Введение

Ленивость вычислений. В большинстве языков программирования при вычислении суперпозиции функций сначала вычисляются самые вложенные функциональные обращения. Например, при вычислении

```
f1( f2(a), f3(b) )
```

сначала будут вычислены обращения к функциям `f2` и `f3`, а уже затем обращение к функции `f1`, использующей результаты вычислений `f2(a)` и `f3(b)`. В Haskell сначала будет вызвана функция `f1`, как самая внешняя, которой будут переданы выражения `f2(a)` и `f3(b)` в первоначальном, невычисленном виде. Вычисление выражения, заданного в качестве аргумента функции, происходит в тот момент, когда он становится востребованным в процессе вычисления тела функции. Т.е. какой-либо аргумент функции (выражение, заданное в качестве аргумента) может вообще остаться невычисленным, если в процессе вычисления тела функции он не понадобился.

Типы данных

Haskell является языком со строгой типизацией, все объекты в нём имеют свой тип (иногда тип объекта не указывается, поскольку может быть выведен компилятором автоматически). Базовые типы данных включают:

- `Int` – целые из некоторого ограниченного диапазона (размерностью 2 или 4 байта);
- `Integer` – целые числа неограниченного размера (размер ограничивается только количеством памяти, выделенной для представления числа); если вы когда-либо писали программы, работающие с длинной арифметикой – это как раз такие числа.
- `Float` – вещественные числа (числа с плавающей точкой);
- `Bool` – булевские значения – константы `True` и `False`;
- Для работы с символами и строками есть типы `Char` и `String`, они будут рассмотрены позже.

Для базовых типов определены различные константы, например, константа `pi`.

Перечислим некоторые встроенные функции.

Арифметические: + (сложение), – (вычитание), `*` (умножение), `div` (деление нацело для типов Int и Integer), `mod` (остаток от деления нацело для типов Int и Integer), `/` (деление для всех числовых типов, но результатом всегда является вещественное число).

Операции сравнения: > (больше), < (меньше), >= (не меньше), <= (не больше), == (равно), `/=` (не равно).

Логические: `not` (отрицание), `&&` (конъюнкция), `||` (дизъюнкция).

Тригонометрические: `sin` (синус), `cos` (косинус), `tan` (тангенс), `asin` (арксинус), `acos` (арккосинус), `atan` (арктангенс).

Некоторые другие функции для чисел: `sqrt` (квадратный корень), `log` (логарифм), `exp` (экспонента), `odd` (проверка нечётности целого), `even` (проверка чётности целого), `abs` (абсолютная величина), `round` (округление вещественного до целого).

Замечание: Функции, имена которых состоят из алфавитных символов, записываются в префиксной форме (имя функции перед аргументами). Бинарные функции, имена которых состоят из неалфавитных символов, записываются в инфиксной форме – функция между своими операндами. Примеры выражений:

```
2 + 2 * 3

div 5 2

True && 2 > 3
```

На основе базовых определяются более сложные типы. Типы функций определяются с помощью символов `->`:

Имя_функции `::` тип1арг `->` тип2арг `->` … `->` типNарг `->` тип_результата

Например:

`Int -> Int` – функция одного аргмента типа `Int`, возвращающая значение типа `Int`;

`Float -> Integer` – тип аргумента `Float`, тип результата `Integer`.

`Int -> Int -> Int` – функция двух аргументов типа `Int`, возвращающая `Int`;

`Float -> Integer -> Bool` – функция двух аргументов (первый имеет тип `Float`, второй тип `Integer`), возвращающая значение типа `Bool`.

Аналогично для функций большего числа аргументов. Видно, что для разделения аргументов используются те же самые символы `->`. Это вызвано тем, что в Haskell функции частично применимы, т.е. функция двух аргументов может быть вызвана с одним – в результате получится функция одного оставшегося аргумента. Функция `N` аргументов в Haskell воспринимается как функция одного аргумента, которая возвращает в качестве результата “остаточную функцию” `N-1` аргумента. Разумеется, это же верно и для остаточной функции и так далее. Например, если мы имеем функцию:

```
f :: Int -> Int -> Int -> Bool
```

То это то же самое, что и:

```
f :: Int -> (Int -> (Int -> Bool))
```

То есть в записи типа функции скобки могут быть расставлены **справа налево**. Теперь рассмотрим процесс вызова функции f:

```
f 1 2 3
```

Применение функции к нескольким аргументам это есть ни что иное, как применение функции к одному аргументу, затем применении полученного результата (остаточной функции) ко второму и так далее. Соответственно, следующая запись эквивалентна предыдущей:

```
((f 1) 2) 3
```

То есть при вызове функций скобки могут быть расставлены **слева направо**.

Такая частично применённая функция может быть использована, например, при определении новой функции. Например, если у нас есть функция `f x y = x+y`, то вызвав её с одним аргументом `f 2` получим в результате функцию, увеличивающую свой единственный аргумент на два:

`f y = 2+y` для для любого `y`.

Таким образом, все функции могут рассматриваться как функции одного аргумента, возвращающие остаточную функцию.

Функции

Перейдём к тому, как определяются функции. Например, функция, возводящая число в квадрат:

```
square :: Integer -> Integer
square x = x * x
```

Функция вычисления удвоенного произведения чисел:

```
dmult :: Integer -> Integer -> Integer
dmult x y = 2 * x * y
```

Первая строка указывает тип определяемой функции (он записывается после `::`), вторая – тело функции. Если вспомнить Рефал, то можно сказать, что определение тела функции выглядит весьма похоже на правило замены – левая часть (образец, расположенный до символа равенства =) заменяется на правую часть (стоящую после символа равенства).

Теперь о том, как функции вызываются. При вызове функции аргумент записывается через пробел после её имени, например:

```
square 2
dmult 3 5
```

Стоит отметить, что здесь нет скобок. **В Haskell скобки вокруг аргументов функций отсутствуют**. Пример выше можно записать как `square(2)`, однако скобки здесь излишне – это не скобки вокруг аргументов. Они имеют тот же смысл, что и скобки в выражении `2+(2)`. Соответственно, если попробовать написать `dmult (3 5)` – то это будет ошибка, поскольку Haskell будет считать что функция dmult применяется к одному аргументу – выражению, заключённому в скобки, а оно в свою очередь является некорректным. Поскольку операция применения функции имеет наибольший приоритет, возникает необходимость заключать аргумент функции в скобки, если он является составным выражением. Например, запись `square square 2` – некорректная, поскольку будет воспринята как обращение к функции `square` с аргументом – функцией `square`. Поэтому в данном случае необходимо заключить в скобки аргумент первого обращения к функции `square`: `square (square 2)`. Вычисление выражения `square 2-2` выведет `2`, а не ноль, поскольку это выражение эквивалентно выражению `(square 2)-2`.

Любую бинарную функцию в Haskell можно использовать в инфиксной форме, для чего её имя при записи между аргументами обрамляется обратными апострофами. К примеру, обращение к определённой выше функции `dmult` может быть записано в следующем виде:

```
3 `dmult` 5
```

Кроме того, возможно определение функций, чьё имя состоит из неалфавитных символов (при построении имени допустимы символы `: # $ % & * + - = . / \ < > ? ! @ ^ |`), тогда при определении такой функции её имя берётся в скобки. Например:

```
(???!) :: Int -> Int -> Int
```

```
(???!) a b = a*b - a - b
```

Для вызова в префиксной форме имя этой функции обрамляется скобками:

```
(???!) 2 3
```

Однако при использовании в инфиксной форме её имя пишется без апострофов:

```
2 ???! 3
```

Заметим, что использование чего-то подобного для своих типов позволяет иногда улучшить (а иногда ухудшить) читаемость кода.

Константы определяются как функции не имеющие аргументов, например:

```
eps :: Float
eps = 0.00001
```

Часто вычисление функции происходит по разному в зависимости от значений её аргументов, поэтому возникает необходимость в средствах ветвления вычислений. Первым средством организации ветвления является запись функции в несколько строк, что хорошо показывается на примере функции, вычисляющей факториал целого числа:

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

Первая строка определяет факториал для числа `0` (база индукции), а вторая задаёт формулу рекурсивного вычисления факториала для всех остальных положительных чисел.

Если обращаться к Рефалу, аналогии с образцами очевидны – выражение вызова функции последовательно сверху вниз сравнивается с образцами в определении, для первого подходящего подставляется тело (выражение после знака равенства). Если подходящий образец не найден – выдается ошибка.

Рассмотрим функцию, вычисляющую конъюнкцию двух булевских значений:

```
and :: Bool -> Bool -> Bool
and True True = True
and _ _ = False
```

Первое предложение функции применимо только в том случае, когда значениями обоих аргументов является истина, в этом случае и результатом будет `True`. Если же первое предложение неприменимо, то будет применено второе предложение, и результатом функции будет `False`. В образце второго предложения используется `_` для обозначения анонимной переменной (как в Прологе), такая переменная сопоставляется с любым выражением. Анонимная переменная используется тогда, когда сопоставляемое с ней значение не используется в правой части (не участвует в дальнейших вычислениях). Различные вхождения анонимной переменной в образец не связаны друг с другом, поэтому образец второго предложения может быть сопоставлен с любой парой булевских значений (в нашей задаче он будет успешно сопоставлен с любой из трёх оставшихся пар значений: `True` и `False`, `False` и `True`, `False` и `False`).

Рассмотрим другой классический пример рекурсивной функции: вычисление n-го числа ряда Фиббоначи:

```
fib :: Integer -> Integer
fib 1 = 1
fib 2 = 1
fib n = fib(n-1) + fib(n-2)
```

Вспомним про то, что помимо корректности от программ часто требуется приемлемое время работы и потребление памяти. Приведённые примеры функций вычисления факториала и вычисления n-го числа ряда Фиббоначи в этом смысле не очень хороши. Несмотря на то, что `Integer` позволяет представление очень больших чисел, вызов `fact 100000` приводит к ошибке `Control stack overflow`, а вызов `fib 30` вычисляется непозволительно долго. Для того, чтобы продемонстрировать почему так происходит, можно для начала рассмотреть вычисление функции как последовательность переписываний. Например, для вычисления факториала шести:

```
fact 6
6* fact 5 (по 2 строке определения)
6 * 5 * fact 4
6 * 5 * 4 * fact 3
6 * 5 * 4 * 3 * fact 2
6 * 5 * 4 * 3 * 2 * fact 1
6 * 5 * 4 * 3 * 2 * 1 * fact 0
6 * 5 * 4 * 3 * 2 * 1 * 1 (по 1 строке определения)
720
```

Видно, что в процессе переписывания размер выражения линейно растёт. Аналогично, растет и память, необходимая для вычисления функции (конкретнее – каждый вызов требует места в стеке). Соотвественно, при больших значениях аргумента памяти не хватает и вычисление прерывается с ошибкой.

Чтобы избежать такой ситуации можно использовать метод накапливающего параметра, который состоит в том, чтобы передавать промежуточный результат вычисления в качестве параметра функции. Например, функцию вычисления факториала с использованием этого метода можно определить так:

```
fact :: Integer -> Integer
fact n = factNew n 1


factNew :: Integer -> Integer -> Integer
factNew 0 f = f
factNew n f = factNew (n-1) (n*f)
```

Здесь `factNew` принимает вторым аргументом множитель, на который должен быть умножен факториал первого. Поскольку вычисление идет сверху вниз, то там хранится “верхняя часть” факториала. Вычисление `fact 6` будет выглядеть следующим образом:

```
fact 6
```

```
factNew 6 1
factNew 5 6
factNew 4 30
factNew 3 120
factNew 2 360
factNew 1 720
factNew 0 720
720
```

В данном случае видно, что размер выражения остаётся постоянным, что позволяет вычислить его с использованием константного объема памяти.

Вернёмся к задаче с числами Фиббоначи. Там дело обстоит еще хуже, поскольку рост объёма выражения не линейный, а экспоненциальный (на каждом следующем уровне до определенного момента длина выражения удваивается). Например, при выичсления седьмого элемента ряда Фиббоначи:

```
fib 7
fib(5)+fib(6)
fib(3)+fib(4)+fib(4)+fib(5)
fib(1)+fib(2)+fib(2)+fib(3)+fib(2)+fib(3)+fib(3)+fib(4)
1+1+1+fib(1)+fib(2)+1+fib(1)+fib(2)+fib(1)+fib(2)+fib(2)+fib(3)
3+1+1+1+1+1+1+1+fib(1)+fib(2)
11+1+1
13
```

Как определить эту функцию с использованием накапливающего параметра? Предположим, что мы вычисляем n-е число не в стандартном ряде Фиббоначи, а в ряде, который начинается с чисел `a` и `b`. Тогда, такое вычисление - это то же самое, как и вычисление `n-1`-го числа в ряде начинающемся с чисел `b` и `a+b`. В коде:

```
fib :: Integer -> Integer
fib n = fibNew 1 1 n


fibNew :: Integer -> Integer -> Integer -> Integer
fibNew a b 1 = a
fibNew a b 2 = b
fibNew a b n = fibNew b (a+b) (n-1)
```

Как теперь будет вычислено 7-е число ряда Фиббоначи:

```
fib 7
fibNew 1 1 7
```

```
fibNew 1 2 6
fibNew 2 3 5
fibNew 3 5 4
fibNew 5 8 3
fibNew 8 13 2
13
```

Опять, достаточно константного объема памяти.

Стоит отметить, что данное описание, хотя и хорошо иллюстрирует использование метода накапливающего параметра, находится очень далеко от реального процесса вычисления в Haskell, в первую очередь за счёт ленивости вычислений. Например, в реальности определение факториала с методом накапливающего параметра точно так же завершается с переполнением стека. ***Чтобы оно работало, необходимо перед вторым аргументом рекурсивного вызова*** `factNew` ***поставить специальный знак (***`$!`***), предписывающий обязятельное вычисление аргумента перед началом вычисления функции.*** Таким образом, исправленное определение функции вычисления факториала с использованием накапливающего параметра выглядит следующим образом:

```
fact :: Integer -> Integer
fact n = factNew n 1


factNew :: Integer -> Integer -> Integer
factNew 0 f = f
factNew n f = factNew (n-1) $! (n*f)
```

Эта функция вычисления факториала будет работать именно так, как было описано выше.

Вернемся к определению функций. В Haskell функции являются такими же объектами, как и, например, числа, в том смысле, что их можно передавать в другие функции, возвращать из функций и т.п. Например, можно написать функцию `twice` (имеющую два агумента – функцию и число), которая дважды применяет заданную функцию к заданному числу:

```
twice f x = f (f x)
```

Выглядит весьма естественно. Необходимо только описать тип функции:

```
twice :: (Integer -> Integer) -> Integer -> Integer
```

Можно написать и более общую функцию, которая применяет заданную функцию к аргументу `n` раз:

```
applyn :: Int -> (Integer -> Integer) -> Integer -> Integer
applyn 0 _ x = x
applyn n f x = applyn (n-1) f (f x)
```

Или, что более интересно, функцию вычисления производной (точнее, её разностного приближения) в точке:

```
diff :: (Float -> Float) -> Float -> Float
```

```
diff f x = (f(x+eps) – f(x-eps))/(2*eps)
```

Синтаксические конструкции

Иногда сопоставления с образцом недостаточно для описания вариантов вычисления функции, например, с ним нельзя написать функцию вычисления знака числа `sign`, или функцию нахождения минимального из двух чисел `min`. Разумеется, в Haskell есть и другие механизмы ветвления вычислений. Во-первых, это обычный `if`:

```
sign :: Integer -> Int
sign 0 = 0
sign x = if x > 0 then 1 else -1
```

Другой, более мощный механизм ветвления вычислений – это ***охранные выражения***, которые могут быть записаны за образцом после вертикальной черты и ограничиают входящие в него переменные. Охранное выражение состоит из условия (булевского выражения), знака равенства и правой части. Например:

```
sign :: Integer -> Int
sign 0 = 0
sign x | x < 0 = -1
sign x | x > 0 = 1
```

Повторяющийся образец можно не переписывать, но тогда необходим отступ (символы вертикальной черты | должны стоять на одинаковых позициях от начала строки):

```
sign :: Integer -> Int
sign 0 = 0
sign x | x < 0 = -1
       | x > 0 = 1
```

Охранные выражения просматриваются последовательно сверху вниз до тех пор, пока не буде найдено условие, результат вычисления которого `True`. Тогда результатом вычисления функции станет выражение, стоящее после этого условия и знака равенства =. Обратим внимание, что если подходящая пара образец-условие не будет найдено, то вычисление прервётся сообщением об ошибке. Чтобы этого избежать можно в последней ветви в качестве условия использовать константу `True` (или её синоним – кнстанту `otherwise`).

Обратим внимание, что в нашем примере последняя строка начинается с отступа (символы вертикальной черты | находятся друг под другом) – это пример так называемого ***двумерного синтаксиса*** в Haskell. В соответствии с ним, отступ влияет на то, как вопринимается строка:

- Если отступ такой же, как у предыдущей строки, то эта строка – новое определение (эквивалентные определения должны иметь равные отступы);
- Если отступ больше – то новая строка является продолжением предыдущей строки;
- Если отступ меньше – продолжение предыдущих строк (или эквивалентных

определений) завершено.

Поскольку иногда требуется использовать сопоставление с образцом и охранные выражения в середине определения функции, то в языке есть конструкция `case`, которая это позволяет. Например с ней можно определить фнкцию `min` так:

```
min x y = case x < y of
              True -> x
              False -> y
```

В Haskell есть две конструкции, позволяющие упростить запись сложных выражений за счёт именования подвыражений в формулах (аналогично дополнительным переменным в императивных языках). Это конструкции `let` и `where`.

Рассмотрим использование этих конструкций на примере задачи вычисления минимального корня квадратного уравнения с заданными коэффициентами `a`, `b` и `c`:

```
minRoot :: Float -> Float -> Float
minRoot a b c = min ((-b-sd)/n) ((-b+sd)/n)
                where sd = sqrt(b*b - 4*a*c)
                      n = 2*a
```

Или та же функция, использующая конструкцию `let`:

```
minRoot a b c = let sd = sqrt(b*b - 4*a*c)
                    n = 2*a
                in min ((-b-sd)/n) ((-b+sd)/n)
```

Единственное отличие этих двух конструкций – в первом случае дополнительные обозначения идут после выражения, во втором до него. Какую из них использовать – дело вкуса.

Запуск интерактивной консоли

Для Haskell есть два основных компилятора – `hugs` и `ghc`, оба могут быть легко найдены (или установлены из репозитория в дистрибутивах Linux).

В случае hugs - команда `hugs` запускает интерактивную консоль, в которой можно вычислять выражения. Определять функции в ней нельзя, поэтому их необходимо описывать в отдельном текстовом файле с расширением `.hs` и загружать в интерактивной консоли с помощью команды `:load <имя файла>`. В случае наличия в файле ошибок, они будут выведены на экран. Если ничего не выведено – ошибок нет и можно использовать функции, определённые в файле.