

Свойства регулярных языков, лексические анализаторы

1 Свойства регулярных языков

- Лемма о накачке
- Проблемы, разрешимые для регулярных языков

2 Лексические анализаторы

- Понятие о лексических анализаторах
- Практическая реализация лексических анализаторов
- Автоматизация поиска в тексте и синтеза лексических анализаторов

Итак, мы с вами увидели, что регулярные языки можно задать тремя разными способами: 1. с помощью порождающих регулярных (автоматных) грамматик; 2. с помощью конечных автоматов; 3. с помощью регулярных выражений.

В последнем случае регулярные выражения фактически обозначают регулярные множества. А регулярные множества оказываются замкнутыми относительно многих операций: объединения, пересечения, дополнения. Это означает, что если L_1, L_2 - регулярные языки, то выполнение соответствующей операции, для которой регулярный язык замкнут, над словами языков L_1, L_2 дает также регулярный язык. В частности справедливо следующее утверждение:

Теорема

Класс автоматных языков замкнут относительно итерации, конкатенации и объединения.

Доказательство этого утверждения прямо следует из определения регулярного множества: если имеем два регулярных множества, то их объединение, конкатенация, итерация также являются регулярными множествами.

Очевидно, что не все языки регулярны, и существует мощный механизм доказательства нерегулярности некоторых языков, известный как лемма о накачке (или лемма о разрастании).

Лемма о накачке

Для начала рассмотрим на некотором примере, как доказать, что язык не является регулярным. Для этого прежде всего рассмотрим довольно простое утверждение, которое известно как принцип Дирихле:

Теорема

Если кролики рассажены в клетки, причём число кроликов больше числа клеток, то хотя бы в одной из клеток находится более одного кролика.

Пусть имеется язык $L = \{0^n 1^n : n \geq 1\}$, докажем, что он не является регулярным. Этот язык состоит из всех цепочек вида 01, 0011, 000111 и так далее, содержащих один или несколько нулей, за которыми следует точно такое же количество единиц.

Если бы данный язык был регулярным, то для него существовал бы соответствующий детерминированный конечный автомат. Заметим, что автомат должен иметь только конечное множество состояний, допустим это число M . Поскольку n может быть любым числом, превосходящим конечное число состояний автомата M , то всегда существуют два разных $n_1 \neq n_2$, для которых на входных цепочках 0^{n_1} и 0^{n_2} автомат вынужден находиться в одном и том же состоянии, обозначим его q_x (принцип Дирихле). Если далее для такого случая на входе 1, то детерминированный автомат переходит из состояния q_x в какое-то новое состояние q_y , независимо от того, сколько до этого было нулей на входе. Но теперь из состояния q_y автомат должен уметь различить входные цепочки 1^{n_1} и 1^{n_2} , что очевидно невозможно. Поскольку ясно, что автомат уже не может различить, какое количество нулей n_1 или n_2 было на входе.

Несмотря на представленное доказательство этого частного случая, существует более универсальное утверждение, которое известно как теорема о накачке.

Теорема (теорема о накачке)

Пусть L — регулярный бесконечный язык. Существует константа n (зависящая от L), для которой каждую цепочку w из языка L , удовлетворяющую неравенству $|w| \geq n$, можно разбить на три цепочки $w = xuz$ так, что выполняются следующие условия.

- 1 $y \neq \epsilon$.
- 2 $|xy| \leq n$.
- 3 Для любого $k \geq 0$ цепочка xy^kz также принадлежит L .

Это значит, что всегда можно найти такую цепочку y недалеко от начала цепочки w , которую можно “накачать”. Таким образом, если цепочку y повторить любое число раз или удалить (при $k = 0$), то результирующая цепочка все равно будет принадлежать языку L .

Доказательство.

Пусть L какой-то регулярный язык. Тогда рассмотрим соответствующий ему автомат. Данный автомат принимает конечное множество состояний, обозначим их количество n . Тогда рассмотрим такую цепочку $w = a_0a_1 \dots a_m, m \geq n$, которая принадлежит этому языку. Это означает, что на цепочке w автомат переходит в заключительное состояние. А поскольку цепочка длиннее количества состояний n , то для цепочки w существует хотя бы одно состояние, через которое мы проходим несколько раз. Обозначим такое состояние q_x .

Разобьем цепочку w на три части $w = xyz, x = a_0a_1 \dots a_i, y = a_{i+1}a_{i+2} \dots a_j, z = a_{j+1}a_{j+2} \dots a_m, 0 \leq i \leq j \leq n$. Предположим, что на цепочке x мы как раз переходим в состояние q_x , тогда существует такое j , что на цепочке y мы тоже перейдем в состояние q_x . Поскольку по окончании цепочки y мы опять перейдем в состояние q_x , тогда ясно, что повторение цепочки y приведет опять в состояние q_x . Поэтому повторение цепочки y возможно сколько угодно раз и при этом получаемая цепочка xy^kz будет все равно принадлежать языку L . □

Использование леммы о накачке

Лемма о накачке позволяет с общих позиций доказывать нерегулярность некоторых языков. Например, нерегулярность того же языка $L = \{0^n 1^n : n \geq 1\}$ можно доказать следующим образом. Возможно три отдельных случая применения леммы о накачке в цепочках $0^n 1^n$:

- Можно попробовать накачать какую-то цепочку из нулей, т.е. если $0^n 1^n$ принадлежит языку, то и $0^{n+m} 1^n$ принадлежит языку, что очевидно неверно.
- Можно попробовать накачать какую-то цепочку из единиц, т.е. $0^n 1^{n+m}$ должно принадлежать языку, что очевидно также неверно.
- Можно попробовать накачать какую-то цепочку из нулей и единиц, т.е. $0^n (0^k 1^l)^m 1^n$ должно принадлежать языку, что очевидно также неверно.

Таким образом, нет ни одной возможности накачать цепочку $0^n 1^n$, принадлежащую языку L . Однако выполнение леммы о накачке автоматически не означает регулярность соответствующего языка. Так есть нерегулярные языки, которые также допускают накачку. Есть конечные регулярные языки, которые, очевидно, не допускают накачку. Лемма лишь утверждает, что если бесконечный язык регулярен, то он обязан допускать накачку. Т.е. это условие является необходимым, но не является достаточным. Поэтому лемма о накачке обычно используется именно для доказательства нерегулярности языка, если удастся показать, что строки никак не могут накачиваться.

Задание 1.

Доказать нерегулярность языка $L = \{0^k 1^m : k \leq m\}$.

Заметим, что этот язык допускает накачку. В частности можно взять любую подстроку 1^m и накачивать ее. Таким образом, прямое использование леммы о накачке не проходит.

Для доказательства заметим, что в лемме о накачке утверждается существование некоторой константы n , определяемой количеством состояний соответствующего детерминированного автомата, начиная с которой все подцепочки длины больше n можно накачивать. Возьмем цепочку $0^{n+k} 1^m$, $0 \leq k, m \geq n + k$, тогда подцепочку 0^{n+k} можно пытаться накачивать, но до бесконечности этого следать не получится, так как мы быстро достигнем условия $m \leq n + k$, а в этом случае цепочка уже не будет принадлежать L .

Задание 2.

Доказать нерегулярность языка правильных скобочных выражений.

Данный язык ранее мы описывали с помощью следующей порождающей грамматики $S \rightarrow ()|(S)|SS$. По виду правил, эта грамматика конечно не является регулярной. Однако возможно существует и регулярная грамматика, задающая тот же язык.

Но, оказывается, можно доказать, что такой грамматики не существует.

Используем для этого следующее утверждение: если L_1, L_2 - регулярные языки, то $L = L_1 \cap L_2$ - тоже регулярный язык. Для доказательства нерегулярности языка правильных скобочных выражений L_1 , рассмотрим регулярный язык $L_2 = (^*)^*$ (он является регулярным, поскольку мы используем только операцию итерации над скобками), данный язык можно задать $L_2 = \{(^m,)^n : n \geq 0, m \geq 0\}$. Предположим, что L_1 является регулярным, тогда $L_1 \cap L_2 = (^n)^n$ (поскольку L_1 содержит только сбалансированное количество скобок, то от L_2 остаются только цепочки $(^n)^n$.) Значит язык $(^n)^n$ тоже должен быть регулярным. Но это не так, поскольку на языке $0^n 1^n$ мы показали, что он не является регулярным.

Проблемы, разрешимые для регулярных языков

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

Ниже представлен перечень таких разрешимых для регулярных языков проблем:

- *Проблема эквивалентности.* Для двух регулярных языков L_1, L_2 требуется проверить, являются ли они эквивалентными.
- *Проблема принадлежности.* Для данного регулярного языка L и цепочке символов α требуется проверить, принадлежит ли эта цепочка α данному языку L .
- *Проблема пустоты языка.* Требуется проверить, содержит ли заданный язык хотя бы одну непустую цепочку.
- *Проблема однозначности.* Для заданной регулярной грамматики построить эквивалентную ей однозначную регулярную грамматику.

Определение

Лексема - структурная единица языка, которая состоит из элементарных символов языка и не содержит в своем составе других структурных единиц языка.

Лексемами естественных языков являются слова. Лексемами языков программирования являются идентификаторы, константы, ключевые слова, знаки операций и т.д.

Определение

Лексический анализатор - 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 другим компонентам компилятора (в частности синтаксическому анализатору).

Понятие о лексических анализаторах

Перечень лексем, полученных лексическим анализатором записывается в так называемую таблицу лексем. Каждой лексеме в таблице лексем соответствует некий уникальный условный код, зависящий от типа лексем. Информация о некоторых типах лексем также заносится в таблицу идентификаторов.

Таблица лексем содержит весь текст программы, разбитый на лексемы. Один идентификатор может встречаться в этой таблице много раз, в тех местах, где он встречается в самой программе. В таблице идентификаторов каждый идентификатор употребляется только один раз, а сама таблица идентификаторов служит для хранения текущего значения идентификатора.

Как правило, язык констант, идентификаторов и т.д. является регулярным. Следовательно, основой для реализации лексических анализаторов служат регулярные грамматики и конечные автоматы.

При работе лексический анализатор должен, кроме распознавания лексем, определять границы лексем, которые в тексте явно не указаны, выполнять действия по сохранению информации об обнаруженной лексеме, или выдавать сообщение об ошибке, если лексема неверна.

Понятие о лексических анализаторах

Нужно отметить, что определение границ лексем является для лексического анализатора наиболее сложной задачей. В ряде случаев он не может с ней справиться. Например, в коде на языке C возможна такая конструкция $k = i + + + + + j$. Это означает, что постинкремент переменной i складывается с прединкрементом переменной j . Увидеть это лексический анализатор сможет только просмотрев всю строчку и перебрав все возможные неправильные варианты. Но это можно сделать только на этапе синтаксического анализа. Поэтому в таких случаях лексический анализатор вынужден взаимодействовать с синтаксическим анализатором. Возможно две стратегии такого взаимодействия: последовательное взаимодействие, параллельное взаимодействие.

При последовательной стратегии лексический анализатор просматривает весь код от начала до конца и преобразует его в таблицу лексем. Если лексический анализатор не смог распознать какую-то лексему, то она считается ошибочной.

При параллельном взаимодействии лексический анализатор выделяет очередную лексему и передает ее синтаксическому анализатору. Синтаксический анализатор, выполнив разбор очередной конструкции языка, может подтвердить правильность найденной лексемы и обратиться за следующей или же отвергнуть найденную лексему.

Понятие о лексических анализаторах

Последовательная работа лексического анализатора является более простой. Поэтому разработчики языков программирования стремятся организовать взаимодействие именно таким образом. Для большинства языков программирования границы лексем распознаются по заданным терминальным символам - пробелам, знакам операций, разделителям - точкам с запятой, запятым.

На практике разработчики компиляторов сознательно идут на то, что исключают некоторые правильные, но неоднозначные варианты. Попытки усложнить лексический анализ неизбежно приведут к необходимости взаимодействовать с синтаксическим анализатором, а это неизбежно снизит эффективность работы всего компилятора. Возникающие накладные расходы никак не оправдаются достигаемым эффектом - распознаванием строк с сомнительными лексемами.

Практическая реализация лексических анализаторов

При практической реализации компилятор может иметь в своем составе не один, а несколько лексических анализаторов, каждый из которых предназначен для распознавания своего типа лексем. Тогда алгоритм работы может выглядеть следующим образом:

- из входного потока выбирается очередной символ, в зависимости от которого запускается нужный лексический анализатор.
- запущенный сканер просматривает входной поток, выделяя символы, которые входят в лексему, до обнаружения очередного символа, который может ограничить лексему, либо до обнаружения ошибочного символа.
- при успешном распознавании лексемы, она заносится в таблицу лексем и идентификаторов, алгоритм возвращается к начальному этапу с того места, на котором остановился сканер.
- если выявлена ошибка, то в зависимости от реализации лексического анализатора, он либо прекращает работу или пытается проанализировать и выделить следующую лексему.

В целом техника построения лексических анализаторов основывается на синтезе детерминированного или недетерминированного конечного автомата с дополнением функциями распознавания ошибок и заполнения таблиц лексем и идентификаторов.

Практическая реализация лексических анализаторов

Рассмотрим пример синтеза лексического анализатора на примере выделения из текста лексем, представляющих собой целочисленные константы.

Каждая целочисленная константа должна состоять из цифр 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, она может начинаться со знаков плюс + или минус − и не может начинаться с незначащих нулей.

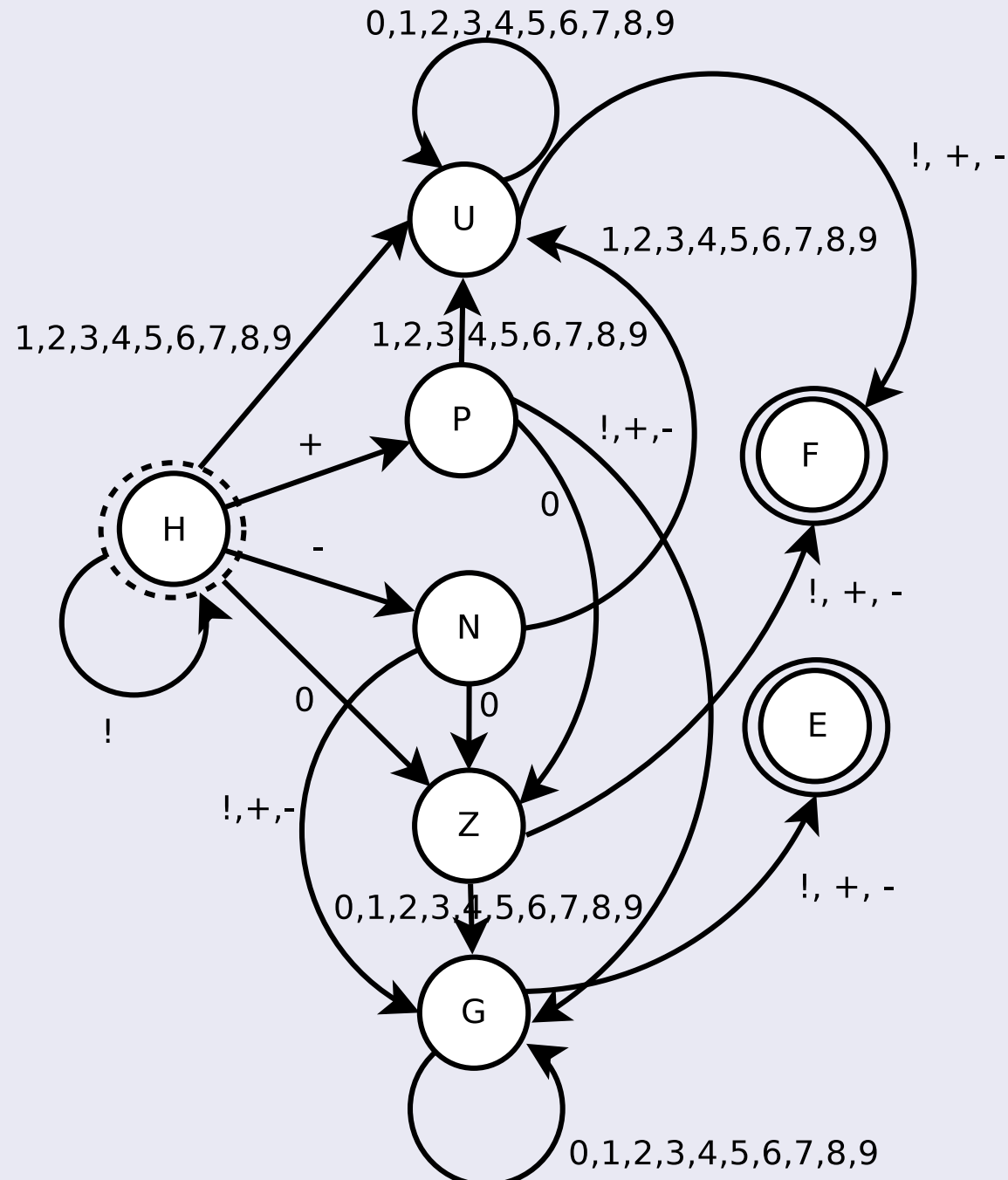
Например, такие варианты допустимы: +24210, −2197, 975, 0, +0, −0, а такие варианты нет: +001233, −0344, 000328, 021.

Будем обозначать все символы отличные от смысловых из алфавита +, −, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 одним символом !. Такое преобразование над входным текстом всегда можно без труда выполнить. Разделителями идентификаторов является символ !, а также символы +, −, если они встречены в середине числа.

Будем считать, что при работе конечного автомата после прихода его в конечное состояние он либо распознал идентификатор, либо выявил ошибку. Далее его можно опять запускать с текущей позиции для выявления следующей лексемы.

Построим детерминированный конечный автомат, отвечающий данной задаче лексического анализа.

Синтез лексического анализатора в виде конечного автомата



Функция формирования текста из файла

```
def read_file(f):  
    ff = open(f, 'r')  
    result=""  
    for i in ff:  
        result+=i  
    return result
```

Реализация вспомогательных функций

Функция заменяет все символы, кроме символов алфавита, на заданный символ

```
def replace(text, abc, sep):  
    lst=list(text)  
    for i in range(len(lst)):  
        if not(lst[i] in abc):  
            lst[i]=sep  
    result=""  
    for i in lst:  
        result=result+str(i)  
    return result
```

Модифицируйте программный код детерминированного конечного автомата для реализации лексического анализатора целочисленных констант.

Утилита grep

Для поиска в тексте различных образцов, а также построения лексических анализаторов, распознающих заданные лексемы в тексте, очень удобно использовать регулярные выражения.

Рассмотрим данный вопрос на практических примерах. Для начала рассмотрим систему обозначений, используемую в Unix/Linux для расширенных регулярных выражений. Расширения UNIX/Linux включают некоторые особенности, в частности, возможность именовать и ссылаться на предыдущие цепочки, соответствующие шаблону, что, фактически, позволяет распознавать нерегулярные языки.

Наиболее функциональный механизм в Linux/Unix - это утилита grep (Global Regular Expression and Print - глобальный поиск регулярного выражения и печать). Утилита grep понимает три разные версии синтаксиса регулярных выражений: «Базовый» (BRE), «расширенный» (ERE) и «perl» (PCRE). В GNU grep нет разницы в доступном функционале между базовым и расширенным синтаксисом. В других реализациях базовые регулярные выражения менее мощный механизм. Регулярные выражения, совместимые с Perl, дают дополнительную функциональность и задокументированы в описании утилит `pcresyntax` и `pcgrepattern`.

Утилита grep

- Символ `.` означает один любой символ.
- Символы, заключенные в квадратные скобки `[...]` означают любой символ из перечисленных в этих скобках, если перед символами в квадратных скобках стоит знак крышки `^` `[...]` это означает любой, кроме перечисленных в квадратных скобках. Также можно указывать диапазоны, например `[0-9]` - означает любую цифру, `[a-w0-9]` - означает любую строчную букву и любую цифру. Определенные диапазоны имеют специальные обозначения `[:alnum:]`, `[:alpha:]`, `[:digit:]`, `[:lower:]`, `[:space:]`, `[:upper:]`, которые в целом понятны из названий. Например, `[: alnum:]` означает символьный класс чисел и буквы в текущей локали.
- Если перед символом поставить знак крышки `^`, то это будет означать, что строка должна начинаться с этого символа, а если после символа поставить знак доллара `$`, это будет означать, что строка должна заканчиваться этим символом.

Утилита grep

- ? - предыдущий элемент является необязательным и соответствует не более одного раза.
- * предыдущий элемент будет найден ноль или более раз.
- + предыдущий элемент будет найден один или несколько раз.
- {n} предыдущий элемент встречается ровно n раз.
- {n,} предыдущий элемент встречается n или более раз.
- {, m} предыдущий элемент встречается не более m раз. Это GNU расширение.
- {n, m} предыдущий элемент соответствует как минимум n раз, но не более чем в m раз.

Конкатенация Два регулярных выражения могут быть объединены; в результате регулярное выражение соответствует любой строке, образованной объединением двух подстрок, которые отвечают конкатенированным выражениям.

Чередование

Два регулярных выражения могут быть соединены инфиксным оператором |, тогда результирующее регулярное выражение соответствует либо первой строке, либо второй строке (работает, если запускать grep с расширением E).

Приоритет

Повторение имеет приоритет перед конкатенацией, которая, в свою очередь, имеет приоритет перед чередованием. Целое выражение может быть заключено в круглые скобки, чтобы переопределить эти правила приоритета и сформировать подвыражение.

Для того, чтобы использовать в регулярных выражениях символы ?, +, {, }, |, |, (,) их нужно экранировать символом \

Утилита grep - примеры использования

```
juna@artamonov: ~  
Файл Правка Вид Поиск Терминал Справка  
juna@artamonov:~$ echo "12345678"|grep ...  
12345678  
juna@artamonov:~$ echo "qwerty764 42fd"|grep "[wrt]"  
qwerty764 42fd  
juna@artamonov:~$ echo "qwerty764 42fd"|grep "[^wrt]"  
qwerty764 42fd  
juna@artamonov:~$ echo "qwerty764 42fd"|grep "[2-4a-h]"  
qwerty764 42fd  
juna@artamonov:~$ echo "qwerty764 42fd"|grep "[[:digit:]]"  
qwerty764 42fd  
juna@artamonov:~$ echo "qwerty764 42fd"|grep "[[:space:]]"  
qwerty764 42fd  
juna@artamonov:~$ echo "qwerty764 42fd"|grep "^[[:space:]]"  
qwerty764 42fd  
juna@artamonov:~$ echo "qwerty764 42fd"|grep "[[:alpha:]]"  
qwerty764 42fd  
juna@artamonov:~$ echo "qwerty764 42fd"|grep "^[[:alpha:]]"  
qwerty764 42fd  
juna@artamonov:~$ echo "qWERTty764 42fD"|grep "[[:upper:]]"  
qWERTty764 42fD  
juna@artamonov:~$ echo "qWERTty764 42fD"|grep "[[:lower:]]"  
qWERTty764 42fD  
juna@artamonov:~$
```


Утилита grep - примеры использования

```
juna@artamonov: ~  
Файл Правка Вид Поиск Терминал Справка  
juna@artamonov:~$ echo "qWertTty764 42fD"|grep "^q"  
qWertTty764 42fD  
juna@artamonov:~$ echo "qWertTty764 42fD"|grep "D$"  
qWertTty764 42fD  
juna@artamonov:~$ echo "qWertTty764 42fD"|grep -E "T?"  
qWertTty764 42fD  
juna@artamonov:~$ echo "qWerttty764 42fD"|grep -E "t*"  
qWerttty764 42fD  
juna@artamonov:~$ echo "qWertttyyyy764 42fD"|grep -E "t+ "  
qWertttyyyy764 42fD  
juna@artamonov:~$ echo "qWertttyyyy764 42fD"|grep -E "t{2}"  
qWertttyyyy764 42fD  
juna@artamonov:~$ echo "qWertttyyyy764 42fD"|grep -E "y{2}"  
qWertttyyyy764 42fD  
juna@artamonov:~$ echo "qWertttyyyy764 42fD"|grep -E "y{3}"  
qWertttyyyy764 42fD  
juna@artamonov:~$ echo "qWertttyyyy764 42fD"|grep -E "y{3,}"  
qWertttyyyy764 42fD  
juna@artamonov:~$ echo "qWertttyyyy764 42fD"|grep -E "y{,3}"  
qWertttyyyy764 42fD  
juna@artamonov:~$ echo "qWertttyyyy764 42fD"|grep -E "y{1,2}"  
qWertttyyyy764 42fD  
juna@artamonov:~$ echo "qWertttyyyy764 42fD"|grep -E "y{1,2}"  
qWertttyyyy764 42fD
```

Утилита grep - примеры использования

```
juna@artamonov: ~  
Файл Правка Вид Поиск Терминал Справка  
juna@artamonov:~$ echo "qWertTty764 42fD"|grep -E "T?"  
qWertTty764 42fD  
juna@artamonov:~$ echo "qWertttyyyy764 42fD"|grep -E "t+"  
qWertttyyyy764 42fD  
juna@artamonov:~$ echo "qWerttty764 42fD"|grep -E "t*"  
qWerttty764 42fD  
juna@artamonov:~$ echo "qWertttyyyy764 42fD"|grep -E "We{1}"  
qWertttyyyy764 42fD  
juna@artamonov:~$ echo "qWertttyyyy764 42fD"|grep -E "We{2}"  
juna@artamonov:~$ echo "qWertttyyyy764 42fD"|grep -E "We{1}|y{,3}"  
qWertttyyyy764 42fD  
juna@artamonov:~$ echo "qWertttyyyy764 42fD"|grep -E "tt*+(y)"  
qWertttyyyy764 42fD  
juna@artamonov:~$ echo "qWerttt{{{}}}yyy)()(64 42fD"|grep -E "[\\{\\}\\(\\)]"  
qWerttt{{{}}}yyy)()(64 42fD  
juna@artamonov:~$
```

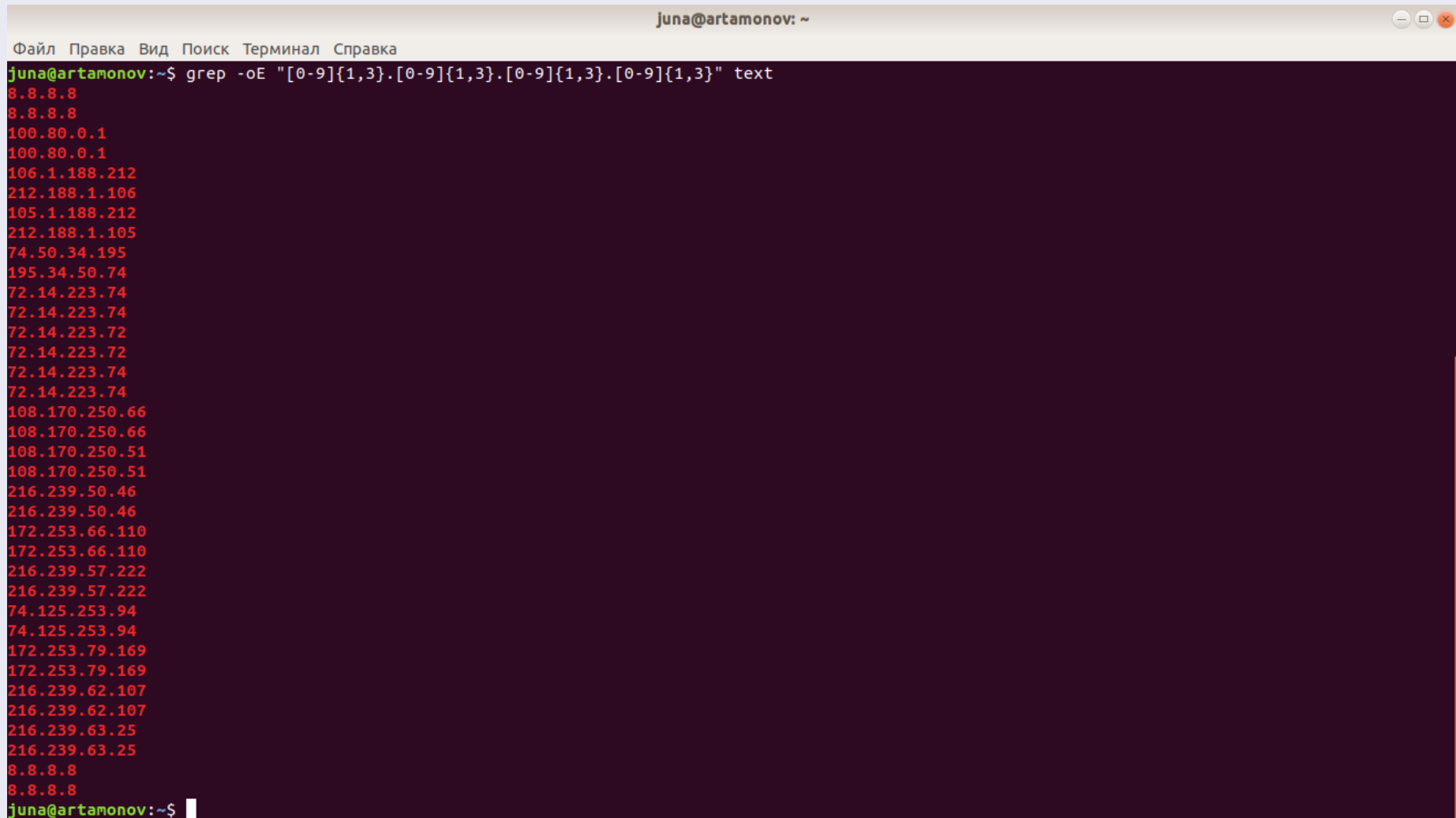
Утилита grep - решение практических задач

Выделить из текста список IP адресов

```
juna@artamonov: ~  
Файл Правка Вид Поиск Терминал Справка  
juna@artamonov:~$ traceroute 8.8.8.8  
traceroute to 8.8.8.8 (8.8.8.8), 30 hops max, 60 byte packets  
1  * * *  
2  100.80.0.1 (100.80.0.1)  6.532 ms  7.008 ms  6.643 ms  
3  106.1.188.212.in-addr.arpa (212.188.1.106)  8.600 ms  9.041 ms  8.951 ms  
4  105.1.188.212.in-addr.arpa (212.188.1.105)  6.974 ms  7.787 ms  7.574 ms  
5  74.50.34.195.in-addr.arpa (195.34.50.74)  10.009 ms  9.935 ms  9.980 ms  
6  72.14.223.72 (72.14.223.72)  9.963 ms  8.307 ms  8.367 ms  
7  108.170.250.146 (108.170.250.146)  8.402 ms  108.170.250.99 (108.170.250.99)  
9.228 ms  108.170.250.113 (108.170.250.113)  7.428 ms  
8  * * 216.239.50.46 (216.239.50.46)  24.034 ms  
9  72.14.238.168 (72.14.238.168)  36.251 ms  72.14.235.69 (72.14.235.69)  21.958  
ms  172.253.66.110 (172.253.66.110)  25.078 ms  
10  172.253.79.237 (172.253.79.237)  22.717 ms  209.85.251.41 (209.85.251.41)  32  
.966 ms  142.250.56.215 (142.250.56.215)  24.458 ms  
11  * * *  
12  * * *  
13  * * *  
14  * * *  
15  * * *  
16  * * *  
17  * * *  
18  * * *  
19  * * *  
20  8.8.8.8.in-addr.arpa (8.8.8.8)  24.369 ms * 26.424 ms
```

Утилита грег - решение практических задач

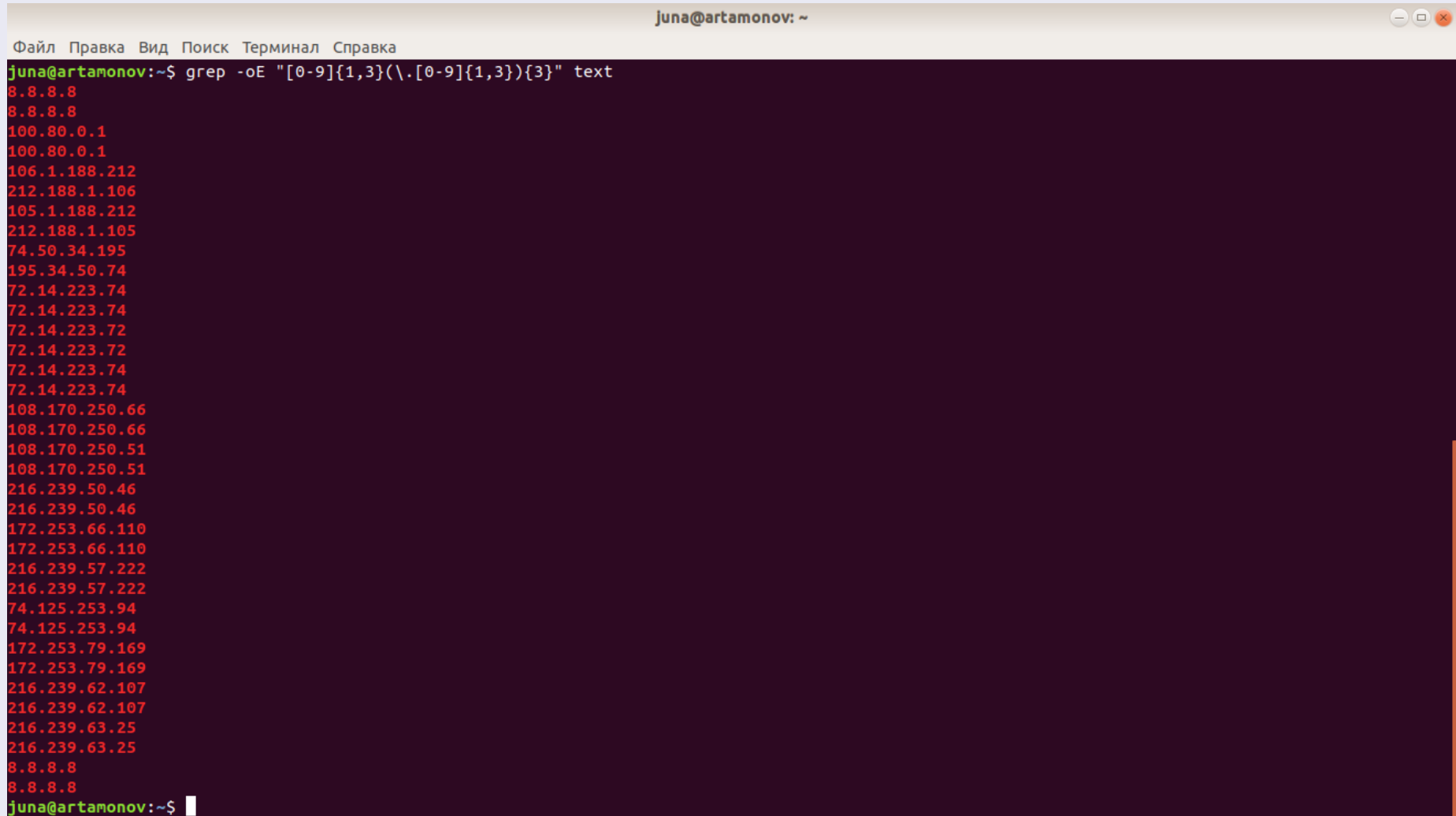
Решение



```
juna@artamonov: ~  
Файл Правка Вид Поиск Терминал Справка  
juna@artamonov:~$ grep -oE "[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}" text  
8.8.8.8  
8.8.8.8  
100.80.0.1  
100.80.0.1  
106.1.188.212  
212.188.1.106  
105.1.188.212  
212.188.1.105  
74.50.34.195  
195.34.50.74  
72.14.223.74  
72.14.223.74  
72.14.223.72  
72.14.223.74  
72.14.223.74  
108.170.250.66  
108.170.250.66  
108.170.250.51  
108.170.250.51  
216.239.50.46  
216.239.50.46  
172.253.66.110  
172.253.66.110  
216.239.57.222  
216.239.57.222  
74.125.253.94  
74.125.253.94  
172.253.79.169  
172.253.79.169  
216.239.62.107  
216.239.62.107  
216.239.63.25  
216.239.63.25  
8.8.8.8  
8.8.8.8  
juna@artamonov:~$
```

Утилита grep - решение практических задач

Решение (усовершенствование)



```
juna@artamonov: ~  
Файл Правка Вид Поиск Терминал Справка  
juna@artamonov:~$ grep -oE "[0-9]{1,3}(\.[0-9]{1,3}){3}" text  
8.8.8.8  
8.8.8.8  
100.80.0.1  
100.80.0.1  
106.1.188.212  
212.188.1.106  
105.1.188.212  
212.188.1.105  
74.50.34.195  
195.34.50.74  
72.14.223.74  
72.14.223.74  
72.14.223.72  
72.14.223.72  
72.14.223.74  
72.14.223.74  
108.170.250.66  
108.170.250.66  
108.170.250.51  
108.170.250.51  
216.239.50.46  
216.239.50.46  
172.253.66.110  
172.253.66.110  
216.239.57.222  
216.239.57.222  
74.125.253.94  
74.125.253.94  
172.253.79.169  
172.253.79.169  
216.239.62.107  
216.239.62.107  
216.239.63.25  
216.239.63.25  
8.8.8.8  
8.8.8.8  
juna@artamonov:~$
```

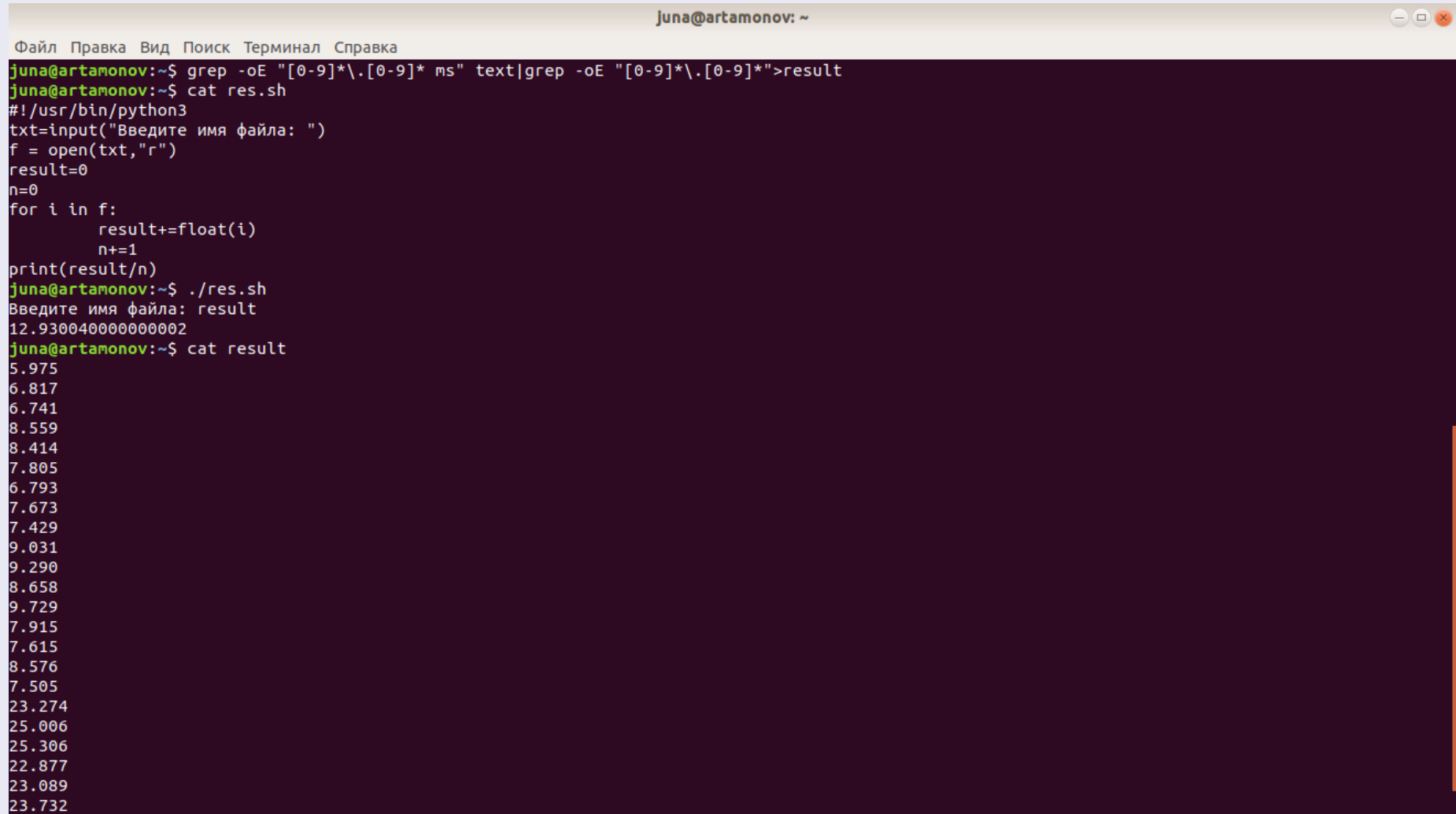
Утилита грег - решение практических задач

Выбрать в полученном файле все временные промежутки и рассчитать среднее арифметическое.

```
juna@artamonov: ~  
Файл Правка Вид Поиск Терминал Справка  
juna@artamonov:~$ grep -E "[0-9]*\.[0-9]* ms" text  
2 100.80.0.1 (100.80.0.1) 5.975 ms 6.817 ms 6.741 ms  
3 106.1.188.212.in-addr.arpa (212.188.1.106) 8.559 ms 8.414 ms 7.805 ms  
4 105.1.188.212.in-addr.arpa (212.188.1.105) 6.793 ms 7.673 ms 7.429 ms  
5 74.50.34.195.in-addr.arpa (195.34.50.74) 9.031 ms 9.290 ms 8.658 ms  
6 72.14.223.74 (72.14.223.74) 9.729 ms 72.14.223.72 (72.14.223.72) 7.915 ms 72.14.223.74 (72.14.223.74) 7.615 ms  
7 108.170.250.66 (108.170.250.66) 8.576 ms * 108.170.250.51 (108.170.250.51) 7.505 ms  
8 * 216.239.50.46 (216.239.50.46) 23.274 ms *  
9 172.253.66.110 (172.253.66.110) 25.006 ms 216.239.57.222 (216.239.57.222) 25.306 ms 74.125.253.94 (74.125.253.94) 22.877 ms  
10 172.253.79.169 (172.253.79.169) 23.089 ms 216.239.62.107 (216.239.62.107) 23.732 ms 216.239.63.25 (216.239.63.25) 24.309 ms  
20 8.8.8.8.in-addr.arpa (8.8.8.8) 21.133 ms * *  
juna@artamonov:~$ grep -oE "[0-9]*\.[0-9]* ms" text|grep -oE "[0-9]*\.[0-9]*"  
5.975  
6.817  
6.741  
8.559  
8.414  
7.805  
6.793  
7.673  
7.429  
9.031  
9.290  
8.658  
9.729  
7.915  
7.615  
8.576  
7.505  
23.274  
25.006  
25.306  
22.877  
23.089  
23.732  
24.309  
21.133  
juna@artamonov:~$
```

Утилита грег - решение практических задач

Окончательное решение



```
juna@artamonov: ~  
Файл Правка Вид Поиск Терминал Справка  
juna@artamonov:~$ grep -oE "[0-9]*\.[0-9]*" ms" text|grep -oE "[0-9]*\.[0-9]*">result  
juna@artamonov:~$ cat res.sh  
#!/usr/bin/python3  
txt=input("Введите имя файла: ")  
f = open(txt,"r")  
result=0  
n=0  
for i in f:  
    result+=float(i)  
    n+=1  
print(result/n)  
juna@artamonov:~$ ./res.sh  
Введите имя файла: result  
12.930040000000002  
juna@artamonov:~$ cat result  
5.975  
6.817  
6.741  
8.559  
8.414  
7.805  
6.793  
7.673  
7.429  
9.031  
9.290  
8.658  
9.729  
7.915  
7.615  
8.576  
7.505  
23.274  
25.006  
25.306  
22.877  
23.089  
23.732
```

Задание

Разработать регулярное выражение, которое выделяет из текста все адреса электронной почты.

Задание

Разработать регулярное выражение, которое выделяет из текста все номера телефонов, записанных в формате 8-XXX-XXX-XX-XX или +7-XXX-XXX-XX-XX.

Описание программного обеспечения

Для автоматизации построения лексических анализаторов существуют различные программы.

- lex — стандартный генератор в Unix
- Flex — альтернативный вариант классической утилиты lex для Linux: <https://github.com/westes/flex>, <http://gnuwin32.sourceforge.net/packages/flex.htm>
- JFlex — генератор на Java <http://jflex.de/>
- Quex — генератор лексических анализаторов для C/C++ <http://quex.sourceforge.net/>
- RE2C - инструмент для генерации основанных на C распознавателей по регулярным выражениям <http://re2c.org/>
- Lex под Windows <http://www.bumblebeesoftware.com/>
- alex — генератор лексических анализаторов для Haskell <http://www.haskell.org/alex/>
- Ply - генератор Lex, Yacc для python <http://dabeaz.com/ply/>

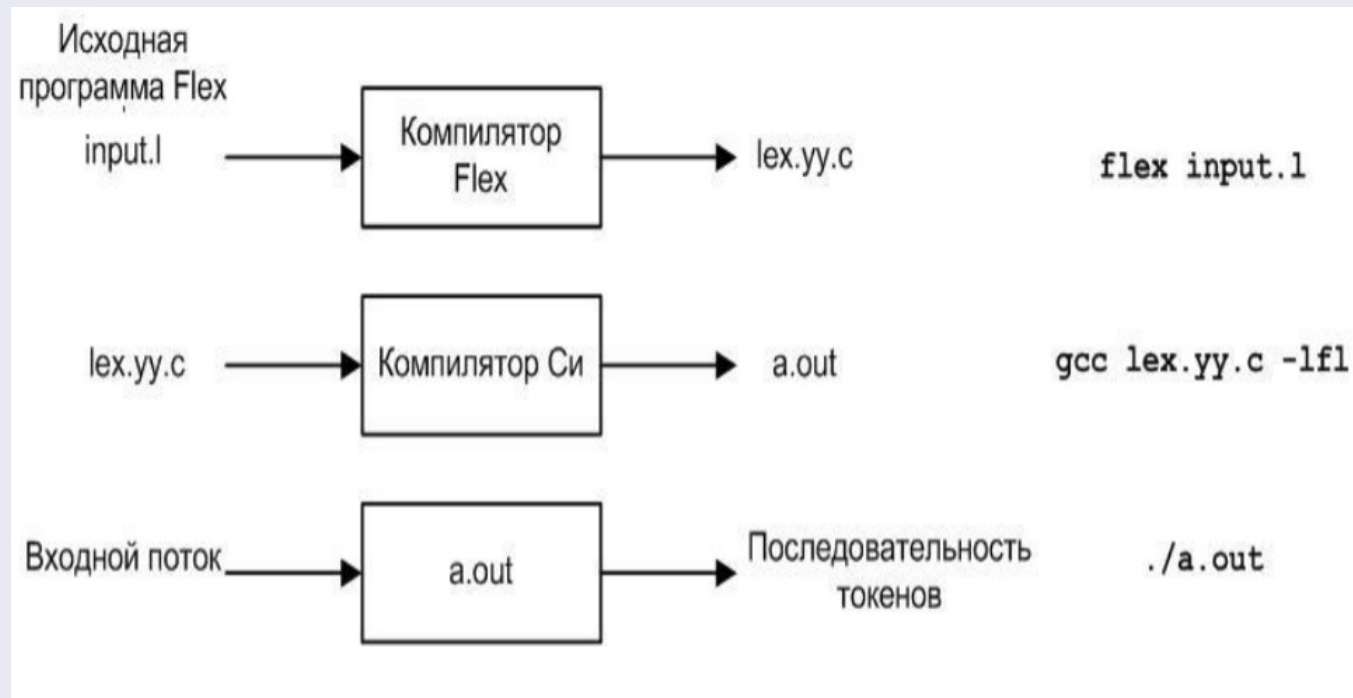
Мы будем рассматривать классический Flex. 1975 – М. Леск (Mike Lesk) и Э. Шмидт (Eric Schmidt) разработали lex, генератор лексических анализаторов. 1987 – В. Пакссон (Vern Paxson) переписал версию lex, написанную на ratfor (расширение Фортрана), на Си и назвал ее flex (Fast Lexical Analyzer Generator)

Литература по flex

- Levine J.R. Flex & bison. – O'Reilly Media, Inc., 2009. – 274 p.
- Niemann T. A Compact Guide To Lex & Yacc. Режим доступа: <https://www.epaperpress.com/lexandyacc/download/LexAndYacc.pdf>
- Hubert B. Lex и YACC в примерах. Режим доступа: <http://rus-linux.net/lib.php?name=/MyLDP/algol/lex-yacc-howto.html>

Принципы работы flex

Принцип работы Flex достаточно прост: на вход ему подается текстовый файл, содержащий описания нужных лексем в терминах регулярных выражений, а на выходе получается файл с текстом исходной программы сканера на языке программирования C. Далее текст программы нужно скомпилировать компилятором C. После этого полученный исполняемый файл можно использовать, направляя ему на вход поток лексем.



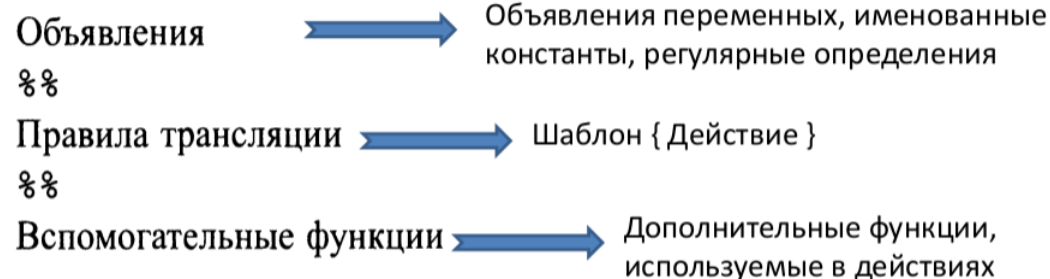
Структура программы flex

Строки, содержащие в первой позиции пробел или заключенные в скобки % и % просто копируются в С-программу, являющуюся результатом работы LEX'a. Они могут содержать описания переменных, реализацию функций и т.д.

FLEX-программа состоит из трех секций, отделяемых друг от друга символом %%.

Первая из них содержит определения макросимволов. Каждое описание начинается с первой позиции строки и имеет вид "имя_макросимвола строка". Вместо последовательности имя_макросимвола, встреченной после его определения, будет подставлена соответствующая ему строка. Макросимволы полезны для задания сложных выражений, например: letter [a-zA-Z_#] digit [0-9] ident {letter}({letter}|{digit})*

Вторая секция содержит правила, которые имеют вид "регулярное выражение { действие }". Действие представляют собой последовательность операторов языка С, выполняемые при успешном распознавании регулярного выражения. Выражение записывается с начала строки. Фигурная скобка, начинающая действие, должна находиться в той же строке, что и регулярное выражение, действие может продолжаться на нескольких строках. Текст, содержащийся в последней секции, просто копируется в выходной файл.



Метасимволы регулярных выражений, используемые в flex

.	Любой символ, кроме \n
\n	Новая строка
*	Повторение 0 или более раз
+	Повторение 1 или более раз
?	Повторение 0 или 1 раз
^	Соответствует началу строки, если указан в начале регулярного выражения. В квадратных скобках соответствует исключению символа из диапазона
\$	Конец строки
a b	a или b
(ab)+	Группировка (повторение последовательности ab 1 или более раз)
"a+b"	Строка a+b
[]	Класс (диапазон) символов
{ }	A{1,3} соответствует повторению буквы A от одного до трех раз 0{5} соответствует 00000 {reg_def} – обращение к регулярному определению

Примеры метасимволов регулярных выражений, используемые в flex

abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbcb abcbcbcb ...
a(bc)?	a abc
[abc]	a или b или c
[a-z]	любая буква в диапазоне от a до z
[a\ -z]	a или – или z
[-az]	– или a или z
[A-Za-z0-9]+	любая буква или цифра, повторенная 1 или более раз
[\t\n]+	один или несколько пробелов, знаков табуляции или перехода на новую строку
[^ab]	кроме a и b
[a^b]	a или ^ или b
[a b]	a или или b
a b	a или b

Функции, макросы, predefined переменные flex

int yylex(void)	вызов лексического анализатора, возвращает токен
char *yytext	распознанная строка
yylen	длина распознанной строки
yyval	значение, ассоциирующееся с токеном
FILE *yyout	выходной файл (по умолчанию stdout)
FILE *yyin	входной файл (по умолчанию stdin)
ECHO	вывод распознанной строки в yyout

Порядок работы flex

Файл, полученный в результате работы FLEX, является C-программой, которая содержит таблицы, описывающие построенный конечный автомат, функции, реализующие интерпретатор автомата, описания структур данных, используемые интерпретатором, и пользовательские программы. Для запуска автомата надо вызвать функцию `yulex()`, содержащей все действия, описанные в секции правил FLEX-программы. Она, в свою очередь вызывает функцию `yulook`, реализующую собственно конечный автомат. В процессе работы автомат читает символ за символом из потока `yuin`, назначенного по умолчанию на стандартный ввод. После успешного распознавания одного из выражений происходит возврат в функцию `yulex` и выполнение соответствующего ему действия. При этом, переменная `char yutext` содержит терминированную нулем строку считанных символов, соответствующую данному регулярному выражению. Переменная `int yuleng` содержит длину этой строки.

Порядок работы flex

Если возникает неоднозначность при выделении цепочки символов, то она разрешается стандартным образом. Предпочтение отдается правилу, порождающему наиболее длинную цепочку. Если два правила соответствуют одной и той же цепочке, то применяется правило, описанное раньше в секции правил. Например, если заданы следующие правила

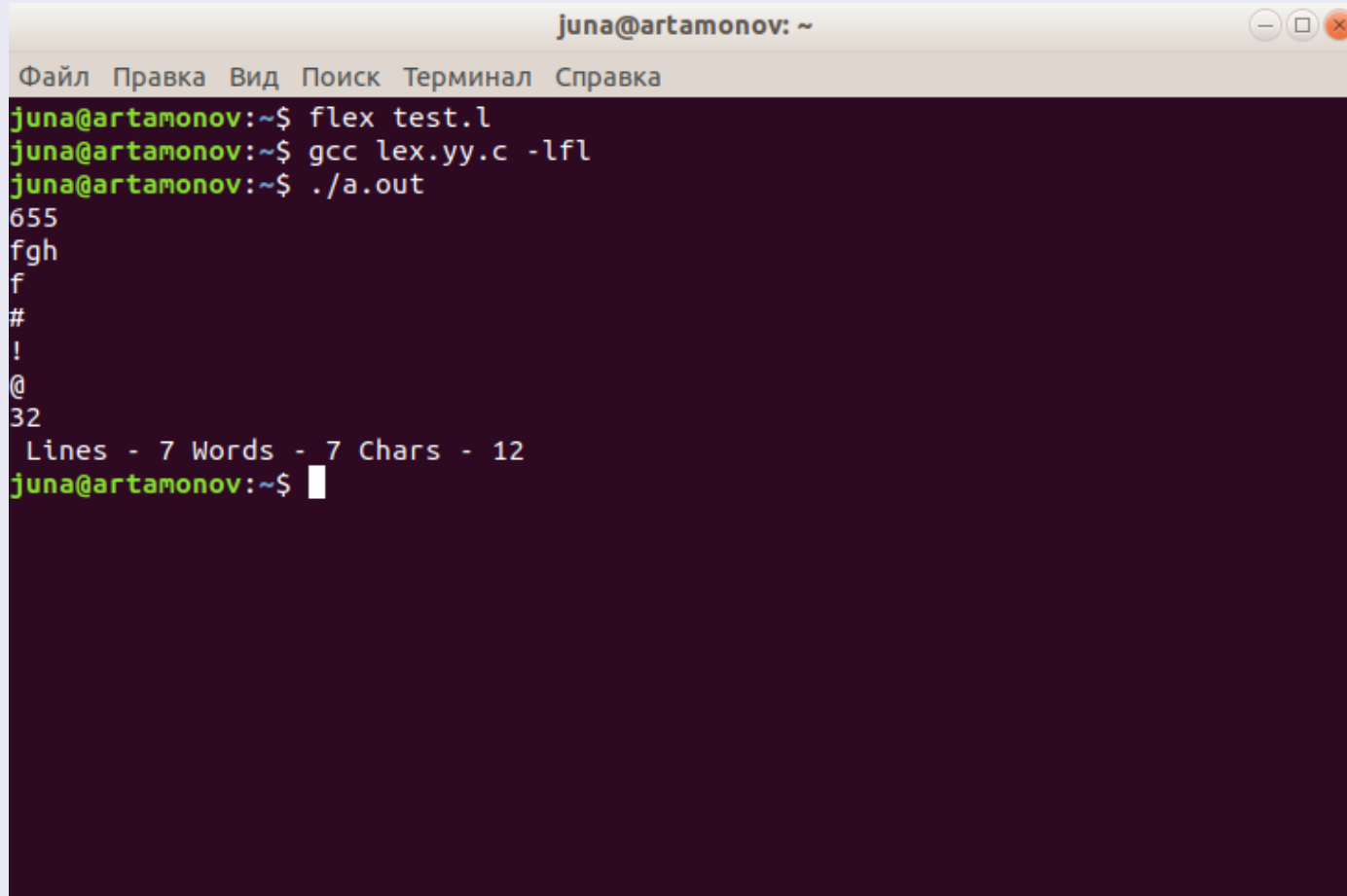
```
[0-9]+ { printf( "Без знака <%s>", yytext ); }  
(\+|\-)?[0-9]+ { printf( "Целое <%s>", yytext ); }  
(\+|\-)?[0-9]+ "." [0-9]* { printf( "Плавающее <%s>", yytext ); }
```

то цепочка «123», удовлетворяющая первым двум правилам будет разобрана по первому, а цепочка 3.14, первый символ которой удовлетворяет первым двум правилам, - по третьему правилу.

Пример программы на flex

```
%{  
int l=0, w=0, c=0;  
%}  
NODELIM [^" "\t\n]  
%%  
{NODELIM}+ { w++; c+=yyleng; /* Слово */ }  
\n { l++; /* Перевод строки */ }  
. { c++; /* Остальные символы */ }  
%%  
int main() {  
yylex();  
printf( " Lines – %d Words – %d Chars – %d\n", l, w, c );  
return 0;  
}
```

Пример программы на flex



```
juna@artamonov: ~  
Файл Правка Вид Поиск Терминал Справка  
juna@artamonov:~$ flex test.l  
juna@artamonov:~$ gcc lex.yy.c -lfl  
juna@artamonov:~$ ./a.out  
655  
fgh  
f  
#  
!  
@  
32  
Lines - 7 Words - 7 Chars - 12  
juna@artamonov:~$
```

Задание на flex

Используя утилиту `ps` с ключем `uax` получить ID процесса, который потребляет больше всего ресурсов процессора.

```
juna@artamonov: ~  
Файл Правка Вид Поиск Терминал Справка  
juna@artamonov:~$ ps uax  
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND  
root         1  0.0  0.1 225536  9096 ?        Ss   18:44   0:14 /sbin/init  
root         2  0.0  0.0      0      0 ?        S    18:44   0:00 [kthreadd]  
root         4  0.0  0.0      0      0 ?        I<   18:44   0:00 [kworker/0:0H]  
root         6  0.0  0.0      0      0 ?        I<   18:44   0:00 [mm_percpu_wq]  
root         7  0.0  0.0      0      0 ?        S    18:44   0:02 [ksoftirqd/0]  
root         8  0.0  0.0      0      0 ?        I    18:44   0:14 [rcu_sched]  
root         9  0.0  0.0      0      0 ?        I    18:44   0:00 [rcu_bh]  
root        10  0.0  0.0      0      0 ?        S    18:44   0:00 [migration/0]  
root        11  0.0  0.0      0      0 ?        S    18:44   0:00 [watchdog/0]  
root        12  0.0  0.0      0      0 ?        S    18:44   0:00 [cpuhp/0]  
root        13  0.0  0.0      0      0 ?        S    18:44   0:00 [cpuhp/1]  
root        14  0.0  0.0      0      0 ?        S    18:44   0:00 [watchdog/1]  
root        15  0.0  0.0      0      0 ?        S    18:44   0:00 [migration/1]  
root        16  0.0  0.0      0      0 ?        S    18:44   0:00 [ksoftirqd/1]  
root        18  0.0  0.0      0      0 ?        I<   18:44   0:00 [kworker/1:0H]  
root        19  0.0  0.0      0      0 ?        S    18:44   0:00 [cpuhp/2]  
root        20  0.0  0.0      0      0 ?        S    18:44   0:00 [watchdog/2]  
root        21  0.0  0.0      0      0 ?        S    18:44   0:00 [migration/2]  
root        22  0.0  0.0      0      0 ?        S    18:44   0:00 [ksoftirqd/2]  
root        24  0.0  0.0      0      0 ?        I<   18:44   0:00 [kworker/2:0H]  
root        25  0.0  0.0      0      0 ?        S    18:44   0:00 [cpuhp/3]  
root        26  0.0  0.0      0      0 ?        S    18:44   0:00 [watchdog/3]
```

```
ps uax|grep -Eo "[0-9]+ *[0-9]+[0-9]+"
```

Задание на flex

```
%{
char pid[10], cpu[10];
int pid_n;
float cpu_max=0;
int pr_pid, j, k;
%}

%%
[0-9]+" "+[0-9]+"."[0-9]+ { pr_pid=1; j=0; k=0;
for (int i=0; i<yyleng; i++)
{
    if (pr_pid && (yytext[i]!=' ')) {pid[k]=yytext[i]; k++;}
    if (yytext[i]==' ') pr_pid=0;
    if (!(pr_pid)&& (yytext[i]!=' ')) {cpu[j]=yytext[i]; j++;}
}
```

Задание на flex

```
pid[k]='\0'; cpu[j]='\0';
if ( atof(cpu)>cpu_max){
    cpu_max=atof(cpu); pid_n=atoi(pid);
}
}

%%
int main() {
yylex();
printf( " PID = %d CPU=%f", pid_n,cpu_max);
return 0; }
```

Задание

Построить лексический анализатор, распознающий целочисленные константы с использованием flex.

Задание

С использованием flex построить лексический анализатор, распознающий арифметические выражения вида $a + b$, $a - b$, $a * b$, a / b и выполняющий соответствующие арифметические вычисления.