Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ

Кафедра автоматизированных систем управления

В.Т. Калайда, В.В. Романенко

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Методические указания по выполнению лабораторных работ

2013

Корректор: Осипова Е.А.

Калайда В.Т., Романенко В.В.
Теория языков программирования и методы трансляции: методические указания по выполнению лабораторных работ. — Томск: Факультет дистанционного обучения, ТУСУР, 2013. — 105 с.

Содержание

Введение

Учебной программной в рамках изучения дисциплины «Теория языков программирования и методы трансляции» («ТЯПМТ») для студентов, обучающихся с применением дистанционных образовательных технологий, предусмотрено выполнение двух лабораторных работ:

1. Синтаксический анализ с использованием конечных автоматов и регулярных выражений.
2. Синтаксический анализ с использованием КС-грамматик.

В данном методическом пособии изложены задания для лабораторных работ, в приложении — пример оформления титульного листа отчета по лабораторным работам.

Работы выполняются по вариантам. Выбор варианта лабораторных работ осуществляется по общим правилам с использованием следующей формулы:

$$V = (N*K) \text{ div } 100,$$

где V — искомый номер варианта,
N — общее количество вариантов,
div — целочисленное деление,
при V= 0 выбирается максимальный вариант,
K — значение 2-х последних цифр пароля.

Предполагается, что учащиеся хорошо владеют хотя бы одним высокоуровневым объектно-ориентированным языком программирования (Object Pascal, C++, C# и т.д.) и имеют хорошую математическую подготовку.

1 Лабораторная работа № 1

Цель выполнения лабораторной работы № 1 — научиться применять на практике такие средства синтаксического анализа, как детерминированные конечные автоматы с магазинной памятью (ДМП-автоматы) и регулярные выражения (РВ).

1.1 Варианты заданий на лабораторную работу № 1

Вариант № 1. На вход программы подается текстовый файл (с именем INPUT.TXT), содержащий **только** описания переменных на выбранном языке (Pascal, C++, C# и т.д.).

Программа должна проанализировать имеющееся в текстовом файле описание переменных при помощи **ДМП-автомата** 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. 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.

Вариант № 2. 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.

Программа должна с помощью **ДМП-автомата** построить дерево, соответствующее заданному во входном файле выражению, и выдать (в текстовый файл OUTPUT.TXT) для данного выражения:

1) таблицу имен;
2) неоптимизированный код;
3) оптимизированный код.

Вариант № 3. На вход программы подается текстовый файл (с именем INPUT.TXT), содержащий **только** описания переменных на выбранном языке (Pascal, C++, C# и т.д.).

Программа должна проанализировать имеющееся в текстовом файле описание переменных при помощи **регулярного выражения** 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. 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.

Вариант № 4. 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.

Программа должна с помощью **регулярного выражения** построить дерево, соответствующее заданному во входном файле выражению, и выдать (в текстовый файл OUTPUT.TXT) для данного выражения:

1) таблицу имен;
2) неоптимизированный код;
3) оптимизированный код.

2 Лабораторная работа № 2

Цель выполнения лабораторной работы № 2 — научиться применять на практике такие средства синтаксического анализа, как контекстно-свободные грамматики (КС-грамматики).

2.1 Варианты заданий на лабораторную работу № 2

Вариант № 1. На вход программы подаются два текстовых файла (с именами GRAMMAR.TXT и INPUT.TXT). Первый содержит LL(1)-грамматику, второй — описание процедур и функций на выбранном языке (Pascal, C++) либо делегатов на языке C#. Необходимо проверить, является ли описание процедур/функций/делегатов корректным с точки зрения заданной грамматики и не содержатся ли в нем конфликты имен.

Таким образом, задание разбивается на две части:

1. Проверка синтаксиса.
2. Проверка семантики.

Семантика 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 процедур/функций/делегатов (и не только). Например, должны быть доступны изменения: ключевых слов, знаков пунктуации, правил разбора идентификаторов, а также добавление новых языковых конструкций и т.п.

Программа должна проанализировать имеющееся в текстовом файле описание процедур/функций/делегатов и выдать (в текстовый файл OUTPUT.TXT) результат проверки. Это может быть:

1. Сообщение о том, что грамматика во входном файле не является LL(1)-грамматикой.
2. Сообщение о том, что описание корректное.
3. 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.

4. Сообщение о конфликте имен. В этом случае на выходе программы необходимо указать конфликтующее имя, а также строку и позицию в строке, где произошел конфликт.

Вариант № 2. На вход программы подаются два текстовых файла (с именами GRAMMAR.TXT и INPUT.TXT). Первый содержит LL(1)-грамматику, второй — описание структуры (записи) на выбранном языке (Pascal, C++, C#). Необходимо проверить, является ли описание структуры корректным с точки зрения заданной грамматики и не содержатся ли в нем конфликты имен.

Таким образом, задание разбивается на две части:

1. Проверка синтаксиса.
2. Проверка семантики.

Семантика 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 изменения: ключевых слов, знаков пунктуации, правил разбора идентификаторов, а также добавление новых языковых конструкций и т.п.

Программа должна проанализировать имеющееся в текстовом файле описание структуры и выдать (в текстовый файл OUTPUT.TXT) результат проверки. Это может быть:

1. Сообщение о том, что грамматика во входном файле не является LL(1)-грамматикой.
2. Сообщение о том, что описание корректное.
3. 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.
4. Сообщение о конфликте имен. В этом случае на выходе программы необходимо указать конфликтующее имя, а также строку и позицию в строке, где произошел конфликт.

Вариант № 3. На вход программы подаются два текстовых файла (с именами GRAMMAR.TXT и INPUT.TXT). Первый содержит LR(1)-грамматику, второй — описание процедур и функций на выбранном языке (Pascal, C++), либо делегатов на языке C#. Необходимо проверить, является ли описание процедур/функций/делегатов корректным с точки зрения заданной грамматики и не содержатся ли в нем конфликты имен.

Таким образом, задание разбивается на две части:

1. Проверка синтаксиса.
2. Проверка семантики.

Семантика 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 процедур/функций/делегатов (и не только). Например, должны быть доступны изменения: ключевых слов, знаков пунктуации, правил разбора идентификаторов, а также добавление новых языковых конструкций и т.п.

Программа должна проанализировать имеющееся в текстовом файле описание процедур/функций/делегатов и выдать (в текстовый файл OUTPUT.TXT) результат проверки. Это может быть:

1. Сообщение о том, что грамматика во входном файле не является LR(1)-грамматикой.
2. Сообщение о том, что описание корректное.
3. 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.
4. Сообщение о конфликте имен. В этом случае на выходе программы необходимо указать конфликтующее имя, а также строку и позицию в строке, где произошел конфликт.

Вариант № 4. На вход программы подаются два текстовых файла (с именами GRAMMAR.TXT и INPUT.TXT). Первый содержит LR(1)-грамматику, второй — описание структуры (записи) на выбранном языке (Pascal, C++, C#). Необходимо проверить, является ли описание структуры корректным с точки зрения заданной грамматики и не содержатся ли в нем конфликты имен.

Таким образом, задание разбивается на две части:

1. Проверка синтаксиса.
2. Проверка семантики.

Семантика 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 изменения: ключевых слов, знаков пунктуации, правил разбора идентификаторов, а также добавление новых языковых конструкций и т.п.

Программа должна проанализировать имеющееся в текстовом файле описание структуры и выдать (в текстовый файл OUTPUT.TXT) результат проверки. Это может быть:

1. Сообщение о том, что грамматика во входном файле не является LR(1)-грамматикой.
2. Сообщение о том, что описание корректное.
3. 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.
4. Сообщение о конфликте имен. В этом случае на выходе программы необходимо указать конфликтующее имя, а также строку и позицию в строке, где произошел конфликт.

3 Краткая теория

3.1 Синтаксис языковых конструкций

3.1.1 Синтаксис описания переменных

Далее приведены общие правила описания переменных на языках Pascal, C++ и C#. Однако различные компиляторы могут иметь отличия в синтаксисе. Поэтому разрабатываемая программа должна считать описание переменных правильным лишь в том случае, если используемый компилятор также считает его правильным (естественно, с учетом используемых в данной лабораторной работе ограничений на синтаксис языка). Аналогично для неправильного описания переменных. Можно проверять описание переменных на других языках программирования. Кроме того, программа может быть написана на одном языке, но проверять правильность описания переменных для другого языка.

3.1.1.1 Язык Pascal

Для языка Pascal описание переменных может выглядеть следующим образом:

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

То есть 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 (целые типы и типы с плавающей точкой, массивы, строки, символьный и булевский типы). Тип может быть также массивом (в том числе многомерным) или строкой. У массива нижняя граница индекса не должна быть меньше верхней. Типом элементов массива может быть любой другой тип данных. 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*…*z*, *A*…*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 ($N \geq 8$), 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 используемой среды разработки). 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) …
```

Пробелы можно обозначать просто в виде пробела, заключенного в кавычки (`' '`).

3.1.1.2 Язык C++

Для языка C++ описание переменных может выглядеть так:

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

То есть описание состоит из указаний типов данных со следующими за ними списками имен переменных. Язык C++ поддерживает 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**. Без данных модификаторов используются типы **float**, **char** и другие.

Та как язык C++ не делает 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)) …
```

или

```
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') …
```

Можно также использовать входной файл в кодировке Unicode, тогда вместо типа **char** используется тип **wchar_t** со всеми вытекающими последствиями, но в данной лабораторной работе в этом нет необходимости.

В языке 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 можно обозначать таким же образом, как и в языке Pascal (`' '`).

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

3.1.1.3 Язык C#

Описание локальных переменных на языке C# может выглядеть следующим образом:

```
double? a, b, c;
int[,] d;
string s1, s2;
long n;
```

Во-первых, в отличие от языка C++, модификаторы здесь отсутствуют, т.е. **`long`** — это отдельный тип данных. Также требуется реализация всех остальных целых типов, типов с плавающей точкой, символьного и булевского типа, массивов и строк, а также обнуляемых типов и типа **`object`**. При указании типа можно использовать как имя класса, так и соответствующий ему синоним языка C# (см. табл. 3.1).

Табл. 3.1 — Описание типов языка C#

Класс	Синоним	Описание
`System.SByte`	**`sbyte`**	Байт со знаком
`System.Int16`	**`short`**	Короткое целое со знаком
`System.Int32`	**`int`**	Целое со знаком
`System.Int64`	**`long`**	Длинное целое со знаком
`System.Byte`	**`byte`**	Байт
`System.UInt16`	**`ushort`**	Короткое целое без знака
`System.UInt32`	**`uint`**	Целое без знака
`System.UInt64`	**`ulong`**	Длинное целое без знака
`System.Single`	**`float`**	Число с плавающей точкой одинарной точности
`System.Double`	**`double`**	Число с плавающей точкой двойной точности
`System.Decimal`	**`decimal`**	Число с плавающей точкой повышенной точности
`System.Boolean`	**`bool`**	Логический (булев) тип
`System.Char`	**`char`**	Символьный тип
`System.String`	**`string`**	Строка
`System.Object`	**`object`**	Объект

Пространство имен `System` указывать не обязательно. Таким образом, например, `System.Boolean`, `Boolean` и **`bool`** — это синонимы. Массивы могут быть одномерными или многомерными — прямоугольными, ортогональными (неровными) или комбинированными. За основу для описания обнуляемого типа может быть взят лишь тип по значению (т.е. в данном случае все типы, кроме **`string`** и **`object`**).

Для сравнения символа с его кодом можно использовать либо явное приведение типа

```
if(ch == (char)13) …
```

или его запись в виде символа в кодировке Unicode (UTF-16):

```
if(ch == '\u000d') …
```

Вместо записи `'\u000d'` можно использовать `'\x000d'`.

Также можно использовать описанные выше для языка C++ спецсимволы.

3.1.2 Синтаксис описания записей и структур

Здесь изложены типовые правила описания записей в языке Pascal и структур в языках C++/C#. Как и в предыдущем разделе, для обнаружения синтаксических или семантических ошибок в различных частных случаях следует ориентироваться на соглашения, принятые в используемом компиляторе. Дополнительную информацию (правила описания переменных, идентификаторов, наличие разделителей и т.п.) см. в п. 3.1.1. Также можно проверять описание аналогичных конструкций в других языках программирования. Кроме того, программа может быть написана на одном языке, но проверять правильность описания записей или структур для другого языка.

3.1.2.1 Язык Pascal

В языке Pascal описание экземпляра записи выглядит как описание обычной переменной, имеющей тип **`record`**:

```
var a, b, c: record
    …
end;
d: array [1..6, 6..9] of record
    …
end;
```

Если же требуется описание нового типа записи, а не экземпляра, используется оператор **type**:

```
type a = record
    …
end;
d = array [1..6, 6..9] of record
    …
end;
```

Внутри записи могут находиться объявления полей — т.е. это описание переменных любого типа (подобно примеру из п. 3.1.1.1), но уже без ключевого слова **var**. Поля, в свою очередь, могут быть вложенными записями. Переменные также могут быть объявлены глобально (т.е. не вложенными ни в какую запись), тогда уже наличие ключевого слова **var** обязательно.

Конфликтом имен считается наличие двух и более идентификаторов (имен переменных или типов) в глобальной области видимости либо внутри одной и той же записи на одинаковом уровне вложенности.

3.1.2.2 Язык C++

В языке C++ объявить структуру можно либо при помощи оператора **typedef**, либо без него:

```
typedef struct a1
{
    …
} b1, c1[10];
struct a2
{
    …
} b2, c2[10];
```

Разница в том, что с оператором **typedef** объявляются новые типы, а без него — экземпляры структур. Так, `a1`, `b1` и `c1` — это типы данных (первые два эквивалентны и описывают структуру, третий описывает массив из 10 структур). Идентификатор `a2` — это тоже тип данных, соответствующий описанной далее структуре, а вот `b2` и `c2` — это уже экземпляр структуры `a2` и массив из 10 экземпляров структуры `a2` соответственно.

Внутри записи могут находиться объявления вложенных типов структур и полей — это описание переменных любого типа (подобно примеру из п. 3.1.1.2). Переменные также могут быть объявлены глобально (т.е. не вложенными ни в какую структуру). Дополнительно необходимо реализовать поддержку модификаторов знака (**signed**/**unsigned**). Поля, в свою очередь, могут быть вложенными структурами.

При объявлении структуры можно опустить ее заголовок:

```
typedef struct
{
    …
} b1, c1[10];
struct
{
    …
} b2, c2[10];
```

Либо можно опустить список идентификаторов после закрывающей скобки:

```
typedef struct a1
{
    …
};
struct a2
{
    …
};
```

Можно опустить и то, и другое, тогда структура станет *анонимной*. Анонимная структура может быть только вложенной в другую структуру:

```
struct a2
{
    struct {
        int a, b;
    };
    struct {
        int c, d;
    };
};
```

Ее поля считаются полями структуры вышестоящего уровня, что необходимо учитывать при определении конфликтов имен.

Конфликтом имен считается наличие двух и более идентификаторов (имен переменных или типов) в глобальной области видимости либо внутри одной и той же записи на одинаковом уровне вложенности. Причем имя типа и экземпляра может совпадать, это конфликтом не является:

```
int a;
struct a {};
```

Также в языке C++ конфликтной является ситуация, когда имя экземпляра или типа, вложенного в структуру, совпадает с именем этой структуры:

```
struct a {
    int a;
};
```

Причина в том, что в классе или структуре с именем *X* только один член может иметь имя *X* — это конструктор.

Также при описании структуры можно использовать модификаторы доступа (**public**, **protected** и **private**), но добавлять их в синтаксис не обязательно.

3.1.2.3 Язык C#

В языке C# описание структуры может предваряться модификатором доступа. Если структура описана глобально, то доступны два модификатора — **public** и **internal**. Если структура вложенная, то еще модификатор **private**:

```
public struct a {
    private struct b {
    }
}
```

Если модификатор доступа опущен, то по умолчанию предполагается самый строгий модификатор (**internal** для глобальных и **private** для вложенных структур соответственно). Также, кроме вложенных структур, структура может содержать поля — это описание переменных любого типа (подобно примеру из п. 3.1.1.3). Глобальное описание переменных запрещено. Поля

также могут предваряться модификатором доступа **public**, **internal** или **private**. Если модификатор опущен, по умолчанию предполагается **private**. В конце описания структуры может стоять точка с запятой, но ее наличие не обязательно.

Конфликтом имен считается наличие двух и более идентификаторов (имен переменных или типов) в глобальной области видимости либо внутри одной и той же записи на одинаковом уровне вложенности.

В языке C#, как и в языке C++, конфликтной является ситуация, когда имя экземпляра или типа, вложенного в структуру, совпадает с именем этой структуры:

```
struct a {
    int a;
}
```

Причина та же самая.

3.1.3 Синтаксис описания функций, процедур и делегатов

Здесь изложены типовые правила описания процедур и функций в языках Pascal/C++ и делегатов в языке C#. Как и в предыдущих разделах, для обнаружения синтаксических или семантических ошибок в различных частных случаях следует ориентироваться на соглашения, принятые в используемом компиляторе. Дополнительную информацию (правила описания переменных, идентификаторов, наличие разделителей и т.п.) см. в п. 3.1.1. Также можно проверять описание аналогичных конструкций в других языках программирования. Кроме того, программа может быть написана на одном языке, но проверять правильность описания функций или процедур для другого языка.

3.1.3.1 Язык Pascal

Функции и процедуры в языке Pascal могут быть описаны глобально, либо внутри других процедур и функций:

```
procedure f1(a, b: integer); forward;
procedure f2;
  procedure f3(var c: integer);
  begin
```

```
    end;
    procedure f4; forward;
    function f5(i: integer; var obj): integer;
    begin
    end;
    procedure f4;
    begin
    end;
begin
end;
```

Описание функции начинается с ключевого слова **function**, процедуры — с ключевого слова **procedure**. Имя функции или процедуры является обычным идентификатором. Далее в скобках приводится список параметров, который включает описание формальных аргументов, подобно описанию переменных (см. п. 3.1.1.1). Отличие в том, что наличие ключевого слова **var** не обязательно. Однако оно может ставиться перед любыми аргументами, передаваемыми в функцию или процедуру по ссылке, а не по значению. Также, если аргумент передается по ссылке, его тип можно не указывать (в этом случае предполагается, что аргумент может иметь любой тип). Если список аргументов пуст, скобки не ставятся.

Затем, если описывается функция, ставится двоеточие с типом возвращаемого значения, после чего — точка с запятой. Если описывается процедура, то сразу ставится точка с запятой.

После точки с запятой указывается ключевое слово **forward**, если это предварительное описание прототипа процедуры или функции, либо операторные скобки **begin**/**end** с телом процедуры или функции, если это ее реализация. Для упрощения синтаксиса будем полагать тела процедур и функций пустыми. Прототип какой-либо процедуры или функции можно описать лишь один раз. Проверка, что всем прототипам соответствует реализация, не требуется. Завершается описание процедуры или функции точкой с запятой.

Конфликтом имен будет ситуация, когда:

– совпадают имена аргументов в списке у одной и той же процедуры или функции;

– глобально или на одном и том же уровне вложенности имеются процедуры или функции с одинаковыми именами.

3.1.3.2 Язык C++

Функции и процедуры в языке C++ могут быть описаны глобально либо внутри пространства имен. Пространства имен также могут быть вложенными друг в друга:

```
int f1(int a, double b, char c[10]);
namespace N1 {
    namespace N2 {
        void f2(void);
        void f3();
        int f4(long int, ...);
    }
    void f5(int, unsigned char) {}
    void f5(int) {}
    void f5(double) {}
}
```

Описание пространства имен начинается с ключевого слова **namespace** и имени пространства, являющегося обычным идентификатором. Содержимое пространства имен заключается в фигурные скобки.

Описание функции начинается с типа возвращаемого значения. Это все возможные типы, описанные в п. 3.1.1.2, дополненные модификаторами знака (**signed**/**unsigned**). Описание процедуры — со специального пустого типа **void**. Имя функции или процедуры является обычным идентификатором. Далее в скобках приводится список параметров, который может:

– Включать описание формальных аргументов, подобно описанию переменных (см. п. 3.1.1.2), но разделяемых запятыми, а не точками с запятой. Также, в отличие от обычного описания переменных, после типа может либо вообще не быть идентификатора, либо лишь один идентификатор. Если есть хотя бы один аргумент подобного типа, то последним аргументом может быть «...», что означает переменное число аргументов в списке.

– Содержать специальное ключевое слово **`void`**, означающее, что функция или процедура не имеет параметров.
– Быть пустым. Это также означает, что аргументов у функции или процедуры нет.

После скобок ставится точка с запятой, если это предварительное описание прототипа процедуры или функции, либо фигурные скобки с телом процедуры или функции, если это ее реализация. Для упрощения синтаксиса будем полагать тела процедур и функций пустыми. Проверка, что всем прототипам соответствует реализация, не требуется.

Две глобальные реализации функции или процедуры либо реализации функции и процедуры, находящиеся на одном уровне вложенности, могут иметь одинаковое имя, если в списках их формальных аргументов типы аргументов отличаются. Для прототипов такого ограничения нет — можно описать любое количество совпадающих прототипов процедур или функций. Конфликтом имен будет ситуация, когда:

– совпадают имена аргументов в списке у одной и той же процедуры или функции (не важно, прототип это или реализация);
– глобально или на одном и том же уровне вложенности описано пространство имен и процедура или функция с одинаковыми именами (не важно, прототип это или реализация);
– глобально или на одном и том же уровне вложенности имеются реализации процедуры или функции с одинаковыми именами и одинаковыми типами аргументов в списках формальных аргументов.

Ситуация, когда глобально или на одном и том же уровне вложенности описаны пространства имен с одинаковыми именами, ошибкой не является. В этом случае их содержимое просто объединяется.

3.1.3.3 Язык C#

Методы в языке C# могут быть описаны лишь внутри класса, структуры или интерфейса, поэтому глобально либо внутри пространства имен можно описать лишь делегаты. Пространства имен также могут быть вложенными друг в друга:

```
delegate int f1(int a, double b, char[] c);
namespace N1 {
    namespace N2.N3 {
        delegate void f2();
        delegate int f3(params object[] obj);
    }
    delegate void f4(int i, ref byte b);
}
```

Описание пространства имен начинается с ключевого слова **namespace** и имени пространства, являющегося либо обычным идентификатором, либо идентификаторами, разделенными точками. В последнем случае запись

```
namespace N2.N3 {
    ...
}
```

аналогична записи

```
namespace N2 {
    namespace N3 {
        ...
    }
}
```

Содержимое пространства имен заключается в фигурные скобки. Описание делегата начинается с ключевого слова **delegate** и типа возвращаемого значения. Это все возможные типы, описанные в п. 3.1.1.3, либо специальный пустой тип **void**. Имя делегата является обычным идентификатором. Далее в скобках приводится список параметров, который может:

– Включать описание формальных аргументов, подобно описанию переменных (см. п. 3.1.1.3), но разделяемых запятыми, а не точками с запятой. Также, в отличие от обычного описания переменных, после типа должен располагаться лишь один идентификатор. У аргумента, передаваемого по ссылке, ставится модификатор **ref** или **out**. У последнего аргумента в списке может быть модификатор **params**, означающий список аргументов переменной длины. При этом типом аргумента должен быть одномерный массив, например, **object**[], **int**[][,] и т.д.

– Быть пустым. Это означает, что аргументов у делегата нет.

После скобок ставится точка с запятой. Конфликтом имен будет ситуация, когда:

– совпадают имена аргументов в списке у одного и того же делегата;
– глобально или на одном и том же уровне вложенности описано пространство имен и делегат с одинаковыми именами;
– глобально или на одном и том же уровне вложенности описаны делегаты с одинаковыми именами.

Ситуация, когда глобально или на одном и том же уровне вложенности описаны пространства имен с одинаковыми именами, ошибкой не является. В этом случае их содержимое просто объединяется.

3.2 Разбор математического выражения

3.2.1 Построение дерева

Разбор математического выражения начинается с разбивки входной цепочки на бинарное дерево лексем. Далее по этому дереву проводится лексический и синтаксический анализ, строится таблица имен и код. Элементом дерева является следующая структура:

```
declare 1 TREE,
          2 oper string,
          2 level int,
          2 code string,
          2 left pointer,
          2 right pointer;
declare root pointer;
```

Идентификатор `root` ссылается на корень дерева. Каждый элемент дерева содержит строку `oper` (это знак операции или операнд — переменная либо константа), уровень `level` и строку кода `code` (эти поля заполняются на этапе построения кода), а также указатели `left` и `right` на левое и правое поддерево. Если `oper` — это знак операции, имеем узел дерева. Если это операнд, имеем лист, в этом случае указатели `left` и `right` являются нулевыми. В выражении используются только бинарные

операции, поэтому дерево является полным бинарным деревом — либо оба указателя нулевые, либо оба ненулевые.

Рекурсивный алгоритм построения дерева:

1. На входе алгоритма имеем некоторый узел дерева `node` и строку с частью математического выражения `expr`.
2. Ищем в строке знаки операций, не заключенные в скобки, в порядке приоритета. Наименьший приоритет у присваивания («=»), средний у сложения («+»), наивысший — у умножения («*»).
3. Если знак операции найден в позиции `pos` строки `expr`, то записываем его в поле `oper` узла дерева `node`. Далее два раза вызываем рекурсивно данный алгоритм — сначала для левого поддерева `node.left` и для подстроки строки `expr`, расположенной слева от позиции `pos`, и затем для правого поддерева `node.right` и для подстроки строки `expr`, расположенной справа от позиции `pos`.
4. Если знак не найден, то:

   4.1. Если при этом первым символом строки `expr` является открывающая скобка «(», а последним — закрывающая «)», то удалить их и вернуться на шаг 2.

   4.2. В противном случае имеем лист дерева. Записываем в поле `oper` узла дерева `node` все выражение `expr` — оно содержит либо идентификатор, либо константу. Указатели `node.left` и `node.right` обнулить.

Для начала работы алгоритма запускаем его, передавая на вход указатель `root` и строку, содержащую исходное выражение. В качестве примера возьмем выражение

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

Для него данный алгоритм получит дерево, изображенное на рис. 3.1.

Замечания:

1. Строку для выделения подстрок можно не резать на части, а просто вместе со строкой передавать в алгоритм позиции первого и последнего символа, подлежащих обработке.

2. Код может получаться разной степени оптимальности, в зависимости от структуры дерева. Здесь предложен простейший алгоритм построения дерева. Для достижения большей опти-

мальности кода дерево после построения можно модифицировать, сопоставляя структуру исходного математического выражения, структуру дерева и полученный код.

Далее рассмотрим краткую теорию преобразования математического выражения в псевдокод, а также оптимизации кода [1].

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

3.2.3 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. 3.2).

Табл. 3.2 — 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.

3.2.4 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. 3.1.

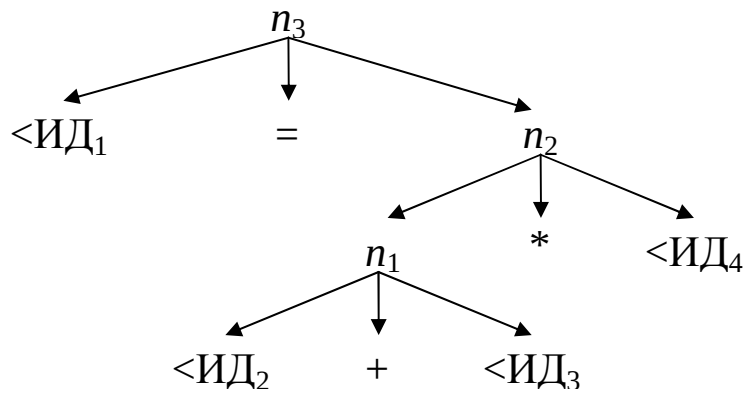

Рис. 3.1 — Последовательность действий при вычислении выражения

То есть 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.

3.2.5 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. 3.3).

Табл. 3.3 — 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. 3.1). Есть три типа внутренних вершин, зависящих от того, каким из знаков помечен средний потомок: «=», «+» или «*» (рис. 3.2). Здесь треугольники — произвольные поддеревья (в том числе, состоящие из единственной вершины).

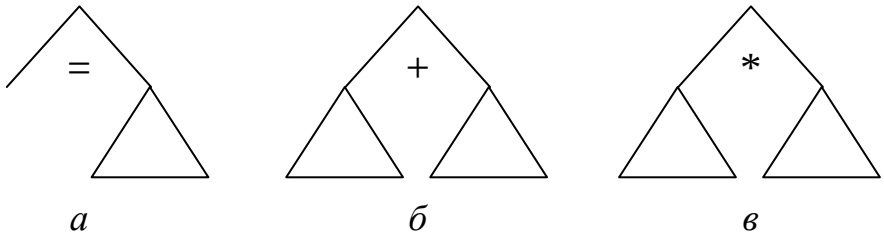

Рис. 3.2 — Типы вершин

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. 3.3).

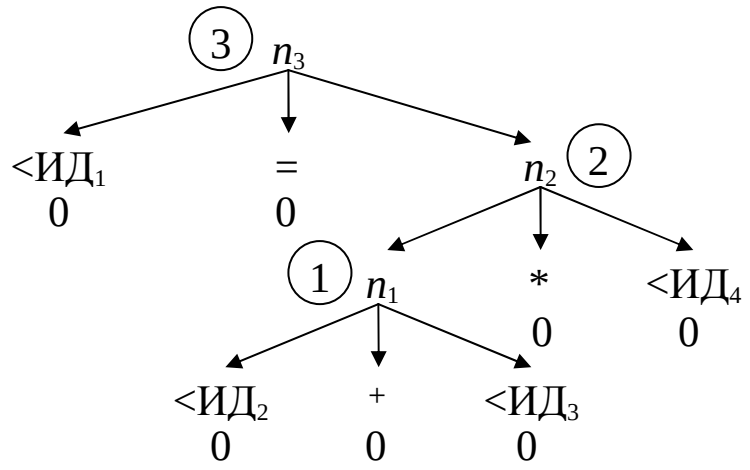

Рис. 3.3 — Дерево с уровнями

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.

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 выражений 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. 3.4).

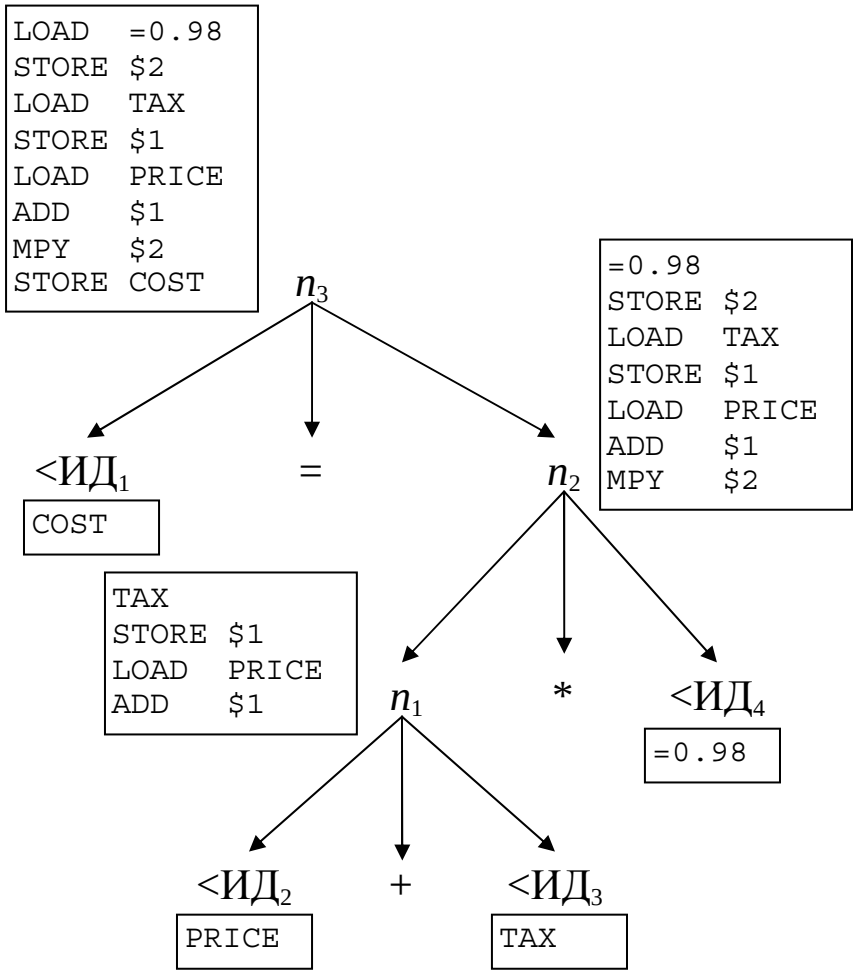

Рис. 3.4 — Дерево с генерированными кодами

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.

3.2.6 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 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. 3.4).

Табл. 3.4 — 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`
	`LOAD =0.98`
	`STORE $2`
	`LOAD TAX`
		`LOAD TAX`
		`ADD PRICE`
		`MPY =0.98`

Окончание табл. 3.4

Этап 1	Этап 2	Этап 3
STORE $1 LOAD $1 ADD PRICE MPY $2 STORE COST	ADD PRICE MPY $2 STORE COST	STORE COST

3.3 Программирование ДМП-автоматов

Все языки программирования определяются детерминированными конечными автоматами с магазинной памятью [1].

3.3.1 Основные определения

Детерминированный конечный автомат с магазинной памятью (ДМП-автомат, или ДМПА) — это семерка

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

где:

- Q — конечное множество символов состояния, представляющих всевозможные состояния управляющего устройства;
- Σ — конечный входной алфавит;
- Γ — конечный алфавит магазинных символов;
- δ — функция переходов, отображение множества $Q\times(\Sigma\cup\{e\}\cup\{\perp\})\times\Gamma$ во множество $Q\times\Gamma^*$;
- $q_0\in Q$ — начальное состояние управляющего устройства;
- $Z_0\in\Gamma$ — символ, находящийся в магазине в начальный момент (начальный символ);
- $F\subseteq Q$ — множество заключительных состояний.

Конфигурацией ДМП-автомата P называется тройка $(q, w, \alpha)\in Q\times\Sigma^*\times\Gamma^*$, где:

- q — текущее состояние устройства;
- w — неиспользованная часть входной цепочки; первый символ цепочки w находится под входной головкой; если $w = e$, то считается, что вся входная лента прочитана;
- α — содержимое магазина; самый левый символ цепочки α считается верхним символом магазина; если $\alpha = e$, то магазин считается пустым.

Здесь «⊥» — специальный маркер, обозначающий конец входной цепочки. То есть считаем, что на входе ДМПА находится символ «⊥», если вся входная цепочка прочитана ($w = e$).

Такт работы ДМП-автомата P будет представляться в виде бинарного отношения $\vdash$, определенного на конфигурациях. Будем писать

$$(q, aw, Z\alpha) \vdash (q', w, \gamma\alpha),$$

если функция переходов $\delta(q, a, Z) = (q', \gamma)$, где $q, q' \in Q$, $a \in \Sigma \cup \{e\} \cup \{\perp\}$, $w \in \Sigma^*$, $Z \in \Gamma$, $\alpha, \gamma \in \Gamma^*$.

Если $\delta(q, a, Z) = (q', \gamma)$, то говорят о том, что ДМП-автомат P, находясь в состоянии q и имея a в качестве текущего входного символа, расположенного под входной головкой, а Z — в качестве верхнего символа магазина, может:

- перейти в состояние q';
- сдвинуть головку на одну ячейку вправо;
- заменить верхний символ магазина цепочкой γ магазинных символов.

Если $Z = e$, то символ из стека не удаляется, т.е. верхний символ магазина не принимается в расчет. Если $\gamma = e$, то верхний символ удаляется из магазина, тем самым магазинный список сокращается. Если $a = e$, будем называть этот такт e-тактом. В e-такте текущий входной символ не принимается во внимание и входная головка не сдвигается. Однако состояние управляющего устройства и содержимое памяти могут измениться.

Начальной конфигурацией ДМП-автомата P называется конфигурация вида (q_0, w, Z_0), где $w \in \Sigma^*$, т.е. управляющее устройство находится в начальном состоянии, входная лента содержит цепочку, которую нужно распознать, и в магазине есть только начальный символ Z_0. *Заключительная конфигурация* — это конфигурация вида (q, e, α), где $q \in F$ и $\alpha \in \Gamma^*$.

Говорят, что цепочка w допускается ДМП-автоматом P, если $(q_0, w, Z_0) \vdash^* (q, e, \alpha)$ для некоторых $q \in F$ и $\alpha \in \Gamma^*$. А $L(P)$ — т.е. *язык, определяемый автоматом* P, — это множество цепочек, допускаемых автоматом P.

В простейшем случае, когда ДМПА не использует стек, т.е. все его конфигурации имеют вид (q, w, e), он вырождается в обычный детерминированный конечный автомат (ДКА) $M = (Q, \Sigma, \delta, q_0, F)$, где:

- Q — конечное множество состояний;
- Σ — конечное множество допустимых входных символов;
- δ — функция переходов, отображение множества $Q \times (\Sigma \cup \{\perp\})$ во множество Q;
- $q_0 \in Q$ — начальное состояние управляющего устройства;
- $F \subseteq Q$ — множество заключительных состояний.

То есть ДКА — это частный случай ДМПА.

3.3.2 Способы задания ДМП-автомата

Существуют различные способы задания ДМПА — например, математическое описание (в виде рассмотренной выше семерки). Однако для наглядности функцию переходов можно задавать в виде графа переходов или таблицы переходов.

Так, на рис. 3.5 изображена часть графа переходов ДМПА, соответствующая переходу $\delta(q, a, Z) = (q', \gamma)$.

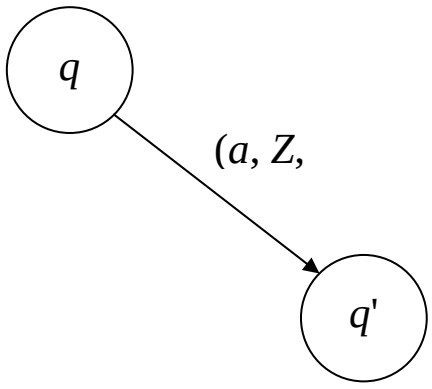

Рис. 3.5 — Переход в графе ДМПА

В графе переходов ДКА дуги достаточно помечать только символами алфавита Σ (рис. 3.6).

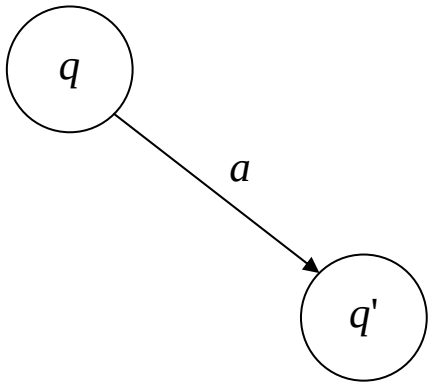

Рис. 3.6 — Переход в графе ДКА

Если состояние q является конечным, т.е. $q \in F$, то на графе оно отображается в виде окружности с двойной границей (рис. 3.7).

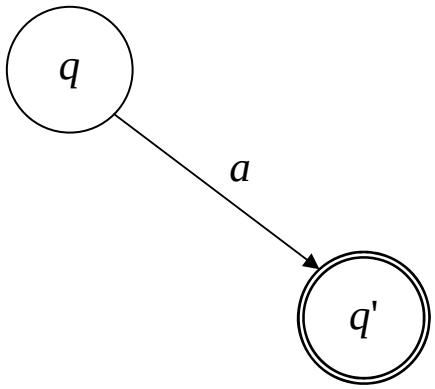

Рис. 3.7 — Переход в конечное состояние ДКА

Проблема в том, что в ДМПА конечное состояние может быть достижимо только при определенном состоянии стека. На графе для этого приходится вводить новые искусственные состояния. Например, на рис. 3.8 изображена ситуация, когда состояние q' является конечным лишь в том случае, если на вершине стека находится символ Z'.

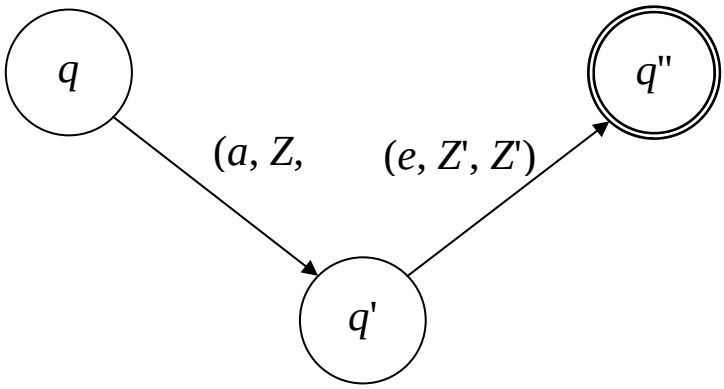

Рис. 3.8 — Переход в конечное состояние ДМПА

Дополнительно примем соглашение, что если $Z = \varnothing$, то это соответствует требованию пустоты стека.

В таблице переходов ДМПА каждому состоянию соответствует отдельная строка таблицы, а каждой допустимой комбинации (a, Z), $a \in \Sigma \cup \{e\} \cup \{\perp\}$, $Z \in \Gamma$, — отдельный столбец. Комбинации (a, Z) должны быть такими, что в любой конфигурации ДМПА существует только один вариант перехода в новое состояние. Ячейки таблицы $\delta(q, a, Z)$ могут принимать следующие варианты значений:

- Пара (q', γ), соответствующая переходу в состояние q' и запись цепочки на вершину γ стека;
- Элемент *HALT*, соответствующий успешному завершению разбора, т.е. ситуации, когда $a = \perp$, $q \in F$ (на графе отображается как окружность с двойной границей);
- Элемент *ERROR*, соответствующий синтаксической ошибке (на графе это ситуация, когда отсутствует дуга из состояния q в состояние q' с меткой (a, Z, γ)).

Табл. 3.5 — Таблица переходов ДМПА

	(a_1, Z_1)	(a_2, Z_2)	(a_3, Z_3)	…
q_0	$\delta(q_0, a_1, Z_1)$	$\delta(q_0, a_2, Z_2)$	$\delta(q_0, a_3, Z_3)$	…
q_1	$\delta(q_1, a_1, Z_1)$	$\delta(q_1, a_2, Z_2)$	$\delta(q_1, a_3, Z_3)$	…
q_2	$\delta(q_2, a_1, Z_1)$	$\delta(q_2, a_2, Z_2)$	$\delta(q_2, a_3, Z_3)$	…
…	…	…	…	…

Так, графу на рис. 3.8 будет соответствовать таблица переходов вида табл. 3.6.

Табл. 3.6 — Таблица переходов ДМПА с конечным состоянием

	(a, Z)	(e, Z')	$(\perp, Z')$
q	(q', γ)	*ERROR*	*ERROR*
q'	*ERROR*	(q'', Z')	*ERROR*
q''	*ERROR*	*ERROR*	*HALT*

Однако, с использованием таблицы переходов, можно обойтись без новых искусственных состояний, чтобы показать достижимость конечного состояния ДМПА (табл. 3.7).

Табл. 3.7 — Упрощенная таблица переходов ДМПА с конечным состоянием

	(a, Z)	$(\perp, Z')$
q	(q', γ)	*ERROR*
q'	*ERROR*	*HALT*

В принципе, оба варианта таблицы допустимы. В дальнейшем для простоты вместо записи элемента *ERROR* будем оставлять ячейки таблиц пустыми.

В таблице переходов ДКА каждому состоянию соответствует отдельная строка таблицы, а каждому допустимому входному символу $a \in \Sigma \cup \{\perp\}$ — отдельный столбец. Ячейки таблицы $\delta(q, a)$ могут принимать следующие варианты значений:

- Элемент q', соответствующий переходу в состояние q';
- Элемент *HALT*, соответствующий успешному завершению разбора, т.е. ситуации, когда $a = \perp$, $q \in F$ (на графе отображается как окружность с двойной границей);
- Элемент *ERROR*, соответствующий синтаксической ошибке (на графе это ситуация, когда отсутствует дуга из состояния q в состояние q' с меткой a).

Табл. 3.8 — Таблица переходов ДКА

	a_1	a_2	…	$\perp$
q_0	$\delta(q_0, a_1)$	$\delta(q_0, a_2)$	…	$\delta(q_0, \perp)$
q_1	$\delta(q_1, a_1)$	$\delta(q_1, a_2)$	…	$\delta(q_1, \perp)$
q_2	$\delta(q_2, a_1)$	$\delta(q_2, a_2)$	…	$\delta(q_2, \perp)$
…	…	…	…	…

В общем случае, построение графа или таблицы переходов конечного автомата — задача неформализованная, и предполагает некоторый творческий подход. Можно лишь дать некоторые рекомендации:

1. Проще всего сначала построить граф переходов, а потом по описанным выше правилам преобразовать его в таблицу переходов.
2. Построение графа начинается с начального состояния q_0. Если начальное состояние может являться также и конечным, помечаем это двойной границей окружности.
3. Для каждого состояния графа q_i определяем, есть ли из данного состояния такие переходы (a, Z, γ), которые соответствуют допустимому символу a из входной цепочки и допустимому символу Z на вершине стека (если автомат с магазинной памятью), которые пока еще отсутствуют в графе. Если есть, то проверяем, ведет ли данный переход в уже имеющееся состояние. Если да, то добавляем в граф только новый переход (a, Z, γ). Если нет, то добавляем в граф новое состояние и переход (a, Z, γ) в него. Если новое состояние может являться конечным, помечаем это двойной границей окружности.
4. Если в процессе выполнения шага 3 в графе появились новые состояния или переходы, возвращаемся на шаг 3, иначе граф переходов построен.

Далее полученный автомат необходимо минимизировать, т.е. убрать из него неразличимые и недостижимые состояния [1]. Очевидно, что для одного и того же языка можно построить целый ряд различных конечных автоматов, хотя и можно будет доказать их эквивалентность.

3.3.3 Включение действий в синтаксис и алгоритм разбора

Согласно теории [1], действия в синтаксический анализатор (конечный автомат или КС-грамматику) включаются в следующем виде:

$$\langle A_1 \rangle, \langle A_2 \rangle, \ldots$$

То есть идентификатор действия заключается в угловые скобки. Таким образом, функция переходов ДМПА при включении действий в синтаксис имеет вид $\delta(q, a, Z) = (q', \gamma, \langle A \rangle)$. Соответственно, для ДКА $\delta(q, a) = (q', \langle A \rangle)$. Если функция переходов не подразумевает никаких действий, то этот факт можно обозначить как $\langle A \rangle = e$ или $\langle A \rangle = \varnothing$.

Само действие не относится к синтаксическому анализу (это уже часть семантического анализа), поэтому в программе реализуется отдельно. Например, если конечный автомат проверяет правильность описания переменных, то для него идентификаторы с одинаковыми именами ошибкой не являются, т.к. для него это просто последовательности букв и цифр. Другое дело, что с семантической точки зрения, т.е. с точки зрения смысла, который мы вкладываем в эти последовательности, а именно — что это имена идентификаторов, их совпадения являются семантической ошибкой. Поэтому в программе необходимо предусмотреть действия, которые обеспечивали бы создание списка уже прочитанных идентификаторов, и при встрече каждого нового идентификатора проверяли бы, не встречался ли идентификатор с таким же именем ранее.

Теперь можно сформулировать полный алгоритм разбора входной цепочки с помощью ДМПА, включая выполнение действий. Для разбора по таблице ДМПА нам потребуется магазин (стек) M и собственно входная цепочка $\alpha = a_1 a_2 \ldots a_n \perp$. Алгоритм разбора:

1. В начале работы автомат находится в начальном состоянии $q := q_0$, содержимое стека $M := Z_0$, разбор начинается с первого символа входной цепочки ($k := 1$).
2. Ищем в таблице переходов ячейку, расположенную на пересечении строки, соответствующей текущему состоянию q, и столбца, соответствующего комбинации (a, Z), где:
   – a — текущий символ входной цепочки ($a = a_k$), Z — элемент на вершине стека ($M = Z\beta$) или
   – a — текущий символ входной цепочки ($a = a_k$), $Z = e$ или
   – $a = e$, Z — элемент на вершине стека ($M = Z\beta$).

Если такая ячейка не найдена, то разбор окончен, имеем во входной цепочке синтаксическую ошибку в позиции k. Если таких ячеек несколько — значит, таблица переходов построена неверно (в разных столбцах таблицы имеются одинаковые комбинации (a, Z)). Если она одна, то дальнейшее действие определяется значением функции переходов $\delta(q, a, Z)$ для данной ячейки.

3. Если $\delta(q, a, Z) = (q', \gamma, \langle A \rangle)$, то:

   3.1. Если действие $\langle A \rangle$ определено (т.е. $\langle A \rangle \neq e$ и $\langle A \rangle \neq \varnothing$), то выполнить действие $\langle A \rangle$.

   3.2. Осуществляем переход в состояние q': $q := q'$.

   3.3. Производим замену символов на вершине стека: $M := \gamma\beta$. При этом, если $Z = e$ и $\gamma \neq e$, цепочка символов γ просто помещается на стек. Иначе сначала символ Z со стека снимается, а затем цепочка γ, если она не пустая, помещается на стек.

   3.4. Если $a \neq e$, то переходим к следующему символу входной цепочки: $k := k + 1$.

   Заметим, что действие $\langle A \rangle$ не обязательно выполнять в начале шага 3. Его можно выполнить в конце шага, уже после перехода в состояние q', если это диктуется логикой решаемой задачи. Таким образом, решение о том, когда именно нужно выполнять действия, программист принимает на основе анализа решаемой задачи.

4. Если $\delta(q, a, Z) = HALT$, то разбор успешно завершен.
5. Если $\delta(q, a, Z) = ERROR$, то имеем во входной цепочке синтаксическую ошибку в позиции k.

Для разбора по таблице ДКА нам потребуется лишь сама входная цепочка $\alpha = a_1 a_2 \ldots a_n \perp$. Алгоритм разбора:

1. В начале работы автомат находится в начальном состоянии $q := q_0$, разбор начинается с первого символа входной цепочки ($k := 1$).
2. Ищем в таблице переходов ячейку, расположенную на пересечении строки, соответствующей текущему состоянию q, и столбца, соответствующего текущему символу входной цепочки ($a = a_k$). Если такая ячейка не найдена (т.е. во входной цепочке встретился неизвестный символ),

то разбор окончен, имеем синтаксическую ошибку в позиции k. Если таких ячеек несколько — значит, таблица переходов построена неверно (одинаковые символы алфавита встречаются в разных столбцах таблицы). Если она одна, то дальнейшее действие определяется значением функции переходов $\delta(q, a)$ для данной ячейки.

3. Если $\delta(q, a) = (q', \langle A \rangle)$, то:
   3.1. Если действие $\langle A \rangle$ определено (т.е. $\langle A \rangle \neq e$ и $\langle A \rangle \neq \varnothing$), то выполнить действие $\langle A \rangle$.
   3.2. Осуществляем переход в состояние q': $q := q'$.
   3.3. Переходим к следующему символу входной цепочки: $k := k + 1$.
   Аналогично предыдущему алгоритму, решение о том, когда именно должно быть выполнено действие $\langle A \rangle$, принимает программист.
4. Если $\delta(q, a) = HALT$, то разбор успешно завершен.
5. Если $\delta(q, a) = ERROR$, то имеем во входной цепочке синтаксическую ошибку в позиции k.

3.3.4 Посимвольный разбор цепочек

В том случае, когда элементами алфавита Σ являются отдельные символы, разбор цепочки является посимвольным. То есть символы a_k из входной цепочки считываются по одному, а k является текущей позицией символа во входной цепочке.

Пример 1. Рассмотрим язык L, описывающий число с фиксированной точкой. Такое число может начинаться со знака «+» или «–», далее следует мантисса числа. Разные языки программирования допускают различные формы записи мантиссы, в общем случае они могут быть следующими: «$N.M$», «$N.$», «$.M$», «N», где N — целая, а M — дробная часть числа. В принципе, оба этих числа имеют одинаковый формат — это последовательность из одной и более цифр в диапазоне от 0 до 9.

Как уже отмечалось, для одного и того же языка можно построить различные конечные автоматы. Таким образом, далее мы построим лишь один из возможных вариантов такого автомата.

Анализ задачи показывает, что в данном случае достаточно построения ДКА с посимвольным разбором без включения дей-

ствий в синтаксис. Процесс построения графа ДКА может состоять из следующих шагов:

1. Рисуем начальное состояние q_0.
2. В начале входной цепочки могут находиться символы «+», «–», «.» и «0-9» (т.е. любая цифра из диапазона 0–9). Если в начале цепочки встречаются символы «+» или «–», то дальнейшее поведение автомата будет эквивалентным (т.е. после обоих этих символов во входной цепочке будет одинаковое содержимое). Сделаем по этим символам переход из состояния q_0 в состояние q_1. Переход по символу «.» будет в состояние q_2, а по символам «0-9» — в q_3.
3. В состоянии q_1 (после знака «+» или «–») во входной цепочке ожидается мантисса, т.е. символы «.» и «0-9». Переходы по этим символам будут в уже имеющиеся состояния — по символу «.» в q_2, а по символам «0-9» — в q_3 (т.к. не важно, был знак или нет, правила записи мантиссы от этого не меняются).
4. В состоянии q_2 (после точки со знаком или без, например, «–.» или «.») ожидается только цифра, т.к. мантисса должна состоять как минимум из одной цифры. Поэтому делаем переход в состояние q_4 по символам «0-9».
5. В состоянии q_4 (после точки с цифрой, например, «.5» или «+.5») ожидается цифра, поэтому по символам «0-9» остаемся в состоянии q_4. Также это состояние является конечным, т.к. прочитанная цепочка уже является правильной записью числа с фиксированной точкой.
6. В состоянии q_3 (после цифры со знаком или без знака, например «5» или «–5») ожидается десятичная точка или цифра. По символам «0-9» остаемся в состоянии q_3, а по символу «.» переходим в состояние q_5. Также это состояние является конечным.
7. В состоянии q_5 (после цифры с точкой, например, «5.» или «+5.») ожидается цифра, поэтому по символам «0-9» остаемся в состоянии q_5. Также это состояние является конечным.

Получили граф переходов, изображенный на рис. 3.9.

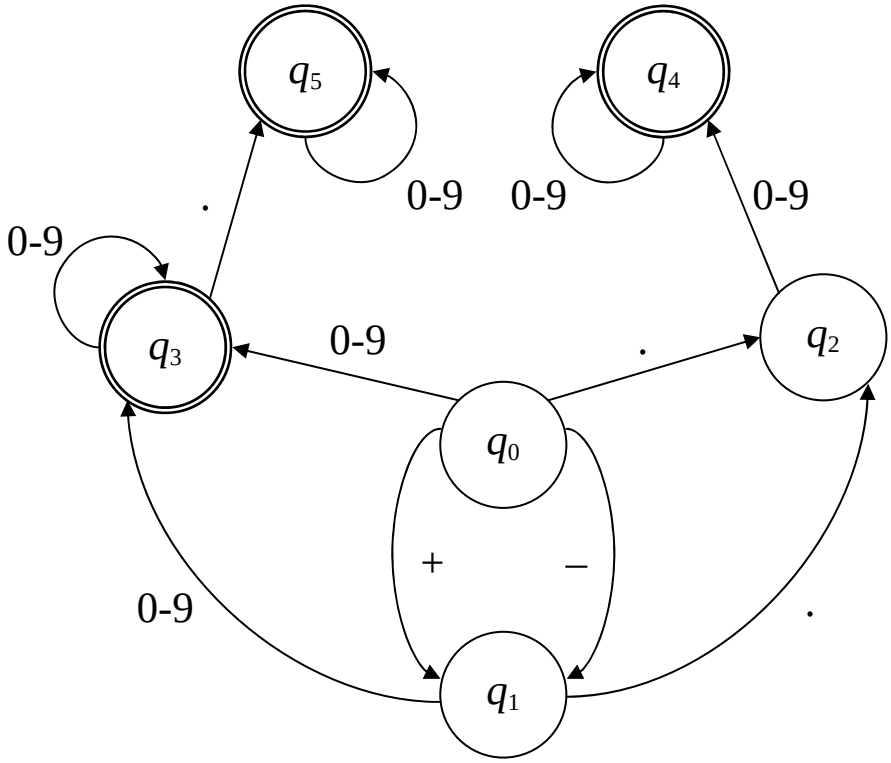

Рис. 3.9 — Граф ДКА, описывающего число с фиксированной точкой

Однако используя алгоритм из учебного пособия [1], можно показать, что состояния q_4 и q_5 являются неразличимыми. Поэтому одно из них можно убрать (рис. 3.10).

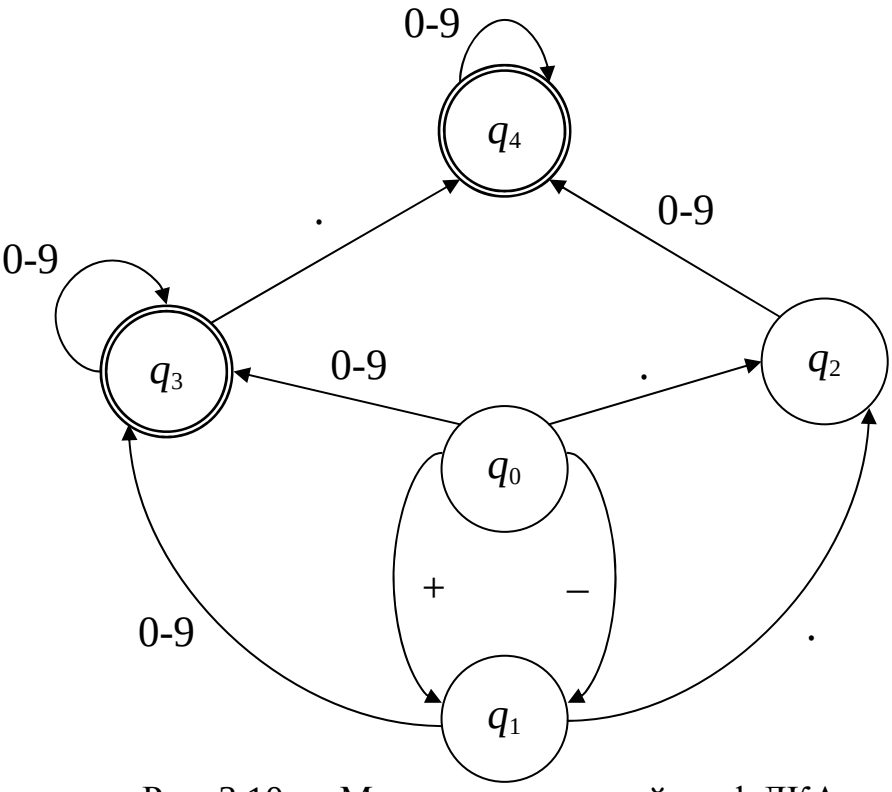

Рис. 3.10 — Минимизированный граф ДКА,
описывающего число с фиксированной точкой

Данному графу будет соответствовать таблица переходов табл. 3.9.

Табл. 3.9 — Таблица переходов ДКА, описывающего число с фиксированной точкой

	+ –	.	0-9	⊥
q_0	q_1	q_2	q_3	
q_1		q_2	q_3	
q_2			q_4	
q_3		q_4	q_3	*HALT*
q_4			q_4	*HALT*

Примечание. Если для ряда символов поведение автомата является идентичным (как для символов «+» и «–» в данном примере или для символов «0», «1», …, «9»), то для уменьшения размеров таблицы их можно помещать в один общий столбец. Но множества символов в различных столбцах не должны пересекаться.

Получили ДКА $M = (Q, \Sigma, \delta, q_0, F)$, где:

– $Q = \{q_0, q_1, q_2, q_3, q_4\}$ — множество состояний;
– $\Sigma = \{+, –, ., 0\text{-}9\}$ — алфавит (подразумевается, что 0–9 — это десять отдельных символов алфавита 0, 1, …, 9, но для удобства они объединены в виде диапазона, как и в столбце таблицы переходов);
– $\delta(Q \times \Sigma) = \{\{q_1, q_2, q_3, \mathit{ERROR}\}, \{\mathit{ERROR}, q_2, q_3, \mathit{ERROR}\}, \{\mathit{ERROR}, \mathit{ERROR}, q_5, \mathit{ERROR}\}, \{\mathit{ERROR}, q_4, q_3, \mathit{HALT}\}, \{\mathit{ERROR}, \mathit{ERROR}, q_4, \mathit{HALT}\}\}$ — функция переходов;
– $F = \{q_3, q_4\}$ — множество конечных состояний.

Пример разбора цепочки «–15.2» по данной таблице:

$$
\begin{aligned}
(q_0, \text{«–15.2⊥»}) &\vdash^1 (q_1, \text{«15.2⊥»}) \\
&\vdash^2 (q_3, \text{«5.2⊥»}) \\
&\vdash^3 (q_3, \text{«.2⊥»}) \\
&\vdash^4 (q_4, \text{«2⊥»}) \\
&\vdash^5 (q_4, \text{«⊥»}) \\
&\vdash^6 \mathit{HALT}
\end{aligned}
$$

Разбор завершен успешно. Другой пример:

$$(q_0, \text{«.2.}\perp\text{»}) \vdash^1 (q_2, \text{«2.}\perp\text{»}) \vdash^2 (q_4, \text{«.}\perp\text{»}) \vdash^3 \mathit{ERROR}$$

Имеем синтаксическую ошибку в позиции 3.

В реальных языках программирования число с плавающей или фиксированной точкой не может иметь бесконечное количество значащих цифр. Например, тип **`float`** в языках C/C++/C# (или **`single`** в языке Pascal) имеет 7–8 значащих цифр, тип **`double`** — 15–16. Тип **`real`** в языке Pascal — 11–12. Чтобы учесть эти ограничения, в ДКА нужно добавить действия (табл. 3.10).

Табл. 3.10 — Таблица переходов ДКА с действиями, описывающего число с фиксированной точкой

	+ –	.	0-9	⊥
q_0	$q_1, \varnothing$	$q_2, \varnothing$	$q_3, \langle A_1 \rangle$	
q_1		$q_2, \varnothing$	$q_3, \langle A_1 \rangle$	
q_2			$q_4, \langle A_1 \rangle$	
q_3		$q_4, \varnothing$	$q_3, \langle A_2 \rangle$	*HALT*
q_4			$q_4, \langle A_2 \rangle$	*HALT*

Здесь действия имеют следующий смысл:

– $\langle A_1 \rangle$ — инициализация счетчика значащих цифр значением 1 (count := 1);
– $\langle A_2 \rangle$ — увеличение счетчика значащих цифр на 1 (count := count + 1) и проверка: если count > N, где N — максимальное количество значащих цифр, то перевести ДКА в состояние *ERROR*.

Пример 2. Пусть язык L описывает идентификатор, заключенный в произвольное количество скобок, от нуля до бесконечности (например, «xyz», «(((abc)))» и т.п.). Скобки должны быть парными, т.е. каждой открывающей скобке должна соответствовать закрывающая. Идентификатор начинается с буквы или подчеркивания («_a-zA-Z»), далее идут буквы, цифры и подчеркивания.

Очевидно, что в данном случае необходимо использовать ДМПА с посимвольным разбором. Используя стек (магазин), ДМПА будет следить за парностью скобок. Включение действий в синтаксис не требуется.

На этот раз построим граф ДМПА сразу минимизированным:

1. Рисуем начальное состояние q_0.
2. В начале входной цепочки может находиться открывающая скобка, или сразу идентификатор, если скобок нет. Таким образом, по символу «(» возвращаемся в состояние q_0 (при этом скобку помещаем на стек), а по символам «_a-zA-Z» — в состояние q_1.
3. В состоянии q_1 во входной цепочке может находиться дальнейшее описание идентификатора (символы «_a-zA-Z» или «0-9»), в этом случае остаемся в состоянии q_1, либо начинаются закрывающие скобки. Встречая символ «)», необходимо убедиться, что на стеке есть соответствующая открывающая скобка, и в этом случае перейти в состояние q_2, убрав эту скобку со стека. Также это состояние может являться конечным, если стек пуст (т.е. если скобок вокруг идентификатора не было).
4. В состоянии q_2 продолжаем считывать лишь оставшиеся скобки. То есть поведение автомата в этом состоянии для символов «)» и «⊥» эквивалентно его поведению в состоянии q_1.

Получили граф переходов, изображенный на рис. 3.11.

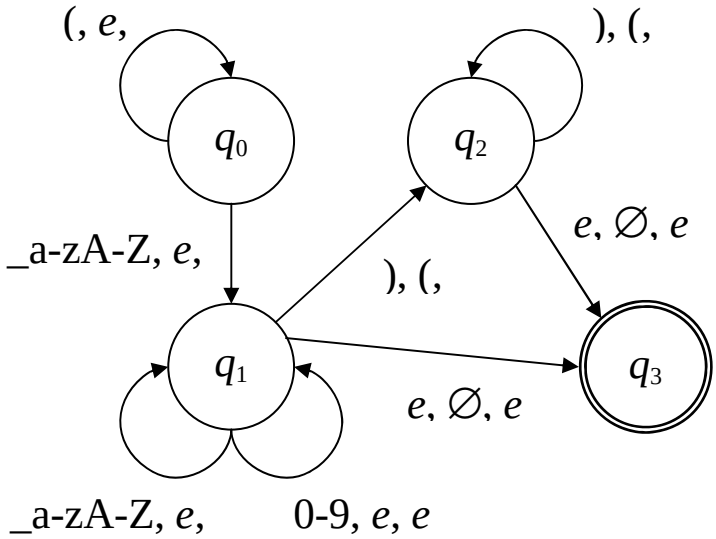

Рис. 3.11 — Граф ДМПА, описывающего идентификатор в скобках

Данному графу будет соответствовать таблица переходов табл. 3.11.

Табл. 3.11 — Таблица переходов ДМПА, описывающего идентификатор в скобках

	(, *e*	_a-zA-Z, *e*	0-9, *e*	), (	*e*, ∅	⊥, ∅
q_0	q_0, (	q_1, *e*				
q_1		q_1, *e*	q_1, *e*	q_2, *e*	q_3, *e*	
q_2				q_2, *e*	q_3, *e*	
q_3						*HALT*

Как уже отмечалось выше, состояния, подобные q_3, на графе переходов вводятся лишь затем, чтобы отразить тот факт, что успешное завершение разбора возможно не при любом состоянии стека. Например, в данном случае оно возможно лишь в том случае, если стек пуст. Но в таблице переходов от этих состояний можно избавляться (табл. 3.12).

Табл. 3.12 — Минимизированная таблица переходов ДМПА, описывающего идентификатор в скобках

	(, *e*	_a-zA-Z, *e*	0-9, *e*	), (	⊥, ∅
q_0	q_0, (	q_1, *e*			
q_1		q_1, *e*	q_1, *e*	q_2, *e*	*HALT*
q_2				q_2, *e*	*HALT*

Очевидно, что в данном случае в начале разбора стек должен быть пустым. Таким образом, получили ДМПА $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, где:

- $Q = \{q_0, q_1, q_2\}$ — множество состояний;
- Σ = {(,), _, a-z, A-Z, 0-9} — алфавит;
- Γ = {(— алфавит магазинных символов;
- $\delta(Q \times (\Sigma \cup \{e\} \cup \{\perp\}) \times \Gamma)$ = {{(q_0, «(»), (q_1, *e*), *ERROR*, *ERROR*, *ERROR*}, {*ERROR*, (q_1, *e*), (q_1, *e*), (q_2, *e*), *HALT*}, {*ERROR*, *ERROR*, *ERROR*, (q_2, *e*), *HALT*}} — функция переходов;
- $Z_0 = e$ — начальное состояние стека;
- $F = \{q_1, q_2\}$ — множество конечных состояний.

Примечание. При программной реализации ДМПА для обозначения специальных символов (пустой цепочки *e*, пустого множества ∅, маркера конца цепочки ⊥) используются символы того же алфавита, на котором пишется программа (обычно это

кодировка ASCII или Unicode). Например, часто пустую цепочку обозначают буквой английского алфавита «e», пустое множество — цифрой «0», а маркер конца цепочки — знаком «#». Здесь важно, чтобы эти обозначения пересекались с алфавитом автомата Σ. Так, в данном примере буква «e» входит в алфавит, поэтому для обозначения пустой цепочки в программной реализации автомата следует использовать какой-то другой символ.

Пример разбора цепочки «((a123))»:

$(q_0, «((a123))\perp», e) \vdash^1 (q_0, «(a123))\perp», «(»)$

$\vdash^2 (q_0, «a123))\perp», «((»)$

$\vdash^3 (q_1, «123))\perp», «((»)$

$\vdash^4 (q_1, «23))\perp», «((»)$

$\vdash^5 (q_1, «3))\perp», «((»)$

$\vdash^6 (q_1, «))\perp», «((»)$

$\vdash^7 (q_2, «)\perp», «(»)$

$\vdash^8 (q_2, «\perp», e)$

$\vdash^9 HALT$

Разбор завершен успешно. Пример разбора неправильной цепочки «(x))»:

$(q_0, «(x))\perp», e) \vdash^1 (q_0, «x))\perp», «(»)$

$\vdash^2 (q_1, «))\perp», «(»)$

$\vdash^3 (q_1, «)\perp», e)$

$\vdash^4 ERROR$

Имеем ошибку в позиции 4, т.к. в таблице нет столбца с комбинацией из символа закрывающей скобки и пустого стека или игнорирующей состояние стека — («)», ∅) или («)», *e*). Пример разбора неправильной цепочки «()»:

$(q_0, «()\perp», e) \vdash^1 (q_0, «)\perp», «(»)$

$\vdash^2 ERROR$

Имеем $\delta(q_0, «)», «(») = ERROR$, поэтому во входной цепочке синтаксическая ошибка в позиции 2.

Добавив в данный автомат действия, можно определить дополнительные ограничения на синтаксис. Например, ограничить максимальную вложенность скобок или длину идентификатора.

3.3.5 Разбор цепочек по лексемам

В тех случаях, когда входная цепочка содержит некоторое ограниченное множество ключевых слов, посимвольный разбор может привести к существенному увеличению размеров таблицы переходов.

Пример 3. Пусть язык L описывает вложенные операторы языка Pascal «**`begin end;`**». Учитывая, что необходимо проверять их парность, используем ДМПА с посимвольным разбором (табл. 3.13). Операторы отделяются друг от друга разделительными символами (пробелами, табуляциями, знаками возврата каретки и перехода на новую строку) в произвольном количестве, но не менее одного (в таблице обозначены символом подчеркивания). Также пробелы могут окружать знак «`;`». При считывании оператора **`begin`** на стек помещается символ «b» (можно выбрать любой другой символ). При считывании оператора **`end`** проверяется наличие на стеке символа «b». Если он там есть, то снимаем его со стека, в противном случае получаем ошибку непарного оператора **`end`**.

Табл. 3.13 — Таблица переходов ДМПА для операторов «**begin end;**» с посимвольным разбором

	`b`, e	**`e`**, b	**`g`**, e	**`i`**, e	**`n`**, e	**`d`**, e	«`;`», e	`_`, e	$\perp$, $\varnothing$
q_0	q_1, b	q_6, e						q_0, e	*HALT*
q_1		q_2, b							
q_2			q_3, e						
q_3				q_4, e					
q_4					q_5, e				
q_5								q_0, e	
q_6					q_7, e				
q_7						q_8, e			
q_8							q_0, e	q_8, e	

Данную таблицу можно уменьшить в размерах, если составить алфавит автомата не из отдельных символов, а из ключевых слов. То есть вместо алфавита Σ = {**b**, **e**, **g**, **i**, **n**, **d**, «;», _} получим алфавит Σ' = {**begin**, **end**, «;»} (табл. 3.14).

Табл. 3.14 — Таблица переходов ДМПА для операторов «**begin end;**» с разбором по лексемам

	begin, *e*	**end**, *b*	«;», *e*	⊥, ∅
q_0	q_0, *b*	q_1, *e*		*HALT*
q_1			q_0, *e*	

Перед разбором по лексемам необходимо входную цепочку разбить на поток лексем. Причем вместе с каждой лексемой необходимо хранить ее позицию в исходной цепочке, чтобы иметь возможность указать позицию ошибки. Позиция — это либо номер первого символа лексемы во входной цепочке, либо, если входная цепочка может состоять из нескольких строк (т.е. содержит символы возврата каретки и перехода на новую строку), — номер строки и позиция в строке первого символа лексемы. Тогда в алгоритме разбора величина *k* будет являться номером лексемы, а не номером символа во входной цепочке.

Чтобы разбить входную цепочку на лексемы, алфавит языка делится на три подмножества:

1. Подмножество символов-разделителей (или пробельных символов) $\Sigma_S \subset \Sigma$. Обычно это символы пробела, табуляции, перехода на следующую строку и возврата каретки (см. раздел 3.1). Эти символы отделяют лексемы друг от друга и в алфавит Σ' не включаются.
2. Подмножество символов, являющихся знаками пунктуации $\Sigma_P \subset \Sigma$. Обычно это точки, запятые, точки с запятой, скобки и т.п. Эти символы также отделяют лексемы друг от друга, но и сами, в свою очередь, являются лексемами.
3. Подмножество лексемных символов (т.е. символов, из которых составляются лексемы) $\Sigma_L \subset \Sigma$. Это все остальные символы алфавита Σ: $\Sigma_L = \Sigma — (\Sigma_S \cup \Sigma_P)$.

Алфавит автомата с разбором по лексемам Σ' включает в себя знаки пунктуации Σ_P и непустые цепочки, составленные из символов алфавита Σ_L:

$$\Sigma' \subseteq \Sigma_P \cup \Sigma_L^+.$$

То есть лексема — это либо знак пунктуации из алфавита Σ_P, либо последовательность символов алфавита Σ_L, отделенная от других лексем символами алфавитов Σ_S и Σ_P. Например:

```
begin
  begin end ;
end;
begin
end;
```

В данном случае получим следующий поток лексем:

1. **begin** (строка 1, позиция 1);
2. **begin** (строка 2, позиция 3);
3. **end** (строка 2, позиция 9);
4. ; (строка 2, позиция 13);
5. **end** (строка 3, позиция 1);
6. ; (строка 3, позиция 4);
7. **begin** (строка 4, позиция 1);
8. **end** (строка 5, позиция 1);
9. ; (строка 5, позиция 4).

Процесс разбора будет следующим (для краткости лексема **begin** обозначена буквой «b», а лексема **end** — буквой «e»):

$$\begin{aligned}
(q_0, \text{«bbe;e;be;}\perp\text{»}, e) &\vdash^1 (q_0, \text{«be;e;be;}\perp\text{»}, b) \\
&\vdash^2 (q_0, \text{«e;e;be;}\perp\text{»}, bb) \\
&\vdash^3 (q_1, \text{«;e;be;}\perp\text{»}, b) \\
&\vdash^4 (q_0, \text{«e;be;}\perp\text{»}, b) \\
&\vdash^5 (q_1, \text{«;be;}\perp\text{»}, e) \\
&\vdash^6 (q_0, \text{«be;}\perp\text{»}, e) \\
&\vdash^7 (q_0, \text{«e;}\perp\text{»}, b) \\
&\vdash^8 (q_1, \text{«;}\perp\text{»}, e) \\
&\vdash^9 (q_0, \text{«}\perp\text{»}, e) \\
&\vdash^{10} HALT
\end{aligned}$$

Разбор завершен успешно. Другой пример:

```
begin end;
end;
```

Имеем поток лексем:

1. **`begin`** (строка 1, позиция 1);
2. **`end`** (строка 1, позиция 7);
3. `;` (строка 1, позиция 10);
4. **`end`** (строка 2, позиция 1);
5. `;` (строка 2, позиция 4);

Процесс разбора:

$$(q_0, \text{«be;e;}\perp\text{»}, e) \vdash^1 (q_0, \text{«e;e;}\perp\text{»}, b) \vdash^2 (q_1, \text{«;e;}\perp\text{»}, e) \vdash^3 (q_0, \text{«e;}\perp\text{»}, e) \vdash^4 \mathit{ERROR}$$

Получили ошибку, т.к. в состоянии q_0 лексема **`end`** допустима лишь при наличии символа «b» на стеке. Ошибка в лексеме № 4, поэтому выдаем сообщение об ошибке в строке 2, позиции 1.

В принципе, эту же задачу можно решить и посимвольным разбором с включением действий в синтаксис (табл. 3.15).

Табл. 3.15 — Таблица переходов ДМПА для операторов «**begin end;**» с включением действий в синтаксис

	{**b**, **e**, **g**, **i**, **n**, **d**}, e	«;», e	_, e	⊥, ∅
q_0	q_1, e, $\langle A_1 \rangle$	q_0, e, $\langle A_3 \rangle$	q_0, e, ∅	*HALT*
q_1	q_1, e, $\langle A_2 \rangle$	q_0, e, $\langle A_4 \rangle$	q_0, e, $\langle A_5 \rangle$	

Для выполнения действий нам потребуется буфер-строка buf. Перед началом разбора инициализируем его пустой строкой (buf := ''). Семантика действий:

- $\langle A_1 \rangle$ — если буфер пуст (buf ≡ ''), инициализируем его текущим символом входной цепочки (buf := a_k), в противном случае переводим автомат в состояние *ERROR*.
- $\langle A_2 \rangle$ — добавление в буфер текущего символа входной цепочки (buf := buf + a_k).
- $\langle A_3 \rangle$ — если в буфере находится ключевое слово **`end`** (buf ≡ 'end'), то очистить буфер (buf := ''), иначе переводим автомат в состояние *ERROR*.

– $\langle A_4 \rangle$ — если в буфере находится ключевое слово **`end`** (buf ≡ 'end'), то выполняем сначала действие $\langle A_5 \rangle$, затем $\langle A_3 \rangle$, иначе переводим автомат в состояние *ERROR*.
– $\langle A_5 \rangle$: если…

  а) в буфере находится ключевое слово **`begin`** (buf ≡ 'begin'), то поместить в стек символ «*b*» ($M \leftarrow b$) и очистить буфер (buf := '');

  б) в буфере находится ключевое слово **`end`** (buf ≡ 'end') и на вершине стека находится символ «*b*», извлекаем его со стека ($M \rightarrow b$), иначе переводим автомат в состояние *ERROR*.

Как видно, все три способа имеют определенные недостатки. В первом случае получаем увеличенный размер таблицы переходов. Во втором случае требуется организация разбора по лексемам, в третьем случае — программирование действий.

В некоторых случаях разбор может быть смешанным. Например, если рассматривать описание переменных на каком-либо языке, то оно включает как ограниченное множество ключевых слов (типов данных и т.п.), так и неограниченное множество идентификаторов. Поэтому автомат должен работать либо посимвольно, либо в смешанном режиме (т.е. ключевые слова анализировать как лексемы, а идентификаторы — посимвольно).

3.4 Работа с регулярными выражениями

3.4.1 Основные определения

Определение. Регулярные выражения в алфавите Σ и регулярные множества, которые они обозначают, определяются рекурсивно следующим образом [1]:

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 PQ;

в) p^* — регулярное выражение, обозначающее P^*;

5) ничто другое не является регулярным выражением.

Расстановка приоритетов:

– * (итерация) — наивысший приоритет;
– конкатенация;
– + (объединение).

Таким образом, $0 + 10^* = (0 + (1\ (0^*)))$. Примеры:

1. 01 означает {01};
2. 0^* — $\{0^*\}$;
3. $(0+1)^*$ — $\{0, 1\}^*$;
4. $(0+1)^*$ 011 — означает множество всех цепочек, составленных из 0 и 1 и оканчивающихся цепочкой 011;
5. $(a+b)\ (a+b+0+1)^*$ означает множество всех цепочек $\{0, 1, a, b\}^*$, начинающихся с a или b.
6. $(00+11)^*\ ((01+10)(00+11)^*\ (01+10)(00+11)^*)$ означает множество всех цепочек нулей и единиц, содержащих четное число 0 и четное число 1.

Таким образом, для каждого регулярного множества можно найти регулярное выражение, ему соответствующее, и наоборот.

Регулярные выражения (РВ) — такой же способ задания языков, как и конечные автоматы или КС-грамматики. Если быть более точными, то регулярные множества (РМ) — это языки, порождаемые праволинейными грамматиками [1]. Регулярные выражения порождают некоторое множество допустимых в данном языке L цепочек символов, которое собственно и является регулярным множеством.

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) $\alpha+\varnothing = \alpha$

Чтобы проверить, что РВ совпадают, достаточно построить соответствующие им РМ и убедиться, что они содержат одинаковые элементы. Для этого рассмотрим, к чему приводит применение трех рассмотренных выше операций.

Использование операции объединения двух РВ объединяет РМ, соответствующие этим РВ. Например, если даны два РВ

$$x = a+b,\ y = c+d,$$

то им соответствуют РМ

$$x \Leftrightarrow X = \{a, b\},\ y \Leftrightarrow Y = \{c, d\}.$$

После объединения имеем

$$x + y \Leftrightarrow X \cup Y = \{a, b, c, d\}.$$

Операция конкатенации приводит к результату, примерно аналогичному декартовому произведению РМ соответствующих РВ, только в этом множестве будут не пары элементов, а их конкатенация:

$$xy \Leftrightarrow XY = \{ac, ad, bc, bd\}.$$

Так, регулярное выражение «к(и+о)т» описывает язык, состоящий из двух слов — кит и кот:

$$\text{к(и+о)т} \Leftrightarrow \{\text{к}\}\{\text{и, о}\}\{\text{т}\} = \{\text{кит, кот}\}.$$

Или, используя леммы № 5 и № 6,

$$\text{к(и+о)т} = \text{кит} + \text{кот} \Leftrightarrow \{\text{кит, кот}\}.$$

Итерация РВ соответствует итерации элементов РМ:

$$x = a,$$

$$x^* \Leftrightarrow X^* = \{e, a, aa, aaa, \ldots\}.$$

То есть $x^* = e + x + xx + xxx + \ldots$, и так до бесконечности. Рассмотрим, к чему приведет итерация конкатенации и объединения:

$$(xy)^* = e + xy + xyxy + xyxyxy + \ldots$$

$$(x + y)^* = e + (x + y) + (x + y)(x + y) + (x + y)(x + y)(x + y) + \ldots =$$
$$= e + x + xx + xy + yx + yy + xxx + \ldots$$

Соответственно,

$0 + 1(0+1)^* \Leftrightarrow \{0\} \cup (\{1\} \cup \{0, 1\}^*) = \{0, 1, 10, 11, 100, 101, 110, 111\ldots\}$.

Видно, что данное РВ описывает все двоичные числа, у которых отсутствуют незначащие ведущие нули (т.е. нет чисел типа 01, 001 и т.п.).

Важно также учитывать приоритет операций и их свойства. Так, операция объединения является коммутативной, т.е. $x + y = y + x$, а операция конкатенации — нет ($xy \neq yx$). Самый низкий приоритет у объединения:

$$x + yz \Leftrightarrow \{x, yz\},$$
$$(x + y)z \Leftrightarrow \{xz, yz\},$$
$$x + y^* \Leftrightarrow \{e, x, y, yy, yyy, yyyy, \ldots\},$$
$$(x + y)^* \Leftrightarrow \{e, x, y, xx, xy, yx, yy, xxx, \ldots\}.$$

Самый высокий приоритет — у итерации:

$$(xy)^* \Leftrightarrow \{e, xy, xyxy, \ldots\},$$
$$xy^* \Leftrightarrow \{x, xy, xyy, xyyy, \ldots\}.$$

Исходя из этого приведенные выше леммы легко доказываются. При желании можно составить и доказать множество других лемм, например,

- $a^* + e = a^*$;
- $(a + e)^* = a^*$;
- $a^*a^* = a^*$;
- $e^* = e$;
- и т.д.

При описании языков 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.

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 члена, 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 стандартной системы уравнений с регулярными коэффициентами:

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.

Однако следует отметить, что не все уравнения с регулярными коэффициентами обладают единственным решением. Например, если

$$X = \alpha X + \beta$$

является уравнением с регулярными коэффициентами и α означает множество, содержащее пустую цепочку, то $X = \alpha^*(\beta + \gamma)$ будет решением этого уравнения для любого γ. Таким образом, уравнение имеет бесконечно много решений. В ситуациях такого рода мы будем брать наименьшее решение, которое назовем *наименьшей неподвижной точкой*. В нашем случае наименьшая неподвижная точка — $\alpha^*\beta$.

Пример. Рассмотрим пример решения системы с тремя неизвестными ($N = 3$). Пусть

$$x_1 = q + ex_1 + ex_2 + \varnothing x_3,$$

$$x_2 = e + px_1 + rx_2 + ex_3,$$

$$x_3 = \varnothing + \varnothing x_1 + ex_2 + qx_3.$$

Применим описанный выше алгоритм, применяя леммы для сокращения полученных выражений.

Шаг 1. $i = 1$.

Шаг 2. $i \neq N$, поэтому переписываем уравнение для x_1 в требуемом виде:

$$x_1 = \alpha x_1 + \beta = ex_1 + \left(q + ex_2 + \varnothing x_3\right).$$

Далее в уравнения для x_2 и x_3 подставляем $\alpha^*\beta$ выражение вместо x_1:

$$x_2 = e + pe^*\left(q + ex_2 + \varnothing x_3\right) + rx_2 + ex_3 =$$

$$= e + pe^*q + \left(pe^*e + r\right)x_2 + \left(pe^*\varnothing + e\right)x_3 =$$

$$= pq + \left(p + r\right)x_2 + ex_3,$$

$$x_3 = \varnothing + \varnothing e^*\left(q + ex_2 + \varnothing x_3\right) + ex_2 + qx_3 =$$

$$= \varnothing + \varnothing e^*q + \left(\varnothing e^*e + e\right)x_2 + \left(\varnothing e^*\varnothing + q\right)x_3 =$$

$$= \varnothing + ex_2 + qx_3.$$

Здесь и далее применены леммы $a\varnothing = \varnothing$, $a+\varnothing = a$, $ae = a$, а также $e^* = e$.

Шаг 3. $i = 2$, возврат на шаг 2.

Шаг 2. $i \neq N$, поэтому переписываем уравнение для x_2 в требуемом виде:

$$x_2 = \alpha x_2 + \beta = (p + r)x_2 + (pq + ex_3).$$

Далее в уравнение для x_3 подставляем $\alpha^*\beta$ выражение вместо x_2:

$$x_3 = \varnothing + ex_2 + qx_3 = \varnothing + e(p + r)^*(pq + ex_3) + qe_3 =$$

$$= (p + r)^* pq + \left((p + r)^* + q\right)x_3.$$

Шаг 3. $i = 3$, возврат на шаг 2.

Шаг 2. $i = N$, поэтому переходим на шаг 4.

Шаг 4. Запишем уравнение для x_3 в требуемом виде:

$$x_3 = \alpha x_3 + \beta = \left((p + r)^* + q\right)x_3 + \left((p + r)^* pq\right).$$

Переходим к шагу 5.

Шаг 5. Запишем решение для x_3:

$$x_3 = \alpha^*\beta = \left((p + r)^* + q\right)^*(p + r)^* pq.$$

Далее в уравнения для x_1 и x_2 подставляем полученное выражение вместо x_3:

$$x_1 = q + ex_1 + ex_2 + \varnothing x_3 = q + ex_1 + ex_2 + \varnothing\left((p + r)^* + q\right)^*\left((p + r)^* pq\right) =$$

$$= q + ex_1 + ex_2,$$

$$x_2 = pq + (p + r)x_2 + ex_3 = pq + (p + r)x_2 + \left((p + r)^* + q\right)^*(p + r)^* pq.$$

Шаг 6. $i \neq 1$, поэтому полагаем $i = 2$ и переходим на шаг 5.

Шаг 5. Имеем:

$$x_2 = (p + r)x_2 + \left(pq + \left((p + r)^* + q\right)^*(p + r)^* pq\right).$$

Запишем решение:

$$x_2 = \alpha^*\beta = (p + r)^*\left(pq + \left((p + r)^* + q\right)^*(p + r)^* pq\right).$$

Далее в уравнение для x_1 подставляем полученное выражение вместо x_2:

$$x_1 = q + ex_1 + (p + r)^*\left(pq + \left((p + r)^* + q\right)^*(p + r)^* pq\right).$$

Следовательно,

$$x_1 = e^* \left((p+r)^* \left(pq + \left((p+r)^* + q \right)^* (p+r)^* pq \right) + q \right) =$$

$$= (p+r)^* \left(pq + \left((p+r)^* + q \right)^* (p+r)^* pq \right) + q.$$

Шаг 6. $i = 1$, решение окончено. Результат:

$$x_1 = (p+r)^* \left(pq + \left((p+r)^* + q \right)^* (p+r)^* pq \right) + q,$$

$$x_2 = (p+r)^* \left(pq + \left((p+r)^* + q \right)^* (p+r)^* pq \right),$$

$$x_3 = \left((p+r)^* + q \right)^* (p+r)^* pq.$$

3.4.2 Применение регулярных выражений

В принципе, РВ используются во многих областях [2]:

- для описания языков в различных трансляторах и интерпретаторах;
- как основа для систем искусственного интеллекта (в частности баз знаний и экспертных систем), хотя чаще в этой области применяются контекстно-зависимые грамматики;
- в поисковых системах (типа Google и Yandex) при составлении запросов;
- при поиске и замене текста в различных программных продуктах, например, в диалоге «Поиск и замена» Visual Studio, в аналогичном диалоге программы Microsoft Word и т.д.;
- в программировании, и особенно web-программировании (при поиске, замене, обработке текста и т.п.)
- и т.д.

Составление РВ является такой же слабо формализованной задачей, как и составление функции переходов ДКА или ДМПА. Однако если имеется функция переходов δ ДКА $M = (Q, \Sigma, \delta, q_0, F)$, можно построить соответствующее ему регулярное выражение. Для этого составляем систему уравнений с регулярными ко-

эффициентами, где неизвестными будут состояния ДКА ($\Delta = Q$), а после решения искомое РВ будет являться значением $X_1 = q_0$. Система уравнений составляется следующим образом:

1) полагаем все коэффициенты $\alpha_{ij} := \varnothing$;
2) если имеется функция переходов $\delta(X_i, a) = X_j$, $a \in \Sigma$, то коэффициент α_{ij} объединить с a: $\alpha_{ij} := \alpha_{ij} + a$;
3) если состояние X_i является конечным ($X_i \in F$ или $\delta(X_i, \perp) = HALT$), то $\alpha_{i0} := e$.

Для примера рассмотрим функцию переходов ДКА, описывающего число с фиксированной точкой (табл. 3.9). Используя приведенный выше алгоритм, получим следующую систему уравнений с регулярными коэффициентами для неизвестных $\Delta = \{q_0, q_1, q_2, q_3, q_4\}$:

$$q_0 = \varnothing + \varnothing q_0 + sq_1 + pq_2 + dq_3 + \varnothing q_4;$$
$$q_1 = \varnothing + \varnothing q_0 + \varnothing q_1 + pq_2 + dq_3 + \varnothing q_4;$$
$$q_2 = \varnothing + \varnothing q_0 + \varnothing q_1 + \varnothing q_2 + \varnothing q_3 + dq_4;$$
$$q_3 = e + \varnothing q_0 + \varnothing q_1 + \varnothing q_2 + dq_3 + pq_4;$$
$$q_4 = e + \varnothing q_0 + \varnothing q_1 + \varnothing q_2 + \varnothing q_3 + dq_4.$$

Здесь для краткости приняты следующие обозначения:

– s — знак числа, s = '+' + '–';
– p — десятичная точка, p = '.';
– d — цифры, d = '0' + '1' + … + '9'.

Решаем систему:

$$q_0 = \varnothing^*(sq_1 + pq_2 + dq_3 + \varnothing q_4 + \varnothing) = sq_1 + pq_2 + dq_3 \Rightarrow$$
$$q_1 = \varnothing + \varnothing q_0 + \varnothing q_1 + pq_2 + dq_3 + \varnothing q_4 = pq_2 + dq_3,$$
$$q_2 = \varnothing + \varnothing q_0 + \varnothing q_1 + \varnothing q_2 + \varnothing q_3 + dq_4 = dq_4,$$
$$q_3 = e + \varnothing q_0 + \varnothing q_1 + \varnothing q_2 + dq_3 + pq_4 = dq_3 + pq_4 + e,$$
$$q_4 = e + \varnothing q_0 + \varnothing q_1 + \varnothing q_2 + \varnothing q_3 + dq_4 = dq_4 + e.$$

Из третьего уравнения:

$$q_3 = dq_3 + pq_4 + e = d^*(pq_4 + e).$$

Из четвертого уравнения:

$$q_4 = dq_4 + e = d^*e = d^*.$$

Обратный ход:

$$q_3 = d^*(pq_4 + e) = d^*(pd^* + e),$$
$$q_2 = dq_4 = dd^*,$$
$$q_1 = pq_2 + dq_3 = pdd^* + dd^*(pd^* + e),$$
$$q_0 = sq_1 + pq_2 + dq_3 = s(pdd^* + dd^*(pd^* + e)) + pdd^* + dd^*(pd^* + e).$$

Таким образом, данному ДКА соответствует РВ

$$s(pdd^* + dd^*(pd^* + e)) + pdd^* + dd^*(pd^* + e).$$

Используя леммы, это выражение можно сделать короче:

$$s(pdd^* + dd^*(pd^* + e)) + pdd^* + dd^*(pd^* + e) =$$
$$= spdd^* + sdd^*(pd^* + e) + pdd^* + dd^*(pd^* + e) =$$
$$= (s + e)(pdd^* + dd^*(pd^* + e)).$$

Для более короткой записи можно использовать положительную итерацию $aa^* = a^*a = a^+$:

$$(s + e)(pdd^* + dd^*(pd^* + e)) = (s + e)(pd^+ + d^+(pd^* + e)) =$$
$$= (s + e)(pd^+ + d^+pd^* + d^+).$$

То есть запись «d^+» описывает последовательность из одной и более цифр, а запись «d^*» — последовательность из нуля и более цифр. В этом выражении цепочка «$(s + e)$» отражает тот факт, что знак «+» или «–» у числа может как присутствовать, так и отсутствовать. Далее перечисляются варианты мантиссы. Выражение «pd^+» описывает мантиссу вида «.*M*», выражение «d^+» — мантиссу вида «*N*», а выражение «d^+pd^*» — мантиссы вида «*N*.*M*» и «*N*.».

В принципе, подобное выражение можно составить по графу или таблице переходов ДКА самостоятельно, принимая во внимание, что переход из одного состояния в несколько других — это объединение, последовательный переход из одного состояния в другое — конкатенация, а цикл в таблице или графе переходов — это итерация. Состояние будет конечным, если на нем РВ может заканчиваться (рис. 3.12).

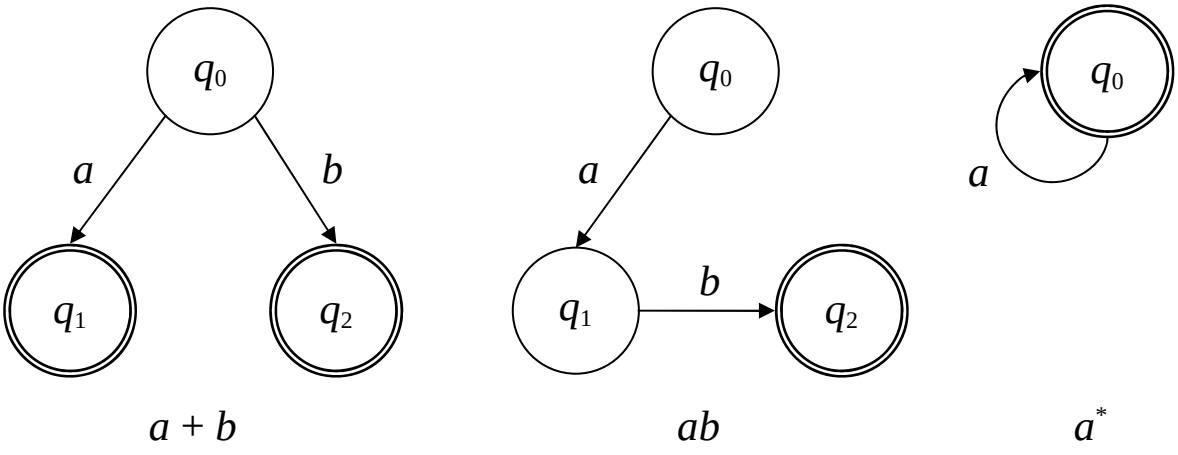

Рис. 3.12 — Сопоставление графа переходов ДКА и базовых операций РВ

На основе этих операций можно строить и более сложные выражения (рис. 3.13). Однако если граф переходов имеет сложную структуру, лучше воспользоваться описанным формальным методом.

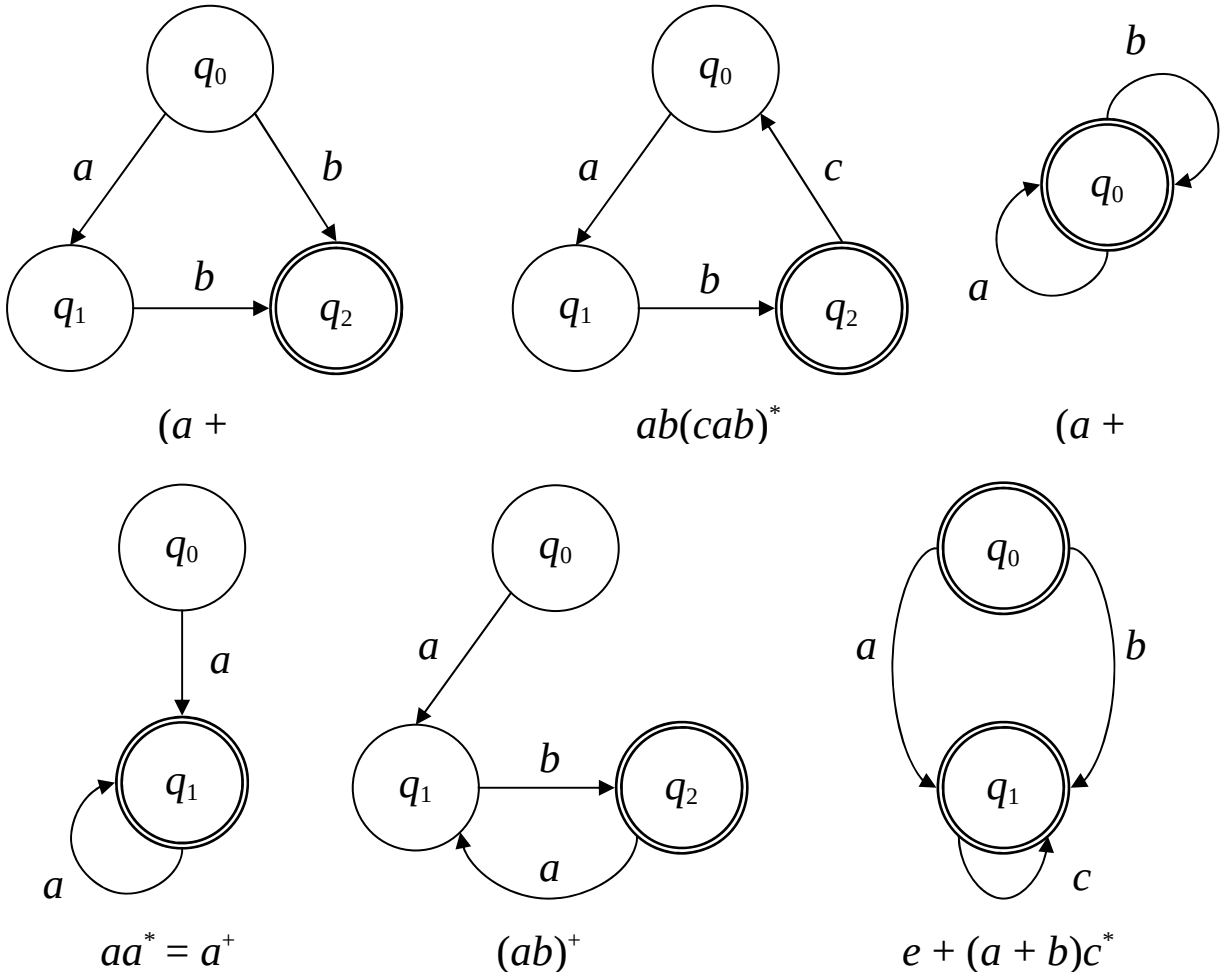

Рис. 3.13 — Сопоставление графа переходов ДКА и РВ

Подобным образом можно совершить обратный переход — от РВ к функции переходов ДКА, заменяя объединения параллельными переходами из одного состояния в другие, конкатенации — последовательными переходами между состояниями и итерации — циклами. Например, по выражению «$(s + e)(pd^+ + d^+(pd^* + e))$» можно построить граф, изображенный на рис. 3.14.

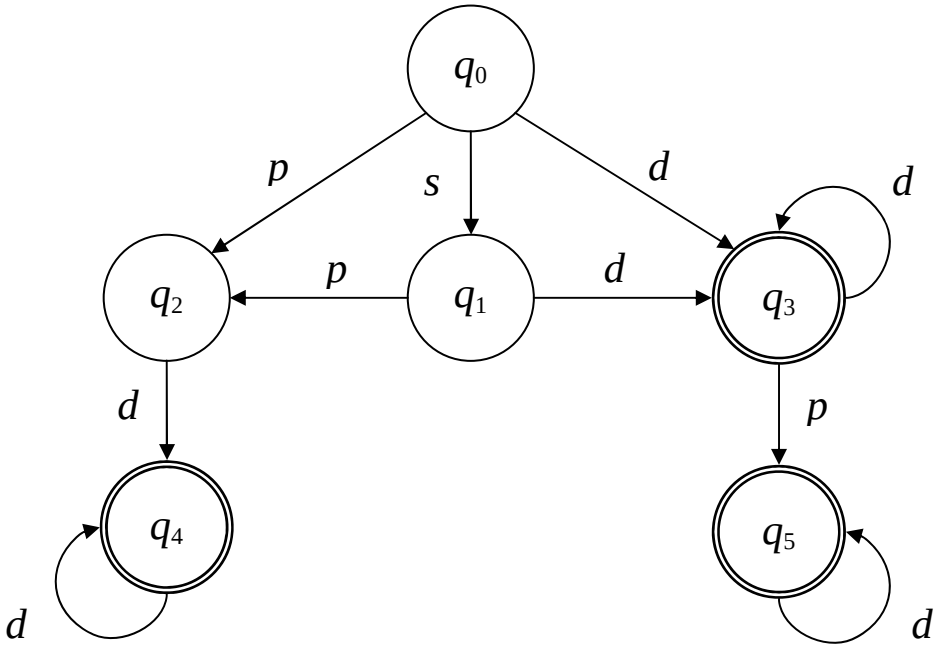

Рис. 3.14 — Граф переходов ДКА, построенный по РВ

Далее его можно минимизировать, что даст ДКА, эквивалентный построенному ранее автомату.

3.4.3 Программирование регулярных выражений

Все современные языки программирования позволяют работать с регулярными выражениями. При этом часть языков (PHP, JavaScript и т.д.) имеют встроенные средства для работы с РВ, а в других есть готовые библиотеки или классы для работы с ними, например класс Regex для платформы .NET [3, 4].

Программные реализации РВ для различных языков программирования и сред разработки имеют гораздо больше операций, чем мы рассмотрели выше, но они лишь упрощают работу с РВ. Данные три операции являются базовыми, и все остальные операции могут быть выражены через них. Например, в классе Regex запись «*x*?» означает, что элемент *x* не является обязательным, т.е. в цепочке данный элемент может присутствовать либо отсутствовать. Очевидно, что

$$x? = x + e.$$

Или, например, запись «*x*{1,3}» означает повторение элемента *x* от 1 до 3 раз, т.е.

$$x\{1,3\} = x + xx + xxx.$$

Далее рассмотрим, какие возможности предоставляет класс Regex для описания РВ (табл. 3.16). Многие из этих конструкций являются универсальными, т.е. поддерживаются другими реализациями РВ.

Табл. 3.16 — Конструкции класса Regex для составления РВ

Символ	Интерпретация
Escape-последовательности	
\b	При использовании его в квадратных скобках соответствует символу «←» (\u0008)
\t, \r, \n, \a, \f, \v	Табуляция (\u0009), возврат каретки (\u000D), новая строка (\u000A) и т.д.
\cX	Управляющий символ (например, \cC — это Ctrl+C, \u0003)
\e	Escape (\u001B)
\ooo	Символ ASCII в восьмеричной системе
\xhh	Символ ASCII в шестнадцатеричной системе
\uhhhh	Символ Unicode
\	Следующий символ не является специальным символом РВ. Этим символом нужно экранировать все специальные символы
Пример (в примере приведен шаблон и строка поиска, в строке найденные совпадения подчеркнуты): @"\r\n\w+" — "\r\nЗдесь имеются\nдве строки".	
Подмножества символов	
.	Любой символ, кроме конца строки (\n)
[xxx]	Любой символ из множества
[^xxx]	Любой символ, кроме символов из множества
[x-x]	Любой символ из диапазона
[xxx-[xxx]]	Вычитание одного множества или диапазона из другого
\p{name}	Любой символ, заданный категорией Unicode с именем name
\P{name}	Любой символ, кроме заданных категорией Unicode с именем name
\w	Множество символов, используемых при задании идентификаторов
\W	Множество символов, не используемых при задании идентификаторов
\s	Пробелы
\S	Все, кроме пробелов

Продолжение табл. 3.16

Символ	Интерпретация
\d	Цифры
\D	Не цифры

Примеры:
@".+" – "\r\nЗдесь имеются\nдве строки";
@"[fx]+" – "0xabcfx";
@"[^fx]+" – "0xabcfx";
@"[a-f]+" – "0xabcfx";
@"[^a-f]+" – "0xabcfx";
@"[a-z-[c]]+" – "0xabcfx";
@"\p{Lu}" – "City Lights"; // Lu – прописные буквы
@"\P{Lu}" – "City";
@"\p{IsCyrillic}" – "xaOS"; // IsCyrillic – русские буквы
@"\P{IsCyrillic}" – "xaOS".	

Привязка	
^, \A	В начале строки
$, \Z	В конце строки или до символа «\n» в конце строки
\z	В конце строки
\G	В том месте, где заканчивается предыдущее соответствие
\b	Граница слова
\B	Любая позиция не на границе слова

Примеры:
@"\G\(\d\)" – "(1)(3)(5)[7](9) "; // три соответствия (1), (2) и (3)
@"\bn\S*ion\b" – "nation donation";
@"\Bend\w*\b" – "end sends endure lender".	

Операции (кванторы)	
*, *?	Итерация
+, +?	Положительная итерация
?, ??	Ноль или одно соответствие
{n}, {n}?	Точно *n* соответствий
{n,}, {n,}?	По меньшей мере, *n* соответствий
{n,m},{n,m}?	От *n* до *m* соответствий

Примеры (первые кванторы — *жадные*, ищут как можно большее число элементов, вторые — *ленивые*, ищут как можно меньшее число элементов):
@"\d{3,}" – "888-555-5555";
@"^\d{3}" – "913-913-913";
@"-\d{3}$" – "913-913-913";
@"5+?5" – "888-555-5555"; // три совпадения – 55, 55 и 55
@"5+5" – "888-555-5555".	

Продолжение табл. 3.16

Символ	Интерпретация
Группирование	
()	Группа, автоматически получающая номер
(?:)	Не сохранять группу
(?<имя>) или (?'имя')	При обнаружении соответствия создается именованная группа
(?<имя–имя>) или (?'имя– имя')	Удаление ранее определенной группы и сохранение в новой группе подстроки между ранее определенной группой и новой группой
(?imnsx:) (?–imnsx:)	Включает или выключает в группе любую из пяти возможных опций: i — нечувствительность к регистру; s — одна строка (тогда «.» — это любой символ); m — многострочный режим («^», «$» — начало и конец каждой строки); n — не захватывать неименованные группы; x — исключить не преобразованные в escape-последовательность пробелы из шаблона и включить комментарии после знака номера (#)
(?=)	Положительное утверждение просмотра вперед нулевой длины
(?!)	Отрицательное утверждение просмотра вперед нулевой длины
(?<=)	Положительное утверждение просмотра назад нулевой длины
(?<!)	Отрицательное утверждение просмотра назад нулевой длины
(?>)	Невозвращаемая (жадная) часть выражения

Примеры:
@"(an)+" – "bananas annals";
@"an+" – "bananas annals"; // сравните, три совпадения – an, an и ann
@"(?i:an)+" – "baNAnas annals";
@"[a-z]+(?=\d)" – "abc xyz12 555w";
@"(?<=\d)[a-z]+" – "abc xyz12 555w".	

Ссылки	
\число	Ссылка на группу
\k<имя>	Ссылка на именованную группу
Примеры: @"(\w)\1" – "deep"; @"(?<char>\w)\k<char>" – "deep".	

Окончание табл. 3.16

Символ	Интерпретация
Конструкции изменения	
|	Альтернатива (соответствует операции объединения)
(?(выражение)да|нет)	Сопоставляется с частью «да», если выражение соответствует; в противном случае сопоставляется с необязательной частью «нет»
(?(имя)да|нет), (?(число)да|нет)	Сопоставляется с частью «да», если названное имя захвата имеет соответствие; в противном случае сопоставляется с необязательной частью «нет»
Пример: @"th(e|is|at)" – "this is the day";	
Подстановки	
$число	Замещается часть строки, соответствующая группе с указанным номером
${имя}	Замещается часть строки, соответствующая группе с указанным именем
$$	Подставляется $
$&	Замещение копией полного соответствия
$`	Замещение текста входной строки до соответствия
$'	Замещение текста входной строки после соответствия
$+	Замещение последней захваченной группы
$_	Замещение всей строки
Комментарии	
(?#)	Встроенный комментарий
#	Комментарий до конца строки

При поиске по шаблону во входной цепочке в классе Regex вводятся понятия *совпадения* (или *соответствия*), *группы записи* и *захвата*. Захват инкапсулирован в классе Capture. Группы записи — в классе Group, который является потомком класса Capture. Совпадение — в классе Match, который, в свою очередь, является потомком класса Group.

В общем случае, обработка соответствий, групп и захватов проходит по схеме на рис. 3.15.

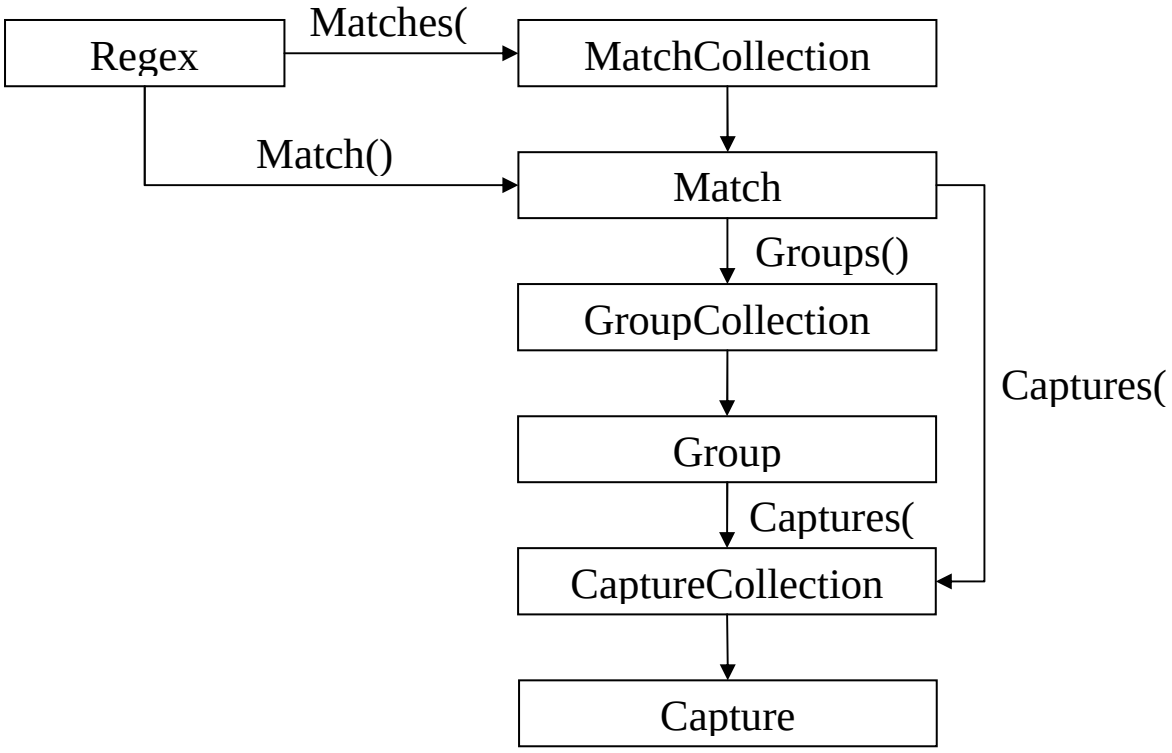

Рис. 3.15 — Схема обработки соответствий при использовании класса Regex

Класс Regex может выдать либо сразу все совпадения в виде коллекции MatchCollection (при вызове метода Matches), либо выдавать их по очереди в виде экземпляров класса Match при вызове метода Match и последующих вызовов метода NextMatch. В свою очередь, совпадение содержит коллекцию групп записи GroupCollection, получаемую при вызове метода Groups. А каждая группа записи содержит коллекцию захватов CaptureCollection, получаемую при вызове метода Captures. В каждой коллекции есть свойство Count, возвращающее количество элементов коллекции, а также индексатор для доступа к отдельным элементам коллекции по индексу. Пример кода на языке C#:

```
Regex r = new Regex(@"((\d)+)+");
Match m = r.Match("123 456");
int matchCount = 0;
while (m.Success)
{
     Console.WriteLine("Соответствие {0}", ++matchCount);
     for (int i = 1; i < m.Groups.Count; i++)
     {
```

```
            Group g = m.Groups[i];
            Console.WriteLine("    Группа {0} = '{1}'", i,
g.Value);
            for (int j = 0; j < g.Captures.Count; j++)
            {
                Capture c = g.Captures[j];
                Console.WriteLine("        Захват {0} = '{1}',
позиция = {2}, длина = {3}", j, c, c.Index, c.Length);
            }
        }
        m = m.NextMatch();
}
```

Необходимо также подключить пространство имен, в котором описан класс Regex (`System.Text.RegularExpressions`). Группа с индексом 0 всегда содержит полный текст всех захватов группы, поэтому в примере просмотр групп начинается с индекса 1. После выполнения данного кода получим следующий результат:

```
Соответствие 1
    Группа 1 = '123'
        Захват 0 = '123', позиция = 0, длина = 3
    Группа 2 = '3'
        Захват 0 = '1', позиция = 0, длина = 1
        Захват 1 = '2', позиция = 1, длина = 1
        Захват 2 = '3', позиция = 2, длина = 1
Соответствие 2
    Группа 1 = '456'
        Захват 0 = '456', позиция = 4, длина = 3
    Группа 2 = '6'
        Захват 0 = '4', позиция = 4, длина = 1
        Захват 1 = '5', позиция = 5, длина = 1
        Захват 2 = '6', позиция = 6, длина = 1
```

Итак, в строке было найдено 2 соответствия из последовательностей одной и более цифр. В регулярном выражении «((\d)+)+» группы две: одна описывает отдельные цифры (\d), другая последовательности цифр ((\d)+). Поэтому в каждом совпадении две группы — одна содержит единственный захват со всей последовательностью цифр, а другая — захваты с отдельными цифрами.

Класс Regex можно использовать и в других языках, поддерживающих библиотеку .NET. Например, если в Visual Studio выбрать создание проекта «Visual C++/CLR/Консольное прило-

жение CLR», то синтаксис на языке C++ CLI (модификация языка C++ для работы с .NET) будет следующим:

```
int main()
{
     Regex ^r = gcnew Regex(L"((\\d)+)+");
     Match ^m = r->Match(L"123 456");
     int matchCount = 0;
     while (m->Success)
     {
          Console::WriteLine(L"Соответствие {0}",
++matchCount);
          for (int i = 1; i < m->Groups->Count; i++)
          {
               Group ^g = m->Groups[i];
               Console::WriteLine(L"    Группа {0} = '{1}'", i,
g->Value);
               for (int j = 0; j < g->Captures->Count; j++)
               {
                    Capture ^c = g->Captures[j];
                    Console::WriteLine(L"        Захват {0} =
'{1}', позиция = {2}, длина = {3}", j, c, c->Index, c-
>Length);
               }
          }
          m = m->NextMatch();
     }
     return 0;
}
```

Перед этим необходимо также подключить пространство имен, в котором описан класс Regex — `System::Text::RegularExpressions`. В других средах и языках синтаксис, соответственно, будет другим. С полным списком свойств и методов рассмотренных классов можно ознакомиться в справочной системе используемой среды разработки. В дальнейшем все примеры кода будут даваться на языке C#.

Рассмотрим наиболее часто используемые члены рассмотренных классов.

1. Конструкторы класса Regex:

```
Regex(string pattern)
Regex(string pattern, RegexOptions options)
```

Здесь «`pattern`» — шаблон РВ, «`options`» — опции (табл. 3.17). Флаги — это опции для отдельных групп, перечисленные в табл. 3.16, соответствующие некоторым опциям РВ.

Табл. 3.17 — Опции конструктора класса Regex

Опции	Флаги	Описание
None		Поведение по умолчанию
IgnoreCase	i	Сопоставление без учета регистра
Multiline	m	Многострочный режим
Singleline	s	Однострочный режим
ExplicitCapture	n	Не захватывать неименованные группы
Compiled		Компилировать РВ в сборку
IgnorePatternWhitespace	x	Исключить не преобразованные в escape-последовательность пробелы из шаблона и включить комментарии после знака номера (#)
RightToLeft		Поиск справа налево
ECMAScript		Включает ECMAScript-совместимое поведение для выражения
CultureInvariant		Игнорировать региональные различия в языке

При вызове конструктора могут генерироваться следующие исключения:

– ArgumentException — ошибка синтаксического анализа регулярного выражения;
– ArgumentNullException — значение `pattern = null`;
– ArgumentOutOfRangeException — Значение «`options`» содержит недопустимый флаг.

2. Экземплярные методы класса Regex:

```
string[] GetGroupNames(); // возвращает массив имен групп
int[] GetGroupNumbers(); // возвращает массив номеров групп
bool IsMatch(string input); // обнаружено ли соответствие
bool IsMatch(string input, int startat);
Match Match(string input); // поиск первого соответствия
Match Match(string input, int startat);
Match Match(string input, int beginning, int length);
MatchCollection Matches(string input); // поиск всех
соответствий
MatchCollection Matches(string input, int startat);
string Replace(string input, string replacement); // замена
string Replace(string input, string replacement, int count);
string Replace(string input, string replacement, int count,
int startat);
string[] Split(string input); // разделяет строку в позициях,
определенных шаблоном РВ
string[] Split(string input, int count);
string[] Split(string input, int count, int startat);
```

3. Статические методы класса Regex:

```
static string Escape(string str); // преобразует символы «\», «*», «+», «?», «|», «{«, «[«, «(«, «)», «^», «$», «.», «#» и « », заменяя их escape-кодами
static bool IsMatch(string input, string pattern); // обнаружено ли соответствие
static bool IsMatch(string input, string pattern, RegexOptions options);
static Match Match(string input, string pattern); // поиск первого соответствия
static Match Match(string input, string pattern, RegexOptions options);
static MatchCollection Matches(string input, string pattern); // поиск всех соответствий
static MatchCollection Matches(string input, string pattern, RegexOptions options);
static string Replace(string input, string pattern, string replacement); // замена
static string Replace(string input, string pattern, string replacement, RegexOptions options);
static string[] Split(string input, string pattern); // разделяет строку
static string[] Split(string input, string pattern, RegexOptions options);
static string Unescape(string str); // Преобразует все escape-коды обратно в символы
```

4. Свойства класса Capture:

```
int Index; // позиция в исходной строке
int Length; // длина подстроки
string Value; // захваченная подстрока
```

5. Свойства класса Group:

```
CaptureCollection Captures; // коллекция захватов
bool Success; // успешно ли совпадение
```

Также класс Group наследует свойства класса Capture. При этом позиция и длина захваченной подстроки соответствуют последнему захвату в группе.

6. Члены класса Match:

```
static Match Empty; // пустая группа (если match == Match.Empty, совпадение не найдено)
GroupCollection Groups; // коллекция групп
string Result(string replacement); // замена указанного шаблона
Match NextMatch(); // следующее соответствие
```

Также класс Match наследует свойства класса Group. При этом позиция и длина захваченной подстроки, а также коллекция захватов соответствуют первой группе в совпадении.

3.4.4 Включение действий и поиск ошибок

Как таковое, включение действий в синтаксис регулярные выражения не поддерживают. Если какие-либо конструкции РВ требуют выполнения дополнительных действий, то их включают в группы (обычно именованные), и затем обрабатывают отдельно. Например, вспомним РВ, описывающее число с фиксированной точкой:

$$«(s + e)(pd^{+} + d^{+}(pd^{*} + e))»,$$

где s = '+' + '–', p = '.', d = '0' + '1' + '2' + '3' + '4' + '5' + '6' + '7' + '8' + '9'.

Если его переписать в терминах РВ класса Regex, получим следующее:

$$s = \+|-$$
$$p = \.$$
$$d = \d$$
$$s + e = s? = (\+|-)?$$
$$pd^{*} + e = (pd^{*})? = (\.\d*)?$$

Итого имеем:

$$@"(\+|-)?(\.\d+|\d+(\.\d*)?)".$$

Данный шаблон позволит найти все числа с фиксированной точкой в некоторой входной цепочке. Если же необходимо убедиться, что входная цепочка содержит **только** такое число, добавим еще маркеры начала и конца строки:

$$@"^(\+|-)?(\.\d+|\d+(\.\d*)?)$".$$

Попробуем теперь ограничить число цифр в мантиссе восемью. Для этого все цифры занесем в специальную группу с именем digit. Если в данной группе окажется больше восьми элементов, то позиция девятого элемента в исходной строке будет позицией ошибки:

```
Regex r = new Regex(@"^(\+|-
)?(\.(?'digit'\d)+|(?'digit'\d)+(\.(?'digit'\d)*)?)$");
Match m = r.Match("+1.23456789");
if (m.Success)
{
     Group g = m.Groups["digit"];
     if (g.Captures.Count < 9) Console.WriteLine("OK");
```

```
     else Console.WriteLine("Ошибка в позиции {0}: мантисса
содержит больше 8 значащих цифр", g.Captures[8].Index + 1);
}
else Console.WriteLine("Строка не содержит число с
фиксированной точкой");
```

Заметим, что во втором случае, когда в строке синтаксическая ошибка (например, «+1.2345!678»), информация о позиции ошибки на консоль не выводится. Дело в том, что если совпадение не было найдено, то ошибочной считается вся строка. Чтобы определить позицию ошибки, необходимо тщательно продумать структуру РВ. Например, для обнаружения ошибки в строках типа «+1.2345!678» или «!1.2345678» можно предложить следующий код:

```
Regex r = new Regex(@"(\+|-
)?(\.(?'digit'\d)+|(?'digit'\d)+(\.(?'digit'\d)*)?)");
string str = "+1.2345!678";
Match m = r.Match(str);
if (m.Success)
{
     Group g = m.Groups["digit"];
     if (g.Captures.Count < 9)
     {
          if (m.Index > 0) Console.WriteLine("Ошибка в позиции
1: неожиданный символ '{0}'", str[0]);
          else if (m.Length < str.Length)
Console.WriteLine("Ошибка в позиции {0}: неожиданный символ
'{1}'", m.Length + 1, str[m.Length]);
          else Console.WriteLine("OK");
     }
     else Console.WriteLine("Ошибка в позиции {0}: мантисса
содержит больше 8 значащих цифр", g.Captures[8].Index + 1);
}
else Console.WriteLine("Строка не содержит число с
фиксированной точкой");
```

Здесь шаблон уже не привязан к началу и концу входной цепочки, но затем проверяется отдельно — совпадает ли начало и конец числа с фиксированной точкой с началом и концом цепочки.

Таким образом, структура РВ должна быть такой, чтобы не просто проверить соответствие шаблона входной цепочке, но найти максимальное количество соответствий, пусть даже частичных. Тогда позицией ошибки будет:

1) первая позиция входной цепочки (1), если первое соответствие не начинается с позиции `Index = 0`;

2) позиция, следующая за последним соответствием (`Index + Length + 1`), если она не совпадает с последней позицией входной цепочки;
3) позиция первого разрыва между соответствиями, если символ, следующий за предыдущим соответствием, не является первым символом следующего соответствия.

Пример:

```
Regex r = new Regex(@"\w+(\.\w+)*");
string str = "abc.xyz.pqr";
MatchCollection m = r.Matches(str);
if (m.Count == 1 && m[0].Value == str)
Console.WriteLine("OK");
else if (m.Count == 0) Console.WriteLine("Ошибка в позиции 1
'{0}'", str[0]);
else
{
     int index = 0;
     for (int i = 0; i < m.Count; i++)
     {
          if (m[i].Index > index) break;
          index = m[i].Index + m[i].Length;
     }
     Console.WriteLine("Ошибка в позиции {0} '{1}'", index + 1,
str[index]);
}
```

В данном примере описан шаблон для строк, состоящих из символов идентификаторов (букв и цифр), разделенных точками. Примеры работы данной программы:

– «abc.xyz.pqr» — правильно;
– «+abc.xyz.pqr» — ошибка в позиции 1 («+»);
– «abc.xyz.pqr!» — ошибка в позиции 12 («!»);
– «abc.xyz!.pqr» — ошибка в позиции 8 («!»).

Однако для строки «abc.xyz.+pqr» данная программа сообщит об ошибке в позиции 8 («.»), хотя точка стоит на своем месте, а неправильным является уже следующий знак «+». Дело в том, что описанный шаблон требует обязательного наличия символов после точки, иначе соответствие не будет найдено для всей группы, включая точку. Исправить ситуацию можно следующим образом: заменить шаблон строкой @"\w+(\.\w+)*(\.(?!$))?". Здесь последняя группа «(\.(?!$))?» допускает наличие точки в конце соответствия, но не в конце строки (использовано отрицательное

утверждение просмотра вперед для маркера конца строки «$»). Таким образом, с новым шаблоном имеем:

– «abc.xyz.+pqr» — ошибка в позиции 9 («+»);
– «abc.xyz.pqr.» — ошибка в позиции 12 («.»).

Этот пример еще раз подчеркивает, насколько внимательно нужно подходить к описанию шаблона РВ, чтобы правильно определить позицию ошибки в строке.

3.4.5 Сбалансированные определения

Выше было сказано, как преобразовать ДКА в РВ, но ничего не говорилось про ДМПА. Как включение действий в синтаксис, так и работа со стеком напрямую в РВ невозможна. Можно лишь, как и в предыдущем случае, имитировать ее с помощью конструкций группирования.

Для проверки рекурсивно вложенных описаний (см. п. 3.3.5) можно использовать так называемые *сбалансированные определения*. В этом случае именованные группы играют роль стека. Так, конструкция «(?'x')» добавляет в коллекцию с именем «x» один элемент, а конструкция «(?'-x')» убирает из коллекции «x» один элемент. Поэтому в конце остается лишь проверить, что в коллекции не осталось элементов — «(?(x)(?!))».

Тогда язык L, описывающий вложенные операторы языка Pascal «**`begin end;`**» (см. п. 3.3.5) можно представить в виде следующего регулярного выражения:

```
@"^\s*((?'begin'begin\s+)+(?'-begin'end\s*;\s*)+)*(?(begin)(?!))$".
```

Здесь сопоставление начинается в начале строки, где могут располагаться пробелы в произвольном количестве. Далее все встречаемое ключевые слова **`begin`** с последующими пробелами (как минимум, одним), помещаются в группу begin. Каждое встречаемое ключевое слово **`end`** с последующей точкой с запятой (окруженной произвольным количеством пробелом) убирает один элемент из группы begin. Если в этой группе при этом не окажется элементов (т.е. имеется непарный элемент **`end`**), то получим ошибку сопоставления. В конце делаем дополнительную проверку, чтобы в группе begin не было элементов (т.е. отслеживаем ситуации с непарными элементами **`begin`**). Такая структура РВ даст лишь ответ на вопрос — является ли описание правиль-

ным, но ничего не скажет о позиции ошибки в неправильном описании. Для получения позиции ошибки, как уже было сказано выше, необходимо продумать конструкцию шаблона РВ.

3.5 Работа с КС-грамматиками

В данном разделе описывается программирование синтаксических анализаторов на основе контекстно-свободных грамматик (КС-грамматик). Дополнительные сведения можно найти в учебном пособии [1], слайд-лекциях, входящих в состав курса, а также дополнительной литературе [5, 6, 7].

3.5.1 Составление правил грамматик

Составление правил грамматик — такая же слабо формализованная задача, как и составление функции переходов ДМП-автомата или регулярного выражения. Можно использовать те же рекомендации, которые были описаны выше для построения функции переходов ДМПА. Только вместо состояний будем получать новые нетерминалы грамматики, а вместо переходов — новые правила.

Однако если имеется функция переходов δ ДКА $M = (Q, \Sigma, \delta, q_0, F)$, можно построить соответствующую ей праволинейную грамматику $G = (Q, \Sigma, P, q_0)$. То есть нетерминалами этой грамматики являются состояния ДКА ($N = Q$), алфавиты совпадают, а стартовым символом является q_0. Правила грамматики P формируются следующим образом:

1) если имеется функция переходов $\delta(X_i, a) = X_j$, $a \in \Sigma$, то добавить в грамматику правило $X_i \to a\ X_j$;
2) если состояние X_i является конечным ($X_i \in F$ или $\delta(X_i, \perp) = HALT$), то добавить в грамматику правило $X_i \to e$.

Несложно выполнить обратное действие — преобразовать праволинейную грамматику в ДКА.

Для примера рассмотрим функцию переходов ДКА, описывающего число с фиксированной точкой (табл. 3.9). Используя приведенный выше алгоритм, получим грамматику со следующими правилами:

$q_0 \to + q_1 \mid - q_1 \mid . q_2 \mid$ 0-9 q_3

$q_1 \to . q_2 \mid$ 0-9 q_3

$q_2 \to$ 0-9 q_4

$q_3 \to$. q_4 | 0-9 q_3 | e

$q_4 \to$ 0-9 q_4 | e

В принципе, данная грамматика является корректной LL(1)-грамматикой. Чтобы получить из нее LR(1)-грамматику, необходимо избавиться от e-правил:

$q_0 \to$ + q_1 | – q_1 | . q_2 | 0-9 | 0-9 q_3

$q_1 \to$. q_2 | 0-9 | 0-9 q_3

$q_2 \to$ 0-9 | 0-9 q_4

$q_3 \to$. | . q_4 | 0-9 | 0-9 q_3

$q_4 \to$ 0-9 | 0-9 q_4

Получили два одинаковых правила, избавляемся от одного из них:

$q_0 \to$ + q_1 | – q_1 | . q_2 | 0-9 | 0-9 q_3

$q_1 \to$. q_2 | 0-9 | 0-9 q_3

$q_2 \to$ 0-9 | 0-9 q_2

$q_3 \to$. | . q_2 | 0-9 | 0-9 q_3

Обе полученные грамматики можно сокращать и далее. Смысл сокращения грамматик состоит в том, что тем самым мы уменьшаем размер таблицы разбора. То есть таблица будет быстрее строиться и быстрее осуществлять разбор. Например, для LL(1)-грамматики:

FIXED → *SIGN MANT*

SIGN → + | – | e

MANT → . *NUM* | *NUM FRACT*

NUM → 0-9 *NUM*2

*NUM*2 → *NUM* | e

FRACT → . *NUM*2 | e

Здесь для удобства чтения грамматики были введены нетерминалы «*FIXED*» для числа с фиксированной точкой, «*SIGN*» для знака, «*MANT*» для мантиссы, «*NUM*» для последовательности из одной и более цифр, «*NUM*2» для последовательности из любого количества цифр, «*FRACT*» для дробной части числа, имеющего целую часть. В итоге вместо 34 строк в таблице разбора останется только 27. Для LR(1)-грамматики:

FIXED → *SIGN MANT* | *MANT*

SIGN → + | –
MANT → . *NUM* | *NUM* | *NUM* . | *NUM* . *NUM*
NUM → 0-9 | 0-9 *NUM*

В данном случае количество состояний уменьшится с 20 до 16.

Выше было сказано, как связаны грамматики и ДКА, но ничего не говорилось про ДМПА. Вообще, сама по себе грамматика манипуляций со стеком (магазином) не содержит. В этом просто нет необходимости, т.к. грамматика по своей сути является рекурсивным описанием языка. Стек используется лишь на этапе разбора по таблице.

Например, для языка, описывающего идентификатор, заключенный в скобки (табл. 3.11), LL(1)-грамматика строится достаточно просто:

ID → *LETTER TAIL* | (*ID*)
LETTER → _ | a-z | A-Z
TAIL → *LETTER TAIL* | 0-9 *TAIL* | *e*

Соответственно, LR(1)-грамматика будет следующей:

ID → *LETTER* | *LETTER TAIL* | (*ID*)
LETTER → _ | a-z | A-Z
TAIL → *LETTER TAIL* | 0-9 *TAIL* | *LETTER* | 0-9

3.5.2 Включение действий в синтаксис

Как уже отмечалось, действия в синтаксический анализатор (конечный автомат или КС-грамматику) включаются в следующем виде:

$$\langle A_1 \rangle, \langle A_2 \rangle, \ldots$$

Например, если мы хотим ограничить количество значащих цифр в числе с фиксированной точкой, получим следующую LL(1)-грамматику:

FIXED → *SIGN MANT*
SIGN → + | – | *e*
MANT → . *NUM* | *NUM FRACT*
NUM → 0-9 $\langle A_1 \rangle$ *NUM*2
*NUM*2 → *NUM* | *e*
FRACT → . *NUM*2 | *e*

LR(1)-грамматика будет следующей:

FIXED → *SIGN MANT* | *MANT*

SIGN → + | –

MANT → . *NUM* | *NUM* | *NUM* . | *NUM* . *NUM*

NUM → 0-9 ⟨A_1⟩ | 0-9 ⟨A_1⟩ *NUM*

Действие, по сравнению с табл. 3.10, для простоты оставлено лишь одно: ⟨A_1⟩ — увеличение счетчика значащих цифр на 1 (count := count + 1) и проверка: если count > *N*, где *N* — максимальное количество значащих цифр, то перевести анализатор в состояние *ERROR*. Инициализацию счетчика значащих цифр значением 0 (count := 0) необходимо выполнить перед началом разбора.

Соответственно, при чтении грамматики и построении таблицы разбора необходимо учесть, что заключенные в угловые скобки символы не являются элементами правил. Но в ячейки таблицы они попадают. То есть в LL(1)-таблице появится еще один столбец действий, а в ячейке LR(1)-таблицы дополнительное значение — идентификатор действия.

3.5.3 Разбор по символам и по лексемам

Все, что было сказано про посимвольный разбор и разбор по лексемам для ДКА и ДМПА в п. 3.3.4 и 3.3.5, верно и для КС-грамматик. То есть разбор можно осуществлять лишь посимвольно (что, естественно, приводит к увеличению грамматики и таблицы разбора), по лексемам (если входная цепочка представляет собой поток фиксированных ключевых слов) либо в смешанном режиме.

При посимвольном разборе все символы входной цепочки, включая символы-разделители, входят в алфавит грамматики. Например, LL(1)-грамматику для языка *L*, описывающего вложенные операторы языка Pascal «**`begin end;`**» (табл. 3.13), посимвольно можно записать следующим образом:

PASCAL → *SE BE*

BE → b e g i n *S BE* e n d *SE* ; *PASCAL* | $

S → #9 *SE* | #10 *SE* | #13 *SE* | #32 *SE*

SE → *S* | $

Здесь «*BE*» — это вложенные операторы **begin**/**end**, «*S*» — последовательность из одного и более символов-разделителей (табуляций, переходов на следующую строку, возвратов каретки и пробелов), «*SE*» — последовательность из произвольного количества символов-разделителей. Сами символы-разделители обозначены решеткой с кодами символов, при программировании грамматики можно выбрать любые другие обозначения, не конфликтующие с алфавитом языка и другими спецсимволами. Символ пустой цепочки в данном случае обозначен как «$», как раз с той целью, чтобы не конфликтовать с символом «e», входящим в алфавит языка. Получаем таблицу разбора, состоящую из 35 строк. Если использовать алгоритм, устраняющий *e*-правила, получим LR(1)-грамматику

PASCAL → *S* | *S BE* | *BE*

BE → b e g i n *S BE* e n d *S* ; *PASCAL*

BE → b e g i n *S BE* e n d ; *PASCAL*

BE → b e g i n *S* e n d *S* ; *PASCAL*

BE → b e g i n *S* e n d ; *PASCAL*

BE → b e g i n *S BE* e n d *S* ;

BE → b e g i n *S BE* e n d ;

BE → b e g i n *S* e n d *S* ;

BE → b e g i n *S* e n d ;

S → #9 *S* | #10 *S* | #13 *S* | #32 *S* | #9 | #10 | #13 | #32

Однако:

1. При таком «механическом» подходе структура грамматики оказывается далека от идеальной. При построении таблицы мы получим 78 различных состояний анализатора.
2. В чистом виде, грамматики типа LR не позволяют описывать языки, содержащие пустые цепочки. В данном же случае входная цепочка может быть пустой. Поэтому таблицу разбора необходимо модифицировать (см. п. 3.5.5.3).

Модификация грамматики позволяет уменьшить количество состояний до 55:

PASCAL → *BE* | *BES*

BE → b e g i n *BES* e n d *ES*

BES → *S* | *S BE*

ES → *S* ; | ; | *S* ; *PASCAL* | ; *PASCAL*

S → x9 | x10 | x13 | x32 | *S S*

Если же разбор будет производиться по лексемам, размер таблиц резко уменьшится. Так, для следующей LL(1)-грамматики:

PASCAL → begin *PASCAL* end ; *PASCAL* | $

в таблице разбора будет всего 8 строк, а в соответствующей LR(1)-грамматике — всего 18 состояний:

PASCAL → BE

BE → begin BE end ES

BE → begin end ES

ES → ; BE | ;

Но для этого в программе необходимо реализовать лексический анализатор, разбивающий входной поток символов на лексемы (см. п. 3.3.5).

Более сложный случай — смешанный разбор, когда одна часть терминалов грамматики (символов алфавита) является лексемами, а другая часть — отдельными символами. В этом случае в правилах грамматики встречаются три возможные ситуации:

1. Ситуация, когда символы-разделители в данной позиции необходимы (например, для отделения ключевых слов друг от друга, ключевых слов от идентификаторов и т.п.).
2. Ситуация, когда символы-разделители в данной позиции допускаются, но их наличие не обязательно (например, вокруг знаков пунктуации).
3. Ситуация, когда символы-разделители в данной позиции недопустимы (например, внутри идентификаторов или ключевых слов).

Для правильной обработки в программе всех этих ситуаций, помимо разбития алфавита символов входной цепочки на три подмножества, необходимо также выделить правила, при разборе которых наличие символов-разделителей во входной цепочке недопустимо (например, это правила, описывающие идентификаторы, числа и другие лексемы, не являющиеся ключевыми словами). Это требует дополнительного усложнения драйвера синтаксического анализатора.

3.5.4 LL(1)-ГРАММАТИКИ

3.5.4.1 Общие определения

Грамматикой типа LL(1) называется четверка $G = (N, \Sigma, P, S)$, где

- N — конечное множество нетерминалов;
- Σ — непересекающееся с N конечное множество терминалов;
- P — конечное подмножество множества $N\times(N\cup\Sigma)^*$, элемент (A, α) множества P называется правилом и записывается $A\to\alpha$;
- S — выделенный символ из N, называемый начальным (стартовым, исходным) символом.

LL(1)-грамматику $G = (N, \Sigma, P, S)$ можно задавать лишь множеством правил при условии, что правила со стартовым символом в левой части записаны первыми. Тогда:

- нетерминалами N будут те элементы грамматики, которые встречаются слева от знака вывода в порождающих правилах;
- терминалами Σ будут все остальные элементы грамматики, за исключением символа пустой цепочки e;
- стартовым символом грамматики S будет нетерминал из левой части первого порождающего правила.

Грамматика представляет собой одномерный массив структур следующего вида:

```
declare 1 RULE,
          2 left string,
          2 right LIST,
          2 actions LIST,
          2 mark_left int,
          2 mark_right int,
          2 start LIST,
          2 follow LIST,
          2 terms LIST;
declare GRAMMAR(N) RULE;
```

Каждая структура соответствует правилу грамматики, а N — количество правил. Здесь `LIST`

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

Поле «left» соответствует нетерминалу в левой части правила, поле «right» — списку элементов правой части правила, «actions» — идентификаторы действий для соответствующих элементов правой части правила, «mark_left» — метке левой части правила, «mark_right» — метке первого элемента правой части правила, «start», «follow» и «terms» — множествам предшествующих, последующих и направляющих символов соответственно. То есть идентификаторы действий из правил грамматики удаляются и хранятся отдельно.

3.5.4.2 Определение множеств направляющих символов

Множество терминальных *символов-предшественников* (от англ. start) определяется следующим образом:

$$a \in S(\alpha) \Leftrightarrow \alpha \Rightarrow^{*} a\beta,$$

где

- a — терминал или пустая цепочка, $a \in \Sigma \cup \{e\}$;
- α и β — произвольные цепочки терминалов и/или нетерминалов, $\alpha, \beta \in (N \cup \Sigma)^{*}$;
- $S(\alpha)$ — множество символов-предшественников цепочки α.

Пусть имеется цепочка $\alpha = X_1 X_2 \ldots X_n$, где $X_i \in N \cup \Sigma \cup \{e\}$. Тогда множество символов-предшественников

$$S(\alpha) = \bigcup_{i=1}^{k} \left(S(X_i) - \{e\} \right) \cup \Delta,$$

где

$$k = \begin{cases} j \,\middle|\, e \notin S(X_j) \wedge e \in S(X_i), i < j, \\ n \,\middle|\, e \in S(X_i), i = 1, 2, \ldots, n, \end{cases}$$

$$\Delta = \begin{cases} \{e\} \,\middle|\, e \in S(X_i), i = 1, 2, \ldots, n, \\ \varnothing \,\middle|\, e \notin S(X_j) \wedge e \in S(X_i), i \neq j. \end{cases}$$

То есть k — это номер первого символа цепочки, для которого $e \notin S(X_k)$. Если же пустая цепочка присутствует во всех $S(X_i)$, $i = 1,2,\ldots,n$, то $k = n$. Пустая цепочка войдет во множество $S(\alpha)$ только в том случае, если для любого X_i, $i = 1,2,\ldots,n$, выполняется условие $e \in S(X_i)$. В этом случае получаем $\Delta = \{e\}$, в противном случае $\Delta = \varnothing$.

Для нахождения множества символов-предшественников для левых частей всех правил грамматики $G = (N, \Sigma, P, S)$ существует следующий нерекурсивный алгоритм:

1. Для всех правил грамматики $(A \to \alpha) \in P$ положить $S(A) = \varnothing$.
2. Для каждого правила грамматики $(A \to \alpha) \in P$ добавить к множеству $S(A)$ элементы множества $S(\alpha)$, полученные по приведенной выше формуле:
$$S(A) = S(A) \cup S(\alpha).$$
При этом
$$S(X_i) = \begin{cases} \{X_i\} \mid X_i \in \Sigma \cup \{e\}, \\ \bigcup_j S(X_j) \mid X_i \in N \wedge (X_j \to \beta) \in P \wedge X_j = X_i. \end{cases}$$
То есть для терминала или пустой цепочки во множестве $S(X_i)$ будет лишь один элемент — данный терминал или пустая цепочка. Для нетерминала множество $S(X_i)$ получается путем объединения всех множеств символов-предшественников тех правил грамматики, у которых данный нетерминал стоит слева от знака вывода.
3. Если при выполнении шага 2 хотя бы в одном множестве $S(A)$ появился хотя бы один новый элемент, вернуться на шаг 2. Иначе алгоритм заканчивает свою работу.

Множество терминальных *последующих символов* (от англ. follow) определяется следующим образом:

$$a \in F(A) \Leftrightarrow \alpha A \beta \Rightarrow^* \alpha A a \gamma,$$

где:

- a — терминал или признак конца цепочки, $a \in \Sigma \cup \{\perp\}$;
- α, β и γ — произвольные цепочки терминалов и/или нетерминалов, $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$;
- A — нетерминал, $A \in N$;

– $F(A)$ — множество последующих символов для нетерминала A.

Для правила грамматики вида $B \to \alpha A \beta$ во множество $F(A)$ входят элементы множества $S(\beta)$, кроме e, а если после нетерминала A цепочка β может заканчиваться ($\beta \Rightarrow^* e$), то также и элементы множества $F(B)$:

$$F(A) = \left(S(\beta) - \{e\}\right) \cup \begin{cases} F(B) \mid \beta = e \vee e \in S(\beta), \\ \varnothing \mid \beta \neq e \wedge e \notin S(\beta). \end{cases}$$

Для нахождения множества последующих символов для нетерминалов грамматики $G = (N, \Sigma, P, S)$ существует следующий нерекурсивный алгоритм:

1. Для всех нетерминалов грамматики $A \in N$ положить $F(A) = \varnothing$. Для стартового нетерминала положить $F(S) = \{\perp\}$.
2. Для каждого вхождения нетерминала A в правую часть порождающих правил грамматики вида $(B \to \alpha A \beta) \in P$ добавить к множеству $F(A)$ новые элементы, найденные по приведенной выше формуле:
$$F(A) = F(A) \cup \left(S(\beta) - \{e\}\right) \cup \begin{cases} F(B) \mid \beta = e \vee e \in S(\beta), \\ \varnothing \mid \beta \neq e \wedge e \notin S(\beta). \end{cases}$$
3. Если при выполнении шага 2 хотя бы в одном множестве $F(A)$ появился хотя бы один новый элемент, вернуться на шаг 2. Иначе алгоритм заканчивает свою работу.

Если A — нетерминал в левой части правила, то его *направляющими символами* $T(A)$ будут символы-предшественники A и все символы, следующие за A, если A может генерировать пустую строку:

$$T(A) = \left(S(A) - \{e\}\right) \cup \begin{cases} F(A) \mid e \in S(A), \\ \varnothing \mid e \notin S(A). \end{cases}$$

Для грамматики типа LL(1) множества направляющих символов для порождающих правил с одинаковым символом в левой части не должны пересекаться. То есть если имеются альтернативы порождающего правила:

$A_1 \to \alpha_1$
$A_2 \to \alpha_2$
……
$A_n \to \alpha_n$

и $A_1 = A_2 = \ldots = A_n$ соответственно, то

$$T(A_i) \cap T(A_j) = \varnothing \text{ при } i \neq j.$$

Для нетерминала A в правой части правила $B \to \alpha A \beta$ его направляющие символы определяются так:

$$T(A) = \begin{cases} \left(S(A\beta) - \{e\}\right) \cup F(B) \;\middle|\; e \in S(A\beta), \\ S(A\beta) \;\middle|\; e \notin S(A\beta). \end{cases}$$

3.5.4.3 Построение таблицы разбора

Таблица разбора (табл. 3.18) в общем виде представляет собой одномерный массив структур следующего вида:

```
declare 1 TABLEROW,
          2 terminals LIST,
          2 action int,
          2 jump int,
          2 accept bool,
          2 stack bool,
          2 return bool,
          2 error bool;
declare TABLE(N) TABLEROW;
```

где N — количество строк таблицы (или количество элементов грамматики). Здесь `LIST` — множество направляющих символов:

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

Кроме того, во время разбора потребуется стек для хранения адресов (номеров строк таблицы) возврата.

Табл. 3.18 — Структура таблицы разбора для грамматик типа LL(1)

№	terminals	action	jump	accept	stack	return	error
1							
2							
…	…	…	…	…	…	…	…
N							

Алгоритм построения таблицы разбора следующий:

1. Разметить грамматику. При этом порядковые номера $i \in M$ присваиваются всем элементам грамматики, от первого правила к последнему, от левого символа к правому. При наличии у порождающего правила альтернатив сначала нумеруются левые части всех альтернативных правил, а уже затем их правые части. Еще одним важным требованием является то, что все альтернативные правила должны следовать друг за другом в списке правил.

2. Построить два вспомогательных множества $M_L \subset M$ и $M_R \subset M$. В первое включить порядковые номера $i \in M$ элементов грамматики, расположенных в левой части порождающего правила, во второе — порядковые номера элементов, которыми заканчиваются правые части порождающих правил:

$$M_L = \{i \mid (X_i \to \alpha) \in P\},$$
$$M_R = \{j \mid (A \to \alpha X_j) \in P\}.$$

3. Построить таблицу, состоящую из столбцов **terminals**, **action**, **jump**, **accept**, **stack**, **return**, **error**. Количество строк таблицы определяется количеством элементов во множестве M — по одной строке на каждый элемент грамматики.

4. Заполнить ячейки таблицы.

4.1. Для нетерминала множество **terminals** совпадает с множеством направляющих символов. Для терминала множество **terminals** включает лишь сам терминал. Для пустой цепочки множество **terminals** совпадает с множеством направляющих символов соответствующего правила, у которого данная пустая цепочка стоит в правой части:

$$\mathbf{terminals}_i = \begin{cases} T(X_i) \mid X_i \in N, \\ \{X_i\} \mid X_i \in \Sigma, \\ T(X_j) \mid X_i = e \wedge (X_j \to X_i) \in P. \end{cases}$$

4.2. Действие **action** соответствует действию, хранимому в списке `actions` (см. п. 3.5.4.1) для элемента грамматики X_i.

4.3. Переход **jump** от нетерминала в левой части правила осуществляется к первому символу правой части этого правила. Переход от нетерминала в правой части правила осуществляется на такой же нетерминал в левой части. Причем если имеется

несколько альтернатив соответствующего порождающего правила, переход осуществляется к первой из альтернатив. От терминала, не последнего в цепочке правой части правила, переход осуществляется к следующему символу цепочки. Если терминал завершает цепочку либо это символ *e* (он в правой части может быть только один, поэтому будет одновременно начинать и завершать ее), то значение **jump** равно нулю:

$$\mathbf{jump}_i = \begin{cases} k \,\middle|\, i \in M_L \wedge (X_i \to X_k \alpha) \in P, \\ k \,\middle|\, X_i \in N \wedge i \notin M_L \wedge X_k = X_i \wedge j \in M_L \wedge j-1 \notin M_L, \\ i+1 \,\middle|\, i \in \Sigma \wedge i \notin M_R, \\ 0 \,\middle|\, i \in \Sigma \cup \{e\} \wedge i \in M_R. \end{cases}$$

4.4. Символ принимается (**accept**), если это терминал:

$$\mathbf{accept}_i = \begin{cases} \text{true} \,\middle|\, X_i \in \Sigma, \\ \text{false} \,\middle|\, X_i \notin \Sigma. \end{cases}$$

4.5. Номер строки таблицы разбора помещается в стек (**stack**), если соответствующий символ грамматики — нетерминал в правой части порождающего правила, но не в конце цепочки правой части:

$$\mathbf{stack}_i = \begin{cases} \text{true} \,\middle|\, X_i \in N \wedge i \notin M_L \wedge i \notin M_R, \\ \text{false} \,\middle|\, X_i \notin N \vee i \in M_L \vee i \in M_R. \end{cases}$$

4.6. Возврат (**return**) по стеку при разборе осуществляется, если символ грамматики является терминалом, расположенным в конце цепочки правой части порождающего правила, или символом *e* (т.е. когда $\mathbf{jump}_i = 0$):

$$\mathbf{return}_i = \begin{cases} \text{true} \,\middle|\, i \in \Sigma \cup \{e\} \wedge i \in M_R, \\ \text{false} \,\middle|\, i \notin \Sigma \cup \{e\} \vee i \notin M_R. \end{cases}$$

4.7. Ошибка (**error**) при разборе не генерируется, если следующий символ грамматики находится в левой части альтернативного порождающего правила:

$$\mathbf{error}_i = \begin{cases} \text{false} \,\middle|\, i \in M_L \wedge i+1 \in M_L, \\ \text{true} \,\middle|\, i \notin M_L \vee i+1 \notin M_L. \end{cases}$$

3.5.4.4 Разбор цепочки по таблице

Для разбора цепочки $\alpha = a_1 a_2 \ldots a_n \perp$ нам потребуется магазин (стек) *M*. Операцию помещения элемента в стек будем обозначать как $M \leftarrow x$, операцию извлечения элемента из стека — как $M \rightarrow x$. Номер текущей строки таблицы разбора обозначим как *i*, номер текущего символа во входной строке — *k*, список действий — *actions*.

Алгоритм разбора цепочки по таблице:

1. Положить $i := 1$ (разбор начинается с первой строки таблицы).
2. Положить $k := 1$ (разбор идет слева направо, начиная с первого символа цепочки).
3. $M \leftarrow (0, (\varnothing))$ (поместить в стек значение 0 и пустой список действий).
4. Если $a_k \in$ **terminals**$_i$, то:
   - 4.1. Если **accept**$_i$ = true, то:
     - 4.1.1. $k := k + 1$ (перейти к следующему символу входной цепочки);
     - 4.1.2. Если **action**$_i \neq 0$, то выполнить действие с идентификатором **action**$_i$.
   - 4.2. Если **stack**$_i$ = true, то $M \leftarrow (i, ($**action**$_i))$ (поместить в стек номер текущей строки таблицы разбора и действие **action**$_i$).
   - 4.3. Если **return**$_i$ = true, то:
     - 4.3.1. $M \rightarrow (i, actions)$;
     - 4.3.2. Если список *actions* не пуст, выполнить все действия из него;
     - 4.3.3. Если $i = 0$, то перейти на шаг 6;
     - 4.3.4. $i := i + 1$;
     - 4.3.5. Вернуться на шаг 4.
   - 4.4. Если **jump**$_i \neq 0$, то:
     - 4.4.1. Если **action**$_i \neq 0$ и **accept**$_i$ = false и **stack**$_i$ = false, то для элемента $(j, actions)$ на вершине стека *M* к списку действий *actions* добавить действие **action**$_i$;
     - 4.4.2. $i :=$ **jump**$_i$;
     - 4.4.3. Вернуться на шаг 4.

5. Иначе если **error**$_i$ = false, то у правила есть еще одна альтернатива и нужно просто перейти к следующей строке таблицы разбора:
   5.1. $i := i + 1$;
   5.2. Вернуться на шаг 4.
6. В противном случае разбор окончен. Если при этом стек M пуст, а $a_k = \perp$, то разбор завершен успешно. Иначе цепочка содержит синтаксическую ошибку и k — позиция этой ошибки.

3.5.5 LR(1)-ГРАММАТИКИ

3.5.5.1 Общие определения

Грамматикой типа LR(1) называется четверка $G = (N, \Sigma, P, S)$, где

- N — конечное множество нетерминалов;
- Σ — непересекающееся с N конечное множество терминалов;
- P — конечное подмножество множества $N\times(N\cup\Sigma)^+$, элемент (A, α) множества P называется правилом и записывается $A\to\alpha$;
- S — выделенный символ из N, называемый начальным (стартовым, исходным) символом.

LR(1)-грамматику $G = (N, \Sigma, P, S)$ можно задавать лишь множеством правил при условии, что правила со стартовым символом в левой части записаны первыми. Тогда:

- нетерминалами N будут те элементы грамматики, которые встречаются слева от знака вывода в порождающих правилах;
- терминалами Σ будут все остальные элементы грамматики;
- стартовым символом грамматики S будет нетерминал из левой части первого порождающего правила.

Грамматика представляет собой одномерный массив структур следующего вида:

```
declare 1 RULE,
          2 left string,
          2 right LIST,
```

```
            2 actions LIST,
            2 start LIST;
declare GRAMMAR(N) RULE;
```

Каждая структура соответствует правилу грамматики, а N — количество правил. Здесь `LIST`

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

Поле «`left`» соответствует нетерминалу в левой части правила, поле «`right`» — списку элементов правой части правила, «`actions`» — идентификаторы действий для соответствующих элементов правой части правила, «`start`» — множеству предшествующих символов. Как и для LL-грамматик, идентификаторы действий из правил LR-грамматики удаляются и хранятся отдельно.

Для LR-грамматик нахождение множества символов-предшественников осуществляется гораздо проще, чем для LL-грамматик, т.к. в них отсутствуют e-правила. Если вспомнить формулу нахождения множества символов-предшественников цепочки $\alpha = X_1X_2\ldots X_n$:

$$S(\alpha) = \bigcup_{i=1}^{k} \left(S(X_i) - \{e\} \right) \cup \Delta,$$

и учесть, что в LR-грамматиках не используются e-правила, т.е. $e \notin S(X_i)$, то получим:

$$S(\alpha) = S(X_1).$$

Грамматики типа LR не могут иметь e-правил. Кроме того, на грамматику накладывается дополнительное ограничение — стартовый символ не должен иметь альтернатив порождающего правила. Чтобы это гарантировать, в грамматику вводится новый искусственный стартовый символ S' и новое искусственное стартовое правило $S' \to S$. Таким образом, получаем модифицированную грамматику:

$$G' = (N', \Sigma, P', S'),$$

где $N' = N \cup \{S'\}$, $P' = \{S' \to S\} \cup P$.

Конфигурацией LR-анализатора (LR-конфигурацией) будем называть пару вида:

$$(S_0\, X_1\, S_1\, X_2\, S_2 \ldots\, X_m\, S_m,\, a_i a_{i+1} \ldots a_n \perp),$$

где S_i — состояния LR-анализатора ($S_i \in \mathcal{I}$, $\mathcal{I}$ — множество всех состояний), X_i — элементы грамматики ($X_i \in \mathcal{X}$, $\mathcal{X}$ — множество всех элементов грамматики, $\mathcal{X}=N \cup \Sigma$), а a_i — терминалы из входной цепочки ($a_i \in \Sigma$). При этом последовательность элементов $S_0\, X_1\, S_1\, X_2\, S_2 \ldots\, X_m\, S_m$ представляет собой содержимое магазина (стека) анализатора, а символы $a_i a_{i+1} \ldots a_n$ — непрочитанную часть входной цепочки.

Текущее состояние анализатора находится на вершине стека — это S_m. Текущий символ входной цепочки — a_i.

Правая сентенциальная форма может быть получена конкатенацией элементов грамматики, расположенных на стеке, с непрочитанной частью входной цепочки:

$$\omega = X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n.$$

Префикс данной цепочки $X_1 X_2 \ldots X_m$ называется *активным префиксом* LR-анализатора.

3.5.5.2 Определение множества состояний и графа переходов

Решим задачу нахождения множества состояний LR-анализатора $\mathcal{I}$ и графа переходов между ними $\mathcal{G}$.

Состояние анализатора, в свою очередь, является множеством, элементами которого являются *LR-ситуации*. Таким образом, множество состояний LR-анализатора — это множество множеств (мультимножество) LR-ситуаций.

LR-ситуация — это пара вида:

$$[A \to \alpha \bullet \beta \mid a],$$

где $(A \to \alpha\beta) \in P$ — правило грамматики, $a \in \Sigma \cup \{\perp\}$ — *символ приведения* или *символ свертки*, один из множества символов, которые могут появиться на входе анализатора (это могут быть терминальные символы или маркер окончания входной цепочки).

Граф переходов между состояниями LR-анализатора — это помеченный граф $\mathcal{G} = (\mathcal{I}, R)$, вершинами которого являются состояния $I \in \mathcal{I}$, а разметка g отображает множество дуг R ($R \subset \mathcal{I} \times \mathcal{I}$) во множество $\mathcal{X}$.

Алгоритм построения множества состояний и графа переходов LR-анализатора:

1. Формируем состояние, состоящее лишь из одной LR-ситуации $[S' \to \bullet\ S \mid \bot]$, т.е. за ее основу берется стартовое правило $(S' \to S) \in P'$, точка ставится в начале правой части, а в качестве символа приведения берется маркер конца входной цепочки. Далее выполняем замыкание этого состояния (алгоритм работы этой функции рассмотрим ниже), что даст начальное состояние анализатора S_0:
$$S_0 = \textit{ЗАМЫКАНИЕ}(\{[S' \to \bullet\ S \mid \bot]\}).$$
2. Полагаем, что множество состояний LR-анализатора $\mathcal{I}$ пока состоит только из одного состояния — S_0:
$$\mathcal{I} = \{S_0\},$$
а граф переходов не имеет дуг:
$$R = \varnothing.$$
3. Организуем два вложенных цикла:
   – для каждого состояния $I \in \mathcal{I}$,
   – для каждого символа грамматики $X \in \mathcal{X}$.
   3.1. Строим новое состояние, вызывая функцию перехода (ее алгоритм также дан ниже) для аргументов I и X:
   $$J = \textit{ПЕРЕХОД}(I, X).$$
   3.2. Если $J \neq \varnothing$ и $J \notin \mathcal{I}$, т.е. состояние J не пустое уникальное (во множестве состояний нет точно такого же состояния), добавляем его во множество состояний:
   $$\mathcal{I} = \mathcal{I} \cup \{J\}.$$
   3.3. Если $J \neq \varnothing$ и $(I, J) \notin R$, т.е. состояние J не пустое и в графе переходов $\mathcal{G}$ отсутствует дуга, ведущая от вершины I в вершину J, добавляем ее в граф и помечаем меткой X:
   $$R = R \cup (I, J),\ g((I, J)) = X.$$
4. Если в процессе выполнения шага 3 во множество состояний $\mathcal{I}$ было добавлено хотя бы одно новое состояние или был добавлен хотя бы один новый переход в граф $\mathcal{G}$, вернуться на шаг 3. Иначе построение множества $\mathcal{I}$ закончено.

Алгоритм функции замыкания *ЗАМЫКАНИЕ*(I):
Вход: Аргумент I — некоторое LR-состояние.

Выход: Измененное состояние I с новыми LR-ситуациями. Цель замыкания — найти все ситуации в грамматике, в которых состояние анализатора (следовательно, и дальнейшее поведение) идентично.

1. Организуем три вложенных цикла:
   – для каждой LR-ситуации вида $[A \to \alpha \bullet B\beta \mid a] \in I$, $A \in N'$, $B \in N$, $\alpha \in \mathcal{X}^*$, $\beta \in \mathcal{X}^*$, $a \in \Sigma \cup \{\perp\}$, т.е. ситуаций, где точка стоит перед нетерминалом,
   – для каждого терминала b из множества символов-предшественников цепочки βa, т.е. $b \in S(\beta a)$, $b \in \Sigma \cup \{\perp\}$,
   – для каждого правила вывода $(B \to \gamma) \in P$, т.е. правила, у которого в левой части находится нетерминал B (перед которым стояла точка в рассматриваемой LR-ситуации).
   1.1. Формируем новую LR-ситуацию вида:
   $$\sigma = [B \to \bullet\, \gamma \mid b].$$
   1.2. Если $\sigma \notin I$, т.е. в состоянии I не было ситуации σ, добавляем ее:
   $$I = I \cup \{\sigma\}.$$
2. Если в процессе выполнения шага 1 в состояние I была добавлена хотя бы одна новая ситуация, вернуться на шаг 1. Иначе работа функции закончена.

Алгоритм функции перехода *ПЕРЕХОД*(I, X):

Вход: Аргументы I — некоторое LR-состояние, $X \in \mathcal{X}$ — символ грамматики.

Выход: Новое состояние J, полученное после распознания символа X в правых частях правил грамматики (т.е. перенос точки в LR-ситуациях через символ X).

1. Полагаем $J = \varnothing$.
2. Для всех ситуаций $[A \to \alpha \bullet X\beta \mid a] \in I$, т.е. ситуаций исходного состояния, в которых точка стоит перед X, добавляем в состояние J такую же ситуацию, но с точкой после X:
   $$J = J \cup \{[A \to \alpha X \bullet \beta \mid a]\}.$$
3. Результатом будет *ЗАМЫКАНИЕ*(J).

Таким образом, множество состояний является списком состояний, представленных структурами следующего вида:

```
declare 1 STATE,
```

```
            2 situations SITUATION,
            2 next pointer;
    declare STATES pointer;
```

Здесь SITUATION — список LR-ситуаций:

```
    declare 1 SITUATION,
            2 rule RULE,
            2 pos integer,
            2 term string,
            2 next pointer;
```

Каждая ситуация ссылается на правило грамматики «rule», а также содержит позицию разделяющей точки «pos» и символ свертки «term». Граф переходов представлен в виде списка ребер:

```
    declare 1 EDGE,
            2 term string,
            2 from STATE,
            2 to STATE,
            2 next pointer;
    declare GRAPH pointer;
```

Для каждого ребра имеем символ грамматики «term», по которому осуществляется переход, а также состояния, между которыми он осуществляется.

3.5.5.3 Построение таблицы разбора

Таблица разбора (табл. 3.19) в общем виде представляет собой двумерный массив действий:

```
    declare TABLE(N,M) ACTION;
```

где N — количество строк таблицы (количество состояний анализатора), *M* — количество элементов грамматики (терминалов, нетерминалов и маркера конца цепочки). Здесь ACTION — действия, определяющие поведение анализатора:

```
    declare 1 ACTION,
            2 action integer,
            2 shift integer,
            2 reduce integer,
            2 halt bool,
            2 error bool;
```

Элемент «`action`» представляет собой идентификатор действия (если это –1, то действие не требуется), «`shift`» — сдвиг в новое состояние (если это –1, то сдвиг не производится), «`reduce`» — номер правила, по которому выполняется свертка (если это –1, то свертка не производится), «`halt`» — признак успешного завершения разбора, «`error`» — признак синтаксической ошибки.

Табл. 3.19 — Структура таблицы разбора для грамматик типа LR(1)

	a_1	a_2	…	a_n	⊥	A_1	A_2	…	A_m
S_0									
S_1									
…									

Алгоритм построения LR-таблицы разбора следующий:

1. Создаем таблицу из $\#\mathcal{X}+1$ столбца (по столбцу на каждый терминал $a \in \Sigma$ и нетерминал $A \in N$, а также столбец для маркера конца цепочки «⊥») и $\#\mathcal{I}$ строк (по одной на каждое состояние анализатора).
2. В каждую ячейку таблицы записываем элемент *ERROR*.
3. Для всех состояний $S_i \in \mathcal{I}$ выполняем следующие действия:
   3.1. Если:
   а) в состоянии S_i присутствует LR-ситуация с точкой не в конце правила, т.е. $[A \to \alpha \bullet X\beta \mid a] \in S_i$, $A \in N'$, $X \in \mathcal{X}$, $\alpha \in \mathcal{X}^*$, $\beta \in \mathcal{X}^*$, $a \in \Sigma \cup \{\perp\}$ и
   б) в графе переходов есть переход из состояния S_i в некоторое состояние S_j, помеченный меткой X, т.е. $(S_i, S_j) \in R$, $g((S_i, S_j)) = X$,
   то $T(S_i, X) = (S_j, action_k)$, где $action_k$ — идентификатор действия, соответствующего элементу грамматики X.
   3.2. Если в состоянии S_i присутствует LR-ситуация с точкой в конце правила, т.е. $[A \to \alpha \bullet \mid a] \in S_i$, $A \in N'$, $\alpha \in \mathcal{X}^+$, $a \in \Sigma \cup \{\perp\}$, то:
   а) если $(A \to \alpha) \in P'$ — это искусственное стартовое правило, т.е. $A = S'$ (в этом случае $a = \perp$), то $T(S_i, a) = HALT$,
   б) иначе $T(S_i, a) = R_k$, где k — номер правила $(A \to \alpha) \in P$ в грамматике, т.е. $P_k = (A \to \alpha)$.

Как уже отмечалось, грамматики типа LR не позволяют описывать языки, содержащие пустые цепочки. Поэтому, если $e \in L$, то таблицу разбора необходимо модифицировать:

$$T(S_0, \perp) = HALT.$$

3.5.5.4 Разбор цепочки по таблице

Для разбора требуется один или два стека (магазина). В начале в стек состояний помещается состояние S_0 (или просто его индекс — 0). То есть конфигурацией LR-автомата будет

$$(S_0, a_1 a_2 \ldots a_n \perp).$$

Рассмотрим, как меняется конфигурация анализатора при сдвиге и свертке:

1) Если в состоянии S_m для текущего символа входной цепочки a_i таблицей разбора T приписывается сдвиг в состояние S_k, т.е. $T(S_m, a_i) = (S_k, action_j)$, то на стек помещается новая пара $(a_i\ S_k)$, а во входной цепочке происходит переход к следующему символу:

   $$(S_0\ X_1\ S_1\ X_2\ S_2 \ldots\ X_m\ S_m\ a_i\ S_k,\ a_{i+1} \ldots a_n \perp).$$

   То есть новым состоянием анализатора будет S_k, а текущим символом входной цепочки станет a_{i+1}. Затем выполняется действие $action_j$, если оно задано.

2) Если в состоянии S_m для текущего символа входной цепочки a_i таблицей приписывается свертка по правилу P_k, т.е. $T(S_m, a_i) = R_k$, то:

   2.1) Со стека снимается r пар символов, где r — количество элементов в правой части правила $P_k = (A \to \alpha) \in P$. При этом на вершине стека остается состояние S_{m-r}:

   $$(S_0\ X_1\ S_1\ X_2\ S_2 \ldots\ X_{m-r}\ S_{m-r},\ a_i a_{i+1} \ldots a_n \perp),\ r = |\alpha|.$$

   2.2) На стек помещается пара $(A\ S_j)$, где A — левая часть правила P_k, а $S_j = T(S_{m-r}, A)$:

   $$(S_0\ X_1\ S_1\ X_2\ S_2 \ldots\ X_{m-r}\ S_{m-r}\ A\ S_j,\ a_i a_{i+1} \ldots a_n \perp).$$

   Входная цепочка при этом остается без изменений. Новым состоянием анализатора становится S_j.

Конфигурация меняется по описанным правилам до тех пор, пока не будет достигнута ячейка таблицы с элементом *HALT* или *ERROR*. Во втором случае позицией ошибки будет позиция первого непрочитанного символа входной цепочки.

СПИСОК ЛИТЕРАТУРЫ

1. Калайда В. Т. Теория языков программирования и методы трансляции : учеб. пособие / В. Т. Калайда, В. В. Романенко. — Томск : ТМЦДО, 2013.
2. Регулярные выражения. Материал из Википедии. — URL: http://ru.wikipedia.org/wiki/Регулярные_выражения
3. Регулярные выражения в .NETFramework.MSDN Library. — URL : http://msdn.microsoft.com/ru-ru/library/hs600312
4. Основы программирования на C#. Лекция № 15: Пространство имен RegularExpression и классы регулярных выражений. Учебный курс Интуит. — URL : http://www.intuit.ru/department/pl/csharp/15/.
5. Серебряков В. А. Основы конструирования компиляторов / В. А. Серебряков, М. П. Галочкин. — М. : УРСС, 2001. — 174 с.
6. Фомичев В. С. Формальные языки, грамматики и автоматы [Электронный ресурс]. — URL : – http://old.eltech.ru/misc/LGA_2007_FINAL/Index.html
7. Теория и реализация языков программирования [Электронный ресурс] / В. А. Серебряков [и др.] — URL : http://www.intuit.ru/department/sa/pltheory/

Приложение А. Структура отчета

Федеральное агентство по образованию

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Факультет дистанционного образования (ФДО)

Кафедра автоматизированных систем управления (АСУ)

НАЗВАНИЕ РАБОТЫ

Отчет по лабораторной работе № X по дисциплине
«Теория языков программирования и методы трансляции»

Выполнил: _____

_____

«____» _____ 20__ г.

Проверил: _____

_____

«____» _____ 20__ г.

Томск — 20__

Содержание