

Лабораторная работа №3

1 let-связывание

При определении функций часто бывает необходимо использовать некоторые временные переменные для хранения промежуточных результатов. Вспомним, как вычисляются корни квадратного уравнения вида $ax^2 + bx + c = 0$: $x_{1,2} = (-b \pm \sqrt{b^2 - 4ac}) / 2a$. Можно записать следующую функцию для вычисления пары корней уравнения:

```
roots a b c =  
  ((-b + sqrt (b*b - 4*a*c)) / (2*a),  
   (-b - sqrt (b*b - 4*a*c)) / (2*a))
```

Написание функций в таком стиле чревато проблемами. Во-первых, легко допустить ошибку, второй раз записывая одно и то же выражение. Во-вторых, при чтении этой программы приходится сопоставлять два выражения, чтобы понять, что они представляют собой одно и то же. В-третьих, программа становится длиннее. И наконец, она менее эффективна, чем могла бы быть, потому что компьютеру приходится два раза проводить одинаковые вычисления.

Для избежания этих проблем в языке можно вводить локальные переменные. Функцию можно записать так:

```
roots a b c =  
  let det = sqrt (b*b - 4*a*c)  
  in ((-b + det / (2*a),  
      (-b - det / (2*a))
```

Локальная переменная `det` доступна только в определении функции `roots`.

Можно определять несколько локальных переменных:

```
roots a b c =  
  let det = sqrt (b*b - 4*a*c)  
      twice_a = 2*a  
  in ((-b + det) / twice_a,  
      (-b - det) / twice_a)
```

Заметьте, что в конструкции `let ... in ...` используется правило выравнивания: первый символ, отличный от пробела, следующий за ключевым словом `let`, задает колонку, относительно которой должны выравниваться последующие определения. С использованием символов `{`, `}` и `;` правило выравнивания становится необязательным, и функцию `roots` можно было бы записать так:

```
roots a b c =
  let { let = sqrt (b*b - 4*a*c); twice_a = 2*a }
  in ((-b + det) / twice_a,
      (-b - det) / twice_a)
```

Помимо конструкции `let ... in ...` иногда удобнее использовать конструкцию `... where ...`, в которой определения локальных переменных следуют после основной функции:

```
roots a b c =
  ((-b + det) / twice_a,
   (-b - det) / twice_a)
  where det = sqrt (b*b - 4*a*c)
        twice_a = 2*a
```

Заметим, что в принципе можно было не вводить локальных переменных и использовать вместо них глобальные функции:

```
det a b c = sqrt (b*b - 4*a*c)
```

```
twice_a a = 2*a
```

```
roots a b c =
  ((-b + det a b c) / twice_a a,
   (-b - det a b c) / twice_a a)
```

Однако недостатки такого подхода очевидны: помимо того неприятного факта, что мы ввели две вспомогательные функции в глобальном пространстве имен (а это значит, что мы теперь не сможем использовать, например, имя `det` для какой-нибудь другой полезной функции), для вычисления значений $(\sqrt{b^2 - 4ac})$ и $2a$ приходится передавать соответствующие параметры в функции, тогда как локальные определения могут свободно использовать параметры функции, в рамках которой они определены.

В конструкциях `let` и `where` можно определять не только переменные, но и функции. Рассмотрим, например, функцию, возвращающую по заданному числу `n` список натуральных чисел `[1, 2, ..., n]`. Введем вспомогательную

функцию `numsFrom`, которая по заданному числу `m` возвращает список `[m, m+1, m+2, ..., n]` и сделаем его определение локальным:

```
numsTo n =  
    let numsFrom m = if m == n then [m] else m : numsFrom (m + 1)  
    in numsFrom 1
```

Заметьте, что функция `numsFrom` использует в своем определении переменную `n`, хотя она не передается в нее в качестве параметра.

2 Сигнализация об ошибках

Определяемые нами функции могут не вычисляться при некоторых значениях аргумента. Вспомним определение функции факториала:

```
factorial 0 = 1  
factorial n = n * factorial (n - 1)
```

Эта функция замечательно работает до тех пор, пока мы не попытаемся вычислить факториал отрицательного числа. Нетрудно заметить, что в этом случае вычисление уходит в бесконечную рекурсию, поскольку базовый случай никогда не достигается.

Простейший способ сигнализировать о таких ошибках — использовать стандартную функцию `error`. Она принимает в качестве аргумента строку, а ее вычисление приводит к остановке программы и выдаче на экран этой строки. Таким образом, функцию факториала запишем так:

```
factorial 0 = 1  
factorial n = if n > 0 then  
    n * factorial (n - 1)  
    else  
        error "factorial: negative argument"
```

3 Охраняющие условия

Сопоставление с образцом предоставляет широкие возможности в определении функций. Однако с его помощью, по существу, можно выделить только структуру переданных в функцию параметров и равенство элементов

этих параметров константным значениям. Однако зачастую этого недостаточно: необходимо накладывать более сложные условия на входные параметры.

Например, в приведенном выше определении функции `factorial` мы использовали сочетание сопоставления с образцом и условного оператора. Сопоставление с образцом выглядит экономнее и нагляднее. Можно ли использовать похожий синтаксис для условий? Да, при использовании *охраняющих условий*. С ними функция факториала запишется следующим образом:

```
factorial 0 = 1
factorial n | n < 0 = error "factorial: negative argument"
            | n >= 0 = n * factorial (n - 1)
```

Синтаксис очевиден из приведенного примера. Также заметим, что вместо последнего условия можно использовать слово `otherwise` (англ. иначе, в противном случае). Например, функция определения знака числа выглядит следующим образом:

```
signum x | x < 0      = -1
         | x == 0     = 0
         | otherwise = 1
```

Определение функций в таком стиле обычно нагляднее и в программах на Haskell охраняющие условия используются очень широко (соответственно, условный оператор используется редко). Для иллюстрации наглядности приведем определение функции `signum` с использованием условных операторов:

```
signum x = if x < 0 then
            -1
          else
            if x == 0 then
              0
            else
              1
```

4 Полиморфные типы

В языке Haskell используется полиморфная система типов. В сущности, это означает, что в языке присутствуют *типовые переменные*. Рассмотрим уже известную нам функцию `tail`, которая возвращает первый элемент списка. Каков тип этой функции? Она одинаково применима и к списку целых чисел, и к списку символов, и к списку строк:

```
Prelude>tail [1,2,3]
[2,3]
Prelude>tail ['a','b','c']
['b','c']
Prelude>tail ["list","of","lists"]
["of", "lists"]
```

Функция `tail` имеет полиморфный тип: `[a] -> [a]`. Это означает, что она принимает в качестве аргумента любой список и возвращает список того же самого типа. Здесь `a` обозначает типовую переменную, т.е. подразумевается, что вместо нее можно подставить любой конкретный тип. Таким образом, запись `[a] -> [a]` задает целое семейство типов, представителями которого являются, например, `[Integer] -> [Integer]`, `[Char] -> [Char]`, `[[Char]] -> [[Char]]` и т.п.

Аналогично функция `tail`, возвращающая первый элемент списка, имеет тип `[a] -> a`. Представителями этого семейства являются типы `[Integer] -> Integer`, `[Char] -> Char` и т.п.

Многие функции, работающие со списками, парами и кортежами, имеют полиморфные типы. Так, функция `fst` имеет тип `(a,b) -> a` (заметьте, что в определении этого типа используются две типовые переменные).

5 Пользовательские типы

Помимо использования стандартных типов, программисту предоставлена возможность определять свои собственные, специфические типы данных. Для этого используется ключевое слово `data`.

5.1 Пары

Для примера рассмотрим определение пары, очень похожей на стандартную:

```
data Pair a b = Pair a b
```

Рассмотрим этот код подробно. Ключевое слово `data` показывает, что мы собираемся определять тип данных. Затем следует название этого типа, в данном случае `Pair` (напомним, что имена типов начинаются с заглавной буквы). Символы `a` и `b`, следующие за этим, являются типовыми переменными, обозначающими параметры типа. Таким образом, мы определяем структуру данных, параметризованную двумя типами `a` и `b`¹.

После знака равенства мы указываем *конструкторы данных* этого типа. В данном случае у нас есть единственный конструктор `Pair` (Имя конструктора данных не обязательно должно совпадать с именем типа, но в нашем примере это выглядит естественно). После имени конструктора данных мы

¹ Это напоминает шаблоны классов языка C++

снова пишем `a b`, что означает, что для того, чтобы сконструировать пару, нам нужно два значения: одно, принадлежащее типу `a` и другое, из типа `b`.

Это определение вводит функцию `Pair :: a -> b -> Pair a b`, которая используется для конструирования пар типа `Pair`. Загрузив этот код в интерпретатор, можно посмотреть, как конструируются пары:

```
Main> :t Pair
Pair :: a -> b -> Pair a b
Main> :t Pair 'a'
Pair 'a' :: a -> Pair Char a
Main> :t Pair 'a' "Hello"
Pair 'a' "Hello" :: Pair Char [Char]
```

Функции, соответствующие конструкторам данных, обладают тем свойством, что их можно использовать при сопоставлении с образцом. Так, функции для получения первого и второго элемента нашей пары можно определить следующим образом:

```
pairFst (Pair x y) = x
pairSnd (Pair x y) = y
```

Данный пример, возможно, вызывает вопрос: зачем определять свой тип `Pair`, если есть стандартная возможность определить пару? Во-первых, с использованием типа `Pair` можно определить набор функций, работающих только с этим типом и отделить их от функций, работающих с парами «вообще». Во-вторых, сделав один шаг вперед, можно определить некоторые ограничения на получаемые пары, которых невозможно добиться с помощью стандартного типа. Например, представьте, что нам нужен тип, хранящий пару элементов *одного и того же* типа. Его можно определить следующим образом:

```
Data SamePair a = SamePair a a
```

Здесь тип имеет один параметр, однако конструктор данных принимает два параметра одного и того же типа.

5.2 Множественные конструкторы

В предыдущем примере мы рассматривали тип данных с единственным конструктором. Также возможно (и зачастую очень полезно) определить тип с несколькими конструкторами. Конструкторы отделяются друг от друга символом `|`.

Рассмотрим тип `Color`, представляющий цвет, с возможными значениями `Red`, `Green` и `Blue`. Его можно определить так:

Data Color = Red | Green | Blue

Здесь Color – название типа, а Red, Green и Blue – конструкторы данных. Заметьте, что этот тип не принимает параметров. Такие типы называются *перечислимыми* и соответствуют конструкции `enum` в языке Си. Такие типы очень полезны. Например, стандартный тип `Bool` определен таким образом:

Data Bool = True | False

Однако множественные конструкторы могут также принимать параметры. Так, можно заметить, что наш тип `Color` позволяет определить только три фиксированных цвета. Расширим его так, чтобы он позволял определять произвольный цвет, задаваемый тремя целыми числами, соответствующим уровням красного, зеленого и синего (стандартное `rgb`-представление):

Data Color = Red | Green | Blue | RGB Int Int Int

Здесь тип `Color`, помимо стандартных цветов `Red`, `Green` и `Blue` (их список можно, конечно, расширить), позволяет определять произвольный цвет с помощью конструктора `RGB`, принимающего три целых числа, определяющих `rgb`-компоненты цвета. Тогда, например, функция для выделения `rgb`-компонента цвета запишется так:

```
redComponent :: Color -> Int
```

```
redComponent Red = 255
```

```
redComponent (RGB r _ _) = r
```

```
redComponent _ = 0
```

Типы с множественными конструкторами также могут быть полиморфными. Рассмотрим следующую проблему. Пусть функция должна вернуть некоторый результат, либо сообщить об ошибке. Например, функция для решения линейного уравнения возвращает найденный корень; функция для поиска первого неотрицательного числа в списке возвращает это число и т.п. Вместе с тем решение уравнения может не существовать, в списке может не оказаться неотрицательных чисел и т.д. Как сообщить об этом тому, кто вызвал функцию? Иногда (как в случае получения неотрицательного элемента) можно вывести договоренность, что возврат какого-либо специального значения (например, `-1`) означает «нет результата». Однако это не всегда возможно: в случае решения линейного уравнения такого выделенного значения не существует. Проблема изящно решается с помощью стандартного типа `Maybe`, определенного так:

Data Maybe a = Nothing | Just a

Тип `Maybe` (от английского `maybe` – возможно) параметризован типовой переменной `a` и предоставляет два конструктора: `Nothing` (англ. ничего) для представления отсутствия результата и `Just` (англ. просто, в точности) для осмысленного результата. Тогда наши функции можно записать так:

-- Функция возвращает корень уравнения $ax + b = 0$

```
solve :: Double -> Double -> Maybe Double
```

```
solve :: 0 b = Nothing
```

```
solve a b = Just (-b / a)
```

-- Функция возвращает первый неотрицательный элемент списка

```
findPositive :: [Integer] -> Maybe Integer
```

```
findPositive [] = Nothing
```

```
findPositive (x:xs) | x > 0 = Just x
```

```
                  | otherwise = findPositive xs
```

Использование типа `Maybe` обладает рядом преимуществ. Его использование явно показывает, что функция может вернуть «отсутствие результата». В случае выделенного значения для того, чтобы узнать, как функция сообщает об этом, необходимо изучить документацию к функции (которая может отсутствовать или быть неверной). С `Maybe` эта информация содержится в типе функции и ее может предоставить сам интерпретатор. Более того, при обработке возвращаемого значения функции необходимо явно сделать сопоставление с образцом, и если мы забудем обработать случай с `Nothing`, компилятор может выдать предупреждение.

5.3 Классы типов

Классы типов будут подробно изучаться позднее. Здесь мы дадим только базовое представление о них, поскольку они существенно облегчают работы с пользовательскими типами.

Класс типов представляет собой некоторое множество типов, обладающих рядом общих свойств. Например, в класс типов `Eq` входят те типы, для объектов которых определено отношение равенства, т.е., если переменные `x` и `y` принадлежат одному и тому же типу, входящему в класс `Eq`, мы можем вычислять выражения `x == y` и `x /= y`. Все простые типы, а также списки и кортежи входят в этот класс, однако, например, для функции отношение равенства не определено и типы функций не принадлежат классу `Eq`.

Другим важным для нас классом является класс Show. В него входят те типы, объекты которых могут быть преобразованы в строку для того, чтобы ее возможно было отобразить на экране. Простые типы, кортежи и списки входят в этот класс, поэтому интерпретатор может напечатать, например, строку. Функции не входят в этот класс.

По умолчанию пользовательские типы не входят ни в какой класс, поэтому значения этих типов нельзя сравнивать и интерпретатор не может напечатать их. Это, разумеется, неудобно. Поэтому можно при определении типа задать их принадлежность желаемым классам. Для этого после определения типа необходимо добавить ключевое слово `deriving` и в скобках перечислить классы, к которым должен принадлежать тип.

Пример:

-- Тип, представляющий время дня

```
Data DayTime = Morning
              | Aftrenoon
              | Evening
              | Night deriving (Eq, Show)
```

При определении типов в заданиях относите их к классам `Eq` и `Show`. Это существенно облегчит вашу работу.

6 Задания

Для выполнения лабораторной работы необходимо выполнить одно из представленных ниже заданий. Номер варианта определяет преподаватель. Правила оформления отчета – такие же, как в предыдущих работах.

1. В современных web-магазинах часто продают книги, видеокассеты и компакт-диски. База данных такого магазина для каждого типа товаров должна содержать следующие характеристики:

- Книги: название и автор
 - Видеокассеты: название
 - Компакт-диск: название, исполнитель и количество композиций
- 1) Разработать тип данных **Product**, который может представлять эти виды товаров.
 - 2) Определите функцию **getTitle**, возвращающую название товара.
 - 3) На ее основе определите функцию **getTitles**, которая по списку товаров возвращает список их названий.
 - 4) Определите функцию **bookAuthors**, которая по списку товаров возвращает список авторов книг.
 - 5) Определите функцию **lookupTitle :: String -> [Product] -> Maybe Product**, которая возвращает товар с заданным названием (обратите внимание на тип результата функции)
 - 6) Определите функцию **lookupTitles :: [String] -> [Product] -> [Product]**

Она принимает в качестве параметров список названий и список товаров и для каждого названия извлекает из второго списка соответствующие товары. Названия, которым не соответствует никакой товар, игнорируется. При определении функции *обязательно* используйте функцию **lookupTitle**.

2. Определите тип данных, представляющий информацию о карте в карточной игре. Каждая карта характеризуется одной из четырех мастей. Карта может быть либо младшей (от двойки до десятки), либо картинкой (валет, дама, король туз). Определите функции:

- 1) Функция **isMinor**, проверяющая, что ее аргумент является младшей картой.
- 2) Функция **sameSuit**, проверяющая, что переданные в нее карты – одной масти.
- 3) Функция **beats :: Card -> Card -> Bool**, проверяющая, что карта, переданная ей в качестве первого аргумента, бьет карту, являющуюся вторым аргументом.
- 4) Функция **beats2**, аналогичная **beats**, но принимающая в качестве дополнительного аргумента козырную масть.

- 5) Функция **beatsList**, принимающая в качестве аргументов список карт, карту и козырную масть и возвращающая список тех карт из первого аргумента, которые бьют указанную карту с учетом козырной масти.
 - 6) Функция, по заданному списку карт возвращающая список чисел, каждое из которых является возможной суммой очков указанных карт, рассчитанных по номиналу, валет, дама и король считаются за 10 очков, туз может рассматриваться как 1 и как 11 очков. Функция должна вернуть все возможные варианты.
3. Определите тип, представляющий геометрические фигуры на плоскости. Фигура может быть либо окружностью (характеризуется координатами центра и радиусом), прямоугольником (характеризуется координатами верхнего левого и нижнего правого углов), треугольником (координаты вершин) и текстовым полем (для него необходимо хранить положение левого нижнего угла, шрифт и строку, представляющую надпись). Шрифт задается из множества трех возможных шрифтов: Courier, Lucida и Fixedsys. Определите следующие функции.
- 1) Функция **area**, возвращающая площадь фигуры. Для текстового поля площадь зависит от высоты и ширины буквы в шрифте. Поскольку выбранные нами шрифты – моноширинные (т.е. ширина всех букв в них одинакова), вам необходимо также определить вспомогательную функцию, для каждого шрифта возвращающую его габариты.
 - 2) Функция **getRectangles**, из списка фигур выбирающая только прямоугольники.
 - 3) Функция **getBound**, по заданной фигуре возвращающая ограничивающий ее прямоугольник.
 - 4) Функция **getBounds**, по списку фигур возвращающая ограничивающий ее прямоугольник.
 - 5) Функция **getFigure**, по заданному списку фигур и координатам точки возвращающая первую фигуру, для которой точка попадает в ее ограничивающий прямоугольник. Используйте тип **Maybe** для возвращаемого значения.
 - 6) Функция **move**, по заданной фигуре и вектору сдвига возвращающая новую фигуру, сдвинутую относительно заданной на указанный вектор.
4. В агентстве недвижимости продают квартиры, комнаты и частные дома. Квартира характеризуется этажом, площадью и этажностью дома. Комната характеризуется, помимо этого, площадью комнаты (в дополнение к площади всей квартиры). Частный дом характеризуется только площадью. В базе данных хранятся пары значений, первое из которых представляет объект недвижимости, а второе – его цену.

Определите тип данных, представляющий информацию о таких объектах недвижимости. Определите следующие функции:

- 1) **getHouses**, выбирающая из базы данных только частные дома.
- 2) **getByPrice**, выбирающая из базы данных те объекты недвижимости, цена которых меньше указанной.
- 3) **getByLevel**, выбирающая из базы данных квартир, находящихся на указанном этаже.
- 4) **getExceptBounds**, выбирающая из базы данных квартиры, не находящиеся на крайних этажах (первых и последних).

Разработайте тип данных, представляющий различные требования к объектам недвижимости: желаемый тип объекта недвижимости, минимальная площадь, максимальная цена, ограничения на этаж. Разработайте функцию **query**, которая по списку требований выбирает из базы данных те объекты недвижимости, которые удовлетворяют всем требованиям.

5. В библиотеке хранятся книги, газеты и журналы. Книга характеризуется именем автора и названием; журнал – названием, месяцем и годом выпуска; газета – названием и датой выпуска. База данных представляет собой список этих объектов. Разработайте тип данных, представляющий объекты библиотечного хранения. Определите следующие функции:

- 1) **isPeriodic**, проверяющую, что ее аргумент является периодическим изданием.
- 2) **getByTitle**, выбирающая из списка объектов хранения (базы данных) объекты с указанным названием.
- 3) **getByMonth**, выбирающую из базы данных периодические издания, выпущенные в указанный месяц и указанный год (заметьте, что газеты выходят несколько раз в месяц).
- 4) **getByMonths**, действующая так же, как и предыдущая, но принимающая список месяцев.
- 5) **getAuthors**, возвращающая список авторов изданий, хранящихся в базе данных.

6. В некотором языке программирования существуют следующие типы данных:

- Простые типы: целые, вещественные и строки.
- Сложные типы: структуры. Структура имеет название и состоит из нескольких полей, каждое из которых, в свою очередь, имеет название и простой тип.

База данных идентификаторов программы представляет собой список пар, состоящих имени идентификатора и его типа. Разработайте тип данных, представляющий описанную информацию. Определите следующие функции:

- 1) **isStructure**, проверяющая, что ее аргумент является сложным типом.
- 2) **getType**, по заданному имени и списку идентификаторов (базе данных) возвращающая тип идентификатора с указанным именем (помните о том, что такого идентификатора в базе может и не оказаться).
- 3) **getFields**, по заданному имени возвращающая список полей идентификатора, если он имеет тип структуры.
- 4) **getByType**, возвращающая список имен идентификаторов указанного типа из базы данных.
- 5) **getByTypes**, аналогичная предыдущей, но принимающей вместо одного типа список типов (с помощью этой функции можно получить, например, список всех идентификаторов с числовым типом).

7. Определим следующий набор операций над строками:

- Очистка: удаление всех символов из строки
- Удаление: удаление всех вхождений указанного символа
- Замена: замена всех вхождений одного символа на другой
- Добавление: добавление в начало строки указанного символа

Разработайте тип данных, характеризующий операции над строками.

Определите следующие функции:

- 1) **process**, получающая в качестве аргумента действие и строку и возвращающая строку, модифицированную в соответствии с указанным действием.
- 2) **processAll**, аналогичная предыдущей, но получающая список действий и выполняющая их по порядку.
- 3) **deleteAll**, принимающая две строки и удаляющей из второй строки все символы первой. При реализации *обязательно* использовать функцию **processAll**,

8. В электронной записной книжке хранятся записи следующих видов: напоминания о днях рождения знакомых, телефоны знакомых и назначенные встречи. Напоминание состоит из имени знакомого и даты (день и месяц). Запись о телефоне должна содержать имя человека и его телефон. Информация о назначенной встрече содержит дату встречи (день, месяц, год) и краткое описание (можно представлять строкой). Разработайте тип данных, представляющий такую запись. Записная книжка является списком записей. Определите следующие функции:

- 1) **getName**, возвращающая информацию о человеке с указанным именем (его телефон и дату рождения).

- 2) **getByLetter**, возвращающая список людей, о которых есть информация в записной книжке и чье имя начинается на указанную букву.
- 3) **getAssignment**, возвращающая по указанной дате список дел (информация о назначенных встречах и телефоны друзей, которых нужно поздравить в этот день).

9. Клавиши на клавиатуре могут быть либо управляющими, либо алфавитно-цифровыми. Нажатие алфавитно-цифровой клавиши может сопровождаться нажатием клавиши Shift. Из управляющих клавиш нас интересует только клавиша CapsLock, остальные можно не различать. Каждое нажатие алфавитно-цифровой клавиши несет с собой информацию в виде символа. После нажатия CapsLock последующие символы переводятся в верхний регистр (если они не были нажаты вместе с Shift) до следующего нажатия CapsLock. Если режим CapsLock не активирован, символы, нажатые с Shift, переводятся в верхний регистр. Разработайте тип данных, представляющий указанную информацию в виде списка. Основная задача состоит в том, чтобы разработать функцию, переводящую эту последовательность в строку символов. Например, последовательность нажатий **Shift+’h’ ‘e’ CapsLock ‘l’ ‘l’ Shift+’o’ CapsLock** должна дать в результате строку **HeLLo**. Определите следующие функции:

- 1) **getAlNum**, возвращающая из списка нажатий только нажатия алфавитно-цифровых клавиш.
- 2) **getRaw**, возвращающая строку, составленную из нажатых символов без учета информации о **Shift** и **CapsLock**.
- 3) **isCapsLocked**, по последовательности нажатий определяющая, остался ли после нее режим **CapsLock** в активном состоянии.
- 4) **getString**, переводящая последовательность нажатий в строку.

При реализации функций можно воспользоваться стандартными функциями `toUpper` и `toLowerCase`, переводящими символ в верхний и нижний регистры соответственно. Они определены в модуле `Char`; чтобы их использовать, добавьте в начало программы строку: `import Char`

10. За время учебы в семестре студенты должны сдать определенное количество лабораторных работ, расчетно-графических заданий и рефератов. Лабораторная работа характеризуется названием предмета и номером, РГЗ – названием предмета, реферат – названием предмета и названием темы реферата. Разработайте тип данных, представляющий информацию по заданию. Учебный план студента представляет собой список, состоящий из пар, первый элемент которых является заданием, а второй – номером недели, в которую он был сдан. Если задание еще

не сдано, второй элемент пары должен быть пустым (используйте тип `Maybe`). Определите следующие функции:

- 1) **getByTitle** – возвращает задания, которые необходимо сдать по указанному предмету.
- 2) **getReferats** – возвращает список тем рефератов.
- 3) **getRest** – возвращает список заданий, оставшихся несданными на указанный неделе.
- 4) **getrestForWeek** – возвращает список заданий, оставшихся несданными на указанной неделе.
- 5) **getPlot** – создает список, состоящий из пар, первый элемент которых равен номеру недели, а второй – количеству сданных на эту неделю заданий.

7 Контрольные вопросы

1. Определение локальных переменных
2. Охраняющие условия
3. Полиморфизм
4. Определение пользовательских типов данных