Лабораторная работа №5

1 Определение операторов

Бинарные операторы, такие как +, - и т. п. в языке Haskell являются такими же функциями, как и все остальные, за тем исключением, что для их вызова можно использовать инфиксную нотацию. Если взять бинарный оператор в скобки, то для его вызова можно использовать префиксную нотацию и обращаться с ним, как с обычной функцией. Так, следующие пары записей эквивалентны:

```
2 + 2
(+) 2 2

x < y
(<) x y

x /= y
(/=) x y
```

Наоборот, любую функцию, принимающую два аргумента, можно использовать в инфиксном стиле. Для этого ее имя нужно окружить обратными кавычками (символ ‘). Например, если определить функцию:

```
func x y = (x + y) / (x - y)
```

то ее можно вызывать в следующих видах:

```
func 5 2
5 ‘func‘ 2
```

Далее, если в имени функции встречаются только «символы» (не буквы и не цифры), то она автоматически считается инфиксным оператором. При определении ее имя нужно заключать в скобки. Например, определим оператор «приближенно равно», проверяющий, что числа отличаются не более, чем на 0.001:

```
(~=) x y = abs (x - y) < 0.001
```

Теперь этот оператор можно использовать так же, как и все остальные:

```
testApproxEqual x y = if x ~= y then "equal"
                                else "not equal"
```

2 Рекурсивные типы

При определении типов данных в правой части определения можно использовать определяемый этой конструкцией тип. Это дает возможность определять рекурсивные структуры данных. Одной из основных таких структур является дерево.

Определим бинарное дерево, в листьях которого находятся элементы типа `a`, следующим образом:

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

Branch :: Tree a->Tree a->Tree a
Leaf ::a->Tree a
```

Это определение говорит, что дерево (Tree) является либо листом (Leaf), т. е. узлом, у которого нет потомков, либо ветвью (Branch), т. е. узлом, у которого есть левое и правое поддерево. Заметьте, что в приведенном определении `Leaf` и `Branch` — конструкторы данных, а `Tree a`, встречающееся и в левой, и в правой части определения — название типа.

Работа с рекурсивными типами практически не отличается от работы с обычными типами, за тем исключением, что практически все функции, работающие с рекурсивными типами, сами также рекурсивны.

Например, определим функцию `treeSize`, возвращающую количество листьев в дереве. Она записывается следующим образом:

```
treeSize (Leaf _) = 1
treeSize (Branch l r) = treeSize l + treeSize r
```

Применение этой функции выглядит следующим образом:

```
Main>treeSize (Branch (Branch (Leaf 1) (Leaf 2)) (Leaf 3))
3
```

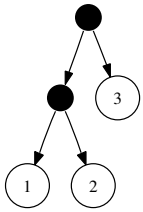

Рис. 1: Пример дерева

Здесь мы применили ее для дерева следующего вида (см. рис. 1.)

Другим примером функции для работы с деревьями служит функция для получения списка всех листьев дерева:

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

3 Списки как рекурсивные типы

Список также является рекурсивным типом. Рассмотрим следующий полиморфный тип:

```
data List a = Nil | Cons a (List a)
```

Значение типа `List a` либо пусто (`Nil`), либо содержит элемент типа `a` и значение типа `List a`. Нетрудно заметить прямую аналогию со списками, которые также либо пусты (`[]`), либо содержат голову типа `a` и хвост, являющийся также списком. Сходство станет еще очевиднее, если конструктор `Cons` записать в инфиксном виде:

```
data List a = Nil | a `Cons` (List a)
```

Таким образом, списочный тип мог бы быть определен следующим образом:

```
-- Это не настоящий код языка Haskell
data [a] = [] | a : [a]
```

Для значений типа `List` можно определить все функции, определенные для списков. Приведем примеры функций `head`, `tail` и `map`:

```
headList (Cons x _) = x
headList Nil        = error "headList: empty list"

tailList (Cons _ y) = y
tailList Nil        = error "tailList: empty list"
```

Проиллюстрируем работу этих функций:

```
Main>headList (Cons 1 (Cons 2 Nil))
1
Main>tailList (Cons 1 (Cons 2 Nil))
Cons 2 Nil
```

4 Синтаксические деревья

Структуры, подобные древесной, широко используются в программировании. Например, результатом грамматического разбора программы в любом компиляторе является синтаксическое дерево. Приведем пример такого дерева для выражений, содержащих константы, символы сложения и умножения:

```
data Expr =   Const Integer
            | Add Expr Expr
            | Mult Expr Expr
```

Из этого определения видно, что выражение (Expression) является либо целочисленной константой (Constant), либо суммой или произведением двух выражений. Например, для выражения $1+2*(3+4)$ соответствующее значение типа `Expr` имеет вид:

```
Add (Const 1) (Mult (Const 2) (Add (Const 3) (Const 4)))
```

Функцию вычисления значения выражения можно определить следующим образом:

```
eval :: Expr -> Integer
eval (Const x) = x
eval (Add x y) = eval x + eval y
eval (Mult x y) = eval x * eval y
```

Можно расширить тип `Expr`, введя возможность использования переменных в выражениях:

```
data Expr =   Const Integer
            | Var String
            | Add Expr Expr
            | Mult Expr Expr
```

Конструктор `Var` определяет переменную с указанным именем. Такой тип `Expr` позволяет определить, например, функцию для дифференцирования выражения:

```
diff :: Expr -> Expr
diff (Const _) = Const 0
diff (Var x) = Const 1
diff (Add x y) = Add (diff x) (diff y)
diff (Mult x y) = Add (Mult (diff x) y) (Mult x (diff y))
```

Проверим работу этой функции на примере дифференцирования выражения $x+x^2$ (не забудьте добавить `deriving(Show)` после определения типа `Expr`):

```
Main>diff (Add (Var "x") (Mult (Var "x") (Var "x")))
Add (Const 1) (Add (Mult (Const 1) (Var "x"))
 (Mult (Var "x") (Const 1)))
```

Таким образом, в результате дифференцирования мы получили выражение $1+(1\cdot x+x\cdot 1)$, которое является правильным, но, конечно, нуждается в упрощении.

Другим ограничением функции `diff` является то, что она не различает, по какой переменной производится дифференцирование. Соответственно, в реальности она должна принимать дополнительный параметр — имя переменной дифференцирования.

Задание значений типа `Expr` напрямую довольно неудобно. В принципе, можно написать функцию, которая преобразует строку вида `"1+x*y"` в соответствующее значение типа `Expr`. Однако написание такой функции довольно трудоемко, поэтому студентам предлагается воспользоваться готовой функцией. Она определена в файле `expr.hs` и называется `parseExpr`. В этом же файле определен тип `Expr`. Для того, чтобы подключить этот файл, скопируйте его в каталог, где находится ваша программа и в ее начале добавьте строку

```
import Expr
```

(!!!Этот файл откроется, если стоит интерпретатор 'hugs98-Nov2003-2.msi' и запушен WinHugs (Hugs mode)!!!)

Функция `parseExpr` имеет следующий тип:

```
parseExpr :: String -> Expr
```

По заданной строке она возвращает ее представление в виде значения типа `Expr`:

```
Main>parseExpr "1+x"
Add (Const 1) (Var "x")
```

5 Задания

1. Функции для работы с типом `List`. Для введенного ранее типа `List` определите следующие функции:

   1) `lengthList`, возвращающую длину списка типа `List`.
   2) `nthList`, возвращающую `n`-й элемент списка.
   3) `removeNegative`, которая из списка целых (тип `List Integer`) удаляет отрицательные элементы.
   4) `fromList`, преобразующую список типа `List` в обычный список.
   5) `toList`, преобразующую обычный список в список типа `List`.

2. Функции работы с бинарными деревьями поиска. Определите тип данных, представляющий бинарные деревья поиска. В отличие от деревьев, представленных в методических указаниях, в деревьях поиска данные могут находиться не только в листьях, но и в промежуточных узлах дерева. Будем использовать деревья для представления ассоциативного массива, сопоставляющие значения *ключей* (представляемых как строки) целым числам. Для каждого узла с некоторым ключом в левом поддереве должны содержаться элементы с меньшими значениями ключа, а в правом — с бóльшими. При поиске соответствия между строкой и числом необходимо учитывать эту информацию, поскольку она позволяет более эффективно извлекать информацию из дерева. Определите описанный тип данных и следующие функции:

   1) `add`, добавляющую в дерево заданную пару ключа и значения.
   2) `find`, возвращающую число, соответствующее заданной строке.
   3) `exists`, проверяющую, что элемент с заданным ключом содержится в дереве.

4) `toList`, преобразующая заданное дерево поиска в список, упорядоченный по значениям ключей.

3. Разработать тип данных, представляющий содержимое каталога файловой системы. Считаем, что каждый файл либо содержит некоторые данные, либо является каталогом. Каталог включает в себя другие файлы (которые, в свою очередь могут быть каталогами) вместе с их именами и размерами в байтах. В данной работе содержимое файлов можно игнорировать: тип данных должен представлять только их имена, размеры и структуру каталогов. Определите следующие функции:

   1) `dirAll`, возвращающую список полных имен всех файлов каталога, включая подкаталоги.
   2) `find`, возвращающая путь, ведущий к файлу с заданным именем. Например, если каталог содерит файлы `a`, `b` и `c`, и `b` является каталогом, содержащим `x` и `y`, тогда функция поиска для x должна вернуть строку `"b/x"`.
   3) `du`, для заданного каталога возвращающая количество байт, занимаемых его файлами (включая файлы в подкаталогах).

4. Утверждением будем называть логическую формулу, имеющую одну из следующих форм:

   - имя переменной (строка)
   - `p & q`
   - `p | q`
   - `~p`

   где `p` и `q` — утверждения. Например, утверждениями являются следующие формулы:

   - `x`
   - `x | y`
   - `x & (x | ~y)`

   Разработайте тип данных Prop, представляющий утверждения такого вида. Определите следующие функции:

   1) `vars :: Prop -> [String]`, которая возвращает список имен переменных (без повторений), встречающихся в утверждениях.

2) Пусть задан список имен переменных и их значений типа `Bool`, например `[("x",True),("y",False)]`. Определите функцию `truthValue :: Prop -> [(String,Bool)] -> Bool`, которая определяет, верно ли утверждение, если переменные имеют заданные списком значения.

3) Определите функцию `tautology :: Prop -> Bool`, которая возвращает True, если утверждение верно при любых значениях переменных, встречающихся в нем (например, это выполняется для утверждения `(x | ~x)`).

5. Лексические деревья (trie-деревья) используются для представления словарей. Каждый узел дерева содержит следующую информацию: символ, булевское значение и список поддеревьев (у каждого узла может быть произвольное количество дочерних деревьев). Пример trie-дерева приведен на рис. 2. Булевское значение, равное

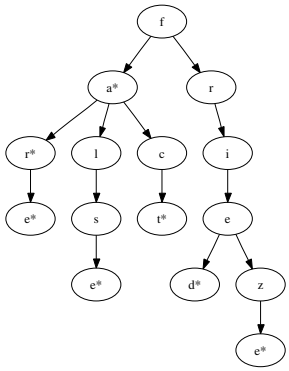

Рис. 2: trie-дерево

`True` отмечает конец слова, читаемого, начиная с корня дерева. На рисунке узлы с такими значениями помечены символом `*`. Таким образом, в дереве представлены слова fa, false, far, fare, fact, fried, frieze. Определите следующие функции:

- `exists`, которая проверяет, что заданное слово содержится в trie-дереве.
- `insert`, которая по дереву и слову возвращает новое дерево, в которое включено это слово. Если слово уже содержится в дереве, оно должно возвращаться без изменений.
- `completions`, которая по заданной строке возвращает список всех слов из дерева, началом которых служит указанная стро-

ка (например, для приведенного на рис. 2 дерева по строке `"fri"` должен возвращаться список `["fried","frieze"]`.)

6. Теоретически возможно, хотя и неэффективно, определить целые числа с помощью рекурсивных типов данных следующим образом:

   ```
   data Number = Zero | Next Number
   ```

   Т. е. число является либо нулем (`Zero`), либо определяется, как число, следующее за предыдущим числом. Например, число 3 записывается как `Next (Next (Next Zero))`. Определите для такого представления следующие функции:

   1) `fromInt`, для заданного целого числа типа `Integer` возвращающую соответствующее ему значение типа `Number`.
   2) `toInt`, преобразующую значение типа `Number` в соответствующее целое число.
   3) `plus :: Number -> Number -> Number`, складывающую свои аргументы.
   4) `mult :: Number -> Number -> Number`, умножающую свои аргументы.
   5) `dec`, вычитающую единицу из своего аргумента. Для `Zero` функция должна возвращать `Zero`.
   6) `fact`, вычисляющую факториал.

7. Иерархия должностей в некоторой организации образует древовидную структуру. Каждый работник, однозначно характеризующийся уникальным именем, имеет несколько подчиненных. Определите тип данных, представляющий такую иерархию и опишите следующие функции:

   1) `getSubordinate`, возвращающую список подчиненных указанного работника.
   2) `getAllSubordinate`, возвращающую список всех подчиненных данного работника, включая косвенных.
   3) `getBoss`, возвращающую начальника указанного работника.
   4) `getList`, возвращающую список пар, первым элементом которых является имя работника, а вторым — количество его подчиненных (включая косвенных).

8. Область на плоскости является либо прямоугольником, либо кругом, либо объединением областей, либо их пересечением. Прямоугольник характеризуется координатами левого нижнего и правого верхнего углов, круг — координатами центра и радиусом. Разработайте структуру данных, представляющую область описанного вида. Определите следующие функции:

    1) `contains`, проверяющая, что заданная точка попадает в область.
    2) `isRectangular`, проверяющая, что область задается только прямоугольниками.
    3) `isEmpty`, проверяющая, что область пуста, т. е. ни одна точка плоскости не попадает в нее.

9. Класс в объектно-ориентированном языке содержит набор методов (в данной работе будем игнорировать поля-данные класса). Кроме того, он может иметь единственный родительский класс (не рассматриваем случай множественного наследования). Однако существуют классы и без родителей. При наследовании класса методы предка добавляются к методам потомка. Определите тип данных, представляющий информацию об иерархии классов. Опишите следующие функции:

    1) `getParent`, возвращающую непосредственного предка класса с указанным именем.
    2) `getPath`, возвращающую список всех предков данного класса (непосредственный предок, предок предка и т. д.)
    3) `getMethods`, возвращающую список методов указанного класса с учетом наследования.
    4) `inherit`, добавляющая в иерархию классов класс с заданным именем, унаследованый от указанного предка.

10. Работа с типом `Expr`. Используя тип `Expr`, определенный выше, реализуйте следующие функции (используйте для тестирования функцию `parseExpr`)

    1) Определите корректную функцию `diff`, которая принимает в качестве дополнительного аргумента имя переменной, по которой необходимо осуществлять дифференцирование.
    2) Определите функцию `simplify`, которая упрощает выражения типа `Expr`, применяя очевидные правила вида:

- $x + 0 = 0 + x = x$
- $x \cdot 1 = 1 \cdot x = x$
- $x \cdot 0 = 0 \cdot x = 0$
- и т. д.

3) Определите функцию `toString`, преобразующую выражение типа `Expr` в строку. Например, результатом применения функции к выражению `Add (Mult (Const 2) (Var "x")) (Var "y")` должна быть строка `"2*x+y"`. Учтите возможность использования скобок, например, выражение `Mult (Const 2) (Add (Var "x") (Var "y"))` должно преобразовываться в строку `"2*(x+y)"`

4) Определите функцию `eval`, которая принимает два параметра: выражение типа `Expr` и список пар типа `(String,Integer)`, задающий соответствие имен переменных и их значений. Функция должна вычислять значение выражение с учетом заданных значений выражений. Например, выражение `eval (Add (Var "x") (Var "y")) [("x",1),("y",2)]` должно выдавать число 3.