Лабораторные работы по курсу "Функциональное программирование"

Федеральное государственное бюджетное образовательное учреждение высшего образования

«РОССИЙСКАЯ АКАДЕМИЯ

НАРОДНОГО ХОЗЯЙСТВА И ГОСУДАРСТВЕННОЙ СЛУЖБЫ

ПРИ ПРЕЗИДЕНТЕ РОССИЙСКОЙ ФЕДЕРАЦИИ»

Нижегородский институт управления – филиал РАНХиГС

(наименование института (факультета))

Кафедра информатики и информационных технологий

(наименование кафедры)

*Дисциплина: «***Б1. В.05 ОСНОВЫ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ***»*

Лабораторные и практические задания

Нижний Новгород, 2025 г

Тема 1 История и проблематика функционального программирования
Теоретическая часть

Haskell — функциональный язык программирования, характеризующийся полной, строгой и статической типизацией с автоматическим выведением типов, поддержкой ленивых вычислений. Был разработан в 1990 году и первоначально использовался как инструмент для математических и научных исследований. Сегодня является одним из самых востребованных функциональных языков и достаточно широко применяется в промышленном программировании.

История

Предпосылкой к появлению компьютерного языка «Хаскелл» стала разработка в 1985 году другого функционального ЯП — «Миранда», в котором впервые была реализована концепция «ленивых» вычислений. Несмотря на то что «Миранда» получила хорошую коммерческую поддержку, этот язык распространялся по проприетарной лицензии и был закрыт для свободного исследования и развития.

Чтобы обойти это ограничение, различные разработчики начали создавать во второй половине 80-х годов разные свободные функциональные языки программирования, одним из которых и стал Haskell. Свое название он получил в честь американского математика Хаскелла Карри, который изобрел комбинаторную логику и развивал теорию типов.

Первая версия Haskell появилась в 1990 году, а через 8 лет комитет разработчиков представил The Haskell 98 Report. Именно этот стандарт по сей день остается основным, хотя совершенствование «Хаскелла» не останавливается и в него постоянно вносятся новые улучшения. Во многом это стало возможным благодаря открытому характеру языка: комитет, который занимается разработкой, бесплатно принимает предложения от всех желающих.

Общая характеристика языка Haskell

Язык программирования «Хаскелл» относится к категории функциональных. Написанная на нем программа представляет собой набор функций в математическом смысле, то есть как зависимостей (соответствий) между элементами (аргументом и значением) двух множеств. При этом для вычисления значения функции имеют только исходные данные независимо от того, в каком порядке оно происходит.

Функция в Haskell — это основная структурная единица кода. Задача программиста заключается в описании ее таким образом, чтобы компилятору (программе, переводящей текст в машинный код) было понятно:

- какие параметры могут перейти в функцию;
- какие действия с ними нужно выполнить;
- в каком виде представить полученный результат.

Важной характеристикой языка «Хаскелл» является «ленивость». То есть функции вычисляются по мере их необходимости. Если для исполнения программы в каком-то случае не нужно знать значение конкретной функции — она откладывается «на потом». Этот принцип получил название «ленивых» (или отложенных) вычислений. Например, в программе «Калькулятор» заложено много функций, в том числе сложение, вычитание, умножение, деление и т.д. Но если пользователю в данный момент требуется именно сложение, будет выполнено только оно.

К другим важным характеристикам Haskell относятся:

Строгая статическая типизация. Тип данных — это категория элементов некоторого множества, к которым могут быть применены какие-либо определенные операции. Иными словами, он определяет, что это за данные, какие у них могут быть

значения и что с ними можно сделать. У «Хаскелла» строгая и статичная система типов — это значит, что они четко различаются между собой, и это разделение задается еще на этапе компиляции, а не в процессе выполнения программы. Однако в данном языке есть особый тип данных, который называется монадой. Он представляет собой контейнер, содержащий значение произвольного типа. Наличие монад помогает выполнять на функциональном Haskell некоторые действия, свойственные императивному программированию: например, задавать последовательность операций, выполнять функции с побочными эффектами и т.д.

Модульность. Программы, написанные на «Хаскелле», представляют собой совокупность независимых друг от друга блоков, каждый из которых выполняет определенную функцию. При этом программный код разбивается на отдельные файлы, содержащие модули. Такая структура упрощает разработку и тестирование ПО — при необходимости внести изменения или исправить ошибку можно только в конкретном модуле или элементе, не переделывая всю программу. Кроме того, модульность помогает ускорить разработку программного продукта, разделив модули между отдельными командами разработчиков.

Чистота. «Хаскелл» работает только с чистыми функциями, для которых характерны:

- **строгая детерминированность —** то есть каждому значению аргумента соответствует только одно значение функции;
- **отсутствие побочных эффектов —** воздействий функции на среду ее выполнения, таких как вызов обработчиков исключительных ситуаций (проще говоря, ошибок), чтение и изменение значений глобальных переменных и т.д.

Однако, так как для некоторых задач программирования требуются и недетерминированность, и присутствие побочных эффектов, в «Хаскелле» эти возможности предусмотрены с помощью монад.

Параметрический полиморфизм. Это способность оперировать значениями различных типов одним и тем же образом. Она повышает выразительность языка программирования и позволяет многократно использовать один фрагмент кода для работы с разными данными, что существенно упрощает и ускоряет разработку программного обеспечения. Проще говоря, программисту не нужно писать для обработки каждого типа данных свой код — достаточно использовать уже имеющийся, если он подходит.

Частичное применение. В «Хаскелле» по умолчанию функции с несколькими аргументами (многоместные) представляются как функции высшего порядка, в которых аргументами являются другие функции. Благодаря этому к функции можно «привязать» не особо важные или редко меняющиеся аргументы, а принимать (то есть непосредственно работать) она будет с важными и часто меняющимися.

Преимущества языка Haskell

Из особенностей «Хаскелла» вытекают и основные его преимущества перед другими языками программирования:

- «Чистые» и «нечистые» слои языка четко разделены, благодаря чему программисту легче работать с обоими по отдельности, что способствует пониманию кода не только тем, кто его написал, но и сторонним разработчиком.
- За счет разделения кода на отдельные блоки существенно упрощается тестирование программного обеспечения. Найти и исправить ошибку в отдельном модуле гораздо легче, к тому же не нужно переделывать весь программный продукт. Также это повышает устойчивость кода к ошибкам: если они происходят в одном модуле, то не обязательно отразятся на всей системе.
- Благодаря открытому характеру языка существует множество инструкций, библиотек готовых решений, справочных рекомендаций, книг по Haskell и других полезных материалов. Сформировалось довольно мощное комьюнити профессионалов и

любителей: если разработчик столкнулся с проблемой во время работы над продуктом, у него есть где бесплатно попросить помощи или самостоятельно найти решение.

- Строгая типизация не позволяет смешивать значения различных типов и проводить неявные вычисления. Это существенно снижает вероятность ошибки или побочных эффектов при программировании. Если же разработчик все же использует неявные преобразования (Haskell это допускает), то сразу будет видеть, где код может сработать не так, как надо.
- За счет того, что типы присваиваются и проверяются еще на этапе компиляции, этот процесс не нужно проводить во время работы программы. Также за счет «ленивости» вычислений программа не тратит ресурсы на выполнение ненужных в данный момент функций, задействуя только необходимые. Соответственно, повышается скорость исполнения программного кода.
- Благодаря тому что функции в Haskell исполняются по мере востребованности, а не в порядке, прописанном программистом, с помощью этого языка достаточно просто добиться многозадачности и параллельного вычисления. Это тоже положительно влияет на быстродействие программы.
- Для «Хаскелла» разработано множество инструментов тестирования и отладки, интеграции с кодом на других языках программирования и т.д. Все это способствует тому, что он может успешно применяться не только для обучения или в академических интересах, но и для решения практических задач — от математических и научных расчетов до разработки приложений.

Недостатки языка Haskell

- Специфичный синтаксис, который может поначалу восприниматься как очень сложный и нерациональный. Особенно часто с этой проблемой сталкиваются опытные разработчики, привыкшие работать с языками в других, более мейнстримных парадигмах программирования. Напротив, Haskell для начинающих свой путь в программировании довольно прост, так как их сознание еще не настроено на конкретную парадигму.
- «Хаскелл» развивается неравномерно на различных платформах — его основные разработчики делают упор на Linux и MacOS и недостаточно уделяют внимание Windows. Из-за этого программисты, работающие на последней, сталкиваются с нехваткой инструментов, запозданием обновлений и другими проблемами.
- Хотя для Haskell существует множество репозиториев и библиотек, зачастую в них имеются только решения для типовых проблем. Если программист сталкивается с нестандартной задачей, то придется либо решать ее самостоятельно, либо обращаться за помощью к представителям сообщества.
- Активно развивается только основной компилятор Haskell GHC, причем это развитие происходит неравномерно. Из-за этого часто возникают проблемы с обратной совместимостью.
- Для Haskell на данный момент не существует хороших интегрированных сред разработки. Те, которые имеются сейчас, предлагают очень ограниченный набор возможностей вроде подсветки синтаксиса или простого автозаполнения по словарю. Другие функции, например навигация или рефакторинг (изменение внутренней структуры программы, не затрагивающее ее поведение), либо отсутствуют, либо очень неоптимизированные.

Применение Haskell

Язык программирования Haskell появился и первое время развивался в академической среде. Поэтому первой сферой его применения стало решение математических задач, проведение научных исследований, а также теоретическое изучение концепции функционального программирования как такового. Часто программисты, привыкшие работать с другими языками программирования, говорят, что изучать Haskell нужно только ради понимания его самого, а значимой практической

ценности он не имеет. Это утверждение далеко от действительности — «Хаскелл» помогает решать те же самые практические задачи, что и большинство распространенных языков программирования. Вот лишь часть из них:

- **Создание ПО для разработчиков.** На языке «Хаскелл» написаны многие компиляторы для других языков, среды разработки, инструменты тестирования и т.д. За счет его функционального характера разработчикам проще оптимизировать преобразование программного кода в машинный, найти недочеты и уязвимости в ПО.
- **Разработка других языков программирования.** Созданный в академической среде, Haskell представляет собой как бы фундаментальный язык, который может использоваться для создания более прикладных, предметно ориентированных ЯП. Например, на его основе разработан Cryptol, на котором создаются криптографические алгоритмы, Bluespec SystemVerilog, с помощью которого проектируются и верифицируются полупроводниковые схемы, и т.д.
- **Обработка и синтаксический анализ текста.** С помощью Haskell можно создать простые и эффективные инструменты для анализа любого естественного или искусственного языка (включая компьютерные). Это достигается за счет того, что на «Хаскелле» проще задавать правила, которым подчиняется знаковая система. С его помощью можно разделять фразы и предложения на отдельные словосочетания, слова и предлоги, выявлять между ними связи, находить ошибки и неправильные конструкции. Все это используется в системах компьютерного анализа текста, используемых лингвистами, редакторами, копирайтерами, SEO-специалистами и т.д.
- **Финансовые инструменты.** Одно из требований, предъявляемых к ПО в финансовой сфере, — точность вычислений и отсутствие ошибок, цена которых может достигать миллиардов долларов. Поэтому многие системы для торговли на биржах, анализа рисков, учета банковских транзакций, бухучета и т.д. проектируются именно на Haskell.
- **Обработка данных.** Так как в «Хаскелле» данные четко разделены по типам, анализировать их и работать с ними становится значительно проще. Эту особенность языка эффективно используют и для фундаментальных исследований в Data Science, и для решения прикладных задач, связанных с обработкой больших массивов различной информации. Смежной сферой применения Haskell является разработка систем принятия решений, которые неизменно связаны с анализом данных.
- **Прикладное программирование.** «Хаскелл» активно используется для разработки сугубо прикладных приложений — десктопных, мобильных, серверных и т.д. Помимо компиляторов и других программ для разработчиков, на этом языке написаны некоторые операционные системы (например, экспериментальная House), текстовый редактор Yi, фреймовый оконный менеджер Xmonad и другие программные продукты. Также Haskell применяется в игровой сфере — например, для моделирования городских ландшафтов.
- **Обучение.** «Хаскелл» хорошо подходит для изучения основ функционального программирования за счет относительно простого синтаксиса и семантики, наличия большого количества справочных материалов, инструкций и поддержки комьюнити. Многие программисты считают, что этот язык является подходящим для понимания принципов программирования вообще.

Перспективы и проблемы Haskell

Развитие и применение языка программирования «Хаскелл» сталкивается с определенными проблемами:

- **Низкой коммерческой поддержкой.** Хотя Haskell может применяться для решения прикладных задач, сам язык не так активно поддерживается крупными игроками рынка, как его более распространенные собратья вроде JavaScript, C++, Python и т.д. Во многом это объясняется как специфичностью самого функционального программирования, так и закрепившейся за языком репутацией «академического».

- **Непониманием со стороны сообщества.** Многие начинающие и опытные программисты, изучающие другие парадигмы программирования, с трудом переходят на «Хаскелл» из-за его непривычности. Это создает ему репутацию «трудного языка», который к тому же мало востребован на рынке, чтобы тратить время и силы на его изучение.
- **Появление других функциональных языков.** «Хаскелл» во многом был и остается экспериментальной платформой, на которой исследуются различные возможности функционального программирования. Из-за этого, с одной стороны, в языке встречаются недочеты, с другой — его удачные решения «перетекают» в более новые языки, такие как Rust, PureScript. Они лишены его недостатков и «детских болезней», из-за чего выглядят более перспективными.

Тем не менее постепенно востребованность Haskell в прикладном программировании растет, хотя на сегодняшний день он по-прежнему сильно уступает по популярности другим распространенным языкам программирования. Специалисты неоднозначно оценивают его будущее, хотя в общем сходятся на том, что у этого ЯП большой потенциал, в том числе и в сфере прикладного ПО.

Основы языка Haskell»

Структуры данных и их типы

Одна из базовых единиц любого языка программирования — символ. Символом традиционно называется последовательность букв, цифр и специальных знаков ограниченной или неограниченной длины. В некоторых языках строчные и прописные буквы различаются, в некоторых нет. Так в Lisp'е различия между строчными и заглавными буквами нет, а в Haskell'е есть.

Символы чаще всего выступают в качестве идентификаторов — имен констант, переменных, функций. Значениями же констант, переменных и функций являются типизированные последовательности знаков. Так значением числовой константы не может быть строка из букв и т.п. В функциональных языках существует базовое понятие — атом. В реализациях атомами называются символы и числа, причем числа могут быть трех видов: целые, с фиксированной и с плавающей точкой.

Следующим понятием функционального программирования является список. В абстрактной математической нотации использовались символы [], которые также используются в Haskell'е. Но в Lisp'е используются обычные «круглые» скобки — (). Элементы списка в Lisp'е разделяются пробелами, что не очень наглядно, поэтому в Haskell'е было решено ввести запятую для разделения. Таким образом, список [a, b, c] будет правильно записан в синтаксисе Haskell'а, а в нотацию Lisp'а его необходимо перевести как (a b c). Однако создатели Lisp'а пошли еще дальше в своей изощрённости. Допускается использовать точечную запись для организации пары, поэтому приведенный выше список можно записать как (a.(b.(c.NIL))).

Списочные структуры в Lisp'е и Haskell'е описываются в соответствии с нотацией — заключение одного списка в другой. При этом в нотации Lisp'а сделано послабление, т.к. перед скобкой внутреннего списка можно не ставить пробел.

Типы данных в функциональных языках определяются автоматически. Механизм автоматического определения типа встроен и в Haskell. Однако в некоторых случаях необходимо явно указывать тип, иначе интерпретатор может запутаться в неоднозначности (в большинстве случаев будет выведено сообщение об ошибке или предупреждение). В Haskell'е используется специальный символ — :: (два двоеточия), котрый читается как «имеет тип». Т.е. если написать:

```
5 :: Integer
```

Это будет читаться как «Числовая константа 5 имеет тип Integer (Целое число)».

Однако Haskell поддерживает такую незаурядную вещь, как полиморфные типы, или шаблоны типов. Если, например, записать [a], то это будет обозначать тип «список из атомов любого типа», причем тип атомов должен быть одинаковым на протяжении всего списка. Т.е. списки [1, 2, 3] и ['a', 'b', 'c'] будут иметь тип [a], а список [1, 'a'] будет другого типа. В этом случае в записи [a] символ a имеет значение типовой переменной.

Соглашения по именованию

В Haskell'е очень важны соглашения по именованию, ибо они явно входят в синтаксис языка (чего обычно нет в императивных языках). Самое важное соглашение — использование заглавной буквы в начале идентификатора. Имена типов, в том числе и определяемых разработчиком, должны начинаться с заглавной буквы. Имена функций, переменных и констант должны начинаться со строчной буквы. В качестве первого символа идентификатора также возможно использование некоторых специальных знаков, некоторые из которых также влияют на семантику идентификатора.

Определители списков и математические последовательности

Пожалуй, Haskell — это единственный язык программирования, который позволяет просто и быстро конструировать списки, основанные на какой-нибудь простой математической формуле. Этот подход уже был использован при построении функции быстрой сортировки списка методом Хоара (см. пример 3 в лекции 1). Наиболее общий вид определителей списков выглядит так:

```
[ x | x <- xs ]
```

Эта запись может быть прочитана как «Список из всех таких x, взятых из xs». Структура «x ← xs» называется генератором. После такого генератора (он должен быть один и стоять первым в записи определителя списка) может стоять некоторое число выражений охраны, разделённых запятыми. В этом случае выбираются все такие x, значения всех выражений охраны на которых истинно. Т.е. запись:

```
[ x | x <- xs, x > m, x < n ]
```

Можно прочитать как «Список из всех таких x, взятых из xs, что (x больше m) И (x меньше n)».

Другой важной особенностью Haskell'а является простая возможность формирования бесконечных списков и структур данных. Бесконечные списки можно формировать как на основе определителей списков, так и с помощью специальной нотации. Например, ниже показан бесконечный список, состоящий из последовательности натуральных чисел. Второй список представляет бесконечную последовательность нечётных натуральных чисел:

```
[1, 2 ..]
[1, 3 ..]
```

При помощи двух точек можно также определять любую арифметическую прогрессию, как конечную, так и бесконечную. Если последовательность конечна, то в ней задаются первый и последний элементы. Разность арифметической прогрессии вычисляется на основе первого и второго заданного элементов — в приведенных выше примерах разность в первой прогресси равна 1, а во второй — 2. Т.е. чтобы определить список всех нечётных натуральных чисел вплоть до 10, необходимо записать: [1, 3 .. 10]. Результатом будет список [1, 3, 5, 7, 9].

Бесконечные структуры данных можно определять на основе бесконечных списков, а можно использовать механизм рекурсии. Рекурсия в данном случае используется как обращение к рекурсивным функциям. Третий способ создания бесконечных структур данных состоит в использовании бесконечных типов.

Пример 1. Определение типа для представления двоичных деревьев.

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

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

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

В этом примере показан способ определения бесконечного типа. Видно, что без рекурсии тут не обошлось. Однако если нет необходимости создавать новый тип данных, бесконечную структуру можно получить при помощи функций:

```
ones            = 1 : ones
numbersFrom n       = n : numberFrom (n + 1)
squares             = map (^2) (numbersFrom 0)
```

Первая функция определяет бесконечную последовательность, полностью состоящую из единиц. Вторая функция возвращает последовательность целых чисел, начиная с заданного. Третья возвращает бесконечную последовательность квадратов натуральных чисел вместе с нулем.

Вызовы функций

Математическая нотация вызова функции традиционно полагала заключение параметров вызова в скобки. Эту традицию впоследствии переняли практически все императивные языки. Однако в функциональных языках принята иная нотация — имя функции отделяется от её параметров просто пробелом. В Lisp'е вызов функции length с неким параметром L записывается в виде списка: (length L). Такая нотация объясняется тем, что большинство функций в функциональных языках каррированны.

В Haskell'е нет нужды обрамлять вызов функции в виде списка. Например, если определена функция, складывающая два числа:

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

То ее вызов с конкретными параметрами (например, 5 и 7) будет выглядеть как:

```
add 5 7
```

Здесь видно, что нотация Haskell'а наиболее сильно приближена к нотации абстрактного математического языка. Однако Haskell пошел еще дальше Lisp'а в этом вопросе, и в нем есть нотация для описания некаррированных функций, т.е. тип которых нельзя представить в виде $A_1 \to (A_2 \to ... (A_n \to B) ...)$. И эта нотация, как и в императивных языках программирования, использует круглые скобки:

```
add (x, y)      = x + y
```

Можно видеть, что последняя запись — это функция с одним аргументом в строгой нотации Haskell'а. С другой стороны для каррированных функций вполне возможно делать частичное применение. Т.е. при вызове функции двух аргументов передать ей только один. Как показано в предыдущей лекции результатом такого вызова будет также функция. Более чётко этот процесс можно поиллюстрировать на примере функции inc, которая прибавляет единицу к заданному аргументу:

```
inc     :: Integer -> Integer
inc     = add 1
```

Т.е. в этом случае вызов функции inc с одним параметром просто приведет к вызову функции add с двумя, первый из которых — 1. Это интуитивное понимание понятия частичного применения. Для закрепления понимания можно рассмотреть классический пример — функция map (её определение на абстрактном функциональном языке приведено во второй лекции). Вот определение функции map на Haskell'е:

```
map                 :: (a -> b) -> [a] -> [b]
map f []            = []
map f (x:xs)    = (f x) : (map f xs)
```

Как видно, здесь использована инфиксная запись операции **prefix** — двоеточие, только такая запись используется в нотации Haskell'а для обозначения или конструирования пары. После приведенного выше определения можно произвести следующий вызов:

```
map (add 1) [1, 2, 3, 4]
```

Результатом которого будет список [2, 3, 4, 5].

Использование λ-исчисления

Т.к. функциональная парадигма программирования основана на λ-исчислении, то вполне закономерно, что все функциональные языки поддерживают нотацию для описания λ-абстракций. Haskell не обошел стороной и этот аспект, если есть необходимость в определении какой-либо функции через λ-абстракцию. Кроме того, через λ-абстракции можно определять анонимные функции (например, для единичного вызова). Ниже показан пример, где определены функции add и inc именно при помощи λ-исчисления.

Пример 2. Функции add и inc, определённые через λ-абстракции.

```
add     = \x y -> x + y
inc     = \x -> x + 1
```

Пример 3. Вызов аномимной функции.

```
cubes   = map (\x -> x * x * x) [0 ..]
```

Пример 3 показывает вызов анонимной функции, возводящей в куб переданный параметр. Результатом выполнения этой инструкции будет бесконечный список кубов целых чисел, начиная с нуля. Необходимо отметить, что в Haskell'е используется упрощенный способ записи λ-выражений, т.к. в точной нотации функцию add правильней было бы написать как:

```
add     = \x -> \y -> x + y
```

Остаётся отметить, что тип λ-абстракции определяется абсолютно так же, как и тип функций. Тип λ-выражения вида λx.expr будет выглядеть как $T_1 \to T_2$, где T_1 — это тип переменной x, а T_2 — тип выражения expr.

Инфиксный способ записи функций

Для некоторых функций возможен инфиксный способ записи, такие функции обычно представляют собой простые бинарные операции. Вот как, например, определены операции конкатенации списков и композиции функций:

Пример 4. Инфиксная операция конкатенации списков.

```
(++)                :: [a] -> [a] -> [a]
[] ++ ys            = ys
(x:xs) ++ ys    = x : (xs ++ ys)
```

Пример 5. Инфиксная операция композиции функций.

```
(.)             :: (b -> c) -> (a -> b) -> (a -> c)
f . g           = \x -> f (g x)
```

Т.к. инфиксные операции всё-таки являются функциями в смысле Haskell'а, т.е. они каррированы, то имеет смысл обеспечить возможность частичного применения таких функций. Для этих целей имеется специальная запись, которая в Haskell'е носит название «секция»:

```
(x ++)  = \x -> (x ++ y)
(++ y)  = \y -> (x ++ y)
(++)    = \x y -> (x ++ y)
```

Выше показаны три секции, каждая из которых определяет инфиксную операцию конкатенации списков в соответствии с количеством переданных ей аргументов. Использование круглых скобок в записи секций является обязательным.

Если какая-либо функция принимает два параметра, то её также можно записывать в инфиксной форме. Однако если просто записать между параметрами имя функции — это будет ошибкой, т.к. в строгой нотации Haskell'а, это будет просто двойным применением, причем в одном применении не будет хватать одного операнда. Для того чтобы записать функцию в инфиксной форме, её имя необходимо заключить в символы обратного апострофа — `.

Для вновь определённых инфиксных операций возможно определение порядка вычисления. Для этого в Haskell'е есть зарезервированное слово infixr, которое назначает

заданной операции степень её значимости (порядок выполнения) в интервале от 0 до 9, при этом 9 объявляется самой сильной степенью значимости (число 10 также входит в этот интервал, именно эту степень имеет операция применения). Вот так определяются степени для определенных в примерах 4 и 5 операций:

```
infixr 5 ++
infixr 9 .
```

Остается отметить, что в Haskell'е все функции являются нестрогими, т.е. все они поддерживают отложенные вычисления. Например, если какая-то функция определена как:

```
bot      = bot
```

При вызове такой функции произойдет ошибка, и обычно такие ошибки сложно отслеживать. Но если есть некая константная функция, которая определена как:

```
constant_1 x   = 1
```

То при вызове конструкции (constant_1 bot) никакой ошибки не произойдёт, т.к. значение функции bot в этом случае не вычислялось бы (вычисления отложенные, значение вычисляется только тогда, когда оно действительно требуется). Результатом вычисления естественно будет число 1.

Тема 2 Основы лямбда-исчисления

Теоретическая часть

Возможно, у этой системы найдутся приложения не только в роли логического исчисления. (Алонзо Чёрч, 1932)

Вообще говоря, лямбда-исчисление не относится к предметам, которые «должен знать каждый уважающий себя программист». Это такая теоретическая штука, изучение которой необходимо, когда вы собираетесь заняться исследованием систем типов или хотите создать свой функциональный язык программирования. Тем не менее, если у вас есть желание разобраться в том, что лежит в основе Haskell, ML и им подобных, «сдвинуть точку сборки» на написание кода или просто расширить свой кругозор, то прошу под кат. Начнём мы с традиционного (но краткого) экскурса в историю. В 30-х годах прошлого века перед математиками встала так называемая проблема разрешения (*Entscheidungsproblem*), сформулированная Давидом Гильбертом. Суть её в том, что вот есть у нас некий формальный язык, на котором можно написать какое-либо утверждение. Существует ли алгоритм, за конечное число шагов определяющий его истинность или ложность? Ответ был найден двумя великими учёными того времени Алонзо Чёрчем и Аланом Тьюрингом. Они показали (первый — с помощью изобретённого им λ-исчисления, а второй — теории машины Тьюринга), что для арифметики такого алгоритма не существует в принципе, т.е. Entscheidungsproblem в общем случае неразрешима. Так лямбда-исчисление впервые громко заявило о себе, но ещё пару десятков лет продолжало быть достоянием математической логики. Пока в середине 60-х Питер Ландин не отметил, что сложный язык программирования проще изучать, сформулировав его ядро в виде небольшого базового исчисления, выражающего самые существенные механизмы языка и дополненного набором удобных производных форм, поведение которых можно выразить путем перевода на язык базового исчисления. В качестве такой основы Ландин использовал лямбда-исчисление Чёрча.

λ-исчисление: основные понятия

Синтаксис

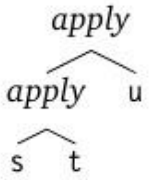
В основе лямбда-исчисления лежит понятие, известное ныне каждому программисту, — анонимная функция. В нём нет встроенных констант, элементарных операторов, чисел, арифметических операций, условных выражений, циклов и т. п. — только функции, ~~только хардкор~~. Потому что лямбда-исчисление — это **не** язык программирования, а формальный аппарат, способный определить в своих терминах любую языковую конструкцию или алгоритм. В этом смысле оно созвучно машине Тьюринга, только соответствует функциональной парадигме, а не императивной. Мы с вами рассмотрим его наиболее простую форму: чистое нетипизированное лямбда-исчисление, и вот что конкретно будет в нашем распоряжении.

Термы:

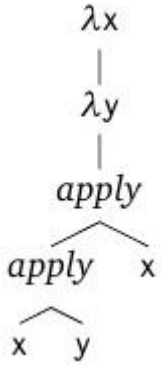
переменная:	x
лямбда-абстракция (анонимная функция):	λx.t, где x — аргумент функции, t — её тело.
применение функции (аппликация):	f x, где f — функция, x — подставляемое в неё значение аргумента

Соглашения о приоритете операций:

- Применение функции левоассоциативно. Т.е. s t u — это тоже самое, что (s t) u



- Аппликация (применение или вызов функции по отношению к заданному значению) забирает себе всё, до чего дотянется. Т.е. λx. λy. x y x означает то же самое, что λx. (λy. ((x y) x))



- Скобки явно указывают группировку действий.

Может показаться, будто нам нужны какие-то специальные механизмы для функций с несколькими аргументами, но на самом деле это не так. Действительно, в мире чистого лямбда-исчисления возвращаемое функцией значение тоже может быть функцией. Следовательно, мы можем применить первоначальную функцию только к одному её аргументу, «заморозив» прочие. В результате получим новую функцию от «хвоста» аргументов, к которой применим предыдущее рассуждение. Такая операция называется *каррированием* (в честь того самого Хаскелла Карри). Выглядеть это будет примерно так:

f	=	Функция с двумя аргументами x и y и телом t
λx.λy.t		

f v w	Подставляем в f значения v и w
(f v) w	Эта запись аналогична предыдущей, но скобки явно указывают на последовательность подстановки
((λy.[x → v]t) w)	Подставили v вместо x. [x → v]t означает «тело t, в котором все вхождения x заменены на v»
[y → w][x → v]t	Подставили w вместо y. Преобразование закончено.

И напоследок несколько слов об *области видимости*. Переменная x называется *связанной*, если она находится в теле t λ-абстракции λx.t. Если же x не связана какой-либо вышележащей абстракцией, то её называют *свободной*. Например, вхождения x в x y и λy.x y свободны, а вхождения x в λx.x и λz.λx.λy.x(y z) связаны. В (λx.x)x первое вхождение x связано, а второе свободно. Если все переменные в терме связаны, то его называют *замкнутым*, или *комбинатором*. Мы с вами будем использовать следующий простейший комбинатор (*функцию тождества*): id = λx.x. Она не выполняет никаких действий, а просто возвращает без изменений свой аргумент.

Процесс вычисления

Рассмотрим следующий терм-применение:

(λx.t) y

Его левая часть — (λx.t) — это функция с одним аргументом x и телом t. Каждый шаг вычисления будет заключаться в замене всех вхождений переменной x внутри t на y. Терм-применение такого вида носит имя *редекса* (от *reducible expression, redex* — «сокращаемое выражение»), а операция переписывания редекса в соответствии с указанным правилом называется *бета-редукцией*.

Существует несколько стратегий выбора редекса для очередного шага вычисления. Рассматривать их мы будем на примере следующего терма:

(λx.x) ((λx.x) (λz. (λx.x) z)), который для простоты можно переписать как id (id (λz. id z)) (напомним, что id — это функция тождества вида λx.x) В этом терме содержится три редекса:

```
id (id (λz. id z))
id (id (λz. id z))
id (id (λz. id z))
```

1. **Полная β-редукция**. В этом случае каждый раз редекс внутри вычисляемого терма выбирается произвольным образом. Т.е. наш пример может быть вычислен от внутреннего редекса к внешнему:

```
   id (id (λz. id z))
→  id (id (λz.z))
→  id (λz.z)
→  λz.z
↛
```

2. **Нормальный порядок вычислений**. Первым всегда сокращается самый левый, самый внешний редекс.

```
  id (id (λz. id z))
→ id (λz. id z)
→ λz. id z
→ λz.z
↛
```

3. **Вызов по имени**. Порядок вычислений в этой стратегии аналогичен предыдущей, но к нему добавляется запрет на проведение сокращений внутри абстракции. Т.е. в нашем примере мы останавливаемся на предпоследнем шаге:

```
  id (id (λz. id z))
→ id (λz. id z)
→ λz. id z
↛
```

Оптимизированная версия такой стратегии (*вызов по необходимости*) используется Haskell. Это так называемые «ленивые» вычисления.

4. **Вызов по значению**. Здесь сокращение начинается с самого левого (внешнего) редекса, у которого в правой части стоит *значение* — замкнутый терм, который нельзя вычислить далее.

```
  id (id (λz. id z))
→ id (λz. id z)
→ λz. id z
↛
```

Для чистого лямбда-исчисления таким термом будет λ-абстракция (функция), а в более богатых исчислениях это могут быть константы, строки, списки и т.п. Данная стратегия используется в большинстве языков программирования, когда сначала вычисляются все аргументы, а затем все вместе подставляются в функцию.

Если в терме больше нет редексов, то говорят, что он *вычислен*, или находится в *нормальной форме*. Не каждый терм имеет нормальную форму, например (λx.xx)(λx.xx) на каждом шаге вычисления будет порождать самоё себя (здесь первая скобка — анонимная функция, вторая — подставляемое в неё на место x значение). Недостатком стратегии вызова по значению является то, что она может зациклиться и не найти существующее нормальное значение терма. Рассмотрим для примера выражение (λx.λy. x) z ((λx.x x)(λx.x x)) Этот терм имеет нормальную форму z несмотря на то, что его второй аргумент такой формой не обладает. На её-то вычислении и зависнет стратегия вызова по значению, в то время как стратегия вызова по имени начнёт с самого внешнего терма и там определит, что второй аргумент не нужен в принципе. Вывод: если у редекса есть нормальная форма, то «ленивая» стратегия её обязательно найдёт. Ещё одна тонкость связана с именованием переменных. Например, терм (λx.λy.x)y после подстановки вычислится в λy.y. Т.е. из-за совпадения имён переменных мы получим функцию тождества там, где её изначально не предполагалось. Действительно, назови мы локальную переменную не y, а z — первоначальный терм имел бы вид(λx.λz.x)y и после редукции выглядел бы как λz.y. Для исключения неоднозначностей такого рода надо чётко отслеживать, чтобы все свободные переменные из начального терма после подстановки оставались свободными. С этой целью используют *α-конверсию* — переименование переменной в абстракции с целью исключения конфликтов имён. Так же бывает, что у нас есть абстракция λx.t x, причём x свободных вхождений в тело t не имеет. В этом случае данное выражение будет эквивалентно просто t. Такое преобразование называется *η-конверсией*.

вопросы теста к теме 2

"Что такое Проблема разрешения?"

- "Задача из области оснований математики, найти алгоритм, который бы принимал в качестве входных данных описание любой проблемы разрешимости (формального языка и математического утверждения «S» на этом языке) — и, после конечного числа шагов, останавливался бы и выдавал один из двух ответов: «Истина!» или «Ложь!», — в зависимости от того, истинно или ложно утверждение «S». Ответ не требует обоснований, но должен быть верным."
- "Утверждение о том, что любое чётное число, начиная с 4, можно представить в виде суммы двух простых чисел."
- "Утверждение о том, что всякое n-мерное многообразие гомотопически эквивалентно n-мерной сфере тогда и только тогда, когда оно гомеоморфно ей."
- "Утверждение, что в классической логике любое высказывание, вне зависимости от степени сложности, истинно или ложно и третьего не дано."

"Выберите определение Лямбда-исчисления."

- "Формальная система, разработанная для формализации и анализа понятия вычислимости, по сути описывает понятие алгоритма."
- "Аксиоматическая логическая система, интерпретацией которой является алгебра высказываний."
- "Символический метод записи чисел, представление чисел с помощью письменных знаков."
- "Комплекс определений, реализующий способ определять положение и перемещение точки или тела с помощью чисел или других символов. "

"Укажите факторы, которые сделали λ-исчисление основной формализаций, применяемой в исследованиях, связанных с языками программирования."

- "Это единственная формализация, которая, хотя и с некоторыми неудобствами, действительно может быть непосредственно использована для написания программ."
- "λ-исчисление дает простую и естественную модель для таких важных понятий, как рекурсия и вложенные среды."
- "Большинство конструкций традиционных языков программирования может быть более или менее непосредственно отображено в конструкции λ-исчисления."
- "Описание исследуемых явлений производится средствами уточненного естественного языка."
- "Предназначение создаваемых моделей состоит в выявлении тех существенных факторов, которые формируют интегральные свойства объекта-оригинала."
- "Процесс построения и использования модели, т.е. такого материального или искусственно созданного кем-то объекта, который в процессе познания (изучения) замещает его оригинал, сохраняя важные (по мнению автора) или типичные черты"

"Установить соответствие между понятиями"

- "Терм"
- "Переменная"
- "Применение функции (аппликация)"
- "λ-абстракция (анонимная функция)"
- "Программный код в λ-исчислении"

и

- "x"
- "F x, где F — функция, x — подставляемое в неё значение аргумента"
- "λx.F, где x — аргумент функции, F — её тело"

- "Если x – переменная, а M – терм, то (λx.M) – терм)"
- "Если M и N – термы, то (MN) – терм."

"Выберите основные правила написания программного кода в λ-исчислении."

- "Переменные x , y , z … являются термами."
- "Если M и N – термы, то (MN) – терм."
- "Если x – переменная, а M – терм, то (λx.M) – терм)"
- "Если M и N формулы, то формулами являются M v N, M ^ N, M -> N и M <-> N"
- "(M · N) · K = M · (N · K) = M · N ·K"

"Что такое аппликация в λ-исчислении?"

- "Применение или вызов функции по отношению к заданному значению."
- "Вырезание и наклеивание (нашивание) фигурок, узоров или целых картин из кусочков бумаги, ткани, кожи, растительных и прочих материалов на материал-основу (фон)"
- "Частичное наложение звуков одной морфемы на другую, в результате чего эти звуки выполняют двойную роль."
- "Функция полученная определенным образом."

"Укажите основные соглашения о приоритете операций в λ-исчислении."

- "Аппликация левоассоциативна. "
- "Аппликация забирает себе всё, до чего дотянется вправо."
- "Аппликация забирает себе всё, до чего дотянется влево."
- "Аппликация правоассоциативна. "
- "Аппликация коммуникативна. "

"Укажите правильную иллюстрацию того, что аппликация левоассоциативна:"

- "s t u — это тоже самое, что (s t) u"
- "s t u — это тоже самое, что s (t u)"
- "s t u — это тоже самое, что (s) (t) (u)"

"Укажите правильную иллюстрацию того, что область действия лямбда-выражения простирается вправо насколько возможно."

- "λx. λy. x y x означает то же самое, что λx. (λy. ((x y) x))"
- "λx. λy. x y x означает то же самое, что λx. (λy. ((x y) λx))"
- "λx. λy. x y x означает то же самое, что λx. (λy. (λ(x y) x))"

"Укажите определение каррирования"

- "Преобразование функции от многих переменных в функцию, берущую свои аргументы по одному."
- "Преобразование функции путем последовательного сокращения значений."
- "Преобразование функции путем последовательного объединения областей значения."
- "Преобразование функции путем исключения аргументов через вырождение."

"Правильно сгруппируйте определения и примеры."

- "Связанные переменные"
- "Свободные переменные"

и

- "Переменная x в терме λx. λy. x"

- "Переменные находятся в позиции, в которой они не связаны никакой вышележащей абстракцией."
- "Переменные по которым выше в дереве разбора были λ-абстракции."
- "Переменная y в терме λx. λy. x"

"Правильно сгруппируйте определения и примеры."

- "Редукция"
- "α -преобразование"

и

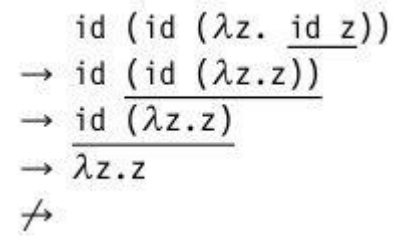
- "Процесс вычисления терма при котором в выражении (λx.t)y каждый шаг вычисления будет заключаться в замене всех вхождений переменной x внутри t на y. "
- "(λb.λx.bxx)z [b=z] <=> λx.zxx"
- "Замене в выражении λx.t имени переменной x на любое другое имя с одновременной заменой всех свободных вхождений этой переменной в выражение t, с целью ликвидации лишнего связывания переменных."
- "(λxy.x) y <=> (λxz.x) y"

"Правильно сгруппируйте определения и примеры стратегий редукции. Примеры рассматриваются на базе терма (λx.x) ((λx.x) (λz. (λx.x) z)), содержащего три редекса:"

- "Полная β-редукция."
- "Нормальный порядок вычислений."

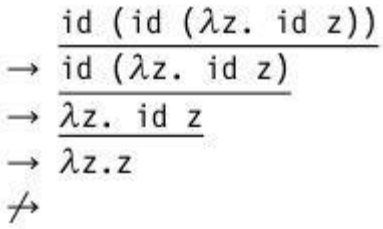
и

- "В этом случае каждый раз редекс внутри вычисляемого терма выбирается



произвольным образом. Пример вычислен от внутреннего редекса к внешнему."

- "Первым всегда сокращается самый левый, самый внешний редекс."

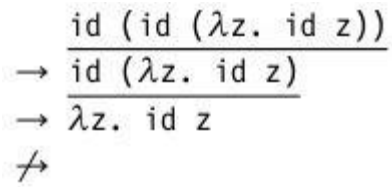


"Правильно сгруппируйте определения и примеры стратегий редукции. Примеры рассматриваются на базе терма (λx.x) ((λx.x) (λz. (λx.x) z)), содержащего три редекса:"

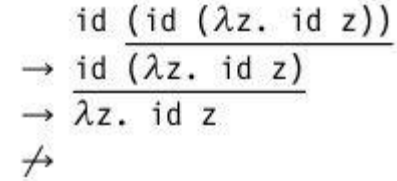
- "Вызов по имени."
- "Вызов по значению"

и

- "Порядок вычислений в этой стратегии аналогичен нормальному порядку вычислений, но к нему добавляется запрет на проведение сокращений внутри абстракции. "



- "Сокращение начинается с самого левого (внешнего) редекса, у которого в правой части стоит замкнутый терм, который нельзя вычислить далее. "



"Сопоставьте теоремы"

- "Теорема Карри"
- "Теорема Чёрча-Россера"

и

- "Если у терма есть нормальная форма, то последовательное сокращение самого левого внешнего редекса приводит к ней"
- "Если терм X редуцируется к термам Y1 и Y2 , то существует терм L , к которому редуцируются и терм Y1 и терм Y2. "
- "Каждый типизируемый терм имеет нормальную форму и каждая возможная последовательность редукций, начинающаяся с типизируемого терма, завершается"

вопросы теста к теме 3

"Что такое парадигма программирования?"

- "Совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию)"
- "Совокупность стилей, определяющих идеи написания компьютерных программ"
- "Совокупность фундаментальных научных установок, представлений и терминов, принимаемая и разделяемая научным сообществом и объединяющая большинство его членов"
- "Список словоформ, принадлежащих одной лексеме и имеющих разные грамматические значения"

"*Укажите основные модели программирования*"

1 "Императивное программирование"
2 "Декларативное программирование"
3 "Функциональное программирование"
4 "Объектно-ориентированное программирование"
5 "Линейное программирование"
6 "Последовательное программирование"
7 "Объектно-образное программирование"
8 "Каскадное программирование"
9"Итеративное программирование"

"*Установите соответствие между понятиями и определениями*"

– "Императивное программирование"
– "Объектно-ориентированное программирование"
– "Декларативное программирование"

– "Функциональное программирование"

– "парадигма программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования"

– "парадигма программирования, в которой задаётся спецификация решения задачи, то есть описывается, ЧТО представляет собой проблема и ожидаемый результат"

– "парадигма программирования, которая описывает процесс вычисления в виде инструкций, изменяющих состояние программы, очень похожа на приказы, выражаемые повелительным наклонением в естественных языках, то есть это последовательность команд, которые должен выполнить компьютер"

– "парадигма программирования в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании"

– "парадигма разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков"

"Установить соответствие между понятиями и определениями"

- "Интерпретация"
- "Компиляция"
- "пооператорный (покомандный, построчный) анализ, обработка и тут же выполнение исходной программы или запроса"
- "трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду"
- "трансляция команды составленной на исходном языке высокого уровня, в эквивалентную команду на низкоуровневом языке, близком машинному коду"
- "перевод исходного кода программы в байт-код"

"Укажите свойства функциональных языков"

1. "краткость и простота"
2. "строгая типизация"
3. "функции — это значения"
4. "чистота (отсутствие побочных эффектов)"
5. "отложенные (ленивые) вычисления"
6. "выразительность"
7. "концептуальная целостность"
8. "универсальность"

"Выберите определение функции"

1. "Соответствие между элементами двух множеств, установленное по такому правилу, что каждому элементу одного множества ставится в соответствие некоторый элемент из другого множества"

2. "Соответствие между двумя выражениями используемых в вычислении"

3. "Преобразование данных выполненное по определенному правилу"

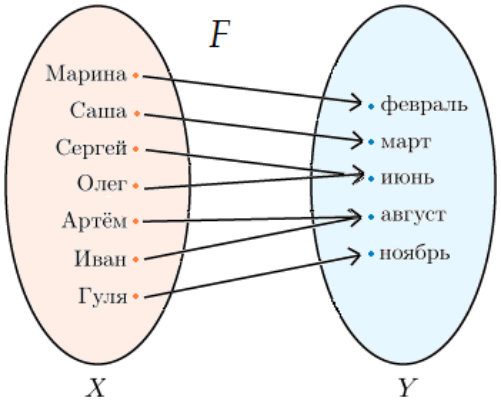
4. "Преобразование двух множеств установленное по определенному правилу, при котором преобразование выполняется над каждым элементом преобразуемого множества"

"Что такое функция высшего порядка?"

- "Функция, принимающая в качестве аргументов другие функции или возвращающая другую функцию в качестве результата"
- "Функция, принимающая в качестве аргументов значения со степенью больше 1."
- "Функция, возвращающая в качестве результата значения со степенью больше 1."

•"Функция, принимающая в качестве аргументов производные и первообразные высших порядков"

"Установить соответствие обозначений и понятий на данном рисунке"



6. "X"
7. "Y"
8. "F"

и

1. "Область определения"
2. "Область значений"
3. "Функция"
4. "Область отображения"
5. "Правило обработки"

"Установить соответствие."

1. "f(a)=p, f(b)=q, f(c)=q, f(d)=r"
2. "f(x)=2x+5"

и

- "Определение функции перечислением"
- "Определение функции аналитически"
- "Графическое определение функции"
- "Словесное описание функции"

"Установить соответствие"

- "Функция частично определенная над множеством A"
- "Функция полностью определенная над множеством A"

и

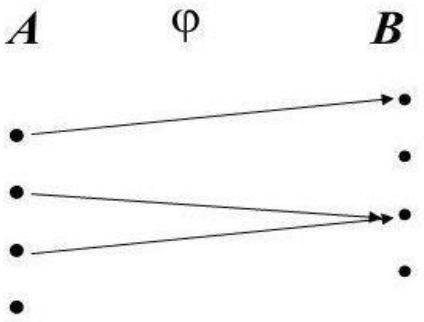
- "если в множестве A существуют элементы, для которых образ посредством этой функции не определен"
- "если она не является частичной над множеством"
- "если области определения и значения функции совпадают"
- "если области определения и значения функции не совпадают"
- "если области определения и значения функции пересекаются хотя бы в одной точке"

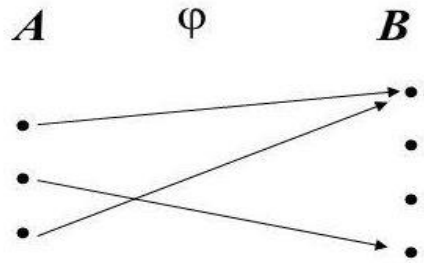
"Провести классификацию."

- "Частично определенная функция"

- "Полностью определенная функция"

и





"Провести классификацию. Символ ∈ означает «принадлежит». R - Множество вещественных чисел"

- "Частично определенная функция"
- "Полностью определенная функция"

и

- "f(x)=1/x, x ∈ R"
- "f(x)=sqr(x), x ∈ R (sqr - Извлечение квадратного корня)"
- "f(x)=x^2, x ∈ R"
- "f(x)=sqr(x), x ∈ R, x>0 (sqr - Извлечение квадратного корня)"
- "f(x)=1/x, x ∈ R, x≠0"

"Есть функция МАКС(a,b), которая выдает в качестве результата наибольшее из аргументов. Укажите правильно составленные функции."

- "НАИБ(a,b,c)=МАКС(МАКС(a,b),c)"
- "НАИБ(a,b,c)=МАКС(a,МАКС(b,c))"
- "НАИБ(a,b,c)=МАКС(МАКС(a,b),МАКС(b,c))"
- "НАИБ(a,b,c)=МАКС(a,b,c)"

"Есть функция МАКС(a,b), которая выдает в качестве результата наибольшее из двух аргументов, и функция НАИБ(a,b,c), которая выдает в качестве результата наибольшее из трех аргументов. Укажите правильно составленные функции находящие наибольшее из шести аргументов (a,b,c,d,e,f)"

- "МАКС(НАИБ(a,b,c),НАИБ(d,e,f))"
- "НАИБ(МАКС(a,b),МАКС(c,d),МАКС(e,f))"
- "МАКС(МАКС(МАКС(a,b),МАКС(c,d)),МАКС(e,f))"
- "НАИБ(НАИБ(a,b,c),НАИБ(d,e,f))"

"Есть функции: "ПЛЮС (x,y) = x+y","МИНУС (x,y) = x-y", "УМН (x,y) = x*y", "ДЕЛ (x,y) = x/y"

"Используя их запишите функцию a+b*c"

- "ПЛЮС (a,УМН (b,c))"
- "ПЛЮС (a,УМН (б,c))"
- "ПЛЮС (a,УМН (в,c))"

"Есть функции: "ПЛЮС (x,y) = x+y","МИНУС (x,y) = x-y", "УМН (x,y) = x*y", "ДЕЛ (x,y) = x/y"

"Используя их запишите функцию (2*x+3*y)/(x-y)"

- "ДЕЛ (ПЛЮС (УМН (2,x),УМН (3,y)),МИНУС (x,y))"
- "ДЕЛ (ПЛЮС (УМН (2,x),УМН (3,y)),МИНУС (x,y))"

Практическая работа к теме 4.

1. Разобрать работу с командной строкой. Вычисление простых математических выражений типа:

2*2

2+2*2 и т. д. Убедиться в том что haskell правильно понимает порядок вычислений и применения скобок.

2. Убедиться в том, что haskell правильно оценивает истинность. (3>2 и 2>3)

Идеология написания программ в Haskell – программы пишутся в блокноте и загружаются. Затем описанная функция вызывается с аргументами.

3. Написание функций вычисляющих математические выражения при заданных аргументах:

x^2, ax^2+bx+c:

*kvad (x) = x*x*

*kvadvyr (a,b,c,x) = a*x*x+b*x+c* или с использованием описанной функцией kvad - *a*kvad (x) b*x+c*

3. Рекурсивные функции. Вычисление степени числа, факториала числа, числа фибоначчи:

Математическая запись		Программа на haskell
n!	| 0!=1	factorial (0)=1
	| n!=(n-1)!*n	factorial (n) = factorial (n-1)*n
x^y =	| x^0 = 1	stepen (x,0) = 1
	|x^y= $x^{(y-1)}$*x	stepen (x,y) = stepen (x,y-1)*x
F_x =	|F_0=0	fib (0) = 0
	|F_1=1	fib (1) = 1
	|F_x=F_{x-2}+F_{x-1}	fib (x) = fib(x-2)+fib(x-1) -(это не список а вычисление x-ового числа фибоначчи)

5. Работа со списками. Убедиться в том, что haskell понимает списки как перечисления. Ввести в командной строке [1..10], [1,3..10], [2,4..10]

6. Написание функций выходящих на экран списки:

Все натуральные числа от 1 до n: *spisnat n = [1..n]* или *spisnat n =[x|x←[1..n]]*

Все четные числа от 1 до n: *spischet n = [2,4..n]* или *spischet n = [x|x←[2,4..n]]*

Все нечетные числа от 1 до n: *spisnechet n = [1,3..n]* или *spisnechet n = [x|x←[1,3..n]]*

7. Комбинирование функций:

Задание	Программа на haskell
Список квадратов натуральных чисел.	spiskvat n=[x*x|x<-[1..n]]

Список факториалов.	fact(0)=1 fact(x)=fact(x-1)*x spisfact n=[fact(x)|x<-[1..n]]
Список степеней двойки.	spisstepdvoik n =[2^x|x←[1..n]]
Список треугольных чисел Ферма.	treugferma n= [(x*(x+1))/2|x<-[1..n]]
Список пирамидальных чисел Ферма.	piramferma n= [(x*(3*x-1))/2|x<-[1..n]]

Практическая работа к теме 5.

1. Составить программу, которая выводит на экран все алфавитно-цифровые последовательности, содержащиеся в файле. Алфавитно-цифровой называется непрерывная последовательность цифр и букв латинского алфавита. Пример: если файл содержит символы asdf-+^&@qwert1y!@# то результатом работы программы должно быть: asdf qwert1y
2. Составить программу, принимающую в командной строке имя файла и один из двух символов 'u' или 'l'. В зависимости от того, какой из символов передан, она преобразует содержимое файла к верхнему или нижнему регистру и выводит на экран.

вопросы теста к теме 6

"Что такое рекурсивная функция?"

- "Это функция, которая в своей записи содержит себя же"
- "Это композиция функций от одного аргумента"
- "Это функция вызываемая с различными аргументами"
- "Это функция модифицирующая себя в ходе вычисления"

"Что необходимо, чтобы рекурсивная функция вычислялась для любого N?"

- "Необходимо, чтобы для некоторых N функция была определена нерекурсивно."
- "Необходимо, чтобы N было целое и положительное"
- "Необходимо, чтобы N было вещественным"
- "Необходимо, чтобы N было ненулевое"

"Укажите основные проблемы рекурсии"

- "Повторные вычисления одних и тех же значений"
- "Дерево рекурсивных вызовов может оказаться бесконечным"
- "Рекурсия - неконтролируемый процесс"
- "Слишком длительное время компиляции кода"

"Укажите основные способы решения проблем рекурсии"

- "Использование накапливающего параметра — аккумулятора"
- "Использование хвостовой рекурсии, которая представляет собой специальный вид рекурсии, в которой имеется единственный вызов рекурсивной функции и при этом этот вызов выполняется после всех вычислений"
- "Полностью исключить рекурсию из вычислений"
- "Использование рекурсивных определений, позволяющих при трансляции обеспечить вычисления в постоянном объёме памяти через итерацию вызываемой функции"

"Сопоставьте обозначения и названия базисных операций"

- "prefix(x,y)≡x:y≡[x|y]"
- "head(x)≡h(x)"

- "tail(x)≡t(x)"

и

- "Операция отсечения головы"
- "Операция создания пары"
- "Операция отсечения хвоста"
- "Операция модификации списка"
- "Тождественная операция"

"Укажите основные аксиомы базисных операций"

- "head (x : y) = x"
- "tail (x : y) = y"
- "prefix (head (x : y), tail (x : y)) = (x : y)"
- "head (x : y) = y"
- "tail (x : y) = x"

"Укажите правильную запись функции определения длины списка Length, через базовые операции prefix, head, tail, где"

"prefix (x, y) - операция создания пары "

"head (x) - операция отсечения головы"

"tail (x) - операция отсечения хвоста"

- "Length ([ ]) = 0" "Length (L) = 1 + Length (tail (L))"
- "Length ([ ]) = 0" "Length (L) = 1 + Length (head (L))"
- "Length ([ ]) = 0" "Length (L) = Length (tail (L)) + Length (head (L))"

"Укажите правильную запись функции слияния двух списков Append, через базовые операции prefix, head, tail, где"

"prefix (x, y) - операция создания пары "

"head (x) - операция отсечения головы"

"tail (x) - операция отсечения хвоста"

- "Append ([ ], L2) = L2" "Append (L1, L2) = prefix (head (L1), Append (tail (L1), L2))"
- "Append ([ ], L2) = L2" "Append (L1, L2) = prefix (tail (L1), Append (head (L1), L2))"
- "Append ([ ], L2) = L2" "Append (L1, L2) = prefix (head (L2), Append (tail (L1), L2))"

"Сопоставьте понятия и определения"

- "Полиморфизм"
- "Параметрический полиморфизм"
- "Специальный (ad-hoc) полиморфизм"

и

- "это возможность использования в одном и том же контексте различных программных сущностей (объектов, типов данных и т. д.) с одинаковым интерфейсом."
- "это возможность определения обобщённых структур данных и функций, поведение которых не зависит от типов значений, которыми они оперируют."
- "(или «перегрузка имён») это возможность давать одинаковые имена программным сущностям с различным поведением"

Лабораторная работа № 1

Цель работы

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

1. 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 (Hugs (Haskell98 mode)) 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.

2. 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; тип **Int** представляет целые числа фиксированной длины.
- 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.

2. 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 231 -1= 2147483647), тип **Integer** в языке Haskell может хранить целые числа произвольной длины.

3. 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 **firs**(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 храниться 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' :: Char

5.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 **last** возвращает последний элемент списка.
- Функция **tail** возвращает список без первого элемента
- Функция **init** возвращает список без последнего элемента
- Функция **null** проверяет список на пустоту.

Если в качестве аргумента этой операции будет задан пустой список, то функция выдаст значение **True**, в противном случае – **False**

- Функция **length** возвращает длину списка.
- Функция elem проверяет наличие элемента в списке.
- Функция take возвращает список, состоящий из **n** первых элементов исходного списка.
- Функция **zip** возвращает список, состоящий из пар объединенных исходных списков.
- Функция **!!** возвращает элемент, номер которого задан (начиная с 0).

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
Prelude> elem 2 [1,2,3]
True :: Bool
Prelude> take 2 [1,2,3]
[1,2] :: [Integer]
Prelude> zip ["20","30"] [1,2,3]
[("20",1),("30",2)] :: [([Char],Integer)]
Prelude> [1,2,3,4,5] !! 3
4 :: Integer

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
```

6. 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.

7. 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'

8. 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
```

*9. **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.

Задания на лабораторную работу

1. Приведите пример *нетривиальных* выражений, принадлежащих следующему типу:

- ((Char,Integer), String, [Double])
- [(Double,Bool,(String,Integer))]
- ([Integer],[Double],[(Bool,Char)])
- [[[(Integer,Bool)]]]
- (((Char,Char),Char),[String])
- (([Double],[Bool]),[Integer])
- [(Integer, (Integer,[Bool]))]
- (Bool,([Bool],[Integer]))
- [([Bool],[Double])]
- [([Integer],[Char])]

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

2. Напишите функции, решающие следующие задачи:

2.1. Три точки A, B, C лежат на одной прямой. Заданы длины AB, BC, AC. Может ли точка A лежать между точками B и C.

2.2. Чему равен угол, если два смежных с ним угла составляют в сумме заданное число градусов.

2.3. Периметр равнобедренного треугольника равен P метрам, а боковая сторона L. Найти основание треугольника.

2.4. Найти углы треугольника, если они пропорциональны заданным числам A, B, C.

2.5. В равнобедренном треугольнике боковая сторона A, а основание B. Найти высоту, опущенную на основание.

2.6. Даны координаты центров и радиусы двух окружностей на плоскости. Может ли вторая окружность целиком содержится внутри первой? Если нет, то сколько точек пересечения имеют окружности?

2.7. Можно ли из отрезков с заданными длинами A, B, C построить прямоугольный треугольник?

2.8. Можно ли из круглого листа железа диаметром D метров вырезать квадрат со стороной A метров?

2.9. Стороны треугольника A, B, C. Найти высоту, опушенную на строну C.

2.10. Высота равнобедренного треугольника H метров, основание L. Найти углы треугольника и длину боковой стороны.

2.11. По данной хорде A найти длину дуги, если она соответствует центральному углу в C градусов.

2.12. Найти точку, равноудаленную от осей координат и от точки с заданными координатами (x,y).

2.13. Даны четыре точки A, B, C, D на плоскости. Является ли четырехугольник ABCD параллелограммом?

2.14. Составить уравнение прямой, проходящей через 2 заданные точки.

2.15. Даны четыре точки A, B, C, D на плоскости. Является ли четырехугольник ABCD квадратом?

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.

Лабораторная работа № 2

Цель работы

Научиться определять рекурсивные функции. Получить представление о механизме сопоставления с образцом. Приобрести навыки определения функций для обработки списков.

1. Комментарии

Необходимость наличия комментариев в программе очевидна. К сожалению, авторы различных языков программирования расходятся между собой в вопросе о том, каким образом обозначать комментарии в коде. Haskell не стал исключением. В языке Haskell, как и в C++, определены два вида комментариев: строчные и блочные. Строчный комментарий начинается с символов -- и продолжается до конца строки (аналогом в C++ служит комментарий, начинающийся с //). Блочный комментарий начинается символами - и продолжается до символов - (аналог в C++– комментарий, ограниченный символами /* и */. *Разумеется, все, что является комментарием, игнорируется интерпретатором или компилятором языка Haskell. Пример:*

```
f x = x -- Это комментарий g x y = - Это тоже комментарий.
Только длиннее. – x + y
```

2. Рекурсия

В императивных языках программирования основной конструкцией является цикл. В Haskell вместо циклов используется рекурсия. Функция называется рекурсивной, если она вызывает сама себя (или, точнее, определена в терминах самой себя). Рекурсивные функции существуют в императивных языках, но используются не столь широко. Одной из простейших рекурсивных функций является факториал:

```
factorial :: Integer -> Integer
factorial n = if n == 0 then 1 else n * factorial (n - 1)
```

(Заметьте, что мы пишем factorial (n - 1)*, а не* factorial n -1 *– вспомните о приоритетах операций.) Использование рекурсии может вызвать трудности. Концепция рекурсии напоминает о применяющемся в математике приеме доказательства по индукции. В нашем определении факториала мы выделяем "базу индукции" (случай* n == 0*) и "шаг индукции" (переход от* factorial n *к* factorial (n - 1)*. Выделение таких компонен – важный шаг вопределении рекурсивной функции.*

3. Операция выбора и правила выравнивания

Ранее был рассмотрен условный оператор. Его естественным продолжением является оператор выбора case*, аналогичный конструкции* switch *языка Си. Предположим, нам надо определить некоторую (довольно странную) функцию, которая возвращает* 1*, если ей передан аргумент* 0*;* 5*, если аргумент был равен* 1*;* 2*, если аргумент равен* 2 *и* -1 *во всех остальных случаях. В принципе, эту функцию можно записать с помощью операторов* if*, однако результат будет длинным и малопонятным. В таких случаях помогает использование* case*:*

```
f x = case x of
0 -> 1
```

```
1 -> 5
2 -> 2
_ -> -1
```

Синтаксис оператора case *очевиден из приведенного примера; следует только сделать замечание, что символ _ аналогичен конструкции* default *в языке Си. Однако у внимательного читателя может возникнуть закономерный вопрос: каким образом интерпретатор языка Haskell распознает, где закончилось определение одного случая и началось определение другого? Ответ заключается в том, что в языке Haskell используется двумерная система структурирования текста (аналогичная система используется в более широко известном языке Python). Эта система позволяет обойтись без специальных символов группировки и разделения операторов, подобным символам* { , } *и* ; *языка Си. В действительности в языке Haskell также можно использовать эти символы в том же смысле 1 . Так, вышеприведенную функцию можно записать и таким образом (демонстрирующем, как не надо оформлять тексты программ):*

```
f x = case x of
0 -> 1;
1 -> 5;
2 -> 2;
_ -> -1
```

Такой способ явно задает группировку и разделение конструкций языка. Однако можно обойтись и без него. Общее правило таково. После ключевых слов where , let , do *и* of *интерпретатор вставляет открывающую скобку ({) и запоминает колонку, в которой записана следующая команда. В дальнейшем перед каждой новой строкой, выровненной на запомненную величину, вставляется разделяющий символ ';'. Если следующая строка выровнена меньше (т.е. ее первый символ находится левее запомненной позиции), вставляется закрывающая скобка. Это может выглядеть несколько сложно, но в действительности все довольно просто. Применяя описанное правило к определению функции* f *, получим, что оно воспринимается интерпретатором следующим образом:*

```
f x = case x of
;0 -> 1
;1 -> 5
;2 -> 2
;_ -> -1
```

В любом случае можно не использовать этот механизм и всегда явно указывать символы {,} и ;. *Однако, помимо экономии на количестве нажатий клавиш, применение описанного правила приводит к тому, что получаемые программы более "читабельны". Таким образом, для лабораторных работ предлагается сделать употребление такого оформления обязательным. Необходимо сделать еще одно замечание. Поскольку в программе на языке Haskell пробелы являются значимыми, необходимо быть внимательными к использованию символов табуляции. Интерпретатор полагает, что символ табуляции равен 8 пробелам. Однако некоторые текстовые редакторы позволяют настраивать отображение табуляции и делать его эквивалентным другому числу пробелов (например, по умолчанию в редакторе Visual Studio табуляция отображается как 4 пробела). Это может привести к ошибкам, если совмещать в одной программе пробелы и табуляцию. Лучше всего при программировании на Haskell вообще не использовать табуляцию (многие редакторы позволяют вводить по нажатию клавиши табуляции указанное число пробелов). 1 За тем исключением, что в Haskell, как и в языке Паскаль, символ ` ; ' используется как разделитель операторов, а не как признак завершения оператора*

4. Кусочное задание функций

Функции могут быть определены кусочным образом (вспомните понятие кусочнопостоянных или кусочно-линейных функций в математике). Это означает, что можно определить одну версию функции для определенных параметров и другую версию для других параметров. Так, функцию f *из предыдущего раздела можно определить следующим образом:*

```
f 0 = 1
f 1 = 5
f 2 = 2
f _ = -1
```

Порядок определения в данном случае важен. Если бы мы записали сначала определение f _ = -1*, то* f *возвращала бы* -1 *для любого аргумента. Если бы мы вовсе не указали эту строчку, мы получили бы ошибку, если бы попытались вычислить ее значение для аргумента, отличного от* 0, 1 *или* 2. *Такой способ определения функций довольно широко используется в языке Haskell. Он зачастую позволяет обойтись без операторов* if *и* case . *Так, функцию факториала можно определить в таком стиле:*

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

5. Сопоставление с образцом

Помимо рекурсивных функций на целых числах, можно определять рекурсивные функции на списках. В этом случае "базой рекурсии" будет пустой список ([] *). Определим функцию вычисления длины списка (поскольку имя* length *уже занято стандартной библиотекой, назовем ее* len*):*

```
len [] = 0
len s = 1 + len (tail s)
```

Вспомним, что список, первым элементом которого (головой) является x *, а остальные элементы (хвост) задаются списком* xs *, записывается как* x:xs *. Оказывается, подобную конструкцию можно применять при описании функции:*

```
len [] = 0
len (x:xs) = 1 + len xs
```

(Строку xs *следует понимать, как множественное число от* x*, образованное по правилам английского языка.) Приведем еще один пример. Функцию, принимающую на вход пару чисел и возвращающую их сумму, можно определить таким образом:*

```
sum_pair p = fst p + snd p
```

Однако что делать, если необходимо определить функцию, принимающую тройку чисел и возвращающую их сумму? В нашем распоряжении нет функций, подобных fst *и* snd *, для извлечения элементов тройки. Оказывается, можно записывать такие функции следующим образом:*

```
sum_pair (x,y) = x + y
sum_triple (x,y,z) = x + y + z
```

Такой прием называется сопоставление с образцом2. Он является очень мощной конструкцией языка, применяемой во многих его местах, в частности, в аргументах функций и в вариантах оператора case . *"Образцы", записываемые в аргументах функции, "сопоставляются" с переданными в нее фактическими параметрами. Если происходит сопоставление с образцом, упомянутые в нем переменные получают соответствующие значения. Если эти значения не нужны при вычислении 2 От английского "pattern matching" функции (как в функции* my_tail *в следующем примере), то, чтобы не вводить лишних имен, можно использовать символ _. Он означает образец, с которым может сопоставиться любое значение, но само это значение не связывается ни с какой переменной. Следующие примеры показывают различные варианты применения сопоставления с образцом:*

```
-- Функция суммирования двух первых элементов списка
f1 (x:y:xs) = x + y
```

```
-- Определение функции, аналогичной head
my_head (x:xs) = x
-- Определение функции, аналогичной tail.
-- Мы используем _, поскольку нам не нужно значение
-- первого элемента списка my_tail (_:xs) = xs
-- Функция извлечения первого элемента тройки
 fst3 (x,_,_) = x
```

Сопоставление с образцом можно применять и в операторе case :

```
-- Еще одно определение функции длины списка
my_length s = case s of
 [] -> 0
 (_:xs) -> 1 + my_length xs
```

Можно задавать довольно сложные образцы. Определим функцию, принимающую список пар чисел и возвращающую сумму их разностей (т.е. f [(x1,y1),(x2,y2),...,(xn,yn)]= (x1 - y1) + (x2 -y2) + ... + (xn - yn) *):*

```
f [] = 0
f ((x,y):xs) = (x - y) + f xs
```

6. Построение списков

При определении функций, возвращающих список, часто используется оператор :.

Например, функция, принимающая список чисел и возвращающая список их квадратов, может быть определена следующим образом:

```
square [] = []
square (x:xs) = x*x : square xs
```

Ещё один способ построения списков - при помощи ограничивающих параметров:

```
--обратите внимание: здесь генерируется список ограниченной длины square n = [i^2|i<-[1..n]]
```

7. Некоторые полезные функции

При выполнении лабораторной работы могут понадобиться следующие стандартные функции языка Haskell:

- even – *возвращает True для четного аргумента и False для нечетного.*
- odd – *аналогично предыдущей, но аргумент проверяется на нечетность.*

Задания на лабораторную работу

*1.1.Определите функцию, принимающую на вход целое число и возвращающую список, содержащий элементов, упорядоченных по возрастанию.*n n

1.2.Список натуральных чисел.

1.3.Список нечетных натуральных чисел.

1.4.Список четных натуральных чисел.

1.5.Список кубов натуральных чисел.

1.6.Список факториалов.

1.7.Список степеней десятки.

1.8.Список треугольных чисел3

1.9.Список пирамидальных чисел4

2. Определите следующие функции:

2.1.Функция, вычисляющая сумму квадратов целых чисел от 1 до n .

2.2.Функция, вычисляющая произведение целых чисел от 1 до n .

2.3.Функция, принимающая на входе список вещественных чисел и вычисляющую их арифметическое среднее. Функция, принимающая на входе список вещественных чисел и вычисляющую их геометрическое среднее.

2.4.Функция вычленения n -го элемента из заданного списка.

2.5.Функция вычленения n-m -го элемента из заданного списка.

2.6.Функция умножения элементов двух списков. Возвращает список, составленный из произведения элементов списков-параметров. Учесть, что переданные списки могут быть разной длины.

2.7.Функция перестановки местами соседних четных и нечетных элементов в заданном списке.

2.8.Функция, которая вычисляет , исходя из следующих соображений. Пусть необходимо возвести в степень . Если n *четно, т.е.* 2n 2 n n = 2k , *то* 2n = 22k =(2k). *Если* n *нечетно, т.е.* n = 2k *+1 , то* 2n = 22k+1 = 2·(2k). *Функция не должна использовать оператор ^ или любую функцию возведения в степень из стандартной библиотеки.*

2.9.Функция, которая удаляет из заданного списка целых чисел все четные числа. Например: по списку [1,4,5,6,10] *должно возвращаться* [1,5] .

2.10.Функция f, *которая удаляет из заданного списка целых чисел все числа, меньшие заданного . Например:* f 5 [1,4,5,6,10] *должно возвращаться* [5,6,10] .

2.11. Функция, которая удаляет пустые строки из заданного списка строк. Например: при ["", "Hello", "", "", "World!"] функция возвращает ["Hello","World!"] . Функция типа [Bool] -> Integer , возвращающая количество элементов списка, равных False .

2.12.Функция, которая меняет знак всех положительных элементов списка чисел, например: по [-1, 0, 5, -10, -20] *дает* [-1,0,-5,-10,-20]

2.13. Функция f *типа* Char -> String -> String *, которая принимает на вход строку и символ и возвращает строку, в которой удалены все вхождения символа. Пример:* f 'l' "Hello world!" *должно возвращать* "Heo word!".

2.14.Функция f *типа* Char -> String -> String *, которая принимает на вход строку и символ и возвращает строку, в которой продублированы все вхождения символа. Пример:* f 'o' "Hello world!" *должно возвращать* "Helloo woorld!".

Примечание n-е треугольное число равно количеству одинаковых монет, из которых можно построить равносторонний треугольник, на каждой стороне которого укладывается n монет. Нетрудно убедиться, что

$t_1 = 1$, $t_n = n + t_{n-1}$

n -е пирамидальное число P_n *равно количеству одинаковых шаров, из которых можно построить правильную пирамиду с треугольным основанием, на каждой стороне которой укладывается n шаров.*

Нетрудно убедиться, что $p_1 = 1$, $p_n = p_{n-1} + t_n$

2.15.Функция f типа Char -> Char -> String -> String , которая заменяет в строке указанный символ на заданный. Пример f 'e' 'i' "eigenvalue" возвращает "iiginvalui"

Лабораторная работа № 3

1. Функции высшего порядка

Рассмотрим две задачи. Пусть задан список чисел. Необходимо написать две функции, первая из которых возвращает список квадратных корней этих чисел, а вторая – список их логарифмов. Эти функции можно определить так:

```
sqrtList [] = []
sqrtList (x:xs) = sqrt x : sqrtList xs

logList [] = []
logList (x:xs) = log x : logList xs
```

Можно заметить, что эти функции используют один и тот же подход, и все различие между ними заключается в том, что в одной из них для вычисления элемента нового списка используется функция квадратного корня, а в другой – логарифм. Можно ли абстрагироваться от конкретной функции преобразования элемента? Оказывается, можно. Вспомним, что в Haskell *функции являются элементами «первого класса»: их можно передавать в другие функции в качестве параметров. Определим функцию* transformList*, которая принимает два параметра: функцию преобразования и преобразуемый список.*

```
transformList f [] = []
transformList f (x:xs) = f x : transformList f xs
```

Теперь функции sqrtList *и* logList *можно определить так:*

```
sqrtList l = transformList sqrt l
logList l = transformList log l
```

Или, с учетом каррирования:

```
sqrtList = transformList sqrt
logList = transformList log
```

1.1 Функция map

В действительности функция, полностью аналогичная transformList *,уже определена в стандартной библиотеке языка и называется* map *(от англ. map – отображение). Она имеет следующий тип:*

```
map :: (a -> b) -> [a] -> [b]
```

Это означает, что ее первым аргументом является функция типа a->b *,отображающая значения произвольного типа* a *в значения типа* b *(вообще говоря, эти типы могут совпадать). Вторым аргументом функции является список значений типа* a*. Тогда результатом функции будет список значений типа* b *. Функции, подобные* map *, принимающие в качестве аргументов другие функции, называются* **функциями высшего порядка**. *Их очень широко используют при написании функциональных программ. С их помощью можно явно отделить частные детали реализации алгоритма (например, конкретную функцию преобразования в* map*) от его высокоуровневой структуры (поэлементное преобразование списка). Алгоритмы, представленные с использованием функций высшего порядка, как правило, более компактны и наглядны, чем реализации, ориентированные на конкретные частности.*

1.2 Функция filter

Следующим примером широко используемой функции высшего порядка является функция filter*. По заданному предикату (функции, возвращающей булевское значение) и списку она возвращает список тех элементов, которые удовлетворяют заданному предикату:*

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x = x : filter xs
| otherwise = filter xs
```

Например, функция, получающая из списка чисел его положительные элементы, определяется так:

```
getPositive = filter isPositive
isPositive x = x > 0
```

1.3 Функции foldr *и* foldl

Более сложным примером являются функции foldr *и* foldl*. Рассмотрим функции, возвращающие сумму и произведение элементов списка:*

```
sumList [] = 0
sumList (x:xs) = x + sumList xs
```

```
multList [] = 1
multList (x:xs) = x * multList xs
```

Здесь также можно увидеть общие элементы: начальное значение (0 для суммирования, 1 для умножения) и функция, комбинирующая значения между собой. Функция foldr *является очевидным обобщением такой схемы:*

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Функция foldr *принимает в качестве первого аргумента комбинирующую функцию (заметим, что она может принимать аргументы разных типов, но тип результата должен совпадать с типом второго аргумента). Вторым аргументом функции* foldr *является начальное значение для комбинирования. Третьим аргументом передается список. Функция осуществляет «свертку» списка в соответствие с переданными параметрами. Для того, чтобы лучше понять, как работает функция* foldr *, запишем ее определение с использованием инфиксной нотации:*

```
foldr f z [] = z
foldr f z (x:xs) = x `f` (foldr f z xs)
```

Представим список элементов [a,b,c,...,z] *с использованием оператора : . Правило применения функции* foldr *таково: все операторы : заменяются на применение функции* f *в инфиксном виде (`f`), а символ пустого списка* [] *заменяется на начальное значение комбинирования. Шаги преобразования можно изобразить так (предполагаем, что начальное значение равно* init *)*

```
 [a,b,c,...,z]
a : b : c : ... z : []
a : (b : (c : (... (z : []))))
a `f` (b `f` (c `f` (... (z `f` init)...)))
```

С помощью функции foldr *функции суммирования и умножения элементов списка определяется так:*

```
sumList = foldr (+) 0
multList = foldr (*) 1
```

Рассмотрим, как вычисляются значения этих функций на примере списка [1,2,3] :

```
[1,2,3]
1 : 2 : 3 : []
1 : (2 : (3 : []))
1 + (2 + (3 + 0))
```

Аналогично для умножения:

```
[1,2,3]
1 : 2 : 3 : []
1 : (2 : (3 : []))
1 * (2 * (3 * 1))
```

Название функции происходит от английского слова fold *– сгибать, складывать (например, лист бумаги). Буква r в названии функции происходит от слова* right *(правый) и показывает ассоциативность применяемой для свертки функции. Так, из приведенных примеров видно, что применение функции группируется вправо. Определение функции* foldl*, где* l *указывает на то, что применение операции группируется влево, приведено ниже:*

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Шаги преобразования можно изобразить налогично:

```
[a,b,c,...,z]
```

```
[]:a : b : c : ... : z
(([]: a) : b) : c) : ... : z
(((init `f` a) `f` b) `f` c) `f` ... `f` z
```

Для ассоциативных операций, таких как сложение и умножение, функции foldr *и* foldl *эквивалентны, однако если операция не ассоциативна, их результат будет отличаться:*

```
Main>foldr (-) 0 [1,2,3]
2
Main>foldl (-) 0 [1,2,3]
-6
```

Действительно, в первом случае вычисляется величина 1 - (2 - (3 - 0)) = 2*, а во–величина* ((0 - 1) - 2) - 3 = -6.

1.4 Другие функции высшего порядка

В стандартной библиотеке определена функция zip . *Она преобразует два списка в список пар:*

```
zip :: [a] -> [b] -> [(a,b)]
zip (a:as) (b:bs) = (a,b):zip as bs
zip _ _ = []
```

Пример применения:

```
Prelude>zip [1,2,3] ['a','b','c']
[(1,'a'),(2,'b'),(3,'c')]
Prelude>zip [1,2,3] ['a','b','c','d']
[(1,'a'),(2,'b'),(3,'c')]
```

Заметьте, что длина результирующего списка равна длине самого короткого исходного списка. Обобщением этой функции является функция высшего порядка zipWith, *«соединяющая» два списка с помощью указанной функции:*

```
zipWith :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs) = z a b : zipWith z as bs
zipWith _ _ _ = []
```

С помощью этой функции легко определить, например, функцию поэлементного суммирования двух списков:

```
sumList xs ys = zipWith (+) xs ys
```

или, с учетом каррирования:

```
sumList = zipWith (+)
```

2. Лямбда-абстракции

При использовании функций высшего порядка зачастую необходимо определять много небольших функций. С ростом объема программы необходимость придумывать имена для вспомогательных функций все больше мешает. Однако в языке Haskell*, как и в лежащем в его основе лямбда-исчислении, можно определять безымянные функции с помощью конструкции лямбда-абстракции. Например, безымянные функции, возводящая свой аргумент в квадрат, прибавляющие единицу и умножающие на два, записывается следующим образом:*

```
\x -> x * x
\x -> x + 1
\x -> 2 * x
```

Их теперь можно использовать в аргументах функций высших порядков. Например, функцию для возведения элементов списка в квадрат можно записать так:

```
squareList l = map (\x -> x * x) l
```

Функция getPositive *может быть определена следующим образом:*

```
--Функции для получения положительных элементов списка
getPositive = filter (\x -> x> 0)
```

Можно определять лямбда-абстракции для нескольких переменных:

```
\x y -> 2 * x + y
```

Лямбда-абстракции можно использовать наравне с обычными функциями, например, применять к аргументам:

```
Main>(\x -> x + 1) 2
3
Main>(\x -> x * x) 5
25
Main>(\x -> 2 * x + y) 1 2
4
```

С помощью лямбда-абстракций можно определять функции. Например, запись square = \x -> x * x

полностью эквивалентна

square x = x * x *3.*

3. Секции

Функции можно применять частично, т. е. не задавать значение всех аргументов. Например, если функция add *определена как*

```
add x y = x + y
```

то можно определить функцию inc*, увеличивающую свой аргумент на* 1 *следующим образом:*

```
inc = add 1
```

Оказывается, бинарные операторы, как встроенные в язык, так и определенные пользователям, также можно применять лишь к части своих аргументов (поскольку количество аргументов у бинарных операторов равно двум, эта часть состоит из одного аргумента). Бинарная операция, примененная к одному аргументу, называется **секцией** *. Пример:*

```
(x+) = \y -> x+y
(+y) = \x -> x+y
(+) = \x y -> x+y
```

Скобки здесь обязательны. Таким образом, функции add *и* inc *можно определить так:*

```
add = (+)
inc = (+1)
```

Секции особенно полезны при использовании их в качестве аргументов функций высшего порядка. Вспомним определение функции для получения положительных элементов списка:

```
getPositive = filter (\x -> x > 0)
```

С использованием секций она записывается более компактно:

```
getPositive = filter (>0)
```

Функция для удвоения элементов списка:

```
doubleList = map (*2)
```

Задания на лабораторную работу

1. Определите следующие функции с использованием функций высшего порядка:

1.1 Функция вычисления геометрического среднего элементов списка вещественных чисел с использованием функции foldr *. Функция должна осуществлять только один проход по списку.*

1.2 Функция, вычисляющая скалярное произведение двух списков (используйте функции foldr *и* zipWith *).*

1.3 Функция countNegat *, возвращающая количество отрицательных элементов в списке.*

1.4 Функция quicksort *, осуществляющая быструю сортировку списка по следующему рекурсивному алгоритму. Для того, чтобы отсортировать список* xs*, из него выбирается первый элемент (обозначим его* x*). Остальной список делится на две части: список, состоящий из элементов* xs*, меньших* x *и список элементов, больших* x *. Эти списки сортируются (здесь проявляется рекурсия, поскольку они сортируются этим же алгоритмов), а затем из них составляется результирующий список вида* as ++ [x] ++ bs*, где* as *и* bs *– отсортированные списки меньших и больших элементов соответственно.*

1.5 Определенная в предыдущем пункте функция quicksort *сортирует список в порядке возрастания. Обобщите ее: пусть она принимает еще один– функцию сравнения типа* a -> a-> Bool *и сортирует список в соответствие с нею.*

2. Вернитесь к заданиям из лабораторной работы № 2 (вторая часть заданий) и реализуйте их с помощью функций высшего порядка. Постарайтесь полностью исключить из определений функций явный проход по списку.

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 is Included, 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 is Triangle, 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.

Порядок выполнения лабораторной работы

Лабораторная работа состоит из двух заданий. Пункт 1 – для всех один и тот же, пункт 2 – по лабораторной работы № 2.

Контрольные вопросы

1. Функции высшего порядка

2. Лямбда-абстракции

3. Секции

Лабораторная работа № 4

Цель работы. Научиться определять рекурсивные функции. Получить представление о механизме сопоставления с образцом.

Задания на лабораторную работу

Определите функцию, принимающую на вход целое число n и возвращающую список, содержащий n элементов, упорядоченных по возрастанию.

1) Список натуральных чисел.
2) Список нечетных натуральных чисел.
3) Список четных натуральных чисел.
4) Список квадратов натуральных чисел.
5) Список факториалов.
6) Список степеней двойки.
7) Список степеней тройки.
8)Список кубов натуральных чисел
9) Список треугольных чисел.

(n-е треугольное число t_n равно количеству одинаковых монет, из которых можно построить равносторонний треугольник, на каждой стороне которого укладывается *n* монет. Нетрудно убедиться, что $t_1 = 1$ и $t_n = n + t_{n-1}$)

10) Список пирамидальных чисел.

(n-е пирамидальное число p_n равно количеству одинаковых шаров, из которых можно построить правильную пирамиду с треугольным основанием, на каждой стороне которой укладывается *n* шаров. Нетрудно убедиться, что $p_1 = 1$ и $p_n = t_n + p_{n-1}$*)*

Построить функции, вычисляющие N-ый элемент следующих рядов.

1. $F(x, n) = x^n$
2. $F(x, n) = x^{n+3/n}$
3. $F(x, n) = (x+x^2)^n$
4. $F(n) = \Sigma_{i = 1,n}\ i$
5. $F(n) = \Sigma_{j = 1,n}\ (\Sigma_{i = 1,j}\ i)$
6. $F(n) = \Sigma_{i = 1,p}\ n^{-i}$
7. $F(n) = 2^n$
8. $F(n) = 3^n$
9. $F(n) = e^n = \Sigma_{i = 0,\infty}\ (n^i / i!)$
10. $F(n) = n^n$

Лабораторная работа № 5

Цель работы. Приобрести навыки определения функций для обработки списков.

Задания на лабораторную работу

Построить следующие бесконечные списки.

1) Список натуральных чисел.
2) Список нечетных натуральных чисел.
3) Список четных натуральных чисел.
4) Список квадратов натуральных чисел.
5) Список факториалов.
6) Список степеней двойки.
7) Список степеней тройки.
8)Список кубов натуральных чисел
9) Список треугольных чисел.

(n-е треугольное число t_n равно количеству одинаковых монет, из которых можно построить равносторонний треугольник, на каждой стороне которого укладывается *n* монет. Нетрудно убедиться, что $t_1 = 1$ и $t_n = n + t_{n-1}$)

10) Список пирамидальных чисел.

(n-е пирамидальное число p_n равно количеству одинаковых шаров, из которых можно построить правильную пирамиду с треугольным основанием, на каждой стороне которой укладывается *n* шаров. Нетрудно убедиться, что $p_1 = 1$ и $p_n = t_n + p_{n-1}$*)*

Определите следующие функции:

1) Функция, принимающая на входе список вещественных чисел и вычисляющую их арифметическое среднее. Постарайтесь, чтобы функция осуществляла только один проход по списку.

2) Функция вычленения n-го элемента из заданного списка.

3) Функция сложения элементов двух списков. Возвращает список, составленный из сумм элементов списков - параметров. Учесть, что переданные списки могут быть разной длины.

4) Функция перестановки местами соседних четных и нечетных элементов в заданном списке

5) Функция removeOdd, которая удаляет из заданного списка целых чисел все нечетные числа. Например: removeOdd [1,4,5,6,10] должен возвращать [4,10].

6) Функция removeEmpty, которая удаляет пустые строки из заданного списка строк. Например:

removeEmpty ["", "Hello", "", "", "World!"] возвращает ["Hello", "World!"].

7) Функция countTrue : [Bool] > Integer , возвращающая количество элементов списка, равных True.

8)Функция makePositive, которая меняет знак всех отрицательных элементов списка чисел, например: makePositive [-1, 0, 5, -10, -20] дает [1,0,5,10,20]

9) Функция delete : : Char > String > String , которая принимает на вход строку и символ и возвращает строку, в которой удалены все вхождения символа. Пример: delete ' l' "Hello world! " должно возвращать "Heo word!".

10) Функция substitute:: Char > Char > String > String которая заменяет в строке указанный символ на заданный. Пример: substitute 'e' 'i' "eigenvalue" возвращает "iiginvalui"

Лабораторная работа № 6

Цель работы. Приобрести навыки обработки списков и строк.

Задания на лабораторную работу

1. addStars :: [String] -> [String]

Составить список из строк исходного списка, добавив в начало каждой строки списка символ '*' («звездочка»), если только строка уже не начинается с этого символа.

2. hasLetters :: [String] -> Int

Определить количество строк в списке, содержащих хотя бы одну букву (буквой будем называть символ, для которого функция isAlpha :: Char -> Bool выдает значение True).

3. hasString :: [String] -> String -> Bool

Определить, имеется ли в списке, заданном первым аргументом, строка, совпадающая со строкой, заданной вторым аргументом.

4. firstSymbols :: [String] -> String

Составить строку из первых символов строк-элементов списка (пустые строки пропускать).

5. replaceToFirst :: [String] -> [String]

По заданному списку строк составить новый список, содержащий строки, составленные из первых символов исходных строк. Пустые строки оставить без изменения.

6. removeLong :: [String] -> Int -> [String]

По заданному списку строк (первый аргумент) составить новую строку, содержащую те же строки, кроме строк, длина которых превышает значение, заданное вторым аргументом.

7. truncateList :: [String] -> Int -> [String]

По заданному списку строк (первый аргумент) составить новую строку, содержащую те же строки, укороченные до длины, заданной вторым аргументом. Строки, длина которых меньше или равна второго аргумента, оставить без изменения.

8. longest :: [String] -> String

Найти в списке самую длинную строку.

9. shortest :: [String] -> String

Найти в списке самую короткую строку.

10. bracketsOnly :: [String] -> [String]

Составить список из тех строк исходного списка, которые начинаются с символа '[' и заканчиваются символом ']'.

11. removeStars :: [String] -> [String]

Составить список из строк исходного списка, в который входят все строки, кроме строк, начинающихся с символа '*' («звездочка»).

12. startsWithDigit:: [String] -> Int

Определить количество строк в списке, начинающихся с цифры (цифрой будем называть символ, для которого функция isDigit :: Char -> Bool выдает значение True).

13. notEmpty:: [String] -> Int

Найти количество непустых строк в списке.

14. numLongs:: [String] -> Int -> Int

Определить количество строк в списке, длина которых превышает значение второго аргумента.

15. allDigits:: [String] -> Int

Определить количество строк в списке, состоящих только из цифр (цифрой будем называть символ, для которого функция isDigit :: Char -> Bool выдает значение True).

16. onlyLetters:: [String] -> [String]

Составить список из строк исходного списка, состоящих только из букв (буквой будем называть символ, для которого функция isAlpha :: Char -> Bool выдает значение True).

17. lastSymbols:: [String] -> String

Составить строку из последних символов строк-элементов списка (пустые строки пропускать).

18. removeShortest :: [String] -> [String]

Составить список строк из строк исходного списка кроме самой короткой строки. Если исходный список пустой, то и результат будет пустым списком; если строк с самой маленькой длиной несколько - удалить одну из них (любую).

19. remove :: [String] -> String -> [String]

По заданному списку строк (первый аргумент) составить новую строку, содержащую те же строки, кроме строк, совпадающих со вторым аргументом.

20. removeLongest :: [String] -> [String]

Составить список строк из строк исходного списка кроме самой длинной строки. Если исходный список пустой, то и результат будет пустым списком; если строк с самой большой длиной несколько - удалить одну из них (любую).

21. hasEmpty :: [String] -> Bool

Проверить, есть ли в списке строки нулевой длины.

Лабораторная работа № 7

Лабораторные работы по курсу "Функциональное программирование"

Цель работы. Приобрести навыки определения и обработки сложных структур данных.

Задания на лабораторную работу

Дерево произвольной структуры имеет следующее описание типа данных data Tree a = Node a [Tree a] то есть каждый узел дерева имеет список поддеревьев. Определить следующие функции обработки деревьев:

1. toList :: Tree a -> [a]

Создает список содержимого всех узлов дерева (порядок несуществен)

2. hasElement :: Eq a => Tree a -> a -> Bool

Проверяет, есть ли среди узлов дерева, заданного первым аргументом узел, содержащий значение второго аргумента.

3. sumTree :: Real a => Tree a -> a

Вычисляет сумму значений, хранящихся в узлах дерева.

4. isHeap :: Ord a => Tree a -> Bool

Проверяет, верно ли, что дерево является пирамидой, то есть значение в каждом из его узлов меньше, значений, хранящихся в поддеревьях этого узла.

5. levelList :: Tree a -> Int -> [a]

Создает список значений узлов дерева, лежащих на уровне с номером, заданным вторым аргументом.

6. height :: Tree a -> Int

Вычисляет высоту дерева.

7. maxElement :: Ord a => Tree a -> a

Находит максимальный элемент в дереве.

8. minElement :: Ord a => Tree a -> a

Находит минимальный элемент в дереве.

9. deepElements :: Tree a -> Int -> [a]

Создает список значений узлов дерева, лежащих на уровнях, не меньших уровня, заданного вторым аргументом.

10. rootElements :: Tree a -> Int -> [a]

Создает список значений узлов дерева, лежащих на уровнях, не превышающих уровня, заданного вторым аргументом.

11. allEven :: Tree Integer -> Bool

Проверяет, верно ли, что все узлы дерева содержат только четные числа.

12. hasNegatives :: Tree Integer -> Bool

Проверяет, есть ли среди узлов дерева узлы, содержащие отрицательные числа.

13. isSuperHeap :: Real a => Tree a -> Bool

Проверяет, верно ли, что каждое из значений, хранящихся в узлах дерева, меньше, чем сумма корней поддеревьев этого узла (если они есть).

14. allOdds :: Tree Integer -> Bool

Проверяет, верно ли, что все узлы дерева содержат только нечетные числа.

15. isHeap :: Ord a => Tree a -> Bool

Проверяет, верно ли, что дерево является пирамидой, то есть значение в каждом из его узлов меньше, значений, хранящихся в поддеревьях этого узла.

16. levelList :: Tree a -> Int -> [a]

Создает список значений узлов дерева, лежащих на уровне с номером, заданным вторым аргументом.

17. height :: Tree a -> Int

Вычисляет высоту дерева.

18. rootElements :: Tree a -> Int -> [a]

Создает список значений узлов дерева, лежащих на уровнях, не превышающих уровня, заданного вторым аргументом.

19. maxElement :: Ord a => Tree a -> a

Находит максимальный элемент в дереве.

20. minElement :: Ord a => Tree a -> a

Находит минимальный элемент в дереве.

Практическая работа к теме 8.

1. Составить программу, которая должна считывать с командной строки имена двух файлов и выводить на экран те строки этих файлов, которые отличаются друг от друга.
2. Составить программу, принимающую имя файла и выводящая его строки, перед каждой из которых записывается ее номер.
3. Составить программу, которая считывает из файла матрицу и выводит на экран суммы столбцов этой матрицы.
4. Составить программу, которая считывает из файла матрицу и выводит на экран сумму диагональных элементов этой матрицы.
5. Составить программу, которая считывает две матрицы из файлов и записывает в третий файл матрицу, являющуюся их суммой.
6. Составить программу, которая считывает из одного файла матрицу, а из другого — вектор и записывает в третий файл результат умножения матрицы на вектор.

Практическая работа к теме 9.

Практическое задание выполняется индивидуально или в команде из двух человек.

Вариант задания и состав команды должны быть предварительно согласованы с преподавателем. Факт согласования отражается в таблице. **Несогласованные практические задания не принимаются!**

В случае командной работы задание разделяется на две части:

- базовая часть, которая совместно выполняется всеми участниками команды.
- индивидуальная часть, которая представляет собой расширение функционала базовой части и выполняется самостоятельно каждым участником команды.

Во время выполнения задания настоятельно рекомендуется следовать методическим указаниям.

По результатам работы должен быть составлен письменный отчёт, состоящий из следующих частей:

1. Постановка задачи, основной функционал приложения.
2. Модули, на который разделён проект, их взаимосвязь. Для каждого модуля должно быть указано:
   - предназначение модуля (например, работа с графикой, искусственный интеллект для игры и т.д.);
   - основные типы данных и функции, реализованные в модуле.
3. Используемые библиотеки.
4. Сценарии работы с приложением и примеры использования.

Попробуйте использовать LaTeX. Это даст вам отличную возможность освоить ещё один полезный навык, который скоро вам пригодится для написания курсовых и выпускных работ.

⏵⏵⏵ Фатальные недостатки

Ниже приведён список фатальных недостатков программы, при нахождении которых программа **не может быть принята** до их устранения (как бы хорошо она ни работала):

- наличие предупреждений при компиляции (обнаружить их можно, используя флаги -Wall и -Werror);
- использование транслита и/или русского языка в идентификаторах и/или комментариях;
- отсутствие проверки корректности входных данных (конфигурационные файлы, ввод из stdin, аргументы командной строки и т.д.) - программа должна не аварийно "падать", а выводить человеко-читаемое сообщение об ошибке;
- строчки кода длиной больше 100 символов;
- использование магических констант (исключением могут быть только константы, используемые для отрисовки: размеры объектов, RGB-значения для цвета и т.д.);
- использование функций undefined и unsafePerformIO.

Варианты заданий

Можно выбрать один из предложенных вариантов или придумать интересную для себя задачу!

Две различные команды не могут выбрать один вариант практического задания. Два участника одной команды не могут выбрать одну индивидуальную часть одного задания.

Если несколько студентов собираются индивидуально (не в команде) писать один вариант, то это возможно только при следующих ограничениях:

- один вариант могут писать не более чем n+1 человек, где n — количество индивидуальных частей в варианте;
- студент, который раньше всех занял вариант, пишет только базовую часть;
- остальные студенты должны выполнить также одну из индивидуальных частей, но у всех они должны быть разные;
- постановка индивидуальной задачи должна быть дополнительно согласована.

Возможные варианты заданий:

- ***Интерпретаторы***
  - λ-исчисление

- Диалект Лисп
- Машина Тьюринга
- ***Моделирование***
- Игра "Жизнь"
- Визуализация различных клеточных автоматов
- ***Игры***
- Настольные игры (шашки, шахматы, реверси, нарды и т.п.)
- Классические компьютерные игры (арканоид, тетрис, змейка, сапёр и т.п.)
- Бесконечный платформер
- ***Головоломки***
- Чайнворд
- Японский кроссворд
- Судоку
- Лабиринты
- Wordle
- Kanban-доска

Если вы собираетесь выполнить другое задание по своему выбору, необходимо составить краткое описание базовой и дополнительных частей и зафиксировать эту информацию (письмо преподавателю, файл README в проекте).

λ-исчисление

Описание

λ-исчисление (лямбда-исчисление) лежит в основе большинства функциональных языков программирования: семейства Лиспа (Common Lisp, Scheme, Clojure и др.) и семейства ML (Standard ML, Haskell, Agda и пр.).

λ-исчисление состоит из языка λ-выражений и набора правил преобразования. Базовые правила построения λ-выражений:

- переменная x является λ-выражением;
- если e — λ-выражение, а x — переменная, то λ x . e также λ-выражение (**лямбда-абстракция**);
- если e_1 и e_2 — λ-выражения, то e_1 e_2 также λ-выражение (**аппликация**).

Для удобства работы с λ-выражениями, при записи могут использоваться следующие упрощения:

- внешние скобки могут быть опущены (λ x . e_1) e_2;
- аппликация считается лево-ассоциативной e_1 e_2 e_3 ≡ ((e_1 e_2) e_3);
- тело λ-выражения распространяется вправо насколько возможно λ x . e_1 e_2 ≡ λ x . (e_1 e_2);

- последовательные λ-абстракции схлопываются в одну λ x . λ y . λ z . e ≡ λ x y z . e.

Оператор λ *связывает* переменную в выражении λ x . e. Переменные, подпадающие под какой-либо оператор λ-абстракции, называются *связанными*. Все прочие переменные называются *свободными*. Например, переменная y свободна в выражении λ x . y x. Переменная связывается *ближайшим* оператором λ. Например, единственное вхождение переменной x в выражении λ x . y (λ x . x) связано со вторым оператором λ.

Значение λ-выражения определяется правилами редукции:

- α-**конверсия**: переименовывание связанных переменных
- β-**редукция**: применение функции к аргументам
- η-**конверсия**: выражает принцип *две функции идентичны, если имеют одинаковый результат на всех входах*

Применение функции к аргументам осуществляется за счёт подстановки выражения-аргумента вместо связанной переменной в теле λ-выражения:

x[x -> e] ≡ e
y[x -> e] ≡ y, если y ≠ x
(e_1 e_2) [x -> e] ≡ (e_1 [x -> e]) (e_2 [x -> e])
(λ x . e_1) [x -> e] ≡ λ x . e_1
(λ y . e_1) [x -> e] ≡ λ y . (e_1 [x -> e]), если y ≠ x и y не входит свободно в e

Язык λ-выражений может быть расширен:

- базовыми типами данных (например, числа, булевы значения, строки);
- базовыми контейнерными типами (списки и кортежи);
- встроенными операциями (например, +, -, sin, cos, and, or, ++)
- специальными конструкциями (например, if e1 then e2 else e3 или let x1 = e1 in e2);
- системой типов;
- пользовательские структуры данных;
- и т.д.

Минимальные требования (базовая часть)

Базовая реализация проекта, в которой должны разбираться все участники, должна:

- определять структуру данных для синтаксического дерева;
- содержать раздельно парсер и интерпретатор (с любой стратегией редукций);
- предоставлять простую интерактивную среду программирования (REPL).

Расширенный парсер (индивидуальная часть)

Расширенный парсер должен добавлять хотя бы 2 различные возможности к базовому варианту. Ниже перечислены возможные варианты расширения парсера, однако этим списком они не ограничены:

- разбор расширенного λ-исчисления;
- восстановление после ошибок (например, если пользователь написал запятую (,) вместо точки (.), парсер может запомнить эту ошибку и продолжить разбор программы);
- поддержка пользовательских инфиксных операций с возможностью задать приоритет и ассоциативность;
- и т.д.

Система типов (индивидуальная часть)

Система типов помогает определить семантику λ-выражений, но также может быть использована для оптимизаций при интерпретации и генерации кода.

Система типов должна поддерживать хотя бы 2 различные возможности:

- базовые типы (числа, булев тип, строки);
- контейнерные типы (списки, кортежи, суммы);
- параметрический полиморфизм;
- пользовательские типы данных;
- механизм автоматического вывода типа для терма;
- и т.д.

Расширенная интерактивная среда (индивидуальная часть)

Расширенная интерактивная среда должна добавлять хотя бы 2 различные возможности к базовому интерфейсу. Ниже перечислены возможные варианты расширения интерактивной среды, однако этим списком они не ограничены:

- команды интерпретатора:
  - показать все возможные варианты редукции терма (одного шага) и выбрать один;
  - показать тип выражения;
  - поменять порядок редукции;
  - перевести терм из/в кодировку Чёрча;
  - загрузить программу из файла;
- интерпретация расширенного λ-исчисления;
- дружелюбные сообщения об ошибках (например, для замкнутых термов при опечатке в имени переменной можно предложить имена переменных, отличающихся одной буквой, которые находятся в области видимости);
- и т.д.

Диалект Лисп

Описание

В данном задании предлагается реализовать интерпретатор диалекта языка программирования Лисп.

Базовым объектом Лисп является S-выражение и в предлагаемом диалекте оно может быть представлено

- атомом (IAMATOM, numberp, setf),

- числовым литералом (10, 34.2),
- строковым литералом ("hello, world!"),
- пустым списком (nil, ()),
- точечной парой ((1 . 2)).

Как обычно, непустой список представляется точечной парой, вторым элементом которой является другой список: (1 "asd" atom) эквивалентно (1 . ("asd" . (atom . nil))).

Каждое S-выражение может быть вычислено по следующим правилам:

- числовые, строковые литералы, пустой список и атом вычисляются в себя;
- список (f ...) вычисляется применением функции (или раскрытием макроса) f к остальным элементам списка;
- функция перед применением вычисляет все свои аргументы;
- макросы и специальные формы не вычисляют аргументы перед применением.

Программа на этом диалекте состоит из последовательности S-выражений.

Диалект языка выражений может быть расширен:

- ***встроенными специальными формами:***
  - quote (');
  - if;
  - let;
  - macro;
  - setf;
- ***пользовательскими функциями: lambda, defun;***
- ***лексическим/динамическим связыванием;***
- ***операциями ввода-вывода;***
- ***функциями с переменным числом аргументов;***
- ***и т.д.***

Минимальные требования (базовая часть)

Базовая реализация проекта, в которой должны разбираться все участники, должна:

- определять структуру данных для синтаксического дерева;
- содержать раздельно парсер и интерпретатор;
- предоставлять простую интерактивную среду программирования (REPL);
- предоставлять встроенные предикаты и функции:
  - atomp, numberp, listp, =;
  - car, cdr, cons.

Расширенный парсер (индивидуальная часть)

Расширенный парсер должен добавлять хотя бы 2 различные возможности к базовому варианту. Ниже перечислены возможные варианты расширения парсера, однако этим списком они не ограничены:

- разбор расширенного диалекта;

- восстановление после ошибок (например, если пользователь написал лишнюю закрывающую скобку, парсер может запомнить эту ошибку и продолжить разбор программы);
- и т.д.

Расширенная интерактивная среда (индивидуальная часть)

Расширенная интерактивная среда должна добавлять хотя бы 2 различные возможности к базовому интерфейсу. Ниже перечислены возможные варианты расширения интерактивной среды, однако этим списком они не ограничены:

- команды интерпретатора:
  - загрузить программу из файла;
  - провести один шаг вычисления;
  - раскрыть вызовы макросов;
- интерпретация расширенного диалекта;
- дружелюбные сообщения об ошибках (например, при опечатке в имени переменной можно предложить имена переменных, отличающихся одной буквой, которые находятся в области видимости);
- и т.д.

Генерация кода (индивидуальная часть)

Модуль генерации кода — предпоследний этап компиляции. Генерация кода может быть реализована многими способами, но чтобы простым образом получить портируемый компилятор, можно генерировать промежуточный код на низкоуровневом языке программирования, таком как C или еще ниже, например, LLVM.

Генерация кода должна переводить пользовательские функции в соответствующие функции целевого языка (для этого диалект должен быть расширен возможностью определения пользовательских функций).

Демонстрация генерации кода должна включать в себя программу на любом языке, использующую

Машина Тьюринга

Описание

Машина Тьюринга (МТ) — абстрактная вычислительная машина, предложенная Аланом Тьюрингом для формализации понятия алгоритма.

В состав машины Тьюринга входит неограниченная в обе стороны лента, разделённая на ячейки, и управляющее устройство, способное находиться в одном из конечного множества заранее определённых состояний.

Управляющее устройство может перемещаться влево и вправо по ленте, читать и записывать в ячейки символы некоторого конечного алфавита. Выделяется особый пустой символ, заполняющий все клетки ленты, кроме тех из них (конечного числа), на которых записаны входные данные.

МТ работает согласно правилам перехода. Каждое правило перехода предписывает машине, в зависимости от текущего состояния и наблюдаемого в текущей клетке символа, записать в эту клетку новый символ, перейти в новое состояние и переместиться на одну клетку влево или вправо, либо либо остаться на месте. Некоторые состояния машины Тьюринга могут быть помечены как терминальные, и переход в любое из них означает конец работы, остановку алгоритма. Результатом работы МТ является слово, записанное на ленте после остановки вычислителя.

В данном практическом задании предлагается реализовать систему, визуализирующую процесс вычислений МТ для произвольных программ и входных слов.

Минимальные требования (базовая часть)

Базовая реализация проекта, в которой должны разбираться все участники, должна включать:

- загрузку программы из файла;
- интерфейс для визуализации вычислений на МТ (лента, набор состояний, текущая конфигурация);
- возможность указать входное слово;
- возможность прервать/продолжить/остановить/начать вычисления заново.

Описание программы в виде файла должно также состоять из определения алфавитов и множества состояний, в которых выделены начальное и конечные состояния. Формат описания может быть как придуманным самостоятельно, так и одним из существующих языков разметки (например, JSON, YAML, TOML).

Расширенный интерфейс (индивидуальная часть)

Расширенный интерфейс должен добавлять хотя бы 2 различные возможности к базовому интерфейсу. Ниже перечислены возможные варианты расширения интерфейса, однако этим списком они не ограничены:

- настройки максимального количества шагов, скорости их отображения и т.д.;
- пошаговый режим;
- проверка корректности входных данных;
- и т.д.

Графический интерфейс для построения программ (индивидуальная часть)

Модуль должен включать в себя графический интерфейс для создания программ для МТ. У пользователя должна быть возможность:

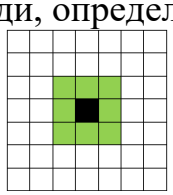
- указать используемый алфавит ленты и входных слов;
- построить таблицу переходов;
- обозначить начальное и конечные состояния;
- запустить программы сразу;
- сохранить программу в формате, воспринимаемом интерпретатором, для дальнейшей загрузки;

Игра «Жизнь»

Описание

Игра «Жизнь» — это клеточный автомат, придуманный английским математиком Джоном Конвеем в 1970 году.

Игра происходит во вселенной — плоскости или поверхности, разбитой на клетки. Каждая клетка в каждый момент времени может находиться в одном из двух состояний: быть «живой» или «мёртвой» (пустой). У каждой клетки есть соседи, определяемые окрестностью Мура.



Время в игре разбито на шаги, за каждый шаг все клетки одновременно обновляют своё состояние по следующим правилам:

- в пустой (мёртвой) клетке зарождается жизнь, если рядом находится ровно 3 живые клетки;
- если у живой клетки ровно 2 или 3 живых соседа, она остаётся жить; иначе она умирает (от одиночества или перенаселённости).

Игрок не принимает прямого участия в игре, а лишь расставляет или генерирует начальную конфигурацию живых клеток, которые затем взаимодействуют согласно правилам уже без его участия (он является наблюдателем).

В игре жизнь присутсвует множество интересных структур. Самым популярным, вероятно, является планер.

Минимальные требования (базовая часть)

Базовая реализация проекта, в которой должны разбираться все участники, должна содержать:

- визуализацию и моделирование заданной сцены;
- загрузку начального состояния сцены из файла.

Расширенный интерфейс (индивидуальная часть)

Расширенный интерфейс должен добавлять хотя бы 2 различные возможности к базовому интерфейсу. Ниже перечислены возможные варианты расширения интерфейса, однако этим списком они не ограничены:

- меню выбора сцены;
- меню выбора модификаций игры «Жизнь»;
- настройки поля-вселенной;
- редактор сцены с панелью заготовленных объектов;
- управление моделированием:
  - пауза/продолжение;
  - ускорение/замедление;
  - перемотка;
- интерфейс сохранения/загрузки;
- и т.д.

Работа с базой данных (индивидуальная часть)

Модуль для работы с базой данных должен предоставлять хотя бы 2 различных возможности:

- база сцен;
- база объектов;
- и т.д.

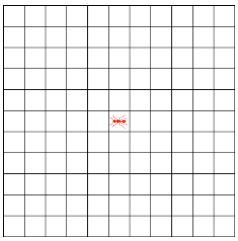
Визуализация клеточных автоматов

Описание

Клеточный автомат — дискретная модель, включающая в себя регулярную решётку ячеек, каждая из которых может находиться в одном из конечного множества состояний, таких как 1 и 0. Решетка может быть любой размерности. Для каждой ячейки определено множество ячеек, называемых окрестностью. Для работы клеточного автомата требуется задание начального состояния всех ячеек и правил перехода ячеек из одного состояния в другое. На каждой итерации, используя правила перехода и состояния ячеек из окрестности, определяется новое состояние каждой ячейки.

В этом варианте задания предлагается реализовать приложение, визуализирующее эволюции различных клеточных автоматов из заранее выбранного набора. Этот набор может включать в себя такие автоматы, как:

- элементарные клеточные автоматы, например, правило 30 или правило 110;
- муравей Лэнгтона;
- черви Патерсона.



Минимальные требования (базовая часть)

Базовая реализация проекта, в которой должны разбираться все участники, должна содержать:

- выбор клеточного автомата из каталога с описанием (не менее 4 различных автоматов);
- визуализацию и моделирование фиксированной сцены для выбранного автомата;

Расширенный интерфейс (индивидуальная часть)

Расширенный интерфейс должен добавлять хотя бы 2 различные возможности к базовому интерфейсу. Ниже перечислены возможные варианты расширения интерфейса, однако этим списком они не ограничены:

- меню выбора сцены;
- интерфейс сохранения/загрузки;
- настройки поля-вселенной (размер поля, цвет клеток и т.п.);
- редактор сцены (указание начального состояния);
- управление моделированием:
  - пауза/продолжение;
  - ускорение/замедление;
  - перемотка;
- и т.д.

Работа с базой данных (индивидуальная часть)

Модуль для работы с базой данных должен предоставлять хотя бы 2 различных возможности:

- база сцен;
- база объектов;
- и т.д.

Настольные игры

Описание

В данном задании предлагается реализовать любую из настольных игр (например, реверси, шашки, шахматы, нарды, го и пр.).

Минимальные требования (базовая часть)

Базовая реализация проекта, в которой должны разбираться все участники, должна:

- предоставлять возможность играть в игру в режиме «человек против человека», используя графический интерфейс;
- не допускать невозможных по правилам игры ходов игроков;
- определять момент победы или ничьей и демонстрировать пользователям результат.

Расширенный интерфейс (индивидуальная часть)

Расширенный интерфейс должен добавлять хотя бы 2 различные возможности к базовому интерфейсу. Ниже перечислены возможные варианты расширения интерфейса, однако этим списком они не ограничены:

- показ возможных ходов для выбранной фигуры/шашки;
- отмена хода/ходов;
- меню выбора режима игры:
  - человек против человека;
  - человек против компьютера;
  - компьютер против компьютера;
  - сетевая игра;
- меню настроек ИИ;
- меню выбора разновидности настольной игры (например, шашки с дамками или без);
- просмотр таблицы рекордов;
- проигрывание записанных игр;
- интерфейс сохранения/загрузки игр;
- и т.д.

Искусственный интеллект (индивидуальная часть)

Алгоритм, делающий ход, никак не оценивая ситуацию (например, случайный или первый доступный ход), не считается за реализацию искусственного интеллекта.

Реализация искусственного интеллекта должна предоставлять:

- настройки сложности;
- как минимум 2 различных стратегии (это может быть один алгоритм с разными эвристиками).

Клиент-серверная архитектура (индивидуальная часть)

Помимо возможности просто играть в настольную игру по сети (см. Минимальные требования), клиент-серверная архитектура должна предоставлять хотя бы 2 дополнительные возможности:

- поддержка нескольких игровых сессий одновременно;
- запуск ИИ на серверной стороне в качестве противника;
- регистрация, аутентификация и авторизация (вход в систему и права на доступ);
- доступ к таблице рекордов;
- сохранение/загрузка игр;
- и т.д.

Работа с базой данных (индивидуальная часть)

Модуль для работы с базой данных должен предоставлять хотя бы 2 различных возможности:

- таблица рекордов;
- сохранения игровых сессий;
- база пользователей;
- и т.д.

Классические компьютерные игры

Описание

В этом варианте предлагается реализовать одну из "классических" компьютерных игр (например, змейка, тетрис, арканоид и т.п.).

Минимальные требования (базовая часть)

Базовая реализация проекта должна:

- предоставлять возможность играть в игру, используя графический интерфейс или консольные виджеты;
- не допускать невозможных по правилам игры действий игроков;
- определять момент завершения игры и демонстрировать пользователям результат.

Расширенный интерфейс (индивидуальная часть)

Расширенный интерфейс должен добавлять хотя бы 2 различные возможности к базовому интерфейсу. Эти возможности во по большей части зависят от выбора конкретной игры. Ниже перечислены возможные варианты расширения интерфейса, однако этим списком они не ограничены:

- меню:
  - просмотр таблицы рекордов;
  - интерфейс сохранения/загрузки игр;
- расширения игры:
  - бонусные баллы;
  - выбор режима игры;
  - ограничение прохождения уровня по времени;
- и т.д.

Конструктор уровней (индивидуальная часть)

Конструктор уровней даёт игрокам возможность создавать свои собственные уровни, если это применимо к выбранной игре.

В модуле должно быть реализовано следующее:

- графический интерфейс для создания уровней;
- возможность сохранять уровень с названием и именем автора;
- возможность загрузить уровень для редактирования в конструкторе;
- пункт меню для выбора пользовательских уровней для игры.

Работа с базой данных (индивидуальная часть)

Модуль для работы с базой данных должен предоставлять хотя бы 2 различных возможности:

- таблица рекордов;

- сохранения игровых сессий;
- база пользователей;
- и т.д.

Бесконечный платформер

Описание

Платформер — это жанр компьютерных игр, в которых основной чертой игрового процесса является прыгание по платформам, лазанье по лестницам, собирание предметов, обычно необходимых для завершения уровня.

Бесконечны платформер — это платформер, в котором персонаж бежит в одном направлении по миру, который генерируется по ходу движения. Таким образом, игровой мир потенциально бесконечен.

Основная цель игры — добежать как можно дальше.

В данном задании предлагается любую разновидность бесконечного платформера.

Минимальные требования (базовая часть)

Базовая реализация проекта, в которой должны разбираться все участники, должна:

- предоставлять простейшую генерацию игрового мира — повторение заданного шаблона;
- создавать игровой мир с возможность проиграть или двигаться дальше;
- предоставлять минимальное управление персонажем;
- определять момент поражения и подсчитывать пройденную дистанцию.

Расширенный интерфейс (индивидуальная часть)

Расширенный интерфейс должен добавлять хотя бы 2 различные возможности к базовому интерфейсу. Ниже перечислены возможные варианты расширения интерфейса, однако этим списком они не ограничены:

- меню:
  - просмотр таблицы рекордов;
  - интерфейс сохранения/загрузки игр;
  - выбор режимов игры:
    - однопользовательская/многопользовательская;
    - уровни сложности;
- расширения игры:
  - многопользовательский режим;

- бонусы и специальные объекты, которые может подбирать персонаж;
- различные препятствия;
- отмотка времени;
- и т.д.

Генератор игрового мира (индивидуальная часть)

Генератор игрового мира должен постепенно повышать сложность трассы.

Для реализации различных режимов игры, генератор должен быть конфигурируемым. В частности, пользователь генератора должен иметь возможность

- ограничить объекты игрового мира, используемые для генерации;
- определить относительную сложность трассы, на которой может появляться заданный объект;
- определить частоту встречаемости объекта на трассе (в зависимости от сложности);
- и т.д.

Искусственный интеллект (индивидуальная часть)

Алгоритм поведения, никак не оценивающий ситуацию (например, случайное движение), не считается за реализацию искусственного интеллекта.

Искусственный интеллект в бесконечном платформере может управлять - активными противниками персонажа, встречающиеся на пути; - соперничающим персонажем (в многопользовательском режиме вместо второго игрока).

Реализация искусственного интеллекта должна предоставлять настройки сложности.

Клиент-серверная архитектура (индивидуальная часть)

Помимо возможности просто играть в игру по сети (см. Минимальные требования), клиент-серверная архитектура должна предоставлять хотя бы 2 дополнительные возможности:

- поддержка нескольких игровых сессий одновременно;
- запуск ИИ на серверной стороне;
- регистрация, аутентификация и авторизация (вход в систему и права на доступ);
- доступ к таблице рекордов;
- сохранение/загрузка игр;
- и т.д.

Работа с базой данных (индивидуальная часть)

Модуль для работы с базой данных должен предоставлять хотя бы 2 различных возможности:

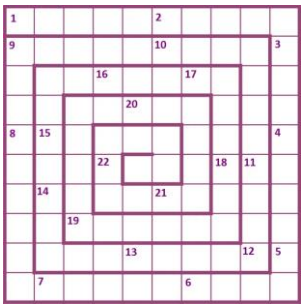
- таблица рекордов;
- сохранения игровых сессий;
- база пользователей;

- и т.д.

Чайнворд

Описание

Чайнворд — это разновидность кроссворда, где слова составляют единую цепочку. Каждое следующее слово начинается с последней буквы предыдущего слова.



Минимальные требования (базовая часть)

Базовая реализация проекта, в которой должны разбираться все участники, должна включать:

- интерфейс для разгадывания чайнворда (чайнворд + описания слов);
- загрузку головоломки из файла.

Расширенный интерфейс (индивидуальная часть)

Расширенный интерфейс должен добавлять хотя бы 2 различные возможности к базовому интерфейсу. Ниже перечислены возможные варианты расширения интерфейса, однако этим списком они не ограничены:

- показ верно угаданных слов;
- меню генерации чайнворда:
  - словарь;
  - сложность (например, количество слов);
  - вариант расположения (например, спираль, змейкой, лабиринтом, и т.д.)
- подсказки (показать одну букву);
- таймер решения головоломки;
- просмотр таблицы рекордов;
- интерфейс сохранения/загрузки игр;
- альтернативные способы отображения чайнворда (многоугольники, круги, спирали и пр.);
- редактор головоломок;
- и т.д.

Генерация головоломок (индивидуальная часть)

Генератор головоломок должен:

- использовать для генерации как минимум два параметра из трёх:
  - словари;
  - схемы расположения цепочки слов;
  - уровни сложности.

- по возможности, предоставлять равномерное распределение вероятности попадания слова из словаря в головоломку.

Выбор из нескольких заранее подготовленных головоломок не считается генерацией.

Работа с базой данных (индивидуальная часть)

Модуль для работы с базой данных должен предоставлять хотя бы 2 различных возможности:

- таблица рекордов;
- сохранения игровых сессий;
- база головоломок;
- и т.д.

Японский кроссворд

Описание

Изображения зашифрованы числами, расположенными слева от строк, а также сверху над столбцами. Числа показывают, сколько групп чёрных (либо своего цвета, для цветных кроссвордов) клеток находятся в соответствующих строке или столбце и сколько слитных клеток содержит каждая из этих групп (например, набор чисел 4, 1, и 3 означает, что в этом ряду есть три группы: первая — из четырёх, вторая — из одной, третья — из трёх чёрных клеток). В чёрно-белом кроссворде группы должны быть разделены, как минимум, одной пустой клеткой, в цветном это правило касается только одноцветных групп, а разноцветные группы могут быть расположены вплотную (пустые клетки могут быть и по краям рядов). Необходимо определить размещение групп клеток.

Минимальные требования (базовая часть)

Базовая реализация проекта, в которой должны разбираться все участники, должна включать:

- интерфейс для разгадывания японского кроссворда;
- загрузку головоломки из файла.

Расширенный интерфейс (индивидуальная часть)

Расширенный интерфейс должен добавлять хотя бы 2 различные возможности к базовому интерфейсу. Ниже перечислены возможные варианты расширения интерфейса, однако этим списком они не ограничены:

- меню выбора головоломок;
- запуск автоматического решателя;
- таймер решения головоломки;
- просмотр таблицы рекордов и архива решенных головоломок;
- интерфейс сохранения/загрузки частичного решения;

- редактор головоломок;
- и т.д.

Генерация и автоматическое решение головоломок (индивидуальная часть)

В этой индивидуальной части требуется реализовать генерацию головоломок и автоматический решатель.

Генерация головоломок должна происходить по картинке, заданной в файле (например, псевдографикой в текстовом файле или в любом медиа-формате). После генерации головоломки, её сложность должна быть оценена при помощи автоматического решателя.

Автоматическое решение головоломки может быть реализовано несколькими способами. Чтобы использовать тот же алгоритм для оценки сложности головоломок, необходимо использовать стратегии, моделирующие решение человеком. При решении на каждом шаге пробуются стратегии, начиная с самой простой, и первая подходящая используется. Средняя сложность использованных стратегий может расцениваться как общая сложность головоломки.

Выбор из нескольких заранее подготовленных головоломок не считается генерацией.

Работа с базой данных (индивидуальная часть)

Модуль для работы с базой данных должен предоставлять хотя бы 2 различных возможности:

- таблица рекордов;
- сохранения игровых сессий;
- база головоломок;
- и т.д.

Судоку

Описание

Игровое поле представляет собой квадрат размером 9×9, разделённый на меньшие квадраты со стороной в 3 клетки. Таким образом, всё игровое поле состоит из 81 клетки. В них уже в начале игры стоят некоторые числа (от 1 до 9), называемые подсказками. От игрока требуется заполнить свободные клетки цифрами от 1 до 9 так, чтобы в каждой строке, в каждом столбце и в каждом малом квадрате 3×3 каждая цифра встречалась бы только один раз.

Сложность судоку зависит не от количества изначально заполненных клеток, а от методов, которые нужно применять для её решения. Самые простые решаются дедуктивно: всегда есть хотя бы одна клетка, куда подходит только одно число. Некоторые головоломки можно решить за несколько минут, на другие можно потратить часы.

Правильно составленная головоломка имеет только одно решение.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Минимальные требования (базовая часть)

Базовая реализация проекта, в которой должны разбираться все участники, должна включать:

- интерфейс для разгадывания судоку;
- загрузку головоломки из файла.

Расширенный интерфейс (индивидуальная часть)

Расширенный интерфейс должен добавлять хотя бы 2 различные возможности к базовому интерфейсу. Ниже перечислены возможные варианты расширения интерфейса, однако этим списком они не ограничены:

- настройки сложности судоку;
- подсказки (например, показать, какие цифры можно поставить в данную клетку);
- запуск автоматического решателя;
- таймер решения головоломки;
- просмотр таблицы рекордов;
- интерфейс сохранения/загрузки игр;
- выбор разновидности головоломки;
- редактор головоломок;
- и т.д.

Генерация и автоматическое решение головоломок (индивидуальная часть)

В этой индивидуальная части требуется реализовать генерацию головоломок с заданной сложностью и автоматический решатель.

Простым способом генерации головоломки «Судоку» является случайная перестановка строк и столбцов и вычёркивание случайных клеток. Для генерации головоломок заданной сложности можно воспользоваться одним из двух вариантов:

- вычёркивать клетки по одной, каждый раз оценивая насколько игра усложняется (насколько трудно человеку будет проставить вычеркнутую цифру);
- генерировать головоломку, используя автоматический решатель.

Автоматическое решение головоломки может быть реализовано несколькими способами. Чтобы использовать тот же алгоритм для генерации головоломок заданной сложности, необходимо использовать стратегии, моделирующие решение человеком:

- при решении на каждом шаге пробуются стратегии, начиная с самой простой, и первая подходящая используется.

- при генерации на каждом шаге добавляется очередная цифра в поле, на основе одной из стратегий заданной сложности.

Выбор из нескольких заранее подготовленных головоломок не считается генерацией.

Работа с базой данных (индивидуальная часть)

Модуль для работы с базой данных должен предоставлять хотя бы 2 различных возможности:

- таблица рекордов;
- сохранения игровых сессий;
- база головоломок;
- и т.д.

Лабиринты

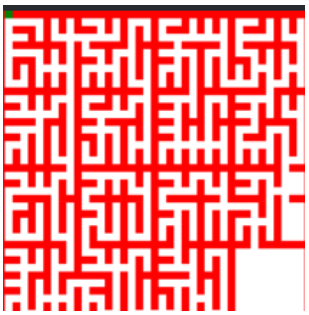
Описание

В данном варианте предлагается реализовать систему генерации и прохождения двумерных лабиринтов.

Для генерации лабиринтов существует множество алгоритмов. Одним из самых простых для реализации на функциональном языке является алгоритм рекурсивного деления.

Работа алгоритма состоит в следующем:

1. В начале работы алгоритма в комнате нет стен.
2. Пространство комнаты разделяется стеной (одной или двумя перпендикулярными) в произвольном отношении.
3. В произвольном месте каждой стены выбирается проход.
4. Процесс рекурсивно повторяется для полученных комнат меньшего размера, пока размер комнаты не достигнет заранее установленного минимального размера.



Минимальные требования (базовая часть)

Базовая реализация проекта, в которой должны разбираться все участники, должна включать:

- генерацию лабиринта и визуализацию этого процесса;

- интерфейс для прохождения сгенерированного лабиринта (управление игроком);
- сохранение и загрузка лабиринтов из файла.

Выбор из нескольких заранее подготовленных лабиринтов не считается генерацией.

Расширенный интерфейс (индивидуальная часть)

Расширенный интерфейс должен добавлять хотя бы 2 различные возможности к базовому интерфейсу. Ниже перечислены возможные варианты расширения интерфейса, однако этим списком они не ограничены:

- настройки размеров лабиринта;
- выбор алгоритма генерации;
- таймер прохождения лабиринта;
- запуск автоматического решателя;
- подсказки (например, показать следующие несколько шагов);
- и т.д.

Автоматическое прохождение лабиринта (индивидуальная часть)

В этой индивидуальной части требуется реализовать автоматический решатель головоломки, который проходит лабиринт от текущего положения игрока до выхода из лабиринта и рисует полученный путь.

Можно использовать любой алгоритм нахождения пути в лабиринте или несколько из них, используя разные цвета для визуализации полученных путей.

Работа с базой данных (индивидуальная часть)

Модуль для работы с базой данных должен предоставлять хотя бы 2 различных возможности:

- таблица рекордов;
- сохранения игровых сессий;
- база лабиринтов;
- и т.д.

Wordle

Описание

Wordle — это игра, где нужно угадать слово из пяти букв за шесть попыток.

Для начала нужно вписать любое слово в верхнюю строчку игрового поля. Если хотя бы одна буква в вашем слове соответствует загаданному, то

она подсветится зелёным, когда стоит в нужном месте, и жёлтым, когда стоит не там, где надо. Если все ячейки вашего слова остаются серыми, значит в искомом слове вообще нет этих букв. Аналогичным образом подсвечиваются буквы на экранной клавиатуре.

Версия на английском и на русском.

В данном варианте предлагается реализовать эту игры. В оригинальной версии новое слово, которое нужно угадать, обновляется раз в сутки, для практического задания этого ограничения нет, слово должно произвольно выбираться из словаря при старте игры.

Набор возможных слов задаётся заранее, пользователь должен иметь возможность указать путь к файлу со словарём (или настройки подключения к базе данных со словарём при выборе соответствующей индивидуальной части).

Минимальные требования (базовая часть)

Базовая реализация проекта, в которой должны разбираться все участники, должна включать:

- интерфейс для отгадывания слова;
- загрузку словаря из файла.

Расширенный интерфейс (индивидуальная часть)

Расширенный интерфейс должен добавлять хотя бы 2 различные возможности к базовому интерфейсу. Ниже перечислены возможные варианты расширения интерфейса, однако этим списком они не ограничены:

- меню:
  - просмотр таблицы рекордов;
  - интерфейс сохранения/загрузки игр;
  - интерфейс для обновления загаданного слова или обнуления действий игрока;
- расширения игры:

- подсчёт времени;
- возможность выбрать размер поля;
- возможность выбрать язык игры (должно поддерживаться как минимум 2 языка);
- интерфейс для работы со словарём;
- и т.д.

Работа с базой данных (индивидуальная часть)

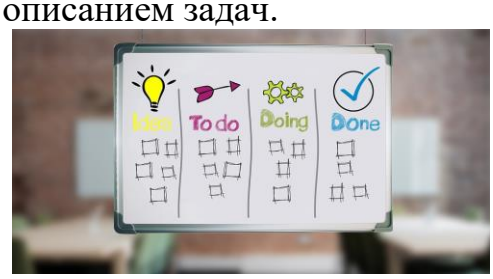
Модуль для работы с базой данных должен предоставлять хотя бы 2 различных возможности:

- таблица рекордов;
- сохранения игровых сессий;
- база пользователей;
- база слов;
- и т.д.

Kanban-доска

Описание

Kanban-доска - инструмент для управления проектами и визуализации рабочего процесса. Самые простые доски состоят из трех колонок: «сделать» (To Do), «в процессе» (In Progress) и «сделано» (Done). Каждая колонка заполняется карточками с описанием задач.



Новые задачи после создания помещаются в самую левую колонку. По мере обновления статуса задачи пользователи перемещают карточки из одной колонки в другую.

Карточка состоит из идентификатора (обычно числового) задачи, её заголовка и описания. У задачи может быть назначен исполнитель из заранее определённого списка пользователей.

Минимальные требования (базовая часть)

Базовая реализация проекта, в которой должны разбираться все участники, должна:

- визуализировать Kanban-доску как минимум с тремя фиксированными колонками;
- предоставлять интерфейс для создания новых задач, а также изменения и удаления существующих;
- предоставлять возможность переносить задачи между колонками, указывать приоритет и ответственного за задачу из заранее определённого списка пользователей.

Расширенный интерфейс (индивидуальная часть)

Расширенный интерфейс должен добавлять хотя бы 2 различные возможности к базовому интерфейсу. Ниже перечислены возможные варианты расширения интерфейса, однако этим списком они не ограничены:

- возможность конфигурировать колонки (добавлять новые, переименовывать и удалять имеющиеся);
- интерфейс для работы со списком пользователей;
- возможность выставлять дополнительные атрибуты карточек, например:
  - дату дедлайна;
  - тип задачи из определённого множества (баг/фича/вопрос/тестирование и т.п.);
  - произвольные текстовые метки/теги (обычно указывают часть системы, к которой относится задача: backend, frontend, tests, integration, ...);
  - и т.д.

Работа с базой данных (индивидуальная часть)

Модуль для работы с базой данных должен предоставлять хотя бы 2 различных возможности:

- сохранения и загрузка списка задач;
- сохранения и загрузка конфигурации колонок;
- база пользователей;
- и т.д.

Порядок выполнения лабораторных работ

- 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.

Перечень теоретических вопросов к экзамену

1. Декларативное и императивное программирование. Основные понятия.
2. Понятие функции. Синтаксис и аргументы..
3. λ-исчисление. Базовые понятия. Синтаксис и вычисление термов.
4. Каррирование и редукция. Стратегии редукции.
5. Символьное представление данных в языке программирования
6. Элементарные функции Работа со списками массивами.
7. Логические выражения и функции. Построение условных переходов в программных вычислениях.
8. Построение рекурсивных функций и процедур.
9. Функционалы и функции планирования.
10. Управляющие конструкции языка программирования.
11. Реализация методов поиска на языке программирования.
12. Процедурное и логическое программирование. Понятие логической программы.
13. Компиляторы и интерпретаторы. Функционал и особенности использования.
14. Отладка и тестирование программ.
15. Организация вывода результатов.

ПРИЛОЖЕНИЕ

А. Языки функционального программирования

В приложении приведено краткое описание некоторых языков функционального программирования (очень немногих). Дополнительную информацию можно почерпнуть, просмотрев интернет-ресурсы, перечисленные ниже.

- **Lisp** (List processor). Считается первым функциональным языком программирования. Нетипизирован. Содержит массу императивных свойств, однако в общем поощряет именно функциональный стиль программирования. При вычислениях использует *вызов-по-значению*. Сейчас наиболее распространён диалект Common Lisp, ушедший от принципов ФП. Язык сложен в изучении, имеет очень непривычный синтаксис. Существует объектно-ориентированный диалект языка — CLOS.
- **Scheme**. Один из многих диалектов языка Lisp, предназначенный для научных исследований в области информатики. При разработке Scheme был сделан упор на элегантность и простоту языка. Благодаря этому язык получился намного меньше, чем базовая версия языка Lisp. Отличается простотой как самого языка, так и стандартной библиотеки функций, хотя несколько проигрывает в универсальности. Кроме того, здесь точнее соблюдаются принципы функционального программирования.
- **ISWIM** (If you See What I Mean). Функциональный язык-прототип. Разработан Ландиным (США) в 60-х годах для демонстрации того, каким может и должен быть язык функционального программирования. Вместе с языком Ландин разработал и специальную виртуальную машину для исполнения программ на языке ISWIM. Эта виртуальная машина, основанная на *вызове-по-значению*, получила название SECD-машины. На синтаксисе языка ISWIM базируется синтаксис многих функциональных языков. На синтаксис ISWIM похож синтаксис ML, особенно Caml.
- **ML** (Meta Language). Целое семейство строгих функциональных языков с развитой полиморфной системой типов и параметризуемыми модулями. ML преподаётся во многих университетах мира (в некоторых даже как первый язык программирования).
- **Standard ML**. Один из первых типизированных языков функционального программирования. Содержит некоторые императивные свойства, такие как ссылки на изменяемые значения, и поэтому не является чистым. При вычислениях использует *вызов-*

по-значению. Очень интересная реализация модульности. Мощная полиморфная система типов. Последний стандарт языка — Standard ML-97, существует формальное математическое определение синтаксиса, а также статической и динамической семантик языка.

- **Caml Light** и **Objective Caml**. Как и Standard ML принадлежит к семейству ML. Objective Caml отличается от Caml Light, в основном, поддержкой классического объектно-ориентированного программирования. Objective Caml, также как и Standard ML, является строгим, но имеет некоторую встроенную поддержку отложенных вычислений.
- **Miranda**. Функциональный язык, разработанный Дэвидом Тернером (США), в качестве стандартного функционального языка, использовавшего отложенные вычисления. Имеет строгую полиморфную систему типов. Как и ML этот язык преподаётся во многих университетах. Оказал большое влияние на разработчиков языка Haskell.
- **Haskell**. Один из самых распространенных нестрогих языков функционального программирования. Имеет очень развитую систему типизации. Несколько хуже разработана система модулей. Последний стандарт языка — Haskell-98.
- **Gofer** (GOod For Equational Reasoning). Упрощенный диалект языка Haskell. Предназначен для обучения основам функционального программирования.
- **Clean**. Функциональный язык, специально предназначенный для параллельного и распределенного программирования. По синтаксису напоминает Haskell. Чистый. Использует отложенные вычисления. С компилятором поставляется набор библиотек (I/O libraries), позволяющих программировать графический пользовательский интерфейс под Win32 или MacOS.
- **Erlang**. Язык, разработанный компанией Ericsson, ориентирован на сферу коммуникаций; происходит от Пролога, хотя сейчас похож на него лишь внешне. Имеет лёгкий в освоении синтаксис, богатейшую библиотеку, в том числе: переносимый графический интерфейс, СУБД, распределенные вычисления и многое другое; причем всё это доступно бесплатно. Язык является параллельным изначально — возможность параллельной работы нескольких процессов и их взаимодействия заложена на уровне синтаксиса. Недостаток пока только один: компиляция в байт-код, для которого нужен "тяжёлый" и не слишком быстрый интерпретатор.
- **Рефал**. Функциональный язык, разработанный в СССР ещё в семидесятых годах XX века. В некоторых отношениях близок к Прологу. Крайне эффективен для обработки сложных структур данных типа текстов на естественном языке, XML и т.д. Эта эффективность обусловлена тем, что Рефал является единственным языком, использующим двунаправленные списки — это позволяет сократить объём некоторых программ и ускорить их работу (за счёт отсутствия необходимости в сборке мусора). К сожалению, в последнее время разрабатывается не очень активно. С другой стороны, по языку много документации на русском языке; имеется суперкомпилятор — система оптимизации программ на уровне исходных текстов (т.е. фактически оптимизация алгоритма!).
- **Joy**. Чистый функциональный язык, основанный на комбинаторной логике. Внешне похож на Форт: используется обратная польская запись и стек. Пока что находится в стадии развития, хотя уже имеет неплохую теоретическую основу. Существует даже прототипная реализация. Основное преимущество перед другими языками — линейная запись без переменных, что должно резко облегчить автоматизацию написания, проверки и оптимизации программ. Помимо недоразвитости, есть еще один серьезный недостаток: как и у Форта, у Joy страдает читабельность.

Б. Интернет-ресурсы по функциональному программированию

В приложении представлена небольшая (и далеко неполная) подборка ресурсов в Интернете, посвящённых функциональному программированию вообще и различным функциональным языкам в частности.

- **www.haskell.org** — очень насыщенный сайт, посвященный функциональному программированию в общем и языку Haskell в частности. Содержит различные справочные материалы, список интерпретаторов и компиляторов языка Haskell (в настоящий момент все интерпретаторы и компиляторы бесплатны). Кроме того, имеется обширный список интересных ссылок на ресурсы по теории функционального программирования и другим языкам (Standard ML, Clean).
- **cm.bell-labs.com/cm/cs/what/smlnj** — Standard ML of New Jersey. Очень хороший компилятор. В бесплатный дистрибутив помимо компилятора входят утилиты MLYacc и MLLex и библиотека Standard ML Basis Library. Отдельно можно взять документацию по компилятору и библиотеке.
- **www.harlequin.com/products/ads/ml/** — Harlequin MLWorks, коммерческий компилятор Standard ML. Однако в некоммерческих целях можно бесплатно пользоваться версией с несколько ограниченными возможностями.
- **caml.inria.fr** — институт INRIA. «Домашний» сайт команды разработчиков языков Caml Light и Objective Caml. Можно бесплатно скачать дистрибутив Objective Caml, содержащий интерпретатор, компиляторы байт-кода и машинного кода, Yacc и Lex для Caml, отладчик и профайлер, документацию, примеры. Качество компилированного кода у этого компилятора очень хорошее, по скорости опережает даже Standard ML of New Jersey.
- **www.cs.kun.nl/~clean/** — содержит дистрибутив компилятора с языка Clean. Компилятор коммерческий, но допускается бесплатное использование в некоммерческих целях. Из того, что компилятор коммерческий, следует его качество (очень быстр), наличие среды разработчика, хорошей документации и стандартной библиотеки.