Лексические анализаторы

Лексические анализаторы (сканеры). Принципы построения сканеров

Назначение лексического анализатора

Прежде, чем перейти к рассмотрению лексических анализаторов, необходимо дать четкое определение того, что же такое лексема.

Лексема (лексическая единица языка) — это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своем составе других структурных единиц языка.

Лексемами языков естественного общения являются слова[1]. Лексемами языков программирования являются идентификаторы, константы, ключевые слова языка, знаки операций и разделители. Состав возможных лексем каждого конкретного языка программирования определяется синтаксисом этого языка.

Лексический анализатор (или сканер) — это часть компилятора, которая читает исходную программу и выделяет в ее тексте лексемы входного языка. На вход лексического анализатора поступает текст исходной программы, а выходная информация передается для дальнейшей обработки компилятором на этапе синтаксического анализа и разбора.

Результатом работы лексического анализатора является перечень всех найденных в тексте исходной программы лексем. Этот перечень лексем можно представить в виде таблицы, называемой *таблицей лексем*. Каждой лексеме в таблице лексем соответствует некий уникальный условный код, зависящий от типа лексемы, и дополнительная служебная информация. Кроме того, информация о некоторых типах лексем, найденных в исходной программе, должна помещаться в таблицу идентификаторов (или в одну из таблиц идентификаторов, если компилятор предусматривает различные таблицы идентификаторов для различных типов лексем).

ВНИМАНИЕ

> Не следует путать таблицу лексем и таблицу идентификаторов — это две принципиально разные таблицы, обрабатываемые лексическим анализатором.

Таблица лексем фактически содержит весь текст исходной программы, обработанный лексическим анализатором. В нее входят все возможные типы лексем,

---

[1] В языках естественного общения *лексикой* называется словарный запас языка. Лексический состав языка изучается лексикологией и фразеологией, а значение лексем (слов языка) - семасиологией. В языках программирования словарный запас, конечно, не столь интересен и специальной наукой не изучается.

кроме того, любая лексема может встречаться в ней любое количество раз. Таблица идентификаторов содержит только определенные типы лексем — идентификаторы и константы. В нее не попадают такие лексемы, как ключевые (служебные) слова входного языка, знаки операций и разделители. Кроме того, каждая лексема (идентификатор или константа) может встречаться в таблице идентификаторов только один раз. Также можно отметить, что лексемы в таблице лексем обязательно располагаются в том же порядке, как и в исходной программе (порядок лексем в ней не меняется), а в таблице идентификаторов лексемы располагаются в любом порядке так, чтобы обеспечить удобство поиска (методы организации таблиц идентификаторов рассмотрены в предыдущей главе).

В качестве примера можно рассмотреть некоторый фрагмент исходного кода на языке Pascal и соответствующую ему таблицу лексем, представленную в таблице 3.1:

```
...
begin
 for I := 1 to N do
    fg := fg * 0.5
...
```

Таблица 3.1. Лексемы программы

Лексема	Тип лексемы	Значение
begin	Ключевое слово	X1
for	Ключевое слово	X2
i	Идентификатор	i : 1
:=	Знак присваивания	S1
1	Целочисленная константа	1
to	Ключевое слово	X3
N	Идентификатор	N : 2
do	Ключевое слово	X4
fg	Идентификатор	fg : 3
:=	Знак присваивания	S1
fg	Идентификатор	fg : 3
*	Знак арифметической операции	A1
0.5	Вещественная константа	0.5

Поле «значение» в таблице 3.1 подразумевает некое кодовое значение, которое будет помещено в итоговую таблицу лексем в результате работы лексического анализатора. Конечно, значения, которые записаны в примере, являются условными. Конкретные

коды выбираются разработчиками при реализации компилятора. Важно отметить также, что устанавливается связка таблицы лексем с таблицей идентификаторов (в примере это отражено некоторым индексом, следующим после идентификатора за знаком :, а в реальном компиляторе определяется его реализацией).

Принципы построения лексических анализаторов

С теоретической точки зрения лексический анализатор не является обязательной частью компилятора. Все его функции могут выполняться на этапе синтаксического разбора, поскольку полностью регламентированы синтаксисом входного языка. Однако существует несколько причин, по которым в состав практически всех компиляторов включают лексический анализ:

- применение лексического анализатора упрощает работу с текстом исходной программы на этапе синтаксического разбора и сокращает объем обрабатываемой информации;
- некоторые задачи, требующие использования сложных вычислительных алгоритмов на этапе синтаксического анализа, могут быть решены более простыми методами на этапе лексического анализа (например, задача различения унарного минуса и бинарной операции вычитания, обозначаемых одним и тем же знаком «-»);
- лексический анализатор отделяет сложный по конструкции синтаксический анализатор от работы непосредственно с текстом исходной программы, структура которого может варьироваться в зависимости от архитектуры вычислительной системы, где выполняется компиляция — при такой конструкции компилятора для перехода на другую вычислительную систему достаточно только перестроить относительно простой лексический анализатор;
- в современных системах программирования лексический анализатор может выполнять обработку текста исходной программы параллельно с его подготовкой пользователем — это дает системе программирования принципиально новые возможности, которые позволяют снизить трудоемкость разработки программ (более подробно об этом сказано в главе «Современные системы программирования»).

Функции, выполняемые лексическим анализатором, и состав типов лексем, которые он выделяет в тексте исходной программы, могут меняться в зависимости от реализации компилятора. То, какие функции должен выполнять лексический анализатор, а какие оставлять для этапа синтаксического разбора, решают разработчики компилятора. В основном лексические анализаторы выполняют исключение из текста исходной программы комментариев и незначащих символов (пробелов, символов табуляции и перевода строки), а также выделение лексем следующих типов: идентификаторов, строковых, символьных и числовых констант, ключевых (служебных) слов входного языка, знаков операций и разделителей.

Лексический анализатор имеет дело с такими объектами, как различного рода константы и идентификаторы (к последним относятся и ключевые слова). Язык констант и идентификаторов является регулярным — то есть может быть описан с помощью регулярных грамматик. Распознавателями для регулярных языков

являются конечные автоматы. Следовательно, основой для реализации лексических анализаторов служат регулярные грамматики и конечные автоматы.

Существуют правила, с помощью которых для любой регулярной грамматики может быть построен конечный автомат, распознающий цепочки языка, заданного этой грамматикой (эти правила рассмотрены далее в этой главе).

Конечный автомат для каждой входной цепочки языка дает ответ на вопрос о том, принадлежит или нет цепочка языку, заданному автоматом. Однако в общем случае задача лексического анализатора несколько шире, чем просто проверка цепочки символов лексемы на соответствие входному языку. Кроме этого, он должен выполнить следующие действия:

- определить границы лексем, которые в тексте исходной программы явно не указаны;
- выполнить действия для сохранения информации об обнаруженной лексеме (или выдать сообщение об ошибке, если лексема неверна).

Эти действия связаны с определенными проблемами. Далее рассмотрено, как эти проблемы решаются в лексических анализаторах.

Проблемы построения лексических анализаторов

Определение границ лексем

Выделение границ лексем является нетривиальной задачей. Ведь в тексте исходной программы лексемы не ограничены никакими специальными символами. Если говорить в терминах лексического анализатора, то определение границ лексем — это выделение тех строк в общем потоке входных символов, для которых надо выполнять распознавание.

Иллюстрацией случая, когда определение границ лексемы вызывает определенные сложности, может служить пример оператора программы на языке FORTRAN: по фрагменту исходного кода `DO 10 I=1` невозможно определить тип оператора языка (а соответственно, и границы лексем). В случае `DO 10 I=1.15` это присвоение вещественной переменной `DO10I` значения константы `1.15` (пробелы в языке FORNTAN игнорируются), а в случае `DO 10 I=1,15` — это цикл с перечислением от `1` до `15` по целочисленной переменной `I` до метки `10`.

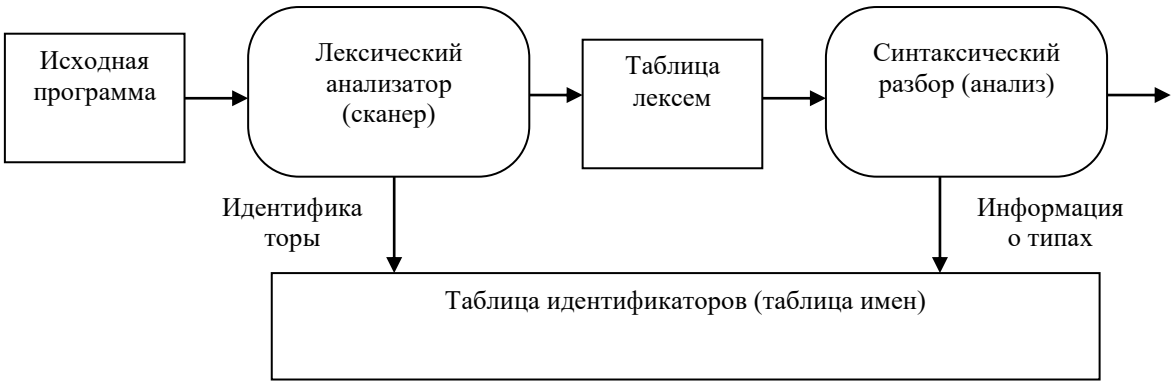
Другой иллюстрацией может служить оператор языка C, имеющий вид: `k=i+++++j;`. Существует только одна единственно верная трактовка этого оператора: `k = i++ + ++j;` (если явно пояснить ее с помощью скобок, то данная конструкция имеет вид: `k = (i++) + (++j);`). Однако найти ее лексический анализатор может, лишь просмотрев весь оператор до конца и перебрав все варианты, причем неверные варианты могут быть обнаружены только на этапе семантического анализа (например, вариант `k = (i++)++ + j;` является синтаксически правильным, но семантикой языка C не допускается). Конечно, чтобы эта конструкция была в принципе допустима, входящие в нее операнды `k`, `i` и `j` должны быть описаны и должны допускать выполнение операций языка `++` и `+`.

Поэтому в большинстве компиляторов лексический и синтаксический анализаторы — это взаимосвязанные части. Возможны два принципиально различных метода организации взаимосвязи лексического и синтаксического анализа:

- последовательный;
- параллельный[2].

При последовательном варианте лексический анализатор просматривает весь текст исходной программы от начала до конца и преобразует его в таблицу лексем. Таблица лексем заполняется сразу полностью, компилятор использует ее для последующих фаз компиляции, но в дальнейшем не изменяет. Дальнейшую обработку таблицы лексем выполняют следующие фазы компиляции. Если в процессе разбора лексический анализатор не смог правильно определить тип лексемы, то считается, что исходная программа содержит ошибку.

Работа синтаксического и лексического анализаторов в варианте их последовательного взаимодействия изображена в виде схемы на 3.1.



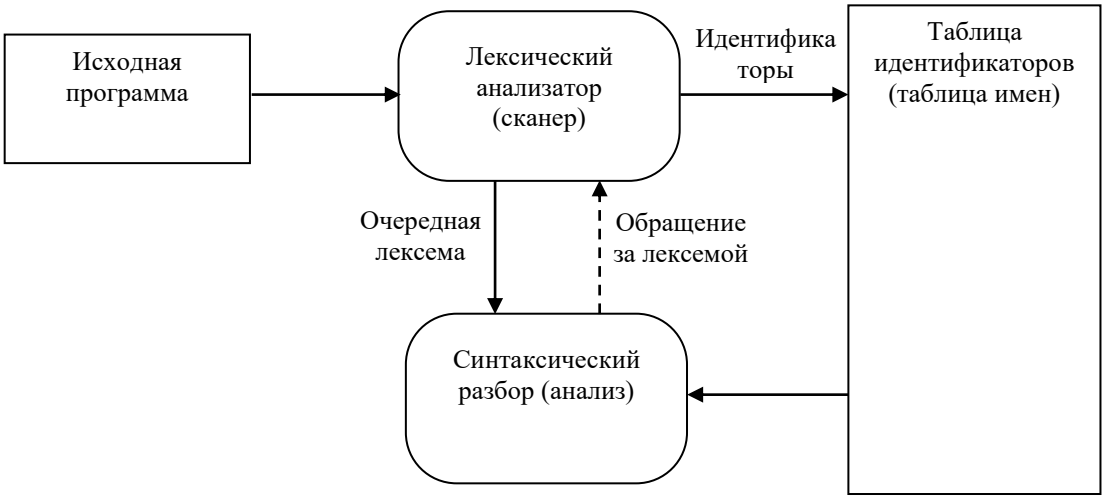
3.1. Последовательное взаимодействие лексического и синтаксического анализаторов

При параллельном варианте лексический анализ текста исходной программы выполняется поэтапно по шагам. Лексический анализатор выделяет очередную лексему в исходном коде и передает ее синтаксическому анализатору. Синтаксический анализатор, выполнив разбор очередной конструкции языка, может подтвердить правильность найденной лексемы и обратиться к лексическому анализатору за следующей лексемой, либо же отвергнуть найденную лексему. Во втором случае он может проинформировать лексический анализатор о том, что надо вернуться назад к уже просмотренному ранее фрагменту исходного кода и сообщить ему дополнительную информацию о том, лексему какого типа следует ожидать. Взаимодействуя между собой таким образом, лексический и синтаксические анализаторы могут перебрать несколько возможных вариантов лексем, и если ни

[2] Параллельный метод работы лексического анализатора и синтаксического разбора вовсе не означает, что они должны будут выполняться как параллельные взаимодействующие процессы. Такой вариант возможен, но не обязателен.

один из них не подойдет, будет считаться, что исходная программа содержит ошибку. Только после того, как синтаксический анализатор успешно выполнит разбор очередной конструкции исходного языка (обычно такой конструкцией является оператор исходного языка), лексический анализатор помещает найденные лексемы в таблицу лексем и в таблицу идентификаторов и продолжает разбор дальше в том же порядке.

Работа синтаксического и лексического анализаторов в варианте их параллельного взаимодействия изображена в виде схемы на 3.2.



3.2. Параллельное взаимодействие лексического и синтаксического анализаторов

Последовательная работа лексического и синтаксического анализаторов, представленная на 3.1, представляет собой самый простой вариант их взаимодействия. Она проще в реализации и обеспечивает более высокую скорость работы компилятора, чем их параллельное взаимодействие, показанное на 3.2. Поэтому разработчики компиляторов стремятся организовать взаимодействие лексического и синтаксического анализаторов именно таким образом.

Для большинства языков программирования границы лексем распознаются по заданным терминальным символам. Эти символы — пробелы, знаки операций, символы комментариев, а также разделители (запятые, точки с запятой и т. п.). Набор таких терминальных символов зависит от синтаксиса входного языка. Важно отметить, что знаки операций и разделители сами также являются лексемами, и необходимо не пропустить их при распознавании текста.

Но для многих языков программирования на этапе лексического анализа может быть недостаточно информации для однозначного определения типа и границ очередной лексемы. Однако даже и тогда разработчики компиляторов стремятся избежать параллельной работы лексического и синтаксического анализаторов. В ряде случаев помогает принцип выбора из всех возможных лексем лексемы наибольшей длины: очередной символ из входного потока данных добавляется в лексему всегда, когда он может быть туда добавлен. Как только символ не может быть добавлен в лексему, то считается, что он является границей лексемы и началом следующей лексемы (если

символ не является пустым разделителем — пробелом, символом табуляции или перевода строки, знаком комментария).

Такой принцип не всегда позволяет правильно определить границы лексем в том случае, когда они не разделены пустыми символами. Например, приведенная выше строка языка C `k = i+++++j;` будет разбита на лексемы следующим образом: `k = i++ ++ + j;` — и это разбиение неверное. Лексический анализатор, разбирая строку из 5 знаков + дважды выбрал лексему наибольшей возможной длины — знак операции инкремента (увеличения значения переменной на 1) ++, хотя это неправильно. Компилятор должен будет выдать пользователю сообщение об ошибке, при том, что правильный вариант распознавания этой строки существует.

Разработчики компиляторов сознательно идут на то, что отсекают некоторые правильные, но не вполне читаемые варианты исходных программ. Попытки усложнить лексический распознаватель неизбежно приведут к необходимости его взаимосвязи с синтаксическим разбором. Это потребует организации их параллельной работы и снизит эффективность работы всего компилятора. Возникшие накладные расходы никак не оправдываются достигаемым эффектом — распознаванием строк с сомнительными лексемами. Достаточно обязать пользователя явно указать с помощью пробелов (или других незначащих символов) границы лексем, что значительно проще[3].

Не для всех входных языков такой подход возможен. Например, для рассмотренного выше примера с языка FORTRAN невозможно применить указанный метод — разница между оператором цикла и оператором присваивания слишком существенна, чтобы ею можно было пренебречь. Здесь придется прибегнуть к взаимосвязи с синтаксическим разбором[4]. В таком случае приходится организовывать параллельную работу лексического и синтаксического анализаторов.

Очевидно, что последовательный вариант организации взаимодействия лексического и синтаксического анализаторов является более эффективным, так как он не требует организации сложных механизмов обмена данными и не нуждается в повторном прочтении уже разобранных лексем. Этот метод является и более простым. Однако не для всех языков программирования возможно организовать такое взаимодействие. Это зависит в основном от синтаксиса языка, заданного его грамматикой. Большинство современных широко распространенных языков программирования, таких как C, C++, Java и Object Pascal, тем не менее, позволяют построить лексический анализ по более простому, последовательному методу.

---

[3] Желающие могут воспользоваться любым доступным компилятором C и проверить, насколько он способен разобрать приведенный здесь пример.

[4] Приведенные здесь два оператора на языке FORTRAN, различающиеся только на один символ — «.» (точка) или «,» (запятая) — представляют собой не столько проблему для компилятора, сколько для программиста, поскольку позволяют допустить очень неприятную и трудно обнаруживаемую ошибку [6].

Выполнение действий, связанных с лексемами

Выполнение действий в процессе распознавания лексем представляет для лексического анализатора гораздо меньшую проблему, чем определение границ лексем. Фактически конечный автомат, который лежит в основе лексического анализатора, должен иметь не только входной язык, но и выходной. Он должен не только уметь распознать правильную лексему на входе, но и породить связанную с ней последовательность символов на выходе. В такой конфигурации конечный автомат преобразуется в конечный преобразователь [3, 4 т.1, 5, 29, 34].

Для лексического анализатора действия по обнаружению лексемы могут трактоваться несколько шире, чем только порождение цепочки символов выходного языка. Он должен уметь выполнять такие действия, как запись найденной лексемы в таблицу лексем, поиск ее в таблице идентификаторов и запись новой лексемы в таблицу идентификаторов. Набор действий определяется реализацией компилятора. Обычно эти действия выполняются сразу же при обнаружении конца распознаваемой лексемы.

В конечном автомате, лежащем в основе лексического анализатора, эти действия можно отобразить достаточно просто — достаточно иметь возможность с каждым переходом на графе автомата (или в функции переходов автомата) связать выполнение некоторой произвольной функции f(q,a), где q — текущее состояние автомата, a — текущий входной символ. Функция f(q,a) может выполнять любые действия, доступные лексическому анализатору:

- помещать новую лексему в таблицу лексем;
- проверять наличие найденной лексемы в таблице идентификаторов;
- добавлять новую лексему в таблицу идентификаторов;
- выдавать сообщения пользователю о найденных ошибках и предупреждения об обнаруженных неточностях в программе;
- прерывать процесс компиляции.

Возможны и другие действия, предусмотренные реализацией компилятора. Такую функцию f(q,a), если она есть, обычно записывают на графе переходов конечного автомата под дугами, соединяющими состояния автомата. Функция f(q,a) может быть пустой (не выполнять никаких действий), тогда соответствующая запись отсутствует.

Регулярные языки и грамматики

Чтобы перейти к примерам реализации лексических анализаторов, необходимо более подробно рассмотреть регулярные языки и грамматики, лежащие в их основе.

Регулярные и автоматные грамматики

Регулярные грамматики

К регулярным, как уже было сказано, относятся два типа грамматик: леволинейные и праволинейные.

Леволинейные грамматики G(**VT**,**VN**,**P**,S), **V** = **VN**∪**VT** могут иметь правила двух видов: A → Bγ или A → γ, где A,B∈**VN**, γ∈**VT***.

В свою очередь праволинейные грамматики G(**VT**,**VN**,**P**,S), **V** = **VN**∪**VT** могут иметь правила также двух видов: A → γB или A → γ, где A,B∈**VN**, γ∈**VT***.

Доказано, что эти два класса грамматик эквивалентны. Для любого регулярного языка, заданного праволинейной грамматикой, может быть построена леволинейная грамматика, определяющая эквивалентный язык; и наоборот — для любого регулярного языка, заданного леволинейной грамматикой, может быть построена праволинейная грамматика, задающая эквивалентный язык.

Разница между леволинейными и праволинейными грамматиками заключается, в основном, в том, в каком порядке строятся предложения языка: слева направо для леволинейных, либо справа налево для праволинейных. Поскольку предложения языков программирования строятся, как правило, в порядке слева направо, то в дальнейшем в разделе регулярных грамматик будет идти речь в первую очередь о леволинейных грамматиках.

Автоматные грамматики

Среди всех регулярных грамматик можно выделить отдельный класс — автоматные грамматики. Они также могут быть леволинейными и праволинейными.

Леволинейные автоматные грамматики G(**VT**,**VN**,**P**,S), **V** = **VN**∪**VT** могут иметь правила двух видов: A → Bt или A → t, где A,B∈**VN**, t∈**VT**.

Праволинейные автоматные грамматики G(**VT**,**VN**,**P**,S), **V** = **VN**∪**VT** могут иметь правила двух видов: A → tB или A → t, где A,B∈**VN**, t∈**VT**.

Разница между автоматными грамматиками и обычными регулярными грамматиками заключается в следующем: там, где в правилах обычных регулярных грамматик может присутствовать цепочка терминальных символов, в автоматных грамматиках может присутствовать только один терминальный символ. Любая автоматная грамматика является регулярной, но не наоборот — не всякая регулярная грамматика является автоматной.

Доказано, что классы обычных регулярных грамматик и автоматных грамматик почти эквивалентны. Это значит, что для любого языка, который задан регулярной грамматикой, можно построить автоматную грамматику, определяющую почти эквивалентный язык (обратное утверждение очевидно).

Чтобы классы автоматных и регулярных грамматик были полностью эквивалентны, в автоматных грамматиках разрешается дополнительное правило вида S→λ, где S — целевой символ грамматики. При этом символ S не должен встречаться в правых частях других правил грамматики. Тогда язык, заданный автоматной грамматикой **G**, может включать в себя пустую цепочку: λ∈L(**G**).

Автоматные грамматики, также как обычные леволинейные и праволинейные грамматики, задают регулярные языки. Поскольку реально используемые языки, как правило, не содержат пустую цепочку символов, разница на пустую цепочку между этими двумя типами грамматик значения не имеет, и правила вида S → λ далее рассматриваться не будут.

Существует алгоритм, который позволяет преобразовать произвольную регулярную грамматику к автоматному виду — то есть построить эквивалентную ей автоматную грамматику. Этот алгоритм рассмотрен ниже. Он является исключительно полезным, поскольку позволяет существенно облегчить построение распознавателей для регулярных грамматик.

Преобразование регулярной грамматики к автоматному виду

Имеется регулярная грамматика G(**VT**,**VN**,**P**,S), необходимо преобразовать ее в почти эквивалентную автоматную грамматику G'(**VT**,**VN'**,**P'**,S'). Как уже было сказано выше, будем рассматривать леволинейные грамматики (для праволинейных грамматик можно легко построить аналогичный алгоритм).

Алгоритм преобразования прост и заключается он в следующей последовательности действий:

Шаг 1. Все нетерминальные символы из множества **VN** грамматики **G** переносятся во множество **VN'** грамматики **G'**.

Шаг 2. Необходимо просматривать все множество правил **P** грамматики **G**.

Если встречаются правила вида $A \to Ba_1$, $A,B \in \mathbf{VN}$, $a_1 \in \mathbf{VT}$ или вида $A \to a_1$, $A \in \mathbf{VN}$, $a_1 \in \mathbf{VT}$, то они переносятся во множество **P'** правил грамматики **G'** без изменений.

Если встречаются правила вида $A \to Ba_1a_2\ldots a_n$, $n > 1$, $A,B \in \mathbf{VN}$, $\forall n > i > 0$: $a_i \in \mathbf{VT}$, то во множество нетерминальных символов **VN'** грамматики **G'** добавляются символы $A_1,A_2,\ldots,A_{n-1}$, а во множество правил **P'** грамматики **G'** добавляются правила:

$A \to A_{n-1}a_n$

$A_{n-1} \to A_{n-2}a_{n-1}$

…

$A_2 \to A_1a_2$

$A_1 \to Ba_1$

Если встречаются правила вида $A \to a_1a_2\ldots a_n$, $n > 1$, $A \in \mathbf{VN}$, $\forall n > i > 0$: $a_i \in \mathbf{VT}$, то во множество нетерминальных символов **VN'** грамматики **G'** добавляются символы $A_1,A_2,\ldots,A_{n-1}$, а во множество правил **P'** грамматики **G'** добавляются правила:

$A \to A_{n-1}a_n$

$A_{n-1} \to A_{n-2}a_{n-1}$

…

$A_2 \to A_1a_2$

$A_1 \to a_1$

Если встречаются правила вида $A \to B$ или вида $A \to \lambda$, то они переносятся во множество правил **P'** грамматики **G'** без изменений.

Шаг 3. Просматривается множество правил **P'** грамматики **G'**. В нем ищутся правила вида $A \to B$ или вида $A \to \lambda$.

Если находится правило вида $A \to B$, то просматривается множество правил **P'** грамматики **G'**. Если в нем присутствует правила вида $B \to C$, $B \to Ca$, $B \to a$ или $B \to \lambda$, то в него добавляются правила вида $A \to C$, $A \to Ca$, $A \to a$ и $A \to \lambda$ соответственно, $\forall A,B,C \in \mathbf{VN'}$, $\forall a \in \mathbf{VT'}$. При этом следует учитывать, что в грамматике не должно быть совпадающих правил, и если какое-то правило уже присутствует в грамматике **G'**, то повторно его туда добавлять не следует. Правило $A \to B$ удаляется из множества правил **P'**.

Если находится правило вида $A \to \lambda$ (и символ A не является целевым символом S), то просматривается множество правил **P'** грамматики **G'**. Если в нем присутствует правила вида $B \to A$ или $B \to Aa$, то в него добавляются правила вида $B \to \lambda$ и $B \to a$ соответственно, $\forall A,B \in \mathbf{VN'}$, $\forall a \in \mathbf{VT'}$ (при этом также следует учитывать, что в грамматике не должно быть совпадающих правил). Правило $A \to \lambda$ удаляется из множества правил **P'**.

Шаг 4. Если на шаге 3 было найдено хотя бы одно правило вида $A \to B$ или $A \to \lambda$ во множестве правил **P'** грамматики **G'**, то надо повторить шаг 3, иначе перейти к шагу 5.

Шаг 5. Целевым символом S' грамматики **G'** становится символ S.

Шаги 3 и 4 алгоритма, в принципе, можно не выполнять, если грамматика не содержит правил вида $A \to B$ (такие правила называются цепными) или вида $A \to \lambda$ (такие правила называются λ-правилами). Реальные регулярные грамматики обычно не содержат правил такого вида. Тогда алгоритм преобразования грамматики к автоматному виду существенно упрощается. Кроме того, эти правила можно было бы предварительно устранить с помощью специальных алгоритмов преобразования грамматик (эти алгоритмы рассмотрены в главе «Синтаксические анализаторы»).

Пример преобразования регулярной грамматики к автоматному виду

Рассмотрим в качестве примера следующую простейшую регулярную грамматику: `G({"a","(","*",")","{","}"},{S,C,K},P,S)` (символы `a, (, *, ), {, }` из множества терминальных символов грамматики взяты в кавычки, чтобы выделить их среди фигурных скобок, обозначающих само множество):

P:

`S → C*) | K}`

`C → (* | Ca | C{ | C} | C( | C* | C)`

`K → { | Ka | K( | K* | K) | K{`

Если предположить, что `a` здесь — это любой алфавитно-цифровой символ, кроме символов `(, *, ), {, }`, то эта грамматика описывает два типа комментариев, допустимых в языке программирования Borland Pascal. Преобразуем ее в автоматный вид.

Шаг 1. Построим множество **VN'** = {S, C, K}.

Шаг 2. Начинаем просматривать множество правил **P** грамматики **G**.

Для правила S → C*) во множество **VN'** включаем символ S_1, а само правило разбиваем на два: S → S_1) и S_1 → C*; включаем эти правила во множество правил **P'**.

Правило S → K} переносим во множество правил **P'** без изменений.

Для правила C → (* во множество **VN'** включаем символ C_1, а само правило разбиваем на два: C → C_1* и C_1 → (; включаем эти два правила во множество правил **P'.**

Правила C → Ca | C{ | C} | C(| C* | C) переносим во множество правил **P'** без изменений.

Правила K → { | Ka | K(| K* | K) | K{ переносим во множество правил **P'** без изменений.

Шаг 3. Правил вида A→B или A→λ во множестве правил **P'** не содержится.

Шаг 4. Переходим к шагу 5.

Шаг 5. Целевым символом грамматики **G '** становится символ S.

В итоге получаем автоматную грамматику:

G'({"a","(","*",")","{","}"}, {S,S_1,C,C_1,K},**P'**,S):

P':

S → S_1) | K}

S_1 → C*

C → C_1* | Ca | C{ | C} | C(| C* | C)

C_1 → (

K → { | Ka | K(| K* | K) | K{

Эта грамматика, так же как и рассмотренная выше, описывает два типа комментариев, допустимых в языке программирования Borland Pascal.

Конечные автоматы

Определение конечного автомата

Конечным автоматом (КА) называют пятерку следующего вида:

M(**Q**, **V**, δ, q_0, **F**),

где

Q — конечное множество состояний автомата;

V — конечное множество допустимых входных символов (алфавит автомата);

δ — функция переходов, отображающая $\mathbf{V}^*\mathbf{Q}$ (декартово произведение множеств) во множество подмножеств $\mathbf{Q}$: R($\mathbf{Q}$), то есть $\delta(a,q)=\mathbf{R}$, $a\in\mathbf{V}$, $q\in\mathbf{Q}$, $\mathbf{R}\subseteq\mathbf{Q}$;

q_0 — начальное состояние автомата $\mathbf{Q}$, $q_0\in\mathbf{Q}$;

$\mathbf{F}$ — непустое множество конечных состояний автомата, $\mathbf{F}\subseteq\mathbf{Q}$, $\mathbf{F}\neq\varnothing$.

КА называют *полностью определенным*, если в каждом его состоянии существует функция перехода для всех возможных входных символов, то есть: $\forall a\in\mathbf{V}$, $\forall q\in\mathbf{Q}$ $\exists\delta(a,q)=\mathbf{R}$, $\mathbf{R}\subseteq\mathbf{Q}$.

Работа конечного автомата представляет собой последовательность шагов (или тактов). На каждом шаге работы автомат находится в одном из своих состояний $q\in\mathbf{Q}$ (в текущем состоянии), на следующем шаге он может перейти в другое состояние или остаться в текущем состоянии. То, в какое состояние автомат перейдет на следующем шаге работы, определяет функция переходов δ. Она зависит не только от текущего состояния q, но и от символа a из алфавита $\mathbf{V}$, поданного на вход автомата. Когда функция перехода допускает несколько следующих состояний автомата, то КА может перейти в любое из этих состояний. В начале работы автомат всегда находится в начальном состоянии q_0. Работа КА продолжается до тех пор, пока на его вход поступают символы из входной цепочки $\omega\in\mathbf{V}^+$.

Видно, что конфигурацию КА на каждом шаге работы можно определить в виде (q,ω,n), где q — текущее состояние автомата, $q\in\mathbf{Q}$; ω — цепочка входных символов, $\omega\in\mathbf{V}^+$; n — положение указателя в цепочке символов, $n\in\mathbf{N}\cup\{0\}$, $n\le|\omega|$ ($\mathbf{N}$ — множество натуральных чисел). Конфигурация автомата на следующем шаге — это $(q',\omega,n+1)$, если $q'\in\delta(a,q)$ и символ $a\in\mathbf{V}$ находится в позиции n+1 цепочки ω. Начальная конфигурация автомата: $(q_0,\omega,0)$; заключительная конфигурация автомата: (f,ω,n), $f\in\mathbf{Q}$, $n=|\omega|$, она является конечной конфигурацией, если $f\in\mathbf{F}$.

Язык, заданный конечным автоматом

КА $M(\mathbf{Q},\mathbf{V},\delta,q_0,\mathbf{F})$ *принимает цепочку символов* $\omega\in\mathbf{V}^+$, если, получив на вход эту цепочку, он из начального состояния q_0 может перейти в одно из конечных состояний $f\in\mathbf{F}$. В противном случае КА не принимает цепочку символов.

Язык L($\mathbf{M}$), заданный КА $M(\mathbf{Q},\mathbf{V},\delta,q_0,\mathbf{F})$ — это множество всех цепочек символов, которые принимаются этим автоматом.

Два КА $\mathbf{M}$ и $\mathbf{M}'$ *эквивалентны*, если они задают один и тот же язык: $\mathbf{M}\equiv\mathbf{M}' \Leftrightarrow L(\mathbf{M})=L(\mathbf{M}')$.

Все КА являются распознавателями для регулярных языков [4 т.1, 5, 15, 29].

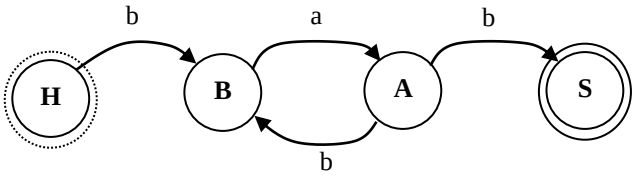
Граф переходов конечного автомата

КА часто представляют в виде диаграммы или графа переходов автомата.

Граф переходов КА — это направленный граф, вершины которого помечены символами состояний КА, и в котором есть дуга (p,q) $p,q\in\mathbf{Q}$, помеченная символом $a\in\mathbf{V}$, если в КА определена $\delta(a,p)$ и $q\in\delta(a,p)$. Начальное и конечные состояния автомата на графе состояний помечаются специальным образом (в данном пособии

начальное состояние — дополнительной пунктирной линией, конечное состояние — дополнительной сплошной линией).

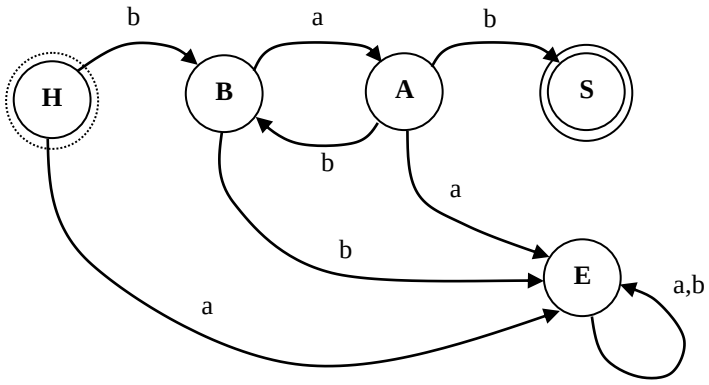
Рассмотрим конечный автомат: `M({H,A,B,S},{a,b},δ,H,{S});` `δ: δ(H,b)=B, δ(B,a)=A, δ(A,b)={B,S}`. Ниже на 3.3 приведен пример графа состояний для этого КА



3.3. Граф переходов недетерминированного конечного автомата

Для моделирования работы КА его удобно привести к полностью определенному виду, чтобы исключить ситуации, из которых нет переходов по входным символам. Для этого в КА добавляют еще одно состояние, которое можно условно назвать «ошибка». На это состояние замыкают все неопределенные переходы, а все переходы из самого состояния «ошибка» замыкают на него же.

Если преобразовать подобным образом рассмотренный выше автомат **M**, то получим полностью определенный автомат: `M({H,A,B,E,S},{a,b},δ,H,{S});` `δ: δ(H,a)=E, δ(H,b)=B, δ(B,a)=A, δ(B,b)=E, δ(A,a)={E},` `δ(A,b)={B,S}, δ(E,a)={E}, δ(E,b)={E}, δ(S,a)={E}, δ(S,b)={E}`. Состояние E как раз соответствует состоянию «ошибка». Граф переходов этого КА представлен на 3.4.



3.4. Граф переходов полностью определенного недетерминированного конечного автомата

Детерминированные и недетерминированные конечные автоматы

Определение детерминированного конечного автомата

Конечный автомат M(**Q**,**V**,δ,q_0,**F**) называют *детерминированным конечным автоматом* (ДКА), если в каждом из его состояний для любого входного символа функция перехода содержит не более одного состояния: ∀a∈**V**, ∀q∈**Q**: либо δ(a,q)={r}, r∈**Q**, либо δ(a,q)=∅.

В противном случае конечный автомат называют *недетерминированным*.

Из этого определения видно, что автоматы, представленные ранее на 3.3 и 3.4 являются недетерминированными КА.

ДКА может быть задан в виде пятерки:

M(**Q**, **V**, δ, q_0, **F**),

где

Q — конечное множество состояний автомата;

V — конечное множество допустимых входных символов;

δ — функция переходов, отображающая **V*Q** в множество **Q**: δ(a,q)=r, a∈**V**, q,r∈**Q**;

q_0 — начальное состояние автомата **Q**, q_0∈**Q**;

F — непустое множество конечных состояний автомата, **F**⊆**Q**, **F** ≠ ∅.

Если функция переходов ДКА определена для каждого состояния автомата, то автомат называется *полностью определенным* ДКА: ∀a∈**V**, ∀q∈**Q**: ∃δ(a,q)=r, r∈**Q**.

Доказано, что для любого КА можно построить эквивалентный ему ДКА. Моделировать работу ДКА существенно проще, чем работу произвольного КА, поэтому произвольный КА стремятся преобразовать в ДКА. При построении компиляторов чаще всего используют полностью определенный ДКА.

Преобразование конечного автомата к детерминированному виду

Алгоритм преобразования произвольного КА M(**Q**,**V**,δ,q_0,**F**) в эквивалентный ему ДКА M'(**Q'**,**V**,δ',q'_0,**F'**) заключается в следующем:

1. Множество состояний **Q'** автомата **M'** строится из комбинаций всех состояний множества **Q** автомата **M**. Если $q_1,q_2,\ldots,q_n$, n>0 — состояния автомата **M**, ∀0<i≤n q_i∈**Q**, то всего будет 2^n-1 состояний автомата **M'**. Обозначим их так: $[q_1q_2\ldots q_m]$, 0<m≤n.
2. Функция переходов δ' автомата **M'** строится так: $\delta'(a,[q_1q_2\ldots q_m])=[r_1r_2\ldots r_k]$, где ∀ 0< i≤ m ∃ 0<j≤ k так, что r_j∈δ(a,q_i);
3. Обозначим $q'_0 = [q_0]$;
4. Пусть $f_1,f_2,\ldots,f_l$, l>0 — конечные состояния автомата **M**, ∀0<i≤l f_i∈**F**, тогда множество конечных состояний **F'** автомата **M'** строится из всех состояний, имеющих вид […f_i…], f_i∈**F**.

Доказано, что описанный выше алгоритм строит ДКА, эквивалентный заданному произвольному КА [4 т.1, 29].

После построения из нового ДКА необходимо удалить все недостижимые состояния.

Состояние $q \in \mathbf{Q}$ в КА $M(\mathbf{Q},\mathbf{V},\delta,q_0,\mathbf{F})$ называется недостижимым, если ни при какой входной цепочке $\omega \in \mathbf{V}^+$ невозможен переход автомата из начального состояния q_0 в состояние q. Иначе состояние называется достижимым.

Для работы алгоритма удаления недостижимых состояний используются два множества: множество достижимых состояний **R** и множество текущих активных состояний на каждом шаге алгоритма $\mathbf{P}_i$. Результатом работы алгоритма является полное множество достижимых состояний **R**. Рассмотрим работу алгоритма по шагам:

1. $\mathbf{R}:=\{q_0\}$; i:=0; $\mathbf{P}_0:=\{q_0\}$;
2. $\mathbf{P}_{i+1}:=\varnothing$;
3. $\forall a \in \mathbf{V}$, $\forall q \in \mathbf{P}_i$: $\mathbf{P}_{i+1}:=\mathbf{P}_{i+1} \cup \delta(a,q)$;
4. Если $\mathbf{P}_{i+1}-\mathbf{R}=\varnothing$, то выполнение алгоритма закончено, иначе $\mathbf{R}:=\mathbf{R} \cup \mathbf{P}_{i+1}$, i:=i+1 и перейти к шагу 3.

После выполнения данного алгоритма из КА можно исключить все состояния, не входящие в построенное множество **R**.

Рассмотрим работу алгоритма преобразования произвольного КА в ДКА на примере автомата `M({H,A,B,S},{a,b},δ,H,{S});` `δ: δ(H,b)=B,` `δ(B,a)=A,` `δ(A,b)={B,S}`. Видно, что это недетерминированный КА (из состояния `A` возможны два различных перехода по символу `b`). Граф переходов для этого автомата был изображен выше на 3.3.

Построим множество состояний эквивалентного ДКА:

```
Q'={[H],[A],[B],[S],[HA],[HB],[HS],[AB],[AS],[BS],[HAB],[HA
S],[HBS],[ABS],[HABS]}.
```

Построим функцию переходов эквивалентного ДКА:

```
δ'([H],b)=[B]

δ'([A],b)=[BS]

δ'([B],a)=[A]

δ'([HA],b)=[BS]

δ'([HB],a)=[A]

δ'([HB],b)=[B]

δ'([HS],b)=[B]

δ'([AB],a)=[A]

δ'([AB],b)=[BS]
```

δ'([AS],b)=[BS]

δ'([BS],a)=[A]

δ'([HAB],a)=[A]

δ'([HAB],b)=[BS]

δ'([HAS],b)=[BS]

δ'([HBS],b)=[B]

δ'([HBS],a)=[A]

δ'([ABS],b)=[BS]

δ'([ABS],a)=[A]

δ'([HABS],a)=[A]

δ'([HABS],b)=[BS]

Начальное состояние эквивалентного ДКА:

q_0' = [H]

Множество конечных состояний эквивалентного ДКА:

F' = {[S],[HS],[AS],[BS],[HAS],[HBS],[ABS],[HABS]}

После построения ДКА исключим недостижимые состояния. Множество достижимых состояний ДКА будет следующим R={[H],[B],[A],[BS]}. В итоге, исключив все недостижимые состояния, получим ДКА:

M'({[H],[B],[A],[BS]},{a,b},[H],{[BS]}),

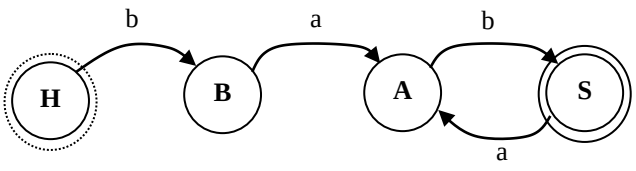
δ([H],b)=[B], δ([B],a)=[A], δ([A],b)=[BS], δ([BS],a)=[A].

Ничего не изменяя, переобозначим состояния ДКА. Получим:

M'({H,B,A,S},{a,b},H,{S}),

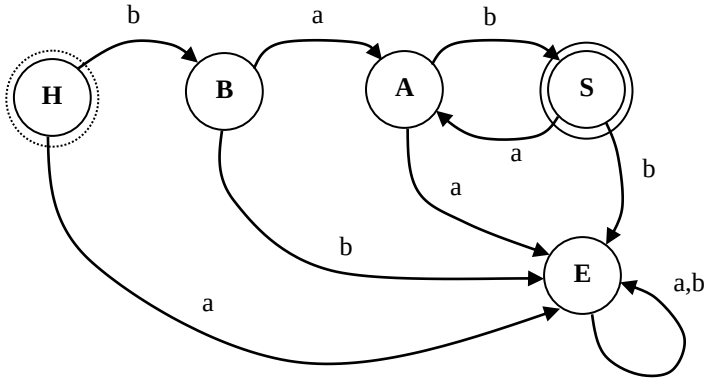
δ(H,b)=B, δ(B,a)=A, δ(A,b)=S, δ(S,a)=A.

Граф переходов полученного ДКА изображен на 3.5.



3.5. Граф переходов детерминированного конечного автомата

Этот автомат можно преобразовать к полностью определенному виду. Получим граф состояний, изображенный на 3.6 (состояние E — это состояние «ошибка»).



3.6. Граф переходов полностью определенного детерминированного конечного автомата

ВНИМАНИЕ

При построении распознавателей к вопросу о необходимости преобразования КА в ДКА надо подходить, основываясь на принципе разумной достаточности. Моделировать работу ДКА существенно проще, чем произвольного КА, но при выполнении преобразования число состояний автомата может существенно возрасти и, в худшем случае, составит $2^n–1$, где n — количество состояний исходного КА. В этом случае затраты на моделирование ДКА окажутся больше, чем на моделирование исходного КА.

Поэтому не всегда выполнение преобразования автомата к детерминированному виду является обязательным.

Минимизация конечных автоматов

Многие КА можно минимизировать. *Минимизация* КА заключается в построении эквивалентного КА с меньшим числом состояний. В процессе минимизации необходимо построить автомат с минимально возможным числом состояний, эквивалентный данному КА.

Для минимизации автомата используется алгоритм построения эквивалентных состояний КА. Два различных состояния в конечном автомате $M(\mathbf{Q},\mathbf{V},\delta,q_0,\mathbf{F})$ $q\in\mathbf{Q}$ и $q'\in\mathbf{Q}$ называются n-эквивалентными (n-неразличимыми), $n\geq 0$ $n\in\mathbf{N}\cup\{0\}$, если, находясь в одном из этих состояний и получив на вход любую цепочку символов ω: $\omega\in\mathbf{V}^*$ $|\omega| \leq n$, автомат может перейти в одно и то же множество конечных состояний. Очевидно, что 0-эквивалентными состояниями автомата $M(\mathbf{Q},\mathbf{V},\delta,q_0,\mathbf{F})$ являются два множества его состояний: **F** и **Q**–**F**. Множества эквивалентных состояний автомата называют классами эквивалентности, а всю их совокупность — множеством классов эквивалентности $\mathbf{R}(n)$, причем: $R(0) = \{\mathbf{F},\mathbf{Q}–\mathbf{F}\}$.

Рассмотрим работу алгоритма построения эквивалентных состояний по шагам:

1. На первом шаге n:=0, строим R(0);
2. На втором шаге n:=n+1, строим R(n) на основе R(n–1): R(n)={r_i(n): {q_{ij}∈**Q**: ∀a∈**V** δ(a,q_{ij})⊆r_j(n–1)} ∀i,j∈**N**}. То есть в классы эквивалентности на шаге n входят те состояния, которые по одинаковым символам переходят в n–1 эквивалентные состояния.
3. Если R(n)=R(n–1), то работа алгоритма закончена, иначе необходимо вернуться к шагу 2.

Доказано, что алгоритм построения множества классов эквивалентности завершится максимум для n=m–2, где m — общее количество состояний автомата.

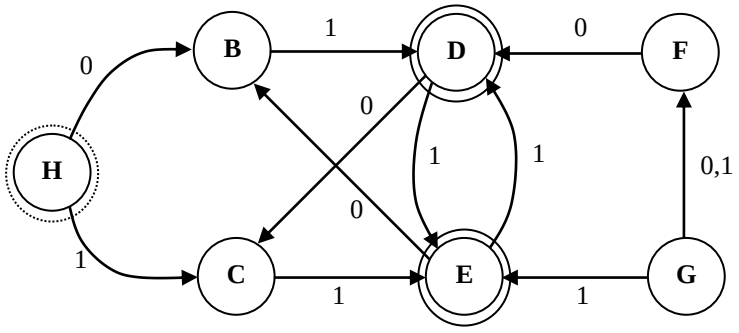
Алгоритм минимизации КА заключается в следующем:

1. Из автомата исключаются все недостижимые состояния.
2. Строятся классы эквивалентности автомата.
3. Классы эквивалентности состояний исходного КА становятся состояниями результирующего минимизированного КА.
4. Функция переходов результирующего КА очевидным образом строится на основе функции переходов исходного КА.

Для этого алгоритма доказано: во-первых, что он строит минимизированный КА, эквивалентный заданному; во-вторых, что он строит КА с минимально возможным числом состояний (минимальный КА).

Рассмотрим пример: задан автомат `M({A,B,C,D,E,F,G},{0,1},δ,A,{D,E})`, `δ: δ(A,0)={B}, δ(A,1)={C}, δ(B,1)={D}, δ(C,1)={E}, δ(D,0)={C}, δ(D,1)={E}, δ(E,0)={B}, δ(E,1)={D}, δ(F,0)={D}, δ(F,1)={G}, δ(G,0)={F}, δ(G,1)={F}`; необходимо построить эквивалентный ему минимальный КА.

Граф переходов этого автомата приведен на 3.7.



3.7. Граф переходов конечного автомата до его минимизации

Состояния F и G являются недостижимыми, они будут исключены на первом шаге алгоритма. Построим классы эквивалентности автомата:

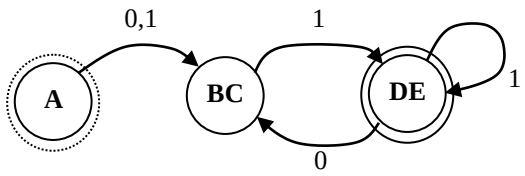
R(0)={{A,B,C},{D,E}}, n=0;

R(1)={{A},{B,C},{D,E}}, n=1;

R(2)={{A},{B,C},{D,E}}, n=2.

Обозначим соответствующим образом состояния полученного минимального КА и построим автомат: M({A,BC,DE},{0,1},δ',A,{DE}), δ': δ'(A,0)={BC}, δ'(A,1)={BC}, δ'(BC,1)={DE}, δ'(DE,0)={BC}, δ'(DE,1)={DE}.

Граф переходов минимального КА приведен на 3.8.



3.8. Граф переходов конечного автомата после его минимизации

Минимизация конечных автоматов позволяет при построении распознавателей получить автомат с минимально возможным числом состояний и, тем самым, в дальнейшим упростить функционирование распознавателя.

Регулярные множества и регулярные выражения

Определение регулярного множества

Определим над множествами цепочек символов из алфавита **V** операции конкатенации и итерации следующим образом:

PQ — конкатенация $\mathbf{P}\in\mathbf{V}^*$ и $\mathbf{Q}\in\mathbf{V}^*$: **PQ** = {pq $\forall p\in\mathbf{P}$, $\forall q\in\mathbf{Q}$};

$\mathbf{P}^*$ — итерация $\mathbf{P}\in\mathbf{V}^*$: $\mathbf{P}^* = \{p^n\ \forall p\in\mathbf{P}, \forall n\geq 0\}$.

Тогда для алфавита **V** регулярные множества определяются рекурсивно:

1. ∅ — регулярное множество;
2. {λ} — регулярное множество;
3. {a} — регулярное множество $\forall a\in\mathbf{V}$;
4. Если **P** и **Q** — произвольные регулярные множества, то множества $\mathbf{P}\cup\mathbf{Q}$, **PQ** и $\mathbf{P}^*$ также являются регулярными множествами;
5. Ничто другое не является регулярным множеством.

Фактически регулярные множества — это множества цепочек символов над заданным алфавитом, построенные определенным образом (с использованием операций объединения, конкатенации и итерации).

Все регулярные языки представляют собой регулярные множества [4 т.1, 29, 58].

Регулярные выражения. Свойства регулярных выражений

Регулярные множества можно обозначать с помощью регулярных выражений.

Эти обозначения вводятся следующим образом:

1. 0 — регулярное выражение, обозначающее $\varnothing$;
2. λ — регулярное выражение, обозначающее $\{\lambda\}$;
3. a — регулярное выражение, обозначающее $\{a\}$ $\forall a \in \mathbf{V}$;
4. если p и q — регулярные выражения, обозначающие регулярные множества **P** и **Q**, то p+q, pq, p^* — регулярные выражения, обозначающие регулярные множества $\mathbf{P} \cup \mathbf{Q}$, **PQ** и $\mathbf{P}^*$ соответственно.

ПРИМЕЧАНИЕ

> Иногда для удобства обозначений вводят также операцию непустой итерации, которая обозначается p^+ и для любого регулярного выражения p справедливо: $p^+=pp^*=p^*p$

Два регулярных выражения α и β равны $\alpha=\beta$, если они обозначают одно и то же множество.

Каждое регулярное выражение обозначает одно и только одно регулярное множество, но для одного регулярного множества может существовать сколь угодно много регулярных выражений, обозначающих это множество.

При записи регулярных выражений будут использоваться круглые скобки, как и для обычных арифметических выражений. При отсутствии скобок операции выполняются слева направо с учетом приоритета. Приоритет для операций принят следующий: первой выполняется итерация (высший приоритет), затем конкатенация, потом — объединение множеств (низший приоритет).

Если α, β и γ — регулярные выражения, то свойства регулярных выражений можно записать в виде следующих формул:

1. $\lambda + \alpha\alpha^* = \lambda + \alpha^*\alpha = \alpha^*$ (или, что то же самое: $\lambda + \alpha^+ = \alpha^*$)
2. $\alpha + \beta = \beta + \alpha$.
3. $\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$
4. $\alpha(\beta + \gamma) = \alpha\beta + \alpha\gamma$
5. $(\beta + \gamma)\alpha = \beta\alpha + \gamma\alpha$
6. $\alpha(\beta\gamma) = (\alpha\beta)\gamma$

7. $\alpha + \alpha = \alpha$
8. $\alpha + \alpha^* = \alpha^*$
9. $\lambda + \alpha^* = \alpha^* + \lambda = \alpha^*$
10. $0^* = \lambda$
11. $0\alpha = \alpha 0 = 0$
12. $0 + \alpha = \alpha + 0 = \alpha$
13. $\lambda\alpha = \alpha\lambda = \alpha$
14. $(\alpha^*)^* = \alpha^*$

Все эти свойства можно легко доказать, основываясь на теории множеств, так как регулярные выражения — это только обозначения для соответствующих множеств.

Следует также обратить внимание на то, что среди прочих свойств отсутствует равенство $\alpha\beta = \beta\alpha$, то есть операция конкатенации не обладает свойством коммутативности. Это и не удивительно, поскольку для этой операции важен порядок следования символов.

Уравнения с регулярными коэффициентами

На основе регулярных выражений можно построить уравнения с регулярными коэффициентами [5, 15, 24]. Простейшие уравнения с регулярными коэффициентами будут выглядеть следующим образом:

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

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

где $\alpha,\beta \in \mathbf{V}^*$ — регулярные выражения над алфавитом $\mathbf{V}$, а переменная $X \notin \mathbf{V}$.

Решениями таких уравнений будут регулярные множества. Это значит, что если взять регулярное множество, являющееся решением уравнения, обозначить его в виде соответствующего регулярного выражения и подставить в уравнение, то получим тождественное равенство. Два вида записи уравнений (правосторонняя и левосторонняя запись) связаны с тем, что для регулярных выражений операция конкатенации не обладает свойством коммутативности, поэтому коэффициент можно записать как справа, так и слева от переменной, и при этом получатся различные уравнения. Обе записи равноправны.

Решением первого уравнения является множество, обозначенное регулярным выражением $\alpha^*\beta$. Проверим это решение, подставив его в уравнение вместо переменной X:

$$\alpha X + \beta = \alpha(\alpha^*\beta) + \beta =^{6} (\alpha\alpha^*)\beta + \beta =^{13} (\alpha\alpha^*)\beta + \lambda\beta =^{5} (\alpha\alpha^* + \lambda)\beta =^{1} \alpha^*\beta = X$$

Над знаками равенства указаны номера свойств регулярных выражений, которые были использованы для выполнения преобразований.

Решением второго уравнения является множество, обозначенное регулярным выражением $\beta\alpha^*$.

$$X\alpha + \beta = (\beta\alpha^*)\alpha + \beta =^6 \beta(\alpha^*\alpha) + \beta =^{13} \beta(\alpha^*\alpha) + \beta\lambda =^4 \beta(\alpha^*\alpha + \lambda) =^1 \beta\alpha^* = X$$

ПРИМЕЧАНИЕ

> Указанные решения уравнений не всегда являются единственными. Однако доказано, что $X = \alpha^*\beta$ и $X = \beta\alpha^*$ — это наименьшие из возможных решений для данных двух уравнений. Эти решения называются наименьшей подвижной точкой [4 т.1, 15].

Из уравнений с регулярными коэффициентами можно формировать систему уравнений с регулярными коэффициентами. Система уравнений с регулярными коэффициентами имеет вид (правосторонняя запись):

$$X_1 = \alpha_{10} + \alpha_{11}X_1 + \alpha_{12}X_2 + \ldots + \alpha_{1n}X_n$$

$$X_2 = \alpha_{20} + \alpha_{21}X_1 + \alpha_{22}X_2 + \ldots + \alpha_{2n}X_n$$

$$\ldots$$

$$X_i = \alpha_{i0} + \alpha_{i1}X_1 + \alpha_{i2}X_2 + \ldots + \alpha_{in}X_n$$

$$\ldots$$

$$X_n = \alpha_{n0} + \alpha_{n1}X_1 + \alpha_{12}X_2 + \ldots + \alpha_{nn}X_n$$

или (левосторонняя запись):

$$X_1 = \alpha_{10} + X_1\alpha_{11} + X_2\alpha_{12} + \ldots + X_n\alpha_{1n}$$

$$X_2 = \alpha_{20} + X_1\alpha_{21} + X_2\alpha_{22} + \ldots + X_n\alpha_{2n}$$

$$\ldots$$

$$X_i = \alpha_{i0} + X_1\alpha_{i1} + X_2\alpha_{i2} + \ldots + X_n\alpha_{in}$$

$$\ldots$$

$$X_n = \alpha_{n0} + X_1\alpha_{n1} + X_2\alpha_{12} + \ldots + X_n\alpha_{nn}$$

В системе уравнений с регулярными коэффициентами все коэффициенты α_{ij} являются регулярными выражениями над алфавитом $\mathbf{V}$, а переменные не входят в алфавит $\mathbf{V}$: $\forall i\ X_i \notin \mathbf{V}$. Оба варианта записи равноправны, но в общем случае могут иметь различные решения при одинаковых коэффициентах при переменных. Чтобы решить систему уравнений с регулярными коэффициентами, надо найти такие регулярные множества $\mathbf{X}_i$, при подстановке которых в систему все уравнения превращаются в тождества. Иными словами, решением системы является некоторое отображение $f(\mathbf{X})$ множества переменных уравнения $\Delta = \{\mathbf{X}_i: n \geq i > 0\}$ на множество языков над алфавитом $\mathbf{V}^*$.

Системы уравнений с регулярными коэффициентами решаются методом последовательных подстановок. Рассмотрим метод решения для правосторонней записи. Алгоритм решения работает с переменной номера шага i и состоит из следующих шагов:

Шаг 1. Положить i:=1.

Шаг 2. Если i=n, то перейти к шагу 4, иначе записать i-ое уравнение в виде: $X_i=\alpha_i X_i+\beta_i$, где $\alpha_i=\alpha_{ii}$, $\beta_i=\beta_{i0}+\beta_{ii+1}X_{i+1}+\ldots+\beta_{in}X_n$. Решить уравнение и получить $X_i=\alpha_i^*\beta_i$. Затем для всех уравнений с переменными $X_{i+1}, \ldots, X_n$ подставить в них найденное решение вместо X_i.

Шаг 3. Увеличить i на 1 (i:=i+1) и вернуться к шагу 2.

Шаг 4. После всех подстановок уравнение для X_n будет иметь вид $X_n=\alpha_n X_n+\beta$, где $\alpha_n=\alpha_{nn}$. Причем β будет регулярным выражением над алфавитом $\mathbf{V}^*$ (не содержит в своем составе переменных системы уравнений X_i). Тогда можно найти окончательное решение для X_n: $X_n=\alpha_n^*\beta$. Перейти к шагу 5.

Шаг 5. Уменьшить i на 1 (i:=i–1). Если i=0, то алгоритм завершен, иначе перейти к шагу 6.

Шаг 6. Берем найденное решение для $X_i=\alpha_i X_i+\beta_i$, где $\alpha_i=\alpha_{ii}$, $\beta_i=\beta_{i0}+\beta_{ii+1}X_{i+1}+\ldots+\beta_{in}X_n$ и подставляем в него окончательные решения для переменных $X_{i+1}, \ldots, X_n$. Получаем окончательное решение для X_i. Перейти к шагу 5.

Для левосторонней записи системы уравнений алгоритм решения будет аналогичным, с разницей только в порядке записи коэффициентов.

Система уравнений с регулярными коэффициентами всегда имеет решение, но это решение не всегда единственное. Для рассмотренного алгоритма решения системы уравнений с регулярными коэффициентами доказано, что он всегда находит решение f(**X**), которое является наименьшей неподвижной точкой системы уравнений. То есть если существует любое другое решение g(**X**), то всегда f(**X**)$\subseteq$g(**X**) [4 т.1, 15].

В качестве примера рассмотрим систему уравнений с регулярными коэффициентами над алфавитом `V={"–","+",.,0,1}` (для ясности записи символы – и + взяты в кавычки, чтобы не путать их со знаками операций):

X_1 = ("–" + "+" + λ)

$X_2 = X_1.(0 + 1) + X_3. + X_2(0 + 1)$

$X_3 = X_1(0 + 1) + X_3(0 + 1)$

$X_4 = X_2 + X_3$

Решим эту систему уравнений.

Шаг 1.

```
i := 1
```

Шаг 2.

Имеем `i=1 < 4`.

Берем уравнение для i=1. Имеем X_1 = ("–" + "+" + λ). Это уже и есть решение для X_1.

Подставляем его в другие уравнения. Получаем:

X_2 = ("–" + "+" + λ).(0 + 1) + X_3. + X_2(0 + 1)

$X_3 = ("–" + "+" + \lambda)(0 + 1) + X_3(0 + 1)$

$X_4 = X_2 + X_3$

Шаг 3.

i:=i+1 = 2

Возвращаемся к шагу 2.

Шаг 2.

Имеем i=2 < 4.

Берем уравнение для i=2. Имеем $X_2 = ("–" + "+" + \lambda).(0 + 1) + X_3. + X_2(0 + 1)$. Преобразуем уравнение к виду: $X_2 = X_2(0 + 1) + (("–" + "+" + \lambda).(0 + 1) + X_3.)$. Тогда $\alpha_2 = (0 + 1)$, $\beta_2 = ("–" + "+" + \lambda).(0 + 1) + X_3.$, а решением для X_2 будет: $X_2 = \beta_2\alpha_2^* = (("–" + "+" + \lambda).(0 + 1) + X_3.)(0 + 1)^* = ("–" + "+" + \lambda).(0 + 1)(0 + 1)^* + X_3.(0 + 1)^*$.

Подставим его в другие уравнения. Получаем:

$X_3 = ("–" + "+" + \lambda)(0 + 1) + X_3(0 + 1)$

$X_4 = ("–" + "+" + \lambda).(0 + 1)(0 + 1)^* + X_3.(0 + 1)^* + X_3$

Шаг 3.

i:=i+1 = 3

Возвращаемся к шагу 2.

Шаг 2.

Имеем i=3 < 4.

Берем уравнение для i=3. Имеем $X_3 = ("–" + "+" + \lambda)(0 + 1) + X_3(0 + 1)$. Преобразуем уравнение к виду $X_3 = X_3(0 + 1) + ("–" + "+" + \lambda)(0 + 1)$.

Тогда $\alpha_3 = (0 + 1)$, $\beta_3 = ("–" + "+" + \lambda)(0 + 1)$. Решением для X_3 будет:

$X_3 = \beta_3\alpha_3^* = ("–" + "+" + \lambda)(0 + 1)(0 + 1)^*$.

Подставим его в другие уравнения. Получаем:

$X_4 = ("–" + "+" + \lambda).(0 + 1)(0 + 1)^* + ("–" + "+" + \lambda)(0 + 1)(0 + 1)^*.(0 + 1)^*$
$+ ("–" + "+" + \lambda)(0 + 1)(0 + 1)^*$.

Шаг 3.

i:=i+1 = 4

Возвращаемся к шагу 2.

Шаг 2.

Имеем i=4 = 4. Переходим к шагу 4.

Шаг 4.

Уравнение для X_4 теперь имеет вид:

X_4 = ("–" + "+" + λ).(0 + 1)(0 + 1)* + ("–" + "+" + λ)(0 + 1)(0 + 1)*.(0 + 1)*
+ ("–" + "+" + λ)(0 + 1)(0 + 1)*.

Оно не нуждается в преобразованиях и содержит окончательное решение для X_4. Переходим к шагу 5.

Шаг 5.

i:=i–1 = 3 > 0

Переходим к шагу 6.

Шаг 6.

Уравнение для X_3 имеет вид X_3 = ("–" + "+" + λ)(0 + 1)(0 + 1)*. Оно уже содержит окончательное решение для X_3. Переходим к шагу 5.

Шаг 5.

i:=i–1 = 2 > 0

Переходим к шагу 6.

Шаг 6.

Уравнение для X_2 имеет вид X_2 = ("–" + "+" + λ).(0 + 1)(0 + 1)* + X_3.(0 + 1)*. Подставим в него окончательное решение для X_3. Получим окончательное решение для X_2: X_2 = ("–" + "+" + λ).(0 + 1)(0 + 1)* + ("–" + "+" + λ)(0 + 1)(0 + 1)*.(0 + 1)*. Переходим к шагу 5.

Шаг 5.

i:=i–1 = 1 > 0

Переходим к шагу 6.

Шаг 6.

Уравнение для X_1 имеет вид X_1 = ("–" + "+" + λ). Оно уже содержит окончательное решение для X_1. Переходим к шагу 5.

Шаг 5.

i:=i–1 = 0 = 0

Алгоритм завершен.

В итоге получили решение:

$X_1 = ("–" + "+" + \lambda)$

$X_2 = ("–" + "+" + \lambda).(0 + 1)(0 + 1)^* + ("–" + "+" + \lambda)(0 + 1)(0 + 1)^*.(0 + 1)^*$

$X_3 = ("–" + "+" + \lambda)(0 + 1)(0 + 1)^*$

$X_4 = ("–" + "+" + \lambda).(0 + 1)(0 + 1)^* + ("–" + "+" + \lambda)(0 + 1)(0 + 1)^*.(0 + 1)^*$
$+ ("–" + "+" + \lambda)(0 + 1)(0 + 1)^*$

Выполнив несложные преобразования, это же решение можно представить в более простом виде:

$X_1 = ("–" + "+" + \lambda)$

$X_2 = ("–" + "+" + \lambda)(.(0 + 1) + (0 + 1)^+.)(0 + 1)^*$

$X_3 = ("–" + "+" + \lambda)\ (0 + 1)^+$

$X_4 = ("–" + "+" + \lambda)(.(0 + 1) + (0 + 1)^+. + (0 + 1))(0 + 1)^*$

Можно заметить, что регулярное выражение для X_4 описывает язык двоичных чисел с плавающей точкой.

Свойства регулярных языков

Основные свойства регулярных языков

Множество называется замкнутым относительно некоторой операции, если в результате выполнения этой операции над любыми элементами, принадлежащими данному множеству, получается новый элемент, принадлежащий тому же множеству.

Например, множество целых чисел замкнуто относительно операций сложения, умножения и вычитания, но оно не замкнуто относительно операции деления — при делении двух целых чисел не всегда получается целое число.

Регулярные множества (и однозначно связанные с ними регулярные языки) замкнуты относительно многих операций, которые применимы к цепочкам символов.

Например, регулярные языки замкнуты относительно следующих операций:

- пересечения;
- объединения;
- дополнения;
- итерации;
- конкатенации;

- гомоморфизма (изменения имен символов и подстановки цепочек вместо символов).

Поскольку регулярные множества замкнуты относительно операций пересечения, объединения и дополнения, то они представляют булеву алгебру множеств. Существуют и другие операции, относительно которых замкнуты регулярные множества. Вообще говоря, таких операций достаточно много.

Проблемы, разрешимые для регулярных языков

Регулярные языки представляют собой очень удобный тип языков. Для них разрешимы многие проблемы, неразрешимые для других типов языков.

Например, доказано, что разрешимыми являются следующие проблемы:

- *Проблема эквивалентности*. Даны два регулярных языка $L_1(\mathbf{V})$ и $L_2(\mathbf{V})$. Необходимо проверить, являются ли эти два языка эквивалентными.
- *Проблема принадлежности цепочки языку*. Дан регулярный язык $L(\mathbf{V})$ и цепочка символов $\alpha \in \mathbf{V}^*$. Необходимо проверить, принадлежит ли цепочка данному языку.
- *Проблема пустоты языка*. Дан регулярный язык $L(\mathbf{V})$. Необходимо проверить, является ли этот язык пустым, то есть найти хотя бы одну цепочку $\alpha \neq \lambda$, такую что: $\alpha \in L(\mathbf{V})$.

Эти проблемы разрешимы вне зависимости от того, каким из трех способов задан регулярный язык. Следовательно, эти проблемы разрешимы для всех способов представления регулярных языков: регулярных множеств, регулярных грамматик и конечных автоматов. На самом деле, достаточно доказать разрешимость любой из этих проблем хотя бы для одного из способов представления языка, тогда для остальных способов можно воспользоваться алгоритмами преобразования, рассмотренными выше.[5]

Для регулярных грамматик также разрешима проблема однозначности — доказано, что для любой регулярной грамматики можно построить эквивалентную ей однозначную регулярную грамматику.

Лемма о разрастании для регулярных языков

Иногда бывает необходимо доказать, является или нет некоторый язык регулярным. Существует способ проверки, является или нет заданный язык регулярным. Этот метод основан на проверке так называемой леммы о разрастании языка. Доказано, что если для некоторого заданного языка выполняется лемма о разрастании регулярного языка, то этот язык является регулярным; если же лемма не выполняется, то и язык регулярным не является [4 т.1].

Лемма о разрастании для регулярных языков формулируется следующим образом: если дан регулярный язык и достаточно длинная цепочка символов, принадлежащая этому языку, то в этой цепочке можно найти непустую подцепочку, которую можно

---

[5] Возможны и другие способы представления регулярных множеств, а для них разрешимость указанных проблем будет уже не очевидна.

повторить сколь угодно много раз, и все полученные таким способом новые цепочки будут принадлежать тому же регулярному языку.[6]

Формально эту лемму можно записать так: если дан язык **L**, то ∃ константа $p > 0$, такая, что если $\alpha \in \mathbf{L}$ и $|\alpha| \geq p$, то цепочку α можно записать в виде $\alpha = \delta\beta\varepsilon$, где $0 < |\beta| \leq p$ и тогда $\alpha' = \delta\beta^i\varepsilon$, $\alpha' \in \mathbf{L}$ $\forall I \geq 0$.

Используя лемму о разрастании регулярных языков, докажем, что язык $\mathbf{L} = \{a^n b^n \mid n > 0\}$ не является регулярным.

Предположим, что этот язык регулярный, тогда для него должна выполняться лемма о разрастании. Возьмем некоторую цепочку этого языка $\alpha = a^n b^n$ и запишем ее в виде $\alpha = \delta\beta\varepsilon$. Если $\beta \in a^+$ или $\beta \in b^+$, то тогда для $i = 0$ цепочка $\delta\beta^0\varepsilon = \delta\varepsilon$ не принадлежит языку **L**, что противоречит условиям леммы; если же $\beta \in a^+b^+$, тогда для $i = 2$ цепочка $\delta\beta^2\varepsilon = \delta\beta\beta\varepsilon$ не принадлежит языку **L**. Таким образом, язык **L** не может быть регулярным языком.

Построение лексических анализаторов

Три способа задания регулярных языков

Регулярные (праволинейные и леволинейные) грамматики, конечные автоматы (КА) и регулярные множества (равно как и обозначающие их регулярные выражения) — это три различных способа, с помощью которых можно задавать регулярные языки. Регулярные языки, в принципе, можно определять и другими способами, но именно три указанных способа представляют наибольший интерес.

Для этих трех способов определения регулярных языков можно записать следующие строгие утверждения:

Утверждение 1. Язык является регулярным множеством тогда и только тогда, когда он задан леволинейной (праволинейной) грамматикой.

Утверждение 2. Язык может быть задан леволинейной (праволинейной) грамматикой тогда и только тогда, когда он является регулярным множеством.

Утверждение 3. Язык является регулярным множеством тогда и только тогда, когда он задан с помощью конечного автомата.

Утверждение 4. Язык распознается с помощью конечного автомата тогда и только тогда, когда он является регулярным множеством.

Все три способа определения регулярных языков равноправны. Существуют алгоритмы, которые позволяют для регулярного языка, заданного одним из указанных способов, построить другой способ, определяющий тот же самый язык. Это не всегда справедливо для других способов, которыми можно определить регулярные языки [4 т.1, 15, 21, 29].

---

[6] Если найденную подцепочку повторять несколько раз, то исходная цепочка как бы «разрастается» — отсюда и название «лемма о разрастании языков».

Из всех возможных преобразований практический интерес представляют два преобразования:

- построение регулярного выражения, задающего язык, на основе регулярной грамматики;
- построение КА на основе регулярной грамматики.

Ниже рассмотрены алгоритмы, позволяющие выполнять эти преобразования.

Построение регулярного выражения для языка, заданного леволинейной грамматикой

Постановка задачи

Для любого регулярного языка, заданного регулярной грамматикой, можно получить регулярное выражение, определяющее тот же язык.

Задача формулируется следующим образом: имеется леволинейная грамматика G(**VT**,**VN**,**P**,S), необходимо найти регулярное выражение над алфавитом **VT**, определяющее язык L(**G**), заданный этой грамматикой.

Задача решается в два этапа:

1. На основе грамматики **G** задающей язык L(**G**) строим систему уравнений с регулярными коэффициентами.
2. Решаем полученную систему уравнений. Решение, полученное для целевого символа грамматики S, будет представлять собой искомое регулярное выражение, определяющее язык L(**G**).

Поскольку алгоритм решения системы уравнений с регулярными коэффициентами известен, то далее будет рассмотрен только алгоритм, позволяющий на основе грамматики G(**VT**,**VN**,**P**,S) построить систему уравнений с регулярными коэффициентами.

Построение системы уравнений с регулярными коэффициентами на основе регулярной грамматики

В данном случае преобразование не столь элементарно. Выполняется оно следующим образом:

1. Обозначим символы алфавита нетерминальных символов **VN** следующим образом: VN = $\{X_1, X_2, \ldots, X_n\}$. Тогда все правила грамматики будут иметь вид: $X_i \to X_j\gamma$, или $X_i \to \gamma$ $X_i, X_j \in \mathbf{VN}$, $\gamma \in \mathbf{VT}^*$; целевому символу грамматики S будет соответствовать некоторое обозначение X_k.
2. Построим систему уравнений с регулярными коэффициентами на основе переменных $X_1, X_2, \ldots, X_n$:

$$X_1 = \alpha_{01} + X_1\alpha_{11} + X_2\alpha_{21} + \ldots + X_n\alpha_{n1}$$

$$X_2 = \alpha_{02} + X_1\alpha_{12} + X_2\alpha_{22} + \ldots + X_n\alpha_{n2}$$

$$\ldots$$

$$X_n = \alpha_{0n} + X_1\alpha_{1n} + X_2\alpha_{2n} + \ldots + X_n\alpha_{nn}$$

коэффициенты $\alpha_{01}, \alpha_{02}, \ldots, \alpha_{0n}$ выбираются следующим образом:

$$\alpha_{0i} = (\gamma_1 + \gamma_2 + \ldots + \gamma_m),$$

если во множестве правил **P** грамматики **G** существуют правила $X_i \rightarrow \gamma_1|\gamma_2|\ldots|\gamma_m$;

$$\alpha_{0i} = \varnothing,$$

если правил такого вида не существует;

коэффициенты $\alpha_{j1}, \alpha_{j2}, \ldots, \alpha_{jn}$ для некоторого j выбираются следующим образом:

$$\alpha_{ji} = (\gamma_1 + \gamma_2 + \ldots + \gamma_m),$$

если во множестве правил **P** грамматики **G** существуют правила $X_i \rightarrow X_j\gamma_1|X_j\gamma_2|\ldots|X_j\gamma_m$;

$$\alpha_{ji} = \varnothing,$$

если правил такого вида не существует.

3. Находим решение построенной системы уравнений.

Доказано, что решение для X_k, которое обозначает целевой символ S грамматики **G**, будет представлять собой искомое регулярное выражение, обозначающее язык, заданный грамматикой **G**.

Остальные решения системы будут представлять собой регулярные выражения, обозначающие понятия грамматики, соответствующие ее нетерминальным символам.

СОВЕТ

> В принципе для поиска регулярного выражения, обозначающего язык, заданный грамматикой, не нужно искать все решения — достаточно найти решение для X_k.

Пример построения регулярного выражения для языка, заданного леволинейной грамматикой

Например, рассмотрим леволинейную регулярную грамматику, определяющую язык двоичных чисел с плавающей точкой `G({.,-,+,0,1}, {<знак>,<дробное>,<целое>,<число>}, P, <число>)`:

P:

`<знак>` → `-|+|λ`

`<дробное>` →
`<знак>.0|<знак>.1|<целое>.|<дробное>0|<дробное>1`

`<целое>` → `<знак>0|<знак>1|<целое>0|<целое>1`

`<число>` → `<дробное>|<целое>`

Обозначим символы множества **VN**={<знак>,<дробное>,<целое>,<число>} соответствующими переменными X_i, получим: **VN**={X_1,X_2,X_3,X_4}.

Построим систему уравнений на основе правил грамматики **G**:

X_1 = ("–" + "+" + λ)

X_2 = X_1.(0+1) + X_3. + X_2(0+1)

X_3 = X_1(0+1) + X_3(0+1)

X_4 = X_2 + X_3

Эта система уравнений уже была решена выше. В данном случае нас интересует только решение для X_4, которое соответствует целевому символу грамматики **G** <число>.

Решение для X_4 может быть записано в виде:

X_4 = ("–" + "+" + λ)(.(0+1) + (0+1)(0+1)*. + (0+1))(0+1)*

то есть:

<число> = ("–" + "+" + λ)(.(0+1) + (0+1)(0+1)*. + (0+1))(0+1)*

Это и есть регулярное выражение, определяющее язык двоичных чисел с плавающей точкой, заданный грамматикой **G**.

Построение конечного автомата на основе леволинейной грамматики

Постановка задачи

На основе имеющейся регулярной грамматики можно построить эквивалентный ей КА, и, наоборот, для заданного КА можно построить эквивалентную ему регулярную грамматику.

Это очень важное утверждение, поскольку регулярные грамматики используются для определения лексических конструкций языков программирования. Создав КА на основе известной грамматики, мы получаем распознаватель для лексических конструкций данного языка. Таким образом, удается решить задачу разбора для лексических конструкций языка, заданных произвольной регулярной грамматикой. Обратное утверждение также полезно, поскольку позволяет узнать грамматику, цепочки языка которой допускает заданный автомат.

Все языки программирования определяют нотацию записи «слева направо». В той же нотации работают и компиляторы. Поэтому далее рассмотрены алгоритмы для леволинейных грамматик. Доказано, что аналогичные построения возможно выполнить и для праволинейных грамматик.

Задача формулируется следующим образом: имеется леволинейная грамматика G(**VT**,**VN**,**P**,S), задающая язык L(**G**), необходимо построить эквивалентный ей конечный автомат M(**Q**,**V**,δ,q_0,**F**), задающий тот же язык: L(**G**) = L(**M**).

Задача решается в два этапа:

1. Исходную леволинейную грамматику **G** необходимо привести к автоматному виду **G'**.
2. На основе полученной автоматной леволинейной грамматики G'(**VT**,**VN'**,**P'**,S) строится искомый автомат M(**Q**,**V**,δ,q_0,**F**).

Алгоритм преобразования к автоматному виду был рассмотрен выше, поэтому здесь рассмотрим только алгоритм построения КА на основе автоматной леволинейной грамматики.

Алгоритм построения конечного автомата на основе автоматной леволинейной грамматики

Построение КА M(**Q**,**V**,δ,q_0,**F**) на основе автоматной леволинейной грамматики G(**VT**,**VN**,**P**,S) выполняется по следующему алгоритму:

Шаг 1. Строим множество состояний автомата **Q**. Состояния автомата строятся таким образом, чтобы каждому нетерминальному символу из множества **VN** грамматики **G** соответствовало одно состояние из множества **Q** автомата **M**. Кроме того, во множество состояний автомата добавляется еще одно дополнительное состояние, которое будем обозначать H. Сохраняя обозначения нетерминальных символов грамматики **G**, можно записать: **Q** = **VN**∪{H}.

Шаг 2. Входным алфавитом автомата **M** является множество терминальных символов грамматики **G**: **V** = **VT**.

Шаг 3. Просматриваем все множество правил исходной грамматики.

Если встречается правило вида A→t∈**P**, где A∈**VN**, t∈**VT**, то в функцию переходов δ(H,t) автомата **M** добавляем состояние A: A∈δ(H,t).

Если встречается правило вида A→Bt∈**P**, где A,B∈**VN**, t∈**VT**, то в функцию переходов δ(B,t) автомата **M** добавляем состояние A: A∈δ(B,t).

Шаг 4. Начальным состоянием автомата **M** является состояние H: q_0 = H.

Шаг 5. Множество конечных состояний автомата **M** состоит из одного состояния. Этим состоянием является состояние, соответствующее целевому символу грамматики **G**: **F** = {S}.

На этом построение автомата заканчивается.

Пример построения конечного автомата на основе заданной леволинейной грамматики

Рассмотрим грамматику `G({"a","(","*",")","{","}"},{S,C,K},P,S)` (символы `a`, `(`, `*`, `)`, `{`, `}` из множества терминальных символов грамматики взяты в кавычки, чтобы выделить их среди фигурных скобок, обозначающих само множество):

P:

`S → C*) | K}`

`C → (* | Ca | C{ | C} | C( | C* | C)`

K → { | Ka | K(| K* | K) | K{

Это леволинейная регулярная грамматика. Преобразование ее к автоматному виду уже было выполнено ранее (в разделе «Регулярные и автоматные грамматики» в этой главе).

Получим леволинейную автоматную грамматику следующего вида: G'({"a","(","*",")","{","}"}, {S,S_1,C,C_1,K},**P'**,S):

P':

S → S_1) | K}

S_1 → C*

C → C_1* | Ca | C{ | C} | C(| C* | C)

C_1 → (

K → { | Ka | K(| K* | K) | K{

Для удобства переобозначим нетерминальные символы C_1 и S_1 символами D и E. Получим грамматику G'({"a","(","*",")","{","}"},{S,E,C,D,K},**P'**,S):

P':

S → E) | K}

E → C*

C → D* | Ca | C{ | C} | C(| C* | C)

D → (

K → { | Ka | K(| K* | K) | K{

Построим конечный автомат M(**Q**,**V**,δ,q_0,**F**), эквивалентный указанной грамматике.

Шаг 1.

Строим множество состояний автомата. Получаем: Q=VN∪{H}={S,E,C,D,K,H}.

Шаг 2.

В качестве алфавита входных символов автомата берем множество терминальных символов грамматики. Получаем: V={"a","(","*",")","{","}"}.

Шаг 3.

Рассматриваем множество правил грамматики.

Для правил S→E)|K} имеем δ(E,")")={S}; δ(K,"}")={S}.

Для правила E→C* имеем δ(C,"*")={E}.

Для правил C→D*|Ca|C{|C}|C(|C*|C) имеем δ(D,"*")={C}; δ(C,"a")={C}; δ(C,"{")={C}; δ(C,"}")={C}; δ(C,"(")={C}; δ(C,"*")={E,C}; δ(C,")")={C}.

Для правила D→(имеем δ(H,"(")={D}.

Для правил K→{|Ka|K(|K*|K)|K{ имеем δ(H,"{")={K}; δ(K,"a")={K}; δ(K,"(")={K}; δ(K,"*")={K}; δ(K,")")={K}; δ(K,"{")={K}.

Шаг 4.

Начальным состоянием автомата является состояние q_0 = H.

Шаг 5.

Множеством конечных состояний автомата является множество F = {S}.

Выполнение алгоритма закончено.

В итоге получаем автомат M({S,E,C,D,K,H}, {"a","(","*",")","{","}"},δ,H,{S}) с функцией переходов:

δ(H,"{") = {K}

δ(H,"(") = {D}

δ(K,"a") = {K}

δ(K,"(") = {K}

δ(K,"*") = {K}

δ(K,")") = {K}

δ(K,"{") = {K}

δ(K,"}") = {S}

δ(D,"*") = {C}

δ(C,"a") = {C}

δ(C,"{") = {C}

δ(C,"}") = {C}

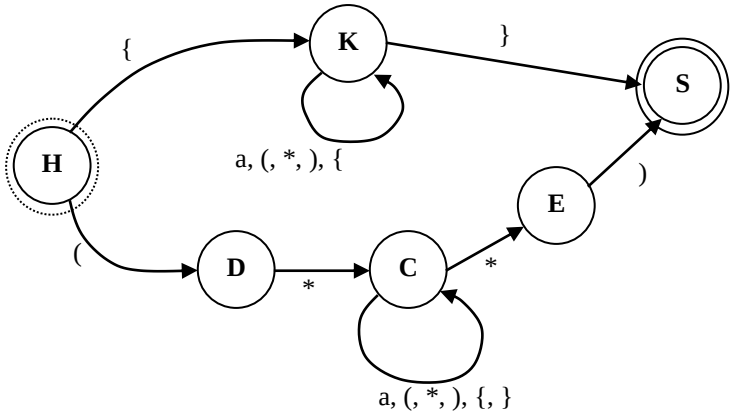
δ(C,"(") = {C}

δ(C,"*") = {E,C}

δ(C,")") = {C}

δ(E,")") = {S}

Граф переходов этого автомата изображен на 3.9.



3.9. Недетерминированный КА для языка комментариев в Borland Pascal

Это недетерминированный конечный автомат, поскольку существует состояние, в котором множество, получаемое с помощью функции переходов по одному и тому же символу, имеет более одного следующего состояния. Это состояние `C` и функция `δ(C,"*")={E,C}`.

Моделировать поведение недетерминированного КА — непростая задача, поэтому можно построить эквивалентный ему детерминированный КА. Полученный таким путем КА можно затем минимизировать.

В результате всех преобразований получаем детерминированный конечный автомат `M'({S,E,C,D,K,H},{"a","(","*",")","{","}"},δ',H,{S})` с функцией переходов:

```
δ'(H,"{") = {K}
δ'(H,"(") = {D}
δ'(K,"a") = {K}
δ'(K,"(") = {K}
δ'(K,"*") = {K}
δ'(K,")") = {K}
δ'(K,"{") = {K}
δ'(K,"}") = {S}
δ'(D,"*") = {C}
δ'(C,"a") = {C}
δ'(C,"{") = {C}
δ'(C,"}") = {C}
δ'(C,"(") = {C}
δ'(C,")") = {C}
δ'(C,"*") = {E}
```

δ'(E,"a") = {C}

δ'(E,"{") = {C}

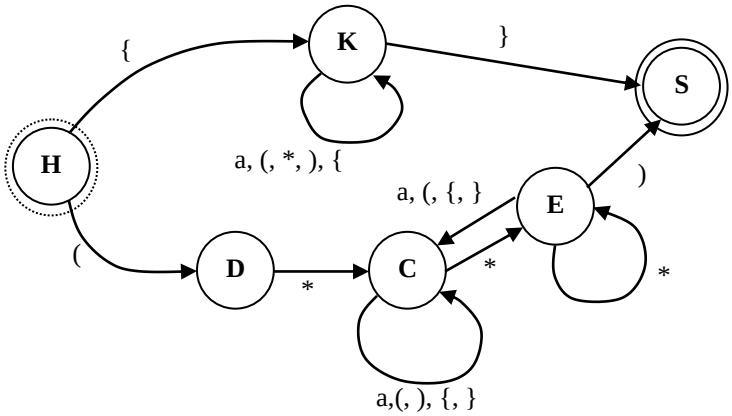
δ'(E,"}") = {C}

δ'(E,"(") = {C}

δ'(E,"*") = {E}

δ'(E,")") = {S}

Граф переходов этого автомата изображен на 3.10.



3.10. Детерминированный КА для языка комментариев в Borland Pascal

На основании этого автомата можно легко построить распознаватель. В данном случае мы можем получить распознаватель для двух типов комментариев языка программирования Borland Pascal, если учесть, что `a` может означать любой алфавитно-цифровой символ, кроме символов `(`, `*`, `)`, `{`, `}`.

Построение леволинейной грамматики на основе конечного автомата

Задача формулируется так: имеется конечный автомат M(**Q**,**V**,δ,q_0,**F**), необходимо построить эквивалентную ему леволинейную грамматику G(**VT**,**VN**,**P**,S).

Построение выполняется по следующему алгоритму:

Шаг 1. Множество терминальных символов грамматики **G** строится из алфавита входных символов автомата **M**: **VT** = **V**.

Шаг 2. Множество нетерминальных символов грамматики **G** строится на основании множества состояний автомата **M** таким образом, чтобы каждому состоянию автомата, за исключением начального состояния, соответствовал один нетерминальный символ грамматики: **VN** = **Q**\{q_0}.

Шаг 3. Просматриваем функцию переходов автомата **M** для всех возможных состояний из множества **Q** для всех возможных входных символов из множества **V**.

Если имеем $\delta(A,t)=\varnothing$, то ничего не выполняем.

Если имеем $\delta(A,t)=\{B_1,B_2,\ldots,B_n\}$, $n>0$, где $A\in\mathbf{Q}$, $\forall n \geq i > 0$: $B_i\in\mathbf{Q}$, $t\in\mathbf{V}$, тогда для всех состояний B_i выполняем следующее:

- добавляем правило $B_i\rightarrow t$ во множество **P** правил грамматики **G**, если $A=q_0$;
- добавляем правило $B_i\rightarrow At$ во множество **P** правил грамматики **G**, если $A\neq q_0$.

Шаг 4. Если множество конечных состояний **F** автомата **M** содержит только одно состояние $\mathbf{F}=\{F_1\}$, то целевым символом S грамматики **G** становится символ множества **VN**, соответствующий этому состоянию: $S=F_1$; иначе, если множество конечных состояний **F** автомата **M** содержит более одного состояния $F=\{F_1,F_2,\ldots,F_n\}$, $n>1$, тогда во множество нетерминальных символов **VN** грамматики **G** добавляется новый нетерминальный символ S: $\mathbf{VN}=\mathbf{VN}\cup\{S\}$, а во множество правил **P** грамматики **G** добавляются правила: $S\rightarrow F_1|F_2|\ldots|F_n$.

На этом построение грамматики заканчивается.

Примеры построения лексических анализаторов

Теперь можно рассмотреть практическую реализацию лексических анализаторов. В принципе компилятор может иметь в своем составе не один, а несколько лексических анализаторов, каждый из которых предназначен для выборки и проверки определенного типа лексем.

Таким образом, обобщенный алгоритм работы простейшего лексического анализатора в компиляторе можно описать следующим образом:

- из входного потока выбирается очередной символ, в зависимости от которого запускается тот или иной сканер (символ может быть также проигнорирован, либо признан ошибочным);
- запущенный сканер просматривает входной поток символов программы на исходном языке, выделяя символы, входящие в требуемую лексему, до обнаружения очередного символа, который может ограничивать лексему, либо до обнаружения ошибочного символа;
- при успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, алгоритм возвращается к первому этапу и продолжает рассматривать входной поток символов с того места, на котором остановился сканер;
- при неуспешном распознавании выдается сообщение об ошибке, а дальнейшие действия зависят от реализации сканера — либо его выполнение прекращается, либо делается попытка распознать следующую лексему (идет возврат к первому этапу алгоритма).

В целом техника построения сканеров основывается на моделировании работы детерминированных и недетерминированных КА с дополнением функций распознавателя вызовами функций заполнения таблиц лексем и таблиц

идентификаторов, а также обработки ошибок. Такая техника не требует сложной математической обработки и принципиально важных преобразований входных грамматик. Для разработчиков сканера важно только решить, где кончаются функции сканера и начинаются функции синтаксического разбора. После этого процесс построения сканера легко поддается автоматизации.

Лексический анализатор целочисленных констант языка C

Рассмотрим пример анализа лексем, представляющих собой целочисленные константы в формате языка C. В соответствии с требованиями языка, такие константы могут быть десятичными, восьмеричными или шестнадцатеричными. Восьмеричной константой считается число, начинающееся с `0` и содержащее цифры от `0` до `7`; шестнадцатеричная константа должна начинаться с последовательности символов `0x` и может содержать цифры и буквы от `a` до `f`. Остальные числа считаются десятичными (правила их записи напоминать, наверное, не стоит). Константа может начинаться также с одного из знаков `+` или `−`, а в конце цифры, обозначающей значение константы, в языке C может следовать буква или две буквы, явно обозначающие ее тип (`u`, `U` — unsigned; `h`, `H` — short; `l`, `L` — long).

При построении сканера будем учитывать, что константы входят в общий текст программы на языке C. Для избежания путаницы и сокращения объема информации в примере будем считать, что все допустимые буквы являются строчными (читатели легко смогут самостоятельно расширить пример для прописных букв, которые язык C в константах не отличает от строчных).

Рассмотренные выше правила могут быть записаны в форме Бэкуса-Наура в грамматике целочисленных констант для языка C.

G({S,W,U,L,V,D,G,X,Q,Z,N},{0...9,x,a...f,u,l,h,⊥},**P**,S)

P:

S → G⊥ | Z⊥ | D⊥ | Q⊥ | U⊥ | L⊥ | V⊥ | W⊥

W → Lu | Vu | Ul | Uh

U → Gu | Zu | Hu | Qu

L → Gl | Zl | Hl | Ql

V → Gh | Zh | Hh | Qh

D → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

N1 | N2 | N3 | N4 | N5 | N6 | N7 | N8 | N9 |

D0 | D1 | D2 | D3 | D4 | D5 | D6 | D7 | D8 | D9 |

Z8 | Z9 | Q8 | Q9

G → X0 | X1 | X2 | X3 | X4 | X5 | X6 | X7 | X8 | X9 |

Xa | Xb | Xc | Xd | Xe | Xf |

G0 | G1 | G2 | G3 | G4 | G5 | G6 | G7 | G8 | G9 |

```
Ga | Gb | Gc | Gd | Ge | Gf

X → Zx

Q → Z0 | Z1 | Z2 | Z3 | Z4 | Z5 | Z6 | Z7 |

Q0 | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 | Q7

Z → 0 | N0

N → + | −
```

Эта грамматика является леволинейной автоматной регулярной грамматикой. По ней можно построить КА, который будет распознавать цепочки входного языка (см. раздел «Конечные автоматы»). Граф переходов этого автомата приведен на 3.10. Видно, что построенный автомат является детерминированным КА, поэтому в дальнейших преобразованиях он не нуждается. Начальным состоянием автомата является состояние `H`.

<03.cdr>

3.10. Граф конечного автомата для распознавания целочисленных констант языка C

Приведем автомат к полностью определенному виду — дополним его новым состоянием `E`, на которое замкнем все дуги неопределенных переходов. На графе автомата дуги, идущие в это состояние, не нагружены символами — они обозначают функцию перехода по любому символу, кроме тех символов, которыми уже помечены другие дуги, выходящие из той же вершины графа (такое соглашение принято, чтобы не загромождать обозначениями граф автомата). На 3.10 это состояние и связанные с ним дуги переходов изображены пунктирными линиями.

При построении КА знаком ⊥ были обозначены любые символы, которыми может завершаться целочисленная константа языка C. Однако в реальной программе этот символ не встречается, поскольку границы лексем в ней явно не указаны. При моделировании КА надо решить проблему определения границ лексем. Следует принять во внимание, какими реальными символами входного языка может завершаться целочисленная константа языка C.

В языке C в качестве границы константы могут выступать:

- знаки пробелов, табуляции, перевода строки;
- разделители `(`, `)`, `[`, `]`, `{`, `}`, `,`, `:`, `;`;
- знаки операций +, −, *, /, &, |, ?, ~, <, >, ^, =, %.

Теперь можно написать программу, моделирующую работу указанного автомата. Ниже приводится текст функции на языке Pascal, реализующей данный распознаватель. Результатом функции является текущее положение считывающей головки анализатора в потоке входной информации после завершения разбора лексемы при успешном распознавании, либо `0` при неуспешном распознавании (если лексема содержит ошибку).

В программе переменная iState отображает текущее состояние автомата, переменная i является счетчиком символов входной строки. В текст программы вписаны комментарии в те места функции, где возможно выполнить запись найденной лексемы в таблицу лексем и таблицу символов, а также выдать сообщение об обнаруженной ошибке.

```
function RunAuto (const sInput: string; iPos: integer):
integer;
(* sInput – входная строка исходного текста;
   iPos – текущее положение считывающей головки во входной
   строке исходного текста *)
type
 TAutoState = (AUTO_H, AUTO_S, AUTO_W, AUTO_U, AUTO_L,
AUTO_V, AUTO_D, AUTO_G, AUTO_X, AUTO_Q, AUTO_Z, AUTO_N,
AUTO_E);
const
 AutoStop = [' ', #10, #13, #7, '(', ')', '[', ']', '{',
'}', ',', ':', ';', '+', '-', '*', '/', '&', '|', '?', '~',
'<', '>', '^', '=', '%'];
var
 iState : TAutoState;
 i,iL : integer;
 sOut : string;
begin
 i := iPos;
 iL := Length(sInput);
 sOut := '';
 iState := AUTO_H;
 repeat
    case iState of
       AUTO_H:
          case sInput[i] of
             '+','-':  iState := AUTO_N;
             '0':      iState := AUTO_Z;
             '1'..'9': iState := AUTO_D;
             else      iState := AUTO_E;
          end;
```

```
AUTO_N:
   case sInput[i] of
      '0':      iState := AUTO_Z;
      '1'..'9': iState := AUTO_D;
      else      iState := AUTO_E;
   end;
AUTO_Z:
   case sInput[i] of
      'x':      iState := AUTO_X;
      '0'..'7': iState := AUTO_Q;
      '8','9':  iState := AUTO_D;
      else
      if sInput[i] in AutoStop then
         iState := AUTO_S
         else iState := AUTO_E;
   end;
AUTO_X:
   case sInput[i] of
      '0'..'9',
      'a'..'f': iState := AUTO_G;
      else      iState := AUTO_E;
   end;
AUTO_Q:
   case sInput[i] of
      '0'..'7': iState := AUTO_Q;
      '8','9':  iState := AUTO_D;
      'u':      iState := AUTO_U;
      'l':      iState := AUTO_L;
      'h':      iState := AUTO_V;
      else
      if sInput[i] in AutoStop then
         iState := AUTO_S
         else iState := AUTO_E;
```

```
    end;
AUTO_D:
    case sInput[i] of
        '0'..'9': iState := AUTO_D;
        'u':      iState := AUTO_U;
        'l':      iState := AUTO_L;
        'h':      iState := AUTO_V;
        else
        if sInput[i] in AutoStop then
            iState := AUTO_S
            else iState := AUTO_E;
    end;
AUTO_G:
    case sInput[i] of
        '0'..'9',
        'a'..'f': iState := AUTO_G;
        'u':      iState := AUTO_U;
        'l':      iState := AUTO_L;
        'h':      iState := AUTO_V;
        else
        if sInput[i] in AutoStop then
            iState := AUTO_S
            else iState := AUTO_E;
    end;
AUTO_U:
    case sInput[i] of
        'l','h':  iState := AUTO_W;
        else
        if sInput[i] in AutoStop then
            iState := AUTO_S
            else iState := AUTO_E;
    end;
AUTO_L:
```

```
            case sInput[i] of
               'u':      iState := AUTO_W;
               else
               if sInput[i] in AutoStop then
                  iState := AUTO_S
                  else iState := AUTO_E;
            end;
         AUTO_V:
            case sInput[i] of
               'u':      iState := AUTO_W;
               else
               if sInput[i] in AutoStop then
                  iState := AUTO_S
                  else iState := AUTO_E;
            end;
         AUTO_W:
            if sInput[i] in AutoStop then
               iState := AUTO_S
               else iState := AUTO_E;
      end {case};
      if not (iState in [AUTO_E,AUTO_S]) then
      begin
         sOut := sOut + sInput[i];
         i := i + 1;
      end;
until ((iState = AUTO_E) or (iState = AUTO_S)
      or (i > iL));
if (iState = AUTO_S) or (i > iL) then
begin
{ Сюда надо вставить вызов функций записи лексемы sOut }
   RunAuto := i;
end
else
```

```
 begin
 { Сюда надо вставить вызов функции формирования сообщения
об ошибке }
    RunAuto := 0;
 end;
end; { RunAuto }
```

Конечно, рассмотренная программа — это всего лишь пример для моделирования такого рода автомата. Она построена так, чтобы можно было четко отследить соответствие между этой программой и построенным на 3.10 автоматом. Конкретный распознаватель будет существенно зависеть от того, как организовано его взаимодействие с остальными частями компилятора. Это влияет на возвращаемое функцией значение, а также на то, как она должна действовать при обнаружении ошибки и при завершении распознавания лексемы.

В данном примере предусмотрено, что основная часть лексического анализатора сначала считывает в память весь текст исходной программы, а потом начинает его последовательный просмотр. В зависимости от текущего символа вызывается тот или иной сканер. По результатам работы каждого сканера разбор либо продолжается с позиции, которую вернула функция сканера, либо прерывается, если функция возвращает 0.

Лексический анализатор для поиска строк в исходном тексте программы, написанной на языке Pascal

Действия, решаемые этим лексическим анализатором, возникли из реальной практической задачи. Эта задача является удачным примером практического применения лексических анализаторов, причем в данном случае лексический анализатор не является частью компилятора, а выступает как самостоятельная программная единица. Такие примеры довольно часто встречаются в реальной жизни, поскольку, как уже было отмечено, лексический анализ находит применение не только в компиляции.

Задача: имеется исходный текст программы, написанной на языке Pascal, необходимо найти в этом тексте все строковые константы и записать их в отдельный файл.

ПРИМЕЧАНИЕ

> На практике такая задача может возникнуть, например, для перевода интерфейса и текстовых сообщений программы с одного языка на другой (скажем, с русского на английский). В этом случае выбранные из исходного текста строки надо отдать переводчику, которому весь исходный текст не интересен (да и разбираться в нем он не сможет).

Надо отметить, что в данном случае задача лексического анализатора значительно уже, чем задача лексического анализатора компилятора Pascal. Анализатор должен найти только все строковые константы, не обращая никакого внимания на все

остальные лексемы. Также нет необходимости заполнять таблицу лексем и таблицу идентификаторов — сопоставление лексем и дальнейшая обработка текста не входят в постановку задачи. И, конечно же, нет необходимости проверять правильность исходной программы — это не задача лексического анализатора.

Именно по причине наглядности и простоты решаемой задачи данный лексический анализатор выбран в качестве удачного примера.

Построение лексического анализатора начнем с построения грамматики исходного языка. Исходным языком является язык строковых констант Pascal. Для построения его грамматики следует разобраться, что представляет собой строковая константа в языке Pascal.

Строковая константа в языке Pascal может состоять из одной или более частей, соединенных между собой знаком +. Каждая часть строковой константы начинается с символа ' (одинарная кавычка) и заканчивается символом ' (одинарная кавычка). В состав строковой константы могут входить любые алфавитно-цифровые символы, но если в нее надо включить одинарную кавычку, то она должна быть повторена дважды: ''. Также в строку могут быть включены коды символов. В языке Pascal они записываются восьмеричными цифрами, следующими за знаком #. Коды символов включаются в строку либо сразу после завершающей одинарной кавычки, либо с помощью знака +. Для лексического анализатора строковая константа считается законченной, если после очередной ее части он встретил любой алфавитно-цифровой символ, кроме знаков + и #, или конец файла. Все символы, не входящие в строковые константы, анализатором должны игнорироваться.

Для простоты и удобства работы далее будем символом b обозначать все незначащие символы языка (пробелы, знаки табуляции и перевода строки), символом d — все допустимые восьмеричные цифры (от 0 до 7), а символом a — все остальные алфавитно-цифровые символы, кроме символов, обозначенных через b и d, а также символов ', + и #. Символом ⊥ будем обозначать конец файла.

Тогда грамматику языка строковых констант Pascal можно записать следующим образом: $G_1(\{a,b,d,',+,\#\},\{S_1,S_2,S_3,S_4,D_1,D_2,D_3,S\},\mathbf{P},S)$, а ее правила **P**:

$S_1 \to ' \mid S_1a \mid S_1b \mid S_1d \mid S_1+ \mid S_1\# \mid S_2' \mid S_4' \mid D_1'$

$S_2 \to S_1'$

$S_3 \to S_2b \mid S_3b$

$S_4 \to S_2+ \mid S_3+ \mid S_4b \mid D_2+ \mid D_3+$

$D_1 \to \# \mid S_2\# \mid S_4\# \mid D_2\#$

$D_2 \to D_1d \mid D_2d$

$D_3 \to D_2b \mid D_3b$

$S \to a \mid b \mid d \mid + \mid S_2a \mid S_3a \mid S_4a \mid D_2a \mid D_3a \mid S_2\perp \mid S_3\perp \mid D_2\perp \mid D_3\perp$

Эта грамматика учитывает структуру всех возможных строковых констант языка Pascal, но она недостаточна для построения лексического анализатора, поскольку не учитывает комментарии, которые могут встречаться в тексте исходной программы.

Естественно, анализатор должен игнорировать комментарии и все строковые константы, которые могут быть внутри комментариев.

Язык Pascal предусматривает два независимых варианта комментариев. Первый вариант: комментарий начинается последовательностью символов `(*` и заканчивается последовательностью символов `*)`, второй вариант: комментарий начинается с символа `{` и заканчивается символом `}`. Комментарии могут содержать любые алфавитно-цифровые символы.

Тогда грамматику комментариев языка Pascal можно записать следующим образом: $G_2(\{a,b,d,',+,\#,(,*,),\{,\}\},\{C_1,C_2,C_3,C_4,S\},\mathbf{P},S)$, а ее правила **P**:

$C_1 \rightarrow (\mid C_1($

$C_2 \rightarrow C_1* \mid C_2a \mid C_2b \mid C_2d \mid C_2' \mid C_2+ \mid C_2\# \mid C_2(\mid C_2) \mid C_2\{ \mid C_2\} \mid C_3a \mid C_3b \mid C_3d \mid C_3' \mid C_3+ \mid C_3\# \mid C_3(\mid C_3\{ \mid C_3\}$

$C_3 \rightarrow C_2* \mid C_3*$

$C_4 \rightarrow \{ \mid C_1\{ \mid C_4a \mid C_4b \mid C_4d \mid C_4' \mid C_4+ \mid C_4\# \mid C_4(\mid C_4* \mid C_4) \mid C_4\{$

$S \rightarrow a \mid b \mid d \mid + \mid * \mid) \mid C_1a \mid C_1b \mid C_1d \mid C_1+ \mid C_1) \mid C_3) \mid C_4\}$

Здесь `a` обозначает уже все алфавитно-цифровые символы, кроме символов, обозначенных ранее через `b` и `d`, а также символов `'`, `+`, `#`, `(`, `*`, `)`, `{` и `}`.

ПРИМЕЧАНИЕ

> Эта грамматика построена таким образом, чтобы игнорировать исходный текст программы, не входящий в комментарии.

На основе двух построенных грамматик $\mathbf{G}_1$ и $\mathbf{G}_2$ можно реализовать лексический анализатор, решающий задачу в два этапа (он как бы будет состоять из двух лексических анализаторов). На первом этапе из исходной программы исключаются все комментарии анализатором, на основе грамматики $\mathbf{G}_2$, и полученный в результате текст записывается в промежуточное хранилище данных (файл). На втором этапе исходный текст из промежуточного хранилища обрабатывается анализатором на основе грамматики $\mathbf{G}_1$ и в нем ищутся все строковые константы. Получим двухпроходный лексический анализатор.

Однако анализатор можно упростить, если построить грамматику языка, который будет включать в себя и строковые константы, и комментарии языка Pascal. Такую грамматику можно построить на основе $\mathbf{G}_1$ и $\mathbf{G}_2$. Надо только учесть, что строки и комментарии могут следовать в тексте исходной программы друг за другом в произвольном порядке, а также то, что символы, ограничивающие комментарии — `(`, `*`, `)`, `{` и `}` — могут входить в состав строки.

Грамматику строковых констант и комментариев языка Pascal можно записать следующим образом:

$G(\{a,b,d,',+,\#,(,*,),\{,\}\}, \{C_1,C_2,C_3,C_4,S_1,S_2,S_3,S_4,D_1,D_2,D_3,K_1,K_2,K_3,K_4,K_5,S\}, \mathbf{P}, S)$, а ее правила **P**:

$C_1 \to (\mid C_1($

$C_2 \to C_1* \mid C_2a \mid C_2b \mid C_2d \mid C_2' \mid C_2+ \mid C_2\# \mid C_2(\mid C_2) \mid C_2\{ \mid C_2\} \mid C_3a \mid C_3b \mid C_3d \mid C_3' \mid C_3+ \mid C_3\# \mid C_3(\mid C_3\{ \mid C_3\}$

$C_3 \to C_2* \mid C_3*$

$C_4 \to \{ \mid C_1\{ \mid C_4a \mid C_4b \mid C_4d \mid C_4' \mid C_4+ \mid C_4\# \mid C_4(\mid C_4* \mid C_4) \mid C_4\{$

$S_1 \to ' \mid K_1' \mid K_5' \mid S_1a \mid S_1b \mid S_1d \mid S_1+ \mid S_1\# \mid S_1(\mid S_1* \mid S_1) \mid S_1\{ \mid S_1\} \mid S_2' \mid S_4' \mid D_2' \mid C_1'$

$S_2 \to S_1'$

$S_3 \to S_2b \mid S_3b$

$S_4 \to S_2+ \mid S_3+ \mid S_4b \mid D_2+ \mid D_3+ \mid K_5+$

$D_1 \to \# \mid S_2\# \mid S_4\# \mid D_2\# \mid C_1\# \mid K_1\# \mid K_5\#$

$D_2 \to D_1d \mid D_2d$

$D_3 \to D_2b \mid D_3b$

$K_1 \to S_2(\mid S_3(\mid S_4(\mid D_2(\mid D_3(\mid K_5(\mid K_1(\mid K_1b$

$K_2 \to K_1* \mid K_2a \mid K_2b \mid K_2d \mid K_2' \mid K_2+ \mid K_2\# \mid K_2(\mid K_2) \mid K_2\{ \mid K_2\} \mid K_3a \mid K_3b \mid K_3d \mid K_3' \mid K_3+ \mid K_3\# \mid K_3(\mid K_3\{ \mid K_3\}$

$K_3 \to K_2* \mid K_3*$

$K_4 \to S_2\{ \mid S_3\{ \mid S_4\{ \mid D_2\{ \mid D_3\{ \mid K_1\{ \mid K_5\{ \mid K_4a \mid K_4b \mid K_4d \mid K_4' \mid K_4+ \mid K_4\# \mid K_4(\mid K_4* \mid K_4) \mid K_4\{$

$K_5 \to K_3) \mid K_4\} \mid K_5b$

$S \to a \mid b \mid d \mid + \mid * \mid) \mid C_1a \mid C_1b \mid C_1d \mid C_1+ \mid C_1) \mid C_3) \mid C_4\} \mid K_1a \mid K_1d \mid K_1+ \mid K_1) \mid K_5a \mid K_5) \mid S_2a \mid S_2) \mid S_3a \mid S_3) \mid S_4a \mid D_2a \mid D_2) \mid D_3a \mid D_3) \mid S_2\perp \mid S_3\perp \mid D_2\perp \mid D_3\perp \mid K_5\perp$

На основе этой грамматики можно строить однопроходный лексический анализатор, выполняющий поиск строковых констант в исходной программе на языке Pascal. Данный анализатор игнорирует комментарии, а также весь остальной текст исходной программы, не содержащий строковых констант.

Эта грамматика является автоматной грамматикой, поэтому на ее основе элементарно просто построить КА. Он будет иметь 18 состояний. Можно заметить, что построенный КА будет детерминированным, поэтому его сразу можно реализовать в программном коде лексического анализатора.

Лексический анализатор, моделирующий работу данного КА, начинает свое функционирование с начала просмотра исходного текста программы. При этом КА находится в начальном состоянии. Если символ на входе не относится ни к комментарию, ни к строковой константе, то анализатор просто пропускает его — при этом КА сразу переходит в конечное состояние, а потом анализатор должен вернуть его в начальное состояние. Если символ относится к строковой константе, то КА начинает переходить по всем состояниям, относящимся к строковой константе (все состояния S_i и D_i). Лексический анализатор должен при этом запоминать все символы константы (строку). Если встречается комментарий, не следующий после

строковой константы, то лексический анализатор должен пропустить все символы до конца комментария (КА при этом находится в состояниях C_i). Если после строковой константы встречается комментарий, то лексический анализатор должен пропустить его (КА при этом находится в состояниях K_i), а затем ожидать либо продолжения строковой константы, либо ее конца. При переходе КА в конечное состояние `S` анализатор записывает строку в результирующий файл, а КА возвращает в начальное состояние.

Таким образом, граф переходов КА лексического анализатора поиска строковых констант языка Pascal должен быть нагружен функциями:

- создание пустой строки для записи очередной строковой константы;
- добавление символа или подстроки в конец строковой константы;
- запись строковой константы в результирующий файл и очистка строки для следующей строковой константы.

Нагрузив граф переходов КА соответствующими функциями, получим конечный преобразователь, преобразующий исходную программу на языке Pascal в файл строковых констант этой программы.

Можно заметить, что построенный КА не является полностью определенным. Это вызвано тем, что существуют ошибки, которые могут быть обнаружены с помощью данного лексического анализатора (хотя это и не является его основной задачей). Примером такой обнаруживаемой ошибки является незакрытый комментарий. Ошибки можно обозначить отдельным «ошибочным» нетерминальным символом грамматики `E` и соответствующим состоянием КА. Правило обнаружения ошибок можно записать в виде:

$E \rightarrow$ } | C_1} | K_1} | S_2d | S_2* | S_2} | S_3d | S_3' | S_3# | S_3* | S_3} | S_4d | S_4+ | S_4* | S_4) | S_4} | K_5d | K_5* | K_5} | D_1a | D_1b | D_1' | D_1+ | D_1# | D_1(| D_1* | D_1) | D_1{ | D_1} | D_2* | D_2} | D_3d | D_3' | D_3# | D_3* | D_3} | $C_1\perp$ | $C_2\perp$ | $C_3\perp$ | $C_4\perp$ | $K_1\perp$ | $K_2\perp$ | $K_3\perp$ | $K_4\perp$ | $S_1\perp$ | $S_4\perp$

Дополнив грамматику **G** этим правилом, можно построить полностью определенный детерминированный КА. Дуги переходов в состояние `E` этого автомата надо нагрузить функцией выдачи пользователю соответствующего сообщения об ошибке.

СОВЕТ

> Рекомендуем читателям самостоятельно реализовать лексический анализатор на основе построенной грамматики и проверить его работу на любом исходном тексте программы для языка Pascal.

Данный пример достаточно наглядно иллюстрирует, как просто можно решать задачи лексического анализа, используя математический аппарат регулярных грамматик и КА. После построения КА задача разработчика заключается только в

моделировании его работы (что элементарно просто) и реализации всех необходимых функций при переходе КА из одного состояния в другое[7].

Другие примеры построения лексических анализаторов можно найти в книгах [42, 58].

Автоматизация построения лексических анализаторов (программа LEX)

Лексические распознаватели (сканеры) — это не только важная часть компиляторов. Как было показано выше на примерах, лексический анализ применяется во многих других областях, связанных с обработкой текстовой информации на компьютере. Прежде всего, лексического анализа требуют все возможные командные процессоры и утилиты, предусматривающие ввод командных строк и параметров пользователем. Кроме того, хотя бы самый примитивный лексический анализ вводимого текста применяют практически все современные текстовые редакторы и текстовые процессоры. Практически любой более-менее серьезный разработчик программного обеспечения рано или поздно сталкивается с необходимостью решать задачу лексического анализа (разбора)[8].

С одной стороны, задачи лексического анализа имеют широкое распространение, а с другой — допускают четко формализованное решение на основе техники моделирования работы КА. Все это вместе предопределило возможность автоматизации решения данной задачи. И программы, ориентированные на ее решение, не замедлили появиться. Причем, поскольку математический аппарат КА известен уже длительное время, да и с задачами лексического анализа программисты столкнулись довольно давно, то и программам, их решающим, насчитывается не один десяток лет.

Для решения задач построения лексических анализаторов существуют различные программы. Наиболее известной среди них является программа LEX.

LEX — программа для генерации сканеров (лексических анализаторов). Входной язык содержит описания лексем в терминах регулярных выражений. Результатом работы LEX является программа на некотором языке программирования, которая читает входной файл (обычно это стандартный ввод) и выделяет из него последовательности символов (лексемы), соответствующие заданным регулярным выражениям.

---

[7] Рассмотренный выше пример при реализации без использования математического аппарата лексического анализа привел к многочисленным ошибкам разработчика, в результате исправления которых изначально несложный текст программы превратился в путаницу «заплаток», разобраться в которых не смог в конце концов и сам разработчик.

[8] С другой стороны, далеко не каждый разработчик, столкнувшись с этой задачей, понимает, что он имеет дело именно с лексическим анализом текста. Поэтому не всегда такие задачи решаются должным образом, в чем автор мог убедиться на собственном опыте.

История программы LEX тесно связана с историей операционных систем типа UNIX. Эта программа появилась в составе утилит ОС UNIX и в настоящее время входит в поставку практически каждой ОС этого типа. Однако сейчас существуют версии программы LEX практически для любой ОС, в том числе для широко распространенных MS DOS и MS Windows всех версий.

Поскольку LEX появилась в среде ОС UNIX, то первоначально языком программирования, на котором строились порождаемые ею лексические анализаторы, был язык С. Но со временем появились версии LEX, порождающие сканеры и на основе других языков программирования (например, известны версии для языка Pascal).

СОВЕТ

> Для поиска нужной версии программы лексического анализа LEX автор советует заинтересованным лицам прежде всего обратиться во всемирную сеть Интернет. В настоящее время существует огромное множество версий этой и подобных ей программ автоматизации построения лексических анализаторов.

Принцип работы LEX достаточно прост: на вход ей подается текстовый файл, содержащий описания нужных лексем в терминах регулярных выражений, а на выходе получается файл с текстом исходной программы сканера на заданном языке программирования (обычно — на С). Текст исходной программы сканера может быть дополнен вызовами любых функций из любых библиотек, поддерживаемых данным языком и системой программирования. Таким образом, LEX позволяет значительно упростить разработку лексических анализаторов, практически сводя эту работу к описанию требуемых лексем в терминах регулярных выражений.

Более подробную информацию о работе с программой LEX можно получить в [3, 9, 22, 31, 53, 55, 58, 62].

Контрольные вопросы и задачи

Вопросы

1. Какие грамматики относятся к регулярным грамматикам? Назовите два класса регулярных грамматик. Как они соотносятся между собой?
2. В чем заключается отличие автоматных грамматик от других регулярных грамматик? Всякая ли регулярная грамматика является автоматной? Всякая ли регулярная грамматика может быть преобразована к автоматному виду?
3. Если язык, заданный регулярной грамматикой, содержит пустую цепочку, то может ли он быть задан автоматной грамматикой?
4. Можно ли граф переходов конечного автомата использовать для однозначного определения данного автомата (и если нет — то почему)?

5. Всегда ли недетерминированный КА может быть преобразован к детерминированному виду (если нет — то в каких случаях)?
6. Какое максимальное количество состояний может содержать ДКА после преобразования из недетерминированного КА? Всегда ли это количество состояний можно сократить?
7. Чем различаются таблица лексем и таблица идентификаторов? В какую из этих таблиц лексический анализатор не должен помещать ключевые слова, разделители и знаки операций?
8. Являются ли регулярными множествами следующие множества:
   - множество целых натуральных чисел;
   - множество вещественных чисел;
   - множество всех слов русского языка;
   - множество всех возможных строковых констант в языке Pascal;
   - множество иррациональных чисел;
   - множество всех возможных вещественных констант языка C?
9. На основании свойств регулярных выражений докажите следующие тождества для произвольных регулярных выражений α, β, γ и δ:
   1. $(\alpha + \beta)(\delta + \gamma) = \alpha\delta + \alpha\gamma + \beta\delta + \beta\gamma$
   2. $\delta(\alpha + \beta)\gamma = \delta\alpha\gamma + \delta\beta\gamma$
   3. $\beta + \beta\alpha + \beta\alpha^* = \beta\alpha^*$
   4. $\beta + \beta\alpha\alpha^* + \beta\alpha\alpha\alpha^* = \beta\alpha^*$
   5. $\delta\alpha\gamma 0^* + \delta\alpha^*\gamma + \delta\gamma = \delta\alpha^*\gamma$
   6. $(0^* + \alpha\alpha^* + \alpha\alpha\alpha^*)^* = \alpha^*$
10. Используя свойства регулярных множеств, проверьте, являются ли истинными следующие тождества для произвольных регулярных выражений α и β:
    1. $(\alpha + \beta)^* = (\alpha^*\beta^*)^*$
    2. $(\alpha\beta)^* = \alpha^*\beta^*$
    3. $(\alpha + \lambda)^* = \alpha^*$
    4. $\alpha^*\beta^* + \beta^*\alpha^* = \alpha^*\beta^*\alpha^*$
    5. $\alpha^*\alpha^* = \alpha^*$
11. Почему возможны две формы записи уравнений с регулярными коэффициентами? Как они соотносятся с двумя классами регулярных грамматик?
12. Можно ли для языка, заданного леволинейной грамматикой, построить праволинейную грамматику, задающую эквивалентный язык?

13. Всякая ли регулярная грамматика является однозначной? Если нет, то приведите пример неоднозначной регулярной грамматики.

14. Какие из следующих утверждений являются истинными:

- если язык задан регулярным множеством, то он может быть задан праволинейной грамматикой;
- если язык задан КА, то он может быть задан КС-грамматикой;
- если язык задан КС-грамматикой, то для него можно построить регулярное выражение;
- если язык задан КА, то для него можно построить регулярное выражение.

15. Можно ли для двух леволинейных грамматик доказать, что они задают один и тот же язык? Можно ли выполнить то же самое доказательство для леволинейной и праволинейной грамматик?

Задачи

1. Определите, какие из перечисленных ниже грамматик являются регулярными, леволинейными, праволинейными, автоматными:

   G_1({".",+,−,0,1}, {<число>,<часть>,<цифра>,<осн>}, **P**$_1$, <число>)

   P$_1$: <число> → +<осн> | −<осн> | <осн>

   <осн> → <часть>.<часть> | <часть>. | <часть>

   <часть> → <цифра> | <часть><цифра>

   <цифра> → 0 | 1

   G_2({".",+,−,0,1}, {<число>,<часть>,<осн>}, **P**$_2$, <число>)

   P$_2$: <число> → +<осн> | −<осн> | <осн>

   <осн> → <часть>. | <часть> | <осн>0 | <осн.>1

   <часть> → 0 | 1 | <часть>0 | <часть>1

   G_3({".",+,−,0,1}, {<число>,<часть>,<осн>}, **P**$_3$, <число>)

   P$_3$: <число> → +<осн> | −<осн> | <осн>

   <осн> → 0 | 1 | 0.<часть> | 1.<часть> | 0<осн> | 1<осн>

   <часть> → 0 | 1 | 0<часть> | 1<часть>

   G_4({".",+,−,0,1}, {<знак>,<число>,<часть>}, **P**$_4$, <число>)

   P$_4$: <число> → <знак>0 | <знак>1 | <часть>. | <число>0 | <число>1

   <часть> → <знак>0 | <знак>1 | <часть>0 | <часть>1

<знак> → λ | + | −

G_5({".",+,−,0,1}, {<знак>,<число>,<часть>,<осн>}, $\mathbf{P}_5$, <число>)

$\mathbf{P}_5$: <число> → <часть>. | <осн>0 | <осн>1 | <часть>0 | <часть>1

<осн> → <часть>. | <осн>0 | <осн>1

<часть> → 0 | 1 | <знак>0 | <знак>1 | <часть>0 | <часть>1

<знак> → + | −

2. Преобразуйте к автоматному виду грамматики $\mathbf{G}_3$ и $\mathbf{G}_4$ из задачи №1.
3. Постройте КА на основании грамматик $\mathbf{G}_4$ или $\mathbf{G}_5$ из задачи №1. Преобразуйте построенный автомат к детерминированному виду.
4. Задан КА: M({H,I,S,E,F,G}, {b,d}, δ, H, {S}), δ(H,b)={I,S}, δ(H,d)={E}, δ(I,b)={I,S}, δ(I,d)={I,S}, δ(E,b)={E,F}, δ(E,d)={E,F}, δ(F,b)={E}, δ(F,d)={E}, δ(G,b)={F,S}, δ(G,d)={S}; минимизируйте его и постройте эквивалентный ДКА.
5. Задан КА: M({S,R,Z}, {a,b}, δ, S, {Z}), δ(S,a)={S,R}, δ(R,b)={R}, δ(R,a)={Z}. Преобразуйте его к детерминированному виду и минимизируйте полученный КА.
6. Постройте и решите систему уравнений с регулярными коэффициентами для грамматики $\mathbf{G}_4$ из задачи №1. Какой язык задает эта грамматика?
7. Докажите, что уравнение с регулярными коэффициентами $X=\alpha X+\beta$ имеет решение $X=\alpha^*(\beta+\gamma)$, где γ обозначает произвольное множество, в том случае, когда множество, обозначенное выражением α, содержит пустую цепочку. (Указание: поскольку множество, обозначенное выражением α, содержит пустую цепочку, то выражение α можно представить в виде $\alpha=\delta+\lambda$, при этом $\alpha^*=\delta^*$).
8. Решите систему уравнений с регулярными коэффициентами над алфавитом $\mathbf{V}$ = {0,1}:

$X_1 = 0X_2 + 1X_1 + \lambda$

$X_2 = 0X_3 + 1X_2$

$X_3 = 0X_1 + 1X_3$

9. В языке C++ возможны два типа комментариев: обычный комментарий, ограниченный символами /* и */, и строчный комментарий, начинающийся с символов // и заканчивающийся концом строки. Постройте грамматику для этого языка, преобразуйте ее к автоматному виду.

Упражнения

1. Докажите (любым способом), что грамматики $\mathbf{G}_3$, $\mathbf{G}_4$ и $\mathbf{G}_5$ из задачи №1 задают один и тот же язык.
2. Постройте регулярную грамматику, которая бы описывала язык строк, целочисленных констант и строковых констант языка Pascal. Постройте и решите систему уравнений с регулярными коэффициентами на основе этой грамматики.
3. В языке C++ возможны два типа комментариев: обычный комментарий, ограниченный символами `/*` и `*/`, и строчный комментарий, начинающийся с символов `//` и заканчивающийся концом строки. Постройте сканер, который бы находил и исключал из входного текста все комментарии языка C++.
4. С помощью леммы о разрастании для регулярных языков докажите, что язык $L_1 = \{a^n,b^n \mid n > 2\}$ не является регулярным, а язык $L_2 = \{a^n b^m \mid n > 1, m > 2\}$ — является регулярным.
5. Постройте сканер, который выделял бы из текста входной программы на языке C все содержащиеся в ней строковые константы и записывал их в отдельный файл. Строковые константы в языке C состоят из строк, одиночных символов и кодов символов. Длинные строковые константы могут продолжаться на нескольких строках исходной программы подряд. Строки ограничены символами `""` (двойные кавычки), одиночные символы - символами `''`(одинарные кавычки), а коды символов начинаются со знака \ (обратная косая черта) и содержат цифры[9].

[9] Следует заметить, что эта задача отнюдь не так проста, как может показаться. Программа должна уметь не только находить в исходном тексте строки и объединять их между собой, но и правильно игнорировать комментарии (которых в C два типа). Поэтому лучше сначала корректно описать грамматику, а потом на ее основе построить автомат.