Ворожцов Артём Викторович
Кандидат физико-математических наук,
преподаватель кафедры информатики
Московского физико-технического института (МФТИ),
тренер сборной команды МФТИ по программированию.

Перевод из постфиксной в инфиксную нотацию

Постфиксная нотация – это запись выражения по следующему правилу: сначала перечисляются аргументы оператора, а затем сам оператор. Например, выражение `"a-(b+c)*c"` запишется как `"(a ((b c +) c *) -)"`. Конечно, это выглядит несколько непривычно. Но заметим, что эта нотация – одна из самых привычных нотаций для компьютера. Выражения, записанные в таком виде, легко разобрать алгоритмически и вычислить. Одно из интересных наблюдений заключается в том, что нет необходимости использовать скобки. В частности, если в выражении `"(a ((b c +) c *) -)"` удалить все скобки, то получится выражение `"a b c + c * -"`, которое по-прежнему интерпретируется однозначно. В данной статье мы подробно рассмотрим постфиксную нотацию и научимся вычислять выражения, записанные в этой нотации, а также сформулируем исследовательскую задачу о минимизации числа скобок в выражении, записанном в привычном нам виде (в инфиксной нотации).

* * *

Всего имеются три различные нотации – *префиксная*, *инфиксная* и *постфиксная*. Все три уже встречались вам в жизни, но вы на это не обращали внимание.

Например, вот выражение, записанное в инфиксной нотации:

`(1+2)*3 - 4/5.`

В тетрадках по математике и физике вы используете именно эту нотацию. *Знак арифметического действия в инфиксной нотации ставится между двумя выражениями (операндами),* с которыми нужно это действие сделать.

Иногда знак действия или слово, обозначающее действие, ставится перед операндами:

`НОД(max(a,b), c).`

Для записи функции НОД (наибольший общий делитель двух чисел) и функции **max** (максимум из двух чисел) в программировании принята префиксная нотация. Собственно, в программах, когда операция есть функция, то используется *префиксная нотация – сначала пишется имя функции (знака действия, имя оператора), а потом её аргументы (операнды).*

Постфиксная нотация в программах на языке Pascal вам, возможно, не встречалась. Она считается неудобной для человеческого мышления. *В постфиксной нотации сначала пишутся аргументы (операнды), а затем имя функции (знака действия, имя оператора).*

Давайте рассмотрим фразу:

«Учитель сказал, что Вова решил задачу №13».

Глаголы здесь могут играть роль *операторов*, а остальные слова – роль *операндов*. Давайте операторы выделим жирным цветом и расставим скобки:

Инфиксная: «(Учитель **сказал** (Вова **решил** задачу №13))».

Схема этого выражения такая:

«(У **с** (В **р** З))».

Давайте перепишем её в префиксной форме:

«**сказал** (Учитель, **решил**(Вова, задачу №13))».

И, наконец, в постфиксной форме:

«Учитель Вова задачу №13 **решил сказал**».

Последнее звучит совсем не по-русски, а компьютеру почему-то «нравится». Чем же постфиксная нотация так хороша?

Оказывается, алгоритм обработки выражения в постфиксной нотации очень прост. Рассмотрим выражение **"(1+2)*4 – 5/6"** и её запись в постфиксной нотации:

```
1 2 + 4 * 5  6 / –
```

Для его вычисления мы будем использовать стопку листков (соответствующую абстрактную структуру данных программисты называют *стеком*). Изначально эта стопка пуста. Начинаем считывать наше выражение слева направо. Сначала идёт единица. Запишем её на чистом листке и положим на стол. Это будет первый листок в нашей стопке. Затем мы

считаем число 2. Снова возьмём чистый листок, запишем на нём 2 и положим в стопку сверху, на листок с единицей (чистые листки мы берём из тумбочки под столом). Мы так будем поступать каждый раз, когда считывается число – будем брать чистый листок, писать на нём это число и класть сверху в стопку.

Если же мы считали не число, а знак некоторого действия A (+, −, * или /), то будем поступать следующим образом. Возьмём из стопки два верхних листка с числами. Применим к этим двум числам действие A, а результат запишем на новом чистом листке и положим его сверху на стопку. Вот и всё!

Давайте выполним этот алгоритм для входной строки

"1 2 + 4 * 5 6 / −".

Содержимое листков в стопке снизу вверх будем записывать в строку слева направо:

Таблица 1

токен	состояние стопки			
1	1			
2	1	2		
+	3			
4	3	4		
*	12			
5	12	5		
6	12	5	6	
/	12	5/6		
−	11 1/6			

Итак, если приходит число, кладём его в стопку сверху, а приходит знак операции – делаем эту операцию с двумя числами, которые забираем сверху из стопки, и результат операции кладём в стопку.

Элементы, на которые разбивается выражение, называются токенами. *Токен* – это последовательность символов, обозначающая некоторую сущность, запись которой мы не хотели бы делить на части. Например, натуральное число можно было бы разделить на цифры. Но мы не хотим этим заниматься и считаем, что запись натурального числа – это токен. При анализе выражения удобно считать, что на вход нам поступает не последовательность символов, а готовая последовательность токенов. Программа, которая разбивает входную последовательность символов на последовательность токенов называется *токенайзер*.

Постфиксный калькулятор

Давайте напишем простейшую программу на языке Ruby, которая осуществляет вычисление выражения в постфиксной нотации.

```
def post_to_inf(input)
  stack = []
  input.split.each do |token|
    puts stack.inspect
    case token
    when '+', '-', '*', '/'
      a2 = stack.pop
      a1 = stack.pop
      stack.push( a1.send(token, a2) )
```

```
    else
      stack.push(token.to_f)
    end
  end
  stack x = ary.pop
end

input = '1 2 + 4 * 5 6 / -'

puts post_to_inf(input).inspect
```

Данная программа выводит следующий текст:

```
[]
[1.0]
[1.0, 2.0]
[3.0]
[3.0, 4.0]
[12.0]
[12.0, 5.0]
[12.0, 5.0, 6.0]
[12.0, 0.833333333333333]
[11.1666666666667]
```

Эти строки соответствуют тому, что мы писали выше в таблице 1. Последняя строка вывода содержит результат вычислений.

Язык Ruby является объектно-ориентированным языком. Строки, массивы, числа и др. сущности в нём являются объектами определённых классов. У каждого объекта есть методы – действия, которые можно с ним делать (или которые он может делать с переданными аргументами). Для вызова метода объекта нужно написать выражение или имя переменной, поставить точку, затем имя метода и в круглых скобках через запятую перечислить аргументы. Если метод не получает аргументов, то круглые скобки можно не писать. Например, если переменная **ary** есть массив (объявлена как **ary = [1,2,3]**), то можно в конец массива добавить новое число с помощью метода push: **ary.push(5)**. Чтобы извлечь последний элемент массива, есть метод **pop**: **x = ary.pop**. Для того, чтобы добавлять в начало и извлекать из начала, есть методы **unshift** и **shift** соответственно. Таким образом, массив в Ruby предоставляет функциональность не только массива, но и стека и очереди одновременно.

Метод **post_to_inf** получает строку **input** с записью выражения, а возвращает стек (стопку) с числами **stack**, который получился в результате вычислений. Если количество чисел в исходном выражении ровно на один больше количества арифметических операций, то по завершении алгоритма в стеке будет ровно одно число – результат вычислений.

Строка **"input.split.each do |token|"** читается как «разбить входное выражение на отдельные слова, между которыми в исходном выражении стояли пробелы, и для каждого слова **token** проделать следующие действия». Метод **split** играет в нашем случае роль токенайзера. В других нотациях может потребоваться более сложный токенайзер. Так, например, в некоторых нотациях пробелы между элементами необязательны и алгоритм деления последовательности символов на токены более сложный. Но мы для простоты будем полагать, что все элементы (числа и знаки операций) разделены пробелами.

Строка **"puts stack.inspect"** печатает текущее состояние массива **stack**. В языке Ruby очень интересно оформляется оператор случая **case**.

```
case token
when '+', '-', '*', '/'
  действия, когда token равен
  одному из перечисленных знаков
  операций
else
  действия для случая, когда
  token не есть знак операции
end
```

После ключевого слова **when** можно перечислять через запятую возможные варианты, которые мы хотим рассмотреть в данном блоке **when**. Итак, когда к нам приходит знак действия, мы проделываем следующее:

`a2 = stack.pop`	Взять число сверху стопки.
`a1 = stack.pop`	Взять следующее число сверху стопки.
`stack.push(a1.send(token, a2))`	Для числа **a1** выполнить операцию **token** с числом **a2**.

Метод **send** довольно необычен. В аргументе он получает имя метода (строку), который нужно выполнить, и аргументы для этого метода. Это позволяет конструировать название методов и выполнять их в процессе выполнения программы. В языке Ruby знаки арифметических операций являются полноценными методами у численных объектов (есть пять классов, являющихся численными: **Fixnum**, **Bignum**, **Float**, **Complex**, **Rational**).

Перевод из постфиксной в инфиксную

Мы научились вычислять постфиксные выражения. Теперь рассмотрим новую задачу – необходимо не вычислить, а перевести постфиксное выражение в инфиксную нотацию. Это делается малым изменением

представленного выше кода. Всё, что нужно сделать – это подправить пару строк в операторе **case**:

```
    case token
    when '+', '-', '*', '/'
      a2 = stack.pop
      a1 = stack.pop
      stack.push('(' + a1 + token + a2+ ')')
    else
      stack.push(token)
    end
```

Теперь мы не осуществляем вычисления, а храним в стеке строки с инфиксными выражениями. После извлечения двух выражений из стека мы с помощью конкатенаций (слияний) строк конструируем новую строку с выражением, которую и помещаем в стек. Вывод программы теперь будет иметь следующий вид:

```
[]
["1"]
["1", "2"]
["(1+2)"]
["(1+2)", "4"]
["((1+2)*4)"]
["((1+2)*4)", "5"]
["((1+2)*4)", "5", "6"]
["((1+2)*4)", "(5/6)"]
["(((1+2)*4)-(5/6))"]
```

Но надо сказать, что представленный код не самый эффективный. Выполним следующие строки, чтобы определить время работы нашего алгоритма:

```
require 'benchmark'
[1000,2000,4000,8000].each do |n|
  puts Benchmark.measure {
    input = '0 ' + (1..n).to_a.join(' + ') + ' +'
    post_to_inf(input)
  }
end
```

Эти строки измеряют время работы метода **post_to_inf** для больших входных данных. Входные данные здесь просто запись суммы N целых чисел, где N последовательно пробегает значения 1000, 2000, 4000 и 8000.

```
>ruby post2inf.rb
  user time   system time   total      real
  0.047000    0.000000     0.047000 (  0.047000)
  0.109000    0.047000     0.156000 (  0.156000)
```

```
 0.500000   0.187000     0.687000 (  0.719000)
 1.875000   0.860000     2.735000 (  2.750000)
```

Несложно заметить, что при увеличении размера входных данных в два раза время работы вырастает примерно в четыре раза (см. третий столбец **total**). В таких случаях говорят, что время растёт согласно квадратичному закону.

Это связанно с тем, что при получении знака операции каждый раз конструируется новая строка из двух существующих. Длины строк могут расти линейно с каждой итерацией, и на слияние строк на каждой итерации будет уходить время, пропорциональное размеру этих строк. Мы не будем здесь строго доказывать этот факт. Удовлетворимся экспериментальным данными, к тому же причина неэффективности алгоритма более-менее ясна и легко устранима. В действительности не нужно осуществлять лишние операции по слиянию строк. Можно хранить списки строк, не соединяя их в одну строку. Идеально было бы хранить в стеке не записи выражений в виде строк, а списки токенов в том порядке, в котором они должны идти в выражении. Слияние списков («дописать один список в конец другого») делается быстро и выполняется в константное время – время, не зависящее от длины списков. А слияние строк требует времени, пропорционального размеру строк. В стандартной библиотеке Ruby нет списков, зато есть возможность делать вложенные массивы. Давайте используем эту возможность. В новой версии программы будет изменена лишь одна строка в операторе **case**:

```
stack.push(['(', a1, token, a2, ')'])
```

В результате вывод будет выглядеть так:

```
[["(", ["(", ["(", "1", "+", "2", ")"], "*", "4", ")"], "-",
["(", "5", "/", "6", ")"], ")"]]
```

Это довольно громоздкое выражение. Оно представляет собой систему вложенных массивов. Но если из него удалить квадратные скобки и лишние кавычки, то получится нужный нам ответ. Удаление скобок (раскрытие всех вложенных массивов в один) осуществляет метод **flatten**, а слияние массива строк в одну осуществляет метод **join**. Таким образом, строка

```
puts post_to_inf(input).flatten.join.inspect
```

выведет нужный нам ответ

```
"(((1+2)*4)-(5/6))".
```

Заметим, что вызов метода **flatten** можно опустить, так как **join** действует рекурсивным образом. Методы работают линейное время в зависимости от размера результата (начиная с версии ruby-v1.9.1). Можно самим реализовать упрощённый вариант метода **flatten**, который работает линейное время:

```
class Array
  def flatten
    ary = self.dup
    res = []
    while !ary.empty?
      e = ary.pop
      if e.is_a?(Array)
        ary.push(*e)
      else
        res.push(e)
      end
    end
    res.reverse
  end
end
```

Задача. Изучите логику работы данного метода самостоятельно. Попробуйте написать рекурсивную версию метода **flatten** и сравните скорости работы двух реализаций. Как отличается функционал предложенной версии от функционала стандартной реализации? Для того, чтобы ответить на последний вопрос, необходимо прочитать документацию к методу **Array#flatten** и, возможно, посмотреть исходные коды

```
array.c (ftp://ftp.ruby-lang.org/pub/ruby/stable-snapshot.tar.gz).
```

Итак, вы можете сами убедиться в том, что время стало расти линейно. Почему это так? В Ruby массивы и другие объекты (кроме нескольких исключений – короткие числа, символы, а также значения **nil**, **true** и **false**) передаются по ссылке, и когда один массив помещается внутрь другого, в действительности помещается лишь ссылка на этот массив, и не происходит копирования всех его «внутренностей». Строка

```
stack.push(['(', a1, token, a2, ')'])
```

выполняется быстро – за константное время, не зависящее от размеров содержимого переменных **a1** и **a2**. В результате работы этой строки создаётся система вложенных массивов (извлечённые из стека значения **a1** и **a2** могут быть числами или массивами), которую несложно в конце раскрыть за линейное время.

Исследовательская задача

Наш код вывел выражение **"(((1+2)*4)-(5/6))"**. В нём много скобок. В действительности достаточно было одной пары скобок:

```
(1+2)*4-5/6
```

Скобки не нужны, так как умножение и деление имеют больший приоритет. Напишите программу на одном из языков программирования, которая переводит из постфиксной нотации в инфиксную и при этом не выводит лишних скобок. При этом в результирующем выражении числа должны идти в том же порядке, что и в исходном выражении. Ниже приведены примеры входа и выхода программы:

Вход	Выход
56 34 213.7 + * 678 -	56*(34+213.7)-678
1 56 35 + 16 9 - / +	1+(56+35)/(16-9)
1 2 3 * *	1*2*3
3 4 5 / *	3*4/5
3 4 * 5 /	3*4/5
3 4 5 / /	3/(4/5)

Обратите внимание на то, что здесь важен не только приоритет операций, но и ассоциативность соответствующих арифметических действий. Например, умножение ассоциативно, то есть **a*(b*c) = (a*b)*c**, в то время как деление – нет: **a/(b/c) ≠ (a/b)/c** (в общем случае равенство не выполнено).

Высылайте свои решения на info@potential.org.ru.

Лучшие решения будут опубликованы в одном из следующих номеров журнала.

Попробовать свои решения вы можете на сайте http://acm.mipt.ru/judge/problems.pl?problem=126.

Там есть возможность послать свои решения на автоматическую проверку. Удачи!

Дерево разбора

И напоследок ещё одна полезная функциональность, которую мы можем получить дёшево с помощью библиотеки ' YAML' .

Ещё раз изменим нашу злополучную строку в методе **post_to_inf**.

Сделаем так, чтобы в массиве **stack** хранились триплеты: [знак операции, описание левой части, описание правой части], то есть будем хранить записи наших выражений в префиксной форме. Вот эта строка:

```
stack.push(['(', a1, token, a2, ')'])
```

Затем подключим библиотеку **'yaml'** и выполним метод **to_yaml** у результата выполнения метода **post_to_inf**:

```
input = '1 2 + 4 * 5 6 / -'
require 'yaml'
```

```
puts post_to_inf(input).last.to_yaml
```

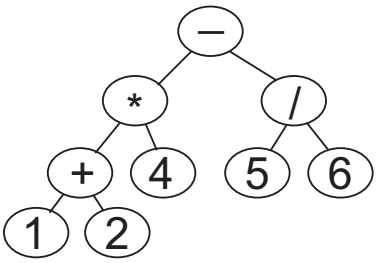
Мы получим следующий вывод:

```
---
- "-"
- - "*"
  - - +
    - "1"
    - "2"
  - "4"
- - /
  - "5"
  - "6"
```

Это префиксное описание дерева разбора нашего выражения. Само дерево нарисовано справа.

Дерево разбора – это набор узлов (круги), из которых исходят линии вниз к *дочерним узлам*, и может идти вверх линия к *родительскому узлу*. Один узел (самый верхний) является корнем – все остальные являются его детьми или (пра)^n-внуками. В узлах дерева разбора помещены либо операторы (в нашем случае – знаки арифметических операций), либо числа. Каждому узлу соответствует некоторое подвыражение. Можно представлять себе, что в каждом узле находится результат вычисления этого подвыражения.

От узла оператора исходят линии к дочерним узлам, в которых как бы находятся результаты вычислений выражений, являющихся операндами данного оператора.



(1 + 2) * 4 – 5 / 6

Дерево разбора является некоторым универсальным форматом хранения выражения, по которому несложно получить как результат вычислений, так и представления выражения в разнообразных нотациях.

Задача. Нарисуйте деревья разбора следующих выражений:

а) (1+(2+(3+(4+5))));
б) ((((1*2)*3)*4)*5);
в) 1−2*3/4+5;
г) 1−2*3/4+5*(6+7*8).

Обратите внимание на то, что корнем дерева является оператор, который выполняется в последнюю очередь. Чем ниже оператор находится в дереве, тем раньше он выполняется при вычислении выражения. Интересно также заметить, что если спроектировать значения узлов на горизонталь, то получится инфиксная запись выражения (без скобок).

Юмор Юмор Юмор Юмор Юмор Юмор

Точный прогноз

Математик заинтересовался таким прогнозом погоды, в котором говорилось: «Вероятность дождя 70%». Он позвонил в бюро прогнозов и спросил, какой теорией они пользуются при расчётах. Служащий ответил:

– Видите ли, у нас в конторе 10 человек. Если семеро говорят, что им кажется, будет дождь, а остальным всё равно, будет дождь или нет, то мы получаем именно 70%.