Лабораторная работа №6

1 Модули

Программы на языке Haskell состоят из набора *модулей*. Модули служат двум целям — управлению пространствами имен и созданию абстрактных типов данных.

Модули имеют имена, начинающиеся с заглавной буквы; в интерпретаторе Hugs текст модуля должен находиться в отдельном файле, имя которого совпадает с именем модуля. Этот файл должен иметь расширение `.hs`.

С практической точки зрения модуль представляет собой просто одно большое объявление, начинающееся с ключевого слова `module`. Приведем пример модуля с именем `Tree`.

```
module Tree ( Tree(Leaf,Branch), leafList) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)

leafList (Leaf x)            =  [x]
leafList (Branch left right) =  leafList left ++
                                leafList right
```

Тип `Tree` и функция `leafList` были введены в лабораторной работе №4.

Модуль явно *экспортирует* `Tree`, `Leaf`, `Branch` и `leafList`. Экспортируемые из модуля имена перечисляются в скобках после ключевого слова `module`. Если это перечисление не указано, по умолчанию из модуля экспортируются *все* имена. Заметьте, что имена типа и его конструкторов должны быть сгруппированы, как в конструкции `Tree(Leaf,Branch)`. В качестве сокращения можно использовать запись `Tree(..)`. Также возможно экспортировать только часть конструкторов данных.

Модуль `Tree` теперь может быть *импортирован* в какой-либо другой модуль:

```
module Main where
import Tree (Tree(Leaf,Branch), leafList)

...
```

Здесь мы явно указали список импортируемых сущностей; если опустить его, импортируются все сущности, экспортируемые из модуля.

Очевидно, если в двух импортируемых модулях содержатся различные сущности с одним именем, возникнет проблема. Для того, чтобы избежать ее в языке существует ключевое слово `qualified`, при помощи которого определяются те импортируемые модули, имена объектов которых приобретают вид: «Модуль.Объект». Например, для модуля `Tree`:

```
module Main where
import qualified Tree

leafList = Tree.leafList
```

2 Абстрактные типы данных

Использование модулей позволяет определять абстрактные типы данных, т. е. типы, внутренняя структура которых скрыта от их пользователя. Например, рассмотрим простейший словарь, по заданному слову возвращающий его значение:

```
module Dictionary where
data Dictionary = Dictionary [(String,String)]

getMeaning :: Dictionary -> String -> Maybe String
getMeaning [] _ = Nothing
getMeaning ((word,meaning):xs) w | w == word = Just meaning
                                 | otherwise = Nothing
```

Функция `getMeaning` по заданному словарю и слову возвращает найденное значение (с использованием типа `Maybe`). Сам словарь представляется списком пар.

Как создавать словарь? Пользователь этого модуля может определить функцию `addWord`, которая добавляет пару «слово-значение» в словарь и возвращает модифицированный словарь:

```
import Dictionary
addWord (Dictionary dict) word meaning = Dictionary ((word,meaning):dict)
```

Здесь пользователь видит представление словаря в виде списка и может воспользоваться этим. Однако в дальнейшем мы можем захотеть изменить представление словаря. Список — довольно неэффективная структура данных для поиска, если он становится велик. Гораздо лучше использовать хеш-таблицы или деревья поиска. Однако, если представление типа `Dictionary` открыто, мы не можем изменить его без риска нарушить функционирование пользовательских программ.

Сделаем тип `Dictionary` абстрактным, чтобы скрыть от пользователей модуля его внутреннее представление. Определим в модуле значение `emptyDict`, представляющее собой пустой словарь и функцию `addWord`. Тогда пользователи смогут общаться с значениями типа `Dictionary` только с помощью разрешенных функций:

```
module Dictionary (Dictionary, getMeaning, addWord, emptyDict) where
data Dictionary = Dictionary [(String,String)]

getMeaning :: Dictionary -> String -> Maybe String
getMeaning [] _ = Nothing
getMeaning ((word,meaning):xs) w | w == word = Just meaning
                                 | otherwise = Nothing

addWord (Dictionary dict) word meaning = Dictionary ((word,meaning):dict)

emptyDict = Dictionary []
```

Абстрактные типы данных предоставляют механизм сокрытия данных, который на языке объектно-ориентированного программирования называется инкапсуляцией.

3 Операции ввода-вывода

Система ввода-вывода в языке Haskell полностью функциональна, однако не уступает в возможностях системам ввода-вывода императивных языков. В императивных языках программа представляет собой последовательность *действий*, которые считывают и изменяют содержимое окружающего мира. Типичными действиями является считывание и установка глобальных переменных, запись в файл, считывание с клавиатуры и т. д. Такие действия также являются и частью Haskell, однако они четко отделены от чисто функционального ядра языка.

Система ввода-вывода в языке Haskell построена вокруг концепции *монад*. Однако для программирования ввода-вывода понимание монад

требуется не больше, чем понимание общей алгебры для выполнения простых арифметических действий. Поэтому мы рассмотрим систему ввода-вывода без привязки к монадам, которые будут изучаться в лекциях.

С помощью языка Haskell действия определяются, а не выполняются. Определение действия не означает, что оно выполняется. Выполнение действия происходит за рамками вычислений выражений.

Действия являются либо атомарными, определенными с помощью системных примитивов, либо последовательными композициями других действий. Монада ввода-вывода содержит примитивы, которые позволяют создавать составные действия, аналогично использованию ';' в императивных языках. Монада служит «клеем», связывающим действия в программе.

3.1 Базовые операции ввода-вывода

Каждое действие возвращает значение. В системе типов это значение «помечено» типом `IO`, который отличает действия от других значений. Например, рассмотрим функцию `getChar`:

```
getChar :: IO Char
```

`IO Char` показывает, что `getChar` при вызове выполняет некоторое действие, которое возвращает символ. Действия, которые не возвращают результата, используют тип `IO ()`. Символ `()` означает пустой тип (похожий на тип `void` в языке Си). Например, функция `putChar`:

```
putChar :: Char -> IO ()
```

Она принимает символ и не возвращает ничего интересного.

Действия связываются друг с другом с помощью оператора `>>=`. Однако мы будем использовать т. н. do-нотацию. Ключевое слово `do` начинает последовательность операторов, которые выполняются по порядку. Оператор может быть либо действием, либо образцом, связываемым с результатом действия с помощью `<-`. do-нотация использует те же правила выравнивания, что и ключевые слова `let` или `where`. Вот простая программа, которая считывает символ и печатает его:

```
main :: IO ()
main = do c <- getChar
          putChar c
```

Использование имени `main` здесь неслучайно: функция `main` модуля `Main` является точкой входа в программу на языке Haskell, подобно функции

`main` в Си. Ее тип должен быть `IO ()`. Представленная программа выполняет два действия последовательно: считывает символ, связывает результат с переменной `c` и затем печатает символ.

Как вернуть значение из последовательности действий? Например, нам необходимо определить функцию `ready`, которая считывает символ и возвращает `True`, если он равен `'y'`:

```
ready :: IO Bool
ready = do c <- getChar
           c == 'y' -- Ошибка!!!
```

Это не работает, поскольку второй оператор в `do` является просто булевским значением, а не действием. Нам нужно взять булевское значение и создать действие, которое ничего не делает, но возвращает это булевское значение в качестве результата. Для этого служит функция `return`:

```
return :: a -> IO a
```

Функция `return` завершает последовательность действий. Таким образом, `ready` определяется так:

```
ready :: IO Bool
ready = do c <- getChar
           return (c == 'y')
```

Теперь можно определить более сложные функции ввода-вывода. Функция `getLine`, возвращающая строку, считанную с клавиатуры с символом конца строки в качестве завершающего:

```
getLine :: IO String
getLine = do c <- getChar
             if c == '\n'
                then return ""
                else do l <- getLine
                        return (c:l)
```

Функция `return` вводит обычное значение в область действий ввода-вывода. Как насчет обратного направления? Можно ли выполнить действие ввода-вывода в обычном выражении? Оказывается, нет! Функция, такая как `f :: Int -> Int` не может выполнять операций ввода-вывода, поскольку `IO` не появляется в типе ее возвращаемого значения.

3.2 Стандартные операции ввода-вывода

Рассмотрим следующие действия и типы для работы с файловым вводом-выводом (они определены в модуле `IO`):

```
type FilePath = String   -- имена файлов в файловой системе
openFile :: FilePath -> IOMode -> IO Handle
hClose :: Handle -> IO ()
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

Чтобы открыть файл, используется функция `openFile`, которой передается имя файла и режим, в котором его необходимо открыть. При этом создается дескриптор файла (типа `Handle`), который затем необходимо закрыть с помощью функции `hClose`.

Для считывания из файла символа и строки служат следующие функции:

```
hGetChar :: Handle -> IO Char
hGetLine :: Handle -> IO String
```

Для записи в файл используются функции:

```
hPutChar :: Handle -> Char -> IO ()
hPutStr  :: Handle -> String -> IO ()
```

Для считывания с клавиатуры и вывода на экран используются следующие функции:

```
getChar :: IO Char
getLine :: IO String
readLn  :: IO Int  --чтение числа целого типа
putChar :: Char -> IO ()
putStr  :: String -> IO ()
```

В пример, приведенном ниже, печатается сумма элементов списка:

```
main :: IO ()
main  = do
      putStr "Enter a list of numbers: "
      listStr <- getLine
      print (sum (read listStr))
```

Результат выполнения:

```
Main> main
Enter a list of numbers: [1,2,3]
6
() :: IO ()
```

Кроме того, очень полезна следующая функция:

```
hGetContents :: Handle -> IO String
```

Она считывает весь файл как одну большую строку. На первый взгляд эта функция очень неэффективна, однако в действительности, из-за использования отложенных вычислений, из файла считается столько символов, сколько необходимо, но не больше.

3.3 Пример

Запишем программу копирования файлов. Она считывает с клавиатуры имена двух файлов, исходного и целевого, и копирует один файл в другой.

```
-- Функция печатает приглашение, считывает имя файла
-- и открывает его в указанном режиме
getAndOpenFile prompt mode = do putStr prompt
                                name <- getLine
                                openFile name mode

main = do fromHandle <- getAndOpenFile "Copy from: " ReadMode
          toHandle <- getAndOpenFile "Copy to" WriteMode
          contents <- hGetContents fromHandle
          hPutStr toHandle contents
          hClose toHandle
          putStr "Done."
```

Несмотря на то, что мы используем функцию `hGetContents`, все содержимое файла не будет находится в памяти, поскольку оно будет прочитываться по мере необходимости и записываться на диск. Это позволит копировать даже большие файлы, объем которых превышает объем оперативной памяти компьютера. Исходный файл будет неявно закрыт, когда из него считается последний символ.

Для доступа к параметрам командной строки программы можно использовать следующую функцию, определенную в модуле `System`:

```
getArgs :: IO [String]
```

Эта функция возвращает список строк, являющихся параметрами командной строки, подобно массиву `argv` в программах на Си. Программу копирования тогда можно определить так:

```
main = do args <- getArgs
          copyFile
          putStr "Done."

copyFile [from, to] = do fromHandle <- openFile from ReadMode
                         toHandle <- openFile to WriteMode
                         contents <- hGetContents fromHandle
                         hPutStr toHandle contents
                         hClose toHandle
copyFile _ = error "Usage: copy <from> <to>"
```

Эта программа принимает имена исходного и целевого файлов из командной строки. Функция `copyFile` печатает сообщение об ошибке, если в программу передано неверное количество аргументов.

4 Создание исполняемых программ

До сих пор мы выполняли программы на языке Haskell с использованием интерпретатора. Однако существует возможность создавать отдельные исполняемые программы, для выполнения которых не нужна среда интерпретатора. Для этого используется компилятор Glasgow Haskell Compiler, вызываемый с помощью команды `ghc`.

Для того, чтобы скомпилировать набор модулей в исполняемую программу, должен быть определен модуль с именем `Main`, в котором необходимо определить функцию `main :: IO ()`. Этот модуль следует поместить в файл `Main.hs`. Для компиляции необходимо ввести в командной строке следующую команду:

```
ghc -o Main.exe Main.hs
```

В случае, если программа содержит ошибки, информация о них будет выведена на экран. Если ошибок нет, компилятор создаст исполняемый файл, который можно запускать на выполнение. Результатом работы программы

```
--Main.hs
      main :: IO () main  = do
      print("Enter first number:")
```

```
    num1Str <- getLine
    print("Enter second number: ")
    num2 <- readLn
    print("Sum=")
    print( read num1Str+num2)
```

будет:

```
C:\ghc\ghc-6.2\bin>Main.exe
"Enter first number:"
4
"Enter second number: "
3
"Sum="
7
```

5 Задания

1. Напишите следующие программы:

    1) Программа, считывающая два числа и возвращающая их сумму.
    2) Программа, распечатывающая переданные в нее аргументы командной строки.
    3) Программа, которая принимает в командной строке имя файла и распечатывает его на экране.
    4) Программа, принимающая в командной строке число `n` и имя файла и выводящая на экран первые `n` строк файла (используйте функцию `lines`, разбивающую строку на список строк в символах конца строки, т.е., например `lines"line1\nline2"` вернет `["line1", "line2"]`. Также полезна функция `unlines`, осуществляющая обратную операцию.)

2. Реализуйте программы, выполняющие задания вашего варианта из первой лабораторной работы. Параметры функций должны считываться с клавиатуры.