Федеральное агентство по образованию

Томский государственный университет систем управления и радиоэлектроники (ТУСУР)

Кафедра автоматизированных систем управления (АСУ)

Калайда В.Т., Романенко В.В.

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Методическое пособие по выполнению практических работ по дисциплине «Теория языков программирования и методы трансляции» для студентов специальности 230105 – «Программное обеспечение вычислительной техники и автоматизированных систем»

Томск
2012

Федеральное агентство по образованию

Томский государственный университет систем управления и радиоэлектроники (ТУСУР)

Кафедра автоматизированных систем управления (АСУ)

УТВЕРЖДАЮ:
зав. каф. АСУ, проф., д.т.н.

_____ Кориков А.М.

«____» _____2012 г.

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Методическое пособие по выполнению практических работ по дисциплине «Теория языков программирования и методы трансляции» для студентов специальности 230105 – «Программное обеспечение вычислительной техники и автоматизированных систем»

Разработчики:
проф. каф. АСУ, д.т.н.

_____ Калайда В.Т.

доц. каф. АСУ, к.т.н.

_____ Романенко В.В.

Томск – 2012

Содержание

Введение

Учебной программной специальности 230105 в рамках изучения дисциплины «Теория вычислительных процессов и структур» («ТВПиС») предусмотрено выполнение четырех обязательных практических работ:

1. Анализ описания переменных.
2. Разбор математического выражения.
3. Разбор описания структуры при помощи LL(1)-грамматики.
4. Разбор описания структуры при помощи LR(1)-грамматики.

К дополнительным относятся другие практические работы, тема которых связана с предметной областью дисциплины. Это различные интерпретаторы, компиляторы, оптимизаторы кода и т.п. Вариант можно получить у преподавателя или предложить самостоятельно.

В данном методическом пособии изложены задания для обязательных практических работ и одной дополнительной («Решение систем регулярных уравнений»).

Предполагается, что учащиеся хорошо владеют хотя бы одним высокоуровневым структурным (или иным) языком программирования и имеют хорошую математическую подготовку.

1. Анализ описания переменных

Создание программы для анализа описания переменных – цель выполнения практической работы №1.

1.1. Входные данные

На вход программы подается текстовый файл (с именем INPUT.TXT), содержащий **только** описания переменных на выбранном языке (Pascal или C/C++). Например, для языка Pascal содержание текстового файла может быть следующим:

```
var a, b, c: real;
d: array [1..6, 6..9] of integer;
s1: string;
s2: string[10];
```

Т.е. описание переменных начинается с ключевого слова «**var**», далее следуют списки имен переменных с указанием типа. В список типов необходимо включить наиболее часто используемые базовые типы. Тип может быть также массивом (в т.ч. многомерным) или строкой. У массива нижняя граница индекса не должна быть меньше верхней. Строка не может быть длиннее 255 символов.

Имя переменной – это последовательность букв и цифр, начинающаяся с буквы. Под буквами понимаются большие и малые буквы латинского алфавита ($a \ldots z$, $A \ldots Z$) и подчеркивание (_). Большинство современных компиляторов не имеют ограничений по длине имен переменных, однако, значащими считаются только первые N символов (число N зависит от конкретного компилятора). Например, если $N = 8$, то переменные `a1234567` и `a12345678` рассматриваются компиляторами как идентичные, хотя обе записи являются синтаксически верными. Значение числа N можно выбрать любое ($N \geq 1$), но обычно это 8, 16, 32 и т.д.

В качестве разделителей, отделяющих друг от друга ключевые слова, имена переменных, знаки пунктуации и т.п. могут выступать:

- пробелы (код ASCII – 32 или $20 в шестнадцатеричном виде);
- переводы строк и возвраты кареток (коды ASCII – 10 ($0A) и 13 ($0D) соответственно);

– табуляции (код ASCII – 9 или $09).

Для указания символа по его коду в языке Pascal используется знак «#». Либо для этих целей можно использовать функцию CHR (см. ее описание в файле справки компилятора языка Pascal). При указании букв A…F в шестнадцатеричных числах можно использовать как большие, так и малые буквы. Так, для проверки символа `ch` на возврат каретки можно написать

```
if ch = #13 …
```

или

```
if ch = #$0D …
```

или же

```
if ch = chr(13) …
```

Для языков C/C++ файл может быть таким:

```
double a[10], b, c;
int x_q[5][5];
```

Т.е. описание состоит из указаний типов данных со следующими за ними списками имен переменных. Языки C/C++ поддерживают различные модификаторы типов – модификаторы размера (**long**/**short**), знака (**signed**/**unsigned**) и прочие (**auto**, **register**, **volatile**, **const**, **static**). Для упрощения задачи будем рассматривать только модификаторы размера. Обратите внимание, что модификатор может использоваться без указания базового типа, в этом случае в качестве базового типа подразумевается тип **int**. Например, запись

```
long
```

эквивалентна записи

```
long int
```

Модификатор **long** может использоваться с типами **int** и **double**, модификатор **short** – только с типом **int**.

Т.к. языки C/C++ не делают различий между символом и целым числом (байтом), то для проверки кода числа можно просто сравнить символ с кодом:

```
if(ch == 13) …
```

Если компилятор выдает в этом случае предупреждение о возможном несоответствии данных, можно использовать явное преобразование типа:

```
if(ch == char(13)) …
```

Либо можно использовать запись `'\OOO'` или `'\xHH'`, где `OOO` – восьмеричная запись кода символа (максимальный

код – 377_8), HH – шестнадцатеричная (максимальный код – FF_{16}). Так, проверку на символ перевода строки можно также записать в таком виде:

```
if(ch == '\x0d') …
```

или

```
if(ch == '\15') …
```

В языках C/C++ существуют также специальные символы для обозначения некоторых непечатных элементов таблицы ASCII:

- '\n' – перевод строки (**n**ext line);
- '\r' – возврат каретки (carriage **r**eturn);
- '\t' – табуляция (**t**abulate);

Пробелы (как в Pascal, так и в C/C++) можно обозначать просто в виде пробела, заключенного в кавычки (' ').

В отличие от языка Pascal, в языках C/C++ размерность массива указывается в виде одной цифры (обязательно положительной), и каждая размерность заключается в квадратные скобки.

1.2. Выходные данные

Ваша программа должна проанализировать имеющиеся в текстовом файле описания переменных и выдать (в текстовый файл OUTPUT.TXT или на экран) результат проверки. Это может быть:

1. Сообщение о том, что описание корректное.
2. Сообщение о синтаксической ошибке (неправильные имена переменных, ошибки при использовании ключевых слов, неверные индексы массивов, отсутствие знаков пунктуации и т.д.). Указывать тип ошибки не обязательно, требуется только указать строку и позицию в строке входного файла, где наблюдается ошибка. Достаточно находить только первую ошибку в описании.
3. Сообщение о дублировании имен переменных. В этом случае на выходе программы необходимо указать имя дублируемой переменной, а также строку и позицию в строке, где встретился дубликат.

2. Разбор математического выражения

Создание программы для разбора математического выражения – цель выполнения практической работы №2.

2.1. Входные данные

На вход программы подается текстовый файл (с именем INPUT.TXT), содержащий единственную строку символов. Данная строка задает присваивание переменной значения арифметического выражения в виде

ПЕРЕМЕННАЯ = ВЫРАЖЕНИЕ.

Выражение может включать:

- Знаки сложения и умножения («+» и «*»);
- Круглые скобки («(» и «)»);
- Константы (например, 5; 3.8; 1e+18, 8.41E–10);
- Имена переменных.

Имя переменной – это последовательность букв и цифр, начинающаяся с буквы. Входное выражение считать правильным.

2.2. Краткая теория

Рассмотрим краткую теорию преобразования математического выражения в псевдокод, а также оптимизации кода. Более подробные данные можно получить в [1] (разделы 2.4-2.8).

В качестве примера возьмем выражение

$$COST = (PRICE+TAX)*0.98.$$

2.2.1. Лексический анализ

Проанализируем выражение:

- *COST*, *PRICE* и *TAX* – лексемы-идентификаторы;
- 0.98 – лексема-константа;
- =, +, * – просто лексемы.

Пусть все константы и идентификаторы можно отображать в лексемы типа <идентификатор> (<ИД>). Тогда выходом лексического анализатора будет последовательность лексем

$$<ИД_1>=(<ИД_2>+<ИД_3>)*<ИД_4>.$$

Вторая часть компоненты лексемы (указатель, т.е. номер лексемы в таблице имен) – показана в виде индексов. Символы «=», «+» и «*» трактуются как лексемы, тип которых представля-

ется ими самими. Они не имеют связанных с ними данных и, следовательно, не имеют указателей.

2.2.2. Работа с таблицей имен

После того, как в результате лексического анализа лексемы распознаны, информация о некоторых из них собирается и записывается в таблицу имен.

Для нашего примера *COST*, *PRICE* и *TAX* – переменные с плавающей точкой. Рассмотрим вариант такой таблицы. В ней перечислены все идентификаторы вместе с относящейся к ним информацией (табл. 2.1).

Табл. 2.1 – Таблица имен

Номер элемента	Идентификатор	Информация
1	*COST*	Переменная с плавающей точкой
2	*PRICE*	Переменная с плавающей точкой
3	*TAX*	Переменная с плавающей точкой
4	0.98	Константа с плавающей точкой

Если позднее во входной цепочке попадается идентификатор, надо справиться в этой таблице, не появлялся ли он ранее. Если да, то лексема, соответствующая новому вхождению этого идентификатора, будет той же, что и у предыдущего вхождения.

2.2.3. Синтаксический анализ

Выходом анализатора служит дерево, которое представляет синтаксическую структуру, присущую исходной программе.

Пример:

$$<ИД_1>=(<ИД_2>+<ИД_3>)*<ИД_4>.$$

По этой цепочке необходимо выполнить следующие действия:

1) $<ИД_3>$ прибавить к $<ИД_2>$;
2) результат (1) умножить на $<ИД_4>$;
3) результат (2) поместить в ячейку, резервированную для $<ИД_1>$.

Этой последовательности соответствует дерево, изображенное на рис. 2.1.

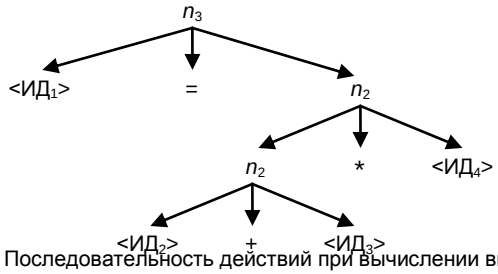


Рис. 2.1 – Последовательность действий при вычислении выражения

Т.е. мы имеем последовательность шагов в виде помеченного дерева.

Внутренние вершины представляют те действия, которые можно выполнять. Прямые потомки каждой вершины либо представляют аргументы, к которым нужно применять действие (если соответствующая вершина помечена идентификатором или является внутренней), либо помогают определить, каким должно быть это действие, в частности, знаки «+», «*» и «=». Скобки отсутствуют, т.к. они только определяют порядок действий.

2.2.4. Генерация кода

Дерево, построенное синтаксическим анализатором, используется для того, чтобы получить перевод входной программы. Рассмотрим машину с одним регистром и команды языка типа «ассемблер» (табл. 2.2).

Табл. 2.2 – Команды языка типа «ассемблер»

Команда	Действие
`LOAD M`	$C(m) \rightarrow$ сумматор
`ADD M`	C(сумматор) + $C(m) \rightarrow$ сумматор
`MPY M`	C(сумматор) * $C(m) \rightarrow$ сумматор
`STORE M`	C(сумматор) $\rightarrow m$
`LOAD =M`	$m \rightarrow$ сумматор
`ADD =M`	C(сумматор) + $m \rightarrow$ сумматор
`MPY =M`	C(сумматор) * $m \rightarrow$ сумматор

Запись «$C(m) \rightarrow$ сумматор» означает, что содержимое ячейки памяти m надо поместить в сумматор. Запись $=m$ означает численное значение m.

С помощью дерева, полученного синтаксическим анализатором и информации, хранящейся в таблице имен, можно построить объектный код.

С каждой вершиной n связывается цепочка $C(n)$ промежуточного кода. Код для вершины n строится сцеплением в фиксированном порядке кодовых цепочек, связанных с прямыми потомками вершины n, и некоторых фиксированных цепочек. Процесс перевода идет, таким образом, снизу вверх (от листьев к корню). Фиксированные цепочки и фиксированный порядок задаются используемым алгоритмом перевода.

Здесь возникает важная проблема: для каждой вершины n необходимо выбрать код $C(n)$ так, чтобы код, приписываемый корню, оказывался искомым кодом всего оператора. Вообще говоря, нужна какая-то интерпретация кода $C(n)$, которой можно было бы единообразно пользоваться во всех ситуациях, где встретится вершина n.

Вернемся к исходному дереву (рис. 2.1). Есть три типа внутренних вершин, зависящих от того, каким из знаков помечен средний потомок: «=», «+» или «*» (рис. 2.2). Здесь треугольники – произвольные поддеревья (в том числе, состоящие из единственной вершины).

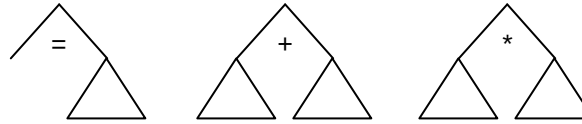


Рис. 2.2 – Типы вершин

Для любого арифметического оператора присвоения, включающего только арифметические операции «+» и «*», можно построить дерево с одной вершиной типа «а» и остальными вершинами только типов «б» и «в».

Код соответствующей вершины будет иметь следующую интерпретацию:

1) если n – вершина типа «а», то $C(n)$ будет кодом, который вычисляет значение выражения, соответствующее правому поддереву, и помещает его в ячейку, зарезервированную для идентификатора, которым помечен левый поток;

2) если n – вершина типа «б» или «в», то цепочка LOAD $C(n)$ будет кодом, засылающим в сумматор значение выражения, соответствующего поддереву, над которым доминирует вершина n.

Так, для нашего дерева код LOAD $C(n_1)$ засылает в сумматор значение выражения <ИД$_2$>+<ИД$_3$>, код LOAD $C(n_2)$ засылает в сумматор значение выражения (<ИД$_2$>+<ИД$_3$>)*<ИД$_4$>, а код $C(n_3)$ засылает в сумматор значение последнего выражения и помещает его в ячейку, предназначенную для <ИД$_1$>.

Теперь надо показать, как код $C(n)$ строится из кодов потомков вершины n. В дальнейшем мы будем предполагать, что операторы языка ассемблера записываются в виде одной цепочки и отделяются друг от друга точкой с запятой или началом новой строки. Кроме того, мы будем предполагать, что каждой вершине n дерева приписывается число $l(n)$, называемое уровнем, которое означает максимальную длину пути от этой вершины до листа, т.е. $l(n) = 0$, если n – лист, а если n имеет потомков $n_1, n_2, \ldots, n_k$, то

$$l(n) = \max_{1 \le i \le k} l(n_i) + 1.$$

Уровни $l(n)$ можно вычислить снизу вверх одновременно с вычислением кодов $C(n)$ (рис. 2.3).

Уровни записываются для того, чтобы контролировать использование временных ячеек памяти. Две нужные нам величины нельзя заслать в одну и ту же ячейку памяти.

Теперь определим синтаксически управляемый алгоритм генерации кода, предназначенный для вычисления кода $C(n)$ всех вершин дерева, состоящих из листьев корня типа «а» и внутренних вершин типа «б» и «в».

Алгоритм.

Вход. Помеченное упорядоченное дерево, представляющее собой оператор присвоения, включающий только арифметические операции «*» и «+». Предполагается, что уровни всех вершин уже вычислены.

Выход. Код в ячейке ассемблера, вычисляющий этот оператор присвоения.

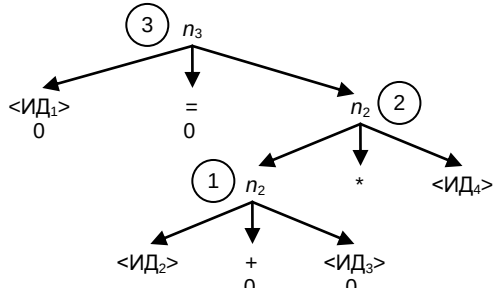


Рис. 2.3 – Дерево с уровнями

Метод. Делать шаги 1) и 2) для всех вершин уровня 0, затем для вершин уровня 1 и т.д., пока не будут отработаны все вершины.

1) Пусть n – лист с меткой <ИД$_i$>.
   1.1. Допустим, что элемент i таблицы идентификаторов является переменной. Тогда $C(n)$ – имя этой переменной.
   1.2. Допустим, что элемент j таблицы идентификаторов является константой k, тогда $C(n)$ – цепочка $=k$.
2) Если n – лист с меткой «=», «+» или «*», то $C(n)$ – пустая цепочка.
3) Если n – вершина типа «а» и ее прямые потомки – это вершины n_1, n_2 и n_3, то $C(n)$ – цепочка LOAD $C(n_3)$; STORE $C(n_1)$.
4) Если n – вершина типа «б» и ее прямые потомки – это вершины n_1, n_2 и n_3, то $C(n)$ – цепочка $C(n_3)$; STORE \$$l(n)$; LOAD $C(n_1)$; ADD \$$l(n)$. Эта последовательность занимает временную ячейку, именем которой служит знак «\$» вместе со следующим за ним уровнем вершины n. Непосредственно видно, что, если перед этой последовательностью поставить LOAD, то значение, которое она поместит в сумматор, будет суммой значений выражением поддеревьев, над которыми доминируют вершины n_1 и n_3. Выбор имен временных ячеек гарантирует, что два нужных значения одновременно не появятся в одной ячейке.

5) Если n – вершина типа «в», а все остальное – как и в 4), то $C(n)$ – цепочка $C(n_3)$; STORE $\$l(n)$; LOAD $C(n_1)$; MPY $\$l(n)$.

Применим этот алгоритм к нашему примеру (рис. 2.4).

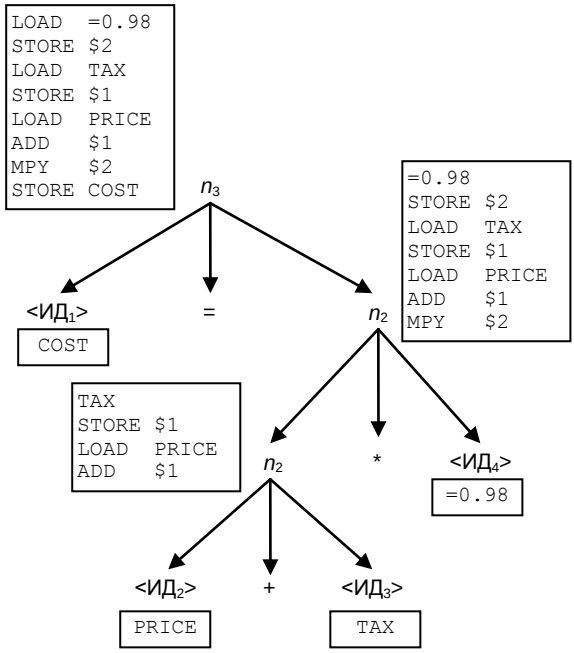


Рис. 2.4 – Дерево с генерированными кодами

Таким образом, в корне мы получили программу на языке типа «ассемблер», эквивалентную исходному выражению.

Естественно, эта программа далека от оптимальной, но это можно исправить на этапе оптимизации.

2.2.5. Оптимизация кода

Рассмотрим приемы, которые делают код более коротким:

1) Если «+» – коммутативная операция, то можно заменить последовательность команд LOAD α; ADD β последовательностью LOAD β; ADD α. Требуется, однако, чтобы в других местах не было перехода к оператору ADD β.

2) Подобным же образом, если «*» – коммутативная операция, то можно заменить LOAD α; MPY β на LOAD β; MPY α.
3) Последовательность операторов типа STORE α; LOAD α можно удалить из программы при условии, что или ячейка α не будет использоваться далее, или перед использованием ячейка α будет заполнена заново.
4) Последовательность LOAD α; STORE β можно удалить, если за ней следует другой оператор LOAD γ, и нет перехода к оператору STORE β, а последующие вхождения β будут заменены на α вплоть до того места, где появится другой оператор STORE β.

Получим оптимизированную программу для нашего примера (табл. 2.3).

Табл. 2.3 – Оптимизация кода

Этап 1	Этап 2	Этап 3

Применяем правило 1 к последовательности
`LOAD PRICE`
`ADD $1`
Заменяем ее последовательностью
`LOAD $1`
`ADD PRICE`
	Применяем правило 3 и удаляем последовательность
	`STORE $1`
	`LOAD $1`
		К последовательности
		`LOAD =0.98`
		`STORE $2`
		применяем правило 4 и удаляем ее. В команде
		`MPY $2`
		заменяем $2 на =0.98

`LOAD =0.98`
`STORE $2`
`LOAD TAX`
`STORE $1`
`LOAD $1`
`ADD PRICE`
`MPY $2`
`STORE COST`
	`LOAD =0.98`
	`STORE $2`
	`LOAD TAX`
	`ADD PRICE`
	`MPY $2`
	`STORE COST`
		`LOAD TAX`
		`ADD PRICE`
		`MPY =0.98`
		`STORE COST`

2.3. Выходные данные

В выходном файле (с именем OUTPUT.TXT) для исходного выражения, заданного во входном файле, необходимо привести:

1) Таблицу имен;
2) Неоптимизированный код;
3) Оптимизированный код.

3. Разбор описания структуры при помощи LL(1)-грамматики

Создание программы для проверки правильности описания структуры при помощи LL(1)-грамматики – цель выполнения практической работы №3.

3.1. Входные данные

На вход программы подаются два текстовых файла (с именами GRAMMAR.TXT и STRUCT.TXT). Первый содержит LL(1)-грамматику или таблицу разбора, второй – описание структуры (записи) на языках C/C++. Необходимо проверить, является ли описание структуры корректным с точки зрения заданной грамматики и не содержит ли конфликтов имен.

Таким образом, задание разбивается на две части:

1. Проверка синтаксиса.
2. Проверка семантики.

Ясно, что семантика (т.е. смысл описания) зависит от выбранного языка, и поэтому ее проверка жестко привязана к анализатору (в данном случае – Вашей программе). Грамматика же должна быть универсальной, т.е. должна позволять задавать любые правила для разбора структуры (и не только структуры). Например, должны быть доступны: смена ключевых слов, изменение знаков пунктуации, правила разбора идентификаторов и т.п.

Желательно задавать грамматику, а не таблицу разбора, т.к. при необходимости грамматику модифицировать проще. Если требуется таблица разбора, ее можно получить из грамматики по описанному ниже алгоритму.

Описание структуры на языках C/C++, в общем случае, выгляди так:

```
struct [имя]
{
    [тип] [список переменных];
    …
    struct [имя]
    {
        …
    } [список переменных];
```

```
    …
} [список переменных];
```

Имя структуры – это имя нового типа, который она описывает. Как и список переменных типа структуры, это необязательный параметр. Если отсутствует и имя типа структуры, и список переменных, структура является анонимной (т.к. к ее полям невозможно получить доступ). Некоторые компиляторы разрешают такие объявления, некоторые – нет. Здесь и в дальнейшем при определении корректности структуры необходимо пользоваться правилами того компилятора, для которого разрабатывается программа.

Частным случаем конфликта имен является дублирование идентификаторов. Но, учитывая, что структуры могут быть вложенными, а также иметь имя типа, задача, по сравнению с первой практической работой, усложняется. Например, структура

```
struct A
{
    int A;
};
```

является корректной с точки зрения языка С, но не с точки зрения языка С++. Язык С++ является объектно-ориентированным, и в структуре с именем типа A ожидает появление конструктора с именем A. Если A – не конструктор, налицо конфликт имен. По тем же причинам некорректна запись

```
struct A
{
    struct A
    {
        …
    };
};
```

Но в большинстве компиляторов языков С/С++ корректной является запись

```
struct A
{
    …
} A;
```

Типы данных при описании переменных использовать стандартные, допускается наличие модификаторов размера

(**long**/**short**) и знака (**signed**/**unsigned**) там, где это возможно.

В заключение еще раз отметим, что правила для поиска конфликтов имен должны соответствовать таковым в компиляторе, для которого предназначена программа.

3.2. Краткая теория

С более полной теорией можно ознакомиться в [1] (глава 4).

Определение. s-грамматика представляет собой грамматику, в которой:

1) правые части каждого порождающего правила начинаются с *терминала*;
2) в тех случаях, когда в левой части более чем одного порождающего правила появляется *нетерминал*, соответствующие правые части начинаются с различных терминалов.

Пример:

$$S \to pX$$
$$S \to qY$$
$$X \to aXb$$
$$X \to x$$
$$Y \to aYd$$
$$Y \to y$$

3.2.1. LL(1)-грамматики

Если возможно написать детерминированный анализатор, осуществляющий разбор сверху вниз, для языка, генерируемого s-грамматикой, то такой анализатор принято называть LL(1)-грамматикой.

Определение. Обозначения в написании LL(1)-грамматики означают:

- L – строки разбираются слева направо;
- L – используются самые левые выводы;
- 1 – варианты порождающего правила выбираются с помощью одного предварительного просмотра символа.

Т.е. грамматику называют LL(1)-грамматикой, если для каждого нетерминала, появляющегося в левой части более од-

ного порождающего правила, множество направляющих символов, соответствующих правым частям альтернативных порождающих правил, – непересекающиеся.

LL(1)-грамматика очень удобна для организации процесса семантического разбора.

3.2.2. LL(1)-ТАБЛИЦА РАЗБОРА

Найдя LL(1)-грамматику для языка, можно перейти к следующему этапу – применению найденной грамматики в фазе разбора. Обычно модуль компилятора, занимающийся семантическим разбором, называется *драйвером*. Драйвер указывает на то место в синтаксисе, которое соответствует текущему входному символу. Составной частью драйвера является стек, который служит для запоминания адресов возврата всякий раз, когда он входит в новое порождающее правило, соответствующее какому-нибудь нетерминалу.

Опишем сначала возможный вид таблицы разбора, а затем рассмотрим возможные способы ее оптимизации относительно используемых вычислительных ресурсов.

Таблица разбора, в общем виде, представляет собой одномерный массив структур следующего вида:

```
declare 1 TABLE,
          2 terminals LIST,
          2 jump int,
          2 accept bool,
          2 stack bool,
          2 return bool,
          2 error bool;
```

где LIST

```
declare 1 LIST,
          2 term string,
          2 next pointer;
```

Кроме того, для работы драйвера нужен стек адресов возврата и указатель стека.

В таблице каждому шагу процесса разбора соответствует один элемент. В процессе разбора осуществляются следующие шаги.

1) Проверка предварительно просматриваемого символа, для того, чтобы выяснить, не является ли он направляющим для какой-либо конкретной правой части порож-

дающего правила. Если этот символ – не направляющий и имеется альтернативная правая часть правила, то она проверяется на следующем этапе. В особом случае, когда правая часть начинается с терминала, множество направляющих символов состоит только из одного терминала.

2) Проверка терминала, появляющегося в правой части порождающего правила.
3) Проверка нетерминала. Она заключается в проверке нахождения предварительно просматриваемого символа в одном из множеств направляющих символов для данного нетерминала, помещению в стек адреса возврата и переходу к первому правилу, относящемуся к этому нетерминалу. Если нетерминал появился в конце правой части правила, то нет необходимости помещать в стек адрес его возврата.

Таким образом, в таблицу разбора включается по одному элементу на каждое правило грамматики и на каждый экземпляр терминала или нетерминала правой части правил. Кроме того, в таблице будут находиться элементы на реализацию пустой строки в правой части правил (по одному на каждую реализацию).

Драйвер содержит процедуру, которая обрабатывает элементы таблицы разбора и определяет следующий элемент для обработки. *Поле перехода* обычно дает следующий элемент обработки, если значение *поля возврата* не окажется истиной. В последнем случае адрес следующего элемента берется из стека, что соответствует концу правила. Если же предварительно просматриваемый символ отсутствует в списке терминалов и значение *поля ошибки* окажется ложью, нужно обрабатывать следующий элемент таблицы с тем же предварительно просматриваемым символом (способ обращения с альтернативными правыми частями).

Рассмотрим схему построения таблицы разбора и соответствующей программы для следующей грамматики:

(1) PROGRAM $\rightarrow$ *begin* DECLIST *semi* STATLIST *end*
(2) DECLIST $\rightarrow$ *d* X
(3) X $\rightarrow$ *comma* DECLIST
(4) X $\rightarrow$ *e*
(5) STATLIST $\rightarrow$ *s* Y

(6) Y → *comma* STATLIST

(7) Y → *e*

Сначала представим грамматику в виде схемы (рис. 3.1). В скобках и справа на рисунке указаны номера элементов таблицы разбора.

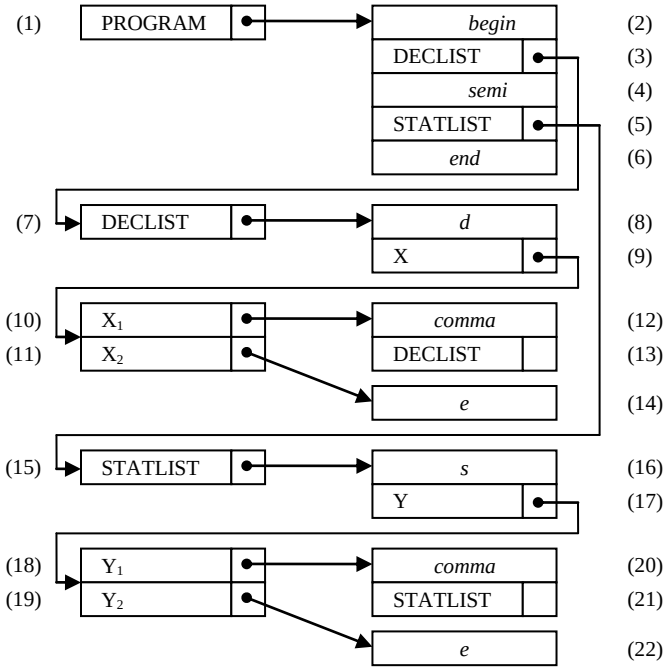


Рис. 3.1 – Схема грамматики

Таблица разбора, соответствующая этой грамматике, может быть представлена в виде табл. 3.1.

Табл. 3.1 – Таблица разбора

№	terminals	jump	accept	stack	return	error
1	{*begin*}	2	false	false	false	true
2	{*begin*}	3	true	false	false	true
3	{*d*}	7	false	true	false	true
4	{*semi*}	5	true	false	false	true
5	{*s*}	15	false	true	false	true
6	{*end*}	0	true	false	true	true

7	{*d*}	8	false	false	false	true
8	{*d*}	9	true	false	false	true
9	{*comma*, *semi*}	10	false	false	false	true
10	{*comma*}	12	false	false	false	false
11	{*semi*}	14	false	false	false	true
12	{*comma*}	13	true	false	false	true
13	{*d*}	7	false	false	false	true
14	{*semi*}	0	false	false	true	true
15	{*s*}	16	false	false	false	true
16	{*s*}	17	true	true	false	true
17	{*comma*, *end*}	18	false	false	false	true
18	{*comma*}	20	false	false	false	false
19	{*end*}	22	false	false	false	true
20	{*comma*}	21	true	false	false	true
21	{*s*}	15	false	false	false	true
22	{*end*}	0	false	false	true	true

Рассмотрим разбор предложения

begin d comma d semi s semi end

Действия приведены в табл. 3.2.

Табл. 3.2 – Разбор предложения

№	Действия	Стек разбора
1	*begin* считывается и проверяется; перейти к 2	0
2	*begin* считывается и принимается; перейти к 3	0
3	*d* считывается и проверяется; 4 помещается в стек; перейти к 7	4 0
7	*d* принимается; перейти к 8	4 0
8	*d* проверяется и принимается; перейти к 9	4 0
9	*comma* считывается и проверяется; перейти к 10	4 0
10	*comma* проверяется; перейти к 12	4 0
12	*comma* проверяется и принимается; перейти к 13	4 0
13	*d* считывается и проверяется; перейти к 7	4

		0
7	*d* проверяется; перейти к 8	4 0
8	*d* проверяется и принимается; перейти к 9	4 0
9	*comma* считывается и проверяется; перейти к 10	4 0
10	*semi* не совпадает с *comma*; ошибка – «ложь», перейти к 11	4 0
11	*comma* проверяется; перейти к 14	4 0
14	*semi* проверяется; возврат – «истина», удаляется 4; перейти к 4	0
4	*semi* проверяется и принимается; перейти к 5	0
5	*s* считывается и проверяется; 6 помещается в стек; перейти к 15	6 0
15	*s* проверяется; перейти к 16	6 0
16	*s* проверяется и принимается; перейти к 17	6 0
17	*comma* считывается и проверяется; перейти к 18	6 0
18	*comma* проверяется; перейти к 20	6 0
20	*comma* проверяется и принимается; перейти к 21	6 0
21	*s* считывается и проверяется; перейти к 15	6 0
15	*s* проверяется; перейти к 16	6 0
16	*s* проверяется и принимается; перейти к 17	6 0
17	*end* считывается и проверяется; перейти к 18	6 0
18	*end* не совпадает с *comma*; ошибка – «ложь», перейти к 19	6 0
19	*end* проверяется; перейти к 22	6 0
22	*end* проверяется; возврат – «истина», удалить	0

	6; перейти к 6	
6	*end* проверяется и принимается; возврат – «истина», удалить 0; перейти к 0	
0	Разбор заканчивается	

3.3. Выходные данные

Ваша программа должна проанализировать имеющееся в текстовом файле описание структуры и выдать (в текстовый файл OUTPUT.TXT или на экран) результат проверки. Это может быть:

1. Сообщение о том, что описание структуры корректное.
2. Сообщение о синтаксической ошибке. Указывать тип ошибки не обязательно, требуется только указать строку и позицию в строке входного файла, где наблюдается ошибка. Достаточно находить только первую ошибку в описании.
3. Сообщение о конфликте имен. В этом случае на выходе программы необходимо указать имя конфликтного типа, а также строку и позицию в строке, где произошел конфликт.

4. Разбор описания структуры при помощи LR(1)-грамматики

Создание программы для проверки правильности описания структуры при помощи LR(1)-грамматики – цель выполнения практической работы №4.

4.1. Входные данные

Соответствуют таковым в практической работе №3, но разбор осуществляется с помощью LR(1)-, а не LL(1)-грамматики. Соответственно, входной файл GRAMMAR.TXT содержит LR(1)-грамматику или таблицу разбора (грамматика, как и прежде, предпочтительнее).

4.2. Краткая теория

С более полной теорией можно ознакомиться в [1] (глава 5).

4.2.1. LR(1)-таблица разбора

Таблица разбора представляет собой матрицу. Она состоит из столбцов для каждого терминала и нетерминала грамматики плюс знака окончания и строк, соответствующих каждому состоянию, в котором может находиться анализатор. Каждое состояние соответствует той позиции в порождающем правиле, которой достиг анализатор. Таблица разбора для грамматики

(1) S → **real** IDLIST
(2) IDLIST → IDLIST, ID
(3) IDLIST → ID
(4) ID → *A* | *B* | *C* | *D*

приведена в табл. 4.1. Драйвер анализатора использует еще и два стека – *стек символов* и *стек состояний*. Таблица разбора включает элементы четырех типов:

1) *Элементы сдвига*. Эти элементы имеют вид «*S*7» и означают: поместить в стек символ, соответствующий столбцу; поместить в стек состояний 7 и перейти к состоянию 7; если входной символ – терминал, принять его.

2) *Элемент приведения*. Он имеет вид «$R4$» и означает: выполнить приведение с помощью правила (4), т.е., допустив, что n есть число символов в правой части правила (4), удалить n элементов из стека символов и n элементов из стека состояний и перейти к состоянию, указанному в верхней части стека состояний. Нетерминал в левой части правила (4) нужно считать следующим входным символом.
3) *Элемент ошибок*. Эти элементы являются пробелами в таблице и соответствуют синтаксическим ошибкам.
4) *Элемент остановки*. Им завершается разбор.

Табл. 4.1 – Таблица разбора

Состояние	S	IDLIST	ID	**real**	,	*A B* *C D*	⊥
1	*HALT*			*S*2			
2		*S*5	*S*4			*S*3	
3					*R*4		*R*4
4					*R*3		*R*3
5					*S*6		*R*1
6						*S*3	
7					*R*2		*R*2

В табл. 4.2 рассмотрим, как с помощью вышеописанной таблицы разбирается строка «**real** *A*, *B*, *C*⊥».

Табл. 4.2 – Разбор строки

Позиция	Стек символов	Стек состояний
↑**real** *A*, *B*, *C*⊥		1
входной символ **real** – из элемента таблицы (1, **real**), сдвиг в состояние 2:		
real↑ *A*, *B*, *C*⊥	**real**	2 1
входной символ *A* – сдвиг в состояние 3:		
real *A*↑, *B*, *C*⊥	*A* **real**	3 2 1
входной символ – «,», приведение по правилу (4):		

real $A_{\uparrow}$, B, $C\perp$	**real**	2 1

входной символ – ID, сдвиг в состояние 4:

real $A_{\uparrow}$, B, $C\perp$	ID **real**	4 2 1

входной символ – «,», приведение по правилу (3):

real $A_{\uparrow}$, B, $C\perp$	**real**	2 1

входной символ – IDLIST, сдвиг в состояние 5:

real $A_{\uparrow}$, B, $C\perp$	IDLIST **real**	5 2 1

входной символ – «,», сдвиг в состояние 6:

real A,$_{\uparrow}$ B, $C\perp$	, IDLIST **real**	6 5 2 1

входной символ – B, сдвиг в состояние 3:

real A, $B_{\uparrow}$, $C\perp$	B , IDLIST **real**	3 6 5 2 1

входной символ – «,», приведение по правилу (4):

real A, $B_{\uparrow}$, $C\perp$	, IDLIST **real**	6 5 2 1

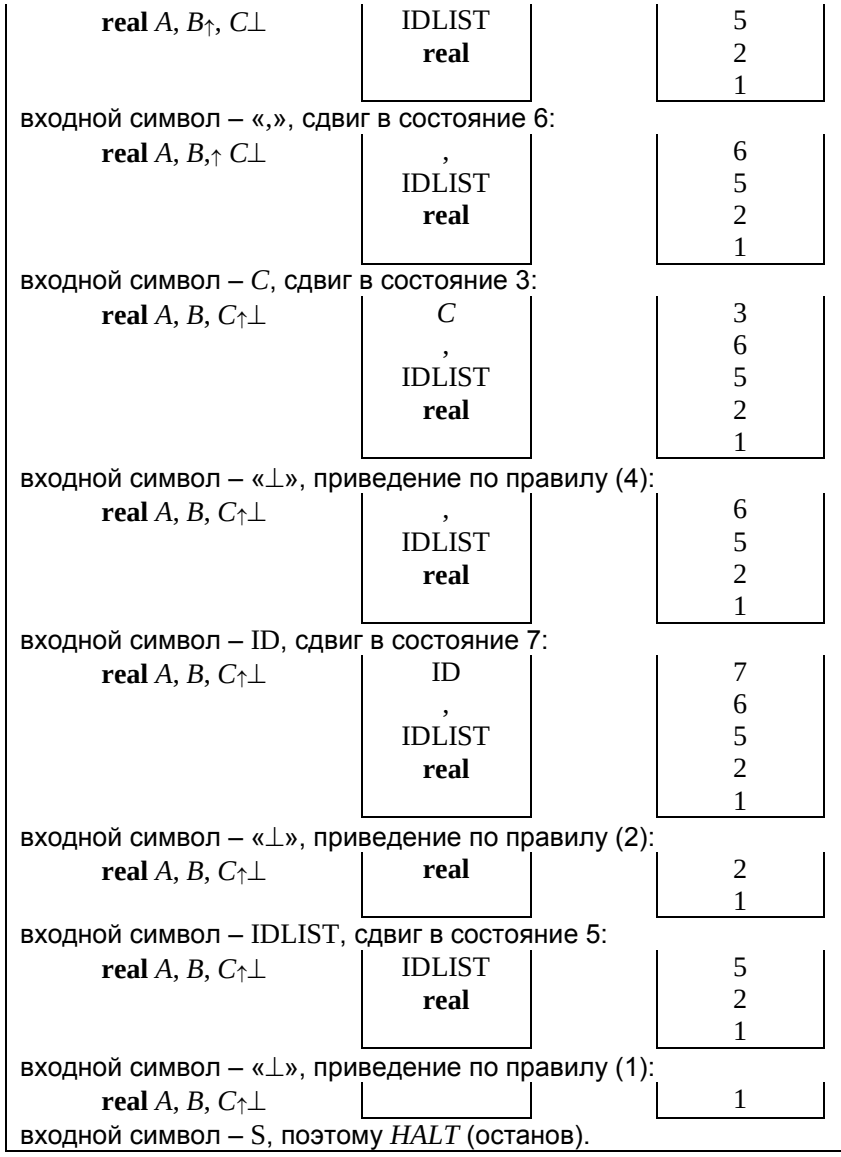
входной символ – ID, сдвиг в состояние 7:

real A, $B_{\uparrow}$, $C\perp$	ID , IDLIST **real**	7 6 5 2 1

входной символ – «,», приведение по правилу (2):

real A, $B_{\uparrow}$, $C\perp$	**real**	2 1

входной символ – IDLIST, сдвиг в состояние 5:



Разбор завершен успешно.

Заметим, что после сдвига входным символом является следующий символ, а после приведения – символ, к которому только что привело действие.

4.2.2. Построение LR-таблицы разбора

При построении LR-таблиц разбора нам необходимо ссылаться на конкретную позицию в правиле, поэтому в правилах вводится понятие «*конфигурация*». Например, в грамматике

(1) $S \rightarrow \bullet$ **real** IDLIST
(2) $IDLIST \rightarrow IDLIST, ID$
(3) $IDLIST \rightarrow ID$
(4) $ID \rightarrow A \mid B \mid C \mid D$

точка (•) соответствует конфигурации (1, 0), т.е. правило (1), позиция 0; конфигурация (1, 1) соответствует точке, появляющейся сразу после **real** в правиле (1), а (2, 0) – точке, появляющейся перед IDLIST в правой части правила (2). Так, конфигурация (2, 2) показывает, что правая часть правила (2) распознана по запятую включительно.

Состояние в таблице разбора примерно соответствует конфигурациям в грамматике с той разницей, что конфигурации, которые неразличимы для анализатора, представляются одним и тем же состоянием. Например, если (1, 0) соответствует состоянию 1, а (1, 1) – состоянию 2, то в вышеприведенной грамматике (2, 0), (3, 0) и (4, 0) будут также соответствовать состоянию 2. В этом случае говорят, что множество конфигураций

{(1, 1), (2, 0), (3, 0), (4, 0)}

образуют замыкание (1, 1).

Из заданного состояния, не соответствующего *концу правила*, можно перейти в другое состояние, введя терминальный или нетерминальный символ. Это состояние называется *преемником* первоначального состояния. Чтобы построить таблицу разбора, необходимо прежде найти все состояния в грамматике. Поэтому, начиная с конфигурации (1, 0), последовательно выполним операции замыкания и образования приемника до тех пор, пока все конфигурации не окажутся включенными в какое-либо состояние. Там, где ряд конфигураций содержится в одном замыкании, каждая из них будет соответствовать одному и тому же состоянию. Новая конфигурация, которая получается при операции образования преемника, называется *базовой*. Если за базовой конфигурацией следует нетерминал, то все конфигура-

ции, соответствующие помещению точки слева от каждой правой части для данного нетерминала, сконцентрируются в замыкании этой базовой конфигурации. В приведенной грамматике можно выделить семь состояний, которые описываются следующим образом (табл. 4.3).

Табл. 4.3 – Состояния грамматики

Состояние	База	Замыкание
1	(1, 0)	{(1, 0)}
2	(1, 1)	{(1, 1), (2, 0), (3, 0), (4, 0)}
3	(4, 1)	{(4, 1)}
4	(3, 1)	{(3, 1)}
5	{(2, 1), (1, 2)}	{(2, 1), (1, 2)}
6	(2, 2)	{(2, 2), (4, 0)}
7	(2, 3)	{(2, 3)}

Эти состояния расположены в грамматике следующим образом:

(1) $\mathrm{S} \rightarrow_1$ **real** $_2\ \mathrm{IDLIST}_5$

(2) $\mathrm{IDLIST} \rightarrow_2 \mathrm{IDLIST}_5$, $_6\ \mathrm{ID}_7$

(3) $\mathrm{IDLIST} \rightarrow_2 \mathrm{ID}_4$

(4) $\mathrm{ID} \rightarrow_{(2,\,6)} A \mid B \mid C \mid D_3$

Заметим, что конфигурация может соответствовать более чем одному состоянию, и в базе может быть более одной конфигурации, если преемники двух конфигураций в одном и том же замыкании неразличимы. Например, за конфигурациями (1, 1) и (2, 0) следует IDLIST, что делает (1, 2) и (2, 1) неразличимыми, пока не осуществится операция замыкания. Число состояний в анализаторе соответствует числу множеств неразличимых конфигураций в грамматике. Причина того, что два и более состояния соответствуют одной конфигурации, раскрывается в ходе разбора.

Действия анализатора со сдвигом аналогичны операции получения преемника. Поэтому действия со сдвигом в таблицу разбора могут вноситься на основании информации о размещении состояний в грамматике (табл. 4.4).

Табл. 4.4 – Таблица со сдвигами

Состояние	S	IDLIST	ID	**real**	,	$A\ B$ $C\ D$	$\perp$

1				$S2$			
2		$S5$	$S4$			$S3$	
3							
4							
5					$S6$		
6						$S3$	
7							

Например, правило (2) означает «из состояния 2 при чтении IDLIST перейти в состояние 5», «из состояния 5 при чтении запятой перейти в состояние 6» и т.д.

Задача внесения приведения в таблицу разбора нетривиальна. Однако, единственные состояния, в которых приведения возможны, – это состояния, соответствующие окончаниям правил (в нашей грамматике 3, 4, 5, и 7). Поэтому мы можем внести $R4$ во все столбцы состояния 3, $R3$ – во все столбцы состояния 4, $R1$ – во все столбцы состояния 5 и $R2$ – во все столбцы состояния 7. Однако в состоянии 5 в одном столбце уже имеется элемент сдвига. Таким образом, возникает конфликт *сдвиг/приведение*. Состояние 5 называется *неадекватным*. Разрешить эту проблему можно, просматривая символы, которые показали бы приведение в состояние 5, а не сдвиг. Из правил (1) и (2) следует, что такими символами могут быть только «⊥» и «,», а приведение возможно лишь в том случае, если символ окажется «⊥», в то время как анализатор осуществит сдвиг в состояние 6, если следующим символом будет «,». Поэтому вносим $R1$ в пятую строку столбца, соответствующего знаку «⊥». Этим действием неадекватность снимается (табл. 4.5).

Табл. 4.5 – Таблица сдвигов и приведений

Состояние	S	IDLIST	ID	**real**	,	*A B* *C D*	⊥
1				$S2$			
2		$S5$	$S4$			$S3$	
3	$R4$	$R4$	$R4$	$R4$	$R4$	$R4$	$R4$
4	$R3$	$R3$	$R3$	$R3$	$R3$	$R3$	$R3$
5					$S6$		$R1$
6						$S3$	
7	$R2$	$R2$	$R2$	$R2$	$R2$	$R2$	$R2$

Аналогичным образом, рассмотрев предварительные символы, можно исключить из таблицы все лишние приведения. В результате таблица разбора приобретает вид табл. 4.6.

Табл. 4.6 – Окончательная таблица разбора

Состояние	S	IDLIST	ID	**real**	,	*A B C D*	⊥
1	*HALT*			*S*2			
2		*S*5	*S*4			*S*3	
3					*R*4		*R*4
4					*R*3		*R*3
5					*S*6		*R*1
6						*S*3	
7					*R*2		*R*2

4.3. Выходные данные

Аналогично практической работе №3.

5. Решение систем регулярных уравнений

Решение системы регулярных уравнений – цель выполнения дополнительной практической работы №5.

5.1. Входные данные

Во входном файле (с именем INPUT.TXT) задается размерность системы регулярных уравнений n ($1 \le n \le 8$) а затем – ее коэффициенты:

$$\alpha_{10}\ \alpha_{11}\ \alpha_{12}\ \ldots\ \alpha_{1n}$$
$$\alpha_{20}\ \alpha_{21}\ \alpha_{22}\ \ldots\ \alpha_{2n}$$
$$\ldots\ldots\ldots\ldots\ldots$$
$$\alpha_{n0}\ \alpha_{n1}\ \alpha_{n2}\ \ldots\ \alpha_{nn}$$

Максимальная длина регулярного выражения для каждого коэффициента равна 3.

5.2. Краткая теория

Рассмотрим краткую теорию решения уравнений с регулярными коэффициентами. Более подробные данные можно получить в [1] (раздел 3.5).

Определение. Регулярные выражения в алфавите Σ и регулярные множества, которые они обозначают, определяются рекурсивно следующим образом:

1) $\varnothing$ – регулярное выражение, обозначающее регулярное множество $\varnothing$;
2) e – регулярное выражение, обозначающее регулярное множество $\{e\}$;
3) если $a \in \Sigma$, то a – регулярное выражение, обозначающее регулярное множество $\{a\}$;
4) если p и q – регулярные выражения, обозначающие регулярные множества P и Q, то
   а) $(p+q)$ – регулярное выражение, обозначающее $P \cup Q$;
   б) pq – регулярное выражение, обозначающее $P \cap Q$;
   в) p^* – регулярное выражение, обозначающее P^*;
5) ничто другое не является регулярным выражением.

Принято обозначать p^+ для сокращенного обозначения pp^*. Расстановка приоритетов:

- * (итерация) – наивысший приоритет;
- конкатенация;

– + (объединение).

Например, $0 + 10^* = (0 + (1\,(0^*)))$.

Таким образом, для каждого регулярного множества можно найти регулярное выражение, его обозначающее, и наоборот.

Введем леммы, обозначающие основные алгебраические свойства регулярных выражений. Пусть α, β и γ регулярные выражения, тогда:

1) $\alpha + \beta = \beta + \alpha$
2) $\varnothing^* = e$
3) $\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$
4) $\alpha(\beta\gamma) = (\alpha\beta)\gamma$
5) $\alpha(\beta + \gamma) = \alpha\beta + \alpha\gamma$
6) $(\alpha + \beta)\gamma = \alpha\gamma + \beta\gamma$
7) $\alpha e = e\alpha = \alpha$
8) $\alpha\varnothing = \varnothing\alpha = \varnothing$
9) $\alpha^* = \alpha + \alpha^*$
10) $(\alpha^*)^* = \alpha^*$
11) $\alpha + \alpha = \alpha$
12) $\alpha + \varnothing = \alpha$

При работе с языками часто удобно пользоваться уравнениями, коэффициентами и неизвестными которых служат множества. Такие уравнения будем называть *уравнениями с регулярными коэффициентами*

$$X = aX + b,$$

где a и b – регулярные выражения. Можно проверить прямой подстановкой, что решением этого уравнения будет a^*b:

$$aa^*b + b = (aa^* + e)b = a^*b,$$

т.е. получаем одно и то же множество. Таким же образом можно установить и решение системы уравнений.

Определение. Систему уравнений с регулярными коэффициентами назовем *стандартной системой* с множеством неизвестных $\Delta = \{X_1, X_2, \ldots, X_n\}$, если она имеет вид:

$$X_1 = \alpha_{10} + \alpha_{11}X_1 + \alpha_{12}X_2 + \ldots + \alpha_{1n}X_n$$
$$X_2 = \alpha_{20} + \alpha_{21}X_1 + \alpha_{22}X_2 + \ldots + \alpha_{2n}X_n$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$X_n = \alpha_{n0} + \alpha_{n1}X_1 + \alpha_{n2}X_2 + \ldots + \alpha_{nn}X_n$$

где α_{ij} – регулярные выражения в алфавите, не пересекающемся с Δ. Коэффициентами уравнения являются выражения α_{ij}.

Если $\alpha_{ij} = \varnothing$, то в уравнении для X_i нет числа, содержащего X_j. Аналогично, если $\alpha_{ij} = e$, то в уравнении для X_i член, содержащий X_j – это просто X_j. Иными словами, $\varnothing$ играет роль коэффициента 0, а e – роль коэффициента 1 в обычных системах линейных уравнений.

Алгоритм решения.

Вход. Стандартная система Q уравнений с регулярными коэффициентами в алфавите Σ и множеством неизвестных $\Delta = \{X_1, X_2, \ldots, X_n\}$.

Выход. Решение системы Q.

Метод: Аналог метода решения системы линейных уравнений методом исключения Гаусса.

Шаг 1. Положить $i = 1$.

Шаг 2. Если $i = n$, перейти к шагу 4. В противном случае с помощью тождеств леммы записать уравнения для X_i в виде

$$X_i = \alpha X_i + \beta,$$

где α – регулярное выражение в алфавите Σ, а β – регулярное выражение вида

$$\beta_0 + \beta_{i+1}X_{i+1} + \ldots + \beta_n X_n,$$

причем все β_i – регулярные выражения в алфавите Σ. Затем в правых частях для уравнений $X_{i+1}, \ldots, X_n$ заменим X_i регулярным выражением $\alpha^*\beta$.

Шаг 3. Увеличить i на 1 и вернуться к шагу 2.

Шаг 4. Записать уравнение для X_n в виде $X_n = \alpha X_n + \beta$, где α и β – регулярные выражения в алфавите Σ. Перейти к шагу 5 (при этом $i = n$).

Шаг 5. Уравнение для X_i имеет вид $X_i = \alpha X_i + \beta$, где α и β – регулярные выражения в алфавите Σ. Записать на выходе $X_i = \alpha^*\beta$, в уравнениях для $X_{i-1}, \ldots, X_1$ подставляя $\alpha^*\beta$ вместо X_i.

Шаг 6. Если $i = 1$, остановиться, в противном случае уменьшить i на 1 и вернуться к шагу 5.

5.3. Выходные данные

В выходной файл (с именем OUTPUT.TXT) необходимо вывести:

1) Полное решение системы регулярных уравнений;
2) Упрощенное решение.

Упрощенное решение получается, если применить к полученному решению леммы 1-12.

Список литературы

1. Калайда В.Т., Теория языков программирования и методов трансляции. Учебное пособие. – Томск: изд-во ТУСУР, 2008. – 253 с.