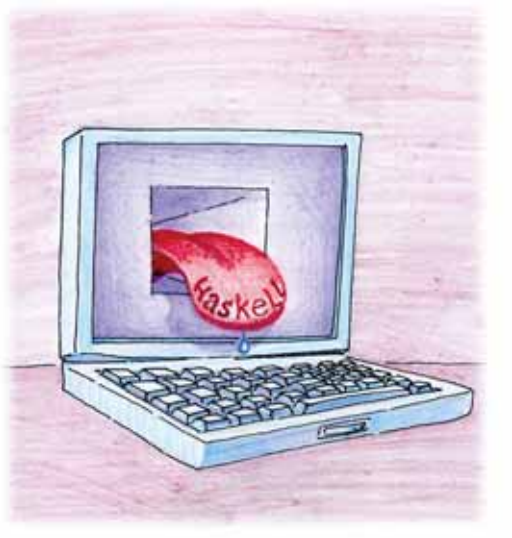
Душкин Роман Викторович
Специалист по функциональному программированию и функциональным парсерам. Автор книги «Функциональное программирование на языке Haskell».

Алгебраические типы данных в языке Haskell

Настоящая статья продолжает собой цикл научно-популярных статей, направленных на предоставление всем желающим необходимого минимума в овладении парадигмой функционального программирования на примере языка Haskell. Пришло время более детально изучить такой немаловажный аспект программирования, как создание пользовательских типов данных, которые в языке Haskell носят наименование «алгебраических типов данных».

Введение

В процессе создания программного обеспечения практически всегда встаёт задача создания собственных типов данных, т. к. обычно базовый набор первичных типов слишком узок. Зачастую этот набор состоит только из трёх или четырёх типов, отражающих основные множества чисел в математике, а также символы, которые могут быть выведены на экран или записаны в файл.



Однако современное высокоуровневое программирование подразумевает, что сама программа должна обладать высокой степенью абстракции данных, что позволило бы не только грамотно описывать проблемную область решаемой задачи, но и повторно использовать созданные определения. Именно поэтому каждый развитый язык программирования предоставля-

ет программисту обширное множество инструментов для определения новых типов. При этом зачастую сами новые типы можно создавать как на основе уже имеющихся, так и непосредственно, что называется, «с нуля».

Не обошёл стороной эту проблему и язык Haskell — современный функциональный язык программирования. В нём определена развитая система типизации, которая позволяет не только определять типы используемых в программе объектов автоматически, но и создавать собственные типы любой сложности, в том числе и с использованием параметрического полиморфизма[1]. Этим язык Haskell отличается от многих императивных языков программирования, в которых полиморфизм в лучшем случае представлен обычной перегрузкой имён объектов (обычно наименований функций или процедур).

Все определения, приведённые в данной статье, можно непосредственно проверить в интерпретаторе языка Haskell — HUGS 98, который можно бесплатно получить на официальном сайте изучаемого языка в интернете (http://www.haskell.org/hugs/). Все представленные определения протестированы в этом интерпретаторе, поэтому их правильность гарантируется. Использование других трансляторов языка Haskell (например, компиляторов GHC или NHC) также возможно, однако для их использования, возможно, придётся вносить в определения функций и типов незначительные изменения.

Для понимания материала, изложенного в статье, необходимо обладать базовыми познаниями в информатике, понимать, что такое типы данных, а также знать основной синтаксис языка Haskell. Для изучения последнего достаточно прочитать предыдущие статьи в журнале «Потенциал» на эту тему, либо обратиться к лекциям по функциональному программированию, которые можно найти в интернете по адресу: http://roman-dushkin.narod.ru/fp.html.

Автор будет признателен любым отзывам и комментариям к статье, которые помогут сделать дальнейшие публикации более интересными и полезными для читателей. Отзывы можно посылать по адресу электронной почты darkus.14@gmail.com.

1. Простые перечисления

Для определения пользовательских типов данных в языке Haskell используется служебное слово **data**, которое должно стоять на самом верхнем уровне определений в модуле. Это служебное слово определяет т.н. «алгебраический тип данных», выделяя для него определённое наименование, а также набор возможных конструкторов, т. е. функций, которые возвращают объект определяемого типа. Все возможные конструкторы должны быть разделены символом (|) — «вертикальная черта». В каждом алгебраическом типе данных должен быть по крайней мере один конструктор.

[1]Понимание параметрического полиморфизма будет предложено в одном из следующих разделов этой статьи.

Например, при помощи алгебраического типа данных в стандартном модуле Prelude определён тип для представления булевских значений истинности[1]. Это сделано так:

```
data Bool = True
          | False
```

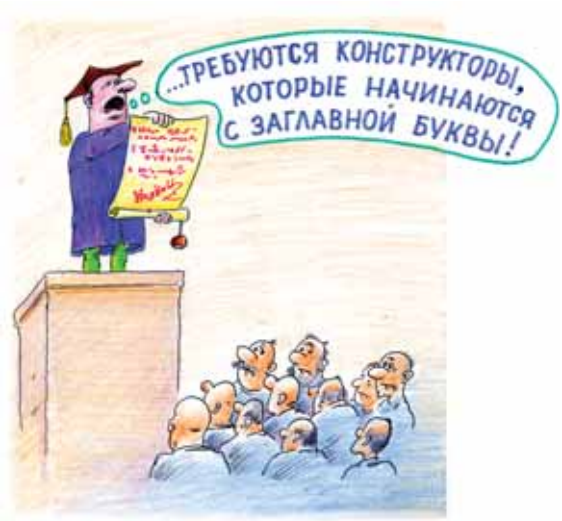
Надо особо отметить, что по соглашению о наименовании объектов в языке Haskell все наименования типов, а также их конструкторы, должны начинаться с заглавной буквы. Это можно видеть в представленном определении. Такое соглашение в том числе помогает чётко разделять встречающиеся в исходном коде объекты. Первая буква идентификатора свидетельствует о роде объекта.

Можно видеть, что представленное определение булевского типа в языке Haskell полностью совпадает с математическим определением этого типа, а именно:

```
𝔹 = <true, false >
```

Здесь можно дополнительно убедиться в необычайной выразительности языка Haskell — его создатели, как обычно, стремились сделать его синтаксис наиболее приближённым к математической нотации.

Представленное определение является собой т.н. простое перечисление, т.е. примерно то же самое, что в языках C и C++ обозначается ключевым словом **enum**. Простое перечисление представляет собой неупорядоченный набор наименований объектов, принадлежащих определённому множеству, составляющему описываемый тип. Другими словами, каждое простое перечисление определяет один или более наименований объектов, которые могут принадлежать типу. При этом надо отметить, что одно из таких наименований может совпадать с наименованием типа (они находятся в разных пространствах имён).



Другим примером, который очень чётко даёт представление о том, что такое простое перечисление, является описание типа для представления различных цветов. При этом каждый цвет можно описать его названием на естественном языке (в данном случае, на английском, т. к. синтаксис языка Haskell пока не разрешает использование символов с кодом выше 127). Это можно сделать примерно так (естественно, надо принимать во внимание,что подобным образом можно описать любые наборы цветов, которые будут отличаться друг от друга как количеством,так и наименованием конкретных элементов):

[1]Можно отметить, что во многих языках программирования булевский тип является одним из примитивных типов, предоставляемых самим языком. Например, в языке Pascal — это тип Boolean, а в языках C, C++ и многих им подобных —тип bool.

```
data Color = Black -- Чёрный
           | Blue -- Синий
           | Brown -- Коричневый
           | Cyan -- Голубой
           | Gray -- Серый
           | Green -- Зелёный
           | Magenta -- Розовый
           | Orange -- Оранжевый
           | Red -- Красный
           | White -- Белый
           | Yellow -- Жёлтый
```

Всё, что стоит после двух символов «дефис» (**--**), является комментарием, который длится до конца строки. Также надо отметить, что в языке Haskell во многих местах действует правило двумерного синтаксиса, когда определения объектов можно располагать на нескольких строчках кода, выравнивая их по одной колонке. Здесь виден именно такой способ записи, хотя это и не обязательно, т. к. можно записывать всё в одну строку (тогда, естественно, комментарии так ставить нельзя).

При помощи простых перечислений можно легко пояснить, что такое сопоставление с образцом. Можно вернуться к определению булевского типа и рассмотреть различные операции над объектами этого типа. Из формальной науки известно множество таких операций — отрицание, конъюнкция, дизъюнкция, исключающее «или», эквивалентность, импликация и многие другие. Вот, к примеру, определение функции для вычисления отрицания:

```
not:: Bool -> Bool
not True = False
not False = True
```

В этом определении, состоящем из трёх строк, первая строка представляет собой ограничение, накладываемое на тип функции **not**. Эта запись говорит интерпретатору, что функция **not** принимает на вход один аргумент типа **Bool** и возвращает значение этого же типа. Вторая строка определяет результат функции **not**, если ей на вход подано значение **True**, а третья — для значения **False** соответственно.

Образцы представляют собой наборы входных значений в определениях функций, с которыми сравниваются фактические параметры, поданные на вход функций. Наборы образцов в идеальном случае должны покрывать всё множество возможных входных параметров. Сопоставление с образцом, т.е. выбор конкретной строки в наборе определений, происходит просто — интерпретатор движется сверху вниз по определениям и пытается сопоставить фактический входной параметр с тем, что представлен в образце. Как только такое сопоставление происходит успешно, выбирается эта строка и вычисляется результат. Так, в приведённом выше определении вторая строка выбирается в случае, если на вход функции **not** подано значение **True** и т. д.

Естественно, что образцы могут быть намного сложнее, ибо для бесконечных множеств (типов) сложно перечислить все возможные варианты значений. Поэтому часто используется т.н. маска подстановки, обозначаемая символом (_) «подчёркивание». Этот символ обозначает любое значение любого типа. Например, при помощи него можно определить операцию конъюнкции на булевских значениях:

```
(&&) :: Bool -> Bool -> Bool
True && True =  True
_    && _    =  False
```

В данном определении показано, что значение конъюнкции равно **True** только в случае, если оба входных аргумента равны **True**, а в противном случае (и это уже не важно, какие конкретно там значения в каких комбинациях) значением конъюнкции будет **False**. Примерно такое же определение можно привести и для дизъюнкции:

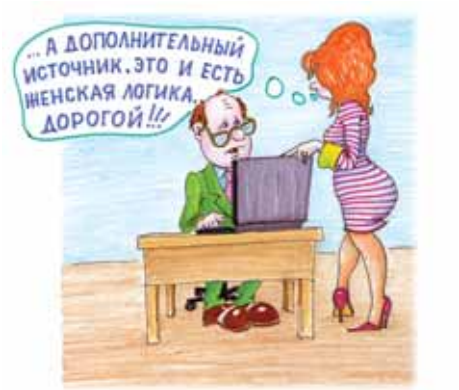
```
(||)  :: Bool  -> Bool -> Bool
False || False =  False
 _    ||  _    =  True
```

Как видно, всё довольно просто. Читателю предлагается самостоятельно разработать функции для вычисления исключающего «или» (наименование — (|+|)), эквивалентности (наименование — (===)) и импликации (наименование — (==>)). Определения этих функций можно выразить как при помощи уже имеющихся, так и создать на основе механизма сопоставления с образцом (что более предпочтительно ввиду возможности дополнительного закрепления пройденного материала).

Остаётся предложить вдумчивому читателю для изучения ещё один тип данных, представляющий собой отображение значений троичной логики. Эта логика используется в тех случаях, когда значением истинности могут выступать не только полностью определённые значения «**ИСТИНА**» и «**ЛОЖЬ**», но и третье значение — «**НЕ ОПРЕДЕЛЕНО**». Определение этого типа может выглядеть так:

```
data Ternary = TTrue
             | TUndefined
             | TFalse
```

В конструкторах этого типа первым поставлен символ **T** в целях исключения конфликта с конструкторами типа **Bool**. Опять же читателю предлагается самостоятельно создать операции для работы с этим типом данных. При необходимости надо обратиться к дополнительным источникам для понимания того, как работает троичная логика.



2. Параметризация

Однако было бы странным полагать, что все возможности языка Haskell для определения типов ограничивались бы простыми перечислениями. Конечно, они предоставляют программисту хороший инструмент для работы, но в природе существуют более сложные объекты, чем перечислимые множества. Например, для представления точек на двумерной плоскости используется две координаты, представляющие собой действительные числа, а в трёхмерном пространстве — соответственно три действительных числа.

Для решения этой и многих подобных задач имеется возможность параметризации алгебраических типов данных. Такая параметризация делается при помощи перечисления после имён конструкторов типа дополнительных типов, от которых зависит определяемый тип.

Например, для представления двумерных координат можно использовать такое определение:

```
data Point2D = Point2D Float
    Float
```

Это определение обозначает, что для получения объекта типа **Point2D** в его конструктор, который называется также, необходимо передать в качестве параметров два действительных числа (тип **Float**, представляющий действительные числа одинарной точности, является встроенным примитивным типом в языке Haskell).

Точно также можно определить и тип данных для представления трёхмерных точек:

```
data Point3D = Point3D Float
Float Float
```

Вполне естественно, что механизм сопоставления с образцами в данном случае работает абсолютно так же, как и в случае простых перечислений. В качестве образцов необходимо указывать конструкторы алгебраических типов данных и множество возможных значений, которыми параметризуется этот тип. Например, функция для сдвига точки в двумерном пространстве на определённое смещение по обеим осям координат может быть определена так:

```
shift2D:: Point2D -> Float ->
    Float -> Point2D
shift2D (Point2D x y) dx dy
    = Point2D (x + dx) (y + dy)
```

Точно так же может быть определена функция и для сдвига в трёхмерном пространстве, однако в этом случае будут две практически одинаковых функции. Ничто не мешает определить единственный тип для представления точек как на плоскости, так и в трёхмерном пространстве. Это можно сделать, объединив два вышеприведённых определения:

```
data Point = Point2D Float Float
    | Point3D Float Float Float
```

Тогда для такого типа данных определение функции для сдвига точки будет выглядеть так:

```
shift:: Point -> Point -> Point
shift (Point2D x y) (Point2D dx
dy)
= Point2D (x + dx) (y + dy)
shift (Point3D x y z) (Point3D
    dx dy dz)
= Point3D (x + dx) (y + dy) (z+
    + dz)
```

Однако представленное определение типа не так красиво, как кажется на первый взгляд. Оно не универсально, а ограничивается только точками на плоскости и в трёхмерном пространстве. Что делать, если вдруг понадобится решить какую-либо задачу в четырёхмерном пространстве? А в n-мерном? В самом деле, не перечислять же в списке конструкторов все возможные размерности. В этом случае на помощь приходит список, чья длина заранее не определена. Поэтому сам тип для представления точек можно определить так:

```
data Point = Point [Float]
```

Такое определение обозначает, что тип **Point** параметризуется списком действительных чисел, а длина списка определяет размерность описываемого этим типом пространства. В этом решении, однако, остаётся одна возможность для множества логических ошибок, которые могут прокрасться в программу — у списков нет заранее указанной длины, поэтому в функциях, которые будут работать с такими точками, необходимо очень внимательно следить за

тем, чтобы размерности точек совпадали. Это можно сделать либо сравнивая длины получаемых на входе списков координат и выдавая сообщение об ошибке при их несовпадении, либо выбирая минимальную длину и отсекая все координаты, лежащие далее такой минимальной длины. По этому пути решено пойти при создании нового определения функции **shift**:

```
shift:: Point -> Point -> Point
shift (Point []) _ = Point []
shift _ (Point []) = Point []
shift (Point (x:xs)) (Point (-
dx:dxs))
= Point ((x + dx):others)
where Point others
= shift (Point xs) (Point dxs)
```

Это определение уже не так очевидно и выразительно, однако работает на точках любой размерности, даже нулевой (можно попробовать осуществить вызов **shift (Point []) (Point [])** — в результате будет выдан результат **Point []**).

Вполне естественно, что ничто не ограничивает программиста использовать в одном определении алгебраического типа данных как конструкторов без параметров (именно такие конструкторы используются в простых перечислениях), так и конструкторов с параметрами. Так, к примеру, можно заново определить тип для представления цветов, добавив дополнительный конструктор, принимающий на вход три целочисленных аргумента и возвращающий цвет в формате **RGB**[1]. Тогда определение типа **Color** будет выглядетьследующим образом:

```
data Color = Black --Чёрный
| Blue             --Синий
| Brown            --Коричневый
| Cyan             --Голубой
| Gray             --Серый
| Green            --Зелёный
| Magenta          --Розовый
| Orange           --Оранжевый
| Red              --Красный
| White            --Белый
| Yellow           --Жёлтый
| RGB Int Int Int  --RGB
```

Остаётся отметить, что в качестве типов, которыми параметризуются создаваемые алгебраические типы, могут быть как примитивные типы, встроенные в интерпретатор, так и типы, созданные пользователем.

3. Параметрический полиморфизм

Но даже такой мощный механизм, как параметризация типов, не может обеспечить всех потребностей в создании новых типов данных. Ведь иногда имеется необходимость создать контейнерный тип[2], внутри которого

[1]RGB — одна из цветовых систем, используемая для кодирования цвета в компьютерах. Представляет собой три целочисленные координаты в цветовом пространстве, соответствующие насыщенности красного (Red), зелёного (Green) и синего (Blue) цветов.

[2]Контейнерным называется такой тип данных, который содержит внутри себя объекты иных типов, в том числе и другие объекты контейнерных типов, иначе называемые просто «контейнерами». Например, список — это контейнерный тип данных.

могут содержаться объекты любого типа. Перечислять все такие типы в конструкторах нецелесообразно, т.к. самих типов может быть огромное число, да и невозможно заранее предусмотреть, объекты каких типов захочет кто-либо положить в контейнер. Здесь на помощь приходит параметрический полиморфизм, иногда называемый истинным полиморфизмом данных.

Параметрический полиморфизм в применении к алгебраическим типам данных в языке Haskell заключается в том, что в конструкторах типов могут употребляться так называемые параметрические переменные, которые обычно обозначаются строчными буквами из начала латинского алфавита. Такие переменные могут обозначать любой тип, а самих переменных может быть столько, сколько необходимо программисту. Единственная особенность заключается в том, что все используемые параметрические переменные должны быть перечислены после наименования типа.

В качестве примера можно привести тип для представления бинарных деревьев, в вершинах которых могут находиться метки любого типа. Этот тип меток должен быть одинаков для всех вершин дерева, но при помощи параметрического полиморфизма, имея единственный конструктор для бинарного дерева, можно создавать такие деревья с метками различных типов в вершинах. Определение такого типа может выглядеть так:

```
data BTree a = Empty | Node (a,
BTree a, BTree a)
```

Первый конструктор **Empty** определяет пустое дерево. Второй конструктор **Node** определяет вершину, на которой стоит пометка типа **a** и которая имеет два дочерних поддерева с метками того же типа. Как уже было сказано, на месте параметрической переменной **a** может стоять любой тип данных. Например, если имеется необходимость в существовании деревьев, метки в вершинах которых представляют собой целые числа, то тип таких деревьев должен быть **BTree Int** и т.п.

Другими словами, параметрический полиморфизм позволяет определять наиболее общие типы данных, в которых конкретные типы не определены, но имеются параметрические переменные типов, которые по соглашению об именовании объектов в языке Haskell должны начинаться с маленькой буквы. Это — достаточно мощный механизм, который позволяет достигать наибольшей степени абстракции при определении типов.

Естественно, что параметризованные подобным образом алгебраические типы всё также могут участвовать в процессе сопоставления с образцами, который используется при вычислении значений функций. Образцы в данном случае ничем не отличаются от рассмотренных ранее, а в типах функций появляются такие же параметрические переменные типов. Для изучения этого аспекта можно рассмотреть несколько функций, работающих с бинарными деревьями.

Первая функция называется **depth** — она вычисляет максимальную глубину дерева, т. е. наибольшую длину из всех путей, которые можно проложить от корневой вершины дерева к его конечным листьевым вер-

шинам. Определение этой функции выглядит так:

```
depth:: (Num a, Ord a) => BTree
   b -> a
depth Empty = 0
depth (Node (_, left, right))
   = 1 + max (depth left)
(depth right)
```

Вторая функция подсчитывает общее количество вершин в заданном дереве. Её определение практически такое же, как и у функции **depth** (по крайней мере, оно построено на тех же самых принципах обхода дерева):

```
count:: Num a => BTree b -> a
count Empty = 0
count (Node (_, left, right))
 = 1 + count left + count right
```

Третья функция возвращает список меток в вершинах дерева при обходе этого дерева по схеме «левый — корень — правый». Её определение уже более интересное:

```
flatten:: BTree a -> [a]
flatten Empty = []
flatten (Node (x, left, right))
   = flatten left ++ [x] ++
   flatten                right
```

Наконец, самой интересной является функция, которая принимает на вход другую функцию и некоторое бинарное дерево, а возвращает другое бинарное дерево, ко всем меткам в вершинах которого применена заданная в качестве первого аргумента функция. Это — аналог функции **map** для списков:

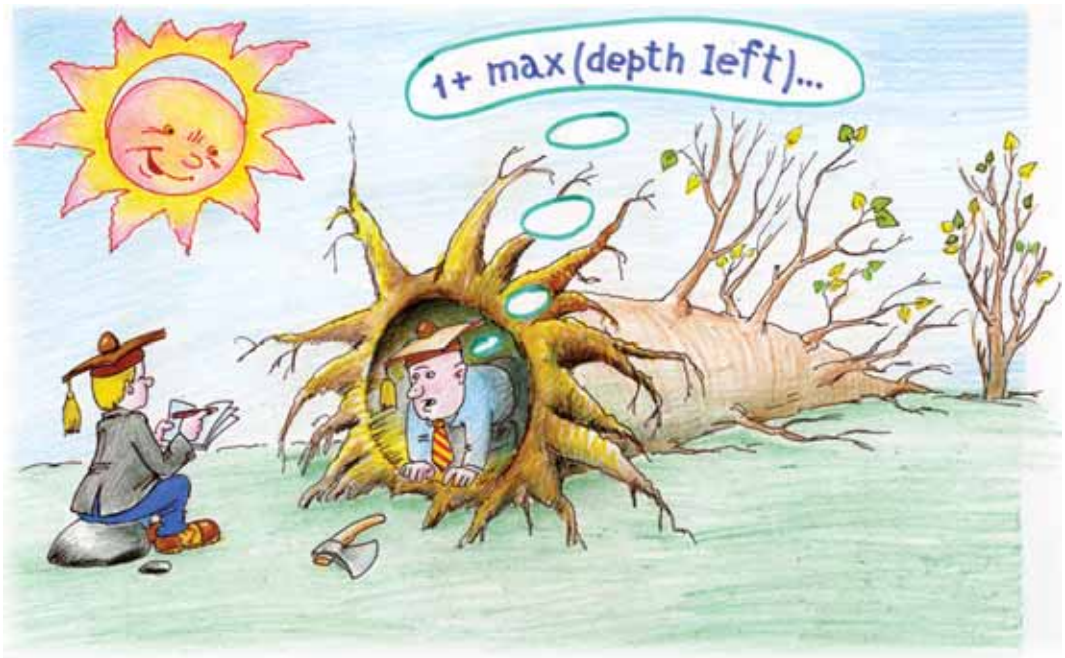
```
mapBTree:: (a -> b) > BTree a ->
      BTree b
mapBTree _ Empty = Empty
mapBTree f (Node (x, left,
right))
   = Node (f x,
      (mapBTree f left),
      (mapBTree f right))
```

Как видно, функция **mapBTree** является функцией высшего порядка, которая принимает на вход другую функцию в качестве своего первого аргумента. Это позволяет решать при помощи такой функции достаточно широкий набор задач, связанных с преобразованием деревьев.

Таким образом, параметрический полиморфизм является весьма мощным механизмом, который при использовании в определениях типов данных позволяет решить практически любую задачу по описанию сущностей любых проблемных областей. Понимание того, как работают полиморфные типы, помогает более полноценно использовать все механизмы, которые предоставляет программисту язык Haskell. Поэтому при изучении этого мощного языка необходимо достаточное внимание уделить именно этой теме.

Заключение

Умение грамотно описывать проблемную область исследуемой задачи — один из главных аспектов технологии программирования, какая бы парадигма и стиль при этом ни использовались. Программист, который может построить оптимальное определение типов для объектов и связей между ними, т.е. для тех сущностей, которыми оперирует программа, дорогого стоит. Поэтому при изучении того или иного языка программирова-



ния необходимо очень внимательно подходить к пониманию способов определения типов.

В одной из следующих статей будет показано слияние функциональной и объектно-ориентированной парадигм программирования в языке Haskell, а также способы написания объектно-ориентированных программ на этом замечательном языке программирования. Кроме того, будут показаны дополнительные аспекты работы с алгебраическими типами данных в языке Haskell. Все перечисленные в этой статье определения типов и функций сведены в отдельные модули, которые каждый желающий может получить, послав электронное письмо с соответствующим запросом (пожалуйста, указывайте в запросе наименование статьи, для которой необходимо прислать определения) на электронный адрес автора статьи — darkus.14@gmail.com.

Кстати, свои решения задач, предложенных в этой статье, посылайте и нам, в новый раздел «Суд Линча» сайта журнала, http://potential.org.ru.