Московский Государственный Технический Университет
им. Н.Э. Баумана

Т.М. Волосатова, С.В. Родионов

АВТОМАТИЗАЦИЯ ПРОЕКТИРОВАНИЯ ЛЕКСИЧЕСКИХ АНАЛИЗАТОРОВ

Учебное пособие
для практических занятий по курсу
Лингвистическое Обеспечение САПР

Москва
Издательство МГТУ им. Н.Э. Баумана
2005

ВВЕДЕНИЕ

Лексический анализ – начальная стадия трансляции формальных языков, которая выполняется во многих практических случаях при обработке текстовых данных. В частности, лексический анализ необходим для обработки входного потока символьных данных в пакетном редакторе или для распознавания директив в диалоговой системе. Однако основной областью практического приложения лексического анализа является реализация начальной стадии обработки входного потока символьной информации в трансляторах универсальных языков программирования и специализированных входных языков различной проблемной ориентации.

В общем случае целью лексического анализа является разбиение входного потока символьной информации на структурные единицы, которые имеют определенное совокупное значение и называются *лексемами*. Например, типичными лексемами в исходных текстах программ, составленных на различных алгоритмических языках, являются операторы, служебные слова, числовые или символьные константы, идентификаторы переменных, директивы препроцессора системы программирования, обозначения типов данных и классов памяти. Выявленные при лексическом анализе входного потока лексемы могут быть обработаны и закодированы соответствующим образом для передачи на следующие стадии трансляции, в частности, на стадию синтаксического анализа.

Выполнение лексического анализа в трансляторах формальных языков обеспечивает специальная программная компонента, которая называется *лексическим анализатором*. Лексический анализатор может быть реализован в формате исполняемого файла или объектного модуля, который вызывается транслятором для выполнения этапа лексического анализа. В простых случаях разработка лексического анализатора может быть выполнена традиционными средствами, которые предоставляют системы программирования алгоритмических языков высокого уровня, например, **C**, **C++** или **Pascal**. Однако в большинстве практически интересных случаев такой подход требует значительных трудозатрат и интеллектуальных усилий разработчиков системного программного обеспечения. По этой причине в настоящее время получили широкое распространение разнообразные систематические технологии и инструментальные средства, которые позволяют в значительной мере автоматизировать проектные процедуры практической разработки основных программных компонент трансляторов формальных языков в различных операционных средах.

В частности, для автоматизации проектирования лексических анализаторов операционная система **OS UNIX** предоставляет специальное инструментальное средство, которое было разработано К.С. Леском и называется *генератор лексических анализаторов* **LEX** (далее по тексту, *генератор **LEX***). Его основными компонентами являются выполняемый модуль **`lex`** и библиотека объектных модулей стандартных функций **`libl.a`** (или разделяемый объект **`libl.so`**), которые обычно располагаются, соответственно, в каталогах **`/usr/bin`** и **`/usr/lib`** файловой системы **OS UNIX.** Исходными данными для генератора **LEX** является высокоуровневая проблемно-ориентированная *спецификация лексем*, которые должен распознавать проектируемый лексический анализатор во входном потоке символьных данных. Эта спецификация имеет формат текстового файла, который может быть подготовлен в текстовом редакторе и сохранен под произвольным именем, обычно с расширением ***.l***, в любом доступном каталоге файловой системы **OS UNIX**.

Файл спецификации лексем должен иметь определенную внутреннюю структуру, в которой для описания лексем используются шаблоны *регулярных выражений*. В результате обработки файла спецификации лексем генератор **LEX** формирует исходный код проектируемого лексического анализатора на языке программирования **C**. Основная часть полученного исходного кода лексического анализатора реализует *детерминированный конечный автомат*, который генератор **LEX** автоматически строит по заданным шаблонам регулярных выражений. Полученный исходный код при необходимости может быть отредактирован и легко преобразуется в объектный модуль или выполняемый файл проектируемого лексического анализатора с помощью стандартных средств компилирующей системы языка программирования **C**.

Таким образом, практическое применение генератора **LEX** позволяет ограничить процедуру проектирования лексического анализатора составлением высокоуровневых спецификаций лексем, по которым автоматически порождается исходный код лексического анализатора на языке программирования **C**. Наиболее важным этапом рассмотренной технологии является формирование файла спецификации лексем. Основные интеллектуальные проблемы, которые при этом возникают, связаны с разработкой шаблонов регулярных выражений. Вопросы конструирования регулярных выражений и спецификации лексем подробно рассмотрены в следующих разделах предлагаемого учебного пособия.

РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

ОБЩАЯ ХАРАКТЕРИСТИКА РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

В теории формальных языков под *регулярным выражением* понимается алгебраическая запись множества символьных слов, которая имеет вид формулы, где символы конечного алфавита чередуются со знаками *регулярных операций*. Теоретически обосновано, что любое регулярное множество слов, составленных из символов конечного алфавита, замкнуто относительно регулярных операций *объединения*, *конкатенации* и *итерации*. Они образуют базовый операционный набор для конструирования регулярных выражений. Операция объединения предоставляет возможность выбора слов по любой из двух альтернатив. Конкатенация используется для слияния слов, когда одно слово приписывается к другому справа. Итерация обеспечивает повторение слова произвольное число или ноль раз.

Первоначально регулярные выражения были введены С. Клини для удобства формального описания множества регулярных событий в нейронных сетях средствами теории конечных автоматов. В практическом программировании регулярные выражения были впервые использованы К. Томпсоном в текстовом редакторе **qed**, на основе которого был разработан популярный в прошлом текстовый редактор **ed** для **OS UNIX**. Регулярные выражения **ed** уступали по возможностям регулярным выражениям **qed**, но именно они получили широкое распространение за пределами теоретических кругов. Одна из служебных команд **ed** обеспечивала вывод на консоль строк редактируемого текстового файла, в которых найдено совпадение фрагментов текста с заданным регулярным выражением. Эта функция оказалась настолько полезной, что была преобразована в отдельную утилиту **grep** (**G**lobal **R**egular **E**xpression **P**rint), которая в настоящее время широко используется в различных операционных средах для поиска информации в текстовых файлах на основе заданного регулярного выражения. В настоящее время различные диалекты регулярных выражений активно применяются в разнообразных инструментальных средствах, которые ориентированы на обработку текстовой информации. В частности, регулярные выражения применяются в сценарных языках **Perl**, **Tcl** и **Pyton**, в текстовых редакторах, например, **Emacs** и **vi**, в языках командных процессоров **sh**, **csh**, **ksh**, **bash** для **OS UNIX**, в генераторе отчетов **awk**, в потоковом редакторе **sed** и в других специализированных утилитах, ориентированных на обработку текстовых данных.

ЭЛЕМЕНТЫ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

Генератор **LEX** использует регулярные выражения для описания символьных конструкций формального языка, обозначающих лексемы. Регулярные выражения, которые обрабатывает генератор **LEX,** строятся из обычных текстовых символов (*литералов*) и управляющих символов (*метасимволов*), которые обозначают знаки регулярных операций и выполняют служебные функции. Генератор **LEX** использует расширенный набор регулярных операций, для обозначения которых применяются следующие метасимволы:

" \ ^ $ / <> | [] . ? * + {} ()

Перечисленные метасимволы имеют специальный смысл и выполняют определенные служебные функции в регулярных выражениях. Остальные отображаемые символы, например, буквы или цифры, считаются обычными символами и интерпретируются в регулярном выражении в соответствии с их кодами ASCII. Исключение составляет только символы *табуляции*, *процента* % и *пробел*, которые не могут быть непосредственно использованы в регулярных выражениях. Особый смысл этих символов связан с особенностями формата спецификации лексем генератора **LEX**, где они применяются для конструирования служебных разделителей.

КОНКАТЕНАЦИЯ ЛИТЕРАЛОВ

В простейшем случае регулярное выражение может состоять только из обычных текстовых символов и интерпретируется как последовательность литералов. Например, регулярное выражение:

INTEGER

составлено только из обычных символов и во входном потоке ему соответствует идентичная последовательность букв, которая образует слово ***INTEGER***. Хотя в этом регулярном выражении отсутствуют знаки операций, его следует формально интерпретировать как результат применения регулярной операции *конкатенации*, которая обеспечивает слияние заданных букв в одно слово.

ЭКРАНИРОВАНИЕ МЕТАСИМВОЛОВ

В практике лексического анализа часто встречается ситуация, когда в регулярном выражении какие-либо из метасимволов должны использоваться как литералы. В этом случае их нужно *экранировать* либо индивидуально с помощью символа *обратной дробной черты* (****), либо в виде последовательности, заключенной в *кавычки* (**"**).

В частности, кавычки указывают, что заключенная в них последовательность должна рассматриваться как обычные текстовые символы. Например, следующее регулярное выражение:

```
ALPHA"++"
```

удовлетворяет входной последовательности *ALPHA++*, где метасимволы, обозначенные знаком плюс, теряют свой специальный смысл, потому что их экранируют кавычки. Здесь, следует обратить внимание, на то, что экранироваться может как вся строка, так и часть строки. В данном случае экранировать пять обычных символов (*ALPHA*) в начале этой строки нет необходимости. В общем случае экранирование обычных текстовых символов необязательно, хотя никакого вреда не приносит. Например, следующее регулярное выражение полностью аналогично предыдущему:

```
"ALPHA++"
```

Следует отметить, что кавычки обычно применятся для группового экранирования метасимволов. Отдельный метасимвол будет интерпретироваться как литерал, если в регулярном выражении ему предшествует знак обратной дробной черты (****). Например, в следующем регулярном выражении знак обратной дробной черты экранирует специальный смысл метасимвола ***:

```
ALPHA\*BETA
```

Еще один пример индивидуального экранирования иллюстрирует следующее регулярное выражение где обратная дробная черта экранирует метасимвол **/**:

```
CAD\/CAM
```

Это регулярное выражение обеспечивает распознавание во входном потоке распространенной аббревиатуры **CAD/CAM**, которая часто встречается в научно-технической литературе по системам автоматизированного проектирования.

Механизм экранирования может быть использован также при необходимости вставки в регулярное выражение символа *пробела*. По определенным причинам, связанным с особенностями обработки регулярных выражений генератором **LEX**, символ пробел не может быть непосредственно включен в регулярное выражение как любой другой текстовый символ, например, буква или цифра. Поэтому любые пробелы, за исключением указанных в классах символов (смотрите ниже), должны экранироваться как, например, в следующем регулярном выражении, где метасимволы кавычек необходимы для вставки пробела между словами *regular* и *expression*:

```
regular" "expression
```

Аналогичный результат предоставляет следующее регулярное выражение, где экранирование пробела обеспечивает метасимвол обратной дробной черты:

```
regular\ expression
```

КОДЫ СИМВОЛОВ И ЛИТЕРАЛЬНЫЕ КОНСТАНТЫ

Экранирующий символ обратной дробной черты также применяется в регулярных выражениях, чтобы специальным образом распознавать несколько наиболее часто употребляемых обозначений литеральных констант языка программирования **C**. В частности, литеральная константа **\n** обозначает символ перевода строки, а **\t** - символ горизонтальной табуляции. Имеется также возможность непосредственно специфицировать в регулярном выражении код ASCII любого символа в системе счисления по основанию **8**, указав его после символа обратной дробной черты. Например, в следующем регулярном выражении для обозначения пробела между словами *Lexical* и *analyzer* используется его ASCII-код, равный **40** в системе счисления по основанию **8**:

```
Lexical\040analyzer
```

Литеральные константы и ASCII-коды символов обычно применяются в регулярных выражениях для обозначения неотображаемых символов, которые проблематично выразить иным способом. В частности, служебный символ *Escape*, который обычно используется для формирования управляющих последовательностей, можно вставить в регулярное выражение, непосредственно специфицируя его ASCII-код, равный **033** в системе счисления по основанию **8**.

Например, следующее регулярное выражение обозначает *Escape последовательность*, которая обеспечивает очистку экрана консоли в режиме прозрачного ввода терминального интерфейса **OS UNIX** и устанавливает текстовый курсор в левый верхний угол экрана:

```
\033\[2J
```

ЯКОРНЫЕ МЕТАСИМВОЛЫ

Несмотря на то, что без обычных текстовых символов в большинстве случаев нельзя получить практически полезные конструкции, основное значение при проектировании регулярных выражений имеют метасимволы, обозначающие регулярные операции. Вероятно, простейшими регулярными операциями, которые поддерживает генератор **LEX**, являются *циркумфлекс* (**^**), и *доллар* (**$**). Эти метасимволы используются, соответственно, для обозначения *начала* и *конца* строки входного потока.

Например, следующее регулярное выражение может быть использовано для поиска директивы **#define** препроцессора системы программирования **C**, которая может находиться только в началах строк исходного текста программы:

```
^#define
```

С другой стороны, следующее регулярное выражение может быть полезно для поиска последовательности символов, в частности слова ***cat***, которое обозначает популярную команду конкатенации **OS UNIX**, в конце входной строки:

```
cat$
```

Если необходимо распознавать входную строку, содержащую только одно слово, например, ***cat***, можно применить следующее регулярное выражение:

```
^cat$
```

В некоторых случаях полезно иметь возможность идентифицировать пустые строки входного потока. Один из возможных способов спецификации пустой строки предоставляет следующее регулярное выражение:

```
^$
```

Следует отметить, что специфика метасимволов **^** и **$** заключается в том, что они совпадают с определенной позицией текста, в частности, действенны только на концах регулярного выражения. По этой причине метасимволы **^** и **$** иногда называют *якорями* или якорными метасимволами. Большая часть остальных метасимволов предназначена для обработки текста без привязки к фиксированной позиции входной строки.

ВЫБОР АЛЬТЕРНАТИВ

Если якорные метасимволы относительно редко встречаются в конструкциях регулярных выражений и поддерживаются далеко не во всех диалектах регулярных выражений, то одной из наиболее популярных регулярных операций является операция *объединения*. Это классическая регулярная операция, которая поддерживается большинством программных продуктов, ориентированных на обработку регулярных выражений. Для ее обозначения в регулярных выражениях используется метасимвол вертикальной черты **|**, который имеет аналогичный смысл логического оператора *ИЛИ*, например, в языках программирования **C**, **C++**, **C#** и **Java**.

В алгебре регулярных выражений эта регулярная операция обеспечивает выбор любой из возможных альтернатив. Она позволяет объединить несколько регулярных фрагментов в общее регулярное выражение, которое специфицирует возможность совпадения данных входного потока с любой из своих компонент.

Пусть, например, имеется следующие два регулярных выражения, обозначающих уменьшительное (*Bob*) и полное (*Robert*) английские имена:

```
Bob
```

и

```
Robert
```

Операция объединения позволяет связать их в одно общее регулярное выражение, которое обеспечивает совпадение с любым из входных слов, *Bob* или *Robert*. Объединенное регулярное выражение имеет следующий вид:

```
Bob|Robert
```

Оно отражает семантическую эквивалентность слов *Bob* и *Robert*, которая является естественной, если рассматривать их как английские имена.

КЛАССЫ СИМВОЛОВ

Концептуально близкой к рассмотренной конструкции объединения является еще одна регулярная операция, которая обеспечивает выбор любого символа из заданного *класса символов*. Данная операция применяется, когда необходимо задать набор символов, которые могут находиться в определенной позиции входной строки. Для обозначения класса символов применяются метасимволы, образующие пару квадратных скобок `[]`. Внутри квадратных скобок должно быть задано множество символов, любой из которых может присутствовать в данной позиции входной строки. При этом следует учитывать, что внутри квадратных скобок класса символов большая часть метасимволов игнорируется. Специальными являются только три символа: *обратная дробная черта* (`\`) , *дефис* (`-`) и *циркумфлекс* (`^`). Причем метасимвол (`^`) имеет различный специальный смысл внутри класс символов и вне его границ.

Генератор **LEX** поддерживает различные способы спецификации множества символов класса. Можно специфицировать *перечислительный*, *интервальный* и *инвертированный* классы символов. Допустимо также комбинировать различные способы спецификации в пределах одного класса символов.

Наиболее простой, но не всегда возможный и рациональный способ – перечислить допустимые символы класса внутри квадратных скобок. Например, необходимо выделять во входном потоке слова *GRAY* или *GREY*, которые почти одинаковы, но отличаются только одной буквой. Для решения этой проблемы может быть применено следующее регулярное выражение, где используется символьный класс **`[EA]`**, состоящий из букв ***E*** и ***A***:

```
GR[EA]Y
```

Это регулярное выражение позволяет интерпретировать входной поток следующим образом: найти символ ***G***, за которым следует символ ***R***, после которого должен быть символ ***E*** либо символ ***A***, и все это завершается символом ***Y***. Кажется целесообразным отметить, что в данном случае вместо класса символов можно применить операцию объединения и построить следующее регулярное выражение:

```
GRAY|GREY
```

Это регулярное выражение также как и предыдущая конструкция с классом символов обеспечивает выбор любого из двух допустимых альтернативных вариантов входного слова: *GRAY* или *GREY*.

Таким образом, оба рассмотренных регулярных выражения гарантируют аналогичную обработку входного потока. Однако этот пример не может быть распространен на общий случай по следующей причине. Класс символов всегда совпадает ровно с одним символом, каким бы длинным или коротким не был список допустимых литералов. С другой сторону операция объединения может содержать альтернативы произвольной длины, совершенно не связанные между собой длиной текста. Например, следующее регулярное выражение образуется объединением трех альтернатив различной длины, которые лексически не связаны друг с другом:

```
1\.000\.000|million|”thousand thousand”
```

Кроме того, конструкция класса символов позволяет реализовать такие возможности выбора, которые не доступны для операции объединения, в частности, спецификация *интервальных* и *инвертированных* символьных классов.

Интервальный класс символов удобно применять, когда коды ASCII символов класса образуют непрерывную возрастающую последовательность. В этом случае класс символов можно задать в форме интервала, где первый и последний символы разделяет дефис, который в таком контексте становится метасимволом класса. Например, следующее регулярное выражение определяет класс символов, состоящий из строчных латинских букв:

```
[a-z]
```

При этом в классе символов может быть задано произвольное число интервалов. Например, следующее регулярное выражение образует класс символов, в котором специфицировано три интервала:

```
[0-9a-fA-F]
```

Это регулярное выражение предоставляет сокращенную форму записи следующей перечислительной спецификации класса символов:

```
[0123456789abcdefABCDEF]
```

Оба рассмотренных регулярных выражения эквивалентны и могут быть полезны для обработки чисел в позиционной системе счисления по основанию **16**.

Как отмечалось выше, допустимо комбинировать различные формы спецификации в пределах одного класса символов. Например, следующее регулярное выражение определяет комбинированный класс символов, который состоит из цифр, латинских букв и знака подчеркивания:

```
[0-9a-zA-Z_]
```

Следует отметить, что диапазоны символов и отдельные литералы в классе могут задаваться в любом порядке. В частности, следующее регулярное выражение эквивалентно предыдущему:

```
[0-9a-z_A-Z]
```

Использование знака дефиса между двумя символами, не являющимися только прописными, только строчными или только цифрами, зависит от реализации и обычно приводит к выдаче предупреждающего сообщения. Если символ дефис должен входить в определяемый класс, его нужно поместить или первым или последним. Например, следующее регулярное выражение удовлетворяют любой цифре, а также знакам *плюс* и *минус*:

```
[-+0-9]
```

В данной транскрипции, также как за границами класса символов, дефис интерпретируется как обычный литерал.

В некоторых случаях более удобно задать символьный класс, указав символы, которые не могут принадлежать ему. Для специфицирования таких инвертированных классов применяется метасимвол **^** (циркумфлекс), который должен быть первым символом после открывающей квадратной скобки. Его использование иллюстрирует следующее регулярное выражение:

```
[Qq][^u]
```

Это регулярное выражение может быть полезно для поиска слов английского языка, где после буквы ***q*** или ***Q*** стоит любая буква, кроме ***u***. Вряд ли такие слова могут быть найдены, если исключить из рассмотрения специальные термины, имена или географические названия.

Аналогичным образом метасимвол **^** позволяет инвертировать интервальный класс, когда целесообразно исключить диапазон символов. Например, следующее регулярное выражение удовлетворяет любому символу, не являющемуся цифрой и прописной или строчной латинской буквой:

```
[^0-9a-zA-Z]
```

Необходимо уточнить, что метасимвол циркумфлекс имеет другой специальный смысл за пределами класса символов, где он обеспечивает привязку к началу строки. Следует также учитывать, что в классе символов он теряет свой специальный смысл и интерпретируется как обычный текстовый символ, когда занимает в квадратных скобках любую позицию, кроме первой.

Циркумфлекс и дефис – типичные примеры множественной интерпретации метасимволов регулярных выражений. Их интерпретация зависит от контекста, в котором они используются в регулярном выражении. Единственным метасимволом регулярных выражений, который сохраняет без изменений свою специфику в классе символов является символ обратной дробной черты.

Обратная дробная черта (\) в классе символов сохраняет специальный смысл экранирующей металитеры для любого символа в квадратных скобках. Это означает, что любой символ в квадратных скобках, перед которым стоит метасимвол обратной дробной черты, рассматривается буквально. По очевидным причинам в большинстве случаев экранирование в символьном классе не имеет практической значимости, кроме ситуации, когда в символьный класс нужно включить сам символ обратной дробной черты или литеральные константы, обозначающие, в частности, символы перевода строки и горизонтальной табуляции. Например, следующее регулярное выражение задает символьный класс, состоящий из управляющих символов горизонтальной табуляции и перевода строки:

```
[\t\n]
```

Особенности интерпретации метасимволов в символьных классах распространяются также на символ пробела, который хотя и не является метасимволом, но в то же время при вставке пробела в регулярное выражение его необходимо экранировать. Следует обратить внимание на то, что символ пробела не требуется экранировать в классе символов, как это необходимо за его пределами. Например, в следующем регулярном выражении для спецификации пробела используется символьный класс, множество допустимых символов которого состоит только из одного символа пробела:

```
Lexical[ ]analysis
```

Во многих практических случаях требуется задать предельно широкий класс символов, который допускает использовать в данной позиции входной строки любой символ, кроме символа *перевода строки*. Для достижения этого эффекта предусмотрен отдельный метасимвол, который обозначает знак *точка* (`.`). Отсутствие в этом классе символа перевода строки объясняется строчной организацией входного потока, при которой символ перевода строки является естественным ограничителем входной информации, поступающей из потока стандартного ввода.

Когда требуется выделить во входном потоке любой символ, включая символ перевода строки, можно использовать следующее регулярное выражение с конструкцией выбора:

```
.|\n
```

Завершая анализ возможностей символьных классов, следует отметить, что классы символов можно рассматривать как своеобразный мини-язык регулярных выражений. Правила, определяющие состав и функции поддерживаемых метасимволов, внутри и за пределами класса символов полностью различны.

КВАНТИФИКАТОРЫ

В большинстве практически интересных случаев требуется специфицировать в регулярном выражении возможность повторения символов входной строки. Регулярные операторы, которые обеспечивают эту возможность, обычно называют *квантификаторами*. Генератор **LEX** поддерживает четыре квантификатора, которые обозначают метасимволы:

```
? * + {}
```

В регулярном выражении квантификаторы указываются после литералов, повторение которых они обозначают. Для каждого квантификатора существует минимальное количество экземпляров символов текста, которые он должен обязательно найти, и максимальное количество повторений, больше которого он даже не пытается искать. Предельные значения количества повторений различны для разных квантификаторов. Однако, при обработке входной информации, любой квантификатор руководствуется *принципом максимального совпадения*, то есть пытается найти максимальный по длине фрагмент входного текста, который соответствует регулярному выражению, где он специфицирован.

Простейшие возможности квантифицирования входного текста предоставляет метасимвол `(?)`, который указывает, что предшествующий литерал регулярного выражения может повторяться *ноль* или *один* раз. Это квантификатор применяется, когда нужно указать, что присутствие предшествующего символа в данной позиции входного текста необязательно.

Допустим, необходимо искать во входном потоке слова *color* или *colour*. Эти слова почти одинаковы и отличаются только одной буквой ***u***. Следующее регулярное выражение позволяет найти любой из указанных вариантов слова:

```
colou?r
```

В данном случае метасимвол **(?)** обозначает, что предшествующая ему буква ***u*** является необязательным символом. Она может находиться в данной позиции текста, но ее наличие не требуется для успеха совпадения входного слова с заданным регулярным выражением. Следует отметить, что эквивалентный результат позволяет обеспечить операция объединения в следующей конструкции выбора:

```
color|colour
```

Более богатыми возможностями обладает квантификатор *****, который обозначает, что во входном потоке может находиться любое, в том числе равное нулю, количество экземпляров предшествующего ему символа регулярного выражения. Это одна из классических базовых операций, которая известно в теории формальных языков как *итерация* или *замыкание Клини*.

Одно из многочисленных практических приложений операции итерации иллюстрирует следующее регулярное выражение, которому соответствуют любые натуральные числа:

```
[1-9][0-9]*
```

В этом регулярном выражении квантификатор ***** применяется к классу символов **[0-9]**, обеспечивая совпадение с цифровой последовательностью любой, в том числе *нулевой* длины. Класс символов **[1-9]** гарантирует наличие первой цифры натурального числа. Из диапазона символов этого класса исключена цифра **0**, с которой, очевидно, не может начинаться не может начинаться натуральное число. Это регулярное выражение также возможность совпадения исключает совпадения с числом **0**, которое не принадлежит натуральному ряду чисел. Следует отметить, что в данном случае весьма проблематично построить эквивалентное регулярное выражение, используя оператор объединения.

Похожими возможностями обладает квантификатор +, который также как метасимвол *****, обеспечивает многократное повторение элементов входного потока, но гарантирует наличие хотя бы одного экземпляра, удовлетворяющего регулярному выражению. В теории формальных языков эту регулярную операцию часто называют *позитивным замыканием Клини*. Она может быть успешно использована, например, для спецификации символьных разделителей информационных структур. Обычно в разделителях допускается произвольное количество разделяющих символов, но обязательно должен быть хотя бы один. В частности, следующее регулярное выражение позволяет идентифицировать разделители из произвольного числа символов пробела и горизонтальной табуляции:

```
[ \t]+
```

Следует отметить, что для данного случая можно построить эквивалентное регулярное выражение, используя квантификатор *****, которое имеет следующий вид:

```
[ \t][ \t]*
```

Кроме рассмотренных одно-символьных квантификаторов генератор **LEX** предоставляет специальную мета-последовательность, которая позволяет конкретно задавать наименьшее обязательное и наибольшее допустимое число экземпляров повторений. Эта конструкция называется *интервальным квантификатором* и имеет следующий формат:

```
R{N,M}
```

В этой конструкции, параметры ***N*** и ***M***, указанные в фигурных скобках, задают целые неотрицательные десятичные числа, которые устанавливают, соответственно, минимальное и максимальное количество повторений во входном потоке символа или фрагмента регулярного выражения, обозначенного литерой ***R*** перед фигурными скобками. В конструкции интервального квантификатора значение первого целочисленного параметра не должно превосходить величины второго целочисленного параметра, а литерал, допустимый диапазон повторений которого они определяют, может быть любым символом. Наиболее часто используется вариант применения интервального квантификатора, когда заданное минимальное количество повторений строго меньше максимального **(N < M)**. Такая интервальная конструкция удобна, например, для распознавания идентификаторов переменных и констант в исходных текстах программ.

Синтаксис большинства современных языков программирования допускает идентификаторы, образованные последовательностью из алфавитно-цифровых символов и знака подчеркивания, которая не может начинаться с цифры. Распознаваемая длина идентификатора часто огранивается **32**-мя символами. Перечисленным условиям удовлетворяет следующее регулярное выражение:

```
[a-zA-Z_][ a-zA-Z_0-9]{0,31}
```

В этом регулярном выражении первый символьный класс определяет возможные варианты первой литеры идентификатора. Второй символьный класс вместе с суффиксом интервального квантификатора задает остальные литеры идентификатора в количестве от **0** до **31** экземпляров.

Следует отметить, что если не требуется ограничивать максимальную длину идентификатора, то в рассмотренном регулярном выражении нужно заменить интервальный квантификатор **{0,31}** метасимволом *****:

```
[a-zA-Z_][a-zA-Z_0-9]*
```

Менее распространенным является вариант спецификации интервального квантификатора, где заданные минимальное и максимальное значения количества повторений равны **(N = M)**. Это позволяет задавать в регулярном выражении фиксированное количество повторений.

Например, следующее регулярное выражение определяет огромное целое число, равное единице с 24-мя нулями (10^{24}), которое согласно представлениям современной астрофизики определяет размер видимой части Вселенной, выраженный в километрах:

```
10{24,24}km
```

Следует отметить, что кроме применения в конструкции интервальных квантификаторов, метасимволы фигурных скобок используются генератором **LEX** для обозначения *регулярных определений*. Регулярные определения предоставляют возможность именовать фрагменты регулярных выражений и конструировать другие регулярные выражения с использованием этих имен в *фигурных скобках*, как если бы это были обычные текстовые символы. Обычно они применяются для упрощения записи сложных регулярных выражений или когда один и тот же регулярный фрагмент присутствует в нескольких регулярных выражениях.

Например, для спецификации инструкции *инкремента* языка программирования **C**, которая обозначается парой знаков *плюс* (**++**) после идентификатора переменной, может быть предложено следующее регулярное выражение:

```
{IDENT}\+\+
```

В этой спецификации присутствует регулярное определение *IDENT*, которое может обозначать любое из двух рассмотренных выше регулярных выражений для идентификатора. Генератор **LEX** автоматически подставляет соответствующий регулярный фрагмент, вместо регулярного определения в фигурных скобках. В результате подстановки получается, например, следующее регулярное выражение:

```
[a-zA-Z_][a-zA-Z_0-9]*\+\+
```

Следует отметить, что концепция регулярных определений, принятая в генераторе **LEX**, напоминает технологию использования макроопределений, которая реализуется, например, в системе программирования **C** директивой предпроцессорной обработки **#define**, в частности, для символического обозначения констант.

Однако целесообразно помнить, что обработка регулярных определений не поддерживается в большинстве других диалектов регулярных выражений и является специфической особенностью генератора **LEX**. Еще одно важное расширение стандартного аппарата регулярных выражений, которое предоставляет генератор **LEX**, связано с поддержкой возможности обработки входной информации в *произвольно заданном контексте*.

ОБРАБОТКА КОНТЕКСТА

Рассмотренные выше якорные операторы **^** и **$**, обеспечивают поиск информации только в контексте начала и конца входной строки. Однако генератор **LEX** поддерживает более широкие возможности распознавания лексем входного потока с учетом произвольного правого или левого контекста. В этой связи якорные операторы следует рассматривать как частный случай контекстной обработки входного потока.

Когда необходимо анализировать входной текст с учетом произвольного заданного *правого контекста*, в регулярном выражении следует использовать метасимвол *дробной черты* **/**, который обозначает регулярный оператор для обработки правого контекста. В регулярном выражении перед метасимволом дробной черты задается требуемая спецификация распознаваемой лексемы, а после него указывается правый контекст, в котором лексема должна употребляться во входном потоке. При этом следует учитывать, что после обработки лексемы ее правый контекст остается во входном потоке. В общем случае регулярное выражение, чувствительное к правому контексту, имеет следующий формат:

```
REGULAR/CONTEXT
```

В этой конструкции *CONTEXT* обозначает правый контекст, с учетом которого необходимо интерпретировать входной поток по регулярному выражению *REGULAR*.

В практических приложениях анализ правого контекста полезен при лексической обработке различных ограничителей информации, которые состоят из нескольких символов. Например, синтаксис языка программирования **Pascal** требует использовать операторные ограничители, образованные символьными парами **(*** и ***)**, для обозначения начала и конца комментария, соответственно. В тоже время символ ***** внутри комментария не должен интерпретироваться как начало или конец его операторного ограничителя. Решение этой проблемы обеспечивает следующее регулярное выражение:

```
\*/[^)]
```

Данное регулярное выражение интерпретирует символ ***** как первый символ операторного ограничителя конца комментария, только когда во входном потоке за ним следует символ закрывающей круглой скобки. Это позволяет включать символы ***** в текст комментария, не опасаясь, что они будут распознаваться как часть завершающего его операторного ограничителя.

Кроме обработки правого контекста генератор **LEX** предоставляет возможность указать в регулярном выражении необходимость анализа распознаваемых лексем с учетом их *левого контекста* во входном потоке. Левый контекст специфицируется в форме алфавитно-цифровой *метки предусловия*, которая указывается в *угловых скобках* **<>** перед регулярным выражением.

Метка предусловия обозначает логическое состояние процесса лексического анализа, в котором входной поток сопоставляется с данным регулярным выражением и/или интерпретируется определенным образом. В общем случае лексическая конструкция, которая обеспечивает анализ входного потока, чувствительный к левому контексту распознаваемых лексем, должна имеет следующий формат:

```
<STATE>REGULAR
```

В этой конструкции *REGULAR* обозначает регулярное выражение, которое становится актуальным, если выполнено предусловие, заданное меткой состояния *STATE*. Лексический анализатор, в котором предусмотрена такая инструкция, будет обрабатывать входной поток по регулярному выражению *REGULAR* только тогда, когда выполнено предусловие, заданное меткой *STATE* в угловых скобках.

Иногда требуется осуществлять обработку входного потока по заданному регулярному выражению при выполнении любого из нескольких предусловий. В этом случае метки соответствующих предусловий должны быть перечислены перед регулярным выражением в угловых скобках через *запятую*. Например, следующая конструкция разрешает анализ входного потока по регулярному выражению *REGULAR*, когда выполнено любое из предусловий, которые обозначены метками **`STATE1`** и **`STATE2`**:

```
<STATE1,STATE2>REGULAR
```

Возможность анализа левого контекста позволяет придать большую практическую значимость рассмотренному выше регулярному выражению для обработки операторных ограничителей комментариев в исходных текстах программ, составленных на языке программирования **Pascal**. Очевидно, что символы **`*)`** следует интерпретировать как правый операторный ограничитель комментария, если предварительно во входном потоке по соответствующему регулярному выражению была обнаружена и соответствующим образом обработана пара символов **`(*`**, которая обозначают левый операторный ограничитель комментария. Эту проблему решает, например, следующее регулярное выражение с левым контекстом:

```
<COMMENT>\*/[^)]
```

В этой конструкции метка предусловия *COMMENT* обозначает состояние лексического анализатора, когда во входном потоке было обнаружено начало комментария. Если предусловие по метке *COMMENT* не выполнено, лексический анализатор считает неактуальным последующее регулярное выражение для распознавания символов ***** внутри комментария и, следовательно, не будет интерпретировать входной поток согласно ему.

Следует отметить, что в этом примере, также как выше в спецификации общего формата регулярных выражений с левым контекстом, не рассматривается вопрос, каким образом устанавливаются и задаются метки предусловий. Исходя из методических соображений, технология конструирования и использования регулярных выражений с левым контекстом, будет рассмотрена ниже, в разделе правил спецификации лексем.

ГРУППИРОВКА И ОГРАНИЧЕНИЕ РЕГУЛЯРНЫХ ФРАГМЕНТОВ

Существенно расширить возможности конструирования регулярных выражений позволяет применение в них метасимволов *круглых скобок* **()**. Аналогично их функциям в арифметических или алгебраических конструкциях, в регулярных выражениях круглые скобки позволяют изменять естественный приоритет операций. Обычно они используются с целью ограничения области действия регулярных операций с низким приоритетом или группировки фрагментов регулярных выражений для применения к ним регулярных операций с высоким приоритетом. В частности, достаточно низкий приоритет имеет оператор объединения. Поэтому область действия оператора объединения распространяется на предельно длинные альтернативы из литералов, которые находятся слева и справа от метасимвола выбора **|**. Вставка круглых скобок в регулярное выражение позволяет ограничить область действия оператора объединения альтернативами, которые оказались внутри них. Например, следующее регулярное выражение обеспечивает распознавание во входном потоке либо альтернативы *this and***,** либо альтернативы *or that*, информативность которых близка к нулю:

```
this[ ]and|or[ ]that
```

Однако вставка круглых скобок превращает его в следующее, потенциально более полезное регулярное выражение:

```
this[ ](and|or)[ ]that
```

Это регулярное выражение позволяет обнаруживать во входном потоке, очевидно, более информативные словосочетания *this and that* и *this or that*, чем рассмотренный выше вариант без круглых скобок. Так происходит потому, что в данном случае круглые скобки ограничивают область действия оператора объединения.

Анализируя возможности скобочных регулярных выражений, следует обратить внимание на случай использовании конструкции выбора в сочетании с якорями **^** и **$**, которые имеют еще более низкий приоритет, чем операция объединения. Такое соотношение приоритетов обычно избавляет от необходимости применения круглых скобок при совместном использовании этих операторов в регулярном выражении. Например, при обработке файлов электронной почты бывает важно находить строки почтового сообщения, которые содержат директивы *From:* или *Subject:* в начале строки. На первый взгляд кажется, что требуемое соответствие обеспечивает только следующее скобочное регулярное выражение, где конструкцию выбора ограничивают круглые скобки, чтобы гарантировать поиск требуемых альтернатив исключительно в началах строк почтового файла:

```
^(From|Subject):
```

Однако, из-за различного приоритета операций **^** и **|**, это регулярное выражение эквивалентно следующей конструкции выбора, где отсутствуют круглые скобки:

```
^From:|Subject:
```

В этом регулярном выражении якорный префикс **^** по-прежнему относится к каждой из альтернатив, потому что приоритет операции оператора циркумфлекс меньше, чем приоритет оператора объединения. Таким образом, оба рассмотренных регулярных выражения гарантируют необходимый результат поиска директив *From:* и *Subject:* в почтовом сообщении. Однако более рациональный второй вариант, где отсутствуют круглые скобки, представляется более предпочтительным.

Более высокий приоритет, чем операция объединения и якорные операторы, имеют квантификаторы, поэтому любой из них применяется непосредственно к литералу, который находится слева от метасимвола, обозначающего квантификатор. Когда требуется применить квантификатор к регулярному выражению, оно должно быть заключено в круглые скобки.

Использование этого технического приема иллюстрирует следующий пример. При обработке различных финансовых документов часто бывает необходимо обнаруживать записи денежных сумм в долларах с необязательным указанием центов. Одно из возможных решений этой задачи предоставляет следующее регулярное выражения:

```
\$[0-9]+(\.[0-9][0-9])?
```

Это регулярное выражение состоит из трех частей: экранированный знак доллара **\$**, последовательность целых десятичных цифр для идентификации количества долларов в денежной сумме **[0-9]+** и необязательный регулярный фрагмент **(\.[0-9][0-9])?**, содержащий экранированный символ точки, за которым следуют две десятичные цифры **[0-9][0-9]**, определяющие число центов. Для группировки центовой части регулярного выражения используются круглые скобки, чтобы получить возможность квантифицировать находящийся в них регулярный фрагмент как будто это один символ.

Еще один пример, где круглые скобки используются в целях группировки различных фрагментов регулярного выражения, предоставляет следующая конструкция, которая может быть полезна для распознавания бинарных последовательностей, где чередуются символы **0** и **1**:

```
(01)*|(10)*|0(10)*|1(01)*
```

Хотя в этом регулярном выражении вместе с квантификаторами ***** присутствуют операторы объединения, круглые скобки используются исключительно для группировки литералов **0** и **1** в бинарные пары. К этим парам применяются квантификаторы *****, чтобы развернуть их в бинарные последовательности, где чередуются символы **0** и **1**. Конструкция выбора используется, чтобы объединить квантифицированные фрагменты в общее регулярное выражение, которое позволяет распознавать во входном потоке все возможные варианты чередующихся бинарных последовательностей.

В частности, две первые альтернативы обеспечивают поиск бинарных последовательностей, у которых первый и последний символы различны, например, **010101** или **101010**. Две другие альтернативы позволяют находить бинарные последовательности, где первый и последний символ совпадают, например, **010** или **101**.

Следует отметить, что рассмотренному регулярному выражению эквивалентны следующие две более лаконичные конструкции:

```
1?(01)*0?
```

и

```
0?(10)*1?
```

В этих регулярных выражениях нет операторов объединения, а круглые скобки по-прежнему выполняют функцию группировки бинарных пар для применения квантификатора *****.

Разумеется, возможна ситуация, когда в регулярное выражение необходимо включить круглые скобки, как для группировки, так и для ограничения его регулярных фрагментов. Комбинированное применение круглых скобок для обеспечения этих двух функций демонстрирует следующее регулярное выражение:

```
((0|1)(0|1)(0|1))*
```

Оно обозначает множество бинарных векторов, длина каждого из которых кратна трем. В частности, это регулярное выражение позволяет обнаружить во входном потоке, бинарные последовательности `010` и `111011`, длиной, соответственно, **3** и **6** символов.

В рассматриваемом регулярном выражении внутренние пары круглых скобок выполняют функцию ограничения области действия операции объединения для трех конструкций выбора. Внешние скобки обеспечивают группировку конкатенации трех внутренних скобочных фрагментов регулярного выражения для комплексной обработки их квантификатором *****.

СТРУКТУРНЫЙ АНАЛИЗ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

До сих пор в качестве примеров умышленно рассматривались простые регулярные выражения, чтобы использование в них метасимволов и литералов было предельно очевидным. Однако в большинстве практически интересных случаев для построения регулярных выражений необходим структурный анализ лексем входного потока. Структурный подход иллюстрируют несколько примеров конструирования регулярных выражений для распознавания времени и даты, которые рассмотрены ниже.

В первом примере рассматривается проблема конструирования регулярного выражения для поиска записей *даты*, которые состоят из сокращенного названия *месяца года* и *числа месяца*. Допустим, требуется составить регулярное выражение для распознавания *январских* дат, например, в следующем формате: **Jan 31**, **Jan 7** или **Jan 07**. На первый взгляд кажется, что проблему легко решает следующее регулярное выражение:

```
Jan[ ][0123][0-9]
```

Однако это регулярное выражение теряет корректные даты, где число задано одной цифрой без лидирующего нуля, например, **Jan 7.** Кроме того, оно будет находить несуществующие даты, например, **Jan 00** или **Jan 32**. Один из способов корректного поиска даты заключается в том, чтобы разделить диапазон чисел января на три группы, которые соответствуют декадам. Каждая декада января должна специфицироваться своим фрагментом регулярного выражения. Спецификации декад объединяются в следующую итоговую конструкцию выбора:

```
Jan[ ](0?[1-9]|[12][0-9]|3[01])
```

В этом регулярном выражении первая альтернатива **0?[1-9]** обеспечивает поиск дат первой декады от **1**-го до **9**-го января, допуская запись чисел с лидирующим нулем. Вторая альтернатива **[12][0-9]** позволяет распознавать даты в диапазоне чисел с **10**-го по **29**-е января. Наконец, последняя альтернатива находит завершающие даты **30**-е и **31**-е января. Таким образом, этим регулярным выражением оказываются покрыты все декады и числа января.

Существует другое решение задачи поиска даты, в котором используется иной принцип декомпозиции диапазона чисел января. Его представляет следующее регулярное выражение:

```
Jan[ ](31|[123]0|[012]?[1-9])
```

Это регулярное выражение также объединяет три альтернативы. Первая альтернатива допускает только одно число, **31**-е января. Вторая альтернатива специфицирует регулярное выражение для поиска дат, в которых числа кратны **10**-ти, то есть **10**-е, **20**-е и **30**-е января. Последняя альтернатива соответствует всем остальным датам января.

На первый взгляд кажется, что рассмотренные регулярные выражения легко приспособить для лексического анализа даты в любом месяце года, если заменить обозначение января следующей конструкцией выбора:

```
(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
```

Однако в этом случае возникает опасность распознавания несуществующих дат, например, **Jun 31** (**31**-е июня), из-за того, что количество дней в разных месяцах года различно. Поэтому целесообразно построить различные альтернативы для групп месяцев равной продолжительности и объединить их с помощью конструкции выбора.

В частности, для распознавания дат месяцев *январь*, *март*, *май*, *июль*, *август*, *октябрь* и *декабрь*, с продолжительностью **31** день, может быть использована следующая альтернатива:

```
(Jan|Mar|May|Jul|Aug|Oct|Dec)[ ](0?[1-9]|[12][0-9]|3[01])
```

Соответствующая альтернатива для месяцев *апрель*, *июнь*, *сентябрь* и *ноябрь*, где только **30** дней, имеет следующий вид:

```
(Apr|Jun|Sep|Nov)[ ]([123]0|[012]?[1-9])
```

Наконец, для *февраля*, где может быть либо **28**, либо **29** дней, требуется составить следующую отдельную альтернативу:

```
Feb[ ](0?[1-9]|[12][0-9])
```

Для более лаконичной записи этих трех альтернатив можно ввести регулярные определения *DATE31*, *DATE30* и *FEBRUARY*, соответственно. Тогда их объединение с помощью конструкции выбора образует следующее регулярное выражение, которое обеспечивает поиск дат любого месяца года, независимо от продолжительности месяца:

```
{DATE31}|{DATE30}|{FEBRUARY}
```

В некоторых случаях необходимо обеспечить поиск дат, где кроме числа и месяца указан год. В общем случае год может представляться любыми натуральными числами. Регулярное выражение для спецификации натуральных чисел было рассмотрено выше, для иллюстрации применения квантификатора итерации. Таким образом, составление регулярного выражения для корректного распознавания даты в общем случае превращается в сложную проблему, особенно когда необходимо различать високосные и не високосные годы при анализе февральских дат.

Еще один пример, где для конструирования регулярного выражения применяется структурный подход, демонстрирует, каким образом может быть решена задача поиска во входном потоке записей *времени суток*. Допустим, во входном потоке требуется находить записи времени в **12**-ти часовом формате **AM/PM**, где часы, минуты и необязательные секунды, разделены символом двоеточия, например, **10:15:00**, **8:30** или **08:30:00**. Очевидно, что регулярное выражение, которому соответствует указанный формат времени суток, должно состоять из трех частей, которые специфицируют часы, минуты и необязательные секунды, соответственно.

Наиболее просто составить регулярное выражение для распознавания записи минут и секунд. Запись минут или секунд должна состоять из двух десятичных цифр, причем первая цифра ограничена диапазоном от **0** до **5**, а вторая цифра может быть любой. Таким образом, для распознавания записи минут или секунд можно рекомендовать следующий регулярный фрагмент, который образует конкатенация двух символьных классов:

```
[0-5][0-9]
```

Чтобы специфицировать регулярное выражение для распознавания часов, нужно рассмотреть два отдельных случая, когда запись часов состоит из двух и из одной значащей цифры. При этом необходимо учесть, что первая цифра записи часа из двух цифр может быть только **1**, а вторая цифра может представляться **0**, **1** или **2**. Когда запись часа состоит только из одной цифры, она может быть любой.

Объединение этих двух альтернатив в конструкции выбора образует следующий фрагмент регулярного выражения для распознавания записи часов:

```
1[012]|[1-9]
```

Конкатенация рассмотренных регулярных фрагментов для записей часов, минут и необязательных секунд позволяет составить следующее регулярное выражение для поиска времени суток в **12**-ти часовом формате **AM/PM**:

```
(1[012]|[1-9]):[0-5][0-9](:[0-5][0-9])?[ ](am|pm)
```

В этом регулярном выражении круглые скобки используются для ограничения области действия конструкций выбора и группировки необязательных символьных классов, которые обозначают секунды.

Допустим теперь, что во входном потоке требуется находить записи времени в **24**-часовом формате, где часы, минуты и необязательные секунды по-прежнему должны быть разделены символом двоеточия, например, **23:59:59**, **8:30** или **08:30**. Очевидно, что регулярное выражение, которому соответствует указанный формат времени суток, также должно состоять из трех частей, которые специфицируют часы, минуты и необязательные секунды, соответственно. Совершенно понятно, что фрагменты регулярных выражений для записи минут и секунд должны быть идентичны в обеих шкалах измерения времени суток. Однако регулярное выражение для записи часов должно строится на основе иных соображений. Чтобы специфицировать регулярное выражение для записи часов, естественно разделить сутки на три группы часов: *утро*, от **0** до **9**-ти часов, *день*, от **10**-ти до **19**-ти часов и *вечер*, от **20**-ти до **23**-х часов. Очевидно, что объединение этих трех альтернатив в регулярном выражении покрывает суточный диапазон в часах. Наиболее простое решение, которое отражает предложенную декомпозицию часов и спецификацию минут, реализует следующее регулярное выражение:

```
(0?[0-9]|1[0-9]|2[0-3]):[0-5][0-9](:[0-5][0-9])?
```

Аналогично рассмотренному выше регулярному выражению для поиска времени суток в **12**-ти часовом формате **AM/PM**, в этом регулярном выражении символьные классы после двоеточия специфицируют запись минут и необязательных секунд, а объединение альтернатив, ограниченное круглыми скобками, обеспечивает распознавание любого часа суток. Композиция первых двух альтернатив в круглых скобках на основании дистрибутивного закона позволяет получить следующее логически менее очевидное, но более компактное регулярное выражение, которое эквивалентно регулярному выражению, рассмотренному выше:

```
([01]?[0-9]|2[0-3]):[0-5][0-9](:[0-5][0-9])?
```

Формальную эквивалентность обоих регулярных выражений для времени суток доказывает следующая цепочка тождественных преобразований первых двух альтернатив исходного регулярного выражения:

```
0?[0-9]|1[0-9] = 0[0-9]|[0-9]|1[0-9] = (0|1)?[0-9] = [01]?[0-9]
```

Полученные регулярные выражения для распознавания даты месяца и времени суток могут быть дополнены спецификацией года, который в общем случае может задаваться любым натуральным числом, а затем объединены в следующую комплексную конструкцию:

```
{DATE}[ ]{TIME}[ ]{YEAR}
```

В этом регулярном выражении для сокращения записи используются регулярные определения *DATE*, *TIME* и *YEAR*, которые обозначают рассмотренные выше регулярные фрагменты, специфицирующие дату, время и год, соответственно. Оно позволяет обнаружить во входном потоке, например, следующую запись: **`Jan 1 00:00:00 1970`**, которая обозначает базовую точку отсчета, принятую в **OS UNIX** для измерения временных интервалов.

Обобщая полученные результаты, следует отметить, что рассмотренные примеры иллюстрируют применение принципа декомпозиции, который наиболее часто используется для составления регулярных выражений в сложных случаях, когда решение неочевидно и необходим структурный анализ проблемы. Принцип декомпозиции состоит в том, чтобы заменить решение сложной исходной проблемы рассмотрением различных частных альтернатив, которые затем объединяются в результирующее регулярное выражение с помощью конструкции выбора или конкатенации обязательных фрагментов.

НЕРЕГУЛЯРНЫЕ МНОЖЕСТВА

В заключение обзора регулярных выражений следует отметить, что их возможности далеко не безграничны даже для расширенных диалектов, которые используют современные инструментальные средства обработки символьной информации, в частности генератор **LEX**. Существуют формальные языки, множества слов которых не могут быть специфицированы с помощью аппарата регулярных выражений.

В частности, регулярные выражения не могут быть использованы для описания сбалансированных или вложенных конструкций, которые характерны для языков программирования высокого уровня. Например, множество всех слов из сбалансированных скобок не может быть задано регулярным выражением.

Регулярные выражения также нельзя применить для описания повторяющихся конструкций, где количество повторений подчиняется определенной закономерности. К этому классу относятся, например, различные множества слов, длины определенных фрагментов которых образуют бесконечную возрастающую последовательность целых чисел, где разности соседних элементов не ограничены в совокупности. Это означает, что в такой последовательности всегда можно найти пару соседних чисел, разность которых превосходит величину сколь угодно большого целого числа.

Такому условию удовлетворяет, например, последовательность квадратов, кубов или других степеней чисел натурального ряда. В частности, невозможно составить регулярную спецификацию для распознавания бинарных векторов, в которых количество нулей между двумя не стоящими рядом единицами постоянно удваивается. Таким свойством обладает, например, следующая бесконечная последовательность:

`101001000011100000001...`

Точно также нельзя построить регулярное выражение для распознавания бесконечной бинарной последовательности, где номера позиций единичных элементов являются квадратами натуральных чисел, а на все остальные позиции заполнены нулями. Начальный фрагмент такой последовательности для, например, пяти единичных элементов образует следующий бинарный вектор, где символы **`1`** находятся, соответственно, в позициях с номерами **`1`**, **`4`**, **`9`**, **`16`** и **`25`**:

`1001000010000001000000001`

Перечень примеров бесконечных последовательностей, для описания которых не удается построить регулярные выражения, можно легко расширить, включая в него любые бесконечные последовательности без повторяющихся фрагментов. Однако, несмотря на указанные ограничения области применения, формальный аппарат регулярных выражений предоставляет удобные инструментальные средства для обработки текстовых данных в подавляющем большинстве других практически интересных случаев.

КОНЕЧНЫЕ АВТОМАТЫ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

Согласно теореме Клини любому регулярному выражению можно поставить в соответствие *конечный автомат*, который является формальной моделью алгоритма распознавания лексем, обозначаемых данным регулярным выражением. В наиболее общих терминах конечный автомат-распознаватель определяется конечным множеством характерных для него *состояний* входного потока и *переходов* между ними. Изменение состояния происходит при получении символов входного потока из заданного *алфавита* в соответствии с *функцией переходов*, которая определяет возможные последующие состояния по входному символу и текущему состоянию. Среди возможных состояний выделяется *исходное* (*начальное*) и *заключительные* (*допускающие*) состояния в которых конечный автомат-распознаватель может находиться, соответственно, при начале и завершении обработки лексем входного потока. Если входная последовательность символов может порождать последовательность переходов, которая может переводить конечный автомат из начального состояния в одно из заключительных, то она считается *допускающей* и принадлежит распознаваемому им регулярному множеству.

Конечный автомат-распознаватель может быть реализован в детерминированном и недетерминированном варианте. *Детерминированный конечный автомат* (далее по тексту ***ДКА***) по любому входному символу допускает не более одного перехода из каждого состояния. Поэтому для любого входного символа может существовать только одно допускающее состояние, в которое он может переходить из текущего состояния. Напротив, *недетерминированный конечный автомат* (далее по тексту ***НКА***) допускает не более одного перехода из каждого состояния. Поэтому он одновременно может находиться в нескольких состояниях. Формально **ДКА** и **НКА** различаются значением функции переходов. В **ДКА** значением функции переходов может быть только одно состояние, а в **НКА** – множество состояний. В общем случае **НКА** более компактны и легче строятся, но **ДКА** имеет преимущество по быстродействию. По этой причине для распознавания лексем входного потока по регулярному выражению обычно реализуется **ДКА**.

В следующем примере рассматривается **ДКА** для распознавания лексем, обозначаемых регулярным выражением:

```
(00|11)*((01|10)(00|11)*(01|10)(00|11)*)*
```

Этому регулярному выражению удовлетворяют бинарные векторы с четным числом нулевых и четным числом единичных разрядов. Например, ему соответствует бинарный вектор **01001000**, который содержит две единицы и шесть нулей.

Очевидно, входной алфавит **ДКА**, реализующего это регулярное выражение, должны составлять символы **0** и **1**. Также вполне естественно использовать состояния данного **ДКА** для подсчета числа нулей и единиц входной последовательности по модулю **2**, чтобы фиксировать четное или нечетное число символов **0** или **1** было получено в каждый момент из входного потока. Поэтому актуальными могут быть только четыре состояния, которые можно охарактеризовать и обозначить следующим образом:

Q1 – получено *четное* число символов **0** и *четное* число символов **1**;
Q2 - получено *четное* число символов **0** и *нечетное* число символов **1**;
Q3 - получено *нечетное* число символов **0** и *нечетное* число символов **1**;
Q4 - получено *нечетное* число символов **0** и *четное* число символов **1**.

Состояние **Q1** одновременно является начальным и единственным допускающим. Начальным оно является потому, что до того как будут прочитаны какие-либо входные данные, количество полученных символов **0** и **1** равно нулю, а нуль есть четное число. Это же состояние является единственным допускающим, поскольку только оно в точности удовлетворяет условиям соответствия регулярному выражению, которое должен реализовывать данный **ДКА**. Чтобы завершить определение **ДКА** нужно задать функцию переходов между его состояниями. При этом, полезно учесть, что возможно всего **8** переходов по символам **0** и **1** между состояниями, которые различаются либо числом нулей, либо числом единиц. Таким образом, взаимообратные переходы при получении символа **1** имеют место в парах состояний **Q1<=>Q2** и **Q3<=>Q4**. Также взаимообратные переходы имеют место при получении символа **0** в парах состояний **Q1<=>Q4** и **Q2<=>Q3**. Эти наблюдения можно представить в виде следующей двумерной таблицы, обычно называемой *таблицей переходов*:

	0	1
Q1	**Q4**	**Q2**
Q2	**Q3**	**Q1**
Q3	**Q2**	**Q4**
Q4	**Q1**	**Q3**

Столбцы таблицы переходов обозначают символы входного алфавита, а строки соответствуют текущим состояниям **ДКА**. Элементы каждой строки указывают состояния **ДКА**, в которые он должен переходить из текущего состояния при получении соответствующих символов входного алфавита. В частности, из первой строки данной таблицы переходов следует, что получение символов **0** и **1** в начальном состоянии **Q1** переводит **ДКА** в состояния **Q4** и **Q2**, соответственно.

При распознавании входной последовательности по таблице переходов легко проследить изменения состояния **ДКА** с целью определить достигается или нет одно из допускающих состояний. В частности, для бинарного вектора **01001000** с четным числом нулей и единиц рассмотренный **ДКА** порождает следующую последовательность переходов, где каждый переход помечен вызывающим его символом входного алфавита:

```
   0     1     0     0     1     1     0     0
Q1 -> Q4 -> Q3 -> Q2 -> Q3 -> Q4 -> Q1 -> Q4 -> Q1
```

Эта последовательность переходов завершается допускающим состоянием **Q1**, следовательно, бинарный вектор **01001000** принадлежит регулярному множеству, распознаваемому рассмотренным **ДКА** и удовлетворяет приведенному выше регулярному выражению.

В заключение следует отметить, что рассмотренный неформальный способ конструирования **ДКА** для регулярного выражения основан на частном логическом исследовании проблемы и не является универсальным. Однако существуют формальные алгоритмы построения конечных автоматов по регулярному выражению. В частности, для получения **НКА** из регулярного выражения используется *построение Томпсона*, а последующее преобразование **НКА** в **ДКА** обеспечивает *метод построения подмножеств состояний*. Для непосредственного получения **ДКА** по регулярному выражению применяется алгоритм, основанный на построении *синтаксического дерева* регулярного выражения.

Перечисленные формальные методы реализованы во всех программах генераторов лексических анализаторов, где для спецификации лексем входного потока используются регулярные выражения. В частности, генератор **LEX** автоматически конструирует **ДКА** для лексического анализатора на основе построения синтаксических деревьев регулярных выражений, которые заданы во входном файле спецификации лексем разработчиком лексического анализатора.

СПЕЦИФИКАЦИЯ ЛЕКСЕМ

СТРУКТУРА ФАЙЛА СПЕЦИФИКАЦИИ ЛЕКСЕМ

Вся исходная информация для построения лексического анализатора средствами генератора **LEX** должна быть сосредоточена в файле *спецификации лексем*, который имеет определенную внутреннюю структуру. В общем случае он может состоять из трех последовательных секций:

секции описаний,
секции правил,
секции подпрограмм.

Для разграничения этих секций в файле спецификации лексем используется специальный разделитель, который образует пара знаков процента %% в начале отдельной строки. Общий формат файла спецификации лексем имеет следующий вид:

```
        /* Секция описаний */
%%
        /* Секция правил */
%%
        /* Секция подпрограмм */
```

В частном случае секции описаний и подпрограмм могут отсутствовать. Секция правил обязательна, но может быть пустой. Поэтому минимальный корректный для генератора **LEX** файл спецификации лексем имеет следующий вид:

```
%%
```

В этом файле спецификации лексем нет описаний, нет правил и нет подпрограмм. По такому файлу спецификации лексем генератор **LEX** построит лексический анализатор, который только копирует без изменений содержимое входного потока стандартного ввода в выходной поток стандартного вывода. Очевидно, что в практическом смысле такой лексический анализ мало полезен, поэтому в общем случае, по крайней мере, секция правил не должна быть пустой. Принципы формирования перечисленных секций файла спецификации лексем рассмотрены ниже.

СЕКЦИЯ ОПИСАНИЙ

Все, что находится в файле спецификации лексем до первой разделительной пары знаков процента %%, генератор **LEX** считает относящимся к *секции описаний*. Эта секция содержит декларативную информацию для секций правил и подпрограмм. В общем случае в секции описаний могут быть декларированы:

блок описаний,
метки предусловий,
макросы регулярных определений,
размеры внутренних массивов программы лексического анализатора.

Перечисленные декларации должны начинаться с начальной позиции строк секции описаний. Только в этом случае они могут быть соответствующим образом интерпретированы генератором **LEX**. Любые строки секции описаний, которые содержат лидирующие символы пробела и табуляции, без обработки копируются в исходный код программы лексического анализатора. Это обычно используется, чтобы включать в секцию описаний необходимые комментарии, которые должны задаваться в формате языка программирования **C**. Аналогичным образом в секцию описаний можно включать объявления внешних переменных программы лексического анализатора или иные законченные конструкции языка программирования **C**, которые не обязаны начинаться с первой позиции строки.

В некоторых случаях важно иметь возможность включать в секцию описаний глобальные спецификации, которые не являются конструкциями генератора **LEX**, но должны начинаться с первой позиции строки, например, директивы препроцессора системы программирования **C**, в частности, директивы **#if**, **#else**, **#ifdef**, **#ifndef**, **#endif**, **#line**, **#define** и **#include**. Такие глобальные спецификации должны быть размещены между служебными ограничителями **%{** и **%}**, которые располагаются в отдельных строках секции описаний, начиная с первой позиции. Все, что находится между ними, образует *блок описаний*, любые инструкции которого имеют глобальный характер для остальных секций файла спецификации лексем и без изменений копируются генератором **LEX** в исходный код проектируемой программы лексического анализатора. Следует отметить, что некоторые реализации генератора **LEX** также допускают включать блок описаний в секцию правил файла спецификации лексем.

Пример оформления блока описаний демонстрирует следующий фрагмент программного кода, макросы и переменные которого могут быть полезны для обработки скобочных конструкций:

```
        /* Блок глобальных описаний для скобочных конструкций */
%{

#include "y.tab.h"
#define LEFTBRACKET      '('  /* код открывающей скобки */
#define RIGHTBRACKET     ')'  /* код закрывающей скобки */
extern int bracketcount;      /* декларация счетчика скобок */
int bracketcount = 0;         /* инициализация счетчика скобок */

%}
```

Этот блок описаний содержит директивы препроцессора системы программирования **C** **#include** и **#define**, которые применяются для подключения заголовочного файла **y.tab.h** и макроопределений скобочных символов *LEFTBRACKET* и *RIGHTBRACET* с целью последующего использования в секциях правил и подпрограмм файла спецификации лексем. В соответствии с правилами языка программирования **C** эти декларации должны начинаться с первой позиции строки и поэтому не могут быть специфицированы за пределами блока описаний. Следующее за ними инструкции объявления, определения и инициализации внешней целочисленной переменной *bracketcount* не обязательно должны начинаться с первой позиции строки. Поэтому эти инструкции могут быть расположены в любой строке за пределами блока описаний, после символов пробела или табуляции аналогично комментарию перед блоком. В данном случае они включены в блок описаний исходя из косметических соображений.

Если содержимое блока описаний и комментарии просто копируются в файл спецификации лексем, то конструкции остальных служебных директив секции описаний ориентированы на обработку генератором **LEX**. На практике наиболее часто используются директивы, которые декларируют метки предусловий и макросы регулярных определений для секции правил.

Метки предусловий используются для идентификации состояний лексического анализатора при обработке регулярных выражений с левым контекстом в секции правил. В секции описаний все необходимые метки предусловий должны быть объявлены служебной директивой **%START**, которая указывается с первой позиции строки.

Например, следующая декларация секции описаний объявляет метку предусловия *STATE* для последующего использования в секции правил:

```
%START STATE
```

Если необходимо задать несколько меток предусловий, то они должны быть перечислены после директивы %**START** через пробел. В частности, следующая декларация объявляет метки предусловий *STATE1* и *STATE2* для секции правил:

```
%START STATE1 STATE2
```

Следует отметить, что название этой служебной директивы допустимо задавать строчными буквами, например, **%start** или **%Start**, а также сокращать до одного символа, то есть, **%s** или **%S**. Обязательно только то, чтобы строка, содержащая данную директиву в любой допустимой форме, была первой строкой секции описаний.

Регулярные определения обычно применяются для упрощения и унификации записи регулярных выражений. Все необходимые регулярные определения специфицируются в секции описаний строками следующего вида:

```
MACRO     REGULAR
```

В этой строке **MACRO** обозначает произвольное алфавитно-цифровое имя, которое начинается с буквы и используется в дальнейшем как макрос для подстановки регулярного выражения **REGULAR** в другие регулярные выражения или определения секций описаний и правил. Генератор **LEX** гарантирует автоматическую подстановку значения регулярного определения в любое регулярное выражение файла спецификации лексем, где его имя указано в фигурных скобках. Например, следующая последовательность строк регулярных определений формирует макрос *IDENT*, который обозначает регулярное выражение, необходимое для поиска идентификаторов в исходных текстах программ:

```
ALPHA     [a-zA-Z]
DIGIT     [0-9]
IDENT     {ALPHA}({ALPHA}|{DIGIT})*
```

В итоговом регулярном выражении использованы регулярные определения *ALPHA* и *DIGIT* для обозначения латинских букв и цифр, соответственно. Генератор **LEX** автоматически подставляет, соответствующие им регулярные выражения в регулярное выражение, обозначаемое макросом *IDENT*. В результате подстановки для макроса *IDENT* получается следующее регулярное определение:

```
IDENT    [a-zA-Z]([a-zA-Z]|[0-9])*
```

согласно которому под идентификатором понимается любая алфавитно-цифровая последовательность символов, которая начинается с латинской буквы. Аналогичным образом, регулярное определение, заданное макросом *IDENT*, может быть использовано в регулярных выражениях секции правил или других регулярных определениях секции описаний также как макросы *ALPHA* и *DIGIT*.

В частности, макрос *DIGIT* может быть применен для формирования следующего регулярного определения вещественных чисел:

```
REAL       [+-]?(({DIGIT}+"."{DIGIT}*)|"."{DIGIT}+)
```

С учетом автоматической подстановки регулярного определения для макроса *DIGIT* оно эквивалентно следующей конструкции:

```
REAL       [+-]?(([0-9]+"."[0-9]*)|"."{DIGIT}+)
```

Кроме декларации перечисленных внешних объектов для секций правил и подпрограмм файла спецификации лексем, в секции описаний могут быть заданы размеры некоторых *внутренних таблиц* программы проектируемого лексического анализатора, если их значения должны отличаться от величин, которые по умолчанию устанавливает генератор **LEX**. Для спецификации внутренних таблиц в программе лексического анализатора используется структура фиксированных массивов, которые имеют ограниченный предельный размер. Поэтому в некоторых случаях, например, когда в секции правил файла спецификации лексем используются сложные регулярные выражения с большим числом операндов и операций, возможно переполнение внутренних таблиц лексического анализатора, размеры массивов которых приняты по умолчанию.

Желаемые размеры внутренних таблиц программы лексического анализатора специфицируются строками секции описаний, которые имеют следующий формат:

%X SIZE

В этой строке литера **X** после служебного символа **%** обозначает код внутренней таблицы лексического анализатора, а параметр **SIZE** устанавливает ее требуемый размер. Каждая из таких деклараций должна начинаться с первой позиции новой строки секции описаний, а части, определяющие код и размер таблицы, могут разделять произвольное число символов пробела или табуляции. При этом значение размера таблицы специфицируется неотрицательным целым десятичным числом. Код таблицы задается одной из латинских букв **E**, **P**, **N**, **A** ,**K** или **O**, которые имеют следующий смысл:

E таблица вершин синтаксического дерева регулярного выражения;
P таблица позиций вершин синтаксического дерева регулярного выражения;
N таблица состояний в конечном автомате лексического анализатора;
A таблица переходов в конечном автомате лексического анализатора;
K таблица классов символов в регулярных выражениях;
O таблица выходных элементов.

Например, следующая декларация секции описаний устанавливает допустимое число состояний в таблице состояний конечного автомата лексического анализатора равным **500** вместо значения, принятого по умолчанию:

%N 500

Следует отметить, что явная декларация размеров внутренних таблиц относительно редко используется на практике при построении программы лексического анализатора генератором **LEX** в отличие от объявления меток предусловий и регулярных определений. Она становится необходимой, только когда при построении программы лексического анализатора может произойти переполнение некоторых внутренних таблиц. Переполнение можно исключить, увеличивая размеры одних таблиц при соответствующем сокращении других таблиц, таким образом, чтобы суммарный размер всех таблиц сохранялся постоянным. Хотя возможность явного изменения размеров внутренних таблиц используется редко, разработчику лексических анализаторов полезно знать, что она существует.

СПЕЦИФИКАЦИЯ ПРАВИЛ

Секция правил – обязательная секция файла спецификации лексем. Она должна начинаться после первой разделительной пары знаков процента %%, и может продолжаться либо до секции подпрограмм, либо до конца файла спецификации лексем, если секция подпрограмм отсутствует. Эта секция предназначена для спецификации *набора правил* распознавания и обработки лексем во входном потоке стандартного ввода символьной информации. Любые входные символы, которые не соответствуют заданному набору правил, передаются в выходной поток стандартного вывода без изменений. Таким образом, лексический анализатор интерпретирует входной поток в соответствии с заданным набором правил.

Каждое правило должно начинаться с первой позиции новой строки секции правил и может занимать произвольное число строк. Между строками правил может произвольное число пустых строк и строк, содержащих комментарии или объявления глобальных переменных, которые оформляются в соответствии с синтаксисом языка программирования **C**. Аналогично секции описаний, комментарии и объявления глобальных переменных могут начинаться с любой позиции своих строк, кроме первой.

Количество правил непосредственно не регламентировано, но в неявной форме ограничено принятыми размерами внутренних таблиц программы лексического анализатора. В большинстве случаев, когда гарантирована однозначность интерпретации лексем, порядок перечисления правил не имеет значения. В общем случае правило состоит из двух частей и имеет следующий формат:

REGULAR ACTION

В этой спецификации **REGULAR** обозначает регулярное выражение для распознавания лексем входного потока, а **ACTION** – действие, которое предусмотрено для обработки лексем. Обе части правила должен разделять, по крайней мере, один символ пробела или табуляции.

В левой части правила может быть задано любое регулярное выражение. При необходимости в нем могут присутствовать макросы регулярных определений и метки предусловий из секции описаний, указанные, соответственно, в фигурных и угловых скобках. В частном случае в правиле может отсутствовать регулярное выражение.

В правой части действия правила могут быть специфицированы любые синтаксически корректные инструкции языка программирования **C**, которые должны выполняться лексическим анализатором, когда обнаружено соответствие символов входного потока регулярному выражению правила. Если соответствие регулярным выражениям правил не обнаружено, выполняется действие по умолчанию, которое обеспечивает автоматическое копирование символов входного потока стандартного ввода в выходной поток стандартного вывода. Таким образом, можно сказать, что действие это набор инструкций, которые выполняются вместо копирования входного потока в выходной.

В действия правил могут быть включены комментарии, декларации локальных переменных, операторы, выражения и вызовы стандартных функций системы программирования **C** или функций, специфицированных в секции подпрограмм, которые необходимы для реализации требуемой обработки лексем входного потока. Кроме того, в блоках действий доступны внешние переменные и макроопределения констант из секции описаний, а также собственные встроенные переменные, стандартные функции и операторы генератора **LEX**.

Количество инструкций в действиях правил не регламентировано. В частности, допустимы правила даже с пустым действием, однако, правила без действий не разрешены. С другой стороны гарантирована корректная обработка многострочных спецификаций действий с произвольным количеством строк и инструкций в них, которые объединяются в блок действий, ограниченный фигурными скобками. При этом блок действий должен начинаться в той же строке, где указано регулярное выражение правила.

На практике в секцию правил могут быть включены правила, которые обладают различными специфическими особенностями своей структуры. Типичными частными разновидностями правил являются:

правила с предусловием и правила без условия,
правила с пустым действием и правила без регулярного выражения,
правила с однострочным и многострочным действием,
правила с альтернативными действиями.

Перечисленные структурные особенности действий и регулярных выражений можно комбинировать для построения правил, которые обеспечивают требуемую обработку входного потока. Различные варианты формирования правил рассмотрены ниже.

ЭЛЕМЕНТАРНЫЕ ДЕЙСТВИЯ ПРАВИЛ

В простейшем случае функциональную обработку лексемы, соответствующей заданному регулярному выражению, обеспечивает единственная инструкция в действие правила, образуя правило с однострочным действием. Конструкцию правила, где действие содержит только одну инструкцию, иллюстрирует следующий пример спецификаций программы лексического анализатора, которая должна копировать входной поток стандартного ввода в выходной поток стандартного вывода, игнорируя символы возврата каретки (***C**arriage **R**eturn*, сокращенно, **CR**) перед символами перевода строки (***L**ine **F**eed*, сокращенно, **LF**) в конце каждой строки. В практическом смысле такой лексический анализ полезен для преобразования текстовой информации в формате операционных систем **Windows 9x/NT/2000/XP** в формат **OS UNIX.** В формате **Windows** каждую строку принято завершать парой символов возврата каретки и перевода строки (**`\r\n`**) с кодами ASCII **`\015`** и **`\012`**, соответственно, в системе счисления по основанию **8**. В формате **OS UNIX** конец строки обычно обозначает только один символ перевода строки (**`\n`**) с кодом ASCII **`\012`** в системе счисления по основанию **8**. Такое преобразование комбинации **CR-LF** в **LF** может быть специфицировано следующим правилом:

```
%%
\015\012        putchar('\n');
```

В единственном правиле приведенной спецификации регулярное выражение образует конкатенация символов **`\r`** и **`\n`**, которые заданы своими кодами ASCII **`\015`** и **`\012`** в системе счисления по основанию **8**. Действие реализовано стандартной библиотечной функцией **putchar** системы программирования **C**, которая в данном случае отображает в поток стандартного вывода символ перевода строки, игнорируя полученный вместе с ним символ возврата каретки. Любые символы входного потока, кроме **`\r`** и **`\n`** не соответствуют регулярному выражению данного единственного правила. Согласно действию по умолчанию они копируются в выходной поток без изменений.

Рассмотренное преобразование **CR-LF** в **LF** полезно в практическом смысле, потому что позволяет минимизировать объем передачи данных при копировании текстовой информации между файловыми системами **Windows** и **OS UNIX.** Кроме того, оно исключает проявление косметических неудобств, связанных с представлением символов возврата каретки в различных текстовых редакторах **OS UNIX**, где они обычно отображаются как литерал **^M**.

Следует отметить, что эквивалентный результат преобразования **CR-LF** в **LF** обеспечивает следующее правило, где используется регулярное выражение с правым контекстом и пустой *оператор точка с запятой* языка программирования **C** в качестве действия:

```
%%
\015/\012       ;     /* Правило с пустым оператором */
```

Аналогично правилу, рассмотренному выше, данное правило блокирует стандартный вывод символа возврата каретки (**\r**) в конце строки. Запрет выражается в том, что указанному символу соответствует действие с пустым оператором, в то время как для остальных символов сохраняется действие по умолчанию, которое обеспечивает их копирование в выходной поток стандартного вывода. В общем случае пустое действие позволяет эффективно игнорировать символы входного потока, которые не должны копироваться в выходной поток и не имеют функциональной обработки в программе лексического анализатора.

Еще один характерный пример технологии пустых действий иллюстрирует следующее правило, которое позволяет исключить из входного текста стандартный вывод символов пробела, табуляции и перевода строки:

```
%%
" "+        ;     /* исключение пробелов */
\t+         ;     /* исключение табуляций */
\n+         ;     /* исключение переводов строк */
```

Эти правила могут быть полезны при разработке лексического анализатора в рамках различных трансляторов языков программирования. Как известно одной из функций транслятора является исключение перечисленных разделителей из исходного текста программы в процессе его преобразования в объектный код.

Следует отметить, что правила с одинаковым, в данном случае, пустым, действием можно объединить в одно *альтернативное правило*. Для этого вместо действия указывается служебный символ **'|'**, который рекурсивно обозначает, что действие данного правила совпадает с действием следующего правила. Общее действие специфицируется только для последнего правила.

В частности, рассмотренные выше три правила исключения разделителей могут быть записаны в альтернативной форме следующим образом:

```
%%
" "+        |    /* исключение пробелов */
\t+         |    /* исключение табуляций */
\n+         ;    /* исключение переводов строк */
```

Кроме правил с пустым действием генератор **LEX** поддерживает правила, в которых отсутствует регулярное выражение, а имеется только действие. Такое правило должно предшествовать остальным правилам секции правил, если они есть, а его действие должно начинаться с любой позиции своей строки, кроме первой. Правила без регулярного выражения гарантированно выполняются перед началом лексического анализа входного потока и могут быть полезны для выполнения некоторых подготовительных операций. Например, действия следующих двух правил без регулярного выражения реализуют Escape-последовательности очистки экрана консоли перед последующей лексической обработкой входного потока:

```
%%
        printf("%c[2J", 033);
        printf("%c[H", 033);
```

Первое из этих правил реализует перевод формата экрана терминала в консольном режиме ввода-вывода. Второе правило обеспечивает установку текущей позиции текстового курсора в левый верхний угол экрана. Для реализации обеих Escape-последовательностей используется библиотечная функция **printf** системы программирования **C**.

БЛОКИ ДЕЙСТВИЙ

В большинстве практически интересных случаев действия правил могут занимать больше, чем одну, строку. Их инструкции оформляются в виде *блока действий*, ограниченного фигурными скобками в соответствие с синтаксисом языка программирования **C**. В блоки действий могут быть включены комментарии, декларации локальных переменных, операторы, выражения и вызовы различных стандартных, библиотечных или прикладных функций. Кроме того, в блоках действий доступны внешние переменные и макроопределения констант из секции описаний.

Использование блоков действий в правилах иллюстрирует следующий пример спецификации правил лексического анализатора, который должен нумеровать строки входного потока:

```
%%
        printf("0)\t");       /* Нумерация начальной строки */
\n      { /* Правило нумерации входных строк */
          static int number = 0;   /* Номер строки */
          putchar('\n');
          printf("%d)\t", ++number);
        }
```

В этом примере первое правило без регулярного выражения нумерует начальную строку входного потока нулем. Блок действий второго правила обеспечивает последнюю нумерацию всех последующих строк, входного потока. Номер очередной входной строки устанавливается по значению целочисленной статической переменной *number*, которая определена в блоке действий и инициализирована нулем. Ее значение увеличивается на **1** после ввода каждой очередной строки. Отображение номеров строк обеспечивает вызов библиотечной функции **printf** системы программирования **C**.

ВСТРОЕННЫЕ ПЕРЕМЕННЫЕ ДЕЙСТВИЙ

Кроме явно декларированных автоматических, статических или внешних переменных системы программирования **C** в действиях правил допускается использовать собственные встроенные переменные генератора **LEX**. При этом наиболее часто используются переменные **yytext** и **yyleng**, объявление и определение которых генератор **LEX** автоматически включает в исходный код проектируемого лексического анализатора.

Переменная **yytext** имеет тип указателя на символы (**char ***) и используется в блоке действия правила для адресации внешнего одномерного символьного массива, составленного из символов входного потока, которые удовлетворяют регулярному выражению данного правила. Символьный массив **yytext** обычно применяется, когда в блоке действий необходимо преобразовать или посимвольно обработать входной текст, соответствующий регулярному выражению правила.

Например, следующие два правила специфицирует криптографическое преобразование входного текста, составленного из заглавных и строчных латинских букв в кодировке ASCII, на основе *шифра Цезаря*:

```
%%

[a-z]     {  /* Шифрование строчных латинских букв */
               putchar((yytext[0] - 'a'+ 1)%26 + 'a');
          }

[A-Z]     {  /* Шифрование заглавных латинских букв */
               putchar((yytext[0] - 'A'+ 1)%26 + 'A');
          }
```

Как известно, шифр Цезаря реализует наиболее популярный вариант *шифра простой замены*, в котором каждая буква исходного текста, кроме последней буквы по алфавиту, заменяется следующей буквой алфавита. Последняя по алфавиту буква исходного текста, заменяется первой буквой алфавита. Учитывая, что в кодировке ASCII буквы упорядочены по алфавиту, образуя два непрерывных диапазона из **26**-ти строчных и **26**-ти заглавных букв, соответственно, требуемое преобразование кодов букв легко реализуется в арифметике вычетов по модулю **26**. В действиях каждого из приведенных правил, код текущей буквы идентифицируется значением начального элемента символьного массива **yytext[0]**. Для отображения результатов шифрования в выходном потоке стандартного вывода используется вызов библиотечной функции **putchar** системы программирования **C**.

В действиях более сложных правил массив **yytext** может содержать больше одного элемента и все его элементы, аналогично начальному элементу, доступны по своему индексу. В этих случаях часто бывает полезно знать число символов в массиве **yytext**, которые удовлетворяют данному правилу, то есть эффективную длину массива **yytext** в байтах. Эту информацию в действиях правил предоставляет значение внешней целочисленной переменной **yyleng**. Таким образом, считается, что все символы массива **yytext**, которые имеют индексы от **0** до (**yyleng – 1**), удовлетворяют текущему правилу. Фактически значение переменной **yyleng** эквивалентно результату измерения длины массива **yytext** с помощью библиотечной функции **strlen** из системы программирования **C**.

Практическое использование переменной **yyleng** совместно с переменной **yytext** иллюстрирует следующее правило поиска *символьных палиндромов* среди слов входного текста:

```
%%
[A-Za-z]+ {    /* Поиск символьных палиндромов */
          int i = yyleng - 1; /* обратный счетчик символов слова */
          int j = 0;          /* прямой счетчик символов слова */
          while(j < yyleng)   /* Цикл сравнения символов слова */
            if(tolower(yytext[j++]) != tolower(yytext[i--]))
              break;
          if( j == yyleng)    /* Оценка результата сравнения */
            puts(yytext);
          }
```

Как известно, символьным палиндромом является слово, которое одинаково читается слева направо и справа налево, например, *rotor* или *radar*. Регулярное выражение данного правила распознает любые слова входного текста, которые составлены из букв латинского алфавита. Проверку свойства палиндрома для полученного слова обеспечивает действие правила, где осуществляется сравнение **yyleng** парных букв, которые находятся на одинаковом расстоянии от противоположных концов массива **yytext**, содержащего все символы данного слова. Чтобы исключить различие регистра сопоставляемых символов, они приводятся к регистру строчных букв с помощью библиотечной функции **tolower** системы программирования **C**. Если все рассмотренные пары символов совпадают с точностью до регистра букв, то данное слово является палиндромом и все символы массива **yytext**, который содержит его, отображается через поток стандартного вывода с помощью библиотечной функции **puts** системы программирования **C**.

СТАНДАРТНЫЕ ФУНКЦИИ ДЕЙСТВИЙ

Как отмечалось выше, в действиях правил могут быть вызваны любые библиотечные функции системы программирования **C** и прикладные функции, определенные в секции подпрограмм. Кроме этого генератор **LEX** предоставляет стандартные функции **yymore**, **yyless** и **reject**, которые также могут быть вызваны в действиях правил. Они обеспечивают специальные возможности по обработке внутренних переменных и управление входным потоком.

В обычной ситуации содержимое символьного массива, адресованного внутренней переменной **yytext**, обновляется всякий раз, когда лексический анализатор распознает во входном потоке очередную лексему, соответствующую регулярному выражению правила. Однако, в некоторых случаях возникает потребность добавить к текущему содержимому массива **yytext** следующую последовательность символов входного потока. Для достижения этого результата в действии правила можно использовать стандартную функцию **yymore**. Она не имеет аргументов и обеспечивает накопление символьной информации в массиве **yytext**, а также соответствующим образом увеличивает значение переменной **yyleng**, которая фиксирует его длину.

Другая стандартная функция **yyless** позволяет получить обратный эффект. Она применяется, когда возникает потребность сохранить в массиве **yytext** не все, а только необходимое число символов распознанной последовательности. Количество сохраняемых символов устанавливает целочисленный аргумент функции **yyless**. Остальные символы возвращаются обратно во входной поток для последующей обработки. При этом соответствующим образом редуцируется содержимое массива **yytext** и уменьшается значение переменной **yyleng**.

Совместное использование стандартных функций **yymore** и **yyless** иллюстрирует следующий пример лексического правила, которое обеспечивает обработку входной символьной информации, заключенной в кавычки, заменяя при этом ограничительные кавычки на апострофы.

```
%%

\"([^\\\n]|\\\")*\"      { /* Обработка экранированной кавычки */
                           if(yytext[yyleng – 2] == '\\') {
                              yyless(yyleng – 1);
                              yyleng--;
                              yymore();
                           } /* if */
                           else { /* Обработка внешних кавычек */
                              yytext[0] = yytext[yyleng –1] = '\'';
                              printf("%s", yytext);
                           } /* else */
                         }
```

Это правило ориентировано на обработку любой строки текста, ограниченной кавычками, в которой могут присутствовать внутренние кавычки, экранированные символом обратной дробной черты (\"). Экранирование необходимо, когда внутренние кавычки являются частью текста и не должны рассматриваться как ограничитель строки, имеющий специальный смысл. Например, в следующей строке внутренний символ экранированной кавычки используется для обозначения размера в дюймах:

```
"Монитор марки NEC MultiSync XE17 имеет размер 17\" по диагонали."
```

При обработке этой строки по правилу рассматриваемого примера, сначала распознаются все символы от начальной до экранированной кавычки, включительно. Они заполняют массив **yytext**, который обрабатывается по альтернативе **if** действия правила. В процессе этой обработки вызов функции **yyless** возвращает во входной поток кавычку, завершающую массив **yytext**, неявно уменьшая его длину на один символ. Затем длина массива **yytext** уже явным образом уменьшается еще на один символ, чтобы исключит экранирующий символ обратной дробной черты, но уже без возврата его во входной поток. Действие правила по альтернативе **if** завершает вызов функции **yymore**, которая обеспечивает дополнение массива **yytext** оставшимися символами входной строки при следующем обращении к данному правилу. Таким образом, после выполнения действия по альтернативе **if** в массиве **yytext** будут находиться первые **49** символов рассматриваемой строки:

```
"Монитор марки NEC MultiSync XE17 имеет размер 17
```

В тоже время с учетом возврата кавычки во входном потоке остаются для последующей обработки **16** конечных символов исходной строки:

```
" по диагонали."
```

Конец исходной строки, также как ее начало соответствуют регулярному выражению рассматриваемого правила. Поэтому конечные символы строки добавляются к текущему содержимому массива **yytext**. Таким образом, в результате двукратного применения данного правила в массиве **yytext** будут сосредоточены все символы исходной строки, кроме экранирующей дробной черты перед внутренними кавычками:

```
"Монитор марки NEC MultiSync XE17 имеет размер 17" по диагонали."
```

Содержимое массива **yytext** обрабатывается по альтернативе **else** действия правила, обеспечивая стандартный вывод всех символов строки между ограничительными кавычками и замену ограничительных кавычек на апострофы. В результате исходная строка преобразуется к следующему виду:

```
'Монитор марки NEC MultiSync XE17 имеет размер 17" по диагонали.'
```

Следует отметить, что исходная (с кавычками) и результирующая (с апострофами) строки будут одинаково интерпретироваться командным процессором **sh** из **OS UNIX**, который игнорирует специальный смысл металитер внутри апострофов, но требует экранирования многих металитер внутри кавычек, чтобы исключить их специальный смысл. Кроме того, рассмотренное правило может быть практически полезно для преобразования символьных строк из формата языка **C** в формат языка **Pascal**.

Последняя из декларированных выше стандартных функций генератор **LEX**, функция **reject**, которая обеспечивает принудительный переход к обработке полученных символов входного потока по следующему подходящему правилу. При этом символы, полученные по текущему правилу, в действии которого специфицирован вызов функции **reject**, возвращаются во входной поток и становятся доступны для последующей обработки.

Функция **reject** обычно применяется для поиска перекрывающихся объектов входного потока. Например, следующие правила обеспечивает поиск и отображение через дефис всех двухбуквенных сочетаний соседних букв во входном слове, которые начинаются на каждой букве:

```
%%

[a-zA-Z][a-zA-Z]    {
                    printf("%s", yytext);
                    putchar('-');
                    reject();
                    }
[a-zA-Z]/[a-zA-Z]   ;
.                   printf("\b ");
```

Первое правило отображает в потоке стандартного вывода текущую пару букв и символ дефис после нее, используя библиотечные функции **printf** и **putchar** системы программирования **C**. Функция **reject** в действии этого правила обеспечивает принудительный переход к обработке их по второму правилу с постусловием и пустым действием, которое исключает первую букву пары. Таким образом, обработка оставшейся части входного слова будет продолжена по первому правилу со второй буквы текущей пары. Последнее правило необходимо, чтобы подавить отображение дефиса, когда достигнут конец входного слова. Для этого в действии правила предусмотрен вызов библиотечной функции **printf** системы программирования **C**, аргумент которой обеспечивает возврат на одну позицию и заменяет пробелом текущий символ стандартного вывода.

В частности, при обработке по этим правилам входного слова ***SCANNER*** будут выделены следующие **6** пар символов: ***SC-CA-AN-NN-NE-ER***. Если из блока действий первого правила исключить вызов функции **reject**, то при обработке того же входного слова будут обнаружены только **3** символьные пары: ***SC-AN-NE***.

В заключение следует отметить, что во многих версиях генератор **LEX** стандартная функция **reject** реализована в форме макроса (оператора) **REJECT**. В этих версиях рассмотренный пример остается корректным при замене функции **reject** оператором **REJECT**.

ОПЕРАТОРЫ ДЕЙСТВИЙ

Кроме встроенных переменных и стандартных функций генератор **LEX** предоставляет специальные *операторы* для использования в действиях правил. Наиболее часто в действиях правил применяются операторы **ECHO** и **BEGIN**.

Оператор **ECHO** введен для сокращенной записи стандартного вывода содержимого массива **yytext**, потому что эта операция особенно часто используется в действиях правил. В исходном тексте лексического анализатора оператор **ECHO** определяется директивой **#define** препроцессора системы программирования **C** следующим образом:

```
#define ECHO        fwrite(yytext, yyleng, 1, stdout)
```

Технику применения оператора **ECHO** демонстрирует следующий пример спецификации правил лексического анализатора, который должен формировать список слов входного текста, располагая каждое слово в отдельной строке стандартного вывода:

```
%%
[A-Za-z0-9]+   ECHO;                /* Отображение слова */
[^A-Za-z0-9]+  putchar('\n');       /* Перевод строки между словами */
```

В правилах этого примера под словом понимается любая алфавитно-цифровая последовательность из десятичных цифр и латинских букв. Разделителями слов считаются любые символы или комбинации символов ASCII, которые не являются буквами и цифрами. Первое правило примера обеспечивает стандартный вывод каждого слова входного текста с помощью оператора **ECHO**. Второе правило компрессирует все соседние разделители слов в один символ перевода строки, который отображается библиотечной функцией **putchar** системы программирования **C**.

Если оператор **ECHO** может быть использован в действиях любой системы правил, то оператор **BEGIN** целесообразно применять при наличии *правил с предусловиями*. Эти правила необходимы, когда важно обеспечить возможность обработки эквивалентных наборов символов по различным правилам в зависимости от предыстории входного потока, которая определяет состояние лексического анализатора. Например, целочисленные константы С-кода имеют различные значения в зависимости от префикса системы счисления.

Как отмечалось выше, для обозначения различных состояний лексического анализатора генератор **LEX** использует *метки предусловий*. Они специфицируются директивой %**Start** секции описаний и указываются в угловых скобках в регулярных выражениях секции правил. В отличие от обычных правил, которые актуальны в любом в любом состоянии лексического анализатора, правила с предусловиями a priori не активны и не рассматриваются в процессе лексического анализа входного потока, пока не установлены метки предусловий их регулярных выражений. Для управления активностью помеченных правил используется оператор **BEGIN**, который может устанавливать или сбрасывать метки предусловий. Он может применяться в действиях правил, распознающих выполнение соответствующих предусловий, в одном из двух форматов:

BEGIN STATE; или **BEGIN 0;**

В первом формате оператор **BEGIN** устанавливает состояние лексического анализатора, которое обозначено меткой *STATE*. После его выполнения становятся активными все правила с меткой <STATE> в левом контексте их регулярных выражений. При этом аргументом оператора **BEGIN** может быть любая метка из списка предусловий, специфицированных директивой %**Start** секции описаний. В общем случае вызов оператора **BEGIN** с ненулевым аргументом в действии обычного правила позволяет расширить список активных правил, добавляя к нему правила, метка предусловия которых совпадает с аргументом оператора **BEGIN**. Однако если вызов оператора **BEGIN** осуществляется в действии помеченного правила, то активность сохраняют только те правила, метка предусловия которых совпадает с аргументом оператора **BEGIN**. Второй формат, когда вызов оператора **BEGIN** осуществляется с нулевым аргументом, позволяет удалить из списка активных правил все правила с предусловиями. Это бывает необходимо для возврата лексического анализатора в исходное состояние, где все помеченные правила неактивны.

Различные форматы использования оператора **BEGIN** и правил с предусловиями иллюстрирует следующий пример спецификаций лексического анализатора четности или нечетности количества единиц в бинарной последовательности, состоящей из символов **0** и **1**, которую завершает символ перевода строки:

```
%Start EVEN ODD
%%
<EVEN>1    BEGIN ODD; /* Переход в нечетное состояние из четного */
<EVEN>\n   { /* Оценка четного состояния */
                puts(" Четное число единичных разрядов");
                BEGIN 0;
           }
<ODD>1     BEGIN EVEN; /* Переход в четное состояние из нечетного */
<ODD>\n    { /* Оценка нечетного состояния */
                puts(" Нечетное число единичных разрядов");
                BEGIN 0;
           }
1          BEGIN ODD; /* Переход в нечетное состояние из исходного */
\n         puts(" Нулевое число единичных разрядов");
.          ; /* Блокировка стандартного вывода */
```

В данном примере лексический анализатор реализует конечный автомат с 3-мя состояниями, в которых он может находиться, когда из входного потока получено, соответственно, нулевое, нечетное и четное число единиц бинарной последовательности. Состояния с четным и нечетным числом единиц обозначены, соответственно, метками предусловий *EVEN* и *ODD*, которые декларированы директивой %**Start** секции описаний. Исходное состояние с нулевым числом единиц не имеет метки и определено по принципу исключения двух других состояний.

Переход между состояниями происходит при стандартном вводе символов единицы (**'1'**) и перевода строки (**'\n'**). При этом, символ **'1'** вызывает переход в состояние, либо с нечетным, либо с четным количеством единиц, в зависимости от текущего числа полученных единиц бинарной последовательности, а символ перевода строки (**'\n'**) означает возврат в исходное состояние для анализа следующей бинарной последовательности. Таблицу переходов между состояниями формализуют **6** лексических правил, по **2** на каждое состояние. Одно правило каждой пары определяет переход при вводе символа **'1'**, а другое – при поступлении символа перевода строки (**'\n'**).

В частности, последняя пара указанных правил, которые не имеют предусловий, обслуживает исходное состояние. Согласно этим правилам при вводе символа перевода строки в исходном состоянии лексический анализатор диагностирует нулевую или пустую бинарную последовательность, отображая соответствующее информационное сообщение в потоке стандартного вывода с помощью библиотечной функции **puts** системы программирования **C**. Ввод символа **'1'** в исходном состоянии вызывает переход в состояние с нечетным числом единиц, которое устанавливается оператором **BEGIN** с аргументом *ODD* в действие этого правила.

После этого становятся активными два средних правила с меткой предусловия *ODD* в левом контексте регулярных выражений, которые обслуживают переходы из состояния с нечетным числом единиц. По этим правилам при вводе символа перевода строки лексический анализатор диагностирует бинарную последовательность с нечетным числом единиц, отображая соответствующее информационное сообщение в потоке стандартного вывода с помощью библиотечной функции **puts** системы программирования **C**, и возвращается в исходное состояние оператором **BEGIN** с нулевым аргументом. Ввод символа **'1'** в этом состоянии вызывает переход в состояние с четным числом единиц, которое устанавливается оператором **BEGIN** с аргументом *EVEN*. При этом метка предусловия *ODD* будет автоматически сброшена, потому что вызов оператора **BEGIN** осуществляется в действии правила с предусловием.

После этого становятся активными два первых правила с меткой предусловия *EVEN* в левом контексте регулярных выражений, которые обслуживают переходы из состояния с четным числом единиц, а средние правила с меткой предусловия *ODD* теперь неактивны. Согласно этим правилам при вводе символа перевода строки лексический анализатор диагностирует бинарную последовательность с четным числом единиц, отображая соответствующее информационное сообщение в потоке стандартного вывода с помощью библиотечной функции **puts** системы программирования **C**, и возвращается в исходное состояние оператором **BEGIN** с нулевым аргументом. Ввод символа `'1'` в этом состоянии вызывает переход в состояние с нечетным числом единиц, которое устанавливается оператором **BEGIN** с аргументом *ODD* в действии этого правила. При этом метка предусловия *EVEN* будет автоматически сброшена, потому что вызов оператора **BEGIN** осуществляется в действии правила с предусловием. После этого снова становятся активны два средних правила с предусловием *ODD* в левом контексте регулярных выражений, а правила с меткой предусловия *EVEN* теперь опять неактивны.

Таким образом, в начале лексического анализа все помеченные правила неактивны и обработка бинарной последовательности происходит по двум правилам исходного состояния. Затем при вводе символа `'1'` поочередно становятся активными пары помеченных правил, обслуживающих состояния с четным или нечетным числом единиц, и дальнейшая обработка бинарной последовательности осуществляется по активной паре помеченных правил. При этом правила исходного состояния по-прежнему активны, но неактуальны, потому что они подавляются эквивалентными активными правилами, которые расположены текстуально выше в секции правил. Они становятся актуальными при достижении конца бинарной последовательности, когда при стандартном вводе символа перевода строки после соответствующей диагностики, отображаемой с помощью библиотечной функции **puts** системы программирования **C,** происходит возврат в исходное состояние, где все помеченные правила неактивны, и лексический анализатор готов к обработке следующей бинарной последовательности из своего исходного состояния.

В заключение следует отметить, что исходя из косметических соображений, кроме рассмотренных правил, в конце секции правил включено еще правило с пустым действием. Оно необходимо для того, чтобы блокировать стандартный вывод любых символов входного потока, которые отсутствуют в регулярных выражениях остальных правил и должны отображаться по умолчанию.

НЕОДНОЗАЧНЫЕ ПРАВИЛА

В процессе лексического анализа входного потока может оказаться так, что входная последовательность символов соответствует регулярным выражениям нескольких правил. В этом случае возникает неоднозначность выбора действия при обработке входного потока. Чтобы разрешить указанную неоднозначность генератор **LEX** использует детерминированный механизм, который основан на следующих двух принципах. Всегда выбирается действие правила, распознающего наиболее длинную последовательность символов входного потока, а если несколько правил распознают одинаковую входную последовательность равной длины, то выполняется действие первого из них в секции правил. Указанные принципы разрешения неоднозначности выбора действий иллюстрирует следующий пример двух правил, которые пересекаются по области определения своих регулярных выражений:

```
%%
[Mm]ake         {/* Действие правила 1 */}
[A-Za-z]+       {/* Действие правила 2 */}
```

Регулярному выражению первого правила соответствуют только слова *Make* и *make*. Регулярному выражению второго правила удовлетворяют любые последовательности строчных и/или заглавных букв, в том числе содержащие фрагменты, которые соответствуют регулярному выражению первого правила, например, или *Makefile* или *makefile*. Автоматическое разрешение неоднозначности выбора действий в данном случае гарантирует обработку отдельных слов *Make* и *make* по первому правилу, в то время как все остальные слова будут распознаваться по второму правилу.

В частности, хотя слово *make* удовлетворяет регулярным выражениям обоих правил, но оно будет обрабатываться по действию первого правила. Так происходит потому, что в данном случае оба правила распознают последовательность равной длины (**4** символа), следовательно, приоритет имеет правило, которое расположено текстуально выше в секции правил. Однако если во входном потоке будет слово *Makefile,* то его обработка должна осуществляться в соответствии с действием второго правила. Так происходит потому, что в данном случае первому правилу удовлетворяют только **4** начальные символа этого слова, в то время как все **8** символов данного слова соответствуют второму правилу. Следовательно, приоритет будет иметь второе правило, которому удовлетворяет более длинная последовательность символов входного потока.

Таким образом, в рассмотренном примере механизм автоматического разрешения неоднозначности правил позволяет различать во входном потоке отдельные слова *Make* или *make* и все другие слова, где они могут присутствовать как фрагмент. Это может быть практически важно, например, для подсчета количества повторений слов *Make* и *make* во входном потоке без учета присутствия таких сочетаний в других словах. В этом случае действие первого правила должно содержать счетчик повторений этих слов, а действие второго правила может быть пустым.

ФУНКЦИОНАЛЬНАЯ РЕАЛИЗАЦИЯ ПРАВИЛ

Как отмечалось выше, по файлу спецификации лексем генератор **LEX** должен формировать на языке программирования **C** исходный текст лексического анализатора. Он имеет определенную функциональную структуру, основными компонентами которой являются функции **yylook** и **yylex**, автоматически формируемые по регулярным выражениям и действиям секции правил файла спецификации лексем.

Функция **yylook** реализует детерминированный конечный автомат, который должен осуществлять разбор входного потока символов по регулярным выражениям секции правил. Формально этот конечный автомат задается таблицей переходов, которая строится автоматически генератором **LEX** по регулярным выражениям секции правил и используется стандартной заготовкой функции **yylook** для распознавания лексем. В ранних версиях генератора **LEX** заготовка функции **yylook** предоставлялась в файле **ncform**, который обычно располагался в каталоге **`/usr/lib/lex`** файловой системы **OS UNIX**. В современных версиях заготовка для функции **yylook** включена непосредственно в выполняемый модуль генератора **LEX**.

Функция **yylex** содержит инструкции всех действий секции правил, обеспечивая обработку лексем, распознаваемых во входном потоке, и управление процедурой лексического анализа. Вызов этой функции реализует обращение к лексическому анализатору, а ее целочисленный код возврата может быть использован для идентификации лексем, например, в интересах последующей процедуры синтаксического анализа или интерпретации результатов лексического анализа.

Следует отметить, что кроме функций **yylook** и **yylex** в исходный текст лексического анализатора без изменений включается содержимое секции подпрограмм и блока описаний, а также инструкции секции описаний, которые начинаются не с первой позиции своих строк. Дополнительно к перечисленному генератор **LEX** включает в исходный текст лексического анализатора определения своих встроенных переменных, функций и операторов, которые используются в действиях секции правил.

СЕКЦИЯ ПОДПРОГРАММ

Секция подпрограмм – необязательная заключительная секция файла спецификации лексем. Она должна быть отделена от предшествующей секции правил разделительной парой символов процента %% и продолжается до конца файла спецификации лексем. Содержимое секции правил без изменений включается в исходный текст лексического анализатора и должно быть оформлено в нотации языка программирования **C**.

Секция подпрограмм в основном предназначена для спецификации исходного кода прикладных функций, которые используются в действиях секции правил. Кроме того, в секции подпрограмм можно перегрузить исходные коды некоторых стандартных функций ввода-вывода лексического анализатора, которые предоставляет генератор **LEX**, а также специфицировать основную функцию **main**, когда это необходимо. При этом следует сохранять неизменными имена, типы аргументов и кодов возврата перегружаемых функций.

На практике наиболее часто встречается перегрузка стандартной функции **yywrap**, которая автоматически вызывается лексическим анализатором при достижении конца входного потока стандартного ввода. Эта функция не имеет аргументов и должна возвращать целочисленное значение **0** или **1**, оценка которого предусмотрена в программе лексического анализатора. При этом возврат значения **1** вызывает корректное завершение процедуры лексического анализа. Возврат нулевого значения предоставляет возможность продолжить выполнение процедуры лексического анализа для обработки данных, поступающих из другого источника. В стандартном варианте функция **yywrap** возвращает значение **1**. Перегрузка стандартной функции **yywrap** обычно применяется, когда необходимо изменить ее стандартный код возврата с **1** на **0** или выполнить определенные завершающие действия в конце лексического анализа входного потока при коде возврата **1**.

Использование прикладных и перегруженных функций секции подпрограмм иллюстрирует следующий пример спецификаций лексического анализатора, который отображает максимальную длину слова входного текста через поток стандартного вывода:

```
%%
[A-Za-z]+      wordlen(yyleng);
.              |
\n             ;
%%

/* Прикладная функция оценки длины слова */
int wordlen(int len) {
static int maxlen = 0;
if(len > maxlen)
  maxlen = len;
} /* wordlen */

/* Перегрузка стандартной функции yywrap */
int yywrap() {
printf("%d\n", wordlen(0));
} /* wordlen */
```

В этом примере, в соответствии с регулярным выражением первого правила, словом считается любая последовательность строчных и/или заглавных латинских букв. Для оценки длины каждого полученного слова в действии этого правила вызывается прикладная функция *wordlen*, которая определена в секции подпрограмм. При вызове в действии правила ей передается значение внутренней переменной **yyleng**, фиксирующей длину текущего слова, чтобы сопоставить его со значением статической переменной *maxlen*, которая сохраняет размер наиболее длинного из полученных слов входного текста. Текущее значение переменной *maxlen* изменяется, если его величина меньше длины очередного слова и всегда может быть получено через код возврата прикладной функции *wordlen*. Однако в данном примере код возврата прикладной функции *wordlen* используется только для контроля итогового значения статической переменной *maxlen*, которое получается после обработки всех слов входного потока и, следовательно, соответствует максимальной длине слова входного текста.

Конец входного потока инициирует автоматический вызов стандартной функции **yywrap**, исходный код которой перегружен в секции подпрограмм. Перегрузка стандартной функции **yywrap** осуществляется с целью обеспечить контрольный вызов прикладной функции *wordlen* в конце лексического анализа, когда ее код возврата определяет максимальную длину слова входного текста. При этом прикладная функция *wordlen* вызывается с нулевым аргументом, чтобы исключить изменение величины статической переменной *maxlen*, а ее код возврата передается библиотечной функции **printf** системы программирования **C** для отображения полученного результата через поток стандартного вывода. Выполнение функции **yywrap** оканчивается возвратом значения `1`, чтобы корректно обозначить завершение процедуры лексического анализа входного потока.

Кроме стандартной функции **yywrap** в секции подпрограмм могут быть перегружены еще две стандартные функции **input** и **unput**, генератор **LEX** использует для обработки входного потока. Перегружаемые функции **input** и **unput** являются **LEX**-ориентированными реализациями библиотечных функций **getc** и **ungetc** системы программирования **C**, которые конкретизированы для обработки потока стандартного ввода в лексическом анализаторе.

В частности, стандартная функция **input** в лексическом анализаторе обеспечивает чтение входного потока, возвращая код каждого полученного символа или **0**, когда достигнут конец входного потока. Стандартная функция **unput** обеспечивает возврат обратно во входной поток для повторного чтения символ, код которого задается ее целочисленным аргументом.

В некоторых реализациях генератора **LEX** дополнительно к стандартным функциям **input** и **unput** предусмотрена также стандартная функция **output**, которая используется для обработки выходного потока и эквивалентна библиотечной функции **putc** из системы программирования **C**. Она обеспечивает стандартный вывод символа, код которого идентифицирован ее аргументом. Однако в большинстве современных версий генератора **LEX** эта стандартная функция либо отсутствует, либо недоступна для явного обращения.

Следует отметить, что перегрузка стандартных функций ввода-вывода относительно редко применяется на практике. Обычно они неявно вызываются лексическим анализатором для стандартной обработки входного потока. Однако в некоторых случаях целесообразно реализовать явный вызов этих функций в действиях правил или в прикладных функциях секции подпрограмм.

Эту ситуацию иллюстрирует следующий пример спецификации лексем для лексического анализатора скобочных выражений, где явный вызов стандартных функций **input** и **unput** исключает остаток входной строки после обнаружения ошибки расстановки скобок:

```
%{
extern int bracketcount;
int bracketcount;
%}
%%
          bracketcount = 0;
\(        ECHO; bracketcount++;
\)        { ECHO;
            if(--bracketcount < 0) resync();
          }
\n        return(bracketcount);
.         ;
%%
void resync() {
while(input() != '\n');
unput('\n');
return;
} /* resync */
```

Правила этого примера обеспечивают проверку соответствия открывающих и закрывающих круглых скобок в каждой входной строке, содержащей произвольное алгебраическое или арифметическое выражение, игнорируя при этом любые символы потока стандартного ввода кроме круглых скобок и перевода строки. Они реализуют упрощенный вариант алгоритма *Рутисхаузера*, где лексический анализ скобочных выражений осуществляется по значению счетчика скобок. В этом алгоритме каждая очередная открывающая скобка увеличивает, а каждая закрывающая скобка уменьшает значение счетчика на **1**. Любые другие символы выражения не изменяют значения счетчика скобок. Анализ скобок завершается, если значение счетчика скобок отрицательно или когда достигнут конец выражения. При правильной расстановке скобок значение счетчика в конце выражения должно быть равно нулю, а внутри выражения всегда неотрицательно.

В правилах, реализующих этот алгоритм, для хранения значения счетчика скобок используется внешняя целочисленная переменная *bracketcount*, которая объявляется в блоке описаний и инициализируется нулем перед обработкой каждой входной строки в действии начального правила без регулярного выражения. Инкремент значения переменной *bracketcount* обеспечивает действие первого полного правила, когда во входном потоке обнаружена открывающая круглая скобка. Второе правило гарантирует декремент значения переменной *bracketcount* при обнаружении во входном потоке закрывающей круглой скобки, а также принудительное исключение всех символов до конца строки, если ее величина становится отрицательной.

Исключение остатка строки реализует прикладная функция *resync*, которая специфицирована в секции подпрограмм для явного вызова стандартных функций **input** и **unput**. При этом циклический вызов функции **input** исключает из входного потока все символы до конца строки. Функция **unput** возвращает обратно во входной поток символ перевода строки для обработки по третьему полному правилу. Это правило обеспечивает естественное или аварийное завершение лексического анализа, в конце входной строки, содержащей скобочное выражение. Последнее правило с пустым действием необходимо для того, чтобы блокировать обработку любых входных символов, кроме специфицированных в регулярных выражениях предшествующих правил.

Результат проверки правильности расстановки скобок с помощью рассмотренной системы правил можно оценить по последнему значению переменной *bracketcount*, которое возвращается оператором **return** системы программирования **C** в действии третьего полного правила. Скобочное выражение считается корректным, если значение переменной *bracketcount* после завершения лексического анализа равно нулю. Если значение переменной *bracketcount* положительно, то в регулярном выражении открывающих скобок больше, чем закрывающих на величину значения счетчика скобок. Если значение переменной *bracketcount* отрицательно, то закрывающих скобок больше, чем предшествующих им открывающих скобок.

Кроме прикладных и перегруженных стандартных функций в секции подпрограмм может быть специфицирован исходный код основной функции **main**, которая необходима, когда лексический анализатор реализуется в формате выполняемого модуля. Основная функция **main** в данном случае должна содержать вызов функции **yylex**, которая формируется генератором **LEX** по действиям правил и обеспечивает выполнение процедуры лексического анализа.

В стандартном варианте, который содержится в библиотеке объектных модулей генератора **LEX**, компонуемой при сборке выполняемого модуля лексического анализатора, исходный код основной функции **main** имеет следующий вид:

```
main() {
yylex();
exit(0);
} /* main */
```

При необходимости расширить стандартный вариант основной функции **main**, ее исходный код, дополненный соответствующими инструкциями, может быть размещен в секции подпрограмм файла спецификации лексем вместе с исходным кодом прикладных и перегруженных стандартных функций. Кроме того, исходный код основной функции **main** следует специфицировать даже в стандартном варианте, если не планируется компоновка библиотеки объектных модулей генератора **LEX** при сборке выполняемого модуля лексического анализатора.

Модификация стандартного варианта основной функции **main** необходима, когда требуется дополнительная постобработка кодов возврата функции **yylex**, например, для визуализации результатов лексического анализа. Эту ситуацию иллюстрирует следующая спецификация основной функции **main** для лексического анализатора скобочных выражений, правила которого были рассмотрены выше:

```
int main() {
while(feof(stdin) == 0)
  switch(yylex()) {
    case 0:    puts(" OK (~)"); /* Правильная расстановка скобок */
               break;
    case -1:   puts((" ER (<)"); /* Пропущена открывающая скобка */
               break;
    default:   puts((" ER (>)"); /* Не хватает закрывающих скобок */
               break;
  } /* switch */
exit(0);
} /* main */
```

В данном случае основная функция **main** обеспечивает визуальную диагностику результатов лексического анализа скобочных выражений. Ее стандартный исходный код может быть перегружен в секции подпрограмм файла спецификации лексем или задан в отдельном файле, который после компиляции компонуется при сборке лексического анализатора.

Для выполнения процедуры лексического анализа входных строк в основной функции **main** предусмотрен циклический вызов функции **yylex**, которая формируется генератором **LEX** по действиям правил. Ее различные коды возврата, передаваемые оператором **return** в действиях правил, соответствуют трем альтернативам оператора **switch** в основной функции **main**. Каждая альтернатива реализует стандартный вывод диагностического сообщения, которое отражает результат лексического анализа расстановки скобок в текущей строке стандартного ввода. Для отображения диагностических сообщений во всех альтернативах используется библиотечная функция **puts** системы программирования **C**. Цикл вызова функции **yylex** и диагностика ее кода возврата продолжается, пока не достигнут конец потока стандартного ввода, который идентифицирует библиотечная функция **feof** системы программирования **C**.

ОБРАБОТКА СПЕЦИФИКАЦИЙ ЛЕКСЕМ

Технологический процесс разработки лексического анализатора с использованием генератора **LEX** разделяется на три этапа:

подготовка файла спецификаций лексем,
генерация исходного кода лексического анализатора,
компиляция объектного или выполняемого модуля лексического анализатора.

На первом этапе необходимая спецификация лексем создается разработчиком лексического анализатора и сохраняется в текстовом файле с произвольным именем в любом доступном рабочем каталоге файловой системы **OS UNIX**. Обычно имя файла спецификации лексем сопровождается расширением ***.l***, которое обозначает его предметную область. Базовое имя файла спецификации лексем часто выбирают по названию проектируемого лексического анализатора. Для формирования файла спецификации лексем может быть использован любой текстовый редактор **OS UNIX**.

На втором этапе по файлу спецификации лексем формируется исходный код лексического анализатора на языке программирования **C**. Для решения этой задачи генератор **LEX** предоставляет одноименную команду **`lex`**, выполняемый модуль которой располагается в каталоге **`/usr/bin`** файловой системы **OS UNIX**. В простейшем варианте команда **`lex`** может быть вызван с единственным аргументом, который обозначает имя файла спецификации лексем. В этом случае исходный код проектируемого лексического анализатора по умолчанию формируется в текстовом файле с предопределенным именем ***lex.yy.c***, который создается в текущем каталоге файловой системы **OS UNIX.** Если, например, спецификация лексем сосредоточена в файле *lexan.l*, то его обработка генератором **LEX** для получения исходного кода в файле ***lex.yy.c*** может быть реализована следующей командной строкой:

```
$ lex lexan.l
```

Когда требуется получить исходный код лексического анализатора в файле с заданным именем, командная строка вызова генератора **LEX** должна содержать опцию **`-t`** перед аргументом, который обозначает имя файла спецификации лексем. В этом случае исходный код лексического анализатора формируется в потоке стандартного вывода**,** который всегда можно перенаправить в файл с указанным именем средствами любого интерпретатора команд **OS UNIX**. Например, следующая командная строка обрабатывает спецификацию лексем в файле *lexan.l* и обеспечивает получение исходного кода лексического анализатора в файле *lexan.c,* куда перенаправляется поток стандартного вывода:

```
$ lex -t lexan.l > lexan.c
```

Чтобы получить возможность практически использовать лексический анализатор, необходимо на третьем этапе преобразовать его исходный код, построенный генератором **LEX** по файлу спецификации лексем**,** в *объектный* или *исполняемый* модуль. Для этого могут быть использованы стандартные инструментальные средства компилирующей системы языка программирования **C**. Обращение к ней в **OS UNIX** обеспечивает команда **`cc`** (или **`gcc`**), результат выполнения которой зависит от опций и аргументов командной строки ее вызова. В частности, компиляцию исходного кода лексического анализатора, который сосредоточен, например, в файле *lex.yy.c* в *объектный модуль* обеспечивает следующая командная строка:

```
$ cc -c lex.yy.c
```

В результате выполнения этой командной строки, образуется объектный файл *lex.yy.o* в текущем каталоге файловой системы **OS UNIX**. Он может компоноваться с другими объектными модулями при конструировании выполняемой программы, обеспечивая в ней чтение и требуемую лексическую обработку входного потока символьной информации. Например, при проектировании разнообразных трансляторов Обычно лексический анализатор обычно реализуется в формате объектного модуля, который компонуется с объектным модулем синтаксического анализатора, передавая ему коды типов и значения распознаваемых лексем входного потока. должен передавать на следующую стадию синтаксического анализа коды типов и значения распознанных лексем. При этом особенно просто осуществляется взаимодействие с объектным модулем синтаксического анализатора, который формируется *генератором синтаксических анализаторов* **YACC**.

Во втором случае, когда требуется получить *исполняемый модуль* лексического анализатора, исходный код которого сосредоточен, например, в файле *lexan.c*, обращение к средствам компилирующей системы языка программирования **C** может быть реализовано следующей командной строкой:

```
$ cc –o lexan lexan.c –ll
```

При выполнении этой командной строки исходный код лексического анализатора в файле *lexan.c* транслируется в указанный после ключа **–o** исполняемый файла *lexan*, который образуется в текущем каталоге файловой системы **OS UNIX**. Последний аргумент (**–ll**) этой командной строки гарантирует компоновку объектных модулей стандартной библиотеки **libl.a** или **libl.so** генератора **LEX**, которые обычно располагаются в каталоге **/usr/lib** файловой системы **OS UNIX**. Это необходимо, когда лексический анализатор использует стандартные варианты перегружаемой функции **yywrap** и основной функции **main**, которые явно не определены в секции подпрограмм файла спецификации лексем.

Построение лексического анализатора в формате исполняемого файла целесообразно, когда он должен решать самостоятельные задачи, связанные с лексической обработкой входного потока символьных данных. Например, преобразование формата входного потока данных, контекстная замена, обработка комментариев в исходном тексте программы, исключение лишних пробелов, измерение количественных характеристик входного потока. В любом из перечисленных случаев лексический анализатор должен выполняться как индивидуальная программа, обеспечивая требуемую лексическую обработку входного потока.

ПРИМЕР РАЗРАБОТКИ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА

Многие практические приложения лексического анализа являются частными разновидностями общей проблемы поиска совпадений для фрагмента входного текста, ограниченного (или разделяемого) известными символьными элементами. Одним из наиболее популярных приложений в данной проблематике является лексический анализ комментариев в исходных текстах программ на различных алгоритмических языках высокого уровня.

Лексическая обработка комментариев может производиться с различными целями. Например, исключение комментариев из исходного кода программы, выделение комментирующего текста, оценка степени "комментированности" программы. Ниже приводится пример файла спецификации лексем для лексического анализатора, который измеряет общую длину комментирующего текста без учета пробелов, табуляций и переводов строк в исходном коде программы на языке **Pascal**:

```
%Start ISCOM
OPENCOM   "(*"
CLOSECOM  "*)"
%{
#include <stdio.h>
extern int comcount;
int comcount = 0;
%}
%%
{OPENCOM}                BEGIN ISCOM;
<ISCOM>[^*\t \n]+        comcount += yyleng;
<ISCOM>\*/[^)]           comcount++;
{CLOSECOM}               BEGIN 0;
.                        |
\n                       ;
%%
int yywrap() {
printf("(* %d *)\n", comcount);
return(1);
} /* yywrap */
```

В языке программирования **Pascal** комментарии ограничивают символьные пары **(*** и ***)**, которые идентифицируют, соответственно, начало и конец комментария. Для их обозначения в файле спецификации лексем введены регулярные определения *OPENCOM* и *CLOSECOM*, которые заданы в секции описаний. Кроме того, в секции описаний директивой %**Start** декларируется метка предусловия *ISCOM*, которая обозначает состояние лексического анализатора при разборе входного текста внутри комментария, а также введен блок описаний для подключения заголовочного файла **<stdio.h>** из системы программирования **C** и определения внешней целочисленной переменной *comcount*, инициализированной нулевым значением.

Перечисленные инструкции секции описаний имеют вспомогательный характер и введены для использования в секции правил, где лексический анализ комментариев осуществляется по следующей схеме: найти открывающий ограничитель комментария, отобразить текст внутри комментария, найти закрывающий ограничитель комментария. В соответствии с этой схемой первое правило с регулярным определением *OPENCOM* необходимо, чтобы обнаружить открывающий ограничитель комментария и установить метку предусловия *ISCOM* оператором **BEGIN**. После этого становятся активны два следующих правила с предусловием *ISCOM*, действия которых соответствующим образом увеличивают значение счетчика комментариев, заданного внешней переменной *comcount*. При этом одно из указанных правил обеспечивает обработку любой последовательности символов без пробелов, табуляций и перевода строки до символа **'*'**, который может быть началом закрывающего ограничителя комментария. Другое правило позволяет учитывать символы **'*'** внутри комментария. Обработку комментария завершает правило с регулярным определением *CLOSECOM*, которое сбрасывает метку предусловия *ISCOM* оператором **BEGIN** с нулевым аргументом. Последнее альтернативное правило с пустым действием позволяет игнорировать все символы вне комментариев, а также разделители внутри них.

Файл спецификации лексем завершает секция подпрограмм, которая в данном случае используется для перегрузки стандартной функции **yywrap**, которая автоматически вызывается в конце лексического анализа входного потока. Перегрузка выполняется с целью обеспечить отображение через поток стандартного вывода результирующего значения суммарной длины комментариев, которое сохраняет внешняя переменная *comcount*. Для этого используется библиотечная функция **printf** системы программирования **C**. Аналогично стандартному варианту перегруженная функция **yywrap** возвращает код **1**, чтобы идентифицировать корректное завершение процедуры лексического анализа.

Рассмотренный файл спецификации лексем может быть подготовлен текстовым редактором и сохранен, например, под именем *compas.l* в любом доступном каталоге файловой системы. Построение файла исходного кода лексического анализатора, например, под именем *compas.c* обеспечивает следующая командная строка вызова генератора **LEX**:

```
$ lex -t compas.l > compas.c
```

Исполняемый модуль лексического анализатора, например, в файле *compas* образуется по исходному тексту при следующем обращении к компилирующей системе языка программирования **C**:

```
$ cc -o compas compas.c -ll
```

Следует отметить, что в данном случае для компоновки стандартного объектного кода основной функции **main** необходимо подключить библиотеку объектных модулей **libl.a** (или разделяемый объект **libl.so**) генератора **LEX**, что указано последним аргументом (**-ll**) этой командной строки.

Полученный исполняемый файл *compas* ориентирован на обработку стандартного ввода, поэтому его непосредственный вызов мало полезен, так как обычно требуется лексический анализ комментариев в программе, исходный текст которой сосредоточен в файле. Требуемую лексическую обработку комментариев, например, в файле *foo.pas* обеспечивает следующий конвейер команд:

```
$ cat foo.pas | compas
```

После завершения лексического анализа файла *foo.pas* общая длина его комментариев будет отображена в потоке стандартного вывода соответствующим целым числом, которое обрамляют ограничители комментариев языка программирования **Pascal**.

В заключение следует отметить, что к классу рассмотренного примера относятся, в частности, следующие лексические задачи: поиск тегов языка **HTML** или текста между одноименными тегами, обработка текста в кавычках или апострофах, распознавание **IP**-адресов или доменных имен сети **Internet**, которые могут быть решены аналогичным образом.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. **С. Баурн**
   Операционная система UNIX. – М.: Мир, 1986.

2. **В.П. Тихомиров**, **М.И**. **Давидов**
   Операционная система ДЕМОС:
   Инструментальные средства программирования. – М.: Финансы и статистика, 1988.

3. **Б.В. Керниган**, **Р. Пайк**
   UNIX – универсальная среда программирования. – М.: Финансы и статистика, 1992.

4. **Д.С. Амстронг**, мл.
   Секреты UNIX. – М.: Диалектика, 2000.

5. **А. Ахо**, **Р. Сети**, **Д. Ульман**
   Компиляторы:
   Принципы, технологии, инструменты. – М.: Вильямс, 2001.

6. **Д. Фридл**
   Регулярные выражения. – СПб.: Питер, 2001.

7. **Р. Хантер**
   Основные концепции компиляторов. – М.: Вильямс, 2002.

8. **Д. Хопкрофт**, **Р. Мотвани**, **Д. Ульман**
   Введение в теорию автоматов, языков и вычислений. – М.: Вильямс, 2002.

СОДЕРЖАНИЕ