1 Интерпретатор WinHugs

Запуск интрепретатора осуществляется командой `winhugs`.

```
__   __ __  __  ____   ___     ------------------------------------------------
||   || ||  || ||  || ||__     Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||    Copyright (c) 1994-2005
||---||         ___||          World Wide Web: http://haskell.org/hugs
||   ||                        Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: Sep 2006      ------------------------------------------------

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs>
```

Приглашением интерпретатора для ввода команды является строка `Hugs>`. Имя в приглашении может меняться в зависимости от загруженной библиотеки функций.

Каждая команда записывается одной строкой. Нажатие клавиши Enter приводит к выполнению команды После ввода команды система её анализирует, компилирует, выполняет и выводит результат выполнения на терминал.

Образец диалога:

```
Hugs> 3+2
5
```

Команды, принимаемые интерпретатором можно разделить на две категории:

1. выражения на языке Haskell, при подаче которых вычисляется их результат;
2. директивы интерпретатора (аргументами которых могут выступать выражения языка Haskell), влияющие на среду приложения WinHugs.

Имена директив интерпретатора начинаются с символа «:».

Директива `:set` (или кратко `:s`) изменяет значение указанного параметра интерпретатора. Для нас будет интересен параметр `t` отвечающий за отображение типов данных вычисленных выражений. Включить этот параметр можно следующим образом.

```
Hugs> :set +t
```

После включения параметра `t` первый диалог с интерпретатором будет выглядеть следующим образом.

```
Hugs> 3+2
5 :: Integer
```

Отключить параметр `t` можно так:

```
Hugs> :s -t
```

Включить или выключить определенный параметр можно непосредственно при запуске интерпретатора указав в его командной строке. Например, для запуска интерпретатора с уже включенным параметром `t` нужно дать команду

```
winhugs +t
```

Директива `:load` (или `:l`) позволяет загрузить файл с программой.

```
Hugs> :load "F:\\Sample.hs"
Main>
```

После успешной загрузки программы приглашение интерпретатора меняется.

Обычное расширение файла с программой на языке Haskell — `hs`. Если программа имеет такое расширение, то в директиве `:load` расширение можно не указывать.

Если директиву `:load` дать без параметров, то все загруженные модули будут выгружены из памяти.

```
Main> :l
Hugs>
```

Директива `:type` (или `:t`) выдает тип данных своего аргумента (аргумент может быть выражением, функцией, константой).

```
Hugs> :type 42
42 :: Num a => a
Hugs> :type sqrt
sqrt :: Floating a => a -> a
Hugs> :type 2+2
2 + 2 :: Num a => a
Hugs> :t 2**4
2 ** 4 :: Floating a => a
```

С помощью директивы `:quit` (или `:q`) можно закончить работу с интерпретатором.

Выражение на языке Haskell может содержать константы и вызовы функций. Существует следующие формы записи вызова функции двух аргументов:

1. если функция имеет имя, заключенное в круглые скобки (например, `(+)`), то для нее можно использовать инфиксную запись, при которой круглые скобки опускаются;

2. если имя функции имеет имя не заключенное в круглые скобки (например, `mod`), то для её вызова можно использовать инфиксную запись, при которой имя функции заключается в обратные апострофы.

```
Hugs> (+) 3 7
10 :: Integer
Hugs> 3 + 7
10 :: Integer
Hugs> mod 8 3
2 :: Integer
Hugs> 8 `mod` 3
2 :: Integer
```

В интерпретаторе языка Haskell не позволяется определять новые функции. Такие определения должны быть вынесены в файл с программой.

Пусть в файле `sample.hs` определена функция `square`:

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

Загрузив в этот файл можем провести следующий диалог

```
Hugs> :load "sample"
Main> square 25
625
Main> square 1.5
ERROR - Cannot infer instance
*** Instance   : Fractional Integer
*** Expression : square 1.5

Main> :t square
square :: Integer -> Integer
```

Для включения в тело программы на языке Haskell однострочных комментариев используется комбинация символов `--`. Текст, расположенный между ними и концом строки, считается комментарием.

Комбинациями символов `{-` ... `-}` ограничивают многострочные комментарии. Текст, заключенный между ними, игнорируется. Такой комментарий может быть вставлен на место любого пробела в программе. Допускаются и вложенные комментарии.

2 Базовые типы данных

2.1 Тип `()`

Тип `()` состоит из единственного значения, записываемого как `()`. Является аналогом типа `unit` в языке Standard ML.

```
Hugs> ()
() :: ()
```

2.2 Тип `Bool`

Тип `Bool` состоит из значений `True` и `False`. Для работы со значениями этого типа имеются одноместная операция `not` (не) и две двухместных операции `(&&)` («и») и `(||)` («или»).

```
Hugs> not True
False :: Bool
Hugs> False && True
False :: Bool
Hugs> False || True
True :: Bool
```

Конструкция `if` аналогична подобной конструкции в Standard ML.

Для удобочитаемости программ в языке определена константа `otherwise` равная `True`.

2.3 Числовые типы данных

Тип `Int` — тип целых чисел, охватывает по меньшей мере диапазон $[-2^{29}, 2^{29} - 1]$.

Тип `Integer` — тип целых чисел произвольной точности.

Типы `Integer` и `Int` являются объектами класса `Integral` — класса всех целочисленных типов.

Типы `Float` и `Double` — типы вещественных чисел одиночной и двойной точности соответственно. Эти типы являются объектами класса `Fractional` — класса всех вещественных значений соответственно, а класс `Num` объединяет в себе все числовые типы данных.

В дальнейшем, если элемент относится к типу данных класса X, то будем говорить, что этот элемент типа X.

К элементам типа `Num` применимы функции `(+)`, `(-)`, `(*)`, `negate` (противоположное по знаку число), `abs` (модуль), `signum` (знак). Значение этих функций будет того же типа, что и аргументы.

Для элементов, принадлежащих типу `Integral` определены стандартные функции: `quot` (целая часть от деления), `rem` (остаток от деления), `div` (целая часть от деления), `mod` (остаток от деления), `quotRem`, `divMod` (пара: целая часть, остаток). Значение этих функций будет того же типа, что и аргументы.

```
Hugs> 1287873 `mod` 376
73 :: Integer
Hugs> divMod 1287873 376
(3425,73) :: (Integer,Integer)
```

Функция `(/)` принимает числовые аргументы, а возвращает результат типа `Fractional`.

Функции `fromInteger` b `fromIntegral` (преобразуют тип данных) получают аргумент типа `Integer` или `Integral` соответственно, а результат — необходимый числовой тип. Функция `toInteger` (преобразует тип данных в тип `Integer`) применяется к аргументу типа `Integral`.

Функции для преобразования вещественного числа к целому: `truncate`, `round`, `ceiling`, `floor`.

```
Hugs> ceiling 56.235466
57 :: Integer
Hugs> floor 145.58787
145 :: Integer
Hugs> round 145.58787
146 :: Integer
```

Определены следующие вещественнозначные функции: `pi`, `exp`, `log`, `sqrt`, `logBase`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`.

```
Hugs> logBase 3 27
3.0 :: Double
Hugs> cos (pi/2)
6.12303176911189e-017 :: Double
Hugs> 2 * asin (cos pi)
-3.14159265358979 :: Double
```

Функции возведения в степень `(^)`, `(^^)` и `(**)` отличаются друг от друга типом аргументов и типом результата: первые две функции — функции возведения в целочисленную степень; первая функция возвращает результат того же типа, что и первый аргумент, а остальные функции возвращают вещественнозначный результат.

```
Hugs> 2^4
16 :: Integer
Hugs> 2^^4
16.0 :: Double
Hugs> 2**4
16.0 :: Double
Hugs> 2.3**3.1
13.2238005912547 :: Double
```

Для операций сравнения используются функции `(<)`, `(<=)`, `(>)`, `(>=)`, `(==)` и `(/=)`. Все эти операции возвращают результат типа `Bool`.

```
Hugs> 3<2
False :: Bool
Hugs> 3*2 > 12 `div` 6
True :: Bool
Hugs> if 4*5 `mod` 3 == 1 then 17 else 51
51 :: Integer
```

2.4 Списки

Тип [α] состоит из списков значений типа α. Так значение типа `[Integer]` является списком целых чисел, а значение типа `[Bool]` является списком логических значений. Также как и в Standard Ml здесь имеется два способа записи списков, основной и сокращенный.

Пустой список записывается как `[]`, а непустой список записывается как e`:`l, где e — выражение типа α, а l — выражение типа [α]. В такой записи используется вызов функции «конс» `(:)`. Список из n элементов можно представить как

$$e_1 \ : \ e_2 \ : \ \cdots \ : \ e_n \ : \ [].$$

В сокращенной записи список представляется в виде последовательности разделенных запятыми элементов, заключенной в квадратные скобки.

```
Hugs> 3:4:[]
[3,4] :: [Integer]
Hugs> (3:[]):(4:5:[]):[]
[[3],[4,5]] :: [[Integer]]
Hugs> ["This", "is", "it"]
["This","is","it"] :: [[Char]]
Hugs> []
[] :: [a]
Hugs> [3+5, 7, 8 `div` 3]
[8,7,2] :: [Integer]
```

Для списков предопределен достаточно большой набор функций. Вот некоторые из них

Функции `head` и `tail` возвращают, соответственно, голову (первый элемент) и хвост (список без первого элемента) списка. Функции `last` и `init` возвращают последний элемент и список без последнего элемента.

```
Hugs> head [1, 2, 3, 4, 5, 6, 7]
1 :: Integer
Hugs> tail [1, 2, 3, 4, 5, 6, 7]
[2,3,4,5,6,7] :: [Integer]
Hugs> last [1, 2, 3, 4, 5, 6, 7]
7 :: Integer
Hugs> init [1, 2, 3, 4, 5, 6, 7]
[1,2,3,4,5,6] :: [Integer]
```

Функция `null` выдает `True` если список пуст.

```
Hugs> null [1, 2, 3, 4, 5, 6, 7]
False :: Bool
Hugs> null (tail [1])
True :: Bool
```

`length` возвращает длину списка.

```
Hugs> length [1, 2, 3, 4, 5, 6, 7]
7 :: Int
```

Функция (!!) извлекает из списка элемент с заданным номером.

```
Hugs> [1, 2, 3, 4, 5, 6, 7] !! 3
4 :: Integer
```

Функция (++) объединяет два списка

```
Hugs> [1, 2, 3, 4, 5] ++ [6, 7, 8, 9]
[1,2,3,4,5,6,7,8,9] :: [Integer]
```

Функция map применяет функцию (первый аргумент) к каждому элементу списка и возвращает список результатов.

```
Hugs> map sin [pi/2, pi/3, pi/4, pi/5]
[1.0,0.866025403784439,0.707106781186547,0.587785252292473] :: [Double]
```

2.5 String и Char

Тип данных `Char` представляет из себя перечисление всех символов. Константы типа `Char` записываются как символы, заключенные в кавычки.

Строка (элемент типа `String`) — это список символов. Все функции, применимые для списков можно применять к строкам.

```
Hugs> ['S','t','r','i','n','g',' ','a','s',' ','a',' ','l','i','s','t']
"String as a list" :: [Char]
Hugs> "Fish knuckles"
"Fish knuckles" :: String
Hugs> "Fish knuckles" ++ " bon"
"Fish knuckles bon" :: [Char]
Hugs> "Fish knuckles" !! 6
'n' :: Char
Hugs> length "Fish knuckles"
13 :: Int
```

2.6 Упорядоченные n-ки

Упорядоченная n-ка представляет собой последовательность выражений, отделенных друг от друга запятыми, заключенную в круглые скобки. Если выражения e_1, e_2, ..., e_n имеют типы α_1, α_2, ..., α_n соответственно, то типом упорядоченной n-ки $(e_1, e_2, \ldots, e_n)$ будет $(\alpha_1, \alpha_2, \ldots, \alpha_n)$.

```
Hugs> (True, ())
(True,()) :: (Bool,())
Hugs> (1, False, 17, "Blubbet")
(1,False,17,"Blubbet") :: (Integer,Bool,Integer,[Char])
Hugs> (if 3==5 then "Yes" else "No", 14 `mod` 3)
("No",2) :: ([Char],Integer)
```

3 Описание функций

3.1 Объявления

В языке Haskell разделено объявление функции, т. е. указание ее области определения и области значений, от ее определения — задания правила, по которому осуществляется преобразование входной информации в выходную.

Объявление функции есть описание ее типа, так, фраза `(Integer, Integer) -> Integer` описывает тип функции, которая берет пару целых чисел в качестве аргумента и выдает целое число в качестве результата. Такое описание типа также называется назначением типа или описанием сигнатуры функции.

Синтаксис объявления таков:

```
имя_функции :: область_определения -> область_значений
```

Определяя функцию, мы добавляем к объявлению правило ее вычисления, например,

```
cube :: Integer -> Integer
cube n = n * n * n
```

Здесь определена функция, получающая число типа `Integer` и вычисляющая его куб типа `Integer`.

Функции с нулевым количеством аргументов называются константами, например,

```
e :: Float
e = exp 1.0
```

Возможно определение функций с несколькими параметрами:

```
abcFormula :: Float -> Float -> Float -> [Float]
abcFormula a b c = [(-b + sqrt(b*b - 4.0*a*c))/(2.0*a),
                    (-b - sqrt(b*b - 4.0*a*c))/(2.0*a)]
```

Функция `abcFormula` выдает список корней квадратного уравнения $^2 + + = 0$.

```
Main> abcFormula 1.0 2.0 1.0
[-1.0,-1.0] :: [Float]
```

Объявления функции одного типа можно объединять.

```
negative, positive, isZero :: Integer -> Bool
```

Несколько команд программы можно размещать на одной строке, разделяя их точкой с запятой.

```
negative x = x < 0; positive x = x > 0
isZero x = x == 0
```

```
Main> negative (-457)
True :: Bool
Main> negative 53
False :: Bool
Main> isZero 567
False :: Bool
Main> positive 11
True :: Bool
```

3.2 Локальные объявления и двумерный синтаксис

Ключевое слово where позволяет организовать локальные объявления внутри внутри выражения

```
Hugs> 25 * f (5) where f x = x*x+x^5
78750 :: Integer
Hugs> f x + g 7 where x = 5; f y = y^3 - 3.5 * y^2; g y = y/13
38.0384615384615 :: Double
```

Двумерный синтаксис языка Haskell позволяет в программах позволяет сделать код программы более удобочитаемым.

```
abcFormula' :: Float -> Float -> Float -> [Float]
abcFormula' a b c  = [(-b+d)/n, (-b-d)/n]
    where
      d = sqrt (b*b-4.0*a*c)
      n = 2.0*a
```

Правило двумерного синтаксиса в языке Haskell заключается в следующем:

1. если строка программы написана с отступом вправо от предыдущей, то она считается её продолжением;

2. если две строки начинаются с одной и той же позиции, то это две самостоятельные команды на одном уровне вложенности.

В последнем примере определения `d` и `n` начинаются с одной позиции. Следовательно это две независимые команды вложенные в конструкцию `where` (так как строка `where` начинается с позиции левее).

Конструкция `let` подобна конструкции с этим названием в языке Standard ML. Но, благодаря двумерному синтаксису необходимость в слове `end` в Haskell отпала.

```
abcFormula'' :: Float -> Float -> Float -> [Float]
abcFormula'' a b c  =
    let
      d = sqrt (b*b-4.0*a*c)
      n = 2.0*a
    in
      [(-b+d)/n, (-b-d)/n]
```

3.3 Охранные выражения

Функцию нахождения знака числа можно определить следующим образом:

```
sign x | x >  0 =  1
       | x == 0 =  0
       | x <  0 = -1
```

Определения различных случаев «охраняются» булевыми выражениями, которые называются охранными выражениями. При вычислении такой функции охранные выражения проверяются последовательно одно за другим до тех пор, пока одно из них не примет значение `True`. В этом случае выражение, стоящее справа от знака `=` и будет являться значением функции. Последнее в списке охранное выражение можно заменить на величину `True` или на специальную константу `otherwise`, например,

```
sign' x | x >  0    =  1
        | x == 0    =  0
        | otherwise = -1
```

3.4 Сопоставление с образцом

Сопоставление фактических параметров при вызове функции с образцами функции в Haskell происходит так же, как и в Standard ML. Так же как и в Standard ML в образцах может применяться джокер `_`. Отличие применения образцов Standard ML от образцов Haskell — применение символа `@` вместо ключевого слова `as` при определении формальных параметров.

```
f s@(x:_) = x:s
```

```
Main> f [5, 7, 9, 15]
[5,5,7,9,15] :: [Integer]
```

Описания нескольких определений функции в зависимости от образца производятся независимо друг от друга (наподобие перегрузки функций).

```
member' _ []    = False
member' x (y:z) = x==y || (member' x z)
```

3.5 Определение полиморфных функций

Распространенной практикой является обозначение одним и тем же именем функций, одинаковых по сути, но применяемых к величинам различного типа. Так, оператор `+` может применяться как к целым числам (типа `Int` или `Integer`), так и к числам с плавающей точкой (`Float`). Результат этой операции имеет тот же тип, что и параметры операции, например, `Int -> Int -> Int` или `Float -> Float -> Float`. Но нельзя считать `+` действительно полиморфным оператором. Если бы тип его был `a -> a -> a`, то это означало бы возможность складывать, например, логические величины или функции, что невозможно. Функции, являющиеся «ограниченно полиморфными» называются перегруженными (overloaded) функциями.

Для того чтобы иметь возможность указывать тип перегруженных функций, все типы разделяют на классы. Класс — это множество типов с некоторыми общими характеристиками. Среди классов языка Haskell можно выделить

- Num — класс, элементы которого могут складываться, вычитаться, умножаться и делиться друг на друга (числовые типы);
- Fractional — подкласс класса Num, содержащий все нецелые типы;
- Floating — подкласс класса Fractional, содержащий все типы с плавающей точкой;
- Integral — подкласс класса Num, содержащий все целые типы данных;
- Ord — класс, элементы которого могут быть упорядочены (упорядоченные типы);
- Eq — класс, элементы которого допускают проверку на равенство (сравниваемые типы).

Оператор `+` имеет тип `Num a=> a -> a -> a`. Эта запись читается так: «Оператор `+` имеет тип `a -> a -> a`, где `a` из класса `Num`».

Примеры.

```
smallBig :: Ord a => a -> a -> (a, a)
smallBig x y | x <= y    = (x, y)
             | otherwise = (y, x)

square :: Num a =>  a -> a
square x = x*x
```

3.6 Неименованные функции

Общий формат неименованной функции:

```
\ образец -> выражение
```

Обычно неименованные функции используют, если требуется одну функцию передать в качестве аргумента другой функции, например,

```
Hugs> map (\ x -> x*x+3*x+1) [1 .. 10]
[5,11,19,29,41,55,71,89,109,131] :: [Integer]
```

Еще неименованные функции используют в функциях более высокого порядка, возвращающих функциональный результат. Например определим функцию, возвращающую производную данной.

```
derivation :: Fractional a => (a -> a) -> (a -> a)
derivation f =
     \ x -> (f (x+dx) - (f x))/ dx
        where dx = 0.0000001

f' = derivation (\x -> 2*x^^2 + 3*x -4)

Main> f' 3
15.0000001752915 :: Double
Main> (derivation (\x -> x^3)) 2
12.0000005843224 :: Double
Main> (derivation log) 2
0.499999986969257 :: Double
Main> (derivation sin) pi
-0.999999998363419 :: Double
```