсивными. Грамматика называется леворекурсивной, если в ней существует нетерминал A, для которого существует вывод A=>+Aα. В простых случаях левая рекурсия вызвана правилами вида

$A \rightarrow A\alpha|\beta$

В этом случае вводят новый нетерминал и исходные правила заменяют следующими.

$A \rightarrow \beta B$
$B \rightarrow \alpha B|\varepsilon$

Рассмотрим правила 2 и 4 грамматики с рисунка 18. Правило 2 делает грамматику леворекурсивной. Воспользуемся вышеизложенным приемом исключения левой рекурсии. α=+T, β=T. Введем новый нетеримнал EXPR1 и получим новые продукции. EXPR→TERM EXPR1; EXPR1→+TERM EXPR1; EXPR1→ε.

На рисунке 20 изображена грамматика, к которой была преобразована грамматика с рисунка 18 путем исключения левой рекурсии.

Применение рекурсивного спуска в вышеизложенном виде может работать очень длительное время за счет откатов. Поэтому важно найти такой алгоритм, который мог бы однозначно выбирать продукцию на каждом шаге.

Есть разновидность грамматик, которые при разборе сверху вниз позволяют выбирать продукцию на основе первых k символов входной последовательности. Будем использовать LL(1)-грамматики, которые позволяют выбирать продукцию на основе первого символа входной последовательности. Первое L означает, что сканирование осуществляется слева направо, второе L, что строится левый вывод.

Грамматика, приведенная на рисунке 20, является LL(1), так как при выборе правила для любого нетерминала достаточно проанализировать первый символ входной последовательности.

1. GOAL→EXPR
2. EXPR→TERM EXPR1
3. EXPR1→+TERM EXPR1
4. EXPR1→-TERM EXPR1
5. EXPR1→ε
6. TERM→FACTOR TERM1
7. TERM1→*FACTOR TERM
8. TERM1→ /FACTOR TERM1
9. TERM1→ε
10. FACTOR→num
11. FACTOR→id

Рисунок 20

5.4 Пример реализации метода рекурсивного спуска

Реализуем грамматику, приведенную на рисунке 16. Будем считать, что уже написан лексический анализатор, который возвращает константы num и id при появлении на входе соответствующих терминалов. Для остальных терминалов лексический анализатор возвращает код соответствующих символов.

Для каждого нетерминала напишем функцию, которая включает анализ входной последовательности и выбор правила. Функция возвращает значение логического типа в зависимости от того, соответствует ли входная последовательность нетерминалу.

Для реализации синтаксического анализатора потребуется глобальная переменная token: integer, которая будет содержать распознанную лексему.

Основная программа в этом случае должна выполнить все установки для лексического анализатора, считать первый входной сигнал и вызвать функцию, соответствующую стартовому терминалу. Результат, который вернет эта функция, будет свидетельствовать о соответствии входной последовательности языку арифметических операций.

```
yymemoinit(memo1, memo3, memo3, memo3);
yyclear;
yylineno:=0;
token:=yylex;
if goal
then showmessage(‘Успех’)
else showmessage(‘Неудача’)
```

В соответствии с первым правилом грамматики, функция goal должна вызвать функцию expr и в зависимости от ее результата сформировать свой результат.

```
function goal: boolean;
begin
  result:=expr;
end;
```

Аналогично, согласно второму правилу входная последовательность соответствует нетерминалу expr в том случае, если первая ее часть соответствует нетерминалу TERM, а вторая нетерминалу EXPR1

```
function expr:boolean;
begin
  result:=(term and expr1);
end;
```

Для нетерминала EXPR1 существует три продукции. Все три опишем в одной функции. По какому из трех правил будет продолжаться доказательство, зависит от первого входного символа. Если этот символ “+”, то синтаксический анализатор будет работать по третьему правилу, если “-”, то по четвертому правилу. Если первый символ не является ни плюсом, ни минусом, то входная последовательность уже соответствует нетерминалу EXPR1, поэтому функция должна вернуть значение “истина” (правило 5 грамматики). Если доказательство ведется по третьему или четвертому правилу, то после распознания первого терминала необходимо считать следующий символ входной последовательности.

```
function expr1: boolean;
begin
  case token of
  ord('+') :
  begin
    token:=yylex;
    result:=term and expr1;
end;
ord('-'):
  begin
    token:=yylex;
    result:=term and expr1;
  end
```

```
  else result:=true;
  end; {case}
end; {function}
```

Функции для реализации продукций 6-9 аналогичны предыдущим и приведены ниже.

```
function term: boolean;
begin
  result:= factor and term1
end;

function term1: boolean;
begin
  case token of
  ord('*'):
  begin
     token:=yylex;
     result:=factor and term1;
  end;
  ord('/'):
  begin
     token:=yylex;
     result:=factor and term1;
  end
  else result:=true;
  end; {case}
end; {function}
```

Для нетерминала FACTOR в грамматике присутствуют два правила, если последовательность не соответствует ни одному из них, то она не соответствует нетерминалу FACTOR.

```
function factor: boolean;
begin
  case token of
  id:
  begin
    token:=yylex;
    result:=true;
  end;
  num:
  begin
```

```
    token:=yylex;
    result:=true;
  end;
else result:=false
end {case}
end; {function}
```

Расширим язык, который должен распознаваться синтаксическим анализатором. Прежде всего, значение арифметического выражения должно присваиваться какой-либо переменной. Введем еще одну команду: запросить значение переменной у пользователя. Эта команда состоит из служебного слова read, за которым следует идентификатор, взятый в круглые скобки. Теперь наш язык представляет собой список команд, разделенных символом “;”. Команда – это либо операция присвоения переменной значения, либо запрос значения переменной у пользователя. Список команд может быть пустым.

Грамматика для нового языка представлена на рисунке 21.

1. GOAL→LIST_INSTRUCTION
2. LIST_ INSTRUCTION →READ_INSTR ; LIST_INSTRUCTION
3. | ASSIGN_INSTR ; LIST_INSTRUCTION
4. | ε
5. READ_INSTR→ read_term (id)
6. ASSIGN_INSTR→ id = EXPR
2. EXPR→TERM EXPR1
3. EXPR1→+TERM EXPR1
4. EXPR1→-TERM EXPR1
5. EXPR1→ε
6. TERM→FACTOR TERM1
7. TERM1→*FACTOR TERM1
8. TERM1→ /FACTOR TERM1
9. TERM1→ε
10. FACTOR→num
11. FACTOR→id

Рисунок 21

Будем считать, что при распознавании строки “read” лексический анализатор возвращает основной программе константу read_term. Если во входной последовательности встречаются симво-

лы “(” или “)”, то LEX возвращает коды этих символов. Для строки “:=” лексический анализатор возвращает код символа “=”.

Ниже приведен текст функции для новых и изменившихся продукций.

```
function goal: boolean;
begin
  result:=list_instruction;
end;

function list_instruction: boolean;
begin
  case token of
  read_term:
  begin
    if read_instr and (token=ord(';'))
    then result:=list_instruction
    else result:=false;
  end;
  id:
  begin
    if assign_instr and (token=ord(';'))
    then result:=list_instruction
    else result:=false
  end;
  else result:= true {case}
  end; {case}
end;{function}
function read_instr: boolean;
begin
  if (token=read_term) and (yylex=ord('('))
     and (yylex=id) and (yylex=ord(')'))
  then
  begin
    result:=true;
    token:=yylex;
  end
  else result:=false;
end;
function assign_instr: boolean;
```

```
begin
  if (token=id) and (yylex=ord('='))
  then
  begin
    token:=yylex;
    result:=expr;
  end
  else result:=false
end;
```

5.5 Контрольные вопросы

– Что такое контекстно-свободная грамматика?
– Какой нетерминал является стартовым в грамматике?
– Когда можно утверждать, что одна последовательность выводится из другой за один шаг?
– В чем различие левого и правого вывода?
– Как вывод можно представить в виде дерева?
– В чем состоит метод разбора сверху вниз?
– Какова процедура рекурсивного разбора сверху вниз?
– Какие грамматики являются леворекурсивными?
– В чем состоит метод исключения левой рекурсии?
– Какие грамматики являются LL(1)-грамматиками?

5.6 Задание

1. Написать грамматику для распознавания программы. Программа состоит из раздела определения переменных и раздела команд. Раздел определения переменных начинается со служебного слова “var”, за которым следует список переменных вида <имя переменной>:<тип переменной>. Раздел команд начинается со служебного слова “begin”, за которым следует список команд. Завершается раздел команд служебным словом “end”.

Команды могут быть следующего вида:

а) Команда присвоения. Состоит из идентификатора, знака присвоения (:=) и арифметического выражения. Арифметическое выражение включает в себя сложение, умножение, вычитание, деление, скобки, округление.

б) Команда чтения значения переменной. Она состоит из служебного слова “read” и идентификатора, взятого в скобки.

в) Команда вывода. Состоит из служебного слова “write” и идентификатора, взятого в скобки или строки, заключенной в кавычки и скобки.

г) Команда ветвления. Состоит из трех разделов: условие, список операторов при выполнении условия, список операторов при не выполнении условия. Раздел условия начинается со служебного слова If. После него следует само условие. Условие состоит из конструкций вида идентификатор|число =|<|> идентификатор|число. Конструкции такого вида заключаются в скобки. Такие конструкции могут быть соединены в условии или операцией and, или операцией or. Если в условии присутствуют такие операции, то связываемые ими конструкции должны быть взяты в круглые скобки. Скобки могут встречаться в условии и для обозначения приоритета действий. Два остальных раздела команды ветвления имеют идентичную структуру. Первый из них начинается со служебного слова “Then”, а второй с “Else”. После служебного слова должна следовать команда или список команд, заключенные в программные скобки “begin” “end”. Раздел “Else” в операции ветвления может отсутствовать.

2. Реализовать полученную грамматику методом рекурсивного спуска.

Первая часть задания выполняется студентом при подготовке к лабораторной работе.

5.7 Содержание отчета

Отчет о лабораторной работе должен включать грамматику для языка, описанного в задании.

6 Лабораторная работа № 4 «Таблично управляемый синтаксический разбор сверху вниз»

6.1 Цель работы

Целью выполнения лабораторной работы является:

– получение навыков построения таблицы для разбора сверху вниз;

– изучение алгоритма работы таблично управляемого синтаксического анализатора.

6.2 Структура анализатора

Структура таблично управляемого синтаксического анализатора представлен на рисунке 22.

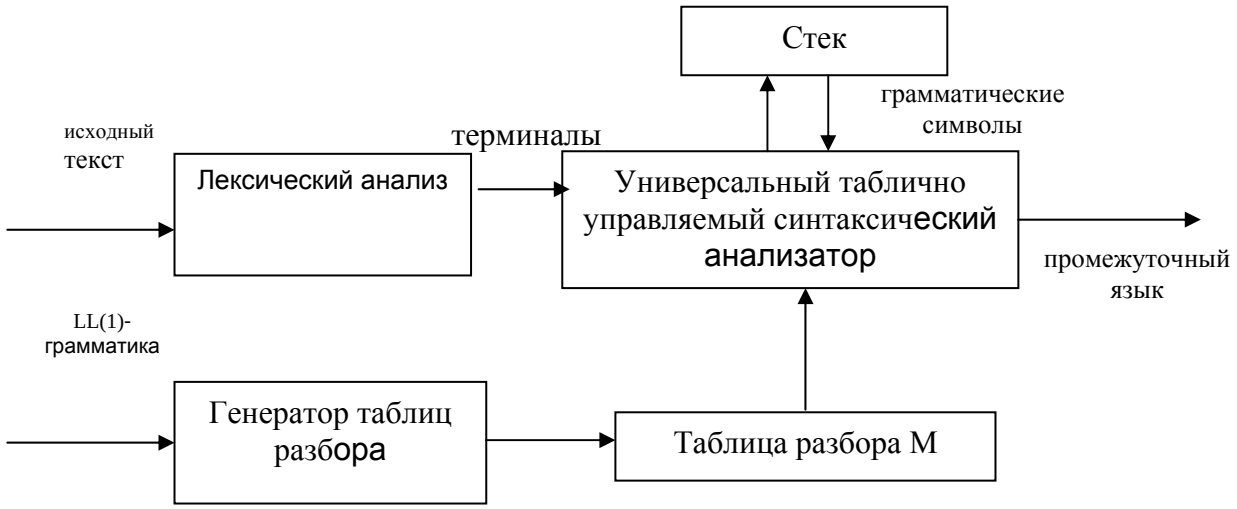


Рисунок 22

В стеке хранятся все грамматические символы, как терминалы, так и нетерминалы. Таблица разбора состоит из продукций грамматики. Столбцы таблицы именованы терминалами грамматики, а строки – нетерминалами. Эта таблица определяет, какую продукцию нужно рассматривать для некоторой пары: терминал на входе и нетерминал в вершине стека. Далее рассмотрим процесс построения таблицы разбора.

6.3 Построение множества first

Для последовательности α определим множество first как множество терминалов, с которых может начинаться последовательность, выводимая из α. Если из α можно вывести пустую строку, то в множестве first последовательности α должно присутствовать ε. Определим множество first для некоторого грамматического символа x.

1. Если x - терминал, то first(x)={x}, так как первым символом последовательности из одного терминала может являться только сам терминал.

2. Если в грамматике присутствует правило $X \rightarrow \varepsilon$, то множество first(x) включает ε. Это означает, что X может начинаться с пустой последовательности, то есть отсутствовать вообще.

3. Для всех продукций вида $X \rightarrow Y_1\ Y_2\ \ldots\ Y_k$ выполняем следующее. Добавляем в множество first(X) множество first(Y_i) до тех пор, пока first(Y_{i-1}) содержит ε, а first(Y_i) не содержит ε. При этом i изменяется от 0 до k. Это необходимо, так как если Y_{i-1} может отсутствовать, то необходимо выяснить, с чего будет начинаться вся последовательность в этом случае.

Проиллюстрируем третье правило на примере продукции X → A B C. Пусть first(A)={+, ε}, first(B)={-}, first (C)={id}. При i=1 можно увидеть, что first(Y_1)={+, ε}, то есть добавляем {+} в множество first(X) и продолжаем рассматривать грамматические символы правой части продукции. First(Y_2)={-} уже не содержит ε, поэтому добавляем {-} в first(X) и прекращаем рассмотрение продукции. Таким образом, first(X) = {+, -}. Заметим, что ε отсутствует в first(X), хотя ε принадлежит first(A). Из рассматриваемой продукции не следует, что пустая строка выводима из X, следовательно, ε не может принадлежать first(X).

Построим множества first для всех грамматических символов языка, описанного на рисунке 20. Согласно первому правилу множество first всех терминалов языка состоит только из самого этого терминала (таблица 1 столбец 2).

Так как в грамматике присутствует правило EXPR1→ε, то во множество first(EXPR1) добавляем ε. Аналогично для множества first(TERM1). Результат шага 2 построения множества first представлен в столбце 3 таблицы 1.

Множество first(FACTOR) строится на основе продукций 10 и 11 согласно правилу 3 алгоритма. Исходя из продукций 7 и 8, во множество first(TERM1) должны быть добавлены множества {*} и {/}. В обоих случаях достаточно просмотреть только первый грамматический символ правой части правила. Так как множество first этого символа не содержит ε, то дальнейший анализ правила не производим. Согласно продукции 6 first(TERM)=first(FACTOR).

Остальные продукции рассматриваются аналогичным образом. Результат выполнения третьего шага алгоритма построения множества first представлен в столбце 4 таблицы 2. В столбце 5 этой таблицы представлено множество first грамматического символа, полученное в результате выполнения всех шагов алгоритма.

Таблица 3

Грамматический символ	Шаг алгоритма			first
	1	2	3	
1	2	3	4	5
GOAL			{num, id}	{num, id}
EXPR			{num, id}	{num, id}

EXPR1		ε	{+, -}	{ε, +, -}
TERM			{num, id}	{num, id}
TERM1		ε	{*, /}	{ε, *, /}
FACTOR			{num, id}	{num, id}
num	{num}			{num}
id	{id}			{id}
+	{+}			{+}
-	{-}			{-}
*	{*}			{*}
/	{/}			{/}

6.4 Построение множества follow

Для каждого нетерминала грамматики можно построить множество follow, то есть множество терминалов, которые можно встретить непосредственно после нетерминала в какой-либо последовательности, соответствующей грамматике.

Алгоритм построения множества follow заключается в следующем. Вначале символ конца входной последовательности помещается во множество follow стартового нетерминала (шаг 1). Затем выполняются шаги 2-4 до тех пор, пока можно еще что-либо добавить в множество follow(X).

2. Если есть правило $A \rightarrow \alpha B \beta$, то в множество follow(B) добавляем множество first(β) без ε. То есть, если есть правило, утверждающее, что за B следует β, то за нетеримналом будут следовать терминалы, с которых начинается последовательность β.

3. Если есть правило $A \rightarrow \alpha B$, то в множество follow(B) добавляется множество follow(A). То есть, если есть правило, утверждающее, что нетерминалом B заканчивается последовательность A, то за нетрминалом B будут следовать те же терминалы, что и за всей последовательностью A.

4. Если есть продукция $A \rightarrow \alpha B \beta$ и пустая последовательность принадлежит first(β), то в множество follow(B) добавляется множество follow(A). Если последовательность β не пуста, то терминалы, которые могут следовать за B, уже найдены на шаге 1. Если же последовательность β пуста, то шаг 3 фактически сводится к шагу 2.

Рассмотрим построение множеств follow для нетерминалов грамматики, приведенной на рисунке 20.

Стартовым является нетерминал GOAL, поэтому на первом шаге follow(GOAL):=eof. Затем рассмотрим продукции грамматики, при этом будем считать, что α может являться пустой последовательностью.

Условию второго шага построения множества follow соответствуют следующие продукции.

EXPR→TERM EXPR1, поэтому в follow(TERM) добавляем first(EXPR1) - ε

TERM→FACTOR TERM1, следовательно, в follow(FACTOR) заносим first(TERM1) - ε

Условию третьего правила удовлетворяют следующие продукции.

GOAL→EXPR, на основании этого заносим в follow(EXPR) follow(GOAL).

EXPR→TERM EXPR1, поэтому в follow(EXPR1) добавляем follow(EXPR).

EXPR1→+ EXPR, эта продукция позволяет занести в follow(EXPR) follow(EXPR1). Однако выполнять это не нужно, так как follow(EXPR1)=eof, а follow(EXPR) уже содержит этот элемент.

TERM→FACTOR TERM1, поэтому в follow(TERM1) добавляем follow(TERM).

TERM1→ * TERM, следовательно, в follow (TERM) можно занести follow(TERM1), но все элементы, которые можно было бы добавить, уже содержаться в этом множестве.

Теперь рассмотрим продукции, соответствующие четвертому шагу алгоритма построения множества follow. Пустая последовательность ε принадлежит множеству first только двух нетерминалов: EXPR1 и TERM1, поэтому на четвертом шаге будут рассмотрены только две продукции.

EXPR→TERM EXPR1, на основе этого в follow(TERM) добавим follow(EXPR).

TERM→FACTOR TERM1, следовательно, в follow(FACTOR) занесем follow(TERM).

Результаты вышеизложенных рассуждений сведены в таблице 3. Заметим, что продукции 4 и 9 не рассматривались, так как они с точки зрения построения множества follow эквиваленты продукциям 3 и

8 соответственно. Также не рассматривались продукции 5, 9, 10, 11, так как они не подходят ни под одно условие.

Таблица 4

Нетерминал	Шаги				follow
	1	2	3	4	
GOAL	{eof}				{eof}
EXPR			{eof}		{eof}
EXPR1			{eof}		{eof}
TERM		{+, -}		{eof}	{+, -, eof}
TERM1			{+, -}		{+, -}
FACTOR		{*, /}		{eof}	{*, /, eof}

Теперь необходимо еще раз повторить шаги 2 – 4. Новые элементы во множество follow можно добавить на шаге 3. Рассматривая продукцию TERM→FACTOR TERM1, в follow(TERM1) нужно добавить follow(TERM). Теперь follow(TERM)={+, -, eof}. Таким образом, в follow(TERM1) добавляем элемент eof. Аналогично на четвертом шаге, рассматривая продукцию TERM→FACTOR TERM1, в follow(FACTOR) занесем элементы + и -, которые отсутствуют в этом множестве, но на данном этапе принадлежат follow(TERM). Еще одно выполнение шагов 2-4 ничего нового к множеству follow не добавит, поэтому алгоритм закончит работу. Результат работы алгоритма представлен в таблице 4.

Таблица 5

Нетерминал	Шаги				follow
	1	2	3	4	
GOAL	{eof}				{eof}
EXPR			{eof}		{eof}

EXPR1			{eof}		{eof}
TERM		{+, -}		{eof}	{+, -, eof}
TERM1			{+, -, eof}		{+, -, eof}
FACTOR		{*, /}		{eof, +, /}	{*, /, eof, +, /}

6.5 Построение таблицы разбора

После построения множеств first и follow можно построить саму таблицу разбора.

Таблица разбора строится следующим образом:

Для всех продукций A→α грамматики выполняем:

1) Для всех терминалов a, принадлежащих first(α), в клетку [A, a] таблицы разбора записываем продукцию A→α.

2) Если ε принадлежит first(α), то для всех b, принадлежащих follow(A), в клетку [A, b] записываем продукцию A→α.

Во все остальные клетки таблицы записываем признак ошибки. Это означает, что при данном нетерминале в стеке не ожидается указанный терминал во входной последовательности.

Если при построении таблицы возникает попытка в одну клетку записать две или более продукции, то разбираемая грамматика не является LL(1)-грамматикой, следовательно, не может разбираться таким способом.

Продолжим построение таблицы разбора для грамматики с рисунка 20.

Рассмотрим первую продукцию. first(EXPR)={num, id}, то есть, записываем продукцию GOAL→EXPR в клетки таблицы [GOAL, num] и [GOAL, id]. ε не принадлежит first(EXPR), поэтому правило 2 построения таблицы разбора не применимо к первой продукции грамматики. Для второй продукции first(α)=first(TERM EXPR1)=first(TERM)={num, id}. Переход first(TERM EXPR1)=first(TERM) возможен, так как first(TERM) не содержит ε. Если бы это было не так, то необходимо было бы рас-

смотреть и first(EXPR1). На основании первого правила построения таблицы разбора в клетки [EXPR, num] и [EXPR, id] записываем продукцию EXPR→TERM EXPR1.

Далее построение таблицы производится аналогичным образом. Интерес представляют продукции 5 и 9. Эти продукции в отличие от остальных соответствуют второму правилу построения таблицы разбора. Follow(EXPR1)={eof}, поэтому заносим продукцию EXPR1→ε в клетку [EXPR1, eof]. Follow(TERM1)={eof, +, -}, поэтому в клетки таблицы [TERM1, eof], [TERM1, +] и [TERM1, -] заносим продукцию TERM1→ε.

Таблица разбора представлена ниже.

Таблица 6

1	2	3	4	5	6	7	8
	num	id	+	-	*	/	eof
GOAL	GOAL → EXPR	GOAL → EXPR					
EXPR	EXPR → TERM EXPR1	EXPR → TERM EXPR1					
EXPR1			EXPR1 →+ EXPR	EXPR1 →- EXPR			EXPR1 →ε

Продолжение таблицы 6

1	2	3	4	5	6	7	8
TERM	TERM →FAC TOR TERM1	TERM →FAC-TOR TERM1					
TERM1			TERM1 → ε	TERM1 → ε	TERM1 →* TERM	TERM1 →/ TERM	TERM1 →ε
FAC-TOR	FAC-TOR→ num	FAC-TOR→i d					

6.6 Алгоритм разбора

Алгоритм работы таблично управляемого синтаксического анализатора представлен ниже.

Занести символ окончания входной последовательности в стек

Занести в стек стартовый нетерминал

Распознать символ входной последовательности

Повторять

Если

в вершине стека находится терминал

то

Если

распознанный символ равен вершине стека

то

Извлечь из стека верхний элемент и распознать символ входной последовательности

иначе

вывести сообщение об ошибке;

иначе {если в вершине стека нетерминал}

Если

в клетке [вершина стека, распознанный символ] таблицы разбора существует правило

то

извлечь из стека элемент и занести в стек все терминалы и нетерминалы найденного в таблице правила в стек в порядке, обратном порядку их следования в правиле

иначе

вывести сообщение об ошибке

пока

вершина стека не равна концу входной последовательности

Если

распознанный символ не равен концу входной последовательности

то

вывести сообщение об ошибке

Рассмотрим, как работает этот алгоритм для входной последовательности x+5, при использовании в качестве таблицы разбора таблицы 5.

Первоначально в стек заносим символ конца входной последовательности и стартовый нетерминал. Таким образом, стек имеет вид “eof GOAL”. Распознаем первый символ входной последовательности, то есть текущий символ=id. В вершине стека находится символ GOAL, который является нетерминалом, поэтому находим продукцию в таблице в клетке [GOAL, id]. Это продукция GOAL→ EXPR. Из стека извлекаем GOAL и заносим EXPR. Далее выполняем следующие действия:

1) Стек=“eof EXPR”. Текущий символ = id. Вершина стека=EXPR, то есть нетерминал. Клетка таблицы [EXPR, id]=EXPR→TERM EXPR1.

2) Стек=“eof EXPR1 TERM”. Заметим, что грамматические символы правой части правила занесены в обратном порядке. Текущий символ = id. Вершина стека нетерминал. Клетка таблицы [TERM, id]=TERM→FACTOR TERM1.

3) Стек=“eof EXPR1 TERM1 FACTOR”. Текущий символ = id. Вершина стека нетерминал. Клетка таблицы [FACTOR, id]=FACTOR→id.

4) Стек=“ eof EXPR1 TERM1 id”. Текущий символ = id. Вершина стека терминал. Текущий символ равен вершине стека.

5) Стек=“eof EXPR1 TERM1”.Текущий символ = +. Вершина стека нетерминал. Клетка таблицы [TERM1, +]=TERM1→ε.

6) Стек=“eof EXPR1”. Заметим, что правая часть правила TERM1→ ε означает, что в стек не нужно ничего заносить. Текущий символ = +. Вершина стека нетерминал. Клетка таблицы [EXPR1, +]=EXPR1→+ EXPR.

7) Стек=“eof EXPR +”.Текущий символ = +. Вершина стека терминал. Текущий символ равен вершине стека

8) Стек=“eof EXPR”.Текущий символ = num. Вершина стека нетерминал. Клетка таблицы [EXPR, num]=EXPR→TERM EXPR1.

9) Стек=“eof EXPR1 TERM”.Текущий символ = num. Вершина стека нетерминал. Клетка таблицы [TERM, num]=TERM→FACTOR TERM1.

10) Стек=“eof EXPR1 TERM1 FACTOR”.Текущий символ = num. Вершина стека нетерминал. Клетка таблицы [FACTOR, num]=FACTOR→num.

11) Стек=“eof EXPR1 TERM1 num”.Текущий символ = num. Вершина стека терминал. Текущий символ равен вершине стека.

12) Стек=“eof EXPR1 TERM1”.Текущий символ = eof. Вершина стека нетерминал. Клетка таблицы [TERM1, eof]=TERM1→ε.

13) Стек=“eof EXPR1”.Текущий символ = eof. Вершина стека нетерминал. Клетка таблицы [EXPR1, eof]=EXPR1→ε.

14) Стек=“eof”. Вершина стека=eof, текущий символ = eof. Следовательно, разбор окончен и последовательность соответствует языку.

Для реализации алгоритма необходимо выполнить следующие действия:

Прежде всего необходимо построить лексический анализатор. Будем строить синтаксический анализатор на основе лексического анализатора, реализованного в лабораторной работе №2 данного учебного пособия.

Вначале объявим тип, соответствующий грамматическим символам. При этом сначала перечислим нетерминалы, затем терминалы. Это нужно для того, чтобы легко проверять, является ли символ терминалом или нетерминалом. Кроме того, этот тип будет включать элемент для отражения конца входной последовательности.

```
type
  symb = (goal, expr, expr1, term, term1, factor, t_num,
        t_id, t_plus, t_minus, t_mult, t_div, eof);
```

Определим таблицу разбора. Ее клетки будут представлять собой грамматики. Заметим, что левую часть грамматики можно опустить, так как она всегда соответствует значению строки. Грамматические символы левой части продукции запишем в обратном порядке для упрощения работы алгоритма. Клетки, соответствующие ошибочной ситуации, будут содержать строку “er”

```
const
Mtable : array [goal..factor, t_num..eof] of string=
(('expr','expr', 'er', 'er', 'er', 'er', 'er'),
 ('expr1 term','expr1 term', 'er', 'er', 'er', 'er', 'er'),
 ('er', 'er', 'expr t_plus','expr t_minus', 'er', 'er', ''),
 ('term1 factor', 'term1 factor', 'er', 'er', 'er', 'er', 'er'),
 ('er', 'er',   '',  '',  'term t_mult',  'term t_div',  ''),
 ('t_num', 't_id',  'er', 'er', 'er', 'er', 'er')
);
```

Для реализации алгоритма необходимо задать стек и указатель на вершину стека.

```
var stack:array [1..100] of symb;     p_s:integer;
```

Для организации стека также потребуются две процедуры: занесения в стек и извлечения из стека.

```
procedure push (value: symb);
begin
  inc(p_s);
  stack[p_s]:=value;
end;

procedure pop;
begin
  dec(p_s);
end;
```

Кроме того, потребуются две вспомогательные функции. Одна необходима для перевода лексем, возвращаемых лексическим анализатором в тип “символ”. Вторая функция необходима для преобразования строк в тип “символ”, так как в таблице хранятся строки, а алгоритм и стек оперируют понятиями грамматических символов.

```
function strtosymb: symb;
begin
if value='goal' then result:=goal;
…
if value='t_num' then result:=t_num;
…
end;
```

```
function lextosymb(value:integer): symb;
begin
case value of
num: result:=t_num;
ord('+'): result:=t_plus;
…
end;
end;
```

6.7 Контрольные вопросы

1) Что представляет собой таблица разбора?

2) Что такое множество first грамматических символов, и каковы правила его построения?

3) Что такое множество follow грамматических символов, для каких символов оно строится и каковы правила его построения?

4) Как строится таблица разбора?

5) В чем состоит алгоритм таблично управляемого синтаксического разбора?

6.8 Задание

1. Написать грамматику для распознавания операции создания таблицы в стандарте SQL.

Создание таблицы начинается с ключевых слов create table, после чего идет имя таблицы и открывающаяся скобка. После этого следует одна или множество конструкций <имя поля> <тип поля> <null | not null>. При этом тип может быть integer или char, если тип символьный, то после char в круглых скобках следует длина строки. После каждой конструкции определения поля следует запятая.

При создании таблицы после определения полей должно следовать объявление первичного ключа. При этом указываются служебные слова primary key и в круглых скобках список полей первичного ключа через запятую.

Таблица может иметь один или множество внешних ключей. Формат объявления внешнего ключа следующий. Foreign key (список полей внешнего ключа) references имя ссылочной таблицы (список полей ссылочной таблицы).

Конструкции первичного ключа и внешних ключей разделяются запятыми. Определение таблицы завершается закрывающейся круглой скобкой.

2. Построить для вышеизложенной грамматики таблицу разбора.

3. Реализовать алгоритм таблично управляемого синтаксического анализатора на основе построенной таблицы разбора.

Части 1 и 2 задания выполняются при подготовке к лабораторной работе.

6.9 Содержание отчета

Отчет о выполнении лабораторной работы должен содержать грамматику для операции создания таблицы в стандарте SQL, а также множества first, follow и таблицу разбора для этой грамматики.

6.10 Защита лабораторной работы

В ходе защиты лабораторной работы студент демонстрирует работу построенного синтаксического анализатора и поясняет состояние стека и входной последовательности на каждом шаге разбора.

Кроме того, студент поясняет, на основании каких фактов, изложенных в задании, были построены продукции грамматики, а также основные моменты построения таблицы разбора.

7 Лабораторная работа № 5 «Синтаксический разбор снизу вверх»

7.1 Цель работы

Целью выполнения лабораторной работы является изучение метода синтаксического разбора “сдвиг-приведение”.

7.2 Разбор снизу вверх

Основная идея разбора снизу вверх состоит в следующем. Находим продукцию, правая часть которой совпадает с фрагментом входной последовательности. Заменяем найденный фрагмент нетерминалом левой части продукции.

При таком разборе применяется метод “сдвиг-приведение”. В процессе разбора этим методом строится дерево разбора входной строки с листьев к корню. Этот процесс можно рассматривать как приведение (свертку) входной строки к стартовому символу грамматики. Если на каждом шаге выбирается правильная подстрока для замены нетерминалом по некоторому правилу, то в обратном порядке прослеживается правосторонний вывод.

Проиллюстрируем эти положения примером. Рассмотрим грамматику, приведенную на рисунке 23.

Пусть необходимо распознать входную последовательность a+b*c. Согласно разбору снизу вверх находим продукцию, которой соответствует фрагмент входной последовательности. Фрагмент “a” соответствует правой части пятой продукции грамматики. После подстановки левой части правила вместо найденного фрагмента получим последовательность FACTOR+b*c. Теперь по четвертому правилу получим последовательность TERM+b*c.

1 EXPR→EXPR TERM
2 EXPR→TERM
3 TEERM→TERM *
FACTOR
4 TERM→FACTOR
5 FACTOR→id

Рисунок 23

Затем применим второе правило и получим последовательность EXPR+b*c. Фрагмент “b” заменим на FACTOR по пятой продукции, а затем FACTOR на TERM по четвертой. Последовательность имеет вид EXPR+TERM*c. После применения пятой продукции получим EXPR+TERM*FACTOR. Если к соответствующему фрагменту применить третью продукцию, то получим EXPR+TERM. Это после применения первой продукции приведет к получению стартового символа.

Если приведенные шаги записать в обратном порядке, то получим правый вывод из стартового нетерминала входной последовательности.

EXPR=>EXPR+TERM=>EXPR+TERM*FACTOR=>EXPR+TER
M*c=>

EXPR+FACTOR*c=>EXPR+b*c=>TERM+b*c=>FACTOR+b*c=
>a+b*c

Основой правой сентенциальной формы γ является продукция A→β и позиция в γ, где находится фрагмент β, который можно заменить на A.

Таким образом, главная задача анализатора, использующего метод “сдвиг-приведение”, состоит в выделении и отсечении основы. Такие анализаторы являются LR(k)-анализаторами.

7.3 Структура LR(k)-анализаторов

В названии LR(k) символ L означает, что разбор осуществляется слева направо, R - что строится правый вывод в обратном порядке, k – число входных символов, на которые заглядывает вперед анализатор.

Структура LR(k)-анализатора представлена на рисунке 24.

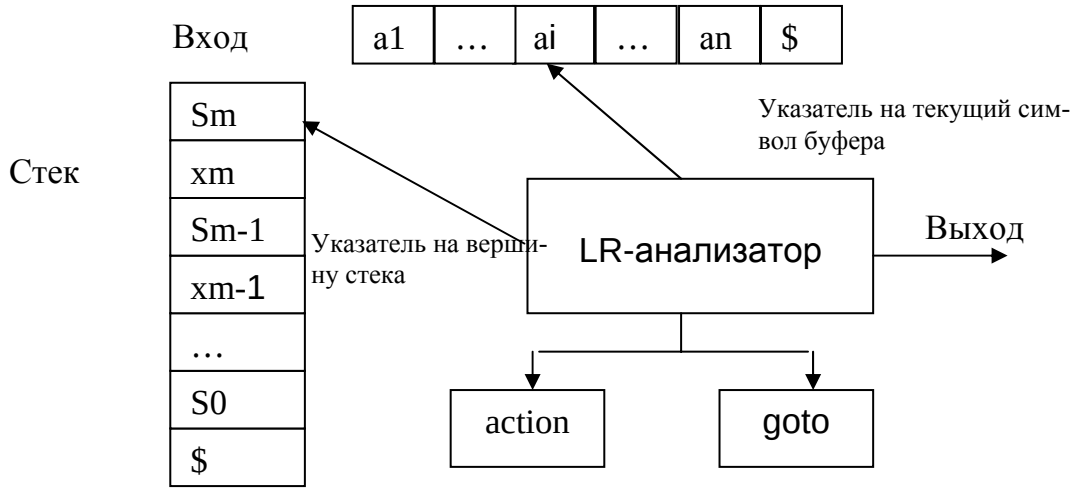


Рисунок 24

LR-анализатор состоит из входа, выхода, стека, управляющей программы и таблиц анализа. Таблиц анализа две. Управляющая программа одна и та же для всех анализаторов, различаются только таблицы. Программа анализатора читает символы из входного буфера по одному за шаг. В процессе анализа используется стек, в котором хранятся пары вида xiSi, где xi – символ терминала или нетерминала грамматики, а Si – символ, характеризующий состояние автомата. Каждый символ состояния выражает информацию, содержащуюся в стек ниже него, а комбинация символа состояния в вершине стека и текущего символа входной последовательности используется для индексации таблиц анализа.

Фактически анализатор представляет собой конечный автомат, который на основе текущего символа входной последовательности и состояния автомата в вершине стека, выполняет действие по таблице action и переходит в новое состояние по таблице goto.

Анализатор может выполнять четыре действия:

1) Сдвиг (shift s). Это действие заключается в занесении в стек текущего символа и состояния s.

2) Приведение (reduce $A \rightarrow \beta$). Это действие заключается в подрезке основы (замене основы на нетерминал) по правилу $A \rightarrow \beta$.

3) Входная последовательность принята (accept). Разбор окончен, входная последовательность соответствует грамматике.

4) Ошибка (error). При разборе произошла ошибка.

При LR-разборе исходную грамматику дополняют правилом S′→S, где S – стартовый нетерминал исходной грамматики. Это необходимо для определения момента выполнения операция accept. Эта операция должна выполняться при приведении по правилу S′→S. На рисунке 25 представлена дополненная грамматика, исходной для которой служила грамматика с рисунка 23.

1 EXPR′→EXPR
2 EXPR→EXPR + TERM
3 EXPR→TERM
4 TEERM→TERM * FACTOR
5 TERM→FACTOR
6 FACTOR→id

Рисунок 25

7.4 Построение таблиц анализа

7.4.1 Понятие ситуации

Как уже отмечалось, LR-анализатор представляет собой конечный автомат. Состояниями недетерминированного конечного автомата являются ситуации. Ситуация – это продукция из заданной грамматики, в которую помещена точка, отделяющая распознанную часть от нераспознанной.

Например, если в грамматике есть продукция A→xyz, то для нее возможна ситуация [A→xy·z], которая соответствует состоянию автомата, в котором анализатор получил на входе x и y и теперь ожидает z. Вообще для продукции A→xyz возможны следующие ситуации: [A→·xyz], [A→x·yz], [A→xy·z], [A→xyz·]. Для продукции вида A→ε возможна только одна ситуация [A→·].

Теперь необходимо сгруппировать ситуации, чтобы получить детерминированный конечный автомат.

7.4.2 Каноническая совокупность множеств ситуаций

Вначале введем понятие замыкание множества ситуаций I (closure(I)). Замыкание I строится по двум правилам:

1) В замыкание I заносится множество ситуаций I.

2) Если ситуация [A→α·Bβ] уже принадлежит closure(I) и есть продукция B→γ, то добавляем в closure(I) ситуацию [B→·γ].

Теперь рассмотрим переходы (goto (I,x)). Переход представляет собой множество ситуаций, в которые можно перейти из множества ситуаций I по грамматическому символу x, то есть по терминалу или нетерминалу. При выполнении функции (goto (I,x)) строят замыкание множества состояний вида [A→αx·β], если в I есть ситуация [A→α·xβ], то есть точка переходит за символ x в ситуациях. После этого строится замыкание этого множества.

Используя функции closure(I) и goto(I,x), можно построить каноническую совокупность всех возможных множеств ситуаций дополненной грамматики. Для этого строят замыкание для ситуации [S′→·S]. После этого в цикле находят все множества ситуаций, в которые можно перейти из уже найденного множества по любому грамматическому символу. Цикл завершается, когда нельзя ничего нового добавить в каноническую совокупность.

Построим каноническую совокупность возможных ситуаций для грамматики с рисунка 25.

Замыкание ситуации [EXPR′ → ·EXPR] = {[EXPR′ → ·EXPR], [EXPR → ·EXPR+TERM], [EXPR → ·TERM], [TERM → ·TERM*FACTOR], [TERM → ·FACTOR], [FACTOR→·id]}. Обозначим это множество как I_0.

Рассмотрим, в какие множества ситуаций можно перейти из I_0 по различным грамматическим символам. По символу EXPR можно перейти в множество {[EXPR′ → EXPR·], [EXPR → EXPR·+TERM]}. Замыкание этого множества будет равно самому этому множеству. Обозначим полученное множество как I_1. По грамматическому символу TERM из I_0 можно перейти во множество I_2={[EXPR → TERM·], [TERM → TERM·*FACTOR]}. По символу FACTOR осуществляется переход в множество I_3={[TERM → FACTOR·]}. Аналогично по символу id переходим в множество ситуаций I_4={[FACTOR→id·]}. Все переходы из множества I_0 рассмотрены.

Теперь рассмотрим множество I_1. Из него возможен переход только по символу + в множество {[EXPR → EXPR+·TERM]}. В каноническую совокупность добавляем множество I_5, представляющее собой замыкание множества {[EXPR → EXPR+·TERM]}, то есть I_5={[EXPR → EXPR+·TERM], [TERM → ·TERM*FACTOR], [TERM → ·FACTOR], [FACTOR → ·id]}.

Из множества I_2 существует переход только по символу *. Построение замыкания полученного множества приведет к занесению в

каноническую совокупность множества I_6={[TERM → TERM*·FACTOR], [FACTOR → ·id]}

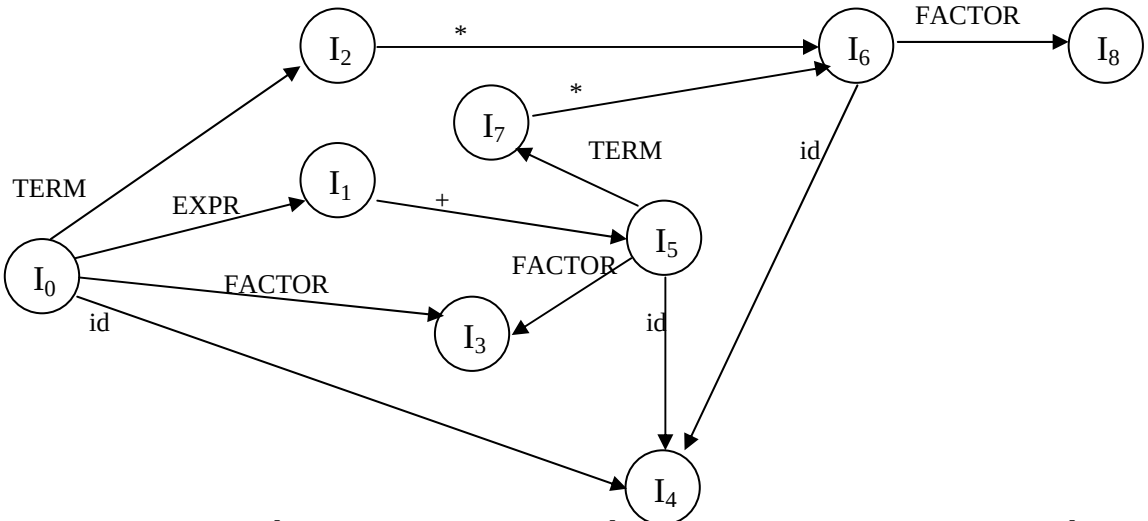
Из множеств I_3 и I_4 нельзя выполнить ни один переход.

Рассматривая символ TERM и множество I_5, добавим в каноническую совокупность I_7={[EXPR → EXPR+TERM·], [TERM → TERM·*FACTOR]}. По символу FACTOR из множества I_5 можно перейти в множество {[TERM → FACTOR·]}, то есть во множество I_3. Таким образом, на данном шаге ничего нового в каноническую совокупность не добавляем. Рассмотрение символа id для множества I_5 также не приведет к появлению нового множества, так как переход будет осуществлен во множество I_4.

Для множества I_6 характерны переходы по символам FACTOR и id. В первом случае к канонической совокупности добавляется множество I_8={[TERM → TERM*FACTOR·]}. По символу id выполняется переход в уже существующее множество I_4.

Из множества I_7 по символу * осуществляется переход в состояние I_6. Из множества I_8 переходов нет.

Результирующая каноническая совокупность изображена на рисунке 26.



I_0={[EXPR′ → ·EXPR], [EXPR → ·EXPR+TERM], [EXPR → ·TERM], [TERM → ·TERM*FACTOR], [TERM → ·FACTOR], [FACTOR→·id]};

I1={[EXPR′ → EXPR·], [EXPR → EXPR·+TERM]}; I2={[EXPR → TERM·], [TERM → TERM·*FACTOR]};

I3={[TERM → FACTOR·]};

I4={[FACTOR→id·]};

I5={[EXPR → EXPR+·TERM], [TERM → ·TERM*FACTOR], [TERM → ·FACTOR], [FACTOR → ·id]}

I6={[TERM → TERM*·FACTOR], [FACTOR → ·id]}

I7={[EXPR → EXPR+TERM·], [TERM → TERM·*FACTOR]}

I8={[TERM → TERM*FACTOR·]}

Рисунок 26

7.4.3 Заполнение таблиц action и goto

Таблица action в зависимости от пары <входной сигнал, состояние> определяет, достигнут ли конец основы. Если конец основы достигнут, то выполняется приведение, в противном случае выполняется сдвиг.

Таблица goto в зависимости от пары <нетерминал в вершине стека, состояние> определяет состояние конечного автомата после выполнения приведения.

Алгоритм заполнения таблиц анализа заключается в следующем:

1) Построить каноническую совокупность множества ситуаций.

2) Каждое множество I канонической совокупности определяет состояние конечного автомата с соответствующим номером, а следовательно, строку в таблицах анализа. Ячейки в таблице заполняются по следующим вариантам:

а) Если ситуация $[A \rightarrow \alpha \cdot a\beta]$ принадлежит множеству I_i и существует переход по терминалу a в некоторое множество I_j, то в ячейку action(i, a) значение shift j. В приведенной ситуации основа не может быть найдена, так как для ее завершения как минимум необходимо распознать терминал a, то есть выполнить сдвиг.

б) Если ситуация $[A \rightarrow \alpha \cdot]$ принадлежит множеству I_i, то для всех терминалов a, принадлежащих follow(A), в ячейку action[i, a] заносим значение reduce $A \rightarrow \alpha$. Это происходит, потому что основа найдена полностью и распознавание любого символа, следующего за A, должно привести к приведению основы.

в) Если ситуация $[S' \rightarrow S \cdot]$ принадлежит множеству I_i, то есть можно осуществить редукцию для стартовой продукции, то в ячейку action[I, eof] записываем значение accept. То есть разбор успешно закончен, если достигнут конец входной последовательности.

3) Если существует переход из множества I_i в множество I_j по некоторому нетерминалу A, то в ячейку goto[i, A] заносим значение j.

4) Все незаполненные ячейки в таблицах action и goto заполняем значением “ошибка”.

5) Начальным состояние автомата является состояние, содержащее ситуацию [S′→S]

Построим таблицы разбора для грамматики с рисунка 25. Каноническая совокупность для этой грамматики приведена на рисунке 26. Для построения таблицы action необходимо построить множество follow для нетерминалов грамматики. Это множество приведено в таблице 7.

Таблица 7 – Множество follow для упрощенной грамматики арифметических действий

Нетерминал	Шаги				follow
	1	2	3	4	
EXPR′	{eof}				{eof}
EXPR		{+}	{eof}		{+, eof}
TERM		{*}	{+, eof}		{*, +, eof}
FACTOR			{*, +, eof}		{*, +, eof}

В канонической совокупности девять множеств, то есть в конечном автомате будет девять состояний с S0 по S8, столько же строк будет в таблицах анализа.

Рассмотрим переходы по терминалам в канонической совокупности. Из множества I_0 по терминалу id есть переход во множество I_4, а ситуация [FACTOR→·id] принадлежит I_0. Поэтому action[S0, id]:=shift S4. Множеству I_1 принадлежит ситуация [EXPR→EXPR·+TERM], и есть переход из I_1 в I_5 по входному символу +, поэтому заносим значение shift S5 в ячейку action[S1, +]. Из I_2 существует переход в I_6 по символу *, ситуация [TERM→TERM·*FACTOR] принадлежит множеству I_2, следовательно, action[S2, *]:=shift S6. Из множеств I_5 и I_6 в канонической сово-

купности существуют переходы во множество I_4 по символу id. Ситуация [FACTOR→·id] принадлежит обоим этим множествам. Поэтому значение shift S4 заносим в ячейки action[S5, id] и action[S6, id]. Множеству I_7 принадлежит ситуация [TERM→TERM·*FACTOR], и существует переход из множества I_7 во множество I_6, поэтому записываем значение shift S6 в ячейку action[S7, *].

Ситуация [EXPR′→EXPR·] принадлежит множеству I_1, follow(EXPR′)={eof}, поэтому должно было быть выполнено присвоение action[S1, eof]:=reduce 1, то есть приведение по первой продукции грамматики. Однако эта ситуация соответствует стартовой продукции, следовательно, action[S1, eof]:=accept. Множеству I_2 канонической совокупности принадлежит ситуация [EXPR→TERM·], follow(EXPR)={+, eof}, следовательно, action[S2, +] и action[S2, eof] принимают значение reduce 3. Множество I_3 содержит ситуацию [TERM→FACTOR·], follow(TERM)={*, +, eof}, поэтому значение reduce 5 заносим в ячейки action[S3, *], action[S3, +] и action[S3, eof]. I_4 содержит ситуацию [FACTOR→id·], follow(FACTOR)={*, +, eof}, поэтому ячейки action[S4, *], action[S4, +] и action[S4, eof] будут содержать значение reduce 6. Наличие ситуации [EXPR→EXPR+TERM·] во множестве I_7 приводит к записи значения reduce 2 в ячейки action[S7, +] и action[S7, eof]. Аналогично, ситуация [TERM→TERM*FACTOR·] во множестве I_8 обосновывает наличие значения reduce 4 в ячейках action[S8, *], action[S8, +] и action[S8, eof].

Теперь рассмотрим переходы по нетерминальным символам между множествами канонической совокупности. Из множества I_0 есть такие переходы в состояние I_1 по нетерминалу EXPR, I_2 по нетерминалу TERM и I_3 по символу FACTOR, поэтому в ячейку goto[S0, EXPR] записываем значение S1, goto[S0, TERM]:=S2 и goto[S0, FACTOR]:=S3. Также переходы по нетерминальным символам существуют из множества I_5 в множества I_3 и I_7. Поэтому goto[S5, FACTOR]:=S3, а goto[S5, TERM]:=S7. В таблице переходов будет еще одно значение, goto[S6, FACTOR]:=S8, так как в канонической совокупности есть переход из I_6 в I_8 по символу FACTOR.

В незаполненные до сих пор ячейки заносим информацию об ошибке. Построенные таблицы приведены ниже.

Таблица 8 – Таблица action для упрощенной грамматики арифметических операций

Состояние	id	+	*	eof
S0	shift S4	-	-	-
S1	-	shift S5	-	accept
S2	-	reduce 3	shift S6	reduce 3
S3	-	reduce 5	reduce 5	reduce 5
S4	-	reduce 6	reduce 6	reduce 6
S5	shift S4	-	-	-
S6	shift S4	-	-	-
S7	-	reduce 2	shift S6	reduce 2
S8	-	reduce 4	reduce 4	reduce 4

Таблица 9 – Таблица goto для упрощенной грамматики арифметических операций

Состояние	EXPR	TERM	FACTOR
S0	S1	S2	S3
S1	-	-	-
S2	-	-	-
S3	-	-	-
S4	-	-	-
S5	-	S7	S3
S6	-	-	S8
S7	-	-	-
S8	-	-	-

7.5 Алгоритм LR-разбора

Алгоритм LR-разбора состоит в следующем.

Занести в стек символ конца входной последовательности (eof)

Занести в стек начальное состояние конечного автомата

Распознать лексему

repeat

Если action[состояние в вершине стека, лексема]=shift S

то

занести в стек лексему и состояние S, после чего распознать следующую лексему

Если action[состояние в вершине стека, лексема]=reduce P

то

извлечь из стека столько пар <состояние, грамматический символ>, сколько грамматических символов составляют правую часть правила P. В стек занести нетерминал левой части продукции P и goto[состояние в вершине стека, нетерминал левой части продукции P]

Если action[состояние в вершине стека, лексема]=accept

то

вывести пользователю сообщение об успешном разборе и выйти из цикла

Если action[состояние в вершине стека, лексема]=ошибка

то

обработать ошибочную ситуацию

until false

Рассмотрим, как работает алгоритм для входной последовательности x*y+z. Грамматика представлена на рисунке 25, таблицы action и goto приведены выше под номерами 8 и 9. Рассмотрим этапы разбора:

1) В начале работы алгоритма в стек заносим символ конца входной последовательности и стартовое состояние. Таким образом, стек= “eof S0”. Распознанная лексема = id.

2) action[S0, id]= “Shift S4”, то есть необходимо выполнить сдвиг и занести в стек символ id и состояние 4. Распознаем следующую лексему. Это символ *. В вершине стека находится состояние S4.

3) action[S4,*]=reduce 6. Шестое правило грамматики имеет вид FACTOR→id. При приведении по нему из стека извлекается верхняя пара <состояние, символ>, так как правая часть продукции состоит из одного символа. Теперь в вершине стека находится состояние S0. В стек заносим левую часть шестой продукции и состояние, найденное как goto[S0, FACTOR]. В результате стек имеет вид “eof S0 FACTOR S3”.

4) action[S3, *]=reduce 5. Пятая продукция имеет вид TERM→FACTOR. После извлечения стек имеет вид “eof S0”. goto[S0, TERM]=S2. Таким образом, стек= “eof S0 TERM S2”.

5) action[S2,*]=shift S6. Стек= “eof S0 TERM S2 * S6”. Следующей распознанной лексемой является id.

6) action[S6,id]=shift S4. Стек= “eof S0 TERM S2 * S6 id S4”. Лексема=+.

7) action[S4,+]=reduce 6. Стек= “eof S0 TERM S2 * S6 FACTOR S8”.

8)action[S8,+]=reduce 4. Четвертая продукция имеет вид TERM→TERM*FACTOR. Длина правой части равна 3, поэтому из стека извлекаем три пары <состояние, символ>. После выполнения приведения стек имеет вид “eof S0 TERM S2”.

8) action[S2, +]=reduce 3. Третья продукция имеет вид EXPR→TERM. Стек после выполнения приведения равен “eof S0 EXPR S1”.

9) action[S1, +]=shift S5. Стек= “eof S0 EXPR S1 + S5”. Распознанная лексема=id.

10) action[S5, id]=Shift S4. Стек=“eof S0 EXPR S1 + S5 id S4”. Лексема=eof.

11) action[S4,eof]=reduce 6. Стек = “eof S0 EXPR S1 + S5 FACTOR S3”.

12) action[S3,eof]=reduce 5. Стек= “eof S0 EXPR S1 + S5 TERM S7”.

13) action[S7,eof]=reduce 2. Стек= “eof S0 EXPR S1”.

14) action[S1,eof]=accept.

Таким образом, разбор завершен успешно, входная последовательность соответствует распознаваемому языку

/*Редукции по правилу EXPR′→EXPR не производилось. В вершине стека не стартовый символ, или стартовый, но без дополненной грамматики. В примере из лекций то же самое. Зачем нужна стартовая продукция? И как тогда работает Yacc, который выполняет приведение по продукции goal : expr? */

Для реализации этого алгоритма выполним следующее.

Объявляем тип для отражения состояний автомата. Дополнительно элементом этого типа будет s_er, необходимый для заполнения ошибочных ячеек таблицы action. Еще один тип объявим для отражения грамматических символов.

```
type
state=(s0, s1,s2,s3,s4,s5,s6,s7,s8, s_er);
symb = (expr1, expr, term, factor, t_id, t_plus, t_mult, eof);
```

Для работы программы нужно будет написать функцию, переводящую значение типа integer в значение типа symb. Это связано с тем, что лексический анализатор возвращает значение типа integer, а

синтаксический анализатор будет работать с собственным типом для грамматических символов.

Грамматики будем хранить в виде массива записей. Каждая запись будет содержать нетерминал левой части и количество символов в правой. Сам текст продукций для работы алгоритма не важен, он уже был использован для построения таблиц анализа. Таким образом, для хранения грамматики выполним следующие объявления.

```
type
  gramm_type=record
  noterm:symb;
  basic_length:integer;
  end;

const
gramm:array[1..6] of gramm_type=
((noterm:expr1; basic_length:1),
 (noterm:expr; basic_length:3),
 (noterm:expr; basic_length:1),
 (noterm:term; basic_length:3),
 (noterm:term; basic_length:1),
 (noterm:factor; basic_length:1)
);
```

Стек должен содержать пары <состояние, символ>. Для его реализации можно написать следующий код.

Декларируем тип элементов массива

```
type
  stack_type=record
  stack_state:state;
  stack_symb:symb;
  end;
```

Объявляем переменные стек и указатель на вершину стека

```
var stack:array[1..100] of stack_type; p_s:integer;
```

Реализуем процедуры извлечения из стека и занесения в стек

```
procedure pop;
begin
  dec(p_s);
end;

procedure push(value_state:state;value_symb:symb);
begin
  inc(p_s);
  stack[p_s].stack_state:=value_state;
  stack[p_s].stack_symb:=value_symb;
end;
```

7.6 Контрольные вопросы

1) В чем состоит основная идея синтаксического разбора сверху вниз?

2) Чему соответствует обратный порядок построения дерева разбора метода “сдвиг-приведение”?

3) Что такое основа?

4) Какова структура LR(k)-анализатора?

5) Какие действия выполняет LR-анализатор?

6) Что такое ситуация?

7) Чему в терминах конечного автомата соответствует ситуация и каноническая совокупность ситуаций?

7.7 Задание

1) Построить грамматику для распознавания запроса по одной таблице в стандарте SQL. Запрос начинается ключевым словом select, за которым через запятую следуют имена полей. После этого списка следует служебное слово from и имя таблицы. В запросе может присутствовать раздел условий, начинающийся со служебного слова where. За ним следует список условий. Каждое условие имеет вид “имя поля <|>|= значение”. Условия соединяются операциями or или and.

2) Построить таблицы анализа для LR(1)-разбора.

3) Реализовать алгоритм LR-разбора для построенной грамматики.

Части 1 и 2 задания выполняются студентом при подготовке к лабораторной работе.

7.8 Содержание отчета

Отчет должен содержать грамматику распознаваемого в ходе лабораторной работы языка, а также таблицы action и goto для синтаксического анализатора.

7.9 Защита лабораторной работы

В ходе защиты лабораторной работы студент демонстрирует работу построенного синтаксического анализатора и поясняет фрагменты написанного кода, а также основные моменты построения таблиц разбора. Демонстрацию работы анализатора необходимо сопроводить описанием состояния стека на каждом шаге распознавания входной последовательности.

8 Лабораторная работа № 6 «Генерация кода и реализация машины»

8.1 Цель работы

Целью данной лабораторной работы является:

– получение навыков построения генератора кода на основе синтаксического разбора методом рекурсивного спуска;

– получение опыта реализации машины, обрабатывающей сгенерированный код.

8.2 Построение транслятора на основе синтаксического анализа методом рекурсивного спуска

Задача генератора кода – построение эквивалентной машиной программы по программе на входном языке.

Таким образом, для реализации транслятора необходимо определить язык машины и построить генератор, который преобразует входной язык к машинному языку.

8.2.1 Входной язык

Будем рассматривать в качестве входного для генератора кода язык арифметических выражений. Рассмотрим грамматику этого языка. Входной язык представляет собой раздел переменных и раздел кода

1. LANGUAGE → VARIABLE PROGRAM

Раздел переменных – это служебное слово var, описание переменной и список описания переменных.

2. VARIABLE→ var VAR_DESCRIBE LIST_VAR

Описание переменной – это ее имя (идентификатор), двоеточие и начальное значение. Описание переменной заканчивается символом “;”.

3. VAR_DESCRIBE → id : num ;

Список переменных – это запятая и описание переменной. Список может быть пустым.

4. LIST_VAR→ VAR_DISCRIBE LIST_VAR
5. LIST_VAR→ε

Раздел кода – это список команд, заключенный между ключевыми словами begin и end.

6. PROGRAM→ begin LIST_ INSTRUCTION end

Список команд - это команды, приведенные через разделитель (;) или пустой список.

7. LIST_ INSTRUCTION→ INSTRUCTION ; LIST_ INSTRUCTION
8. LIST_ INSTRUCTION→ε

Команда это оператор чтения или оператор присвоения

9. INSTRUCTION→READ_INSTR
10. INSTRUCTION→ASSIGN_INSTR

Оператор чтения описывается продукцией 11, а продукции 12-22 описывают оператор присвоения

11. READ_INSTR → read (id)
12. ASSIGN_INSTR→id := EXPR
13. EXPR→TERM EXPR1
14. EXPR1→ + TERM EXPR1

15. EXPR1→ - TERM EXPR1
16. EXPR1→ε
17. TERM→ FACTOR TERM1
18. TERM1→ * FACTOR TERM1
19. TERM1→/ FACTOR TERM1
20. TERM1→ε
21. FACTOR→id
22. FACTOR→num

Будем считать, что синтаксический анализатор для этой грамматики построен с использованием метода рекурсивного спуска.

8.2.2 Структура машины

Машина состоит из области данных (таблицы переменных), кода и стека.

Область данных будет хранить все переменные, которые используются в программе и их типы.

Таблицу переменных для наглядности будем хранить в объекте sgData класса TStringGrid с двумя столбцами: имя и значение.

Стек будет использован в данном трансляторе только для хранения чисел, поэтому зададим его в качестве массива. Кроме того, необходимо определить способы работы со стеком.

```
var
  stack:array [1..100] of real;
  p_s:integer;

function pop:real;
begin
  result:=stack[p_s];
  dec(p_s);
end;

procedure push(value:real);
begin
  inc(p_s);
  stack[p_s]:=value;
end;
```

Машина, работающая с языком арифметических выражений, будет иметь следующие команды.

ldc (value) – поместить в стек константу (число), передаваемую в качестве параметра.

```
procedure ldc(value:real);
begin
  push(value);
end;
```

ldv (value) – поместить в стек значение переменной, передаваемой в качестве параметра. Эта команда производит поиск по таблице переменных, если переменная присутствует в таблице, то в стек помещается ее значение. Если переменная не найдена, то порождается ошибочная ситуация.

```
procedure ldv(value:string);
var i:integer; f:boolean;
begin
  i:=0;  f:=true;
  while (i<sgData.RowCount) and f do
  begin
    if sgData.cells[0,i]=value
    then
    begin
      push(strtofloat(sgData.cells[1,i]));
      f:=false;
    end
    else inc(i);
  end;
  if f
  then raise exception.Create('Неизвестный идентификатор');
end;
```

add – сложение.
sub – вычитание.
mult – умножение.
div – деление.

Операции арифметических действий выполняются одинаково. Они заключаются в чтении из стека двух значений, выполнения операции и записи результата в стек. Приведем в качестве примера процедуру для выполнения вычитания.

```
procedure TForm1.sub;
var a1,a2:real;
begin
  a1:=pop;   a2:=pop;
  push(a2-a1)
end;
```

assignment(value) – присвоение значения переменной, полученной в качестве параметра. Эта команда читает значение из стека и записывает прочитанную величину в качестве значения переменной в таблицу переменных.

```
procedure assignment(value:string);
var i:integer; f:boolean;
begin
  i:=0; f:=true;
  while (i<sgData.RowCount) and f do
  begin
    if sgData.cells[0,i]=value
    then
    begin
      sgData.cells[1,i]:=floattostr(pop);
      f:=false;
    end
    else inc(i);
  end;
  if f
  then raise exception.Create('Неизвестный идентификатор')
end;
```

read(value) – чтение значения переменной, передаваемой в качестве параметра. Для реализации такого диалога с пользователем будем использовать объект класса Tform, создаваемый при необходимости ввода и разрушаемый после вывода. На такой форме будут по-

мещаться объекты класса TLabel для вывода имени переменной и класса TEdit для ввода пользователем значения переменной. Если переменной, значение которой необходимо вести нет в таблице переменных, то порождается ошибочная ситуация.

```
procedure read;
var form:Tform; edit:Tedit; label1:Tlabel;
   i:integer; f:boolean;
begin
  i:=0;  f:=true;
  while (i<sgData.RowCount) and f do
  begin
    if sgData.cells[0,i]=value
    then
    begin
      push(strtofloat(sgData.cells[1,i]));
      f:=false;
    end
    else inc(i);
  end;
  if f
  then raise exception.Create('Неизвестный идентификатор')
  else
  begin
    form:=tform.Create(application);
    edit:=tedit.Create(form);
    edit.Parent:=form;
    edit.top:=20;
    label1:=tlabel.Create(form);
    label1.Parent:=form;
    label1.caption:=sgData.Cells[0,i];
    form.ShowModal;
    sgData.Cells[1,i]:=edit.text;
    form.Destroy;
  end;
end;
```

8.2.3 Генерация кода из синтаксического анализатора

Таблицу переменных необходимо сформировать при декларации переменных во входной последовательности (продукции 2-5). Поэтому заполнение таблицы будем производить в соответствующей функции синтаксического анализатора. Ниже приведен код функции синтаксического анализатора, которая производит заполнение таблицы переменных. В коде подчеркнуты строки, которые дописаны к синтаксическому анализатору для генерации кода.

```
function var_describe;
var help:string;
begin
  help:=yytext;
  if (lex=id) and (yylex=ord(':')) and (yylex=num)
  then
  begin
    sgData.cells[0,sgData.RowCount-1]:=help;
    sgData.cells[1,sgData.RowCount-1]:=yytext;
    if yylex=ord(';')
    then
    begin
      sgData.RowCount:=sgData.RowCount+1;
      lex:=yylex;
      result:=true;
    end
    else result:=false;
  end
  else result:=false;
end;
```

Теперь перейдем непосредственно к формированию кода. Программа должна будет выполняться снизу вверх и слева направо по дереву разбора, которое строится при синтаксическом разборе. С каждым узлом дерева может быть связана какая-либо операция. Операция некоторого узла дерева может быть выполнена тогда, когда выполнены все необходимые операции других узлов.

Если построить дерево разбора для входной последовательности, соответствующей приведенной выше грамматике, то в самом низу этого дерева окажутся терминалы id и num, выведенные из нетер-

минала FACTOR. Поэтому в функцию factor синтаксического анализатора встраиваем генератор. Если распознано число, то оно помещается в стек, если распознан идентификатор, то в стеке необходимо сохранить значение этой переменной.

Ниже приведена функция factor синтаксического анализатора со встроенным генератором кода. При этом код хранится в объекте sgCode: TStringGrid. Первый столбец этого объекта хранит имя команды машины, а остальные два предназначены для хранения параметров команды, если они предусмотрены машиной.

```
function factor;
begin
  case lex of
  id:
  begin
    sgCode.Cells[0, sgCode.RowCount-1]:='ldv';
    sgCode.Cells[1, sgCode.RowCount-1]:=yytext;
    sgCode.RowCount:=sgCode.RowCount+1;
    result:=true;
    lex:=yylex;
  end;
  num:
  begin
    sgCode.Cells[0, sgCode.RowCount-1]:='ldc';
    sgCode.Cells[1, sgCode.RowCount-1]:=yytext;
    sgCode.RowCount:=sgCode.RowCount+1;
    result:=true;
    lex:=yylex;
  end;
  else result:=false;
  end;
end;
```

Код, соответствующий операциям сложения, вычитания, умножения и деления, встраивается одинаковым образом. Рассмотрим это на примере вычитания. Вычитание заключается в извлечении из стека двух значений, вычитании из первого значения второго и сохранения результата в стеке.

Приведем фрагмент дерева разбора для входной строки 5-2-3.

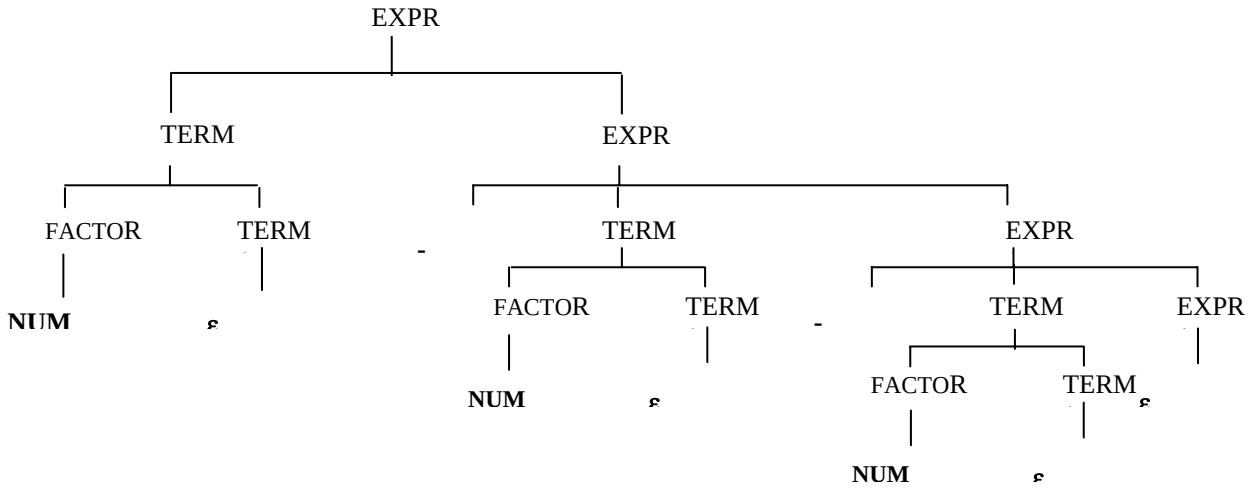


Рисунок 27

Операцию вычитания имеет смысл встроить в функцию, соответствующую узлу EXPR1. Если эту операцию встроить после выполнения всех поддеревьев узла EXPR1, то получится следующая ситуация. При выполнении кода сначала выполнятся действия, соответствующие левому поддереву узла EXPR. В результате в стеке окажется идентификатор 5 (операция ldc, встроенная в функцию factor). Далее выполнятся операции, связанные с центральным поддеревом EXPR1. После этого в стеке окажется идентификатор 5 и идентификатор 2. Операция вычитания, которая должна быть выполнена на этом этапе согласно математическим правилам, выполнена не будет, так как вначале должны выполняться операции правого поддерева узла EXPR1. При выполнении правого поддерева в стеке окажутся идентификаторы 5, 2 и 3. После этого выполнится операция вычитания, соответствующая нижнему узлу EXPR1, что приведет к наличию в стеке значений 5 и –1. Теперь все поддеревья верхнего узла выполнены и выполняется последняя операция вычитания. При этом результат вычислений равен 6, а должен быть равен 0.

Поэтому операцию вычитания встроим в узел EXPR1 после доказательства центрального поддерева, но до доказательства правого. Ниже приведен код функции, соответствующей узлу EXPR1. Заметим, что сложение можно выполнять после выполнения всех поддеревьев узла EXPR1, так как порядок слагаемых на результат не влияет.

```
function expr1;
begin
```

```
  case lex of
  ord('+'):
    begin
      lex:=yylex;
      result:=term and expr1;
      sgCode.Cells[0,sgCode.rowcount-1]:='add';
      sgCode.RowCount:=sgCode.RowCount+1;
    end;
  ord('-'):
    begin
      lex:=yylex;
      if term
      then
      begin
        sgCode.Cells[0,sgCode.rowcount-1]:='sub';
        sgCode.RowCount:=sgCode.RowCount+1;
        result:=expr1;
      end
      else result:=false
    end;
  else result:=true
  end;
end;
```

8.2.4 Работа машины

Для того, чтобы машина могла обрабатывать сгенерированный код, напишем процедуру, которая читает код и выполняет соответствующие команды машины.

```
procedure execute;
var i:integer;
begin
  for i:=0 to sgCode.RowCount-2 do
  begin
    if sgCode.Cells[0,i]='ldc'
    then ldc(strtofloat(sgCode.Cells[1,i]));

    if sgCode.Cells[0,i]='ldv'
    then ldv(sgCode.Cells[1,i]);
```

```
    if sgCode.Cells[0,i]='add'
    then add;

   …
  end;
end;
```

Основная программа теперь должна включать код для установки начальной работы лексического анализатора, очищать таблицу переменных, область кода и стек. После этого должен выполняться синтаксический анализ входной последовательности. В случае соответствия входной последовательности грамматики производится выполнение кода.

8.3 Контрольные вопросы

1) В чем состоит задача генератора кода?

2) Из каких составных частей состоит машина?

3) В каком направлении по дереву разбора выполняется программа?

4) Каким образом влияет положение генератора кода в функции узла дерева? Пояснить на примере операции вычитания.

8.4 Задание

Реализовать транслятор, встроив генератор кода в синтаксический анализатор, построенный в лабораторной работе № 3.

Сложность при реализации может вызвать команда If. Для того, чтобы эта команда работала корректно, необходимо ввести еще одну команду машины goto(номер строки). Эта команда должна перенести выполнение кода с текущей строки на строку, указанную в качестве параметра. Команда if должна иметь в качестве аргумента строку, на которую должно передаваться управление в том случае, если условие ложно. Часть кода, соответствующая истинному условию, должна заканчиваться командой goto, передающей управление на строку, следующую за кодом, обрабатывающим ложное условие.

8.5 Защита лабораторной работы

В ходе защиты лабораторной работы студент демонстрирует работу построенного анализатора и поясняет основные фрагменты генерации кода. При пояснении работы транслятора особое внимание

должно уделяться состоянию стека на различных шагах работы программы.

9 Лабораторная работа № 7 «Автоматизированное построение трансляторов с использованием генератора YACC»

9.1 Цель работы

Целью выполнения лабораторной работы является:

– закрепление теоретических знаний в области атрибутных грамматик;

– получение опыта работы со средствами автоматизированного построения трансляторов на примере генератора компиляторов Yacc.

9.2 Общие сведения

Одним из средств автоматизированного построения трансляторов является генератор компиляторов Yacc.

Yacc строит синтаксический анализатор, работающий на основе метода “сдвиг - приведение”.

Этот генератор работает с атрибутными грамматиками. Атрибутные грамматики характеризуются тем, что с каждым грамматическим символом связывается множество атрибутов.

Атрибуты в атрибутных грамматиках могут быть двух видов: синтезируемые и наследуемые. Синтезируемые атрибуты вычисляют свои значения, используя только значения атрибутов потомков. Наследуемые атрибуты при вычислении значений используют значения атрибутов соседей и родителей.

С каждой продукцией атрибутной грамматики связывается семантическое правило. Такое правило выполняется в тот момент, когда происходит приведение по соответствующей продукции.

Исходный текст для Yacc, основой которого являются продукции и семантические правила, пишется в текстовом редакторе, например, в блокноте, и сохраняется в файле с расширением y. Работа программы yacc.exe, в качестве параметра которой выступает исходный файл с расширением y, строит транслятор. Yacc строит транслятор на основе кода, содержащегося в файле Yyparse.cod. Если исходный текст содержит ошибки, то при генерации транслятора будет создан файл ошибок с расширением lst.

Полученный в результате работы Yacc модуль присоединяется к проекту Delphi, который содержит текст основной программы. Для

корректной работы с этим проектом он должен использовать библиотеку Yacclib.pas.

Транслятор строится Yacc в виде функции yyparse, которая вызывается из основной программы.

При генерации транслятора Yacc включает код лексического анализатора, построенного Lex. На рисунке 28 приведена схема взаимодействия модулей компилятора.

Исходный текст для yacc состоит из трех частей: определения, продукции и вспомогательных процедур. Части разделяются символами %%.

В части определений можно указать код, который должен быть перенесен в текст результирующего транслятора без изменения. Для этого он помещается между символами %{ и %}. В качестве такого кода могут выступать заголовок модуля или объявления типов, используемые затем для описания атрибутов терминалов и нетерминалов.

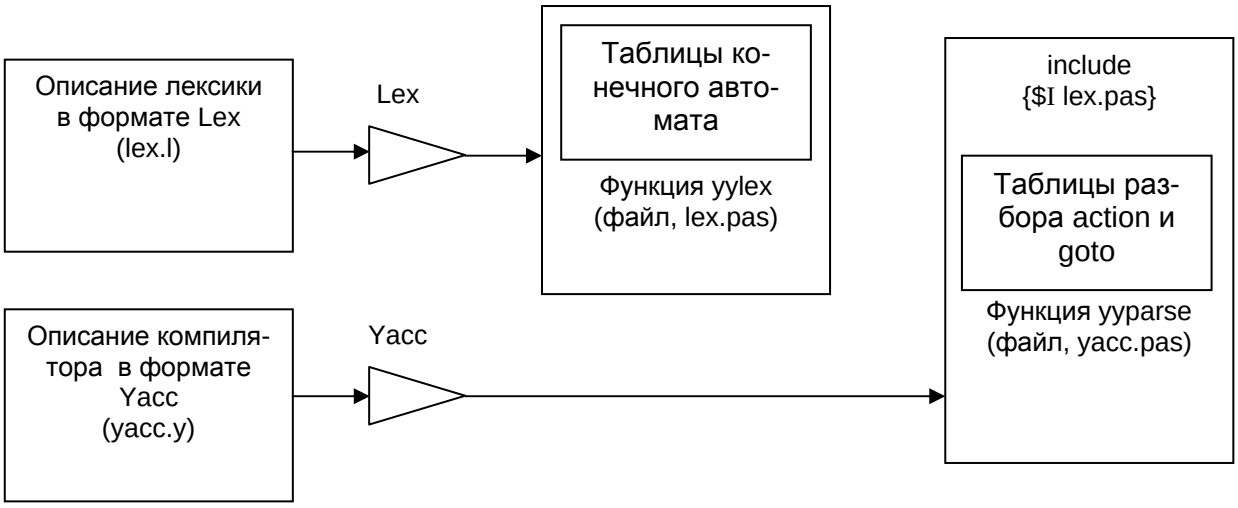


Рисунок 28

9.3 Структура Yacc-программы

В результирующем коде будет объявлена переменная yyLval. Эта переменная всегда автоматически определяется YACC в тексте синтаксического анализатора. Она предназначена для хранения атрибутов терминалов, которые должны быть переданы из лексического анализатора в синтаксический. Тип этой переменной YYSType. Этот тип также автоматически определяется YACC в синтаксическом анализаторе. Он представляет собой запись с переменными полями. Количество и имена полей записи определяются количеством и типами объявленных терминалов. Например, если объявлены терминалы

стандартного типа real и пользовательского типа mytype, то у типа YYSType будет два поля YYReal и YYMytype.

Часть определений включает следующие разделы.

1) Задание стартового символа.

%start символ

Если такое определение отсутствует, то в качестве стартового символа используется нетерминал левой части первой продукции.

2) Определение атрибутов терминалов грамматики.

%token <тип атрибута> терминал

При наличии такого определения в исходном тексте, в модуле, содержащем транслятор, декларируется константа указанного типа с именем, соответствующим терминалу. Эти константы используются в исходном файле Lex без дополнительных объявлений. В качестве типа может быть использован любой стандартный тип Pascal или декларированный ранее.

3) Задание ассоциативности операций.

%left символ – левоассоциативная операция

%right символ – правоассоциативная операция

%noassoc символ – неассоциативная операция

Приоритет операции определяется ее положением при определении. Чем ниже задана операция при определении, тем выше ее приоритет.

4) Задание типов атрибутов нетерминалов

%type <тип атрибута> нетерминал.

Вторая часть Yacc-программы содержит продукции и связанные с ними семантические правила. Левая часть продукции отделяется от правой символом “:”.

Семантические правила записываются на языке Pascal и следуют за продукцией, заключенные между символами “{” и “}”. Для обращения в семантических правилах к атрибуту нетерминала левой части продукции используют символы $$. Для обращения к атрибутам грамматических символов правой части продукции используются символы $i, где i – номер грамматического символа в продукции.

Если для одного и того же нетерминала левой части существует несколько продукций, то это записывается следующим образом.

```
нетерминал : продукция 1  {семантическое правило 1}
           | продукция 2  {семантическое правило 2}
           …
           | продукция n  {семантическое правило n}
```

Третья часть Yacc-программы содержит вспомогательные процедуры. Эта часть переносится в результирующий код без изменения.

9.4 Примеры трансляторов, построенных в Yacc

Пример 1. Распознаем грамматику, задающую математические выражения. Математические выражения состоят из цифр, знаков сложения, умножения, деления и вычитания, а также из выражений, заключенных в скобки.

Решение:

1) Лексический анализатор (lex.l).

Определяем класс “цифра”

```
dig [0-9]
%%
```

Если распознана лексема “Цифра”, то соответствующему полю переменной, хранящей значение атрибута, присваивается значение распознанной лексемы. Кроме этого, возвращается значение константы num.

```
{dig}+      begin
                yylval.yyreal:=strtofloat(yytext);
            returni(num);
          end;
```

Если распознана лексема “символ +”, то возвращаем код этого символа, используя функцию returnc(<символ>). Для остальных символов производим аналогичные действия.

```
\+          returnc('+');
\-          returnc('-');
\*          returnc('*');
\/          returnc('/');
\(          returnc('(');
\)          returnc(')');
```

Заметим, что приведенный текст не имеет фрагментов, необходимых для формирования модуля. Это связано с тем, что файл lex.pas будет включен в другой модуль.

2) Синтаксический анализатор (yacc.y).

Оформляем заголовок модуля

```
%{ unit yacc;
interface
uses lexlib,yacclib, dialogs;
%}
```

Определяем ассоциативность и приоритет операций

```
%left '+'  '-'
%left '*'  '/'
```

Определяем тип атрибутов используемых терминалов.

```
%token <real> num
```

Определяем тип атрибутов нетерминалов. Нетерминал goal, который принадлежит грамматике, отсутствует, так как его атрибуты не используются в семантических правилах.

```
%type <real> expr
%%
```

Часть определений на этом завершена. Теперь опишем продукции и семантические правила. После редукции по правилу 1 пользователь должен получить сообщение с вычисленным значением выражения.

```
goal : expr  {showmessage(floattostr($1));}
```

При редукции по нижеприведенным правилам происходит вычисление фрагмента выражения. Фактически '+' означает, что ожидается возврат лексическим анализатором кода символа '+'.

```
expr : expr '+' expr    {$$:=$1+$3;}
     | expr '-' expr    {$$:=$1-$3;}
```

```
    | expr '*' expr    {$$:=$1*$3;}
    | expr '/' expr    {$$:=$1/$3;}
    | '(' expr ')'     {$$:=$2;}
    | num
%%
```

Включаем в модуль yacc.pas файл lex.pas. При этом файлы должны находиться в одном каталоге.

```
(*$I lex.pas*)
```

В третьей части yacc-программы завершаем формирование модуля.

```
end.
```

После выполнения “yacc.exe yacc.y” YACC построит файл yacc.pas, содержащий функцию yyparse, представляющую собой синтаксический анализатор. Рассмотрим фрагменты кода анализатора.

В исходном тесте был задан один терминал num, поэтому в анализаторе задается соответствующая константа и тип, после чего описывается переменная для сохранения значения атрибута терминала.

```
const num = 257;
type YYSType = record case Integer of
                1 : ( yyreal : real );
              end;
var yylval : YYSType;
```

3) Текст основной программы.
Выполняем установки для работы лексического анализатора.

```
 yymemoinit(memo1,memo2,memo2, memo2);
 yyclear;
 yylineno:=0;
```

Запускаем синтаксический анализатор.

```
 yyparse;
```

Пример 2. Синтаксический анализатор должен распознавать идентификаторы. Результат его работы заключается в выдаче пользователю списка идентификаторов исходной строки без дубликатов и количество повторений идентификатора в исходной строке. Например, если исходная строка имеет вид “first second first fifth”, то результат имеет вид “first – 2 second – 1 fifth - 1”. Так как при работе со строковыми типами в YACC необходимо четко задавать их длину, введем предположение, что длина имени идентификатора не более 5. Список идентификаторов будем хранить в массиве, поэтому предположим, что идентификаторов в строке не более 10.

Решение:

1) Лексический анализатор. Предполагается, что в синтаксическом анализаторе будет объявлен тип myString и константа id этого типа.

```
L [A-Za-z]
%%
{L}+          begin
                yylval.yymyString:=yytext;
              returni(id);
            end;
```

2) Синтаксический анализатор

```
%{ unit yacc;
interface
uses lexlib,yacclib, dialogs;
type
```

Определяем строковый тип фиксированной длины

```
    myString=string[5];
```

Определяем тип, соответствующий элементу списка, то есть идентификатору и количеству его повторений во входной последовательности.

```
    el=record
      ident: myString;
      amount: integer;
    end;
```

Определяем тип массива элементов для хранения результата работы синтаксического анализатора.

```
arType=array[1..10] of el;
```

Описываем все переменные, которые будут необходимы в правилах, терминалы и нетерминалы.

```
var
  point_ar:integer;
  i:integer;
  s:string;
  f:boolean;
%}
%token <myString> id
%type <arType> list
%%
```

При редукции по первому правилу пользователю выводится вся информация из массива.

```
goal : list {s:='';
        for i:=1 to point_ar do
        s:=s+'; '+$1[i].ident+' - '+inttostr($1[i].amount);
        showmessage(s);}
```

При редукции по второму правилу производится поиск в массиве идентификатора. При этом рассматривается массив, сформированный при доказательстве первого нетерминала правой части, и идентификатор, соответствующий терминалу правой части. Если идентификатор уже есть в массиве, то увеличиваем на единицу количество повторений. Если идентификатор не найден, то добавляем новый элемент в массив.

При редукции по третьему правилу, имеющему вид list→ε, никаких действий не выполняем (отсутствует семантическое правило).

```
list : list id {i:=1; f:=true;
            while (i<=point_ar) and f do
            begin
```

```
                if $1[i].ident=$2
                then
                begin
                  $1[i].amount:=$1[i].amount+1;
                  f:=false;
                end
                else     i:=i+1;
              end;
              if f
              then
              begin
                inc(point_ar);
                $1[point_ar].ident:=$2;
                $1[point_ar].amount:=1;
              end;
              $$:=$1;}
        |   ;
%%
(*$I lex.pas*)
initialization
point_ar:=0;
end.
```

9.5 Контрольные вопросы

1) Какой метод разбора используется в Yacc?

2) Что такое атрибутные грамматики?

3) Какие виды атрибутов могут быть в атрибутных грамматиках?

4) Что такое семантическое правило?

5) Для чего служит переменная yyLval? От чего зависит ее тип?

9.6 Задание

Реализовать грамматику, описанную в задании к лабораторной работе№ 5 с использование Yacc.

Семантические правила должны транслировать запрос в операции реляционной алгебры. При работе должна осуществляться проверка на существование таблицы, к которой строится запрос, а также наличие запрашиваемых полей в таблице. Будем считать, что имена

таблиц и их поля хранятся в текстовом файле. Поля раздела where запроса также должны проверяться на существование, но проверку типов осуществлять не нужно. Любой идентификатор в разделе where считать полем таблицы.

9.7 Защита лабораторной работы

В ходе защиты лабораторной работы студент демонстрирует работу построенного транслятора и поясняет семантические правила, написанные в ходе построения транслятора.

На основе записей в объекте yyDbgMemo, произведенных в ходе работы транслятора, студент должен проиллюстрировать процесс сдвига и приведения при распознавании входной последовательности.

10 Задания к практическим работам

1. Определение конечных автоматов (КА). Формы задания КА.
2. Деревья вывода.
3. Грамматика для констант языка программирования.
4. Минимизация грамматик.

СПИСОК ЛИТЕРАТУРЫ

1. Опалева Э.А., Самойленко В.П. Языки программирования и методы трансляции : [учеб.пособие]. – СПб. : БХВ-Петербург , 2005. - 476 с., ил.

2. Ахо А.В. Сети Р. Ульман Дж. Компиляторы: принципы, технологии и инструменты. – М.: Издательский дом «Вильямс», 2001.

3. Хопкрофт Дж.Э., Мотвани Р., Ульман Дж. Введение в теорию автоматов, языков и вычислений. – М.: Издательский дом «Вильямс», 2002.

4. Компаниец Р.И. и др. Системное программирование. Основы построения трансляторов. – СПб.: КОРОНАПРИНТ, 2000.–256 с.

5. Хантер Р. Проектирование и конструирование компиляторов. – М.: Финансы и статистика, 1984 г.