Лабораторная работа №1

1 Цель работы

Приобрести навыки работы с интерпретатором языка Haskell. Получить представление об основных типах языка Haskell. Научиться определять простейшие функции.

2 Основы работы с интерпретатором Hugs

Для выполнения лабораторных работ будет использоваться интерпретатор языка Haskell. Существует несколько реализаций интерпретатора; в настоящем курсе будет использоваться интерпретатор Hugs (это название является аббревиатурой слов «Haskell users' Gofer system», Gofer — название языка программирования, который был одним из предшественников Haskell.)

После запуска интерпретатора Hugs на экране появляется диалоговое окно среды разработчика, автоматически загружается специальный файл предопределений типов и определений стандартных функций на языке Haskell (Prelude.hs), и выводится стандартное приглашение к работе. Это приглашение имеет вид `Prelude>`; вообще, перед символом `>` выводится имя последнего загруженного модуля.

После вывода приглашения можно вводить выражения языка Haskell, либо команды интерпретатора. Команды интерпретатора отличаются от выражений языка Haskell тем, что начинаются с символа двоеточия (`:`). Примером команды интерпретатора является команда `:quit`, по которой происходит завершение работы интерпретатора. Команды интерпретатора можно сокращать до одной буквы; таким образом, команды `:quit` и `:q` эквивалентны. Команда `:set` используется для того, чтобы установить различные опции интерпретатора. Команда `:?` выводит список доступных команд интерпретатора. В дальнейшем мы рассмотрим другие команды.

3 Типы

Программы на языке Haskell представляют собой *выражения*, вычисление которых приводит к *значениям*. Каждое значение имеет *тип*. Интуитивно тип можно понимать просто как множество допустимых значений выражения. Для того, чтобы узнать тип некоторого выражения, можно использовать команду интерпретатора `:type` (или `:t`). Кроме того, можно выполнить команду `:set +t`, для того, чтобы интерпретатор автоматически печатал тип каждого вычисленного результата.

Основными типами языка Haskell являются:

- Типы `Integer` и `Int` используется для представления целых чисел, причем значения типа `Integer` не ограничены по длине.
- Типы `Float` и `Double` используется для представления вещественных чисел.
- Тип `Bool` содержит два значения: `True` и `False`, и предназначен для представления результата логических выражений.
- Тип `Char` используется для представления символов.

Имена типов в языке Haskell всегда начинаются с заглавной буквы.

Язык Haskell является *сильно типизированным* языком программирования. Тем не менее в большинстве случаев программист не обязан объявлять, каким типам принадлежат вводимые им переменные. Интерпретатор сам способен *вывести* типы употребляемых пользователем переменных. Однако, если все же для каких-либо целей необходимо объявить, что некоторое значение принадлежит некоторому типу, используется конструкция вида: `переменная :: Тип`. Если включена опция интерпретатора `+t`, он печатает значения в таком же формате.

Ниже приведен пример протокола сессии работы с интерпретатором. Предполагается, что текст, следующий за приглашением `Prelude>`, вводит пользователь, а следующий за этим текст представляет ответ системы.

```
Prelude>:set +t
Prelude>1
1 :: Integer
Prelude>1.2
1.2 :: Double
Prelude>'a'
'a' :: Char
Prelude>True
True :: Bool
```

Из данного протокола можно сделать вывод, что значения типа `Integer`, `Double` и `Char` задаются по тем же правилам, что и в языке Си.

Развитая система типов и строгая типизация делают программы на языке Haskell *безопасными по типам*. Гарантируется, что в правильной программе на языке Haskell все типы используются правильно. С практической точки зрения это означает, что программа на языке Haskell при выполнении не может вызвать ошибок доступа к памяти (Access violation). Также гарантируется, что в программе не может произойти использование неинициализированных переменных. Таким образом, многие ошибки в программе отслеживаются на этапе ее компиляции, а не выполнения.

4 Арифметика

Интерпретатор Hugs можно использовать для вычисления арифметических выражений. При этом можно использовать операторы `+`, `-`, `*`, `/` (сложение, вычитание, умножение и деление) с обычными правилами приоритета. Кроме того, можно использовать оператор `^` (возведение в степень). Таким образом, сеанс работы может выглядеть следующим образом:

```
Prelude>2*2
4 :: Integer
Prelude>4*5 + 1
21 :: Integer
Prelude>2^3
8 :: Integer
```

Кроме того, можно использовать стандартные математические функции `sqrt` (квадратный корень), `sin`, `cos`, `exp` и т.д. В отличие от многих других языков программирования, в Haskell при вызове функции не обязательно помещать аргумент в скобки. Таким образом, можно просто писать `sqrt 2`, а не `sqrt(2)`. Пример:

```
Prelude>sqrt 2
1.4142135623731 :: Double
Prelude>1 + sqrt 2
2.4142135623731 :: Double
Prelude>sqrt 2 + 1
2.4142135623731 :: Double
Prelude>sqrt (2 + 1)
1.73205080756888 :: Double
```

Из данного примера можно сделать вывод, что вызов функции имеет более высокий приоритет, чем арифметические операции, так что выражение `sqrt 2 + 1` интерпретируется как `(sqrt 2) + 1`, а не `sqrt (2 + 1)`. Для задания точного порядка вычисления следует использовать скобки, как в последнем примере. (В действительности вызов функции имеет более высокий приоритет, чем любой бинарный оператор.)

Также следует заметить, что в отличие от большинства других языков программирования, целочисленные выражения в языке Haskell вычисляются с неограниченным числом разрядов (Попробуйте вычислить выражение `2^5000`.) В отличие от языка Си, где максимально возможное значение типа `int` ограничено разрядностью машины (на современных персональных компьютерах оно равно $2^{31}-1 = 2147483647$), тип `Integer` в языке Haskell может хранить целые числа произвольной длины.

5 Кортежи

Помимо перечисленных выше простых типов, в языке Haskell можно определять значения составных типов. Например, для задания точки на плоскости необходимы два числа, соответствующие ее координатам. В языке Haskell пару можно задать, перечислив компоненты через запятую и взяв их в скобки: `(5,3)`. Компоненты пары не обязательно должны принадлежать одному типу: можно составить пару, первым элементом которой будет строка, а вторым — число и т.д.

В общем случае, если `a` и `b` — некоторые произвольные типы языка Haskell, тип пары, в которой первый элемент принадлежит типу `a`, а второй — типу `b`, обозначается как `(a,b)`. Например, пара `(5,3)` имеет тип `(Integer, Integer)`; пара `(1, 'a')` принадлежит типу `(Integer, Char)`. Можно привести и более сложный пример: пара `((1,'a'),1.2)` принадлежит типу `((Integer,Char),Double)`. Проверьте это с помощью интерпретатора.

Следует обратить внимания, что хотя конструкции вида `(1,2)` и `(Integer,Integer)` выглядят похоже, в языке Haskell они обозначают совершенно разные сущности. Первая является *значением*, в то время как последняя — *типом*.

Для работы с парами в языке Haskell существуют стандартные функции `fst` и `snd`, возвращающие, соответственно, первый и второй элементы пары (названия этих функций происходят от английских слов «first» (первый) и «second» (второй)). Таким образом, их можно использовать следующим образом

```
Prelude>fst (5, True)
```

```
5 :: Integer
Prelude>snd (5, True)
True :: Bool
```

Кроме пар, аналогичным образом можно определять тройки, четверки и т.д. Их типы записываются аналогичным образом.

```
Prelude>(1,2,3)
(1,2,3) :: (Integer,Integer,Integer)
Prelude>(1,2,3,4)
(1,2,3,4) :: (Integer,Integer,Integer,Integer)
```

Такая структура данных называется *кортежем*. В кортеже может хранится фиксированное количество разнородных данных. Функции `fst` и `snd` определены только для пар и не работают для других кортежей. При попытке использовать их, например, для троек, интерпретатор выдает сообщение об ошибке.

Элементом кортежа может быть значение любого типа, в том числе и другой кортеж. Для доступа к элементам кортежей, составленных из пар, может использоваться комбинация функций `fst` и `snd`. Следующий пример демонстрирует извлечение элемента `'a'` из кортежа `(1, ('a', 23.12))`:

```
Prelude>fst (snd (1, ('a', 23.12)))
'a' :: Integer
```

6 Списки

В отличие от кортежей, *список* может хранить произвольное количество элементов. Чтобы задать список в Haskell, необходимо в квадратных скобках перечислить его элементы через запятую. Все эти элементы должны принадлежать одному и тому же типу. Тип списка с элементами, принадлежащими типу `a`, обозначается как `[a]`.

```
Prelude>[1,2]
[1,2] :: [Integer]
Prelude>['1','2','3']
['1','2','3'] :: [Char]
```

В списке может не быть ни одного элемента. Пустой список обозначается как `[]`.

Оператор `:` (двоеточие) используется для добавления элемента в начало списка. Его левым аргументом должен быть элемент, а правым — список:

```
Prelude>1:[2,3]
[1,2,3] :: [Integer]
Prelude>'5':['1','2','3','4','5']
['5','1','2','3','4','5'] :: [Char]
Prelude>False:[]
[False] :: [Bool]
```

С помощью оператора (:) и пустого списка можно построить любой список:

```
Prelude>1:(2:(3:[]))
[1,2,3] :: Integer
```

Оператор (:) ассоциативен вправо, поэтому в приведенном выше выражении можно опустить скобки:

```
Prelude>1:2:3:[]
[1,2,3] :: Integer
```

Элементами списка могут быть любые значения — числа, символы, кортежи, другие списки и т.д.

```
Prelude>[(1,'a'),(2,'b')]
[(1,'a'),(2,'b')] :: [(Integer,Char)]
Prelude>[[1,2],[3,4,5]]
[[1,2],[3,4,5]] :: [[Integer]]
```

Для работы со списками в языке Haskell существует большое количество функций. В данной лабораторной работе рассмотрим только некоторые из них.

- Функция `head` возвращает первый элемент списка.
- Функция `tail` возвращает список без первого элемента.
- Функция `length` возвращает длину списка.

Функции `head` и `tail` определены для непустых списков. При попытке применить их к пустому списку интерпретатор сообщает об ошибке. Примеры работы с указанными функциями:

```
Prelude>head [1,2,3]
1 :: Integer
Prelude>tail [1,2,3]
[2,3] :: [Integer]
Prelude>tail [1]
[] :: Integer
Prelude>length [1,2,3]
3 :: Int
```

Заметьте, что результат функции `length` принадлежит типу `Int`, а не типу `Integer`.

Для соединения (конкатенации) списков в Haskell определен оператор `++`.

```
Prelude>[1,2]++[3,4]
[1,2,3,4] :: Integer
```

7 Строки

Строковые значения в языке Haskell, как и в Си, задаются в двойных кавычках. Они принадлежат типу `String`.

```
Prelude>"hello"
"hello" :: String
```

В действительности строки являются списками символов; таким образом, выражения `"hello"`, `['h','e','l','l','o']` и `'h':'e':'l':'l':'o':[]` означают одно и то же, а тип `String` является синонимом для `[Char]`. Все функции для работы со списками можно использовать при работе со строками:

```
Prelude>head "hello"
'h' :: Char
Prelude>tail "hello"
"ello" :: [Char]
Prelude>length "hello"
5 :: Int
Prelude>"hello" ++ ", world"
"hello, world" :: [Char]
```

Для преобразования числовых значений в строки и наоборот существуют функции `read` и `show`:

```
Prelude>show 1
"1" :: [Char]
Prelude>"Formula " ++ show 1
"Formula 1" :: [Char]
Prelude>1 + read "12"
13 :: Integer
```

Если функция `show` не сможет преобразовать строку в число, она сообщит об ошибке.

8 Функции

До сих пор мы использовали встроенные функции языка Haskell. Теперь пришла пора научиться определять собственные функции. Для этого нам необходимо изучить еще несколько команд интерпретатора (напомним, что эти команды могут быть сокращены до одной буквы):

- Команда `:load` позволяет загрузить в интерпретатор программу на языке Haskell, содержащуюся в указанном файле.
- Команда `:edit` запускает процесс редактирования последнего загруженного файла.
- Команда `:reload` перечитывает последний загруженный файл.

Определения пользовательских функций должны находиться в файле, который нужно загрузить в интерпретатор Hugs с помощью команды `:load`. Для редактирования загруженной программы можно использовать команду `:edit`. Она запускает внешний редактор (по умолчанию это Notepad) для редактирования файла. После завершения сеанса редактирования редактор необходимо закрыть; при этом интерпретатор Hugs перечитает содержимое изменившегося файла. Однако файл можно редактировать и непосредственно из оболочки Windows. В этом случае, для того чтобы интерпретатор смог перечитать файл, необходимо явно вызывать команду `:reload`.

Рассмотрим пример. Создайте в каком-либо каталоге файл `lab1.hs`. Пусть полный путь к этому файлу — `c:\labs\lab1.hs` (это только пример, ваши файлы могут называться по-другому). В интерпретаторе Hugs выполните следующие команды:

```
Prelude>:load "c:\\labs\\lab1.hs"
```

Если загрузка проведена успешно, приглашение интерпретатора меняется на `Main>`. Дело в том, что если не указано имя модуля, считается, что оно равно `Main`.

```
Main>:edit
```

Здесь должно открыться окно редактора, в котором можно вводить текст программы. Введите:

```
x = [1,2,3]
```

Сохраните файл и закройте редактор. Интерпретатор Hugs загрузит файл `c:\labs\lab1.hs` и теперь значение переменной `x` будет определено:

```
Main>x
[1,2,3] :: [Integer]
```

Обратите внимание, что при записи имени файла в аргументе команды `:load` символы \ дублируются. Также, как и в языке Си, в Haskell символ \ служит индикатором начало служебного символа (`'\n'` и т.п.) Для того, чтобы ввести непосредственно символ \, необходимо, как и в Си, экранировать его еще одним символом \.

Теперь можно перейти к определению функций. Создайте, в соответствие с процессом, описанным выше, какой-либо файл и запишите в него следующий текст:

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

Первая строка (`square :: Integer -> Integer`) объявляет, что мы определяем функцию `square`, принимающую параметр типа `Integer` и возвращающую результат типа `Integer`. Вторая строка (`square x = x * x`) является непосредственно определением функции. Функция `square` принимает один аргумент и возвращает его квадрат.

Функции в языке Haskell являются значениями «первого класса». Это означает, что они «равноправны» с такими значениями, как целые и вещественные числа, символы, строки, списки и т.д. Функции можно передавать в качестве аргументов в другие функции, возвращать их из функций и т.п. Как и все значения в языке Haskell, функции имеют тип. Тип функции, принимающей значения типа `a` и возвращающей значения типа `b` обозначается как `a->b`.

Загрузите созданный файл в интерпретатор и выполните следующие команды:

```
Main>:type square
square :: Integer -> Integer
Main>square 2
4 :: Integer
```

Заметим, что в принципе объявление типа функции `square` не являлось необходимым: интерпретатор сам мог вывести необходимую информацию о типе функции из ее определения. Однако, во-первых, выведенный тип был бы более общим, чем `Integer -> Integer`, а во-вторых, явное указание типа функции является «хорошим тоном» при программировании на языке Haskell, поскольку объявление типа служит своего рода документацией к функции и помогает выявлять ошибки программирования.

Имена определяемых пользователем функций и переменных должны начинаться с латинской буквы в нижнем регистре. Остальные символы в имени могут быть прописными или строчными латинскими буквами, цифрами или символами _ и ′ (подчеркивание и апостроф). Таким образом, ниже перечислены примеры правильных имен переменных:

```
var
var1
variableName
variable_name
var′
```

9 Условные выражения

В определении функции в языке Haskell можно использовать условные выражения. Запишем функцию `signum`, вычисляющую знак переданного ей аргумента:

```
signum :: Integer -> Integer
signum x = if x > 0 then 1
           else if x < 0 then -1
                         else 0
```

Условное выражение записывается в виде:

```
if условие then выражение else выражение.
```

Обратите внимание, что хотя по виду это выражение напоминает соответствующий оператор в языке Си или Паскаль, в условном выражении языка Haskell должны присутствовать и `then`-часть и `else`-часть. Выражения в `then`-части и в `else`-части условного оператора должны принадлежать одному типу.

`Условие` в определении условного оператора представляет собой любое выражение типа `Bool`. Примером таких выражений могут служить сравнения. При сравнении можно использовать следующие операторы:

- <, >, <=, >= — эти операторы имеют такой же смысл, как и в языке Си (меньше, больше, меньше или равно, больше или равно).
- == — оператор проверки на равенство.
- /= — оператор проверки на неравенство.

Выражения типа `Bool` можно комбинировать с помощью общепринятых логических операторов `&&` и `||` (И и ИЛИ), и функции отрицания `not`. Примеры допустимых условий:

```
x >= 0 && x <= 10
x > 3 && x /= 10
(x > 10 || x < -10) && not (x == y)
```

Разумеется, можно определять свои функции, возвращающие значения типа `Bool`, и использовать их в качестве условий. Например, можно определить функцию `isPositive`, возвращающую `True`, если ее аргумент неотрицателен и `False` в противном случае:

```
isPositive :: Integer -> Bool
isPositive x = if x > 0 then True else False
```

Теперь функцию `signum` можно определить следующим образом:

```
signum :: Integer -> Integer
signum x = if isPositive x then 1
           else if x < 0 then -1
                         else 0
```

Отметим, что функцию `isPositive` можно определить и проще:

```
isPositive x = x > 0
```

10 Функции многих переменных и порядок определения функций

До сих пор мы определяли функции, принимающие один аргумент. Разумеется, в языке Haskell можно определять функции, принимающие произвольное количество аргументов. Определение функции `add`, принимающей два целых числа и возвращающей их сумму, выглядит следующим образом:

```
add :: Integer -> Integer -> Integer
add x y = x + y
```

Тип функции `add` может выглядеть несколько загадочно. В языке Haskell считается, что операция `->` ассоциативна вправо. Таким образом, тип функции `add` может быть прочитан как `Integer -> (Integer -> Integer)`, т.е. в соответствие с правилом каррирования, результатом применения функции `add` к одному

аргументу будет функция, принимающая один параметр типа `Integer`. Вообще, тип функции, принимающей `n` аргументов, принадлежащих типам `t1`, `t2`, ..., `tn`, и возвращающей результат типа `a`, записывается в виде `t1->t2->...->tn->a`.

Следует сделать еще одно замечание, касающееся порядка определения функций. В предыдущем разделе мы определили две функции, `signum` и `isPositive`, одна из которых использовала для своего определения другую. Возникает вопрос: какая из этих функций должна быть определена раньше? Напрашивается ответ, что определение `isPositive` должно предшествовать определению функции `signum`; однако в действительности в языке Haskell порядок определения функций не имеет значения! Таким образом, функция `isPositive` может быть определена как до, так и после функции `signum`.

11 Задания на лабораторную работу

1. Приведите пример нетривиальных выражений, принадлежащих следующему типу:

   1) `((Char,Integer), String, [Double])`
   2) `[(Double,Bool,(String,Integer))]`
   3) `([Integer],[Double],[(Bool,Char)])`
   4) `[[[(Integer,Bool)]]]`
   5) `(((Char,Char),Char),[String])`
   6) `(([Double],[Bool]),[Integer])`
   7) `[Integer, (Integer,[Bool])]`
   8) `(Bool,([Bool],[Integer]))`
   9) `[([Bool],[Double])]`
   10) `[([Integer],[Char])]`

   Требование нетривиальности в данном случае означает, что встречающиеся в выражениях списки должны содержать больше одного элемента.

2. Определите следующие функции:

   1) Функция `max3`, по трем целым возвращающая наибольшее из них.

2) Функция `min3`, по трем целым возвращающая наименьшее из них.

3) Функция `sort2`, по двум целым возвращающая пару, в которой наименьшее из них стоит на первом месте, а наибольшее — на втором.

4) Функция `bothTrue :: Bool -> Bool -> Bool`, которая возвращает `True` тогда и только тогда, когда оба ее аргумента будут равны `True`. Не используйте при определении функции стандартные логический операции (`&&`, `||` и т.п.).

5) Функция `solve2::Double->Double->(Bool,Double)`, которая по двум числам, представляющим собой коэффициенты линейного уравнения $ax + b = 0$, возвращает пару, первый элемент которой равен `True`, если решение существует и `False` в противном случае; при этом второй элемент равен либо значению корня, либо `0.0`.

6) Функция `isParallel`, возвращающая `True`, если два отрезка, концы которых задаются в аргументах функции, параллельны (или лежат на одной прямой). Например, значение выражения `isParallel (1,1) (2,2) (2,0) (4,2)` должно быть равно `True`, поскольку отрезки $(1,1) - (2,2)$ и $(2,0) - (4,2)$ параллельны.

7) Функция `isIncluded`, аргументами которой служат параметры двух окружностей на плоскости (координаты центров и радиусы); функция возвращает `True`, если вторая окружность целиком содержится внутри первой.

8) Функция `isRectangular`, принимающая в качестве параметров координаты трех точек на плоскости, и возвращающая `True`, если образуемый ими треугольник — прямоугольный.

9) Функция `isTriangle`, определяющая, можно ли их отрезков с заданными длинами `x`, `y` и `z` построить треугольник.

10) Функция `isSorted`, принимающая на вход три числа и возвращающая `True`, если они упорядочены по возрастанию или по убыванию.

12 Порядок выполнения лабораторной работы

Лабораторная работа состоит из двух заданий, номера которых соответствуют номеру варианта бригады. Номер варианта равен порядковому

номеру бригады в списке (возможно, по модулю 10, если число бригад больше 10). Для того, чтобы сдать лабораторную работу, каждой бригаде необходимо:

1. Получить задание в соответствие с вариантом;
2. Выполнить его (самостоятельно);
3. Продемонстрировать преподавателю на компьютере;
4. Оформить и сдать отчет;
5. Защитить работу.

В отчете *обязательно* должны быть приведены:

1. Номер варианта;
2. Текст заданий;
3. При необходимости — пояснения к реализации;
4. Текст программы;
5. Результаты тестов.

Защита лабораторной работы состоит в беседе с преподавателем по материалам представленного отчета и ответе студентов на контрольные вопросы.

13 Контрольные вопросы

1. В чем отличие команд интерпретатора от выражений языка Haskell?
2. Основные типы языка Haskell.
3. Функции для работы с кортежами.
4. Функции для работы со списками.
5. Допустимые имена переменных и функций.
6. Команды интерпретатора для работы с файлами программ.
7. Условные выражения в языке Haskell.
8. Определение функций в языке Haskell.